Festschrift

LNCS 2566

Torben Æ. Mogensen
David A. Schmidt
I. Hal Sudborough (Eds.)

# The Essence of Computation
## Complexity, Analysis, Transformation

Essays Dedicated to Neil D. Jones

Springer

*Berlin*
*Heidelberg*
*New York*
*Barcelona*
*Hong Kong*
*London*
*Milan*
*Paris*
*Tokyo*

Torben Æ. Mogensen   David A. Schmidt
I. Hal Sudborough (Eds.)

# Complexity, Analysis, Transformation

Essays Dedicated to Neil D. Jones

Series Editors

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

Volume Editors

Torben Æ. Mogensen
University of Copenhagen, DIKU
Universitetsparken 1, 2100 Copenhagen, Denmark
E-mail: torbenm@diku.dk

David A. Schmidt
Kansas State University, Department of Computing and Information Sciences
234 Nichols Hall, Manhattan, KS 66506 USA
E-mail: schmidt@cis.ksu.edu

I. Hal Sudborough
University of Texas at Dallas, Department of Computer Science
P.O. Box 830688, Richardson, TX 75083 USA
E-mail: hal@utdallas.edu

The illustration appearing on the cover of this book is the work of Daniel Rozenberg (DADARA).

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek.
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-00326-6 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN 10871869 06/3142 5 4 3 2 1 0

# Preface

During the second half of the twentieth century, computing and computation theory were established as part of a common academic discipline. One distinctive aspect of computation theory is the importance of its notations: Programming languages like Algol 60, Scheme, and Smalltalk are significant and studied in their own right, because they provide the means for expressing the very *essence of computation*. This "essence" is formalized within the subareas of computational complexity, program design and analysis, and program transformation.

This volume presents state-of-the-art aspects of the essence of computation — computational complexity, program analysis, and program transformation — by means of research essays and surveys written by recognized experts in the three areas.

## An Appreciation of Neil D. Jones

Aside from the technical orientation of their writings, a thread that connects all the authors of this volume is that all of them have worked with *Prof. Neil D. Jones*, in the roles of student, colleague, and, in one case, mentor. The volume's authors wish to dedicate their articles to Neil on the occasion of his 60th birthday.

The evolution of Neil Jones's career parallels the evolution of computation theory itself; indeed, inspiration and influence from Neil is evident throughout the area he studied and helped shape. The following survey (and reference list) of Neil's work is necessarily selective and incomplete but will nonetheless give the reader a valid impression of Neil's influence on his field.

Born on March 22, 1941, in Centralia, Illinois, USA, Neil studied first at Southern Illinois University (1959–1962) and then at the University of Western Ontario, Canada (1962–1967), where he worked first under the supervision of Satoru Takasu (who later led the Research Institute of Mathematical Sciences at Kyoto) and then under Arto Salomaa (now at the University of Turku). During this period, Neil led the development of one of the first, if not the very first, Algol 60 compilers in North America [5, 34]. The problems and issues raised by implementing perhaps the quintessential programming language would stimulate Neil for decades to come.

During his tenure as a faculty member at Pennsylvania State University (1967–1973), Neil established himself as an expert in formal language theory and computability, distinguishing himself with his breakthrough solution to the "spectrum" problem, jointly with Alan Selman [6, 7, 36]. Appreciating the significance of the fast-growing academic discipline of computing science, Neil authored the first computation-theory text specifically oriented towards computing scientists, *Computability Theory: An Introduction* [8], which was one of the first volumes published in the newly established ACM Monograph series.

Neil's scholarly work at Pennsylvania State laid the foundation for his move to the University of Kansas in 1973, where Neil provided a crucial push to the blossoming area of computational complexity: His seminal research papers [9, 23, 24] introduced and developed the complexity classes of polynomial time and log space, and they defined and applied the now classic technique of log-space reduction for relating computational problems.

Neil's continuing interest in programming languages influenced his research in computational complexity, which acquired the innovative aspect of exploiting the notations used to code algorithms in the proofs of the complexity properties of the algorithms themselves. Key instances of these efforts were published in collaboration with Steven Muchnick at the University of Kansas [26, 27].

Around 1978, programming-language analysis moved to the forefront of Neil's research. Neil Jones was perhaps the first computing scientist to realize the crucial role that *binding times* play in the definition and implementation of a programming language: Early binding times support detailed compiler analysis and fast target code, whereas late binding times promote dynamic behavior of control and data structure and end-user freedom. His text with Steven Muchnick, *Tempo: A Unified Treatment of Binding Time and Parameter Passing Concepts* [28], remains to this day the standard reference on binding-time concepts.

Implicit in the *Tempo* text was the notion that a program could be mechanically analyzed to extract the same forms of information that are communicated by early binding times. This intuition led Neil to develop several innovative dataflow analysis techniques [11, 25, 29–32], coedit a landmark collection of seminal papers on flow analysis and program analysis, *Program Flow Analysis: Theory and Applications* [43], and coauthor an influential survey paper [33].

A deep understanding of compiler implementation, coupled with many insights into binding times and flow analysis, stimulated Neil to explore aspects of compiler generation from formal descriptions of programming language syntax and semantics. While visiting Aarhus University, Denmark, in 1976-77, Neil encountered the just developed denotational-semantics format, which served as a testbed for his subsequent investigations into compiler generation. His own efforts, e.g., [35, 39, 42], plus those of his colleagues, as documented in the proceedings of a crucial workshop organized by Neil in 1980 [10], display a literal explosion of innovative techniques that continue to influence compiler generation research.

In 1981, Neil accepted a faculty position at the University of Copenhagen, where he was promoted to Professor. His research at Copenhagen was a culmination of the research lines followed earlier in his career: Building on his knowledge in complexity and program analysis, Neil developed a theory of program transformation, which now constitutes a cornerstone of the field of *partial evaluation*.

The story goes somewhat like this: Dines Bjørner introduced Neil to Andrei Ershov's theory of "mixed computation" — a theory of compilation and compiler generation stated in terms of interpreter self-interpretation (self-application). Intrigued by the theory's ramifications, Neil and his research team in Copenhagen worked on the problem and within a year developed the first program — a *partial evaluator* — that generated from itself a compiler generator by means of self-application [37]. The crucial concept needed to bring Ershov's theory to life was a flow analysis for calculating binding times, a concept that only Neil was perfectly posed to understand and apply!

The programming of the first self-applicable partial evaluator "broke the dam," and the flood of results that followed showed the practical consequences of partial-evaluation-based program transformation [2–4, 12–16, 20, 22, 38, 44]. Along with Bjørner and Ershov, Neil co-organized a partial-evaluation workshop in Gammel Avernæs, Denmark, in October 1987. The meeting brought together the leading minds in program transformation and partial evaluation and was perhaps the first truly *international* computing science meeting, due to the presence of a significant contingent of seven top researchers from Brezhnev's USSR [1].

Neil coauthored the standard reference on partial evaluation in 1993 [21].

Coming full circle at the end of the century, Neil returned to computational complexity theory, blending it with his programming languages research by defining a family of Lisp-like mini-languages that characterize the basic complexity classes [18, 19]. His breakthroughs are summarized in his formidable text, *Computability and Complexity from a Programming Perspective* [17], which displays a distinctive blend of insights from complexity, program analysis, and transformation, which could have come only from a person who has devoted a career to deeply understanding all three areas.

If his recent work is any indication [40, 41], Neil's investigations into the *essence of computation* are far from finished!

15 October 2002

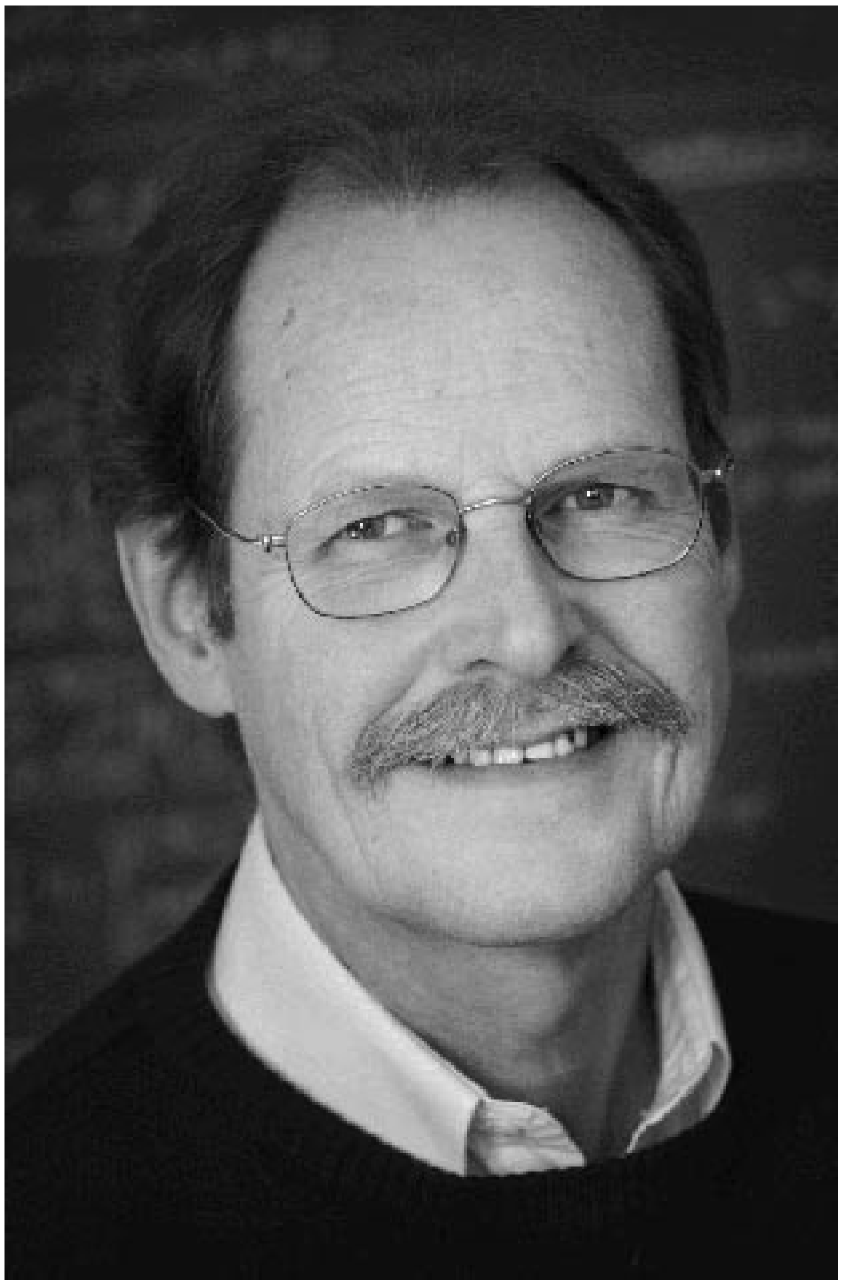
Torben Mogensen
David Schmidt
I. Hal Sudborough

## References

1. Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. Elsevier Science/North-Holland, 1988.
2. Harald Ganzinger and Neil D. Jones, editors. *Programs as Data Objects*. Lecture Notes in Computer Science 217. Springer-Verlag, 1986.
3. Arne John Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, Lecture Notes in Computer Science 1181. Springer-Verlag, 1996.
4. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming* 1(1):21-69, January 1991.
5. Neil D. Jones. 1620 Algol compiler: subroutine package. Technical Report. Southern Illinois University, Computing Center, 1962.
6. Neil D. Jones. Classes of automata and transitive closure. *Information and Control* 13:207-209, 1968.
7. Neil D. Jones. Context-free languages and rudimentary attributes. *Mathematical Systems Theory* 3(2):102-109, 1969. Corrigendum, 11:379-380.

8. Neil D. Jones. *Computability Theory: An Introduction.* ACM Monograph Series. Academic Press, 1973.
9. Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Science* 11:68-85, 1975.
10. Neil D. Jones, editor. *Semantics-Directed Compiler Generation.* Lecture Notes in Computer Science 94. Springer-Verlag, 1980.
11. Neil D. Jones. Flow analysis of lambda expressions. In *Automata Languages and Programming*, Lecture Notes in Computer Science 115. Springer-Verlag, 1981, pp. 114-128.
12. Neil D. Jones. Partial evaluation, self-application and types. In *Automata, Languages and Programming*, Lecture Notes in Computer Science 443. Springer-Verlag, 1990, pp. 639-659.
13. Neil D. Jones. Static semantics, types and binding time analysis. *Theoretical Computer Science* 90:95-118, 1991.
14. Neil D. Jones. The essence of program transformation by partial evaluation and driving. In *Logic, Language and Computation, a Festschrift in Honor of Satoru Takasu,* Masahiko Sato, Neil D. Jones, and Masami Hagiya, editors, Lecture Notes in Computer Science 792. Springer-Verlag, 1994, pp. 206-224.
15. Neil D. Jones. MIX ten years later. In William L. Scherlis, editor, *Proceedings of PEPM '95.* ACM Press, 1995, pp. 24-38.
16. Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann, editors, Lecture Notes in Computer Science 1110. Springer-Verlag, 1996, pp. 216-237.
17. Neil D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, London, 1997.
18. Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Journal of Theoretical Computer Science* 228:151-174, 1999.
19. Neil D. Jones. The expressive power of higher-order types. *Journal of Functional Programming* 11(1):55-94, 2001.
20. Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Mogensen. A self-applicable partial evaluator for the lambda calculus. In *International Conference on Computer Languages.* IEEE Computer Society, 1990, pp. 49-58.
21. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
22. Neil D. Jones, Masami Hagiya, and Masahiko Sato, editors. *Logic, Language and Computation, a Festschrift in Honor of Satoru Takasu.* Lecture Notes in Computer Science 792. Springer-Verlag, 1994. 269 pages.
23. Neil D. Jones and W.T. Laaser. Problems complete for deterministic polynomial time. *Journal of Theoretical Computer Science* 3:105-117, 1977.
24. Neil D. Jones and E. Lien. New problems complete for nondeterministic log space. *Mathematical Systems Theory* 10(1):1-17, 1976.
25. Neil D. Jones and Steven Muchnick. Automatic optimization of binding times. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3, 1976, pp. 77-94.
26. Neil D. Jones and Steven Muchnick. Even simple programs are hard to analyze. *Journal of the ACM* 24(2):338-350, 1977.
27. Neil D. Jones and Steven Muchnick. Complexity of finite memory programs with recursion. *Journal of the ACM* 25(2):312-321, 1978.

28. Neil D. Jones and Steven Muchnick. *Tempo: A Unified Treatment of Binding Time and Parameter Passing Concepts*. Lecture Notes in Computer Science 66. Springer-Verlag, 1978.
29. Neil D. Jones and Steven Muchnick. Flow analysis and optimization of Lisp-like structures. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 6, 1979, pp. 244-256.
30. Neil D. Jones and Steven Muchnick. A flexible approach to interprocedural data flow analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 9, 1982, pp. 66-94.
31. Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 13, 1986, pp. 296-306.
32. Neil D. Jones and Alan Mycroft. A relational approach to program flow analysis. In *Programs as Data Objects,* Harald Ganzinger and Neil D. Jones, editors, Lecture Notes in Computer Science 217. Springer-Verlag, 1986, pp. 112-135.
33. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994, pp. 527-629.
34. Neil D. Jones, P. Russel, A.G. Wilford, and W. Wilson. IBM 7040 Algol compiler. Technical Report. University of Western Ontario, Computer Science Department, 1966.
35. Neil D. Jones and David Schmidt. Compiler generation from denotational semantics. In *Semantics-Directed Compiler Generation*, Neil D. Jones, editor, Lecture Notes in Computer Science 94. Springer-Verlag, 1980, pp. 70-93.
36. Neil D. Jones and Alan L. Selman. Turing machines and the spectra of first order formula with equality. *Journal of Symbolic Logic* 39(1):139-150, 1974.
37. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Conference on Rewriting Techniques and Applications,* J.-P. Jouannoud, editor, Lecture Notes in Computer Science 202. Springer Verlag, 1985, pp. 125-140.
38. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1): 9-50, 1989.
39. Neil D. Jones and Mads Tofte. Towards a theory of compiler generation. In *Proceedings of the Workshop on Formal Software Development Methods,* D. Bjorner, editor. North-Holland, 1986.
40. David Lacey, Neil D. Jones, Eric Van Wyck, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *ACM Symposium on Principles of Programming Languages*, Vol. 29, 2002, pp. 283-294.
41. Chin S. Lee, Neil D. Jones, and Amir Ben-Amran. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, Vol. 28. ACM Press, 2001, pp. 81-92.
42. Kim Marriot, Harald Søndergaard, and Neil D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems* 16(3): 607-648, 1994.
43. Steven Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.
44. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *European Symposium on Programming*, D. Sannella, editor, Lecture Notes in Computer Science 788. Springer-Verlag, 1994, pp. 485-500.



**Neil D. Jones**

# Table of Contents

# General Size-Change Termination and Lexicographic Descent

Amir M. Ben-Amram[*]

The Academic College of Tel-Aviv Yaffo
Tel-Aviv, Israel
amirben@mta.ac.il

**Abstract.** Size-change termination (SCT) is a general criterion to identify recursive function definitions that are guaranteed to terminate. It extends and subsumes the simpler criterion of lexicographic descent in function calls, which in classical recursion theory is known as *multiple recursion.* Neil Jones has conjectured that the class of functions computable by size-change terminating programs coincides with the multiply-recursive functions. This paper proves so.

## 1 Introduction

Consider the following recursive function definition (over the natural numbers — as are all function definitions in this paper). Is computation of this function, by straight-forward evaluation of the defining expressions, guaranteed to terminate?

```
f(x,y) = if x=0 then y
                else f(x-1, y+1)
```

The answer is clrealy positive. In fact, `f` has been defined by *primitive recursion*, a form of recursion which always terminates. In this paper, we take a programming view of function definitions, rather than an abstract equational view; more precisely, we consider a definition as above to be a program in a first-order pure-functional language, with the natural semantics. Thus, we say that the occurrence of `f` on the right-hand side represents a *call* of `f` to itself. If all uses of recursion in the program conform to the primitive recursion scheme, and the built-in operators of the language are appropriate, the functions computable will span the class of primitive recursive functions.

A well-known example of a recursively-defined function which is total but not primitive recursive is Ackermann's function:

```
A(m,n) = if m=0 then n+1
    else if n=0 then A(m-1, 1)
                else A(m-1, A(m,n-1))
```

[*] Part of this research was done while the author was visiting DIKU, the Department of Computer Science at the University of Copenhagen, with support from DART, a project under the Danish Research Councils.

The termination of this computation can be proved by observing that the *pair of arguments* descends lexicographically in every recursive call. This *lexicographic descent criterion* is quite useful in proving termination of functional programs; in recursion theory, the set of functions obtained by recursive definitions of this kind is known as the *multiply-recursive functions* (a more precise definition, in programming terms, appears in Section 2; the classical Recursion-Theoretical reference is Péter [3]). Multiply-recursive definitions extend primitive-recursive ones in two essential ways. First, they use lexicographic descent of a tuple of parameter values as termination guarantee, rather than insisting that the same parameter decrease on every call. Secondly, they allow multiple recursive calls within a definition, which can also nest, as is the case with Ackermann's function. Consider now the following examples:

```
p(m,n,r) = if r>0 then p(m, r-1, n)
      else if n>0 then p(r, n-1, m)
                  else m

g(x,y) = if t=0 then x
    else if x=0 then g(y, y-1)
                else g(y, x-1)
```

Both programs do not satisfy the lexicographic descent criterion, but nonetheless, the termination of these functions can be inferred from the obvious relations between parameter values of calling and called functions. In fact, these programs satisfy the SCT (size-change termination) criterion as formulated by Neil Jones [2][1]. Briefly, satisfaction of this criterion means that if an infinite sequence of recursive calls were possible in a computation, some value would decrease infinitely many times, which is impossible. As is the case with the above examples, this value may meander among parameter positions, and not necessarily decrease in *every* call. SCT also handles mutual recursion naturally, as will be seen from the precise definition in Section 2. SCT clearly subsumes primitive recursion, in which the recursion parameter must decrease in every call; it also subsumes multiple recursion (and does so without reference to lexicographic descent).

Thus, programs satisfying SCT form a superset of multiply-recursive function definitions. But is the class of *functions* that can be computed by such programs larger? Neil Jones has conjectured (in private communication) that these classes coincide. This paper proves that his conjecture is correct.

## 2 Definitions

### 2.1 Programs

We consider programs written in a simple, first-order functional language, with a standard call-by-value semantics (a formal definition can be found in [2] but the

[1] There is earlier work by Sagiv and Lindenstrauss, that used essentially the same idea for termination proofs, see [4, 5].

reader can probably skip it). The only fine detail in the semantics definition is the handling of an "erroneous" expression like `x-1` when `x` is zero (recall that we compute over natural numbers). In [2] we consider the result of evaluating this expression to be a special error value, that essentially aborts the computation. This is a programming-oriented definition, that preserves (even for programs with bugs) the correctness of the basic termination argument, namely that a call sequence that repeatedly replaces `x` with `x-1` must lead to termination.

As in classical recursion theory, the natural numbers will be the only data type in our programs. For practical purposes, size-change termination arguments are also applied to programs using other data types, most notably structures like lists and trees. We note, however, that in analysing a program using such data, a mapping into natural numbers (such as size or height) is typically used in order to argue that recursive calls involve descent. Hence, the essence of the termination arguments is captured completely by working with numbers, with the descent in question relating to the natural order on the natural numbers.

Some notations: the The parameters of function `f` are named, for uniqueness, $\mathit{Param}(\mathtt{f}) = \{\mathtt{f}^{(1)}, \mathtt{f}^{(2)}, \ldots\}$ (though in examples, we prefer to use identifiers as in ordinary programming). We thus consider a parameter name to identify its function uniquely. The set of all parameters in program `p` is $\mathit{Param}(\mathtt{p})$. The *initial function*, denoted $\mathtt{f}_{\mathit{initial}}$, is the entry point of the program, i.e., the function whose termination we wish to prove. We assume that every function in our program is reachable from $\mathtt{f}_{\mathit{initial}}$. Call sites in a function body are identified by (unique) labels. If a call $c$ to function `g` occurs in the body of `f`, we write $c : \mathtt{f} \to \mathtt{g}$, or alternatively, $\mathtt{f} \xrightarrow{c} \mathtt{g}$. A finite or infinite sequence of calls $cs = c_1c_2c_3\ldots$ is *well-formed* if the target function of $c_i$ coincides with the source function of $c_{i+1}$, for every $i$.

We do not fix the set of *primitive operators* available to the program; the successor and predecessor operations, as well as comparison to zero, are necessary and sufficient for computing all multiply-recursive functions, but when writing programs it may be convenient to employ other operators, e.g., subtraction or addition. We do make the (quite natural) assumption that all such operators are primitive recursive, so their presence will not affect the class of computable functions.

## 2.2 Size-Change Graphs

In basic definitions of primitive and multiple recursion the only operator assumed to guarantee descent is the predecessor; however in practice one often makes use of other operators and of knowledge about them. For instance, we may make use of the fact that $x - y \leq x$ to prove termination of the following program:[2]

```
h(m,n) = if m=0 then n
    else if n=0 then h(m-1, n)
                 else h(m-n, n-1)
```

[2] The subtraction operator represents "monus" in this paper, since we deal with non-negative numbers.

The definition of SCT in [2] is made very general by viewing such knowledge about properties of operators or auxiliary functions as part of the problem data. Put otherwise, SCT is defined not as a property of a program *per se* but of a program annotated with size-change information, in the form of *size-change graphs*, defined in the following subsections. Since these definitions are the same as in [2], a reader familiar with the latter may wish to skip to Sec. 2.4.

**Definition 2.1.** *A* state *is a pair* $(\texttt{f}, \boldsymbol{v})$*, where* $\boldsymbol{v}$ *is a finite sequence of integers corresponding to the parameters of* f.
*A* state transition $(\texttt{f}, \boldsymbol{v}) \xrightarrow{c} (\texttt{g}, \boldsymbol{u})$ *is a pair of states connected by a call. It is required that call* $c : \texttt{g}(\texttt{e}_1, \ldots, \texttt{e}_n)$ *actually appear in* f*'s body, and that* $\boldsymbol{u} = (u_1, \ldots, u_n)$ *be the values obtained by expressions* $(\texttt{e}_1, \ldots, \texttt{e}_n)$ *when* f *is evaluated with parameter values* $\boldsymbol{v}$.

**Definition 2.2.** *Let* f, g *be function names in program* p. *A* size-change graph *from* f *to* g*, written* $G : \texttt{f} \to \texttt{g}$*, is a bipartite graph from* $Param(\texttt{f})$ *to* $Param(\texttt{g})$*, with arcs labeled with either* $\downarrow$ *or* $\overline{\downarrow}$*. Formally, the arc set* $E$ *of* $G$ *is a subset of* $Param(\texttt{f}) \times \{\downarrow, \overline{\downarrow}\} \times Param(\texttt{g})$*, that does not contain both* $\texttt{f}^{(i)} \xrightarrow{\downarrow} \texttt{g}^{(j)}$ *and* $\texttt{f}^{(i)} \xrightarrow{\overline{\downarrow}} \texttt{g}^{(j)}$.

The size-change graph is used to describe the given knowledge about relations between parameters of caller and called functions. A *down-arc* $\texttt{f}^{(i)} \xrightarrow{\downarrow} \texttt{g}^{(j)}$ indicates that a value *definitely decreases* in this call, while a *regular arc* $\texttt{f}^{(i)} \xrightarrow{\overline{\downarrow}} \texttt{g}^{(j)}$ indicates that a value does not increase. The absence of an arc between a pair of parameters means that neither of these relations is asserted. For visual clarity, regular arcs will be drawn in diagrams without the $\overline{\downarrow}$ label.

Henceforth $\mathcal{G} = \{G_c \mid c \in C\}$ denotes a set of size-change graphs associated with subject program p, one for each of p's calls. Correctness of the information expressed by the size-change graphs is captured by the following definition of *safety*.

**Definition 2.3.** *Let* f*'s definition contain call* $c : \texttt{g}(\texttt{e}_1, \ldots, \texttt{e}_n)$.

1. *A labeled arc* $\texttt{f}^{(i)} \xrightarrow{r} \texttt{g}^{(j)}$ *is called* safe for call $c$ *if for every state* $(\texttt{f}, \boldsymbol{v})$*, if* $(\texttt{f}, \boldsymbol{v}) \xrightarrow{c} (\texttt{g}, \boldsymbol{u})$*, then* $r = \downarrow$ *implies* $u_j < v_i$*; and* $r = \overline{\downarrow}$ *implies* $u_j \le v_i$.
2. *Size-change graph* $G_c$ *is* safe for call $c$ *if every arc in* $G_c$ *is safe for the call.*
3. *Set* $\mathcal{G}$ *of size-change graphs is a* safe description of program p *if it contains, for every call* $c$*, a graph* $G_c$ *safe for that call.*

*Example.* Recall the definition of Ackermann's function:

```
A(m,n) = if m=0 then n+1
    else if n=0 then A(m-1, 1)
                else A(m-1, A(m,n-1))
```

Number the call sites in the order of their occurrence (from 1 to 3). Then the diagrams in Fig. 1 show a safe size-change graph set for the program.

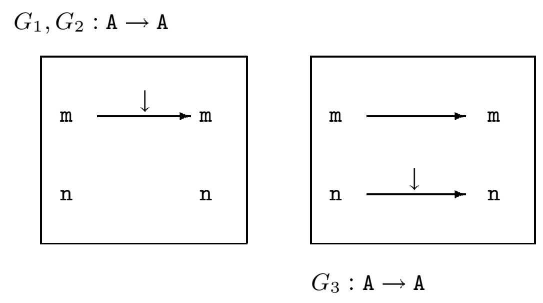


**Fig. 1.** Size-change graphs for the Ackermann function

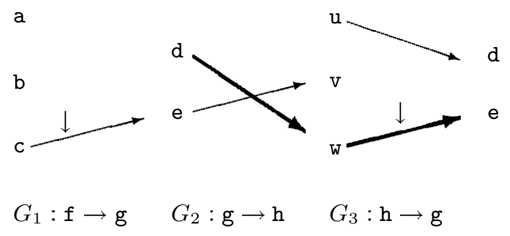


**Fig. 2.** A multipath, with one thread emphasized

## 2.3 The SCT Criterion

**Definition 2.4.** *Let $\mathcal{G}$ be a collection of size-change graphs associated with program* p. *Then:*

1. *A $\mathcal{G}$-*multipath *is a finite or infinite sequence $\mathcal{M} = G_1 G_2 \ldots$ of $G_t \in \mathcal{G}$ such that for all $t$, the target function of $G_t$ is the source function of $G_{t+1}$. It is convenient to identify a multipath with the (finite or infinite)* layered directed graph *obtained by identifying the target nodes of $G_t$ with the source nodes of $G_{t+1}$.*
2. *The* source function of $\mathcal{M}$ *is the source function of $G_1$; if $\mathcal{M}$ is finite, its* target function *is the target function of the last size-change graph.*
3. *A* thread *in $\mathcal{M}$ is a (finite or infinite) directed path in $\mathcal{M}$.*
4. *A thread in $\mathcal{M}$ is* complete *if it starts with a source parameter of $G_1$ and is as long as $\mathcal{M}$.*
5. *Thread th is* descending *if it has at least one arc with $\downarrow$. It is* infinitely descending *if the set of such arcs it contains is infinite.*
6. *A multipath $\mathcal{M}$* has infinite descent *if some thread in $\mathcal{M}$ is infinitely descending.*

To illustrate the definitions, consider the following program fragment; one possible corresponding multipath is shown in Fig. 2.

```
f(a,b,c) =    g(a+b, c-1)
g(d,e)   =    ... h(k(e), e, d)...
h(u,v,w) =    g(u, w-1)
```

**Definition 2.5.** *Let $\mathcal{G}$ be a collection of size-change graphs. $\mathcal{G}$ is SCT (or,* satisfies size-change termination*) if every infinite $\mathcal{G}$-multipath has infinite descent.*

This definition captures the essence of proving termination by impossibility of infinite descent. A program that has a safe set of size-change graphs that satisfies the SCT criterion must terminate on every input (for a formal proof see [2]). For a simple example, consider the Ackermann program; it is easy to see that any infinite multipath, i.e., an infinite concatenation of size-change graphs from $\{G_1, G_2, G_3\}$ (shown in Fig. 1) contains an infinitely descending thread. In fact, in this simple case we have *in-situ* descent, where the descending thread is composed entirely of either arcs $\mathtt{m} \to \mathtt{m}$ or of arcs $\mathtt{n} \to \mathtt{n}$.

*Remark:* We adopt the linguistic abuse of relating the size-change termination property to a program, writing "program `p` is size-change terminating," instead of "`p` has a safe set of size-change graphs satisfying SCT."

For practical testing of the SCT property, as well as for some developments in this paper, it is useful to paraphrase the SCT condition in finitary terms.

**Definition 2.6.** *The* composition *of two size-change graphs $G : \mathtt{f} \to \mathtt{g}$ and $G' : \mathtt{g} \to \mathtt{h}$ is $G;G' : \mathtt{f} \to \mathtt{h}$ with arc set $E$ defined below. Notation: we write $\mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z}$ if $\mathtt{x} \xrightarrow{r} \mathtt{y}$ and $\mathtt{y} \xrightarrow{r'} \mathtt{z}$ are respectively arcs of $G$ and $G'$.*

$$\begin{aligned} E &= \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid \exists \mathtt{y}, r \;.\; \mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \xrightarrow{r} \mathtt{z} \textit{ or } \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{\downarrow} \mathtt{z}\} \\ &\bigcup \{\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z} \mid (\exists \mathtt{y} \;.\; \mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}) \textit{ and} \\ &\qquad\qquad \forall \mathtt{y}, r, r' \;.\; \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z} \textit{ implies } r = r' = \overset{=}{\downarrow}\} \end{aligned}$$

**Definition 2.7.** *For a well-formed nonempty call sequence $cs = c_1 \dots c_n$, define the size-change graph for $cs$, denoted $G_{cs}$, as $G_{c_1}; \dots; G_{c_n}$.*

The composition $G_{cs}$ provides a compressed picture of the multipath $\mathcal{M}(cs) = G_{c_1} \dots G_{c_n}$, in the sense that $G_{cs}$ has an arc $\mathtt{x} \to \mathtt{y}$ if an only if there is a complete thread in $\mathcal{M}$ starting with parameter `x` and ending with `y`; and the arc will be labeled with $\downarrow$ if and only if the thread is descending.

**Definition 2.8.** *Let $\mathcal{G}$ be a collection of size-change graphs for the subject program. Its* closure set *is*

$$\mathcal{S} = \{G_{cs} \mid cs \textit{ is well-formed.}\}$$

Note that the set $\mathcal{S}$ is finite. In fact, its size is bounded by $3^{n^2}$ where $n$ is the maximal length of a parameter list in the program.

**Theorem 2.9 ([2]).** *Program* p *is size-change terminating if and only if for all* $G \in \mathcal{S}$ *such that* $G;G = G$*, there is in* $G$ *an arc of the kind* $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$.

The essence of the theorem is that in order for a program *not to be* size-change terminating, there must be an infinite multipath without infinite descent. Such a multipath can be decomposed into an infinite concatenation of segments, that are all (but possibly the first) represented by the same graph $G$, and $G$ has no arc of the above kind.

## 2.4 Lexicographic Descent

**Definition 2.10.** *Let a program* p *be given, together with safe size-change graphs.*

1. *We say that function* f *has* semantic lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ *if every state-transition sequence from* f *to itself,* $\mathtt{f}(\boldsymbol{v}) \rightarrow \cdots \rightarrow \mathtt{f}(\boldsymbol{u})$*, that can arise in program execution, satisfies* $\boldsymbol{v} >_{lo} \boldsymbol{u}$*, where* $>_{lo}$ *is the standard lexicographic order on* $\mathbb{N}^k$.
2. *We say that function* f *has* observable lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ *if every size-change graph* $G : \mathtt{f} \rightarrow \mathtt{f}$ *in the closure set* $\mathcal{S}$ *for the program satisfies the following condition: there is an* $i \leq k$ *such that for all* $j \leq i$*,* $G$ *contains an arc* $\mathtt{x}_j \xrightarrow{r_j} \mathtt{x}_j$*, and moreover* $r_i = \downarrow$.
3. *We say that* f *has* immediate lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$*, if there are no indirect call paths from* f *to itself; and for each direct call* $c : \mathtt{f} \rightarrow \mathtt{f}$*, the size-change graph* $G_c$ *satisfies the lexicographic descent condition as above.*
4. *We say that program* p *has semantic (resp. observable, immediate) lexicographic descent if every function in* p *has semantic (resp. observable, imediate) lexicographic descent. A function is called* multiply-recursive *if it is computable by a program with immediate lexicographic descent.*

As an example, consider the size-change graphs for the Ackermann program (Fig. 1): this program satisfies immediate lexicographic descent.

Clearly, programs with immediate lexicographic descent form a proper subset of those with observable one, and those with observable descent form a proper subset of those with semantic one.

We next show that all three classes of programs yield the same class of computable functions; thus they all characterize the multiply-recursive functions.

**Theorem 2.11.** *Every program with semantic lexicographic descent can be transformed into an equivalent program, with a matching set of size-change graphs, that has immediate lexicographic descent.*

*Proof.* We first show how to remove mutual recursion from a program, maintaining its *semantic* lexicographic descent.

Let $\mathtt{f}_1, \ldots, \mathtt{f}_n$ be all the functions of the given program. Let the parameters of $\mathtt{f}_i$ be $\mathtt{x}_1^{(i)}, \ldots, \mathtt{x}_{k_i}^{(i)}$, where we assume that they are organized in the order that satisfies lexicographic descent. Let $bit(i) = 2^n - 1 - 2^{i-1}$ (a bit map that has only the $i$th bit clear). We construct the following function, to simulate all functions of the given program, using the parameter `which` as selector.

```
f(S, x_1^(1), x_2^(1), ... , x_1^(2), x_2^(2), ... , x_1^(n), x_2^(n), ..., which) =
    case which of
        1 ⇒ smash(f_1)
        ⋮
        n ⇒ smash(f_n)
```

$$\mathtt{f}(\mathtt{S}, \mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots , \mathtt{x}_1^{(2)}, \mathtt{x}_2^{(2)}, \ldots , \mathtt{x}_1^{(n)}, \mathtt{x}_2^{(n)}, \ldots, \mathtt{which}) =$$

Where smash($\mathtt{f}_i$) is the body of $\mathtt{f}_i$ with every call $\mathtt{f}_j(\mathtt{e}_1, \ldots, \mathtt{e}_{k_j})$ replaced with a call to `f`, in which:

1. Expressions $\mathtt{e}_1, \ldots, \mathtt{e}_{k_j}$ are passed for $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$.
2. The constant $j$ is passed for `which`.
3. `S` is replaced with $(\mathtt{S} \wedge bit(j))$. The *bitwise and* operator $\wedge$, if not present in the language, should be defined as an auxiliary function (it is primitive recursive).
4. All other parameters of `f` (which represent parameters of original functions other than $\mathtt{f}_j$) are passed as received by `f`.

We also add a new initial function (assume w.l.o.g. that the initial function in the source program is $\mathtt{f}_1$):

$$\mathtt{f}_{initial}(\mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots, \mathtt{x}_{k_1}^{(1)}) =$$
$$\quad \mathtt{f}(bit(1), \mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots, \mathtt{x}_{k_1}^{(1)}, 0, \ldots, 0, 1)\,.$$

*Explanation*: by smashing the program into essentially a single function, all recursion loops become loops of immediate recursive calls. The value of the parameter `S` is always a bit map, where the clear bits identify original functions that have already been simulated; in other words, values that `which` has already taken. The usefulness of this parameter is explained below. Note that in the general case, that is, when simulating a call that is not the first call to a given function, the value of `S` is unchanged.

In fact, when simulating a call to $\mathtt{f}_j$ (except for the first one), only the parameters $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ and `which` may change. We know $\mathtt{f}_j$ to have lexicographic descent (in the unrestricted sense, i.e., over any loop). It follows that the new sequence of values of $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ is lexicographically less than the previous sequence. Since all other parameters retain their values, the parameter list of `f` descends lexicographically.

The above argument does not hold with respect to the first call to $\mathtt{f}_j$ that is simulated; here the previous values of $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ are all zero, and the new values are arbitrary. For this reason we introduced the `S` parameter. It strictly decreases whenever an original function is simulated for the first time. Thus, in such a call, lexicographic descent for `f` is ensured as well. We conclude that in the constructed program, `f` has lexicographic descent in every recursive call.

Next we show that semantic lexicographic descent can be generally turned into observable one. We assume that mutual recursion has been removed. Let `f` be any function of the program with parameters $x_1, \ldots, x_k$, ordered for lexicographic descent. We replace every call $f(e_1, \ldots, e_k)$ in the body of `f` with the conditional expression

```
     if $e_1 < x_1$ then $f(\min(e_1, x_1 - 1), e_2, \ldots, e_k)$
else if $e_2 < x_2$ then $f(e_1, \min(e_2, x_2 - 1), \ldots, e_k)$
     ⋮
else if $e_k < x_k$ then $f(e_1, e_2, \ldots, \min(e_k, x_k - 1))$
else $0$
```

It is easy to see that this change does not affect the semantics of the program: the expression $\min(e_i, x_i - 1)$ will only be used when $e_i$ evaluates to less then $x_i$ anyway. However, it is now possible to supply each call with a safe size-change graph so that observable lexicographic descent holds. □

## 3 Stratifiable Programs

In this section we show that for a particular type of programs, called *stratifiable*, size-change termination implies observable lexicographic descent[3]. Since the notion of semantic lexicographic descent will not be used, we omit the adjective "observable" in the sequel.

**Definition 3.1.** *Consider a program with its associated size-change graphs. We call the program* stratifiable *if the parameters of every function can be assigned a partial order* $\preceq$ *so that the existence of a thread leading from* $f^{(i)}$ *to* $f^{(j)}$ *(in any valid multipath) implies* $f^{(i)} \preceq f^{(j)}$.

An example of a stratifiable program is the following:

```
h(m,n) = if m=0 then n
    else if n=0 then h(m, n-1)
                else h(m-1, m)
```

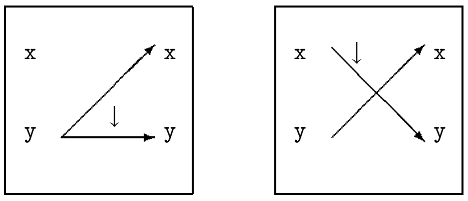
With the obvious size-change graphs for the two calls. The order of parameters is $m \preceq n$; indeed, there is an arc from `m` to `n` (in the graph for the second call) but there is no arc from `n` to `m`.

Fig. 3 shows a program with its size-change graphs that are not stratifiable, since the two parameters depend on each other.

**Theorem 3.2.** *In a stratifiable, size-change terminating program, every function has lexicographic descent. Moreover the lexicographic ordering of the parameters can be effectively (and efficiently) found from the size-change information.*

[3] The term *stratifiable* is used differently in the context of logic programming. However, there should be no conflict of denotations in our context.

```
f(x,y) = if ... then f(y, y-1)
                else f(y, x-1)
```



$G_1 : \texttt{f} \to \texttt{f}$ $G_2 : \texttt{f} \to \texttt{f}$

**Fig. 3.** Example program with size-change graphs.

The theorem follows from the following algorithm to find a lexicographic ordering for a given stratifiable function `f`.

1. Let $\mathcal{S}_{\texttt{f}}$ be the set of all size change graphs $G : \texttt{f} \to \texttt{f}$ in the closure set $\mathcal{S}$ for the given program (see Sec. 2.3). Delete every arc in the graphs of $\mathcal{S}_{\texttt{f}}$ that is not a self-arc ($\texttt{x} \xrightarrow{r} \texttt{x}$). These deletions have no effect on the size-change termination properties, because, due to stratifiability, a thread that visits `f` infinitely often can only move a finite number of times between different parameters of `f`.
2. Let $L = \langle\,\rangle$ (the empty list). $L$ will hold the parameters for the lexicographic ordering.
3. Let `x` be a parameter not in $L$ such that an arc $\texttt{x} \xrightarrow{r} \texttt{x}$ exists in *every* graph of $\mathcal{S}_{\texttt{f}}$, and at least in one of them $r = \downarrow$. Append `x` to $L$.
4. Delete from $\mathcal{S}_{\texttt{f}}$ all size-change graphs in which `x` descends. If $\mathcal{S}_{\texttt{f}}$ remains non-empty, return to Step 3.

*Correctness*: A parameter `x` as required in Step 3 must always exist, because of size-change termination of `p`; specifically, size-change termination implies that every composition of graphs from $\mathcal{S}_{\texttt{f}}$ must contain a descending thread. Note that the parameters already on $L$ will *not* have self-down-arcs in any of the graphs remaining in $\mathcal{S}_{\texttt{f}}$.

It is not hard to verify that $L$ provides a lexicographic ordering for `f`. □

## 4 From SCT to Lexicographic Descent

This section gives an algorithm to transform any size-change terminating program into one with lexicographic descent. The construction can be characterized as *instrumentation* of the given program by adding extra function names and parameters, but without affecting its semantics. A simple example of such an instrumentation is the following. Let `p` have a function `f` which calls itself. Replace `f` with two functions $\texttt{f}_{\texttt{odd}}$ and $\texttt{f}_{\texttt{even}}$ that call each other; the functions are identical to `f` otherwise. We obtain a semantically equivalent program where the function name carries some information on the number of recursive calls performed. Similarly, we can add extra parameters to a function to carry around some information that is not directly used by the program, but may be helpful in its analysis. The goal of our construction is to obtain a semantically equivalent program together with a set of size-change graphs which show that the new program is both SCT and stratifiable. Combined with the result of the last section, this implies lexicographic descent.

## 4.1 Preliminaries

Throughout this section, p denotes the subject program (given with its size-change graphs, and known to satisfy SCT). We construct a new program, p*.

It will be useful to consider a function as receiving an aggregate of parameters, which may be of a different shape than the usual linear list. In fact, in p*, we have functions whose parameters are arranged in a p-multipath. Such a multipath $\mathcal{M}$ is considered to define the shape of the aggregate; a *valued multipath* $M$ is obtained by assigning an integer value to each of the nodes of $\mathcal{M}$. We refer to $\mathcal{M}$ as the *underlying multipath* of $M$.

We introduce some notation for operating with (valued) multipaths. Recall that the expression $\mathcal{M} : \mathtt{f} \to \mathtt{g}$ indicates that f is the *source function* of multipath $\mathcal{M}$ and g its *target function*. The target parameters of $\mathcal{M}$, the front layer of the layered graph, are therefore g's parameters. Remark that the parameters of each subject-program function are assumed to be distinct, so the source and target function are uniquely determined for a multipath.

Say $M$ has $n$ target parameters. The notation:

$$M \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

can be used to name the values of $M$'s target parameters, or to assign them values, according to context.

The basic operation on multipaths is concatenation, written as juxtaposition $\mathcal{M}_1\mathcal{M}_2$. It identifies the target parameters of $\mathcal{M}_1$ with the source parameters of $\mathcal{M}_2$. To name (or set) the values that these parameters take (in a valued multipath) we write

$$M_1 \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} M_2 \,.$$

All we said about multipaths applies in particular to a single size-change graph, which is a multipath of length one, and to a single list of parameters, which we consider as a zero-length multipath.

Recall the composition operation for size-change graphs, denoted by a semicolon (Definition 2.6). Composing all the size-change graphs that form a multipath $\mathcal{M}$ (or the underlying multipath of a valued multipath $M$) yields a single size-change graph which we denote by $\overline{\mathcal{M}}$ (respectively, $\overline{M}$).

## 4.2 Normal Multipaths

The following observation is central to the size-change analysis of the program we construct.

**Observation 4.1.** Let $A$, $B$, $C$, $D$ be multipaths such that $\overline{B} = \overline{C} = \overline{BC}$ (hence, if we let $G = \overline{B}$, then $G$ satisfies $G; G = G$). Every pair of nodes in $A$, $D$ which are connected by a (descending) thread in multipath $ABCD$ are also connected by a (descending) thread in multipath $ACD$.

Informally, *folding* the multipath by replacing $BC$ by $C$ does not affect connectedness properties for nodes outside the folded part (including nodes on the folded part's boundary).

**Definition 4.2.** *A* p*-multipath $\mathcal{M}$ is* foldable *if it can be broken into three parts, $\mathcal{M} = \mathcal{M}_0\mathcal{M}_1\mathcal{M}_2$, such that $\mathcal{M}_1$ and $\mathcal{M}_2$ are of positive length, and $\overline{\mathcal{M}_2} = \overline{\mathcal{M}_1} = \overline{\mathcal{M}_1\mathcal{M}_2}$.*
*A multipath is* normal *if it does not have a foldable prefix.*

**Lemma 4.3.** *There is a finite bound on the length of a normal multipath.*

*Proof.* Let $N$ be the number of possible size-change graphs over p's parameters ($N$ is bounded by $3^{n^2}$, where $n$ is the maximal length of a function's parameter list). Associate a multipath $\mathcal{M} = G_1 G_2 \ldots G_t$ with a complete undirected graph over $\{0, 1, \ldots, t\}$, where for all $i < j$, edge $(i, j)$ is labeled ("colored") with the size-change graph $G_{i+1}; \cdots ; G_j$. By Ramsey's theorem [6, p. 23], there is a number $N'$ such that if $t \geq N'$, there is a monochromatic triangle in the graph. Such a triangle $\{i, j, k\}$, with $i < j < k$, implies that $\mathcal{M}_k = G_1 G_2 \ldots G_k$ is foldable. □

## 4.3 Constructing p*

All functions in the program p* will be called F($M$), with the underlying multipath of $M$ identifing the function referenced. We write F($M : \mathcal{M}$) to name the underlying multipath explicitly.

We define F($M : \mathcal{M}$) for every normal p-multipath $\mathcal{M}$ whose source is p's initial function $\mathtt{f}_{initial}$. The initial function of p* is associated with the zero-length multipath that represents $\mathtt{f}_{initial}$'s parameter list. For each $\mathcal{M} : \mathtt{f}_{initial} \rightarrow$ f, the body of F($M : \mathcal{M}$) is the body of f. Occurrences of f's parameters in the body are replaced with the corresponding target parameters of $M$. Function calls in the body are modified as we next describe. We also describe size-change graphs for the calls.

Consider a call $c : \mathtt{f} \rightarrow \mathtt{g}$ with size-change graph $G$. We distinguish two cases.

*Case* 1: $\mathcal{M}G$ is a normal multipath. In this case the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ becomes

$$\mathtt{F}(MG \begin{pmatrix} \mathtt{e}_1 \\ \vdots \\ \mathtt{e}_n \end{pmatrix}).$$

The associated size-change graph contains a regular arc from each node of $M$ (on the source side) to its counterpart on the target side (in the prefix $M$ of $MG$). This size-change graph is correct because the data in the connected nodes are identical.

*Case* 2: $\mathcal{M}G$ is not normal, so it must be foldable. Let $MG = ABC$ where $\overline{B} = \overline{C} = \overline{BC}$, $B$ and $C$ of positive length. Let $H = \overline{B}$; note that $H$ must have identical sets of source and target parameters, call them $\mathtt{x}_1, \ldots, \mathtt{x}_n$. Assume that

$$MG = A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} B \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} C\,,$$

then the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ becomes

$$\mathtt{F}(A \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} C \begin{pmatrix} \mathtt{e}_1 \\ \vdots \\ \mathtt{e}_n \end{pmatrix})$$

where

$$v_i = \begin{cases} y_i & \text{if } \mathtt{x}_i \xrightarrow{\downarrow} \mathtt{x}_i \in H \\ x_i & \text{if } \mathtt{x}_i \xrightarrow{\downarrow} \mathtt{x}_i \notin H\,. \end{cases}$$

The size-change graph for this call contains an arc from every node in $A$ (on the source side) to its counterpart on the target side. For a parameter $\mathtt{x}_i$ in the front layer of $A$, the arc that connects the source node $\mathtt{x}_i$ to the corresponding target node is a down-arc in case that $v_i = y_i$ (note that the choice of $v_i$ is static). All other arcs are regular.

The correctness of this size-change information should be obvious from the construction of the call.

A call that follows Case 2 is called *a folding call.* The length of multipath $A$ is the *folding depth* of this call.

**Lemma 4.4.** *Program* $\mathtt{p}^*$ *is size-change terminating.*

*Proof.* Consider any infinite multipath $\mathcal{M}^*$ of $\mathtt{p}^*$ (note that such a multipath is a layered graph whose layers are finite $\mathtt{p}$-multipaths, connected by size-change arcs for $\mathtt{p}^*$. See Fig. 4).

Because the number of normal $\mathtt{p}$-multipaths is finite, there must be infinitely many folding calls along $\mathcal{M}^*$. Let $d_j$ be the depth of the $j$th folding call, let

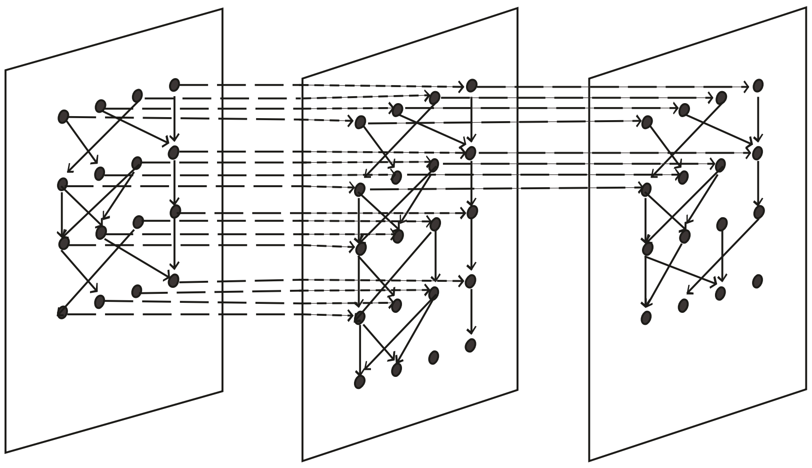


**Fig. 4.** An illustration of a multipath in $\mathtt{p}^*$. The second call is a folding call.

$d = \liminf_{j\to\infty} d_j$, and consider the suffix of $\mathcal{M}^*$ from the first folding at depth $d$ after which there is no folding at a smaller depth. It suffices to show that this multipath has infinite descent.

Let $A$ be the prefix of length $d$ of the $\mathtt{p}$-multipath that starts $\mathcal{M}^*$. Note that $A$ appears unmodified throughout $\mathcal{M}^*$. Each of its $n$ target parameters $\mathtt{x}_i$ carries an in-situ thread throughout $\mathcal{M}^*$. The arc of this thread is a down-arc whenever the condition $\mathtt{x}_i \xrightarrow{\downarrow} \mathtt{x}_i \in H$ is satisfied, where $H$ is the size-change graph representing the folded multipath segment. Now, $H$ is an idempotent size-change graph in the closure set for $\mathtt{p}$ and according to Theorem 2.9 there must be some $i$ such that $\mathtt{x}_i \xrightarrow{\downarrow} \mathtt{x}_i \in H$. Hence in each call, one of the $n$ in-situ arcs has a down-arrow. This clearly means that at least one of the infinite threads is descending. □

## 4.4 Conclusion

From a given size-change terminating program $\mathtt{p}$ we obtained a program $\mathtt{p}^*$ that is size-change terminating and, quite obviously, stratifiable. Therefore, by Theorem 3.2, we obtain

**Theorem 4.5.** *Every size-change terminating program can be effectively transformed into an equivalent program that has lexicographic descent.*

**Corollary 4.6.** *Assume that function $f : \mathbb{N} \to \mathbb{N}$ is computed by a size-change terminating program that uses only primitive recursive operators. Then $f$ is multiply-recursive.*

Suppose we limit the class of multiply-recursive function definitions by disallowing nested recursive calls. This rules out, for instance, the definition of Ackermann's function in Section 1. Péter [3] shows that the functions that can be defined this way are the primitive recursive functions. Since the transformation of `p` to `p`$^*$ never introduces nested calls, we also have

**Corollary 4.7.** *Assume that function* $f : \mathbb{N} \to \mathbb{N}$ *is computed by a size-change terminating program that uses only primitive recursive operators, and has no nested recursion. Then* $f$ *is primitive recursive.*

*A comment on complexity.* Our construction to transform a size-change terminating program into one with lexicographic descent has super-exponential complexity. Is this necessary? It is not hard to verify that the construction is in fact a reduction from the set of size-change terminating program descriptions to the set of such program descriptions that are also stratifiable. We know that the former set is PSPACE-complete [2], while the latter belongs to PTIME [1]. Hence, it is unlikely that a construction of *this kind* can be done in sub-exponential time.

## References

[1] Amir M. Ben-Amram. SCT-1-1 $\in$ PTIME (plus some other cases). unpublished note, Copenhagen, 2001.

[2] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the Twenty-Eigth ACM Symposium on Principles of Programming Languages, January 2001*, pages 81–92. ACM press, 2001.

[3] R. Péter. *Recursive Functions.* Academic Press, 1967.

[4] N. Lindenstrauss, N. and Y. Sagiv. Automatic termination analysis of Prolog programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed, pages 64—77. MIT Press, Leuven, Belgium, 1977. See also: `http://www.cs.huji.ac.il/~naomil/`.

[5] Y. Sagiv. A termination test for logic programs. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct 28–Nov 1, 1991*, V. Saraswat and K. Ueda, Eds. MIT Press, 518–532.

[6] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*, page 23. Cambridge University Press, 1992.

# Comparing Star and Pancake Networks

Linda Morales[1] and I. Hal Sudborough[2] *

[1] Computer Science Department, Texas A&M University-Commerce,
P.O. Box 3011, Commerce, Texas 75429
Linda_Morales@TAMU-Commerce.edu
[2] Department of Computer Science,
Eric Jonnson School of Engineering and Computer Science,
University of Texas at Dallas, Richardson, TX 75083
hal@utdallas.edu

**Abstract.** Low dilation embeddings are used to compare similarities between star and pancake networks. The pancake network of dimension n, $P_n$, has n! nodes, one for each permutation, and an undirected edge between permutations (nodes) when some prefix reversal transforms one permutation into the other. The star network of dimension n, $S_n$, has n! nodes, one for each permutation, and an undirected edge between permutations (nodes) when the exchange of the first element with some other element transforms one permutation into the other. Comparisons with the burnt pancake network are also discussed. The burnt pancake network, $BP_n$, has $2^n \cdot n!$ nodes, one for each signed permutation, and an undirected edge between signed permutations (nodes) when some prefix reversal transforms one signed permutation into the other, and all symbols in the reversed prefix change sign. Some of the embeddings shown are:

- $P_n \overset{\text{dil 1}}{\Rightarrow} S_{2n}$ ,
- $S_n \overset{\text{dil 1}}{\Rightarrow} P_{(n^3-4n^2+5n+4)/2}$
- $BP_n \overset{\text{dil 1}}{\Rightarrow} S_{2n}$ ,
- $S_n \overset{\text{dil 2}}{\Rightarrow} P_{2n-2}$

## 1. Introduction

Star and pancake networks, introduced in [1], have been studied by several researchers [1-5, 7, 9, 10, 11]. Such networks have smaller diameter and fewer connections per processor than other interconnection networks, such as the hypercube [8]. Algorithms are often presented simultaneously for both star and pancake networks, suggesting the networks have similar capabilities. But do they? We consider the structural similarities of star and pancake networks through a study of low dilation embeddings from one to the other. If these networks are similar, then low dilation embeddings should exist. For instance, a one-to-one, dilation one embedding is an isomor-

* This paper has been contributed in honor of Neil Jones' 60th birthday. The second author had the honor of being Neil Jones' first Ph.D. student.

phism and, more generally, a small dilation d>1 embedding means one network can simulate the other with low overhead. We give several low dilation embeddings, but often at the cost of increased dimensionality of the network. We know of no proof that such an increase in dimension is necessary and leave open the issue of whether better embeddings exist. Nonetheless, the embeddings we describe are interesting and describe non-trivial, useful representations of permutation networks.

In addition, we investigate the *burnt pancake network*. The burnt pancake network, suggested by the Burnt Pancake Problem [3, 6, 7], is seemingly a close relative of the pancake network. The burnt pancake network of dimension n has $2^n \cdot n!$ nodes, one node for each permutation on n signed symbols. The edges between nodes are defined by prefix reversals, and, in this case, a prefix reversal changes the signs of the symbols in the reversed prefix. There is a link between two nodes (*i.e.*, permutations on signed symbols) when one permutation is obtained from the other by a prefix reversal. The Burnt Pancake Problem considers the number of steps needed to arrange an arbitrary permutation of signed symbols into sorted order and with all signs positive. It is known that this process takes at most 2n-3 steps [3] and corresponds to the diameter of the burnt pancake network of dimension n. It has been conjectured that $-I_n$, which is the permutation of signed symbols $-I_n = -1, -2, \ldots, -n$, where the symbols are in order but all are negative, is the hardest to sort [3]. It has been shown that $-I_n$ can be sorted in 3(n+1)/2 steps [7].

For $n \geq 2$, let $\Sigma_n$ denote the group of all permutations on n symbols under function composition. There are many distinct sets of generators for $\Sigma_n$. For example:

$$A_n = \{(1 \quad j) \mid j \geq 2\},$$

$$B_n = \left\{ b_i = \begin{pmatrix} 1 & 2 & \ldots & i & i+1 & \ldots & n \\ i & i-1 & \ldots & 1 & i+1 & \ldots & n \end{pmatrix} \middle| \; i \geq 1 \right\}, (B'_n = B_n - \{b_1\}),$$

The notation, (i j) denotes a *transposition* that exchanges the ith and jth symbols, and in particular, (1 j) denotes the *special transposition* that exchanges the first and jth symbols. The permutation $b_i$ denotes a *prefix reversal of size i.*

The Cayley graph defined by $\Sigma_n$ and a set of generators $T_n$, denoted by $G(\Sigma_n, T_n)$, is the graph with one node for each permutation in $\Sigma_n$ and an edge between two nodes (permutations) x and y when y is the composition of x with a generator in $T_n$. Thus, $G(\Sigma_n, A_n)$ is the *star network* of dimension n [1], denoted by $S_n$, and $G(\Sigma_n, B'_n)$ is the *pancake network* of dimension n [1] denoted by $P_n$.

Let $\Phi(X)$ denote the power set of a set X. Given two networks, G=(V,E) and H=(V',E') and an integer d, $f: V \rightarrow \Phi(V')$ is a *one-to-many, dilation d embedding* [5,10] of G into H if, for every pair of vertices x, y in V:

1) $f(x) \cap f(y) = \emptyset$, and
2) if x and y are adjacent in G, then for each $x' \in f(x)$ there corresponds at least one vertex $y' \in f(y)$ for which x' and y' are joined by a path of length at most d.

The one-to-many embedding f is one-to-one embedding if and only if f is a one-to-one function.

One-to-many embeddings have often been used where one-to-one embeddings are not possible. For example, it is not possible for a one-to-one, dilation one embedding to map a cycle of length three into a cycle of length six. However, this can be done with a one-to-many, dilation one embedding. If the cycles are with nodes 1,2,3 and a,b,c,d,e,f, respectively, then one can map 1 to {a,d}, 2 to {b,e}, and 3 to {c,f}. Then, for example, node 1 is adjacent to nodes 2 and 3 in the cycle of length three, and the image node d is adjacent to the image nodes e and c, respectively. Moreover, a one-to-many embedding with dilation d of G into H gives a high level description of how network H can simulate network G. Each image in H of a node x in G computes exactly the same information as any other image of x, therefore no communication is necessary between separate images of x. So, the dilation of the embedding describes the efficiency of the simulation in the same way that it does for one-to-one embeddings. Of course, when an embedding is one-to-many, rather than one-to-one, the host network necessarily is larger than the guest.

We shall often describe one-to-many embeddings through the notion of a co-embedding, which we now define. Let G = (V,E) be a *guest* network and H = (V',E') be a *host* network, and let R be a non-empty subset of V'. A *co-embedding of dilation d* is a pair (R,g), where $g:R \rightarrow V$ is an onto function such that for every $x' \in R$, and any neighbor y of g(x') in G, there exists some $y' \in R$ such that g(y') = y and $\text{distance}_H(x',y') \le d$. We shall often talk about the co-embedding g, its domain R understood implicitly. By the comments above, we see that there exists a one-to-many dilation d embedding of G into H if and only if there exists an appropriate dilation d co-embedding g. Let x and y be adjacent vertices in G. Let $\alpha$ be a generator of G such that $x \circ \alpha = y$. Throughout this paper, we routinely and implicitly specify a one-to-many embedding of G into H by specifying a co-embedding g. In order to demonstrate dilation d, we show how to simulate the action of $\alpha$ on x by the action of an appropriate sequence of d generators $\alpha_1,\alpha_2, ..., \alpha_d$ in H on any vertex x' in H such that g(x') = x. That is, we demonstrate that there is a path of length at most d from x' to a vertex y' in H such that g(y') = y.

The expansion of an embedding is the ratio of the number of nodes in the host network H divided by the number of nodes in the guest network G. Most of our embeddings have unbounded expansion, i.e. the expansion grows with the dimensionality of the embedded network. We therefore use the phrase *"expansion in the dimensionality"* and consider mostly embeddings with linear expansion in the dimensionality of the host network. Specifically, our embeddings are mostly from a network G of dimension k into a network H of dimension at most 2k.

In general, a permutation $\sigma = \begin{pmatrix} 1 & 2 & ... & n \\ \sigma_1 & \sigma_2 & ... & \sigma_n \end{pmatrix}$ will be written as the string $\sigma_1\sigma_2 ... \sigma_n$ or as the sequence $\sigma_1,\sigma_2, ..., \sigma_n$ as is convenient for the particular embedding being described. Also, we use a single arrow ($\rightarrow$) to denote a one-to-one embedding and a double arrow ($\Rightarrow$) to denote a one-to-many embedding.

We begin with embeddings for small dimension networks. For example, in Figure 1, we describe a one-to-one, dilation 2 embedding of $P_4$ into $S_4$. Each node in the illustration has two labels. The unparenthesized label is a permutation in the pancake network, and the parenthesized label beneath it gives the corresponding permutation in the star network. Note that star network nodes 1243 (1423, 4132, 4312) and their neighbors 2143 (4123, 1432, 3412) and 4213 (2413, 3142, 1342), respectively, are assigned to pancake network nodes at distance 2. All other pairs of adjacent nodes in $S_4$ are assigned to adjacent nodes in $P_4$. The reader can easily verify that this is a dilation 2 embedding. Unfortunately, the embedding is not readily generalized to higher dimensions.

There is no dilation 1 embedding of $P_n$ into $S_n$, for n>3, because $P_n$ has cycles of odd length, whereas $S_n$ does not. That is, there can be no isomorphism between the two networks. There is, however, a one-to-one dilation 2 map of $P_4$ into $S_4$. It is simply the inverse of the mapping shown in Figure 1. The reader can easily verify the correctness of the embedding. Also, for the burnt pancake network, Figure 2 specifies a one-to-one dilation 3 embedding of $S_4$ into $BP_3$.

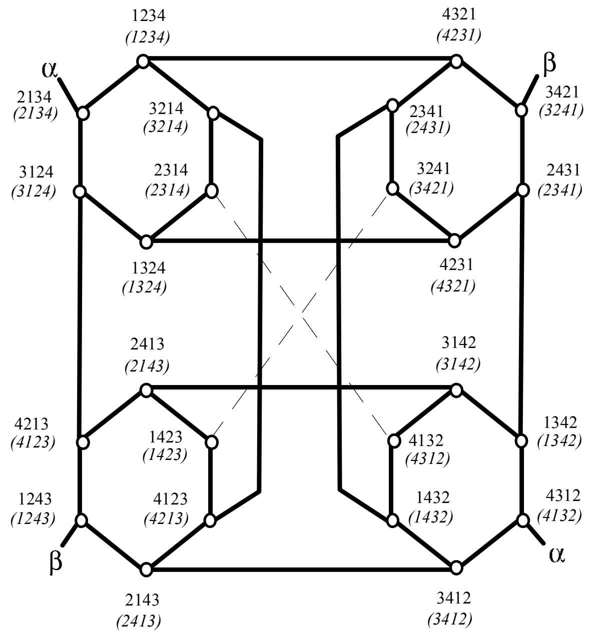


**Figure 1.** A one-to-one, dilation 2 embedding of $S_4$ into $P_4$

## 2. Low Dilation Embeddings with Low Increase in Dimension

To begin our study, we consider a well known natural embedding with dilation 4 of stars into pancakes via the identity map.

**Theorem 1:** $S_n \xrightarrow{\text{dil } 4} P_n$

| Node in $S_4$ | image in $BP_3$ | node in $S_4$ | Image in $BP_3$ |
|---|---|---|---|
| 1234 | 123 | 2143 | 3-21 |
| 2134 | -123 | 4123 | -3-21 |
| 3214 | 2-13 | 1243 | 2-31 |
| 3124 | 213 | 1423 | -231 |
| 2314 | 1-23 | 4213 | -321 |
| 1324 | -1-23 | 2413 | 231 |
| 4231 | -3-2-1 | 3142 | 13-2 |
| 2431 | 23-1 | 4132 | -3-1-2 |
| 3241 | 2-3-1 | 1342 | -13-2 |
| 3421 | -23-1 | 1432 | 1-3-2 |
| 2341 | -2-3-1 | 4312 | -31-2 |
| 4321 | -32-1 | 3412 | -1-32 |

**Figure 2.** A one-to-one, dilation 3 embedding of $S_4$ into $BP_3$

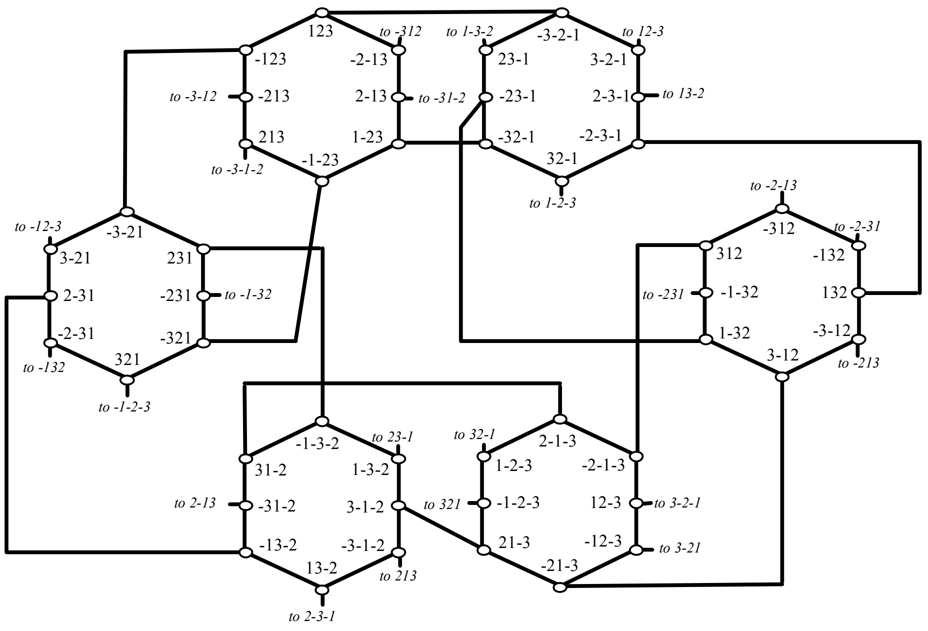


**Figure 3.** $BP_3$, the burnt pancake network of dimension 3

**Proof:** Let $f : S_n \rightarrow P_n$ be the identity map. Let $\sigma$ and $\sigma'$ be permutations at distance 1 in the star network. It follows that there is a special transposition (1 i), for some i, $(2 \le i \le n)$, such that $\sigma' = \sigma \circ (1\ i)$. For i=1,2, the special transpositions (1 2) and (1 3) are identical to prefix reversals of sizes 2 and 3 respectively. For i>3, (1 i) has the same effect as the sequence of four prefix reversals of sizes: i, i-1, i-2, i-1. Hence, f has dilation 4.

We conjecture that the embedding of Theorem 1 has the best dilation possible for any one-to-one embedding of star networks into pancake networks of the *same* dimension. Note that Theorem 1 demonstrates that *fixed* dilation embeddings of $S_n$ into $P_n$ exist. On the other hand, we conjecture that any embedding of the pancake network into the star network of the *same* dimension has *unbounded* dilation. That is, prefix reversals in $P_n$ can change the location and order of arbitrarily many symbols, but a special transposition in $S_n$ changes the position of only two symbols. This suggests that the number of special transpositions needed to simulate a prefix reversal must grow with the size of the prefix reversed. Furthermore, there is a fairly obvious lower bound for the dilation of one-to-one embeddings of pancakes into stars. That is, there is no dilation one embedding of $P_n$ into $S_n$, for $n>3$. This is so, as $P_n$ has cycles of odd length and $S_n$ does not, so $P_n$ is not a subgraph of $S_n$. This also means that there does not exist a one-to-one, dilation one embedding of $S_n$ into $P_n$. If there were, then its inverse would be a one-to-one, dilation one embedding of $P_n$ into $S_n$ (as $P_n$ and $S_n$ have the same number of nodes and edges). That is, $P_n$ and $S_n$ would be isomorphic, which they are not.

It therefore seems interesting, in light of the above conjecture, that one can describe a dilation one embedding of $P_n$ into $S_{2n}$. That is, by doubling the number of symbols used to represent a permutation one can perform by a single special transposition a change that correctly represents an arbitrary prefix reversal. Let us consider details of the representation.

Consider a permutation $\sigma = \sigma_1\sigma_2...\sigma_n$. We develop a representation of $\sigma$ suggestive of a linked list, in which adjacent symbols $\sigma_i$ and $\sigma_{i+1}$ are indicated by an *undirected edge* $\{\sigma_i,\sigma_{i+1}\}$. For example, the permutation $\sigma = 213645$ can be represented by the list 2,{2,1},{1,3},{3,6},{6,4},{4,5},5, where the first symbol, namely 2, represents the *head* of the list, the last symbol, 5, represents the *tail* of the list, and the pairs between the head and the tail represent undirected edges describing adjacencies in $\sigma$. Note that 2n symbols are required to represent a permutation on {1,...,n} in this way. In fact, we shall represent a permutation $\sigma$ on {1,...,n} by a permutation on the 2n symbols {1,...,n,1',...,n'}. (Note that the symbols 1,…,n (unprimed) are distinct from the symbols 1',...,n' (primed)). So, $\sigma_1,\sigma_2,...,\sigma_n$ may be represented, for instance, by $\sigma_1,\sigma'_1,\sigma_2,\sigma'_2,...,\sigma'_{n-1},\sigma_n,\sigma'_n$, where $\sigma'_i$ denotes $\sigma_i$, for all i ($1\leq i\leq n$), and the length two subsequence $\sigma_i,\sigma_{i+1}$ represents the edge $\{\sigma_i,\sigma_{i+1}\}$. In the discussion that follows, we will use the notation $\sigma_1,\sigma'_1,\sigma_2,...,\sigma'_{n-1},\sigma_n,\sigma'_n$ interchangeably with $\sigma_1,\{\sigma'_1,\sigma_2\},...,\{\sigma'_{n-1},\sigma_n\},\sigma'_n$, the latter notation being useful when we wish to emphasize adjacency relationships.

Note that a permutation $\sigma$ is represented by more than one permutation on {1,...,n,1',...,n'}. That is, any reordering of the edge list or of the symbols in any edge represents the same permutation. For example, 2,{2',1},{3,1'},{3',6},{6',4'},{4,5},5' and 2,{2',1},{6,3'},{4,5},{1',3},{4',6'},5' represent the permutation 2,1,3,6,4,5. In addition, any interchanging of $\sigma'_i$ and $\sigma_i$ also represents the same permutation. So, the permutation 2',{2,1},{6',3'},{4',5'},{1',3},{6,4},5 also represents 2,1,3,6,4,5. However, not all permutations on {1,...,n,1',...,n'} represent permutations on {1,...,n} in the desired way. For example, a permutation containing an edge list that denotes a cycle does not represent a permutation on {1,...,n}, and neither does a permutation that contains a pair {j,j'} or {j',j}. Let R denote the set of permutations on {1,...,n,1',...,n'} that *do* represent permutations on {1,...,n} in the manner described. For any permutation $\pi$ in R, we shall always choose the head to be $\pi_1$ and the tail to be $\pi_{2n}$. Note that $\pi$ uniquely determines an edge list E. The edges in E will be pairs of symbols $\{\pi_i, \pi_{i+1}\}$, where i is an even integer. We now give an iterative algorithm to determine which permutation $\sigma$ on {1,...,n}, if any, is represented by a permutation $\pi$ in R.

**Algorithm A:**

```
Initially, set $\sigma_1 = \pi_1$ and t=1. Repeat the following until t = n:
    Each symbol in $\pi$ appears twice: once in primed and once unprimed,
    so *either* there is an edge $\{\sigma_t, \pi_j\}$ (or equivalently, $\{\pi_j, \sigma_t\}$) in E, *or* $\sigma_t$
    is not part of any other edge in E, *i.e.*, $\sigma_t = \pi_{2n}$ is the tail. In the latter
    case, there is no corresponding permutation $\sigma$ on {1,...,n}, and the
    algorithm fails. Otherwise, let $\{\sigma_t, \pi_j\}$ (or $\{\pi_j, \sigma_t\}$) be this edge in E.
    Set $\sigma_{t+1} = \pi_j$, delete the edge $\{\sigma_t,\pi_j\}$ from E, and continue with t=t+1.
```

The set R is exactly that set of permutations that yield a permutation on {1, ..., n} by the above algorithm. We now use this adjacency list representation as the basis of a one-to-many, dilation one embedding of pancakes into stars.

**Theorem 2:** $P_n \overset{\text{dil } 1}{\Rightarrow} S_{2n}$

**Proof:** Let $g:R \rightarrow P_n$ be the co-embedding defined by $g(\pi) = \sigma$, where $\sigma$ is the permutation produced by Algorithm A. We show that g has dilation 1. Let $\sigma$ be an arbitrary permutation on {1,...,n} and let $\pi$ be a permutation in R such that $g(\pi) = \sigma$. Consider a prefix reversal of length i, for any i ($2 \le i \le n$), and let $\sigma'$ be the permutation obtained from $\sigma$ by this reversal. We show that there is a permutation $\pi'$ at distance one from $\pi$ in $S_{2n}$ such that $g(\pi') = \sigma'$.

If i = n, then $\sigma' = \sigma^R$. We simulate a prefix reversal of length n by exchanging $\pi_1=\sigma_1$ and $\pi_{2n}=\sigma_n$. That is, we perform the special transposition (1 2n) on $\pi$. The result is a permutation $\pi'$ with head $\sigma_n$ and tail $\sigma_1$, but otherwise identical to $\pi$. Thus, $\pi$ and $\pi'$ represent exactly the same adjacencies, so $g(\pi') = \sigma'$ as desired. This result is achieved with one special transposition.

For all other permitted values of i, the prefix reversal of length i brings $\sigma_i$ to the front, and in doing so, breaks a single adjacency, *i.e.,* the adjacency between $\sigma_i$ and $\sigma_{i+1}$, and creates a single new adjacency, *i.e.,* the adjacency between $\sigma_1$ and $\sigma_{i+1}$. This adjacency between $\sigma_i$ and $\sigma_{i+1}$ in the permutation $\sigma$ is represented in $\pi$ as an edge between $\sigma_i$ and $\sigma_{i+1}$, and there is also a representation of $\sigma_1$ as $\pi_1$. Therefore, to simulate in the permutation $\pi$ this prefix reversal, it is sufficient to exchange $\pi_1$ with the representative of $\sigma_i$ in the edge in $\pi$ representing the adjacency between $\sigma_i$ and $\sigma_{i+1}$. This has the effect of replacing the adjacency between $\sigma_i$ and $\sigma_{i+1}$ with an adjacency between $\pi_1=\sigma_1$ and $\sigma_{i+1}$, and of moving a representative of $\sigma_i$ to the front. Consequently this does, in fact, result in a representation of $\sigma'$, as desired. This interchange can be done by a single special transposition, as it is an exchange of the first symbol with another, hence the embedding has dilation 1.

One can do better. We show that $BP_n \overset{\text{dil}1}{\Rightarrow} S_{2n}$. In fact, $P_n \overset{\text{dil}1}{\Rightarrow} S_{2n}$ follows directly from $BP_n \overset{\text{dil}1}{\Rightarrow} S_{2n}$, as there is a straightforward dilation 1, one-to-many embedding of $P_n$ into $BP_n$. That is, map any permutation $\pi$ into all signed permutations $\pi'$ such that, if one ignores the signs, each $\pi'$ is equal to $\pi$. For example, $\pi$=132 maps to the permutations $\pi'$ ={(-1)(-3)(-2), 1(-3)(-2), 1,3,2, …}. Use the fact that a sequence of n signed integers can be represented by a sequence of 2n unsigned integers. That is, for each i ($1 \le i \le n$), represent the signed integer i by the integers 2i-1 and 2i, which are adjacent and never separated. The sign of the integer i is represented by the relative order of the integers 2i-1 and 2i. That is, the sign is positive if 2i-1 is before 2i, and is negative otherwise. As the pair 2i-1 and 2i remain adjacent, their adjacency need not be represented in our list of adjacencies. That is, our edge list need only describe adjacencies between even and odd integers. So, the edge list consists of n-1 undirected pairs.

The construction resembles that of Theorem 2. In this case, the co-domain is a subset of the permutations on {1,...,2n}. As noted, adjacencies between certain pairs of integers are implicit and are not explicitly represented. For example, the permutation $\sigma$=23(-1)456 is represented by the list 3,{4,5},{6,2},{1,7},{8,9},{10,11},12, or equivalently by the list, 3,{5,4},{8,9},{2,6},{7,1},{10,11},12. That is, with 3=2∗2-1 at the head of the list, the permutation begins with the signed integer +2. Then, as the other half of the pair representing +2 is 4, and 4 is paired with 5=2∗3-1, the next element of the signed permutation is +3, and so on. Observe that a signed permutation may be represented by more than one unsigned permutation because any reordering of the edge list or of the symbols in any edge represents the same signed permutation. However, not all permutations on {1,...,2n} represent signed permutations on {1,...,n}. For example, a permutation with a pair representing an adjacency between 2i-1 and 2i does not correspond to *any* permutation in the domain. This is so, as integers 2i-1 and 2i are always adjacent, so the representation never represents an edge between them. And, if such an edge were to be listed, it would be at the expense of omitting a needed adjacency.

Let R denote the set of permutations on {1,...,2n} that *do* represent permutations on {1,...,n} in the manner described. For any permutation $\pi = \pi_1\pi_2...\pi_{2n}$ in R, we shall always choose the head to be $\pi_1$ and the tail to be $\pi_{2n}$. Note that $\pi$ uniquely determines an edge list E. That is, the edges in E will be pairs of symbols $\{\pi_{2i-1}, \pi_{2i}\}$, where i is an integer. We now give an iterative algorithm to determine which permutation $\sigma$ on {1,...,n}, if any, is represented by a permutation $\pi$ in R.

**Algorithm B:**

Initially, set t=1. If $\pi_1$ is even, set $\sigma_1 = -\left\lceil \frac{\pi_1}{2} \right\rceil$, and set $j = \pi_1 - 1$.

Otherwise, set $\sigma_1 = +\left\lceil \frac{\pi_1}{2} \right\rceil$, and set $j = \pi_1 + 1$.

```
Do the following until t = n:
    Locate the subset {j, $\pi_k$} in the edge list E. If {j, $\pi_k$}={2i-1, 2i},
    for some i, 1≤i≤n, there is no corresponding signed permutation $\sigma$
    on the symbols {1,...,n}, and the algorithm fails. Otherwise, if $\pi_k$
    is even, set $\sigma_{t+1} = -\left\lceil \frac{\pi_k}{2} \right\rceil$, and set j = $\pi_k$-1. On the other hand, if $\pi_k$
    is odd, set $\sigma_{t+1} = +\left\lceil \frac{\pi_k}{2} \right\rceil$, and set j = $\pi_k$+1. Continue with t = t+1.
```

The set R described earlier is exactly that set of permutations that yield a signed permutation on {1, ..., n} by the above algorithm. We now present a one-to-many, dilation one embedding of the burnt pancake network of dimension n into the star network of dimension 2n.

**Theorem 3:** $BP_n \overset{\text{dil } 1}{\Rightarrow} S_{2n}$.

**Proof:** Let g:R→$BP_n$ be the co-embedding defined by g($\pi$) = $\sigma$, where $\sigma$ is the signed permutation produced by Algorithm B. We show that g has dilation 1. Let $\sigma$ be an arbitrary signed permutation on the symbols {1,...,n} and let $\pi$ be a permutation in R such that g($\pi$) = $\sigma$. Consider a prefix reversal of length i, for any i (1≤i≤n), and let $\sigma'$ be the permutation obtained from $\sigma$ by this reversal. We show that there is a permutation $\pi'$ at distance one from $\pi$ in $S_{2n}$, that is, that g($\pi'$) = $\sigma'$.

If i = n, then $\sigma'$ is the reversal of $\sigma$ with all signs switched. We simulate a prefix reversal of length n by exchanging $\pi_1$ and $\pi_{2n}$. That is, we perform the special transposition (1 2n) on $\pi$. The result is a permutation $\pi'$ with head $\pi_{2n}$ and tail $\pi_1$, but otherwise identical to $\pi$. In particular, $\pi$ and $\pi'$ represent exactly the same adjacencies. The effect is that the edge list is now in reverse order, and the output of Algorithm B is a permutation that is the reversal of $\sigma$, with the signs of all symbols changed. Note that the sign change for each symbol is accomplished simply by the fact that the edge list is now read in reverse order. That is, g($\pi'$) = $\sigma'$, as desired. This result is achieved with one special transposition.

Next suppose that i = 1. The effect of a prefix reversal of length 1 is to change the sign of $\sigma_1$, leaving $\sigma_2$,...,$\sigma_n$ unaltered. If $\sigma_1$ is positive, then $\pi_1$=2|$\sigma_1$|-1, and the adjacency between $\sigma_1$ and $\sigma_2$ is represented in $\pi$ by a subset {$\pi_m$, $\pi_{m'}$} in E, where $\pi_m$ = 2|$\sigma_1$|. We simulate the prefix reversal of length i by exchanging $\pi_1$ with $\pi_m$. The result is a permutation $\pi'$ with head 2|$\sigma_1$|, and in which the edge representing the adjacency between $\sigma_1$ and $\sigma_2$ contains the symbol 2|$\sigma_1$|-1. The permutation $\pi'$ is otherwise identical to $\pi$. Thus, $\pi$ and $\pi'$ represent exactly the same adjacencies, however, $\pi'_1$ is now even. Hence, Algorithm B assigns a negative sign to $\sigma_1$ and yields the same output for the remaining symbols. So g($\pi'$) = $\sigma'$ as desired. This is achieved with one transposition. (The case in which $\sigma_1$ is negative is proved in a similar fashion, noting however that $\pi_1$= 2|$\sigma_1$| and $\pi_m$ = 2|$\sigma_1$|-1.)

For all other permitted values of i, the prefix reversal of length i brings $\sigma_i$ to the front, and in doing so, breaks a single adjacency, *i.e.,* the adjacency between $\sigma_i$ and $\sigma_{i+1}$, and creates a single new adjacency, *i.e.,* the adjacency between $\sigma_1$ and $\sigma_{i+1}$. In the process, the signs of $\sigma_1,\ldots,\sigma_i$ are changed in the same manner as noted earlier. The adjacency between $\sigma_i$ and $\sigma_{i+1}$ is represented in $\pi$ by a subset $\{\pi_m, \pi_{m'}\}$ in E, where

$$\{\pi_m, \pi_{m'}\} = \begin{cases} \{2|\sigma_i|, 2|\sigma_{i+1}|-1\}, & \text{if both } \sigma_i \text{ and } \sigma_{i+1} \text{ are positive,} \\ \{2|\sigma_i|-1, 2|\sigma_{i+1}|\}, & \text{if both } \sigma_i \text{ and } \sigma_{i+1} \text{ are negative,} \\ \{2|\sigma_i|-1, 2|\sigma_{i+1}|-1\}, & \text{if } \sigma_i \text{ is negative and } \sigma_{i+1} \text{ is positive,} \\ \{2|\sigma_i|, 2|\sigma_{i+1}|\}, & \text{if } \sigma_i \text{ is positive and } \sigma_{i+1} \text{ is negative.} \end{cases}$$

We simulate a prefix reversal of length i by exchanging $\pi_1$ and $\pi_m$. The result is a permutation $\pi'$ with head $\pi_m$ and in which the edge $\{\pi_m, \pi_{m'}\}$ is replaced by $\{\pi_1, \pi_{m'}\}$, but otherwise $\pi'$ is identical to $\pi$. The effect is that the portion of the edge list that represents the adjacencies between $\sigma_1,\ldots,\sigma_i$ is now processed by Algorithm B in reverse order. Hence the output of Algorithm B is a permutation in which the order of $\sigma_1,\ldots,\sigma_i$ is reversed, and the signs of these symbols are changed. So $g(\pi') = \sigma'$ as desired. This is achieved with one transposition.

For example, the permutation $\sigma$=23(-1)456 can be represented by $\pi$=3,{4,5},{6,2},{1,7},{8,9},{10,11},12. We simulate a prefix reversal of size four by performing the special transposition (1 8) on $\pi$ as illustrated below:

$$\begin{aligned} \pi &= 3,\{4,5\},\{6,2\},\{1,7\},\{8,9\},\{10,11\},12 \\ &\mapsto 8,\{4,5\},\{6,2\},\{1,7\},\{3,9\},\{10,11\},12 \\ &= \pi'. \end{aligned}$$

It is easily verified that $g(\pi')$ = (-4)1(-3)(-2)56 = $\sigma'$, as desired.

Theorem 1 can also be generalized to yield $S_n \overset{\text{dil } 6}{\rightarrow} BP_{n-1}$, where the dimensionality now decreases. The idea is to represent a permutation on n symbols by an induced permutation on n-1 symbols, each symbol signed, but with at most one symbol being negative at any one time. That is, n-1 symbols are placed in the same relative order as in the represented permutation on n symbols; the location of the missing symbol, which we choose to be the largest integer, n, is determined by the symbol with a negative sign, if there is one. Specifically, the missing symbol n is located immediately to the right of the negative symbol, if there is a negative symbol; otherwise, n is in the initial position. For example, using this interpretation, the permutation 21(-5)34 of five signed symbols on {1,2,3,4,5} represents the permutation 215634 on {1,2,3,4,5,6}. We show in the following theorem that this representation can be maintained throughout a simulation of an arbitrary sequence of special transpositions, using at most six prefix reversals to simulate each special transposition.

**Theorem 4:** $S_n \overset{\text{dil } 6}{\rightarrow} BP_{n-1}$

**Proof:** Let f be the one-to-one function mapping permutations on {1,...,n} into permutations of signed symbols on {1,...,n-1} with at most one negative symbol, defined by

$$f(\sigma_1\sigma_2...\sigma_n) = \begin{cases} \sigma_1\sigma_2...\sigma_{i-1}(-\sigma_i)\sigma_{i+2}...\sigma_n, & \text{if } \sigma_{i+1} = n, \text{for some i } (1 \le i \le n-1), \\ \sigma_2...\sigma_n, & \text{otherwise.} \end{cases}$$

We show that any special transposition can be simulated by a sequence of at most six prefix reversals, by considering four cases based on the position of the symbol 'n' in $\sigma$. Let $\sigma = \sigma_1\sigma_2...\sigma_{i-1}\sigma_i\sigma_{i+1}...\sigma_n$ and $\sigma'$ be permutations on {1,...,n} such that $\sigma'$ is at distance 1 from $\sigma$ in the star network. It follows that there is a special transposition (1 i) for some i, $2 \le i \le n$, such that $\sigma' = \sigma \circ (1\ i)$.

*Case 1:* In $\sigma$, the symbol 'n' is either to the left of $\sigma_i$ or to the right of $\sigma_{i+1}$. That is, $\sigma_j = n$ for some j, where either $1<j<i$, or $i+1<j<n$. By the definition of f this means that in $f(\sigma)$, $\sigma_j$ is missing and the negative symbol, $(-\sigma_{j-1})$, is either to the left of $\sigma_i$ or to the right of $\sigma_{i+1}$, respectively. In either case, the special transposition (1 i) can be simulated by the sequence of six prefix reversals of sizes: 1, i, i-1, i-2, i-1, 1, as demonstrated below for the case in which $(-\sigma_{j-1})$ is to the right of $\sigma_{i+1}$:

$$\begin{aligned} f(\sigma) &= \sigma_1\sigma_2...\sigma_{i-1}\sigma_i\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto (-\sigma_1)\sigma_2...\sigma_{i-1}\sigma_i\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto (-\sigma_i)(-\sigma_{i-1})...(-\sigma_2)\sigma_1\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto \sigma_2...\sigma_{i-1}\sigma_i\sigma_1\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto (-\sigma_{i-1})...(-\sigma_2)\sigma_i\sigma_1\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto (-\sigma_i)\sigma_2...\sigma_{i-1}\sigma_1\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &\mapsto \sigma_i\sigma_2...\sigma_{i-1}\sigma_1\sigma_{i+1}...(-\sigma_{j-1})\sigma_{j+1}...\sigma_n \\ &= f(\sigma'). \end{aligned}$$

*Case 2:* In $\sigma$, the symbol 'n' occupies the (i+1)st position, *i.e.,* $\sigma_{i+1} = n$. By the definition of f this means that in $f(\sigma)$, $\sigma_{i+1}$ is not present, and the negative symbol is $\sigma_i$. The special transposition (1 i) can be simulated by the sequence of four prefix reversals of sizes: i, i-1, i-2, i-1, demonstrated below:

$$\begin{aligned} f(\sigma) &= \sigma_1\sigma_2...\sigma_{i-1}(-\sigma_i)\sigma_{i+2}...\sigma_n \\ &\mapsto \sigma_i(-\sigma_{i-1})...(-\sigma_2)(-\sigma_1)\sigma_{i+2}...\sigma_n \\ &\mapsto \sigma_2...\sigma_{i-1}(-\sigma_i)(-\sigma_1)\sigma_{i+2}...\sigma_n \\ &\mapsto (-\sigma_{i-1})...(-\sigma_2)(-\sigma_i)(-\sigma_1)\sigma_{i+2}...\sigma_n \end{aligned}$$

$\mapsto \sigma_i\sigma_2 \ldots \sigma_{i-1}(-\sigma_1)\sigma_{i+2} \ldots \sigma_n$
$= f(\sigma')$.

*Case 3:* In $\sigma$, the symbol 'n' occupies the ith position, *i.e.,* $\sigma_i = n$. By the definition of f this means that in $f(\sigma)$, $\sigma_i$ is not present, and the negative symbol is $\sigma_{i-1}$. The special transposition (1 i) can be simulated by the sequence of four prefix reversals of sizes: 1, i-1, 1, i-2, demonstrated below:

$f(\sigma) = \sigma_1\sigma_2 \ldots \sigma_{i-2}(-\sigma_{i-1})\sigma_{i+1} \ldots \sigma_n$
$\mapsto (-\sigma_1)\sigma_2 \ldots \sigma_{i-2}(-\sigma_{i-1})\sigma_{i+1} \ldots \sigma_n$
$\mapsto \sigma_{i-1}(-\sigma_{i-2}) \ldots (-\sigma_2)\sigma_1\sigma_{i+1} \ldots \sigma_n$
$\mapsto (-\sigma_{i-1})(-\sigma_{i-2}) \ldots (-\sigma_2)\sigma_1\sigma_{i+1} \ldots \sigma_n$
$\mapsto \sigma_2 \ldots \sigma_{i-2}\sigma_{i-1}\sigma_1\sigma_{i+1} \ldots \sigma_n$
$= f(\sigma')$.

*Case 4:* In $\sigma$, the symbol 'n' occupies the first position, *i.e.,* $\sigma_1 = n$. By the definition of f this means that in $f(\sigma)$, $\sigma_1$ is missing, and all of the symbols are positive. The special transposition (1 i) can be simulated by the sequence of four prefix reversals of sizes: i-2, 1, i-1, 1, demonstrated below: (Note that this is simply the reverse of the sequence of steps performed for Case 3.)

$f(\sigma) = \sigma_2 \ldots \sigma_{i-2}\sigma_{i-1}\sigma_i\sigma_{i+1} \ldots \sigma_n$
$\mapsto (-\sigma_{i-1})(-\sigma_{i-2}) \ldots (-\sigma_2)\sigma_i\sigma_{i+1} \ldots \sigma_n$
$\mapsto \sigma_{i-1}(-\sigma_{i-2}) \ldots (-\sigma_2)\sigma_i\sigma_{i+1} \ldots \sigma_n$
$\mapsto (-\sigma_i)\sigma_2 \ldots \sigma_{i-2}(-\sigma_{i-1})\sigma_{i+1} \ldots \sigma_n$
$\mapsto \sigma_i\sigma_2 \ldots \sigma_{i-2}(-\sigma_{i-1})\sigma_{i+1} \ldots \sigma_n$
$= f(\sigma')$.

These four cases enumerate all possibilities for the position of n in $\sigma$, and prove that a special transposition in the star network of dimension n can be simulated by at most six prefix reversals in the burnt pancake network of dimension n-1. Hence, the embedding has dilation 6.

The embedding given in Theorem 1 had no expansion but had dilation 4. We conjectured that no smaller dilation is possible for embeddings of the star network into the pancake network of the same dimension. However, embeddings with lower dilation than given in Theorem 1 are possible, if one is willing to allow expansion. Again this is done with embeddings using non-standard representations of permutations. There are two pieces of information about a symbol in a permutation that must be represented in any embedding: the symbol's identity and its position. With the identity embedding, this is done in the direct and obvious way, *i.e.,* each symbol represents itself, and its position is exactly where it resides in the permutation. However, there are other possibilities. Each symbol in a permutation can be represented by *two* objects, one describing the symbol's position and the other describing its identity. This approach is used in Theorem 5 where we describe a dilation 2, one-to-many embedding of the star network of dimension n into the pancake network of dimension 2n-2.

Consider a set of symbols $\Psi=\{1,...,n\}$, and let $\Sigma_n$ be the set of permutations over $\Psi$. Let $\Delta$ be a set of n-2 new symbols, say $\Delta = \{a_2, a_3, ... ,a_{n-1}\}$. We represent permutations in $\Sigma_n$ with permutations over $\Psi\cup\Delta$. Specifically, the permutations over $\Psi\cup\Delta$ used to represent permutations in $\Sigma_n$ will be strings in the regular set
$R=\Psi(\Psi\Delta\cup\Delta\Psi)^{n-2}\Psi$. These are permutations whose first and last symbols are from $\Psi$, and in between have n-2 pairs of symbols, each pair having one symbol from $\Psi$ and one symbol from $\Delta$. A permutation $r = \pi_1\lambda_2 ... \lambda_{n-1}\pi_n$ in R, with $\pi_1$ and $\pi_n$ in $\Psi$ and $\lambda_i$ in $\Psi\Delta\cup\Delta\Psi$, for all i ($2\le i\le n-1$), denotes the permutation g(r) in $\Sigma_n$, where the co-embedding g is defined by

$$g(r) = \sigma_1\sigma_2...\sigma_n\text{, where } \sigma_i = \begin{cases} \pi_i, & \text{for } i = 1 \text{ and } i = n, \\ j, & \text{if there is a pair } \lambda_k\text{, for some k, containing} \\ & \text{the two symbols } a_i \text{ and j, for i, } 2 \le i \le n-1. \end{cases}$$

The idea is to represent each symbol $\sigma_i$, ($2\le i\le n-1$), with two objects. That is, $\sigma_i$ is denoted by the pair $\lambda_k$, where $\lambda_k=a_i\,j$ or where $\lambda_k=ja_i$, for some j. The object $a_i$ gives the symbol's position, and the object j identifies the symbol itself. For example, the string $3a_21a_452a_34$ denotes the permutation 31254, as 1 is paired with $a_2$, 2 is paired with $a_3$, 5 is paired with $a_4$, and 3 and 4 are the beginning and ending symbols, respectively.

Observe that a permutation in $\Sigma_n$ has more than one representative in R. That is, if $\pi_1\lambda_2 ... \lambda_{n-1}\pi_n$ in R denotes the permutation $\sigma_1\sigma_2 ... \sigma_n$ in $\Sigma_n$, then so does the string $\pi_1\lambda_{\rho(2)} ... \lambda_{\rho(n-1)}\pi_n$ in R, where $\rho$ is any permutation of $\{2,...,n-1\}$. In other words, changing the order of the pairs in the string does not change the permutation (in $\Sigma_n$) denoted. This is true, as each pair contains both a representation of a position i, by the existence of a symbol $a_i$, and a representation of the ith symbol in the permutation in $\Sigma_n$. So, if the order is changed, the result still denotes the same permutation. Also, note that the order of items within a pair is unimportant.

**Theorem 5:** For all n>3, $S_n \overset{\text{dil 2}}{\Rightarrow} P_{2n-2}$.

**Proof**: Let $g : R\rightarrow \Sigma_n$ be the co-embedding described above. We show that the co-embedding has dilation 2. Let $\sigma = \sigma_1\sigma_2...\sigma_{i-1}\sigma_i\sigma_{i+1}...\sigma_n$ be an arbitrary permutation on $\{1,...,n\}$ and let (1 i), $2\le i\le n$, be one of the special transpositions that define the edges of the star network. Let $\sigma'$ denote the permutation obtained from $\sigma$ by applying the special transposition (1 i), *i.e.,* $\sigma' = \sigma_i\sigma_2...\sigma_{i-1}\sigma_1\sigma_{i+1}...\sigma_n$. Let $r = \pi_1\lambda_2 ... \lambda_{n-1}\pi_n$ be a permutation in R such that $g(r) = \sigma$. We show that with at most two prefix reversals we can obtain from r a string r' such that $g(r')=\sigma'$.

First observe that if the special transposition is (1 n), then the reversal $r^R$ of the entire string (permutation) r is such that $g(r^R) = \sigma'$. That is, $g(r^R)$ is the permutation with $\pi(1)$ and $\pi(n)$ exchanged, *i.e.* the permutation $\sigma' = \sigma_n\sigma_2...\sigma_{n-1}\sigma_1$. Note that this reversal exchanges the first and last elements in $r^R$, but the list of pairs between them is unchanged except for order.

Next consider a special transposition of the form (1 i), where $2 \le i \le n-1$. We consider two cases, based on the permutation $r = \pi_1\lambda_2 \ldots \lambda_{n-1}\pi_n$ in R, and depending on whether the pair $\lambda_j$ that contains $a_i$ has $a_i$ as the first or second object in the pair.

If $\lambda_j$ contains $a_i$ first, then the ith element of the permutation in $\Sigma_n$ that r denotes, say $\sigma_i$, must be second. As the special transposition (1 i) exchanges the first element of the permutation, say $\sigma_1$, with the ith element, $\sigma_i$, we obtain a string in R that denotes this permutation by first doing a reversal of the prefix that includes everything up to, but not including, the symbol $a_i$, and then doing a reversal of the prefix of everything up to and including the symbol $\sigma_i$. These two prefix reversals have the cumulative effect of moving $\sigma_i$ to the front, and putting $\sigma_1$ in a pair with $a_i$. Thus, the result is an appropriate representative.

If $\lambda_j$ contains $a_i$ second, then the ith element of the permutation in $\Sigma_n$ that r denotes, say $\sigma_i$, must be first in this pair. As the special transposition (1 i) exchanges the first element of the permutation, say $\sigma_1$, with the ith element, $\sigma_i$, we obtain a string in R that denotes this permutation by a reversal of the prefix that includes everything up to and including the symbol $\sigma_i$. This prefix reversal has the effect of moving $\sigma_i$ to the front, putting $\sigma_1$ in a pair with $a_i$, and simply reversing the order of the pairs $\lambda_2, \ldots, \lambda_{i-1}$. Thus, the result is an appropriate representative.

## 3. A Dilation One, Large Expansion Embedding

Theorem 5 gave a one-to-many, dilation 2 embedding of the star network of dimension n into the pancake network of dimension 2n-2. We can use an entirely different technique which results in an embedding with dilation one. To achieve this result, we represent each symbol by a *block* of symbols, where the particular symbol being represented is denoted by the number of symbols in the block. In addition, the position of each symbol is uniquely deduced from the size of the block representing it. In Theorem 6 we describe a dilation one, one-to-many embedding of stars into pancakes, in which distinct symbols are represented (in unary notation) by the size of an associated block of symbols in the host network and the sizes of blocks are such that the position of each symbol is encoded uniquely as well. As we shall see below, this embedding requires $O(n^3)$ symbols to represent permutations on $\{1,...,n\}$.

It is helpful to describe our mapping as one from permutations on $\{1,...,n\}$ into permutations defined over the union of two sets of symbols, $A=\{A_1,A_2,A_3,...,A_t\}$, for some $t \ge 1$, and $B= \{B_1,B_2,B_3,...,B_{n-1}\}$. The elements of A are called *A-symbols*, or simply A's, and the elements of B, similarly, *B-symbols*, or simply B's. $A^X$ denotes the concatenation of x many symbols from **A**, and is called an *A-block of size x*. The B-symbols serve only as dividers between A-blocks and will be treated as indistinguishable, so we omit subscripts on the B's. Also, as we use only the *size* of the A-blocks in our mapping, we drop subscripts on the A's as well.

Let m=n+t-1, and $P_m$ be the set of permutations defined on the symbols in $A \cup B$. In particular, each permutation in $P_m$ will be viewed as a string of A-blocks interleaved with B-symbols. We are interested only in *well-formed* permutations, which are permutations for which every pair of B's is separated by at least one A. These permutations have the form $A^{x_1}BA^{x_2}B \ldots BA^{x_i}B \ldots A^{x_{n-1}}BA^{x_n}$, where $x_i \geq 1$ for $1 \leq i \leq n$.

Given a permutation $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ in $\Sigma_n$, our intention is to associate with each integer i ($1 \leq i \leq n$), an A-block in the corresponding permutation $\pi$. Moreover, integers i and j, $i \neq j$, will have disjoint size ranges for their associated A-blocks. Thus, the size, $x_i$, of the A-block representing $\sigma_i$ gives us the position i and the symbol occupying the ith position, namely $\sigma_i$. Specifically, the size of the A-block that represents $\sigma_i$ is:

$$x_i = (i-1)n + \sigma_i. \tag{1}$$

So, the permutation $\sigma = 216453$ can be represented, for example, by the permutation $\pi = A^2BA^7BA^{18}BA^{22}BA^{29}BA^{33}$. By defining a distinct range of values for $\sigma_i$ and $\sigma_j$, $i \neq j$, the order of the A-blocks becomes irrelevant. Thus, the well-formed permutations $\pi = A^2BA^7BA^{18}BA^{22}BA^{29}BA^{33}$ and $\pi' = A^2BA^{33}BA^{18}BA^{29}BA^7BA^{22}$ both represent the permutation $\sigma = 216453$. However, not every well-formed permutation $\pi = A^{x_1}BA^{x_2}B \ldots BA^{x_i}B \ldots A^{x_{n-1}}BA^{x_n}$ represents a permutation in $\Sigma_n$, as one needs to ensure that distinct A-blocks represent distinct elements of {1,...,n}, and that the size of each A-block is in one of the permitted ranges.

To define the set of well-formed permutations that *do* represent permutations on {1,...,n}, we describe a useful shorthand notation. First note that once the size of each A block is known, the A's and B's themselves are no longer necessary. We can simply denote a well-formed permutation $\pi = A^{x_1}BA^{x_2}B \ldots BA^{x_i}B \ldots A^{x_{n-1}}BA^{x_n}$ by the list $x_1, x_2, \ldots, x_i, \ldots, x_{n-1}, x_n$. Call this a *size list* for $\pi$. The size list in which the sizes are arranged in nondecreasing order is referred to as the *canonical size list.* Now define a function s, that maps sizes of A-blocks into {1,...,n}. Let x be the size of an A-block. Define $s(x_i)$ by:

$$s(x_i) = x_i - (n * \lfloor (x_i - 1)/n \rfloor). \tag{2}$$

For a canonical size list $x_1, x_2, \ldots, x_n$, we get the corresponding list of integers $s(x_1)$, $s(x_2), \ldots s(x_n)$. If the integers in the corresponding list of integers are distinct elements of {1,...,n}, then $\pi$ represents a permutation, namely the permutation $\sigma$, where $\sigma_i = s(x_i)$. We call a well-formed permutation $\pi = A^{x_1}BA^{x_2}B \ldots BA^{x_i}B \ldots A^{x_{n-1}}BA^{x_n}$ *proper,* if it represents a permutation $\sigma$ on {1,...,n} in this way. Note that any reordering of A-blocks yields the same canonical size list, due to the distinct ranges of values.

Continuing the previous example, where $\pi = A^2BA^7BA^{18}BA^{22}BA^{29}BA^{33}$ represented the permutation $\sigma = 216453$ on {1, 2,...,6}, we get, for the block $A^{33}$ that s(33)=3. This tells us that the symbol "3" is represented by this A-block, and it occupies the 6th position in the corresponding permutation (according to equation 1). That is, $\sigma_6 = 3$. Note that $\pi = A^2BA^7BA^{18}BA^{22}BA^{29}BA^{33}$ and $\pi' = A^2BA^{33}BA^{18}BA^{29}BA^7BA^{22}$, both of which represent the same permutation $\sigma = 216453$, have the same canonical size list, namely, 2,7,18,22,29,33.

We now compute the dimension of the pancake network needed for the embedding. In order to maintain the required size ranges for the A-blocks, we need zero extra A's in the block that represents the first symbol, n extra A's in the block that represents the second symbol, and, more generally, (i-1)n extra A's in the block that represents the ith symbol. In addition, we need one A to represent the symbol "1", two A's to represent the symbol "2", and more generally, i A's to represent the ith symbol. Finally we need (n-1) B's to delimit the n A-blocks. This gives us a grand total of m symbols, where m is given by

$$m = \sum_{i=1}^{n}(i-1)n + \sum_{i=1}^{n} i + n - 1 = (n+1)\sum_{i=1}^{n} i - n^2 + n - 1$$
$$= \frac{n(n+1)^2}{2} - n^2 + n - 1 = \frac{1}{2}(n^3 + 3n - 2).$$

In Theorem 6 below, we complete our formal description of the dilation one embedding of stars into pancakes. In the proof of the theorem, one must show how to simulate a transposition in the star network with a prefix reversal on a proper permutation in the pancake network. Transpositions in the star network are of the form (1 i), for some i>1. Our construction ensures that the first A-block in any proper permutation representation of a permutation $\sigma$ on {1,...,n} represents the symbol $\sigma_1$. Thus, to simulate the transposition (1 i) one can move the entire contents of the first A-block into another A-block, *i.e.* whatever A-block represents $\sigma_i$. It follows that this reversal is possible only when the proper permutation $\pi = A^{x_1}BA^{x_2}B...BA^{x_i}B...A^{x_{n-1}}BA^{x_n}$ representing $\sigma$ has $x_1 < x_i$. So, our embedding will map $S_n$ into the set $R \subseteq P_m$, where R is defined by:

$R = \{\pi \mid \pi = A^{x_1}BA^{x_2}B...BA^{x_i}B...A^{x_{n-1}}BA^{x_n}$, and $x_1 < x_i$ for all i, $2 \le i \le n$, and $\pi$ is proper$\}$.

**Theorem 6:** $S_n \overset{\text{dil }1}{\Rightarrow} P_m$ , where $m = \frac{1}{2}(n^3 + 3n - 2)$.

**Proof:** Let $P_m$ be the pancake network of size m, and let R be the subset of $P_m$ defined above. Let $\pi$ be a permutation in R and let $x_1, x_2, ..., x_i, ..., x_{n-1}, x_n$ be its canonical size list. We define the co-embedding $g: R \to S_n$ by

$$g(\pi) = \begin{pmatrix} 1 & 2 & \dots & n \\ s(x_1) & s(x_2) & \dots & s(x_n) \end{pmatrix},$$

where the function s is defined in equation 2 above. (Note that since $\pi$ is proper, $g(\pi)$ is a permutation on {1,...,n}, as desired.)

A transposition of the form (1 i) in the star network can be simulated by a single pancake move. Suppose that $\pi$ represents a permutation $\sigma$ in the star network. To simulate the transposition (1 i), one simply locates the A-block whose size is in the range {(i-1)*n+1, ..., i*n}, and performs the prefix reversal that exchanges $\sigma_i$ symbols from this block, say block p, with the first block. That is, if we start with a proper permutation whose first A-block has size $\sigma_1$ and whose pth A-block has size $(i-1)*n+\sigma_i$, then after the appropriate prefix reversal, we would have a permutation $\pi'$ whose first A-block has size $\sigma_i$ and whose pth A-block has size $(i-1)*n+\sigma_1$. Although this reverses the order of the intervening A-blocks, the representation is still correct as the order of the A-blocks is irrelevant. With a small amount of algebra, it is easily confirmed that since $\pi$ is proper, $\pi'$ is also proper, and that the prefix reversal does indeed simulate the desired transposition.

We improve this result by taking advantage of a few special features of the embedding. First, observe that the first symbol, $\sigma_1$, of any permutation $\sigma$ in $S_n$ is always represented by the first A-block of a corresponding permutation in R. Since its location is known, the size range of the first A-block need not be disjoint from others. For example, one can make the size ranges for the first two blocks identical. Similarly, the last symbol, $\sigma_n$, is always represented by the last A-block of a corresponding permutation in R. So the last A-block need not have extra symbols because its size range need not be distinct either. That is, its range can coincide exactly with the ranges of $\sigma_1$ and $\sigma_2$. This means that the size ranges for the remaining A-blocks (the blocks representing the symbols $\sigma_3, ..., \sigma_{n-1}$) can be more economical. That is, n extra A's can be used in the block that represents $\sigma_3$, 2n extra A's can be used in the block that represents $\sigma_4$, and in general, (i-2)*n extra A's can be used in the block that represents $\sigma_i$, for all i, $1 \le i \le n$. These observations allow us to decrease the size of the set A by $2n^2-3n$ symbols.

With a bit more work, further improvements are possible. One can make the size ranges of consecutive *pairs* of A-blocks in the canonical size list overlap in exactly one value. That is, if the range of one block is {(i-2)n+1, ..., (i-1)n}, the range of the next one can be {(i-1)n, ..., in-1}. In other words, the largest valid value for the block representing $\sigma_i$ can coincide with the smallest valid value for the block representing $\sigma_{i+1}$. The appropriate values for $\sigma_i$ and $\sigma_{i+1}$ can still be correctly deduced. To see why, suppose that two adjacent sizes in the canonical size list for a permutation $\pi$ in R are equal. Then in the corresponding list of integers, two adjacent integers will be identical. The first of these integers represents $\sigma_i$, for some i, and has its maximum value. So it must certainly represent n. That is, $\sigma_i=n$. The second represents $\sigma_{i+1}$, and has its minimum value. So it must represent 1. That is, $\sigma_{i+1}=1$. So one has an unambiguous interpretation of the canonical size list that yields a permutation in $S_n$. Consider the simulation of the transposition (1 i). Now, the A-blocks representing $\sigma_i$ and $\sigma_{i+1}$ have the same size, so one doesn't know which of these A-blocks to change. Observe that one can choose *either* block, because whether one changes the size of the A-block representing $\sigma_i$ *or* the size of the A-block representing $\sigma_{i+1}$, the result is a permutation $\pi'$ whose canonical size list accurately represents the appropriate new permutation. This improvement allows us to 'delete' one symbol from each A-block (excluding the first two A-blocks and the last A-block), which adds up to a savings of n-3 A's. Finally, we can use the symbol set {0,...,n-1} instead of {1,...,n} for $S_n$, saving n additional A-symbols.

These observations allow us to decrease the size of the set A by $2n^2$-3n+ n-3+n = $2n^2$-n-3 symbols. Thus, the number of symbols in the host network is

$$m = \frac{1}{2}(n^3 + 3n - 2) - (2n^2 - n - 3) = \frac{1}{2}(n^3 - 4n^2 + 5n + 4) .$$

The improved results are summarized in the following theorem.

**Theorem 7:** $S_n \overset{\text{dil 1}}{\Rightarrow} P_m$ , where $m = \frac{1}{2}(n^3 - 4n^2 + 5n + 4)$.

## 4. Conclusion

This paper has presented several low dilation embeddings of star networks into pancake networks and vice-versa. In many cases, these embeddings require an expansion in the dimension of the host network. Do low dilation, one-to-one embeddings of the pancake network into the star network exist? We conjecture that any embedding of the pancake network into the star network of the *same* dimension has *unbounded* dilation. We further conjecture that any embedding of one network into the other with dilation less than 4 comes at the cost of increased dimensionality of the host network. We know of no proof of these statements. We leave such issues open for future study.

## References

1. S. Akers and B. Krishnamurthy, "A group-theoretic model for symmetric interconnection networks," *IEEE Trans. Comput.,* vol. C-38, no.4, pp. 555-566, 1989.
2. A. Bouabdallah, M. C. Heydemann, J. Opatrny, and D. Sotteau, "Embedding Complete Binary Trees into Star and Pancake Graphs," *Theory of Computer Systems*, v. 31, pp. 279-305, 1998
3. D. S. Cohen and M. Blum, "Improved bounds for sorting pancakes under a conjecture," *Discrete Applied Mathematics*, 1993
4. V. Faber, J.W. Moore, and W. Y. C. Chen, "Cycle prefix digraphs for symmetric interconnection networks," *Networks,* vol. 23, John Wiley and Sons, 1993, pp. 641-649.
5. L. Morales, Z. Miller, D. Pritikin, and I. H. Sudborough, "One-to-Many Embeddings of Hypercubes into Cayley Graphs Generated by Reversals", *Theory of Computing Systems,* 34, 2001, pp. 399-431.
6. W. H. Gates and C. H. Papadimitriou, "Bounds for sorting by prefix reversal," *Discrete Math.,* vol. 27, pp.47-57, 1979.
7. M. H. Heydari and I. H. Sudborough, "On sorting by prefix reversals and the diameter of pancake networks," *J. Algorithms*, October, 1997.
8. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann Publishers, 1992.
9. Z. Miller, D. Pritikin, and I. H. Sudborough, "Bounded dilation maps of hypercubes into Cayley graphs on the symmetric group," *Math Systems Theory* 29 (1996), pp. 551-572

10. Z. Miller, D. Pritikin, and I. H. Sudborough, "Near embeddings of hypercubes into Cayley graphs on the symmetric group," *IEEE Trans. Comput.,* vol. 43, no.1, pp. 13-22, 1994.
11. M. Nigam, S. Sahni, and B. Krishnamurthy, "Embedding hamiltonians and hypercubes in star interconnection graphs," in *Proc. Int. Conf Parallel Processing,* vol. 1990.

# Synchronization of Finite Automata: Contributions to an Old Problem

Arto Salomaa

Turku Centre for Computer Science
Lemminkäisenkatu 14A
20520 Turku, Finland
asalomaa@utu.fi

**Abstract.** In spite of its simple formulation, the problem about the *synchronization* of a finite deterministic automaton is not yet properly understood. The present paper investigates this and related problems within the general framework of a composition theory for functions over a finite domain $N$ with $n$ elements. The notion of *depth* introduced in this connection is a good indication of the *complexity* of a given function, namely, the complexity with respect to the length of composition sequences in terms of functions belonging to a basic set. Our results show that the depth may vary considerably with the target function. We also establish criteria about the reachability of some target functions, notably constants. Properties of $n$ such as primality or being a power of 2 turn out to be important, independently of the semantic interpretation. Most of the questions about depth, as well as about the comparison of different notions of depth, remain open. Our results show that the study of functions of several variables may shed light also to the case where all functions considered are unary.

## 1 Introduction

We will consider an old unsolved problem about finite deterministic automata in a more general setup. The generalization shows the close interconnection of the original problem with issues in many-valued logic, graph theory and combinatorics. Moreover, the notion of *depth* introduced in this connection is a good indication of the *complexity* of a given function, namely, the complexity with respect to the length of composition sequences in terms of functions belonging to a basic set.

The old problem we are concerned with deals with the *synchronization* of finite deterministic automata. It falls within the framework of the classical "Gedanken experiments" by Moore, [9]. We have some information about an automaton but basically the latter remains a "black box" for us. We want to learn more by feeding in some input words and making conclusions based on the behavior of the automaton.

To come to our problem, suppose you know the structure (graph, transition function, transition table) of a given finite deterministic automaton $\mathcal{A}$, but do not know the state $\mathcal{A}$ is in. How can you get the situation under control? For some automata, not always, there are words, referred to as *synchronizing*, bringing the automaton always to the same state $q$, no matter from which state you started from. Thus, you first have to feed $\mathcal{A}$ a synchronizing word, after which you have the situation completely under control. Automata possessing a synchronizing word are referred to as *synchronizable.*

If $w$ is a synchronizing word, then so is $xwy$, for any words $x$ and $y$. This is very obvious as regards $y$: if $w$ merges all states to the same state, so does every continuation of $w$. But it holds also as regards $x$: since $w$ merges *all* states to the same state, it does so also for the *subset* of states where the automaton can be after reading $x$. Thus, every synchronizable automaton possesses arbitrarily long synchronizing words. But suppose we are looking for the *shortest* synchronizing word - a very natural approach. How long is it in the worst case?

J. Černý conjectured in [3] in 1964 that in a synchronizable automaton the length of the shortest synchronizing word never exceeds $(n-1)^2$, where $n$ is the cardinality of the state set. The conjecture remained rather unknown in the early years, although it was discussed in some papers (see [4] and its references). It became known through the many important contributions of J.-E. Pin (for instance, see [12]) around 1980. During very recent years, there has been a considerable interest around this topic, for instance, see [5,6,7] or the surveys [8,2].

The Conjecture of Černý is still open. We also do not know any characterization of synchronizable automata. Both problems belong to the general framework of the composition theory of functions over a finite domain. The state set of our automaton is a finite set, and each input letter defines (via the transition mapping) a function of this finite set into itself. Thus, our automaton defines a *basic* set of functions, and we can consider functions generated by the basic set under composition. (Clearly, in the automaton, compositions correspond to input *words.*) The automaton is synchronizable iff a *constant* is generated by the basic set. We can also view constants as *targets* and search for the shortest composition sequence for a target.

The above general framework can be applied also to the truth-functions of *many-valued logic,* [13,14]. Many combinatorial problems fall also within the same framework. In the *road coloring problem* one is given a finite directed graph $G$, where the edges are unlabeled. The task is to find a labeling of the edges that turns $G$ into a deterministic automaton possessing a synchronizing word, [1]. (Clearly, a condition necessary for making $G$ an automaton is that all vertices have the same outdegree.) The term "road coloring" is due to the following interpretation. A traveler lost in the graph $G$ can always find a way back home regardless of where he/she actually started, provided he/she follows a sequence of labels (colors) constituting a synchronizing word.

We can still summarize the motivation for studying the original conjecture as follows.

– It is an old unsolved problem about the very basics of finite automata.
– The problem fits in the framework of compositions of functions over a finite domain.
– This is a very basic class of functions, coming up in automata, many-valued logic, combinatorics, ...
– Some parts of the theory of such functions are well understood, for instance, *completeness*, some others not *(minimal length of composition sequences).*
– The problem is closely related to various other problems (*road coloring, experiments with automata, structural theory of many-valued truth functions*).

A brief description about the contents of this paper follows. Although the reader is supposed to know the very basics of automata theory, no previous knowledge about the actual subject matter is needed. The next section gives the basic definitions, in particular, the fundamental notion of *depth*. Also some rather general results are established. Section 3 discusses some provably hard cases. The notions of *completeness* and *synchronizability* will be investigated in Section 4. Functions of *several variables*, also with respect to completeness, are investigated in Section 5, whereas the notion of depth is extended to concern functions of several variables in Section 6. The statement corresponding to the Černý Conjecture does not hold in this more general setup.

## 2 Basic Results about Unary Depth

In the following formal definition we introduce a finite deterministic automaton as a triple, without specified initial or final states. This was customary in the early days of the theory and suits very well for our present purposes.

**Definition 1** *Consider a finite deterministic automaton $\mathcal{A} = (Q, \Sigma, \delta)$, where $Q$ and $\Sigma$ are finite nonempty disjoint sets (states and input letters), and $\delta$ (the transition function) is a mapping of $Q \times \Sigma$ into $Q$. The domain of $\delta$ is extended in the usual way to $Q \times \Sigma^*$. If $Q_1 \subseteq Q$ and $w$ is a word over $\Sigma$, we define $\delta(Q_1, w) = \{q' | \delta(q, w) = q', q \in Q_1\}$. A word $w$ is* synchronizing *(for $\mathcal{A}$) if $\delta(Q, w)$ is a singleton set. The automaton $\mathcal{A}$ is* synchronizable *if it possesses a synchronizing word.*

We will consider in this paper functions $g(x)$ whose domain is a fixed finite set $N$ with $n$ elements, $n \geq 2$, and whose range is included in $N$. It is clear that such a setup occurs in many and very diverse situations, interpretations. Depending on the interpretation, different questions can be asked. The two interpretations we have had mostly in mind are *many-valued logic* and *finite automata.* In the former, the set $N$ consists of $n$ *truth values* and the functions are truth functions. In the latter, the set $N$ consists of the *states* of a finite automaton, whereas each letter of the input alphabet induces a specific function: the next state when reading that letter.

To start with, our attention is restricted to functions with *one* variable only. In many contexts, especially in many-valued logic, it is natural to consider functions of *several* variables. We will return to this in Sections 5 and 6.

We make the following **convention**, valid throughout the paper: $n$ always stands for the number of elements in the basic set $N$. In most cases we let $N$ simply be the set consisting of the first $n$ natural numbers:

$$N = \{1, 2, \ldots, n\}.$$

Clearly, there are altogether $n^n$ *unary* functions, that is, functions of one variable.

Consider a couple of examples. If we are dealing with $n$-valued logic, and the function $g$ is defined by the equation

$$g(x) = n - x + 1, x = 1, 2, \ldots, n,$$

then $g$ is the well-known *Łukasiewicz negation.* (1 is the truth value "true", $n$ is the truth value "false", whereas the other numbers represent the intermediate truth values.) If we are dealing with a finite deterministic automaton whose state set equals $N$, the function $g$ defined by the equation above could be viewed as transitions affected by a specific input letter $a$. Under this interpretation, the letter $a$ interchanges the states $n$ and 1, the states $n-1$ and 2, and so forth. Whether or not there is a *loop* affected by the letter $a$, that is, whether or not some state is mapped into itself, depends on the parity of $n$.

When we speak of "functions", without further specifications, we always mean functions in the setup defined above. Clearly, the *composition ab* of two functions $a$ and $b$ is again a function.

Our point of departure will be a nonempty set $\mathbf{F}$ of functions. The only assumption about the set $\mathbf{F}$ is that it is a nonempty subset of the set $N^N$ of all functions; $\mathbf{F}$ may consist of one function or of all functions. We will consider the set $\mathbf{G(F)}$ of all functions *generated* by $\mathbf{F}$, that is, obtained as compositions (with arbitrarily many composition factors) of functions from $\mathbf{F}$. If a particular function $f$ can be expressed as a composition of functions $a_i, i = 1, 2, \ldots, k$, belonging to $\mathbf{F}$:

$$f = a_1 a_2 \ldots a_k,$$

where some of the functions $a_i$ may coincide, then the word $a_1 a_2 \ldots a_k$ is referred to as a *composition sequence* for $f$. (In this brief notation we assume that the set $\mathbf{F}$ is understood.) The number $k$ is referred to as the *length* of the composition sequence. The function $f$ is often referred to as the *target function.* Observe that our composition sequences have to be nonempty, implying that the identity function is not necessarily in $\mathbf{G(F)}$; it is in there exactly in case the set $\mathbf{F}$ contains at least one permutation.

Clearly, $\mathbf{G(F)}$ can be viewed as the *semigroup* generated by $\mathbf{F}$. However, we will prefer the more straightforward approach and will not use semigroup-theoretic terminology in the sequel.

The set $\mathbf{F}$ is termed *complete* if all of the $n^n$ functions are in $\mathbf{G(F)}$. Completeness in this setup is fairly well understood, see [14,15,16,17]. On the other hand, fairly little is known about the length of composition sequences and about the question concerning how completeness affects the length. When do two permutations generate the whole symmetric group $S_n$? (At least two are needed for

$n \geq 3$.) Quite much is known about this problem, [11], but very little about the lengths of the corresponding composition sequences.

Since $n$ is finite, a specific function $f$ can always be defined by a table. Omitting the argument values, this amounts to giving the *value sequence* of $f$, that is, the sequence $f(1),\ f(2), \ldots,\ f(n)$ of its values for the increasing values of the argument. The Łukasiewicz negation can be defined in this way by its value sequence

$$n,\ n-1, \ldots, 2,\ 1.$$

When there is no danger of confusion, we omit the commas from the value sequence. Thus, for $n = 6$, the value sequence of the Łukasiewicz negation reads 654321.

We denote by $L(\mathbf{F}, f)$ the set of all composition sequences for $f$, that is, the *language* over the alphabet $\mathbf{F}$ whose words, viewed as composition sequences, yield the function $f$. (Depending on the context, we will read composition sequences from left to right or from right to left. Whereas the latter way is customary in algebra, the former way corresponds to the usual reading of words in automata theory.) Clearly, the language $L(\mathbf{F}, f)$ can be empty (this is the case when $f$ cannot be expressed as a composition of functions in $\mathbf{F}$) or infinite (composition sequences may contain redundant parts and be arbitrarily long). Clearly, this language is always regular.

We now come to the central notions concerning the length of composition sequences. For any language $L$, we denote by $\min(L)$ the length of the shortest word in $L$. (If $L$ is empty, we agree that $\min(L) = \infty$.)

**Definition 2** *The* depth *of a function $f$* with respect *to the set $\mathbf{F}$, in symbols $D'(\mathbf{F}, f)$, is defined by the equation*

$$D'(\mathbf{F}, f) = \min(L(\mathbf{F}, f)).$$

*Thus, the depth of a function with respect to a particular set can also be $\infty$.*

*The* depth of a function *$f$ is defined by the equation*

$$D'(f) = \max(D'(\mathbf{F}, f)),$$

*where $\mathbf{F}$ ranges over all sets with the property $L(\mathbf{F}, f) \neq \emptyset$.*

Because, for any $f$, there are sets $\mathbf{F}$ with the property mentioned in the definition, we conclude that the depth of a function is always a positive integer. (The notion of depth was introduced in [8], where it was referred to as "complexity".)

**Definition 3** *The* complete depth *$D'_C(f)$ of a function $f$ is defined by the equation*

$$D'_C(f) = \max(D'(\mathbf{F}, f))$$

*where now $\mathbf{F}$ ranges over* complete *sets of functions.*

In the latter definition it is *a priori* clear that $L(\mathbf{F}, f) \neq \emptyset$.

It follows by the definitions that every function $f$ satisfies

$$D'_C(f) \leq D'(f).$$

However, lower bounds are much harder to obtain for $D'_C(f)$, for the simple reason that we have much less leeway if we have to restrict the attention to complete sets $\mathbf{F}$ only. We have used the primed notation $D'$ to indicate that we are, in these definitions, dealing with *unary* functions. The unprimed notation $D$ is reserved to the general case, where functions with arbitrarily many variables are considered.

Among the unary functions, we are especially interested in the *constants*

$$c_i(x) = i, \text{ for all } x \text{ and } i = 1, 2, \ldots, n.$$

We use the notation SYNCHRO for the class of all sets $\mathbf{F}$ such that at least one of the constants $c_i$ is in $\mathbf{G(F)}$. (In defining SYNCHRO we have some fixed $n$ in mind. Thus, SYNCHRO actually depends on $n$.) Analogously to the definitions above, we now define the depths

$$D'(\text{const}),\ D'_C(\text{const}).$$

By definition,

$$D'(\text{const}) = \max_{\mathbf{F}} \min\{D'(\mathbf{F}, c_i) | 1 \leq i \leq n\},$$

where $\mathbf{F}$ ranges over SYNCHRO. The depth $D'_C(\text{const})$ is defined in exactly the same way, except that $\mathbf{F}$ ranges over complete sets. Thus, the notions introduced do not deal with the depth of an individual function but rather give the smallest depth of a constant, among the constants generated. Similarly as SYNCHRO, the notions depend on $n$.

The Černý Conjecture can now be expressed in the following form.

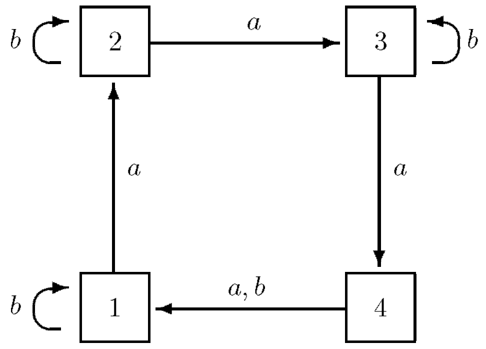
**Conjecture 1 (Černý)** $D'(\text{const}) = (n-1)^2$.

We now present some facts related to Conjecture 1. The first question is: Could one do better, that is, obtain a still smaller depth? The answer is negative, as shown by the following example.

Consider the finite deterministic automaton $\mathcal{A}'$ defined as follows. The automaton $\mathcal{A}'$ has the state set $N$, and two input letters $a$ (affects a circular permutation) and $b$ (identity except sends $n$ to 1). The transitions are defined by the table

| $\delta$ | $1\ 2 \ldots n-1\ n$ |
|---|---|
| $a$ | $2\ 3 \ldots\ \ n\ \ \ 1$ |
| $b$ | $1\ 2 \ldots n-1\ 1$ |

We still depict the automaton for $n = 4$.



The word $(ba^{n-1})^{n-2}b$ is synchronizing and, moreover, there are no shorter synchronizing words and this word is the only synchronizing word of length $(n-1)^2$. The reader might want to prove this fact. The first impression is that there could be shorter synchronizing words. Indeed, there are many possibilities of applying a "greedy" algorithm, that is, using the "reduction" $b$ earlier than after $n-1$ uses of the circular $a$. (Clearly, $b$ has to be used in the reduction sense altogether $n-1$ times.) But if one does so, one has to pay a price later on, which makes the indicated synchronizing word shortest.

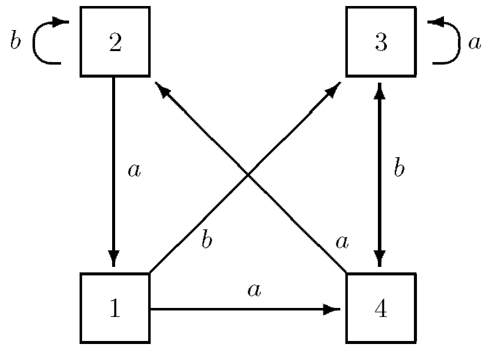
Let $\mathbf{F}$ consist of the functions $a$ and $b$ defining this particular automaton $\mathcal{A}'$, and consider the constant functions $c_i$, $1 \le i \le n$.

**Lemma 1** $D'(\mathbf{F}, c_i) = (n-1)^2 + i - 1$, *for* $i = 1, 2, \ldots, n$.

See [17] for a proof of the lemma. The lemma yields the lower bounds in the following theorem. The reference [17] contains also a proof of the upper bound; various cubic upper bounds are known in the literature for different setups. The upper bound $2^n - n - 1$ (agreeing with $(n-1)^2$ for $n = 2, 3$) follows directly by the definition: if a synchronizing word $w$ can be decomposed as $w = xyz$, where $\delta(Q, x) = \delta(Q, xy)$, then also the word $xz$ is synchronizing. We will discuss this in more detail in Section 3 in connection of *reduced* synchronizing words.

**Theorem 1** *For a constant* $c_i$, *we have* $n(n-1) \le D'(c_i) < n^3/2$. *Moreover,* $D'(\text{const}) \ge (n-1)^2$.

Thus, Conjecture 1 is actually open only as regards the upper bound. On the other hand, the automaton defined above is the only automaton (when isomorphic variants are disregarded) known, which is defined for an arbitrary $n$ and which reaches the lower bound. For $n = 3$, there are many automata reaching the lower bound 4. For $n = 4$, also the following automaton due to [4] reaches the lower bound 9:



Thus, $(n-1)^2$ seems to be a rare exception for the length of the minimal synchronizing word for a finite deterministic automaton, the length being "normally" smaller. This supports Conjecture 1. On the other hand, we will see that if functions of several variables are considered, then the statement analogous to Conjecture 1 is not valid.

The variant of Conjecture 1, dealing with the depths of individual functions, can be expressed as follows.

**Conjecture 2** *If $c_i$ is a constant, then $D'(c_i) = n(n-1)$.*

It is a consequence of Theorem 1 that the depth $n(n-1)$ cannot be reduced. If Conjecure 1 holds, so does Conjecture 2. This is a consequence of the following simple result.

**Lemma 2** *For a constant $c_i$, we have $D'(c_i) \leq D'(\text{const}) + (n-1)$.*

On the other hand, it is conceivable (although very unlikely) that Conjecture 2 holds but Conjecture 1 fails.

It is not known whether the upper bound ($(n-1)^2$ or $n(n-1)$) is reached for the complete depth. Thus, it is possible that

$$D'_C(\text{const}) < (n-1)^2,$$

and $D'_C(c_i) < n(n-1)$ holds for the constant $c_i$.

**Conjecture 3** $D'_C(\text{const}) \leq (n-1)^2 - (n-3)$. *Moreover, the complete depth of a constant $c_i$ satisfies*

$$D'_C(c_i) \leq (n-1)^2 + 2.$$

If true, Conjecture 3 gives also an example of a function whose complete depth is strictly smaller than its depth. Various claims have been made in the literature about the validity of Conjecture 1 for "small" values of $n$. The Conjecture clearly holds for $n = 2, 3$ because, as we already pointed out, in this case the bound $(n-1)^2$ coincides with the bound obtained from the number of all subsets. Really convincing proofs are missing from the other cases. The number of possibilities is huge. For $n = 5$, one has to go through $2^{5^5}$ subcases.

## 3 Exponential Lower Bounds and Np-hard Problems

As seen above, the bound in Conjecture 1 holds as a lower bound, whereas also a cubic (but no quadratic) upper bound is known. We will present in this section results about related, provably intractable problems. We will discuss the depth of functions other than constants, as well as the problem of determining whether or not a specific function is in $\mathbf{G(F)}$. Finally, we present some observations about the length of reduced synchronising words. No polynomial upper bounds are possible in any of these cases.

We show first that there are specific functions and classes of functions whose depth cannot be bounded from above by a polynomial in $n$. We give the basic definition in automata-theoretic terms. Coming back to Gedanken experiments, [9], we might sometimes want to keep the black box as it was, unchanged. This leads to Definition 4. A *stabilizing* word might give some information about the behavior of the automaton, especially if the latter has outputs, but in any case we do not lose anything since we always return to the starting point.

**Definition 4** *A nonempty word $w$ over the alphabet $\Sigma$ of a finite deterministic automaton $(Q, \Sigma, \delta)$ is* stabilizing *if $\delta(q, w) = q$ holds for all states $q$. Similarly, $w$ is* transposing *if there are two distinct states $q_1$ and $q_2$ such that*

$$\delta(q_1, w) = q_2, \ \ \delta(q_2, w) = q_1, \ \ \delta(q, w) = q \text{ for } q \neq q_1, q_2.$$

Observe that, from the point of view of functions over the finite domain $N = Q$, a stabilizing word defines the *identity* function, whereas transposing words correspond to a *class* of functions. Analogously to the notation SYNCHRO, we use the notations STABIL and TRANSP. Thus, TRANSP stands for the class of all sets $\mathbf{F}$ such that at least one of the transposing functions is in $\mathbf{G(F)}$. The depths

$$D'(\text{stabil}), \ \ D'_C(\text{stabil}), \ \ D'(\text{transp}), \ \ D'_C(\text{transp})$$

are defined in the same way as for SYNCHRO and const. For instance,

$$D'(\text{transp}) = \max_{\mathbf{F}} \min\{D'(\mathbf{F}, t) | t \text{ is transposing}\},$$

where $\mathbf{F}$ ranges over TRANSP. All of these notions depend on $n$. Whenever important, we will indicate this in the notation.

**Theorem 2** *There is no polynomial $P(n)$ such that $D'(\text{stabil}(n)) \leq P(n)$, for all $n$. There is no polynomial $P(n)$ such that $D'(\text{transp}(n)) \leq P(n)$, for all $n$.*

*Proof.* It suffices to consider an infinite sequence of numbers $n$ of a specific form. Consider first the case of STABIL. Let $p_i$ be the $i$th prime, and consider numbers $n$ of the form

$$n = p_1 + p_2 + \cdots + p_k.$$

Let $a$ be a permutation in the symmetric group $S_n$, defined as the product of $k$ cycles of lengths $p_1, p_2, \ldots, p_k$. Let the set $\mathbf{F}$ consist of $a$ only. Let the target function $id$ be the identity function. Clearly,

$$D'(\mathbf{F}, id) = p_1 p_2 \cdots p_k = \Pi,$$

where the last equality is only a notation. By the well-known estimate $p_k \leq k^2$, we obtain $n \leq kp_k \leq k^3$, whence $k \geq \sqrt[3]{n}$. Since obviously $\Pi \geq k!$, we obtain finally

$$D'(id) \geq [\sqrt[3]{n}]!$$

which shows that $D'(\text{stabil}(n))$ cannot have any upper bound $P(n)$. (Observe that the situation described above can be viewed as an automaton with only one input letter $a$. A similar technique has often been used in connection with trade-offs between deterministic and nondeterministic finite automata. The automaton is disconnected: the states form cycles with prime lengths. However, the automaton can easily be replaced by a connected one by adding a dummy letter not affecting the shortest stabilizing word.)

The case of TRANSP is handled in almost the same way. We just omit the first prime $p_1 = 2$ from the product $\Pi$, obtaining the product $\Pi'$. Since $\Pi'$ is odd, the word $w' = a^{\Pi'}$ interchanges the two states in the first cycle. For this particular automaton, the word $w'$ is the shortest transposing word. The above argument is now applicable for upper bounds of $D'(\text{transp}(n))$, because division by 2 does not affect any of the conclusions.

Corresponding questions about the complete depth, such as whether or not $D'_C(\text{transp}(n))$ possesses a polynomial upper bound, remain open. If one has to deal with functionally complete automata, lower bounds such as the ones in the above proof are hard to obtain.

Due to finite upper bounds, all reasonable problems are *decidable* in our setup. For instance, given a function $f$ and a set set $\mathbf{F}$, we can decide whether or not $f$ is in $\mathbf{G(F)}$. This follows because of the trivial upper bound $n^n$ for the length of minimal composition sequences. We can also test, by trying out all of the finitely many possibilities, whether or not a given composition sequence for $f$ is minimal. On the other hand many such problems, notably the ones mentioned above, are *NP-hard*. The reader is referred to [17] for results along these lines. Here we just point out, in terms of an example, a very useful technique for establishing intractability in setups like ours. The technique uses reduction to the satisfiability problem.

Consider the following propositional formula $\alpha$ in 3-conjunctive normal form:

$$\begin{aligned}\alpha \;=\; & (\sim x_1 \vee \sim x_2 \vee \sim x_3) \wedge (\sim x_1 \vee x_2 \vee \sim x_3) \wedge (x_1 \vee x_2 \vee \sim x_4) \\ & \wedge(\sim x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \sim x_5) \wedge (x_1 \vee \sim x_2 \vee \sim x_5) \\ & \wedge(x_1 \vee x_3 \vee x_4) \wedge (\sim x_1 \vee x_3 \vee \sim x_5) \wedge (x_1 \vee \sim x_4 \vee x_5) \\ & \wedge(x_2 \vee \sim x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_3 \vee \sim x_4 \vee x_5).\end{aligned}$$

Thus, there are 5 variables and altogether 12 clauses. To get $n$, we take the product of these numbers added by 2, yielding $n = 62$.

The basic set **F** consists of two functions $a$ and $b$ defined by the following array in the way described below.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $1b$ | $2b$ | $3a$ | $4b$ | $5a$ | $6a$ | $7a$ | $8b$ | $9a$ | 10 | 11 | 12 |
| $13b$ | $14a$ | $15a$ | $16a$ | $17a$ | $18b$ | 19 | 20 | 21 | $22a$ | 23 | 24 |
| $25b$ | $26b$ | 27 | 28 | 29 | 30 | $31a$ | $32a$ | 33 | $34b$ | $35a$ | $36a$ |
| 37 | 38 | $39b$ | $40a$ | 41 | 42 | $43a$ | 44 | $45b$ | $46a$ | $47a$ | $48b$ |
| 49 | 50 | 51 | 52 | $53b$ | $54b$ | 55 | $56b$ | $57a$ | 58 | $59a$ | $60a$ |

Each number in the array is mapped by both $a$ and $b$ to the next number in the same column, whereas numbers in the last row are mapped (by both $a$ and $b$) to the number 61 (not shown in the array). *Exceptions* to this general rule, marked in the array by $a$ or $b$, indicate that the number in question is mapped to 62, by $a$ or $b$ as marked. Thus, 26 is mapped by $b$ and 47 by $a$ to 62. Finally, both $a$ and $b$ map both 61 and 62 to itself.

Consider now the following membership problem. Does **G(F)** contain a function mapping each of the numbers from 1 to 12 to the number 62? The answer is positive exactly in case $\alpha$ is satisfiable. This follows by the way the array was constructed: $a$ (resp. $b$) appears in the position $(i, j)$ if and only if the variable $x_i$ appears unnegated (resp. negated) in the $j$th clause of $\alpha$. In this particular example, $\alpha$ turns out to be satisfiable, and the word *baabb* defines a function in **G(F)** satisfying the required condition. As already pointed out, this technique is applicable to various similar problems. For instance, one can show, [17,6], that the problem of deciding, whether a given synchronizing word is shortest, is NP-hard.

Our starting point was *short* synchronizing words. Since a synchronizing word remains synchronizing if arbitrary prefixes or suffixes are added, we can construct arbitrarily long synchronizing words. Therefore, it is natural to consider synchronized words *reduced* in some sense. The following definition is very suitable in this context.

**Definition 5** *Assume that* **F** *is in* SYNCHRO *and $w$ is a composition sequence for a constant. Then $w$ is termed* reduced *if it cannot be written as $w = xyz$, where $y$ is nonempty and the ranges of the functions defined by $x$ and $xy$ coincide.*

It is clear that, in the situation described in the definition, also $xz$ is a composition sequence for the same constant. Since the target is a constant, we are only interested in the sequence of ranges obtained by reading through prefixes of various lengths. For instance, in the case of the exceptional automaton depicted in Section 2, the shortest synchronizing word *baababaab* yields the sequence of (distinct) ranges

$$1234,\ 234,\ 123,\ 134,\ 34,\ 23,\ 24,\ 12,\ 14,\ 3.$$

Words "reduced" according to the definition above were referred to as "range-reduced" in [17]. Observe that if the target function is not a constant then, in general, the information provided by the sequence of ranges is not sufficient and $xz$ is not any more a composition sequence for the target.

If we have enough input letters (enough functions in the set $\mathbf{F}$), we can construct reduced synchronized words of the "full" lenght $2^n - 2$. (We can go through all the nonempty subsets of $N$ in any cardinality-decreasing order.) We hope to return to a detailed discussion of reduced synchronized words in another context. It turns out that, even for two letters (two basic functions), no polynomial upper bound can be given for the length of reduced synchronized words. The basic idea is to go through subsets of half the size of $n$. As an example, we consider the automaton $\mathcal{A}'$ discussed in Section 2, for the specific value $n = 8$. We consider subsets of cardinality 4 obtainable by $a$ and $b$, starting from 1234. (Since there is no danger of confusion, we denote the subset $\{1, 2, 3, 4\}$ in this simple fashion. This particular subset results after reading the word $ba^7ba^7ba^7b$.) The following array should be read by rows. We have indicated only the positions where $b$ is applied; $a$ is applied in the remaining positions. Observe that, even in this very simple setup, 51 of the 70 subsets are obtained. Moreover, the length of the subword of the synchronizing word, resulting by going through subsets of cardinality $n/2$, already exceeds the bound $(n-1)^2$.

| | | | | |
|---|---|---|---|---|
| 1234 | 2345 | 3456 | 4567 | 5678*b* |
| 1567 | 2678 | 1378 | 1248 | 1235 |
| 2346 | 3457 | 4568*b* | 1456 | 2567 |
| 3678 | 1478 | 1258 | 1236 | 2347 |
| 3458*b* | 1345 | 2456 | 3567 | 4678*b* |
| 1467 | 2578 | 1368 | 1247 | 2358 |
| 1346 | 2457 | 3568*b* | 1356 | 2467 |
| 3578 | 1468 | 1257 | 2368 | 1347 |
| 2458*b* | 1245 | 2356 | 3467 | 4578*b* |
| 1457 | 2568*b* | 1256 | 2367 | 3478 |
| 1458 | 145 | | | |

## 4 Completeness and Synchronizability

In functional constructions such as ours, classifications of functions are useful in various connections. We now introduce the simple notions of the *genus* and *type* of a function, similarly as in [14]. A function $f$ is said to be of *genus* $t$ if it assumes exactly $t$ values. Thus, the genus equals the cardinality of the range of $f$. A function $f$ of genus $t$ is said to be of *type* $m_1 \oplus m_2 \oplus \ldots \oplus m_t$, where $m_1+m_2+\ldots+m_t = n$ if, for each $i$ with $1 \leq i \leq t$, there is a number $y_i$ such that $f$ assumes $y_i$ as a value exactly $m_i$ times. Obviously we do not change the type if we change the order of the numbers $m_i$, which means that the operator $\oplus$ is commutative. For instance, permutations are of genus $n$ and of type $1 \oplus 1 \oplus \ldots \oplus 1$, whereas constants are of genus 1 and type $n$. The type of a function $f$ tells us how many values $f$ assumes and how many times it assumes each value. It does not tell us what these values are and in what order they are assumed.

We now present two results useful in many constructions. We use the simple expression "a set **F** of functions *generates* another set **F'** of functions" to mean that **F'** is contained in **G(F)**.

**Lemma 3** *Assume that $n \geq 4$ and $f$ is of genus $< n$. Then the set consisting of $f$ and of the members of the alternating group $A_n$ generates every function of the same type as $f$.*

**Lemma 4** *The set of all functions of type $m_1 \oplus m_2 \oplus \ldots \oplus m_t$, where $1 < t < n$, generates every function of type $(m_1 + m_2) \oplus \ldots \oplus m_t$.*

The proofs of the two lemmas are not difficult. A minor difficulty in the former is to use only *even* permutations in the construction. The lemmas indicate the possibilities for constructing new functions if adequate permutations are available. For instance, by Lemma 4, all constants are generated by the set given in Lemma 3.

It is often possible to show that a composition sequence cannot any more be continued to yield a given target function. Thus, in many cases, a wrong choice at the beginning destroys the possibilities of obtaining the target function. For instance, if the numbers in the type of a composition sequence are all even, then the sequence cannot be continued to yield a target function whose type contains an odd number. Similarly, a composition sequence of type $3 \oplus 2$ cannot be continued to yield a target function of type $4 \oplus 1$. On the other hand, if the target function is a *constant*, then the situation is entirely different: wrong choices may be corrected later on. In other words, if a composition sequence for a constant is at all possible, a composition sequence can also be obtained by continuing an arbitrary prefix. Constants are the only functions having this property. This is obvious also in view of the following lemma, which contains some simple observations along these lines.

**Lemma 5** *The genus of the conposition $fg$ exceeds neither the genus of $f$ nor the genus of $g$. Assume that $f$ is of type $m_1 \oplus \ldots \oplus m_t$. Then the type of the composition $((x)f)g$ is obtained by addition: each number in the latter type is the sum of some of the numbers $m_i$, whereby each of the numbers $m_i$ is used exactly once.*

For the proof of the following main *completeness theorem for unary functions*, we refer to [17].

**Theorem 3** *Given a nonidentical function $f$ of genus $n$ and a function $g$ of genus $n-1$, a function $h$ can be effectively constructed such that the set $\{f, g, h\}$ is complete, provided it is not the case that $n = 4$ and $f$ is one of the three permutations (12)(34), (13)(24) and (14)(23). For $n \geq 3$, no less than three functions suffice to generate all functions and, whenever three functions form a complete set, then two of them constitute a basis of the symmetric group $S_n$ and the third is of genus $n - 1$.*

An essential tool in the proof is a result by Piccard (see [11], pp.80-86), according to which a companion can be found to any nonidentical permutation such that the two permutations constitute a basis of the symmetric group $S_n$. The exceptional case here is $n = 4$: no permutation in the Klein Four-Group can be extended to a basis of $S_4$.

In the remainder of this section, we discuss the problem of *synchronizability*. No good characterization is known for synchronizability, that is, for determining of a given finite deterministic automaton whether or not it is synchronizable. An equivalent formulation for the problem is to ask whether or not a given set **F** of functions is in SYNCHRO. Since a set consisting of permutations only can never be in SYNCHRO, we denote by PRESYNCHRO the collection of all sets **F** containing at least one function of genus less than $n$. Thus the problem consists of finding a characterization for those sets in PRESYNCHRO that are in SYNCHRO.

It is an immediate consequence of Lemmas 3 and 4 that, whenever **F** is in PRESYNCHRO and contains the alternating group, then **F** is in SYNCHRO. The following result is somewhat stronger.

**Theorem 4** *If a set in* PRESYNCHRO *contains a doubly transitive group, then it is in* SYNCHRO.

*Proof.* Let **F** be the given set and $g$ a function in **F** of genus $< n$. Hence, there are two distinct numbers $i$ and $j$ such that $g(i) = g(j)$. Consequently, whenever $f$ is a function whose range contains the numbers $i$ and $j$, then $fg$ is of smaller genus than $f$. (We read here the composition from left to right: first $f$, then $g$.) Suppose we have a composition sequence $w$ for a function of genus $t > 1$. We take two arbitrary numbers $i_1$ and $j_1$ from the range of this function, as well as a permutation $h$ from the doubly transitive group mapping the pair $(i_1, j_1)$ onto the pair $(i, j)$. Now the composition sequence $whg$ defines a function of genus $< t$. Continuing in the same way, a constant (genus 1) is obtained, which concludes the proof.

Observe that the proof does not directly yield any estimates for the *length* of the composition sequence for a constant, because the doubly transitive group might be given in terms of some generators and the required permutations $h$ might have long composition sequences.

The above result cannot be extended to concern (simply) transitive groups. Indeed, it is not necessarily the case that a set belonging to PRESYNCHRO and containing a circular permutation is in SYNCHRO. A general method of providing counterexamples is to consider *self-conjugate* functions. The latter have been widely studied in many-valued logic. With the following sections in mind, the next definition is stated for functions with *several* variables. The variables still range over the set $N$, and also the function values are in $N$.

**Definition 6** *A function $f(x_1, \ldots, x_k)$ is* self-conjugate *under the permutation $g$ if*

$$f(x_1, \ldots, x_k) = g(f(g^{-1}(x_1), \ldots, g^{-1}(x_k))).$$

For unary functions, self-conjugacy means simply that the function and the permutation commute: $gf = fg$.

**Lemma 6** *A set of functions is not in* SYNCHRO *if every function in the set is self-conjugate under a permutation $g$ and, moreover, $g$ maps no element of $N$ into itself.*

*Proof.* Since $g$ commutes with every function in $\mathbf{F}$, it commutes also with every function in $\mathbf{G(F)}$. On the other hand, $g$ cannot commute with any constant $i$ because, by the assumption, $g$ maps $i$ to $j \neq i$.

We now give an example of a set in PRESYNCHRO that contains circular permutations and is not in SYNCHRO. The example could be given for any even value of $n$ but we prefer to consider $n = 4$ and state the example in terms of the four nucleotides of *DNA-computing,* [10]. This means that the functions are not over the set $\{1, 2, 3, 4\}$ but rather over the set {A,T,C,G}. The latter set is listed in the order of the argument values and, thus, we can consider value sequences. Hence, the value sequences ATGC and TACG correspond to the transpositions (CG) and (AT), respectively, whereas the value sequences CGTA and GCAT correspond to the circular permutations (ACTG) and (AGTC).

Let now $\mathbf{F}$ consist of the three functions defined by the value sequences TACG, CGTA and TAAT. Thus $\mathbf{F}$ is in PRESYNCHRO and contains a circular permutation. On the other hand, all the functions in $\mathbf{F}$ are self-conjugate under the permutation (AT)(CG) and, consequently, $\mathbf{G(F)}$ does not contain any constants. (The permutation (AT)(CG) is often referred to as the *Watson-Crick morphism.*) In fact, $\mathbf{G(F)}$ contains altogether 16 functions: 8 permutations and 8 functions of type $2 \oplus 2$. The permutations consist of the Klein Four-Group, as well as of the two transpositions and the two circular permutations already mentioned. The functions of type $2 \oplus 2$ are defined by the value sequences TAAT, ATTA, TATA, ATAT, as well as by the corresponding sequences for C and G.

In DNA-computing, the nucleotides A and T, as well as C and G, are referred to as *complementary.* (In the formation of DNA double strands, A always bonds with T, and C with G.) The functions in the set $\mathbf{G(F)}$ have the following interesting property. Whenever the argument value is changed to its complementary, then also the function value will change to its complementary: each function commutes with the Watson-Crick morphism.

In our setup for functions, stronger results, in general, can be obtained for prime values of $n$. In this case, a set in PRESYNCHRO that contains a circular permutation actually is in SYNCHRO.

The following lemma is not valid for any composite value of $n$. The proof is not difficult and can be found in [20], pp. 105-106.

**Lemma 7** *Assume that $n$ is prime and $f$ is a circular permutation. Let $N = N_1 \cup \ldots \cup N_t$, where $1 < t < n$ and the sets $N_i$ are nonempty and pairwise disjoint. Let $M$ contain exactly one element from each of the sets $N_i$. Then there are numbers $r$ and $j$ such that the set $f^r(M)$ contains at least two elements belonging to the set $N_j$.*

Lemma 7 is valid also for composite values of $n$, under the additional assumption that the sets $N_i$ are not all of the same cardinality. If $n$ is prime, this assumption is always satisfied.

**Theorem 5** *Assume that $n$ is prime, $f$ is a circular permutation and $g$ is a function of genus less than $n$. Then $f$ and $g$ generate all constants.*

*Proof.* Since $f$ is circular, it suffices to prove that some function of genus 1 is generated. If $g$ is of genus 1, there is nothing to prove. We assume, inductively, that a function $h$ of genus $t$, where $1 < t < n$, has already been generated and claim that a function of genus $u < t$ can be generated.

Let $b_1, \ldots, b_t$ be the values assumed by $h$, and let $N_1, \ldots, N_t$ be maximal subsets of the set $N$ satisfying the condition $h(N_i) = b_i$, for $i = 1, \ldots, t$. If $h^2$ is of genus smaller that $t$, we have established the claim. Otherwise, the numbers $b_i$ are in different sets $N_i$. Choose now

$$M = \{b_1, \ldots, b_t\}.$$

Choose, further, $r$ and $j$ according to Lemma 7. Then the function $hf^rh$ is of genus $u < t$.

## 5 Several Variables and Completeness

We have so far restricted our discussions to *unary* functions. ¿From now on we will consider functions of *several* variables. Although the main notions and definitions remain basically the same, some results are remarkably different.

Unary functions are conveniently defined by value sequences. For binary functions $f(x, y)$, we use *tables* where the values of $x$ are read from the rows and those of $y$ from the columns. For instance, consider the *Łukasiewicz implication* defined by $f(x, y) = \max(1, 1 - x + y)$. For $n = 8$, its table is

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 1 | 2 | 3 | 4 | 5 |
| 1 | 1 | 1 | 1 | 1 | 2 | 3 | 4 |
| 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Thus, we consider functions $f(x_1, \dots, x_k)$ whose variables range over a fixed finite set $N$ of $n$ elements and whose function values are also in $N$. For each $k \geq 1$, there are altogether $n^{n^k}$ functions with $k$ variables (or with *arity* $k$). In order to extend the definitions of depth to this more general case, we have to define what we mean by a composition sequence and its length.

When defining composition sequences, we start with a given finite set $\mathbf{F}$ of functions. Intuitively, a *composition sequence* is a well-formed sequence of function symbols (letters) from $\mathbf{F}$ and variables. Its *length* is the maximal amount of nested function symbols in the sequence. The *variables* come from a fixed denumerably infinite collection.

**Definition 7** *Let* $\mathbf{F}$ *be a given finite (nonempty) set of functions. A* composition sequence *(over* $\mathbf{F}$*) and its* length *are defined inductively as follows. A variable alone constitutes a composition sequence of length 0. Assume that* $f$ *of arity* $k$ *(*$\geq 1$*) is in* $\mathbf{F}$*, and* $f_1, \dots, f_k$ *are composition sequences of lengths* $r_i$, $1 \leq i \leq k$. *Then* $f(f_1, \dots, f_k)$ *is a composition sequence of length* $\max\{r_i | 1 \leq i \leq k\} + 1$.

For instance, if $\mathbf{F}$ consists of two functions $f(x, y)$ and $g(x)$, then

$$g(f(g(f(x, g(x))), f(f(x, y), g(z))))$$

is a composition sequence of length 5. Thus, using a binary function, we obtain composition sequences with arbitrarily many variables.

Every composition sequence defines a function; the set of all functions defined in this way is denoted again by $\mathbf{G(F)}$. Observe that, in the case of unary functions, composition sequences and their lengths (in our earlier definitions) are a special case Definition 7. Thus, our previous definition of the depth $D'(\mathbf{F}, f)$ is a special case of the following Definition 8.

**Definition 8** *Consider a set* $\mathbf{F}$ *and a function* $f$*. The* depth *of* $f$ *with respect to* $\mathbf{F}$*, in symbols* $D(\mathbf{F}, f)$*, is defined as follows. If* $f$ *is in* $\mathbf{G(F)}$*, then* $D(\mathbf{F}, f)$ *equals the length of the* shortest *composition sequence defining* $f$*. Otherwise,* $D(\mathbf{F}, f) = \infty$.

We have used the notation $D$, rather than the earlier $D'$, to indicate that we are dealing with the general case. As before, the defined notion of depth *relative* to a given set $\mathbf{F}$ can be extended to the *absolute* notions of depth $D(f)$ and

$D_C(f)$. This will be done in the next section. Now we will go into the notion of *completeness* in the general setup of functions of several variables.

As already mentioned, there are altogether $n^{n^k}$ functions with $k$ variables. A set $\mathbf{F}$ is termed *complete* if all functions, independently of the arity, are in $\mathbf{G(F)}$. It was shown by Post already in the 20's, [13], that $\mathbf{F}$ is complete if all binary functions are in $\mathbf{G(F)}$. There is now a remarkable difference with respect to the unary case and Theorem 3: single functions may constitute complete sets. Such functions are referred to as *Sheffer functions*, analogously to the *Sheffer stroke* in propositional logic. The following functions are examples of Sheffer functions in cases $n = 5, 6, 7$.

$$\begin{array}{|ccccc}\hline 2 & 1 & \ast & \ast & \ast \\ \ast & 3 & 1 & \ast & \ast \\ \ast & \ast & 4 & \ast & \ast \\ \ast & \ast & \ast & 5 & \ast \\ \ast & \ast & \ast & \ast & 1 \end{array} \qquad \begin{array}{|ccccc}\hline 3 & \ast & \ast & \ast & \ast \\ \ast & 4 & \ast & \ast & \ast \\ 2 & \ast & 5 & \ast & \ast \\ \ast & 2 & \ast & 1 & \ast \\ \ast & \ast & \ast & \ast & 2 \end{array}$$

$$\begin{array}{|cccccc}\hline 2 & 2 & \ast & \ast & \ast & \ast \\ \ast & 3 & 1 & \ast & \ast & \ast \\ \ast & \ast & 4 & 3 & \ast & \ast \\ \ast & \ast & \ast & 5 & 4 & \ast \\ \ast & \ast & \ast & \ast & 6 & 5 \\ 6 & \ast & \ast & \ast & \ast & 1 \end{array} \qquad \begin{array}{|ccccccc}\hline 2 & 1 & \ast & \ast & \ast & \ast & \ast \\ \ast & 3 & 1 & \ast & \ast & \ast & \ast \\ \ast & \ast & 4 & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & 5 & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & 6 & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast & 7 & \ast \\ \ast & \ast & \ast & \ast & \ast & \ast & 1 \end{array}$$

The positions marked by the star * can be filled arbitrarily. This shows the remarkable leeway we have in constructing Sheffer functions. The leeway is still considerably greater for prime values of $n$. As shown by the examples, it suffices to fix $n + 2$ values of a binary function in order to guarantee that the function will always be a Sheffer function, no matter how the remaining values are chosen. For composite values of $n$, always more than $n + 2$ values are needed for this purpose. For prime values of $n$, no fewer than $n + 2$ values suffice.

We will now give some *completeness criteria.* In general, completeness is fairly well understood but very little is known about the length of composition sequences, that is, about the various notions of *depth.* Are some complete sets more likely to have shorter composition sequences (for specific functions) than others? Do some functions have inherently longer composition sequences than others, no matter what complete set we are considering? Such problems are open already on the unary level, and also for groups. The knowledge of the bases, [11], does not tell much about the length of composition sequences. Shortest solutions for a configuration in Rubik's Cube are hard to find, which is another example about finding the shortest length of a composition sequence, in terms of the generators for a group.

Since unary functions generate only unary functions, we must have something non-unary in a complete set. In the following definition, mildest conceivable conditions are imposed upon such an additional function.

**Definition 9** *A function* $f(x_1, \ldots, x_k)$ depends essentially *on the variable* $x_i$ *if there are numbers* $b_1, \ldots, b_k, b'_i$ *such that*

$$f(b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_k) \neq f(b_1, \ldots, b_{i-1}, b'_i, b_{i+1}, \ldots, b_k).$$

*A function* $f(x_1, \ldots, x_k)$ *satisfies* Słupecki conditions *if it depends essentially on at least two variables and assumes all* $n$ *values.*

**Theorem 6** *Assume that* $n \geq 5$*. Every set* **F** *containing all permutations and an arbitrary function satisfying Słupecki conditions is complete.*

Theorem 6 has been established in [14]. The condition $n \geq 5$ is necessary. If $n = 4$ and **F** consists of all permutations and the function

| | | | |
|---|---|---|---|
| 2 | 1 | 1 | 2 |
| 4 | 3 | 3 | 4 |
| 3 | 4 | 4 | 3 |
| 1 | 2 | 2 | 1 |

then **G(F)** contains no unary functions of types $2 \oplus 1 \oplus 1$ or $3 \oplus 1$. Counter-examples can be constructed also for the cases $n = 2, 3$.

Natural improvements of Theorem 6 consist of having a smaller group of permutations in **F**. (It is clear that Słupecki conditions cannot be weakened further.) As we saw in the preceding section, a transitive group is not even enough to generate constants. Doubly transitive groups do not in some cases escape self-conjugacy. However, triple transitivity is sufficient, [16].

**Theorem 7** *Assume that* $n$ *is greater than* 3 *and is not a power of* 2*. Then a set* **F** *is complete if it contains a function satisfying Słupecki conditions and permutations generating a triply transitive group of degree* $n$*.*

Again, counter-examples can be constructed for the cases $n = 2^r$. Although much more complicated, they are in some sense analogous to the counter-examples discussed above. As an example, we consider the case $n = 8$ and the function $f(x, y)$ defined by

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Consider, further, the two permutations (1376528) and (17)(46). They generate a triply transitive group of degree 8. However, if **F** consists of $f$ and the two permutations, **G(F)** contains no other unary functions except all constants, some permutations (altogether 1344 of them), some functions of type $2 \oplus 2 \oplus 2 \oplus 2$ (2352 of them), and some functions of type $4 \oplus 4$ (392 of them).

*Remark.* The group generated by the two permutations is, in fact, the holomorph of an abelian group of order 8 and type (1,1,1). If it is expressed as a permutation group of degree 8, it is triply transitive and consists of the identity, 384 7-cycles, as well as permutations with cyclic structures

$$3 \times 3, 6 \times 2, 4 \times 4, 2 \times 2 \times 2 \times 2, 2 \times 2, 4 \times 2.$$

The case $n = 4$ is similarly exceptional, the exceptional group being the holomorph of the Klein Four-Group (which equals the symmetric group $S_4$).

We mention, finally, that the function mentioned after Theorem 6 is self-conjugate (see Definition 6) under the permutation (13)(24). We go back to the DNA interpretation given in Section 4. If we write the nucleotides in the order A=1, C=2, T=3, G=4, which order makes the permutation (13)(24) equal to the Watson-Crick morphism, our function $f(x, y)$ gets the form

$$\begin{array}{|cccc} \hline C & A & A & C \\ G & T & T & G \\ T & G & G & T \\ A & C & C & A \end{array}$$

Which unary functions does $f(x, y)$ generate? For instance, $f(x, x)$ yields the circular permutation (ACTG), $f(x, f(x, x))$ yields the function of type $2 \oplus 2$ with the value sequence ATTA, and $f(x, f(f(x, x), f(x, x)))$ yields the transposition (CG). It is easy to show that $f(x, y)$ generates, among unary functions, exactly the functions in the set **G(F)** discussed after Lemma 6. In particular, no constants are generated. This is obvious because $f(x, y)$ is self-conjugate under the Watson-Crick morphism (AT)(CG).

This example is easily extended to even values of $n \geq 4$. The Watson-Crick morphism will always be a product of transpositions covering the whole $N$. Functions can have more than two variables. But if all functions are self-conjugate under the Watson-Crick morphism, no constants will be generated.

## 6 The Depth of Functions Revisited

We are now ready to generalize the different notions of (absolute) depth to the case, where the basic sets **F** of functions may contain functions with arbitrarily many variables. The depth $D(\mathbf{F}, f)$ was defined in Definition 8. Based on this notion, the (absolute) depth and complete depth can now be defined as in Definitions 2 and 3.

**Definition 10** *The* depth $D(f)$ *of a function* $f$ *is defined by the equation*

$$D(f) = \max(D(\mathbf{F}, f)),$$

*where* **F** *ranges over all sets with the property that $f$ is in* **G(F)**. *The* complete depth $D_C(f)$ *of a function $f$ is defined by the right side of the same equation, where now* **F** *ranges over complete sets.*

The following lemma is immediate by the definitions.

**Lemma 8** *The relation $D_C(f) \leq D(f)$ holds for all functions, and the relations $D'_C(f) \leq D_C(f)$ and $D'_C(f) \leq D'(f) \leq D(f)$ hold for all unary functions.*

Not much is known about the cases where the inequalities are strict. Also the interrelation between $D'(f)$ and $D_C(f)$ is unknown for unary functions $f$. Similarly as before, we use the notation SYNCHRO for the class of all sets **F** such that at least one of the constants $c_i$ is in **G(F)**. (Recall that in defining SYNCHRO we have some fixed $n$ in mind.) Analogously to the definitions above, we now define the depths $D(\text{const})$ and $D_C(\text{const})$.

A detailed case study was given in [18] showing that, for $n = 3$, we have $D(\text{const}) > (n-1)^2$ and even $D_C(\text{const}) > (n-1)^2$. The depths $D(\mathbf{F}, g)$ were considered, where **F** consists of one function only, namely, the Sheffer function $f(x, y)$ defined by the table

| | | |
|---|---|---|
| 2 | 2 | 1 |
| 1 | 3 | 3 |
| 1 | 2 | 1 |

Since **F** is complete it follows that, for any function $g$,

$$D_C(g) \geq D(\mathbf{F}, g).$$

It turns out that all of the constants are of depth 5 or 6 as regards this particular **F**.

For the general $n$, consider the set $\mathbf{F}'$ consisting of two functions $g(x)$ and $f(x, y)$, where $g(x)$ is the circular permutation $(12\ldots n)$ and $f(x, y)$ is defined by

$$f(x, y) = x \text{ except } f(n, 1) = 1.$$

Thus, for $n = 4$, the value sequence of $g$ is 2341, and $f(x, y)$ is defined by the table

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 1 | 4 | 4 | 4 |

The following result was established in [19].

**Lemma 9** *All constants $c_i$ satisfy $D(\mathbf{F}', c_i) > (n-1)^2$. The constant $c_n$ satisfies $D(\mathbf{F}', c_n) > n(n-1)$.*

This lemma yields immediately the following theorem which should be contrasted to Conjectures 1 and 2. Since the set $\mathbf{F}'$ is not complete, analogous conclusions cannot be made with respect to Conjecture 3.

**Theorem 8** *For a constant $c_i$ we have $D(c_i) > n(n-1)$. Moreover,*

$$D(\text{const}) > (n-1)^2.$$

## 7 Conclusion

Completeness, both in the case of functions of one variable and in the general case, has been widely investigated in connection with *many-valued logic.* On the other hand, very little is known about the *complexity* of generating various functions by complete sets (or other sets **F** such that the target function is in **G(F)**). The recent interest in problems such as the *road coloring* or *Černý Conjecture* shows the significance of such complexity issues in *automata theory.* The complexity issues can be formulated in terms of the various notions of *depth* considered above. The general approach dealing with compositions of functions over a finite domain $N$ with $n$ elements has turned out to be very useful: semantically very different problems can be treated under the same formalism.

We have shown that the depth may vary considerably with the target function. We have also established criteria about the reachability of some target functions, notably constants. (It is interesting to note that here the theory comes rather close to some issues in DNA computing.) Properties of $n$ such as primality or being a power of 2 turn out to be important, independently of the semantic interpretation. Most of the questions about depth, as well as about the comparison of different notions of depth, remain open. However, our results show that the study of functions of several variables may shed light also to the case where all functions considered are of one variable.

**Dedication.** This paper is dedicated to *Neil Jones* on his 60th birthday. Our friendship and discussions, ranging from specific technical to general philosophical matters, have meant much to me over the years. Neil was also my doctoral student at the University of Western Ontario in 1966-67, where I took over the job of supervising Neil's Thesis from Dr. Satoru Takasu. I am convinced that I learned more from the student than the other way round. The present paper, although not directly connected with Neil's work, is still related to it: the notion of depth measures complexity and, besides, Neil was always interested in classical problems.

## References

1. R. Adler, I. Goodwin and B. Weiss, Equivalence of topological Markov shifts. Israel J. Math. 27 (1977) 49-63.
2. S. Bogdanović, B. Imreh, M. Ćirić and T. Petković, Directable automata and their generalizations: a survey. Novi Sad J. Math. 29 (1999).
3. J. Černý, Poznámka k homogénnym experimentom s konečnými automatmi. Mat.fyz.čas SAV 14 (1964) 208-215.
4. J. Černý, A. Pirická and B. Rosenauerová, On directable automata. Kybernetika 7 (1971) 289-298.
5. L. Dubuc, Sur les automates circulaires et la conjecture de Černý. Informatique Théorique et Applications 32 (1998) 21-34.
6. D. Epstein, Reset sequences for monotonic automata. Siam J. Comput. 19 (1990) 500-510.
7. B. Imreh and M. Steinby, Directable nondeterministic automata. Acta Cybernetica 14 (1999) 105-116.

8. A. Mateescu and A. Salomaa, Many-valued truth functions, Černý's conjecture and road coloring. EATCS Bulletin 68 (1999) 134-150.
9. E.F. Moore, Gedanken experiments on sequential machines. In C.E. Shannon and J. McCarthy (ed.) Automata Studies, Princeton University Press (1956) 129-153.
10. G. Păun, G. Rozenberg and A. Salomaa, DNA Computing. New Computing Paradigms. Springer-Verlag, Berlin, Heidelberg, New York (1998).
11. S. Piccard, Sur les bases du groupe symétrique et les couples de substitutions qui engendrent un goupe régulier. Librairie Vuibert, Paris (1946).
12. J.-E. Pin, Le problème de la synchronisation, Contribution a l'étude de la conjecture de Černý. Thèse de $3^e$ cycle a l'Université Pierre et Marie Curie (Paris 6) (1978).
13. E. Post, Introduction to a general theory of elementary propositions. Amer. J. Math. 43 (1921) 163-185.
14. A. Salomaa, A theorem concerning the composition of functions of several variables ranging over a finite set. J. Symb. Logic 25 (1960) 203-208.
15. A. Salomaa, On the composition of functions of several variables ranging over a finite set. Ann. Univ. Turkuensis, Ser. AI, 41 (1960).
16. A. Salomaa, On basic groups for the set of functions over a finite domain. Ann. Acad. Scient. Fennicae, Ser. AI, 338 (1963).
17. A. Salomaa, Composition sequences for functions over a finite domain. Turku Centre for Computer Science Technical Report 332 (2000). To appear in Theoretical Computer Science.
18. A. Salomaa, Depth of functional compositions. EATCS Bulletin 71 (2000) 143-150.
19. A. Salomaa, Compositions over a finite domain: from completeness to synchronizable automata. In A. Salomaa, D. Wood and S. Yu (Ed.) A Half-Century of Automata Theory, World Scientific Publ. Co. (2001) 131-143.
20. S.V. Yablonskii, Functional constructions in $k$-valued logic (in Russian). Tr.Matem.Inst. im. V.A.Steklova 51,5 (1958) 5-142.

# Lambda Calculi and Linear Speedups

David Sands, Jörgen Gustavsson, and Andrew Moran[⋆]

Department of Computing Science,
Chalmers University of Technology and Göteborg University, Sweden.
`www.cs.chalmers.se`

**Abstract.** The equational theories at the core of most functional programming are variations on the standard lambda calculus. The best-known of these is the call-by-value lambda calculus whose core is the value-beta computation rule $(\lambda x.M)\,V \rightarrow M[{}^{V}\!/_{x}]$ where $V$ is restricted to be a *value* rather than an arbitrary term.
This paper investigates the transformational power of this core theory of functional programming. The main result is that the equational theory of the call-by-value lambda calculus cannot speed up (or slow down) programs *by more than a constant factor*. The corresponding result also holds for call-by-need but we show that it does not hold for call-by-name: there are programs for which a single beta reduction can change the program's asymptotic complexity.

## 1 Introduction

This paper concerns the transformational power of the core theory of functional programming. It studies some computational properties of call-by-value lambda calculus, extended with constants; the prototypical call-by-value functional language. We show that this equational theory has very limited expressive power when it comes to program optimisation: equivalent programs always have the same complexity. To develop this result we must first formulate the question precisely. What does it mean for lambda terms to have the same complexity? What is a reasonable measure of cost? First we review the role of lambda calculi in a programming language setting.

### 1.1 Lambda Calculi

Lambda-calculus provides a foundation for a large part of practical programming. The "classic" lambda calculus is built from terms (variables, abstraction and application), a reduction relation (built from beta and eta). The calculus itself is an *equational theory* built from reduction by compatible, transitive and symmetric closure of the reduction relation.

Although many (functional) programming languages are based on lambda calculus (Lisp, Scheme, ML, Haskell), the correspondence between the classical

[⋆] Author's current address: Galois Connections, Oregon. `moran@galcon.com` theory and the programming languages is not particularly good, and this mismatch has given rise to a number of *applied* lambda calculi – calculi more finely tuned to the characteristics of programming languages.

Plotkin's criteria [Plo76] for the correspondence between a programming language and a reduction calculus:

1. Standard derivations are in agreement with the operational semantics of the programming language (abstract machine e.g. SECD)
2. Equations for the calculus are operationally sound – that is, they are contained in some suitable theory of observational equivalence.

Lambda calculus (and combinatory-logic) based programming languages are typically implemented using one of two reduction strategies: *call-by-value* – e.g. Scheme, various ML dialects, which dictates that arguments to functions are evaluated before the call is made, and *call-by-need*, exemplified by implementations of Clean and Haskell, in which an argument in any given application is evaluated only when it is needed – and in that case it is not evaluated more than once.

Accepting Plotkin's criteria, the standard lambda calculus is not a good fit for lambda-calculus-based programming languages, since it is

– not operationally sound for call-by-value languages, since beta reduction is not sound for call-by-value, and
– not in agreement with standard derivations for call-by-need languages, since unlike call-by-need computation, beta reduction entails that arguments in any given call might be evaluated repeatedly.

As a consequence a number of "applied lambda calculi" have been studied, most notably:

– $\lambda_v$ calculus [Plo76] and its conservative extensions to handle control and state [FH92]
– $\lambda_{need}$ [AFM+95] – and later refinements [AB97]

The main result of this article is that the call-by-value lambda calculus cannot speed up (or slow down) programs *by more than a constant factor*. That is to say, the equational theory, although sufficient to support computation, can never yield a superlinear speedup (or slowdown). The corresponding result also holds for the lesser-known call-by-need calculus – a result which was outlined in [MS99].

In contrast, we also show that the result does *not* hold for the standard lambda calculus (based on full beta reduction) and the corresponding call-by-*name* computational model. In the case of full beta reduction there are a number of subtleties – not least of which is finding a machine model which actually conforms to the cost model of beta reduction.

## 1.2 Related Work

Much of the work on the border between programming languages and complexity has focused on characterising the computational power of various restricted programming languages. The present work relates most closely to characterisations of the range of transformations that are possible in particular program transformation systems. A few examples of negative results exist. The closest is perhaps Andersen and Gomard's [AG92, JGS93]. They considered a simple form of partial evaluation of flow chart programs and showed that superlinear speedups were not possible. Amtoft suggests that even expressive systems such as unfold-fold, in a particular restricted setting, can give only constant-factor speedups. Incompleteness results for unfold-fold transformation include the fact that these transformations cannot change the *parallel complexity* of terms [Zhu94, BK83], where *parallel complexity* assumes a computational model in which all recursions at the same nesting level are evaluated in parallel. Our earlier work [San96] on the use of improvement for establishing the *extensional* correctness of memoization-based program transformations showed that recursion-based *deforestation* in a call-by-name setting cannot speed up programs by more than a constant factor.

The result in this paper was first proved for call-by-need lambda calculus in [MS99], and the key development via an asymptotic version of *improvement theory*, was also introduced there.

## 1.3 Overview

The remainder of the paper is organised as follows:

**Section** 2 formalises what it means for one program to be asymptotically least as fast as another, beginning with a traditional definition, and strengthening it until it is adequate to handle a higher-order language. **Section** 3 introduces the call-by-value lambda-calculus and **Section** 4 discuss what is a reasonable model of computation cost for the calculus. **Section** 5 establishes the main result via a notion of *strong improvement*. **Section** 6 considers call-by-name computation and show that the result fails there. **Section** 7 concludes.

## 2 Relative Complexity

The main result which we prove in this article is that:

> If $M = N$ is provable in the call-by-value lambda calculus $\lambda_v$ then $M$ and $N$ have the same asymptotic complexity with respect to the call-by-value computational model.

In this section we make precise the notion of "having the same asymptotic complexity" when $M$ and $N$ are (possibly open) lambda terms.

## 2.1 Improvement

We will characterise the notion of one program having the same complexity as another in terms of an *improvement* preorder. Let us begin with a “traditional” view of when one program is better than another. Firstly we will assume that we are comparing extensionally equivalent programs. Consider first the simpler case of two programs $P$ and $Q$ which compute some function $f \in D \to E$. Let $p$ and $q \in \mathbb{N} \to \mathbb{N}$ denote the time functions for $P$ and $Q$ respectively, where $p(n)$ is the worst-case number of computation steps required to compute $P$ on an input of size $n$.

In traditional algorithm analysis style one would say that $P$ is asymptotically at least as fast as $Q$ (or equivalently, $P$ is no more than a constant factor slower than $Q$) if

$$p(n) \in \mathcal{O}(q(n)).$$

More precisely, if there exit constants $a$ and $n_0$ such that

$$\forall n > n_0 . p(n) \leq a \cdot q(n). \tag{2.1}$$

We will work with a definition which is both

– stronger – yielding a stronger main result, and
– more general – since we intend to deal with lambda terms and program fragments rather than whole programs computing over “simple” domains.

*Strengthening* Let us first deal with the strengthening of the condition. The above definition is based on the notion of the *size* of a given input. As well as being difficult to generalise to a lambda calculus, it also makes for a weaker definition, since the time functions give the worst case running time for each input size. For example, suppose we have a quicksort algorithm $Q$ which is $\mathcal{O}(n^2)$ in the worst case (e.g., for fully sorted lists), but performs much better for a large part of the input domain. According to the above definition, a sorting algorithm $B$ implementing bubble sort, which exhibits quadratic behaviour in *every* case, would be considered asymptotically equivalent to $Q$.

By using a stronger notion of “asymptotic equivalence” of programs our main result will be stronger – enabling us to conclude that bubble sort could never be transformed into quicksort using the calculus. With this in mind, an obvious strengthening of the notion of asymptotic speedup – without resorting to average-case complexity – is to quantify over all inputs rather than just the worst-case for each input size. Firstly, we note that 2.1 can be written equivalently as

$$\exists a, b \in \mathbb{N} . \forall n . \, p(n) \leq a \cdot q(n) + b \tag{2.2}$$

The equivalence can be seen by taking $b = \max\{p(m) \mid m \leq n_0\}$. This is a useful form since it eliminates the use of the inequality relation – a prerequisite for generalising to inputs which do not have a natural “size” measure. To make the desired strengthening, let $\text{time}_P(d)$ denote the running time of $P$ on input

$d$, and similarly for $Q$. The strengthening of the definition is to say that $Q$ is *improved by* $P$ iff

$$\exists a, b \in \mathbb{N}. \forall d \in D. \mathrm{time}_P(d) \leq a \cdot \mathrm{time}_Q(d) + b$$

We would then say that $P$ has the same asymptotic complexity as $Q$ if $P$ is improved by $Q$, and $Q$ is improved by $P$.

*Generalisation* Having strengthened our notion of asymptotic improvement, we now seek a *generalisation* which permits us to deal with not just whole programs but also program fragments. The generalisation is to replace quantification over program inputs by quantification over the *contexts* in which the program fragment can occur. A context (for a given programming notation), written $C[\cdot]$, is a program containing a hole $[\cdot]$; $C[M]$ denotes the result of replacing the hole by program fragment $M$ – possibly capturing free variables that occur in $M$ in the process.

We identify the concept of *program* with a closed term.

**Definition 1 (Asymptotic Improvement).** *Given a programming language equipped with*

- *a notion of operational equivalence between terms, and*
- *an operational model (abstract machine) which defines a partial function* $time_P$*, the time (number of steps) to evaluate program* $P$ *(i.e.* $time_P$ *is well defined whenever* $P$ *terminates)*

*define the following relation on terms:*

*Let* $M$ *and* $N$ *be operationally equivalent terms.* $N$ *is no more than a constant factor slower than* $M$ *if there exist integers* $a$ *and* $b$ *such that for all contexts* $C$ *such that* $C[M]$ *and* $C[N]$ *are closed and terminating,*

$$time_{C[N]} \leq a \cdot time_{C[M]} + b$$

*We alternatively say that* $M$ is asymptotically improved by $N$*, and write* $M \underset{\approx}{\triangleright} N$*. If both* $M \underset{\approx}{\triangleright} N$ *and* $N \underset{\approx}{\triangleright} M$ *then we write* $M \underset{\approx}{\triangleleft\!\triangleright} N$*, and say that* $M$ *and* $N$ *are* asymptotically cost equivalent.

A straightforward but important consequence of this definition is that $\underset{\approx}{\triangleright}$ is a *congruence* relation: it is transitive, reflexive, symmetric and preserved by contexts, i.e, $M \underset{\approx}{\triangleright} N$ implies that $\mathbb{C}[M] \underset{\approx}{\triangleright} \mathbb{C}[N]$.

## 3 The $\boldsymbol{\lambda_v}$ Calculus

We will work with a syntax containing just lambda calculus and some arbitrary constants. This is essentially just Landin's ISWIM [Lan66, Plo76]. We follow Felleisen and Hieb's presentation [FH92] fairly closely, which in turn summarises Plotkin's introduction of the $\lambda_v$ calculus [Plo76].

The language $\Lambda$ consists of terms which are either *values* or *applications*:

$$\begin{array}{ll} \text{Terms } L, M, N & ::= V \mid M\,N \\ \text{Values } V, W & ::= b \mid f \mid x \mid \lambda x.M \end{array}$$

Values include variables and two kinds of constants, the basic constants ($b \in$ BConsts) and functional constants ($f \in$ FConsts). The constants are intended to include basic types like integers and booleans, and functional types include arithmetical and logical operations on these types. The exact meaning of the constants is given by a partial function:

$$\delta : \text{FConsts} \times \text{BConsts} \rightarrow \text{Values}^0$$

where Values$^0$ denotes the set of closed values (and similarly $\Lambda^0$ the set of closed terms). We impose the restriction that there is only a finite set of constants so that operations on constants can be implemented in constant time. We adopt the usual conventions of identifying terms which only differ in the names of their bound variables.

*Operational Semantics* Computation is built up from the following basic reduction rules:

$$\begin{array}{rll} f\,b \mapsto \delta(f,a) & & (\delta) \\ (\lambda x.M)\,V \mapsto M[{}^{V}\!/_{x}] & & (\beta_v) \end{array}$$

The operational semantics is a deterministic restriction of the application of the basic reduction rules which follows the "call-by-value" discipline of ensuring that arguments are evaluated before the function application can be evaluated. The computational order can be specified by only permitting reductions to take place in an *evaluation context.* Evaluation contexts are term contexts where the hole appears in the unique position we may perform a reduction:

$$\text{Evaluation Contexts } \mathbb{E} ::= [\cdot] \mid \mathbb{E}\,M \mid V\,\mathbb{E}$$

The small-step operational semantics is then given by the rule:

$$\mathbb{E}[M] \longmapsto \mathbb{E}[N] \text{ iff } M \mapsto N$$

It is easy to see that this one-step computation relation is deterministic. We write $M \longmapsto^n N$ to mean that $M$ computes in $n \geq 0$ steps to $N$. Complete evaluation is defined by the repeated computation until a value is obtained:

**Definition 2.**

- *$M{\Downarrow}^n V$ iff $M \longmapsto^n V$.*
- *$M{\Downarrow}^n$ iff there exists $V$ such that $M{\Downarrow}^n V$*
- *$M{\Downarrow}^{\leq m}$ iff there exists $n$ such that $M{\Downarrow}^n$ and $n \leq m$.*

*When $M{\Downarrow}^n$ we say that $M$* converges *in $n$ steps.*

*The $\lambda_v$ Calculus* The $\lambda_v$ calculus is the least equivalence relation ($=_v$) containing the basic reduction relation ($\mapsto$) and which is closed under all contexts:

$$M =_v N \implies \mathbb{C}[M] =_v \mathbb{C}[N]$$

where $\mathbb{C}$ denotes an arbitrary term context, and $\mathbb{C}[M]$ denotes the textual replacement of the (zero or more) holes in $\mathbb{C}$ by the term $M$. I.e., $=_v$ is the transitive, reflexive, symmetric and compatible closure of $\mapsto$. When $M =_v N$ it is also standard to write $\lambda_v \vdash M = N$.

## 4 Cost Models for the $\lambda_v$ Calculus

It is tempting to take the counting of reduction steps as the natural model of computation cost for the $\lambda_v$ calculus, since it is a simple and high level definition. But is this a reasonable choice? It is a crucial question since if we make an erroneous choice then our results would not say anything about actual implementations and they would be of little value.

We believe that a reasonable requirement of a cost model is that it should be implementable within a program size dependent constant factor. I.e., there should exist a linear function $h$ such that, for every program $M$, if the cost of evaluating $M$ in the model is $n$ then the actual cost is within $|M| \cdot h(n)$ where $|M|$ denotes the size of $M$.

One could alternatively insist that the constant to be independent of the program, i.e., that there should exist a linear function $h$ such that for every program $M$, if the cost of evaluating $M$ in the model is $n$ then the actual cost is within $h(n)$. That is a much stronger requirement, used in e.g. Jones "constant-factor-time hierarchy" work [Jon93, Ros98]. We believe that the results in this paper could be extended to such models, although we have not done so.

In the remainder of this section we will argue that to count reduction steps is a valid cost model for the $\lambda_v$ calculus in former sense. Although it may seem intuitively obvious, it is a subtle matter as we will see when we turn to the call-by-name case later in this paper.

### 4.1 An Abstract Machine

Here we present a heap based abstract machine and argue informally that it can be implemented (e.g., in the RAM model) within a constant factor linearly dependent on the program size. Later we will relate the number of abstract machine steps to the model based on counting reduction steps.

For the purpose of the abstract machine we extend the syntax of the language with heap variables, we use $p$ to range over those and we write $M[p/x]$ for substitution of a heap variable $p$ for term variable $x$. We let $\Gamma$ and $S$ range over heaps and stacks respectively. Heaps are mappings from heap variables to values. We will write $\Gamma\{p = V\}$ for the extension of $\Gamma$ with a binding for $p$ to

$V$. Stacks are a sequence of "shallow" evaluation contexts of the form $[\cdot]\,N$ or $V\,[\cdot]$, given by:

$$\text{Stacks } S := \epsilon \mid [\cdot]\,N : S \mid V\,[\cdot] : S$$

where $\epsilon$ denotes the empty stack. Configurations are triples of the form $\langle\,\Gamma,\ M,\ S\,\rangle$ and we will refer to the second component $M$ as the *control* of the configuration. The transitions of the abstract machine are given by the following rules.

$$\begin{array}{lrrlrrl}
\langle\,\Gamma, & M\,N, & S\rangle \rightsquigarrow & \langle\,\Gamma, & M, & [\cdot]\,N : S\rangle & \\
\langle\,\Gamma, & f, & [\cdot]\,N : S\rangle \rightsquigarrow & \langle\,\Gamma, & N, & f\,[\cdot] : S\rangle & \\
\langle\Gamma\{p{=}V\}, & p, & S\rangle \rightsquigarrow & \langle\,\Gamma\{p{=}V\}, & V, & S\rangle & \\
\langle\,\Gamma, & b, & f\,[\cdot] : S\rangle \rightsquigarrow & \langle\,\Gamma, & \delta(f,a), & S\rangle & (\delta) \\
\langle\,\Gamma, & \lambda x.M, & [\cdot]\,N : S\rangle \rightsquigarrow & \langle\,\Gamma, & N, & (\lambda x.M)\,[\cdot] : S\rangle & \\
\langle\,\Gamma, & V, & (\lambda x.M)\,[\cdot] : S\rangle \rightsquigarrow & \langle\,\Gamma\{p{=}V\}, & M[{}^{p}\!/_{x}], & S\rangle & (\beta)
\end{array}$$

A crucial property of the abstract machine is that all terms in configurations originate from subterms in the original program, or from subterms of the values returned by $\delta(f,b)$. More precisely, for each term $M$ in a configuration there exist a substitution $\sigma$ mapping variables to heap variables and a term $N$ which is a subterm of the original program or a subterm of the values in $\mathit{range}(\delta)$ such that $M \equiv N\sigma$. The property is sometimes called *semi compositionality* [Jon96]. The property is ensured because terms are never substituted for variables, thus the terms in the abstract machine states are all subterms (modulo renaming of variables for heap variables) of the original program. This is not the case for the reduction semantics, where terms may grow arbitrarily large as the computation proceeds.

Thanks to semi compositionality it is easy to see that each step can be implemented in time proportional to the maximum of the size of the original program and the size of the values in $\mathit{range}(\delta)$[1]. This is the only property we require of the machine steps.

## 4.2 Relating the Cost Model to the Abstract Machine

In the remainder of this section we will prove that the number of abstract machine steps required to evaluate a program is within a *program independent* constant factor of the number of reduction steps, as stated in the following lemma.

**Lemma 1.** *If $M\Downarrow^n$ then there exist $\Gamma$ and $V$ such that*

$$\langle\,\emptyset,\ M,\ \epsilon\,\rangle \rightsquigarrow^{\leq 6n} \langle\,\Gamma,\ V,\ \epsilon\,\rangle$$

Together with our informal argument that the abstract machine can be implemented within a program dependent constant factor this shows that to count reduction steps is a valid cost model of the $\lambda_v$-calculus.

[1] This set is finite and independent of the program in question.

We will prove Lemma 1 in two steps. In the first we show that the number of reduction steps in the reduction semantics is the same as the number of abstract machine transitions which use one of the rules $(\beta)$ or $(\delta)$.

**Lemma 2.** *If $M{\Downarrow}^n$ then there exist $\Gamma$ and $V$ such that*

$$\langle \emptyset,\ M,\ \epsilon \rangle \leadsto^* \langle \Gamma,\ V,\ \epsilon \rangle$$

*using exactly $n$ applications of $(\beta)$ and $(\delta)$*

The proof of this lemma is by standard technique relating small-step semantics and abstract machines and is omitted. Next we claim that the total number of abstract machine transitions is at most six times the number of $(\beta)$ and $(\delta)$ transitions.

**Lemma 3.** *If $\langle \emptyset,\ M,\ \epsilon \rangle \leadsto^m \langle \Gamma,\ V,\ \epsilon \rangle$ using $n$ applications of $(\beta)$ and $(\delta)$ then $m \leq 6n$.*

Taken together with Lemma 2 this immediately implies Lemma 1.

In the remainder of this section we prove Lemma 3. First we define a measure $\lceil\cdot\rceil$ on terms and stacks as follows.

$$\begin{aligned}
\lceil M \rceil &= \begin{cases} 0 & \text{if } M = p \\ 1 & \text{otherwise} \end{cases} \\
\lceil \epsilon \rceil &= 0 \\
\lceil [\cdot]\, M : S \rceil &= 2 + \lceil S \rceil \\
\lceil V\, [\cdot] : S \rceil &= 4 + \lceil S \rceil
\end{aligned}$$

With the help of the measure we can generalise Lemma 3 to configurations with a non empty heap and stack.

**Lemma 4.** *If $\langle \Gamma,\ M,\ S \rangle \leadsto^m \langle \Gamma',\ V,\ \epsilon \rangle$ using $n$ applications of $(\beta)$ and $(\delta)$ then $m \leq 6n - \lceil M \rceil - \lceil S \rceil + 1$.*

Proof. The proof of Lemma 4 is by induction over $m$.

*case $m = 0$:* In the base case we have that $n = 0$ and that $\langle \Gamma,\ M,\ S \rangle \equiv \langle \Gamma',\ V,\ \epsilon \rangle$. Thus

$$6n - \lceil M \rceil - \lceil S \rceil + 1 = 6n - \lceil V \rceil - \lceil \epsilon \rceil + 1 = 0 = m$$

as required.

*case $m > 0$:* We proceed by a case analysis on the abstract machine rule in question. We will only consider $(\beta)$ and the rule for applications. The other cases follows similarly.

*subcase* ($\beta$)*:* In this case $\langle \Gamma,\ M,\ S\rangle \equiv \langle \Gamma,\ V,\ (\lambda x.N)\,[\cdot] : T\rangle$ for some $V$, $N$ and $T$ and we know from the induction hypothesis that

$$6(n-1) - \lceil N[{}^{p}\!/\!_{x}]\rceil - \lceil T\rceil + 1 \geq m - 1$$

for some $p$. The required result then follows by the following calculation.

$$\begin{aligned}
6n - \lceil M\rceil - \lceil S\rceil + 1 &= 6n - \lceil V\rceil - \lceil(\lambda x.N)\,[\cdot] : T\rceil + 1\\
&= 6n - 1 - 4 - \lceil T\rceil + 1\\
&= 6(n-1) - \lceil N[{}^{p}\!/\!_{x}]\rceil - \lceil T\rceil + 1 + 1 + \lceil N[{}^{p}\!/\!_{x}]\rceil\\
&\geq m - 1 + 1 + \lceil N[{}^{p}\!/\!_{x}]\rceil\\
&\geq m
\end{aligned}$$

*subcase (application rule):* In this case $\langle \Gamma,\ M,\ S\rangle \equiv \langle \Gamma,\ N\,L,\ S\rangle$ for some $N$ and $L$ and we know from the induction hypothesis that

$$6n - \lceil N\rceil - \lceil[\cdot]\,L : S\rceil + 1 \geq m - 1.$$

The required result then follows by the following calculation.

$$\begin{aligned}
6n - \lceil M\rceil - \lceil S\rceil + 1 &= 6n - \lceil N\,L\rceil - \lceil S\rceil + 1\\
&= 6n - \lceil S\rceil\\
&= 6n - \lceil N\rceil - \lceil[\cdot]\,L : S\rceil + 1 + 1 + \lceil N\rceil\\
&\geq m - 1 + 1 + \lceil N\rceil\\
&\geq m
\end{aligned}$$

□

## 5 Constant Factors

We instantiate the definition of asymptotic improvement with the call-by-value computation model above. I.e., we take

$$\mathrm{time}_M = n \iff M{\Downarrow}^n.$$

In the remainder of this section we will demonstrate the proof of our claim that the call-by-value lambda calculus cannot change asymptotic complexity. In other words, we show that $=_v$ is contained in $\overset{\triangleleft\!\triangleright}{\approx}$.

### 5.1 Strong Improvement

The main vehicle of our proof is a much stronger improvement relation which doesn't even permit constant factors. The reason why we make use of this relation is that it is "better behaved" semantically speaking. In particular it possesses relatively straightforward proof methods such as a *context lemma*.

**Definition 3 (Strong Improvement).** *$M$ is strongly improved by $N$ iff for all contexts $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed*

$$\mathbb{C}[M]\Downarrow^{n} \Rightarrow \mathbb{C}[N]\Downarrow^{\leq n}.$$

*Following [MS99], we write this relation as $M \trianglerighteq N$. If $M \trianglerighteq N$ and $N \trianglerighteq M$ we say that $M$ and $N$ are cost equivalent and we write it as $M \trianglelefteq\!\trianglerighteq N$.*

Cost equivalence is a very strong relation since it requires that the two program fragments use exactly the same number of computation steps in all possible program contexts. For example $(\lambda x.M)\,V$ is not cost equivalent to $M[{}^{V}\!/_{x}]$ because the former takes up one more step than the latter each time it is executed.

## 5.2 The Tick

We introduce a technical device (widely used in our earlier work with improvement) for syntactically representing "a single computation step" – a dummy cost called a *tick*. We will denote the addition of a dummy cost to $M$ as ${}^{\checkmark}M$. It is not a language extension proper since "the tick" can be encoded in the language as

$${}^{\checkmark}M \overset{\text{def}}{=} (\lambda x.M)\,() \qquad \text{where } x \notin \mathsf{FV}\,(M)$$

where $()$ is an arbitrary basic constant[2].

We can now state the fact that a reduct is cost equivalent to its redex if we add a dummy cost to it.

**Lemma 5.**

- $f\,b \trianglelefteq\!\trianglerighteq {}^{\checkmark}\delta(f,a)$
- $(\lambda x.M)\,V \trianglelefteq\!\trianglerighteq {}^{\checkmark}M[{}^{V}\!/_{x}]$

Although this result is intuitively obvious, it is hard to prove it directly since it involves reasoning about computation in arbitrary program contexts. The complications in the proof can be packaged up in a *Context Lemma* [Mil77] which provides a convenient way to show strong improvements by investigating the behaviour of terms in evaluation contexts.

**Lemma 6 (Context Lemma).** *Let $\sigma$ range over substitutions mapping variables to closed values. If for all $\mathbb{E}$ and $\sigma$, $\mathbb{E}[M\sigma]\Downarrow^{n} \Rightarrow \mathbb{E}[N\sigma]\Downarrow^{\leq n}$ then $M \trianglerighteq N$.*

The proof of the context lemma is a simple extension of an established technique and we omit it here. A detailed proof of the context lemma for strong improvement for call-by-need can be found in [MS99]. With the help of the Context Lemma the proof of Lemma 5 is immediate by the virtue of the fact that computation is defined as reduction in evaluation contexts.

The next step in our proof is a claim that a tick cannot change the asymptotic behaviour of a term.

[2] An alternative would be to define tick to be an identity function. But due to the call-by-value model, this would not give ${}^{\checkmark}M \longmapsto M$.

$$\text{Value}\frac{}{V \Downarrow^0 V} \qquad (\delta)\frac{M \Downarrow^{n_0} f \qquad N \Downarrow^{n_1} b}{M\,N \Downarrow^{n_0+n_1+1} \delta(f,b)}$$

$$(\beta_v)\frac{M \Downarrow^{n_0} \lambda x.L \qquad N \Downarrow^{n_1} V \qquad L[{}^{V}\!/\!_{x}] \Downarrow^{n_2} W}{M\,N \Downarrow^{n_0+n_1+n_2+1} W}$$

**Fig. 1.** The big-step cost semantics

**Lemma 7.** ${}^{\checkmark}M \underset{\approx}{\diamond} M$

The intuitive argument for why this holds is that the execution of $\mathbb{C}[{}^{\checkmark}M]$ differs from the execution of $\mathbb{C}[M]$ only in some interleaving steps due to the tick, and these steps are dominated by the other steps. This is because the tick cannot stack up syntacticly during the computation, and as a consequence there is always a bound to the number of consecutive tick steps, and each such group can be associated to a "proper" reduction.

## 5.3 The Main Result

We will soon turn our attention to a rigorous proof of Lemma 7 but let us first show how our main result follows from Lemma 5 and 7.

**Theorem 1.**

$$=_v \subseteq \underset{\approx}{\diamond}.$$

PROOF. Assume that $M \mapsto N$. We can then make the following calculation

$$M \underset{\sim}{\diamond} {}^{\checkmark}N \underset{\approx}{\diamond} N$$

Since $\underset{\sim}{\diamond}$ is contained in $\underset{\approx}{\diamond}$ it follows by transitivity that $M \underset{\approx}{\diamond} N$, so $\mapsto$ is contained in $\underset{\approx}{\diamond}$. Since $\underset{\approx}{\diamond}$ is a congruence, we have that $=_v$, the congruent closure of $\mapsto$, is also contained in $\underset{\approx}{\diamond}$. □

In the remainder of this section we will prove Lemma 7. It turns out that the proof is more conveniently carried out with the big-step semantics provided in Figure 1, which is the standard call-by-value big-step semantics augmented with a cost measure. The proof that the cost measure in the big-step semantics is in agreement with the small-step semantics is a trivial extension to the standard proof relating big-step and small-step call-by-value semantics.

Recall that a key point in the informal argument of why Lemma 7 holds is that ticks cannot "stack up" on top of each other during computation. To make this into a rigorous argument we introduce the ternary relation $M \overset{i\checkmark}{\succ} N, i \in \mathbb{N}$ defined in Figure 2 which intuitively means that $M$ can be transformed into $N$ by removing blocks of up to $i$ consective ticks. A key property of the relation is that it satisfies the following substitution lemma.

$$\frac{}{{}^{j\checkmark}f \overset{i\checkmark}{\succ} f}\; j \le i \qquad \frac{}{{}^{j\checkmark}b \overset{i\checkmark}{\succ} b}\; j \le i \qquad \frac{}{{}^{j\checkmark}x \overset{i\checkmark}{\succ} x}\; j \le i$$

$$\frac{M \overset{i\checkmark}{\succ} N}{{}^{j\checkmark}\lambda x.M \overset{i\checkmark}{\succ} \lambda x.N}\; j \le i \qquad \frac{M_0 \overset{i\checkmark}{\succ} N_0 \qquad M_1 \overset{i\checkmark}{\succ} N_1}{{}^{j\checkmark}(M_0\,M_1) \overset{i\checkmark}{\succ} (N_0\,N_1)}\; j \le i$$

**Fig. 2.** The tick erasure relation

**Lemma 8.** *If $M \overset{i\checkmark}{\succ} N$ and $V \overset{i\checkmark}{\succ} W$ then $M[{}^{V}\!/\!_{x}] \overset{i\checkmark}{\succ} N[{}^{W}\!/\!_{x}]$*

The lemma is easily proven by, for example, an induction over the structure of $N$.

The next step is to show that $\overset{i\checkmark}{\succ}$ is preserved by computation, and at the same time we show that the cost of executing the tick-decorated term is within a constant factor of the cost of executing the term without the ticks.

**Lemma 9.** *If $M \overset{i\checkmark}{\succ} N$ and $N \Downarrow^{n} W$ then there exists $V$ such that*

- *$M \Downarrow^{m} V$,*
- *$m \le (3i+1)n + i$ and*
- *$V \overset{i\checkmark}{\succ} W$.*

Proof. The proof is by well-founded induction over $n$. We will only consider the case when $N \equiv N_0\,N_1$. Then the derivation of $N \Downarrow^{n} W$ must be of the form

$$(\beta_v)\frac{N_0 \Downarrow^{n_0} \lambda x.N_2 \qquad N_1 \Downarrow^{n_1} W' \qquad N_2[{}^{W'}\!/\!_{x}] \Downarrow^{n_2} W}{N_0\,N_1 \Downarrow^{n_0+n_1+n_2+1} W}$$

where $n = n_0 + n_1 + n_2 + 1$. From $M \overset{i\checkmark}{\succ} N$ we know that $M$ must be of the form ${}^{j\checkmark}(M_0\,M_1)$, for some $j \le i$ and that $M_0 \overset{i\checkmark}{\succ} N_0$ and $M_1 \overset{i\checkmark}{\succ} N_1$. Thus it follows by two applications of the induction hypothesis that

- $M_0 \Downarrow^{m_0} \lambda x.M_2$, for some $m_0 \le (3i+1)n_0 + i$,
- $\lambda x.M_2 \overset{i\checkmark}{\succ} \lambda x.N_2$,
- $M_1 \Downarrow^{m_1} V'$ for some $m_1 \le (3i+1)n_1 + i$, and
- $V' \overset{i\checkmark}{\succ} W'$.

From $\lambda x.M_2 \overset{i\checkmark}{\succ} \lambda x.N_2$ and $V' \overset{i\checkmark}{\succ} W'$ it follows by Lemma 8 that $M_2[{}^{V'}\!/\!_{x}] \overset{i\checkmark}{\succ} N_2[{}^{W'}\!/\!_{x}]$ so we can apply the induction hypothesis a third time which gives that

- $M_2[{}^{V'}\!/\!_{x}] \Downarrow^{m_2} V$ where $m_2 \le (3i+1)n_2 + i$, and
- $V \overset{i\checkmark}{\succ} W$.

We can now construct a derivation of $M_0\, M_1 \Downarrow^{m_0+m_1+m_2+1} V$ as

$$\frac{M_0 \Downarrow^{m_0} \lambda x.M_2 \qquad M_1 \Downarrow^{m_1} V' \qquad M_2[V'/x] \Downarrow^{m_2} V}{M_0\, M_1 \Downarrow^{m_0+m_1+m_2+1} V}$$

which gives that ${}^{j\checkmark}(M_0\, M_1) \Downarrow^{m_0+m_1+m_2+1+j} V$. We complete the proof with a calculation which shows that $m_0 + m_1 + m_2 + 1 + j \leq (3i+1)n + i$.

$$\begin{array}{ll} & m_0 + m_1 + m_2 + 1 + j \\ \leq & ((3i+1)n_0 + i) + ((3i+1)n_1 + i) + ((3i+1)n_2 + i) + 1 + i \\ = & (3i+1)(n_0 + n_1 + n_2 + 1) + i \\ = & (3i+1)n + i \end{array}$$

□

Finally we prove Lemma 7, i.e., that ${}^{\checkmark}M \underset{\approx}{\overset{\triangleleft\!\triangleright}{}} M$

Proof. We will start with the straightforward proof that ${}^{\checkmark}M \underset{\approx}{\triangleright} M$ and do the proof of $M \underset{\approx}{\triangleright} {}^{\checkmark}M$ thereafter. This direction is intuitively obvious since we remove the cost due to the tick. It is also easy to prove because it follows directly from the Context Lemma that ${}^{\checkmark}M \underset{\sim}{\triangleright} M$ and thus ${}^{\checkmark}M \underset{\approx}{\triangleright} M$.

Let us turn to the other direction. Assume that $\mathbb{C}[M]$ is closed and that $\mathbb{C}[M]{\Downarrow^n}$. Clearly $\mathbb{C}[{}^{\checkmark}M] \overset{1\checkmark}{\succsim} \mathbb{C}[M]$ so it follows by Lemma 9 that $\mathbb{C}[{}^{\checkmark}M]{\Downarrow^{\leq 4n+1}}$ as required. □

## 6 Call-by-Name

A natural question to ask is whether our result carries over to other programming languages and their respective calculi. In other words, are calculi which fulfil Plotkin's criteria with respect to their intended programming language limited to linear speed-ups (or slow-downs)?

It turns out that the answer to this question is no, not in general. Here we show, somewhat surprisingly, that the main result fails for call-by-name if we take number of reductions as the measure of computation cost: full beta conversion can lead to asymptotic improvements (and worsenings) for programs in a programming language with normal order reduction. We show this with an example program for which a single beta reduction achieves a superlinear speedup.

We also discuss whether the number of reductions is a reasonable cost measure by comparing it to "the natural" abstract machine. It turns out – another surprise – that to count reductions steps in the reduction semantics is *not* in agreement (within a program size dependent constant factor) with a naïve version of the abstract machine. However a small optimisation of the abstract machine achieves an efficient implementation of the reduction semantics and thus we can justify that to count reductions is a cost measure that is implementable.

### 6.1 The $\boldsymbol{\lambda}_{name}$ Calculus

We begin by introducing the syntax and operational semantics. We will work with the same syntax as for the call-by-value language. The calculus $=_{name}$ is the congruent closure of the basic reductions:

$$\begin{aligned} f\,b &\mapsto_{name} \delta(f,a) && (\delta)\\ (\lambda x.M)\,N &\mapsto_{name} M[{}^{N}\!/_{x}] && (\beta)\end{aligned}$$

To specify the computation order we define the following reduction contexts:

$$\text{Call-by-name Evaluation Contexts } \mathbb{E} ::= [\cdot] \mid \mathbb{E}\,M \mid f\,\mathbb{E}$$

Then normal order reduction is just:

$$\mathbb{E}[M] \longmapsto_{name} \mathbb{E}[N] \text{ iff } M \mapsto_{name} N.$$

### 6.2 The $\boldsymbol{\lambda}_{name}$ Calculus and Superlinear Speedups

**Theorem 2.** *Beta-reduction can yield a superlinear speedup with respect to call-by-name computation.*

We will sketch the proof of this surprising proposition. We show that the removal of *a single tick* can lead to a superlinear speedup. I.e.,

$$M \mathrel{\underset{\approx_k}{\not\trianglerighteq}} {}^{\checkmark}M$$

Since ${}^{\checkmark}M =_{name} M$ this shows that we cannot have $=_{name} \subseteq \mathrel{\underset{\approx}{\Diamond}}_{name}$.

So, how can the cost of a single tick dominate the cost of all other steps in the computation? To construct an example where this happens, we consider the proof for call-by-value. The point at which the call-by-value proof fails in the call-by-name setting is the substitution lemma for $M \overset{i\checkmark}{\succsim} N$. In a call-by-name setting we cannot restrict ourselves to substitutions of values for variables as in Lemma 8. Instead we would need that,

If $M_0 \overset{i\checkmark}{\succsim} N_0$ and $M_1 \overset{i\checkmark}{\succsim} N_1$ then $M_0[{}^{M_1}\!/_{x}] \overset{i\checkmark}{\succsim} N_0[{}^{N_1}\!/_{x}]$

which clearly fails. For example ${}^{\checkmark}x \overset{1\checkmark}{\succsim} x$ and ${}^{\checkmark}M \overset{1\checkmark}{\succsim} M$ but ${}^{\checkmark\checkmark}M \overset{1\checkmark}{\succsim} M$ is not true because two ticks are nested on top of each other. Thus, ticks can stack up on top of each other during normal order computation (consider for example $(\lambda x.{}^{\checkmark}x)\,({}^{\checkmark}M)$). We will use this when we construct our counterexample.

The idea of the counterexample is that the computation first builds up a term with $n$ ticks nested on top of each other. The term is then passed to a function which uses its argument $m$ times. Since we are using a call-by-name language the argument will be recomputed each time so the total time will be $\mathcal{O}(nm)$. Let $\mathbb{C}_{m,n}$ denote the family of contexts given by

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x) \\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\ (^{\checkmark}x) \\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1 \\
\longmapsto^{*}_{name} \\
(\lambda f.\lambda x.f\,(^{\checkmark}x))\,((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\ldots((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x))\ldots))\,1 \\
\longmapsto_{name} \\
(\lambda x.(\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\ldots((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x))\ldots)\,(^{\checkmark}x))\,1 \\
\longmapsto_{name} \\
(\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\ldots((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x))\ldots)\,(^{\checkmark}1) \\
\longmapsto_{name} \\
(\lambda x.(\ldots((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x))\ldots)\,(^{\checkmark}x))\ \ (^{\checkmark}1) \\
\longmapsto_{name} \\
(\ldots((\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x))\ldots)\,(^{\checkmark\checkmark}1) \\
\longmapsto_{name} \\
\ldots \\
\longmapsto_{name} \\
(\lambda f.\lambda x.f\,(^{\checkmark}x))\,(\lambda x.x + x + \cdots + x + x)\,(^{\cdots\checkmark\checkmark}1) \\
\longmapsto_{name} \\
(\lambda x.(\lambda x.x + x + \cdots + x + x)\,(^{\checkmark}x))\ \ (^{\cdots\checkmark\checkmark}1) \\
\longmapsto_{name} \\
(\lambda x.x + x + \cdots + x + x)\,(^{\checkmark\cdots\checkmark\checkmark}1) \\
\longmapsto_{name} \\
\overbrace{\underbrace{(^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \underbrace{(^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \cdots + \underbrace{(^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \underbrace{(^{\checkmark\cdots\checkmark\checkmark}1)}_{n}}^{m}
\end{array}
$$

**Fig. 3.** A transition sequence where a tick stacks up

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x) \\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\ [\cdot] \\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1
\end{array}
$$

where we have used $\mathsf{let}\ x = M\ \mathsf{in}\ N$ as a short hand for $(\lambda x.N)\,M$.

The terms which exhibit the "stacking" of tick are $\mathbb{C}_{m,n}[^{\checkmark}x]$. Note that this tick then occurs in the definition of *apply*. When the program is executed this single tick builds up as shown by the reduction sequence in Figure 3. The term with the ticks is then duplicated when passed to $g$ and to compute the resulting sum takes $\mathcal{O}(nm)$ time, where $n$ is the number of calls to *apply* in the original term and $m$ is the number of occurrences of $x$ in $g$. If the tick is removed from the definition of apply then the ticks cannot build up and the resulting program runs in $\mathcal{O}(n + m)$ time. Thus we have a family of contexts, namely $\mathbb{C}_{m,n}$ which illustrate that ${}^{\checkmark}x \mathrel{\underset{\approx}{\trianglelefteq}} x$ cannot hold, since we can make $n$ and $m$ sufficiently large to defeat any constants which attempt to bound the difference in cost.

### 6.3 An Abstract Machine

Here we discuss whether counting reduction steps it is a reasonable cost model for call-by-name. We are not interested in whether we can do *better* than call-by-name (call-by-need and more "optimal" forms of reduction can do this), but whether there is an implementation of the language such that the number of reduction steps is a good model.

We start by introducing "the natural" heap based abstract machine, obtained by a minimal modification of the call-by-value machine. It is very similar to the call-by-value machine but the heap is now a mapping from heap variables to terms rather than a mapping from heap variables to values. The transitions of the machine are:

$$
\begin{array}{lrrl lrrl l}
\langle \Gamma, & M\,N, & S\rangle & \leadsto_{name} & \langle \Gamma, & M, & [\cdot]\,N : S\rangle & \\
\langle \Gamma, & f, & [\cdot]\,N : S\rangle & \leadsto_{name} & \langle \Gamma, & N, & f\,[\cdot] : S\rangle & \\
\langle \Gamma\{p{=}N\}, & p, & S\rangle & \leadsto_{name} & \langle \Gamma\{p{=}N\}, & N, & S\rangle & \text{(lookup)} \\
\langle \Gamma, & b, & f\,[\cdot] : S\rangle & \leadsto_{name} & \langle \Gamma, & \delta(f,a), & S\rangle & (\delta) \\
\langle \Gamma, & \lambda x.M, & [\cdot]\,N : S\rangle & \leadsto_{name} & \langle \Gamma\{p{=}N\}, & M[{}^{p}\!/\!_{x}], & S\rangle & (\beta)
\end{array}
$$

The machine is still semi-compositional and it is thus easy to argue that the individual steps are implementable within a program size dependent constant factor. But is it a good implementation of our cost measure to count reduction steps? The answer is no, since the abstract machine may use arbitrarily more steps:

**Lemma 10.** *For every linear function $h$ there is a program $M$ which requires $n$ reductions and $m$ abstract machine steps where $m > |M| \cdot h(n)$.*

The problem with the abstract machine is that it may create chains in the heap of the following form.

$$\Gamma\{p = p_1, p_1 = p_2, \ldots, p_{n-1} = p_n, p_n = M\}$$

To evaluate $p$ means to follow the chain by doing $n$ consecutive lookups. If $p$ is evaluated repeatedly then these lookup steps may dominate all other steps. To see how this can happen consider the following family of programs (based on our earlier example):

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x) \\
\mathsf{in}\ \mathsf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x \\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1
\end{array}
$$

When we evaluate the term in this prototype abstract machine it builds up a chain

$$\Gamma\{p = p_1, p_1 = p_2, \ldots, p_{n-1} = p_n, p_n = 1\}$$

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\,(g\,x)\\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\,x\\
\quad\mathsf{in}\ \underbrace{(\mathit{apply}\circ\cdots\circ\mathit{apply})}_{n}\ (\lambda x.\overbrace{x+x+\cdots+x+x}^{m})\ 1\\
\longmapsto^{*}_{name}\\
(\lambda f.\lambda x.f\,x)\,((\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x))\ldots))\,1\\
\longmapsto_{name}\\
(\lambda x.(\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x))\ldots)\,x)\,1\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x))\ldots)\,1\\
\longmapsto_{name}\\
(\lambda x.(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x))\ldots)\,x)\ 1\\
\longmapsto_{name}\\
(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x))\ldots)\,1\\
\longmapsto_{name}\\
\ldots\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\,x)\,(\lambda x.x+x+\cdots+x+x)\,1\\
\longmapsto_{name}\\
(\lambda x.(\lambda x.x+x+\cdots+x+x)\,x)\ 1\\
\longmapsto_{name}\\
(\lambda x.x+x+\cdots+x+x)\,1\\
\longmapsto_{name}\\
\overbrace{1+1+\cdots+1+1}^{m}
\end{array}
$$

**Fig. 4.** A transition sequence

of length $n$; eventually $p$ is substituted for $x$ in the body of

$$\lambda x.\overbrace{x+x+\cdots+x+x}^{m}.$$

and the chain is then traversed once for every $p$ in

$$\overbrace{p+p+\cdots+p+p}^{m}.$$

Thus it takes (at least) $\mathcal{O}(nm)$ steps to evaluate a program in this family with the abstract machine. However if we turn to the reduction semantics it uses only $\mathcal{O}(n+m)$ as we can see by the schematic reduction sequence in Figure 4. This family of programs is enough to show that for every linear function $h$ there is a program $M$ which requires $n$ reductions and $m$ abstract machine steps where $m > h(n)$. It is not enough to prove Lemma 10 since the size of the programs in the family grows linearly in $n$ and $m$. However it is possible to construct a family of terms which grows logarithmically with $n$ and $m$ which uses a logarithmic encoding of natural numbers and recursion to achieve the same phenomena. We omit the details.

### 6.4 An Optimised Abstract Machine

To obtain an abstract machine that correctly "models" beta reduction we must eliminate the possibility of pointer chains in the heap. We do so by replacing the $(\beta)$ rule with an optimised form $(\beta')$:

$$\langle \Gamma,\ \lambda x.M,\ [\cdot]\,N : S\,\rangle \leadsto_{name} \begin{cases} \langle \Gamma,\ M[{}^{N}\!/\!_{x}],\ S\,\rangle & \text{if } N \text{ is a heap variable} \\ \langle \Gamma\{p{=}N\},\ M[{}^{p}\!/\!_{x}],\ S\,\rangle & \text{otherwise} \end{cases}$$

The key is that the machine now maintains the invariant that for each binding $p{=}N$ in the heap, $N$ is not a heap variable. This ensures that consecutive lookup steps are never performed, and it leads to the following result.

**Lemma 11.** *If $M{\Downarrow}^n$ then there exist $\Gamma$ and $V$ such that*

$$\langle\, \emptyset,\ M,\ \epsilon\,\rangle \leadsto^{\leq 6n}_{name} \langle\, \Gamma,\ V,\ \epsilon\,\rangle$$

The proof of Lemma 11 is very similar to the call-by-value case. First we show that the number of reduction steps in the small-step semantics is the same as the number of abstract machine transitions which use one of the rules $(\beta)$ or $(\delta)$.

**Lemma 12.** *If $M{\Downarrow}^n$ then there exist $\Gamma$ and $V$ such that*

$$\langle\, \emptyset,\ M,\ \epsilon\,\rangle \leadsto^{*}_{name} \langle\, \Gamma,\ V,\ \epsilon\,\rangle$$

*using exactly $n$ applications of $(\beta)$ and $(\delta)$*

The proof of this lemma is by standard techniques relating small-step semantics and abstract machines and is omitted. Next we claim that the total number of abstract machine transitions is at most six times the number of $(\beta)$ and $(\delta)$ transitions.

**Lemma 13.** *If $\langle\, \emptyset,\ M,\ \epsilon\,\rangle \leadsto^{m}_{name} \langle\, \Gamma,\ V,\ \epsilon\,\rangle$ using $n$ applications of $(\beta)$ and $(\delta)$ then $m \leq 6n$.*

Taken together with Lemma 12 this immediately implies Lemma 11.

Just as in the call-by-value case we generalise Lemma 13 to configurations with a non empty heap and stack.

**Lemma 14.** *If $\langle\, \Gamma,\ M,\ S\,\rangle \leadsto^{m} \langle\, \Gamma',\ V,\ \epsilon\,\rangle$ using $n$ applications of $(\beta)$ and $(\delta)$ then $m \leq 6n - \lceil M \rceil - \lceil S \rceil + 1$.*

Proof. The measure $\lceil\cdot\rceil$ is the same as that used in the call-by-value case and the proof is very similar. We only consider the case for the lookup rule where we use the invariant property of the heap. Then $\langle \Gamma,\ M,\ S\rangle \equiv \langle \Gamma'\{p = N\},\ p,\ S\rangle$ for some $p$ and $N$. From the induction hypothesis we know that

$$6n - \lceil N\rceil - \lceil S\rceil + 1 \geq m - 1.$$

The required result then follows by the following calculation where the last step uses the fact that $N$ is not a heap variable and thus $\lceil N\rceil = 1$

$$\begin{aligned} 6n - \lceil M\rceil - \lceil S\rceil + 1 &= 6n - \lceil p\rceil - \lceil S\rceil + 1 \\ &= 6n - \lceil S\rceil + 1 \\ &= 6n - \lceil N\rceil - \lceil [\cdot]S\rceil + 1 + \lceil N\rceil \\ &\geq m - 1 + \lceil N\rceil \\ &= m \end{aligned}$$

□

## 7 Conclusions

We have shown that the core theory of call-by-value functional languages cannot speed up (or slow down) programs by more than a constant factor, and we have stated that the result also holds for call-by-need calculi. In conclusion we reflect on the robustness of this result.

### 7.1 Machine Variations

The results are clearly dependent on particular implementations. Implementation optimisations which themselves can yield nonlinear speedups can turn innocent transformations into nonlinear speedups or slowdowns. This is illustreded in the call-by-name case. In order to build a machine which matches the beta reduction we were forced to include an optimisation. But the syntactic nature of the optimisation made it very fragile: simply by adding a "tick" to a subterm we can turn off the optimisation and thus get asymptotic slowdown. If we had taken the more naïve implementation model for call-by-name as the basis[3] then the call-by-value result also holds for this variant of call-by-name.

[3] A corresponding high level semantics for the naïve implementation can be obtained by using a modification of beta reduction:

$$(\lambda x.M)\,N \mapsto_{name'} M[{}^{\checkmark}N/_{x}]$$

### 7.2 Language Extensions

An obvious question is whether the result is robust under language extension. For this to be a sensible question one must first ask whether the theory itself is *sound* in richer languages. Fortunately Felleisen *et al* [FH92] have shown that the $\lambda_v$ calculus is sound for quite a number of language extensions including state and control. Given the rather direct operation nature of our proofs (they do not rely on global properties of the language, such as static typing) we have no reason to believe that the same result does not hold in richer languages.

But of course language extensions bring new basic reductions. We believe that the result will also be easy to extend to include basic reductions corresponding to state manipulations – but we have not proved this. It is also straightforward to extend the theory with specific (and useful laws). For example, the following context-distribution law is a strong cost equivalence:

$$\mathbb{E}[\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N] \cong \mathsf{if}\ L\ \mathsf{then}\ \mathbb{E}[M]\ \mathsf{else}\ \mathbb{E}[N]$$

and thus the result is sound if we extend the theory with this rule[4] – or any other rule which is a weak cost equivalence (it need not be a strong cost equivalence as in this example).

We stated the main result for call-by-need calculi. What about other reasonable calculi? The core of the proof rests on the simple cost equivalence ${}^{\checkmark}M \mathrel{\overset{\triangleleft\triangleright}{\approx}} M$. Although we at first thought that this would hold for *any* reasonable programming calculus, the counterexample for call-by-name shows otherwise. As further work one might consider whether it holds for e.g., object calculi [AC96].

### 7.3 Theory Extensions: A Program Transformation Perspective

Where do speedups come from? It is folklore that many program transformation techniques cannot produce superlinear speedups. A good example is partial evaluation (except perhaps in rather pathological cases, such as discarding computations by using full beta-reduction), which, viewed abstractly as a source to source transformation, employs little more than basic reduction rules.

One "simple" source of non constant-factor speedups is the avoidance of repeated computation via common subexpression elimination. For example, the transformation rule:

$$\mathsf{let}\ x = M\ \mathsf{in}\ \mathbb{C}[M] \rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ \mathbb{C}[x]$$

(where $\mathbb{C}$ does is assumed not to capture free variables in $M$) is sound for call-by-value, and can achieve non-constant factor speedups. As an example, consider its application to a recursive definition:

$$\begin{aligned} f\ x = {} & \mathsf{if}\ x = 0\ \mathsf{then}\ 1 \\ & \quad \mathsf{else\ let}\ z = f\ (x-1)\ \mathsf{in}\ z + f\ (x-1) \end{aligned}$$

[4] In the language we have defined, the conditional must be encoded using a strict primitive function, but a suitable version of this rule is easily encoded.

Note that in order to achieve non-constant-factor speedups the subexpression must be a non value (otherwise any instance of the rule is provable in $\lambda_v$). It is for this reason that the memoisation in classical partial evaluation or deforestation does *not* achieve more than a constant factor speedup – because the memoisation there typically shares *functions* rather than arbitrary terms.

Further work along these lines would be to extend the constant factor result to a richer class of transformation systems which include forms of memoisation. This would enable us to, or example, answer the question posed in [Jon90] concerning partial evaluation and superlinear speedups.

**Acknowledgements**

Thanks to Koen Claessen and John Hughes for spotting problems with the naïve implementation model for call-by-name. Thanks also to Dave Schmidt for providing valuable feedback on an earlier draft.

## References

[AB97] Z. M. Ariola and S. Blom, *Cyclic lambda calculi*, Proc. TACS'97, LNCS, vol. 1281, Springer-Verlag, February 1997, pp. 77–106.

[AC96] M. Abadi and L. Cardelli, *A theory of objects*, Springer-Verlag, New York, 1996.

[AFM$^+$95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler, *A call-by-need lambda calculus*, Proc. POPL'95, the 22$^{nd}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, January 1995, pp. 233–246.

[AG92] Lars Andersen and Carsten Gomard, *Speedup analysis in partial evaluation: Preliminary results*, Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (San Francisco, U.S.A.), Association for Computing Machinery, June 1992, pp. 1–7.

[BK83] Gerard Boudol and Laurent Kott, *Recursion induction principle revisited*, Theoretical Computer Science **22** (1983), 135–173.

[FH92] Matthias Felleisen and Robert Hieb, *A revised report on the syntactic theories of sequential control and state*, Theoretical Computer Science **103** (1992), no. 2, 235–271.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial evaluation and automatic program generation*, Prentice Hall International, International Series in Computer Science, June 1993, ISBN number 0-13-020249-5 (pbk).

[Jon90] Neil D. Jones, *Partial evaluation, self-application and types*, Automata, Languages and Programming, LNCS, vol. 443, Springer-Verlag, 1990, pp. 639–659.

[Jon93] N. D. Jones, *Constant time factors* do *matter*, STOC '93. Symposium on Theory of Computing (Steven Homer, ed.), ACM Press, 1993, pp. 602–611.

[Jon96] Neil D. Jones, *What not to do when writing an interpreter for specialisation*, Partial Evaluation (Olivier Danvy, Robert Glück, and Peter Thiemann, eds.), Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, 1996, pp. 216–237.

[Lan66] P. J. Landin, *The next 700 programming languages*, Communications of the ACM **9** (1966), no. 3, 157–164.

[Mil77] R. Milner, *Fully abstract models of the typed λ-calculus*, Theoretical Computer Science **4** (1977), 1–22.

[MS99] Andrew Moran and David Sands, *Improvement in a lazy context: An operational theory for call-by-need*, Proc. POPL'99, ACM Press, January 1999, pp. 43–56.

[Plo76] G. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science **1** (1976), no. 1, 125–159.

[Ros98] Eva Rose, *Linear-time hierarchies for a functional language machine model*, Science of Computer Programming **32** (1998), no. 1–3, 109–143, 6th European Symposium on Programming (Linköping, 1996).

[San96] D. Sands, *Proving the correctness of recursion-based automatic program transformations*, Theoretical Computer Science **167** (1996), no. A.

[Zhu94] Hong Zhu, *How powerful are folding/unfolding transformations?*, Journal of Functional Programming **4** (1994), no. 1, 89–112.

# Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software

Bruno Blanchet[1], Patrick Cousot[1], Radhia Cousot[2], Jérôme Feret[1], Laurent Mauborgne[1], Antoine Miné[1], David Monniaux[1], and Xavier Rival[1]

[1] CNRS & École normale supérieure, 75005 Paris, France
[2] CNRS & École polytechnique, 91128 Palaiseau cedex, France

**Abstract.** We report on a successful preliminary experience in the design and implementation of a special-purpose Abstract Interpretation based static program analyzer for the verification of safety critical embedded real-time software. The analyzer is both precise (zero false alarm in the considered experiment) and efficient (less than one minute of analysis for 10,000 lines of code). Even if it is based on a simple interval analysis, many features have been added to obtain the desired precision: expansion of small arrays, widening with several thresholds, loop unrolling, trace partitioning, relations between loop counters and other variables. The efficiency of the tool mainly comes from a clever representation of abstract environments based on balanced binary search trees.

*Dedicated to Neil Jones, for his 60th birthday.*

## 1 Introduction

### 1.1 General-Purpose Static Program Analyzers

The objective of static program analysis is to automatically determine run-time properties, statically, that is, at compile-time. This problem is in general undecidable, so static program analysis relies on approximations as formalized by Abstract Interpretation [8,9]. For example, typable programs do not go wrong but untypable programs do not all go wrong.

In many contexts, such as program transformation, the uncertainty induced by the approximation is acceptable. For example, interval analysis can be used to eliminate useless array bound checks at run-time [7]. The selection rate (i.e. the proportion of potential alarms that are definitively solved either positively or negatively) is often between 80 and 95%, so the optimization is worthwhile. Moreover, it is correct since the remaining 5 to 20% cases for which tests cannot be eliminated at compile-time will be checked at run-time. Sometimes, static program analysis can discover definite errors at compile-time (e.g. uninitialized variables), which is useful for debugging. The objectives of such *general-purpose static program analyzers* are:

1. to fully handle a general purpose programming language (such as Ada or C, including the standard libraries);
2. to require no user-provided specification/annotation (except maybe light ones such as, e.g., ranges of input variables or stubs for library functions the source of which is not available);
3. to be precise enough to provide interesting information for most programs (e.g. information pertinent to static program manipulation or debugging);
4. to be efficient enough to handle very large programs (from a cost of a few megabytes and minutes to gigabytes and hours when dealing with hundreds of thousands of source code lines).

Such general-purpose static program analyzers are very difficult to design because of the complexity of modern programming languages and systems. Some are commercially available and have had successes in repairing failures or preventing them in the early development of programs. Since the coverage is 100%, the false alarms can be handled, e.g., by classical testing methods, thus reducing the need for actual test of absence of run-time error by an economically significant factor of 80 to 95%.

## 1.2 Program Verification

In the context of safety critical real-time software as found, e.g., in the transportation industry, run-time checking of errors may not be acceptable at all. Hence, the debugging cost is very high and significantly higher than the software development cost itself. In this particular industrial context where correctness is required by safety and criticality, rigorous formal methods should be directly applicable and economically affordable.

**Deductive Methods.** In practice, deductive methods (see, for example, [1,17]) are hard to apply when lacking a formal specification and when the program size is very large. Indeed, the cost for developing the specification and the proof, even with a proof assistant or a theorem prover, is in general much higher than the cost for developing and testing of the program itself (figures of 600 person-years for $80,000$ lines of C code have been reported). Only critical parts of the software can be checked formally and errors appear elsewhere (e.g. at interfaces). Moreover, for embedded software with a lifetime of ten to twenty years, both the program and its proof have to be maintained over that long period of time.

**Software Model Checking.** Software model checking (see, for example, [12]) is also hard to apply when lacking a formal specification and when the program size is very large. This is because the cost of developing both the specification and the model of the program can also be very large. Problems such as the difficulty to provide sensible temporal specifications or the state explosion are well-known. On one hand, if the model is not proved correct, then the program correctness check is not rigorous and mostly amounts to debugging. On the other hand, the model can be proved correct, either manually or using deductive methods, but it is then logically equivalent to the correctness proof of the original program [6] and consequently requires an immense effort. Moreover, the program model and its correctness proof have to be maintained with the program itself, which may be a significant additional cost. Finally, abstraction is often required; in this case, software model checking essentially boils down to static program analysis.

**Static Program Analysis.** Static program analyzers prove program properties by effectively computing an abstract semantics of programs expressed in fixpoint or constraint form. The collected abstract information can then be used as a basis for program manipulation, checking, or partial or total verification. Depending on the considered classes of programs and abstract properties, several types of static program analyzers can be designed, all founded on the Abstract Interpretation theory [5].

### 1.3 On the Use of General-Purpose Static Program Analyzers

General-purpose static program analyzers require no human intervention and, hence, are very cheap to use, in particular during the initial testing phase and, later, during the program maintenance and modification. However, even for simple specifications, they are hardly useful for formal verification because of their execution time (which is required to get precision but may prevent a routine compiler-like utilization during the initial program development process) and the residual false alarms (excluding full verification). A selection rate of 95%—which is very high when considering a general-purpose static program analyzer—means that a significant part of the code still needs to be inspected manually, which remains a prohibitive cost or, if bypassed, is incompatible with severe safety requirements. Moreover, if the analysis time is of several hours, if not days, the refinement of the analysis, e.g., by inserting correctness assertions, is a very slow process and will not, in general, eliminate all false alarms because of the inherent approximations wired in the analyzer.

### 1.4 Special-Purpose Static Program Analyzers

Because of undecidability, automaticity, and efficiency requirements, the absence of false alarm can only be achieved by restricting both the considered classes of specifications and programs. This leads to the idea of *special-purpose static program analyzers*. Their objectives are:

1. to handle a restricted family of programs (usually not using the full complexity of modern programming languages);
2. to handle a restricted class of general-purpose specifications (without user intervention except, maybe, light ones such as, e.g., ranges of input or volatile variables);

3. to be precise enough to eliminate all false alarms (maybe through a redesign or, better, a convenient parameterization of the analyzer by a trained end-user that does not need to be a static analysis or Abstract Interpretation specialist);
4. to be efficient enough to handle very large programs (a cost of a few megabytes and minutes for hundreds of thousands of source code lines).

By handling a family of programs and not only a single program or model of the program, we cope with the evolution over years and the economic cost-effectiveness problems. By restricting the considered class of specifications and, more precisely, considering general-purpose requirements (such as absence of run-time error or unexpected interrupt), we avoid the costly development of specific specifications and can apply the analyzer on legacy software (e.g. decades-old applications the initial specification of which, if any, has not been maintained over time). Moreover, the trade-off between analysis precision and cost can be carefully balanced by the choice of appropriate and reusable abstractions.

### 1.5 Report on a First Experience on the Design of a Special-Purpose Static Program Analyzer

In this paper, we report on a first experience on the design of a special-purpose static program analyzer. The considered class of software is critical real-time synchronous embedded software written in a subset of C. The considered class of specifications is that of absence of run-time error. This experience report explains the crucial design decisions and dead-ends that lead from a too imprecise and too slow initial implementation of a general-purpose static program analyzer to a completely rewritten, very fast, and extremely precise special-purpose static program analyzer. By providing details on the design and implementation of this static analyzer as well as on its precision (absence of false alarm), execution time, and memory usage, we prove that the approach is technically and economically viable.

## 2 The Special Purpose of Our Analyzer

Because of its critical aspect, the class of software analyzed in this first experience was developed through a rigorous process. In this process, the software is first described using schemata. These schemata are automatically translated into a C code using handwritten macros which compute basic functionalities. This C code, organized in many source files, is the input of the analyzer.

Because of the real-time aspect of the application, the global structure of the software consists in an initialization phase followed by a big global synchronized loop. Because of this structure, nearly all variables in the program depend on each other.

Because of the way the C code is generated, the program contains a lot of global and static variables, roughly linear in the length of the code (about

$1,300$ for $10,000$ LOCs[1]). It follows that memory space cannot be saved by a preliminary analysis of the locality of the program data.

### 2.1 Restrictions to C Followed by the Software Class

Fortunately, the strong requirements enforced by the development of critical software imply that some difficult aspects of the C language are not used in this class of software. First, there are neither **goto**s nor recursive function calls. The data structures are quite simple: the software does not manipulate recursive data structures, and the only pointers are statically allocated arrays (no dynamic memory allocation). There is no pointer arithmetic except the basic array operations. Because the code does not contain strings either, alias information is trivial.

### 2.2 Specification to Be Verified

The analyzer has to prove the following:

– absence of out-of-bound array indexes;
– logical correctness of integer and floating-point arithmetic operations (essentially, absence of overflow, of division by 0).

So, the analysis consists in over-approximating the set of reachable states.

## 3 Program Concrete Semantics

The program concrete semantics is a mathematical formalization of the actual execution of the program. A precise definition is necessary to define and prove the soundness of the verifier, checker, or analyzer. For example, in static analysis, the analyzer effectively computes an abstract semantics which is a safe approximation of this concrete semantics. So, the rigorous definition of the program concrete semantics is mandatory for all formal methods.

In practice, the concrete semantics is defined by:

– the ISO/IEC 9899 standard for the C programming language [14] as well as the ANSI/IEEE 754 standard for floating-point arithmetic [2];
– the compiler and machine implementation of these standards;
– the end-user expectations.

Each semantics is a refinement of the previous one where some non-determinism is resolved.

[1] The number of lines of code (LOCs) is counted with the Unix™ command `wc -l` after stripping comments out and macro preprocessing. Then, the abstractions we consider essentially conserve all variables and LOCs, see Sec. 8.

## 3.1 The C Standard Semantics

The C standard semantics is often nondeterministic in order to account for different implementations. Here are three examples:

*unspecified behaviors* are behaviors where the standard provides two or more possibilities and imposes no further requirement on which should be chosen. An example of unspecified behavior is the order in which the arguments to a function are evaluated;
*undefined behaviors* correspond to unportable or erroneous program constructs on which no requirement is imposed. An example of undefined behavior is the behavior on integer overflow;
*implementation-defined behaviors* are unspecified behaviors where each implementation documents how the choice is made. An example of implementation-defined behavior is the number of bits in an **int** or the propagation of the high-order bit when a signed integer is right-shifted.

A static analyzer based on the standard C semantics would be sound/correct for all possible conformant compilers. The approach seems unrealistic since the worst-case assumptions to be made by the concrete semantics are not always easy to imagine in case no requirement is imposed and will anyway lead to huge losses of precision, hence, to unacceptably many false alarms. For instance, the C standard does not impose the sizes and precisions of the various arithmetic types, only some minimal sizes, thus the analysis would be very imprecise in case of suspected overflows.

## 3.2 The Implementation Semantics

A correct compiler for a given machine will implement a refinement of the standard semantics by choosing among the allowed behaviors during the execution. To achieve precision, the design of a static analyzer will have to take into account behaviors which are unspecified (or even undefined) in the norm but are perfectly predictable for a given compiler and a given machine (provided the machine is predictable). For example:

*unspecified behaviors:* the arguments to a function are evaluated left to right;
*undefined behaviors:* integer overflow is impossible because of modulo arithmetic (division and modulo by zero are the only possible integer run-time errors);
*implementation-defined behaviors:* there are 32 bits in an **int** and the high-order bit is copied when right-shifting a signed integer.

## 3.3 The End-User Semantics

The end-user may have in mind a semantics which is a refinement of the implementation semantics. Examples are:

*initialization to zero* which is to be performed by the system before launching the program (whereas the C standard requires this for static variables only);

*volatile variables* for using interface hardware can be assigned a range, so that reads from these variables always return a value in the specified range;

*integer arithmetic computations* which are subject to overflow (since they represent integer bounded quantities for which modulo arithmetic is meaningless) or not (such as shift operations to extract fields of words on interface hardware for which overflow is meaningless).

For meaningful analysis results, one has to distinguish between cases where the execution of the program proceeds or not after hitting undefined or implementation-defined behaviors. In the former case, we take into account the implementation-defined execution; in the latter, we consider the trace to be interrupted. Let us take two examples:

- In a context where $\mathtt{x} \in [0, \mathtt{maxint}]$ is an unsigned integer variable, the analysis of the assignment $\mathtt{y} := 1/\mathtt{x}$ will signal a logical error in case $\mathtt{x} = 0$. In the considered implementation, integer divisions by zero always generate a system exception that aborts the normal execution of the program. Hence we consider that the execution can only go on when there is no run-time error with $\mathtt{y} \in [1/\mathtt{maxint}, 1]$. In that case, the implementation and intended concrete semantics do coincide;
- In a context where $\mathtt{x} \in [0, \mathtt{maxint}]$ is an integer variable, the analysis of the assignment $\mathtt{y} := \mathtt{x} + 1$ will signal a logical error in case $\mathtt{x} = \mathtt{maxint}$. Since the implementation does not signal any error, the end-user can consider the logical error as a simple warning and choose to go on according to several possible concrete semantics:

  *Implementation concrete semantics:* from an implementation point of view, the execution goes on in all cases $\mathtt{x} \in [0, \mathtt{maxint}]$, that is with $\mathtt{y} \in \{-\mathtt{maxint} - 1\} \cup [1, \mathtt{maxint}]$ (since with modulo arithmetic, the implementation does not signal the potential logical error).

  This choice may cause the later analysis to be polluted by the logically infeasible cases ($\mathtt{y} = -\mathtt{maxint} - 1$ in our example). Such a behavior is in fact intentional in certain parts of the program (such as to extract fields of unsigned integers to select volatile quantities provided by the hardware which is logically correct with wrapping);

  *Logical concrete semantics:* from a purely logical point of view, the execution goes on with error-free cases $\mathtt{x} \in [0, \mathtt{maxint} - 1]$, that is with $\mathtt{y} \in [1, \mathtt{maxint}]$ (as if the implementation had signaled the logical error).

  One can think that this point of view would be implementation correct for error-free programs (assuming programs will not be run until all logical warnings are shown to be impossible). This is not the case if the programmer makes some explicit use of the hardware characteristics (such as modulo arithmetic). For example, the correctness of some program constructs (such as field extraction) relies on the absence of overflow in modulo arithmetic and, so, ignoring this fact would lead to the erroneous conclusion that the subsequent program points are unreachable!

Because some constructs (such as signed integer arithmetic) require to take a logical concrete semantics and others (such as field extraction from unsigned integers) require to explicitly rely on the implementation concrete semantics, the analyzer has to be parameterized so as to leave the final choice to the end-user (who can indicate to the analyzer which semantics is intended through a configuration file, for example on a per-type and per-operator basis).

## 4 Preliminary Manipulations of the Program

To reduce the later cost of the static analysis, we perform a few preliminary manipulations of the program. Since the program uses C macros and the semantics of macros in C is not always clear, macros are expanded before the analysis, so the analyzed program is the pre-processed program. Then, all input files are gathered into a single source file. Because the program is automatically generated, it has numerous symbolic constants, so, a classical constant propagation is performed. Note that floating-point constants must be evaluated with the same rounding mode as at run-time, in general to the nearest, whereas during the analysis, interval operations will always be over-approximated: we consider the worst-case assumptions for the rounding mode, to make sure that the computed interval is larger than the actual one. The constant propagation is extended to the partial evaluation [13] of constant expressions including, in particular, accesses to constant arrays with a constant index. This was particularly useful for arrays containing indirections to hardware addresses for interfaces.

Other manipulations can be specified in a configuration file. We can specify volatile variables. Volatile variables should in fact be mentioned as such in the source file; however, they are sometimes omitted from the source because the compiler does not optimize memory accesses, so volatile declarations have no effect on the compilation. We can also specify for volatile variables a range that represents, for instance, the value of sensors. The analyzer then makes the assumption that all accesses to these variables return a value in the indicated range. The first manipulation pass inserts the range as a special kind of initializer for the considered variables. The resulting syntax is then an extension of the C syntax that is taken as input by the other phases of the analyzer.

The user can also declare functions to be ignored. These functions are then given an empty code. (If they were not already defined, then they are defined with an empty code. If they were already defined, then their code is removed.) This declaration has two purposes. The first one is to give a code to built-in system calls that do not influence the rest of the behavior of the program. The second one is to help finding the origin of errors detected by the analyzer: ignoring declarations can be used to simplify the program, and see if the analyzer still finds the error in the simplified program. This usage was not intended at the beginning, but it proved useful in practice.

## 5 Structure of the Analyzer

To over-approximate the reachable states of a well-structured program, the analyzer proceeds by induction on the program syntax. Since the number of global variables is large (about $1,300$ and $1,800$ after array expansion, see Sec. 6.3) and the program is large (about $10,000$ LOCs), an abstract environment cannot be maintained at each program point as usual in toy/pedagogical analyzers [4]. Instead, the analysis proceeds by induction on the syntax with one current abstract environment only. Loops are handled by local fixpoint computations with widenings and narrowings. During this iteration, an abstract environment is maintained at the head of the loop only. So, the number of environments which has to be maintained is of the order of the level of nesting of loops. After the fixpoint is reached, an additional iteration is performed so that all runtime errors can be detected even if environments are not recorded at each program point. Nevertheless, special precaution must be taken for implementing these environments efficiently, as discussed below in Sec. 6.2. Moreover, there are only few user-defined procedures and they are not recursive, so they can be handled in a polyvariant way (equivalent to a call by copy).

## 6 Special-Purpose Abstract Domains

### 6.1 Iterative Construction of the Analyzer

We started with classical analyzes (e.g. interval analysis for integer and floating-point arithmetics, handling of arrays by abstraction into a single element, etc.), which, as expected from a preliminary use of a commercial general-purpose analyzer by the end-user, lead to unacceptable analysis times and too many false alarms.

The development of the analyzer then followed cycles of iterative refinements. A version of the analyzer is run, outputting an abstract execution trace as well as a list of the alarms. Each alarm (or, rather, group of related alarms) is then manually inspected with the help of the abstract trace. The goal is to differentiate between legitimate alarms, coming for instance from insufficient specification of the inputs of the program, and false alarms arising from lack of analysis precision. When a lack of precision is detected, its causes must be probed. Once the cause of the loss of precision is understood, refinements for the analysis may be proposed.

Various refinements of the analyzer were related to memory and time efficiency which were improved either by redesign of data structures and algorithms or by selecting coarser abstract domains.

These refinements are reported below. Some are specific to the considered class of programs, but others are of general interest to many analyzers—such as the use of functional maps as discussed in the following section Sec. 6.2.

### 6.2 Efficient Implementation of Abstract Environments through Maps

One of the simplest abstract domains is the domain of intervals [7]: an abstract environment maps each integer or real variable $\mathtt{x} \in V$ to an interval $\mathtt{X} \in I$. The abstract semantics of arithmetic operations are then ordinary interval arithmetic. The least upper bound and widening operations operate point-wise (i.e. for each variable). More generally, we shall consider the case where the abstract environment is a mapping from $V$ to any abstract domain $I$.

A naive implementation of this abstract domain represents abstract environments as arrays of elements of $I$. If destructive updates are allowed in abstract transfer functions (i.e. the environment representing the state before the operation can be discarded), the abstract functions corresponding to assignments are easy to implement; if not, a new array has to be allocated.

For all its simplicity, this approach suffers from two drawbacks:

- it requires many array allocations; this can strain the memory allocation system, although most of the allocated data is short-lived;
- more annoyingly, its complexity on the class of programs we are considering is prohibitive: the cost of a least upper bound operation, which is performed for each test construct in the program, is linear in the number of variables; on the programs we consider, the number of static variables is linear in the length of the program, thus leading to a quadratic cost.

A closer look at the program shows that most least upper bound operations are performed between very similar environments; that is, environments that differ in a small number of variables, corresponding to the updates done in the two branches of a test construct. This suggests a system that somehow represents the similarities between environments and optimizes the least upper bound operation between similar environments.

We decided to implement the mappings from $V$ to $I$ as balanced binary search trees that contain, at each node, the name of a variable and its abstract value. This implementation is provided by the functional map module Map of Objective Caml [15]. The access time to the environment for reading or updating an element is logarithmic with respect to the number of variables (whereas arrays, for instance, would yield a constant time access).

A salient point is that all the operations are then performed fully functionally (no side effect) with a large sharing between the data structures describing different related environments. The functional nature allows for straightforward programming in the analyzer—no need to keep track of data structures that may or may not be overwritten—and the sharing keeps the memory usage low.

Functional maps also provide a very efficient computation of binary operations between similar environments, when the operation $o : I \times I \to I$ satisfies $\forall x \in I,\ o(x, x) = x$. This is true in particular for the least upper bound and the widening. More precisely, we added the function map2 defined as follows: if $f_1$ and $f_2 : V \to I$ and $o : I \times I \to I$ satisfies $\forall x \in I,\ o(x, x) = x$, then $\mathsf{map2}(o, f_1, f_2) = x \mapsto o(f_1(x), f_2(x))$. This function is implemented by walking recursively both trees representing $f_1$ and $f_2$; when $f_1$ and $f_2$ share a common subtree, the result is the same subtree, which can be returned immediately. The function `map2` has to traverse only the nodes that differ between $f_1$ and $f_2$—which correspond to paths from the root to the modified variables. This strategy leads to a time complexity $\mathcal{O}(m \log n)$ where $m$ the number of *modified* variables between $f_1$ and $f_2$, and $n$ is the total number of variables in the environment ($\log n$ is the maximum length of a path from the root to a variable). When only a few variables in the functional map have different values (for example, when merging two environments after the end of a test), a very large part of the computation can be optimized away thanks to this technique.

In conclusion, functional maps implemented using balanced binary search trees decrease tremendously the practical complexity of the analyzer.

## 6.3 Expansion of Small Arrays

When considering an array variable, one can simply *expand* it in the environment, that is to say, consider one abstract element in $I$ for each index in the array. One can also choose to *smash* the array elements into one abstract element that represent all the possible values for all indices.

When dealing with large arrays, smashing them results in a smaller memory consumption. Transfer functions on smashed arrays are also much more efficient. For example, the assignment `tab[i] := exp` with `i` $\in [0, 99]$ leads to 100 abstract assignments if `tab` is expanded, and only one if `tab` is smashed.

Expanded arrays, however, are much more precise than smashed ones. Not only they can represent heterogeneous arrays—such as arrays of non-zero float elements followed by a zero element that marks the end of the array—but they result in less *weak updates*[2]. For example, if the two-elements array `tab` is initialized to zero, and then assigned by `tab[0] := 1; tab[1] := 1`, smashing the array will result in weak updates that will conclude that `tab[i]` $\in [0, 1]$. The precision gain of expanded arrays is particularly interesting when combined with semantics loop unrolling (see Sec. 6.5).

To address the precision/cost trade-off of smashed vs. expanded arrays, the analyzer is parameterized so that the end-user can specify in a configuration file which arrays should be expanded either by providing an array size bound (arrays of size smaller than the bound are expanded) and/or by enumerating them nominatively.

## 6.4 Staged Widenings with Thresholds

The analyzer is parameterized by a configuration file allowing the user to specify refinements of the abstract domains which are used by the analyzer. An example is the *staged widenings with thresholds.*

[2] A weak update denotes an assignment where some variables may or may not be updated, either because the assigned variable is not uniquely determined by the analyzer, or because the assigned variable is smashed with some others.

The classical widening on intervals is $[a, b] \triangledown [c, d] = [(c < a? -\infty : a), (d > b? +\infty : b)]$[3] [8]. It is known since a long time that interval analysis with this widening is less precise than sign analysis since, e.g., $[2, +\infty] \triangledown [1, +\infty] = [-\infty, +\infty]$ whereas sign analysis would lead to $[0, +\infty]$ (or $[1, +\infty]$ depending on the chosen abstract domain [9]). So, most widenings on intervals use additional thresholds, such as -1, 0 and +1. The analyzer is parameterized by a configuration file allowing the user to specify thresholds of his choice.

The thresholds can be chosen by understanding the origin of the loss of precision. A classical example is ($n$ is a given integer constant):

```
int x;
x := 0;
while x <> n do
  x := x + 1;
end while
```

(whereas writing `x` $< n$ would allow the narrowing to capture the bound $n$ [7,8]). A more subtle example is:

```
volatile boolean b;
int x;
x := 0;
while true do
  if b then
    x := x + 1;
    if x > n then
      x := 0;
    end if
  end if
end while
```

In both cases, a widening at the loop head will extrapolate to $+\infty$ and the later narrowing will not recover the constant $n$ bound within the loop body. This was surprising, since this worked well for the following piece of code:

```
int x;
x := 0;
while true do
  x := x + 1;
  if x > n then
    x := 0;
  end if
end while
```

In the first case however, the test:

```
if x > n then
  x := 0;
end if
```

[3] $(\text{true}?a : b) = a$ whereas $(\text{false}?a : b) = b$.

may not be run at each iteration so once the analyzer over-estimates the range of `x`, it cannot regain precision even with this test. A solution would be to ask for a user hint in the form of an assertion $\mathtt{x} \leq n$. An equivalent hinting strategy is to add the constant $n$ as a widening threshold. In both cases, the widening will not lead to any loss of precision. Another threshold idea, depending on the program, is to add arithmetic, geometric or exponential progressions known to appear in the course of the program computations.

## 6.5 Semantic Loop Unrolling

Although the software we analyze always starts with some initialization code before the main loop, there is still some initialization which is performed during the first iteration of that main loop. The mechanism used in the software is a global boolean variable which is true when the code is in the first iteration of the main loop.

If we try to compute an invariant at the head of the loop and the domain is not relational, then this boolean can contain both the values true and false and we cannot distinguish between the first iteration and the other ones. To solve this problem, we applied semantic loop unrolling.

Semantic loop unrolling consists, given an unrolling factor $n$, in computing the invariants $I_0$ which is the set of possible values before the loop, then $I_k$, $1 \leq k < n$ the set of possible values after exactly $k$ iterations, and finally $J_n$ the set of possible values after $n$ or more iterations. Then, we merge $I_0, \ldots, I_{n-1}, J_n$ in order to get the invariant at the end of the loop. Another point of view is to analyze the loop **while** $B$ **do** $C$ as **if** $B$ **then** (C; **if** $B$ **then** (... **if** $B$ **then** (C; (**while** $B$ **do** $C$))...)). Such a technique is more precise than the classical analysis of while loops when the abstract transfer functions are not fully distributive or when we use widenings.

In our case, the first iteration of the main loop is an initialization phase that behaves very differently than subsequent iterations. Thus, by setting $n = 1$, the invariant $J_n$ is computed taking into account initialized values only so we can get a more precise result and even suppress some false alarms.

## 6.6 Trace Partitioning

The reason why semantic loop unrolling is more precise is that, for each loop unrolling, a new set of values is approximated. So, instead of having one set of values, we have a collection of sets of values which is more precise than their union because we cannot represent this union exactly in the abstract domain. We could be even more precise if we did not merge the collection of sets of values at the end of the loop but later.

Consider, for example, the following algorithm which computes a linear interpolation:

```
t = {-10, -10, 0, 10, 10};
c = {0, 2, 2, 0};
```

```
d = {-20, -20, 0, 20};
i := 0;
while i < 3 and x ≥ t[i+1] do
  i := i+1;
end while
r := (x - t[i]) × c[i] + d[i];
```

The resulting variable `r` ranges in $[-20, 20]$, but if we perform a standard interval analysis the result will be $[\min(-20, 2\mathrm{x}^{-}-40), \max(20, 2\mathrm{x}^{+}+40)]$ (where `x` is in $[\mathrm{x}^{-}, \mathrm{x}^{+}]$). This information is not precise enough because interval analysis is not distributive. It is the case even with semantic loop unrolling because, when we arrive at the statement where `r` is computed, all unrollings are merged and we have lost the relationship between `i` and `x`.

Trace partitioning consists in delaying the usual mergings which might occur in the transfer functions. Such mergings happen at the end of the two branches of an **if**, or at the end of a **while** loop when there is semantic loop unrolling. Control-based trace partitioning was first introduced by [11]. Trace partitioning is more precise for non-distributive abstract domains but can be very expensive as it multiplies the number of environments by 2 for each **if** that is partitioned and by $k$ for each loop unrolled $k$ times. And this is even worse in the case of trace partitioning inside a partitioned loop.

So, we improved [11] techniques to allow the partition to be temporary: the merging is not delayed forever but up to a parameterizable point. It worked well to merge partitions created inside a function just before return points, and partitions created *inside* a loop at the end of the loop. This notion of merging allowed the use of trace partitioning even inside the non-unrolled part of loops. In practice, this technique seems to be a good alternative to the more complex classical reduced cardinal power of [9].

## 6.7 Relation between Variables and Loop Counters

As explained in the beginning of the section, non-relational domains, such as the interval domain, can be efficiently implemented. However, non-relational invariants are sometime not sufficient, even for the purpose of bounding variable values. Consider the following loop:

```
volatile boolean b;
i := 0;
x := 0;
while i < 100 do
  x := x + 1;
  if b then
    x := 0;
  end if
  i := i + 1;
end while
```

In order to discover that $\mathtt{x} < 100$, one must be able to discover the invariant relation $\mathtt{x} \leq \mathtt{i}$. Staged widenings are ineffective here because x is never compared explicitly to 100. Switching to fully relational abstract domains (such as polyhedra, or even linear equality) is clearly impossible due to the tremendous amount of global live variables in our application (in fact, even non-relational domains would be too costly without the representation technique of Sec. 6.2).

Our solution is to consider only relations between a variable and a loop counter $\delta$ (either explicit in a **for** loop or implicit in a **while** loop). We denote by $\Delta$ the interval of the counter $\delta$ ($\Delta$ is either determined by the analysis or specified by the end-user in the configuration file, e.g., because the application is designed to run for a certain maximum amount of time). Instead of mapping each variable x to an interval X, our enhanced invariants map each variable x to three intervals: X, $\mathtt{X}^+$ and $\mathtt{X}^-$ which are, respectively, the possible values for x, for $\mathtt{x}+\delta$, and for $\mathtt{x}-\delta$. When too much information on the interval X is lost (after a widening, for example), $\mathtt{X}^+$, $\mathtt{X}^-$, and $\Delta$ are used to recover some information using a so-called *reduction operator* (see Sec. 6.8 below), which replaces the interval X by the interval $\mathtt{X} \cap (\mathtt{X}^+ - \Delta) \cap (\mathtt{X}^- + \Delta)$. This is a simple abstract way of capturing the evolution of the value of the variables over time (abstracted by the loop counter).

From a practical point of view, this domain is implemented as a non-relational domain using the data-structures of Sec. 6.2. It greatly increases the precision of the analysis for a small speed and memory overhead factor.

## 6.8 Reduction and Its Interaction with Widening

In order to take into account the relations between program variables and the loop counters, we use a reduction operator $\rho$ which is a conservative endomorphism (i.e. such that $\gamma(d) \subseteq \gamma(\rho(d))$). The way this reduction is used has a great impact, not only on accuracy, but also on complexity: on the first hand it is crucial to make the reduction before computing some abstract transfer functions (testing a guard for instance) to gain some precision; on the other hand, the cost of the reduction must not exceed the cost of the abstract transfer function itself.

Our choice was to perform reductions on the fly inside each abstract primitive. This allows us to focus the reduction on the program variables which need to be reduced. It is very simple for unary operators. As for the binary operators, we detect which part of the result must be reduced thanks to the functional map implementation, which leads to a sub-linear implementation of the reduction—which coincides with the amortized cost of abstract transfer functions.

The main problem with this approach is that the reduction may destroy the extrapolation constructed by widening[4]. Usually, the reduction operator cannot be applied directly to the results of a widening. Some solutions already existed, but they were not compatible with our requirement of having a sub-linear implementation of the reduction.

[4] The reader may have a look at the Fig. 3 of [16] to have an illustration of this problem in the context of a relational domain.

To solve this problem, we require non-standard conditions on the reduction: we especially require that there are no cyclic propagation of information between abstract variables. We replace the interval $\mathtt{X}$ by the interval $\mathtt{X} \cap (\mathtt{X}^+ - \Delta) \cap (\mathtt{X}^- + \Delta)$, but we do not propagate the other way round (for instance, we replace neither $\mathtt{X}^+$ by $\mathtt{X}^+ \cap (\mathtt{X} + \Delta)$, nor $\mathtt{X}^-$ by $\mathtt{X}^- \cap (\mathtt{X} - \Delta)$). This allows extrapolation to be first performed on the intervals $\Delta$, $\mathtt{X}^+$, and $\mathtt{X}^-$. Once the iterate of the intervals $\Delta$, $\mathtt{X}^+$, and $\mathtt{X}^-$ have become stable, the extrapolation of the interval $\mathtt{X}$ is not disturbed anymore.

### 6.9 On the Analysis of Floating-Point Arithmetic

A major difficulty of the analysis of floating-point arithmetic is the rounding errors, both in the analyzed semantics and in the analyzer itself. One has to consider that:

– transfer functions should model floating-point arithmetic, that is to say (according to the IEEE standard [2]), infinite-precision real arithmetic followed by a rounding phase;
– abstract operators should be themselves implemented using floating-point arithmetic (for efficiency, arbitrary precision floating-point, rational, and algebraic arithmetics should be prohibited).

In particular, special care has to be taken since most classical mathematical equalities (associativity, distributivity, etc.) no longer hold when the operations are translated into floating-point; it is necessary to know at every point if the quantities dealt with are lower or upper bounds.

Interval arithmetic is relatively easy. Operations on lower bounds have to be rounded towards $-\infty$, operations on upper bounds towards $+\infty$. A complication is added by the use of `float`—IEEE single precision—variables in the analyzed programs: abstract operations on these should be rounded in IEEE single precision arithmetic.

## 7 Dead-Ends

The analyzer went through three successive versions because of dead-ends and to allow for experimentation on the adequate abstractions. In this section, we discuss a number of bad initial design decisions which were corrected in the later versions.

*Syntax.* An initial bad idea was to use a program syntax tailored to the considered class of automatically generated programs. The idea was to syntactically check for potential errors, e.g., in macros. Besides the additional complexity, it was impossible to test the analyzer with simple examples. Finally, expanding the macros and using a standard C grammar with semantic actions to check for local restrictions turned out to be more productive.

*Functional Array Representation of the Environment.* The first versions of the analyzer used Caml arrays to represent abstract environments. As discussed in Sec. 6.2, the idea that $\mathcal{O}(1)$ access time to abstract values of variables makes non-relational analyzes efficient turned to be erroneous.

*Liveness Analysis.* Because of the large number of global variables, liveness analysis was thought to be useful to eliminate useless updates in abstract environments represented as arrays. The gain was in fact negligible. Similar ideas using classical data-flow analysis intermediate representations such as use-definition chains, single static assignment, etc. would probably have also been ineffective. The key idea was to use balanced trees as explained in Sec. 6.2.

*Open Floating-Point Intervals.* The first version of the analyzer used closed and open floating-point intervals. For soundness, the intervals had to be overestimated to take rounding errors into account, as explained in Sec. 6.9, which makes the analysis very complex with no improvement in precision, so, the idea of using open intervals was abandoned.

*Relational Floating-Point Domains.* Most literature consider only relational domains over fields, such as rationals or reals, and do not address the problem of floating-point. With some care, one could design a sound approximation of real arithmetic using floating-point arithmetic: each computation is rounded such that the result is always enlarged, in order to preserve soundness. Then, each abstract floating-point operator can be implemented as an abstract operator on reals, followed by an abstract rounding that simply adds to the result an interval representing the absolute error—or, more precisely, the *ulp* [10]. However, this crude approach of rounding can cause the abstract element to drift at each iteration, which prevents its stabilization using widenings. No satisfying solution has been found yet to address this problem, as well as the time and memory complexity inherent to relational domains, so, they are not used in our prototype.

*Case Analysis.* Case analysis is a classical refinement in static analysis. For example [9, Sec. 10.2] illustrates the reduced cardinal power of abstract domains by a case analysis on a boolean variable, the analysis being split on the true and false cases. Implementations for several variables can be based on BDDs. The same way abstract values can be split according to several concrete values of the variables (such as intervals into sub-intervals). This turned out to be ineffective since the costs can explode exponentially as more splittings are introduced to gain in precision. So, case analysis was ultimately replaced by trace partitioning, as discussed in Sec. 6.6.

*On Prototyping.* The first versions of the analyzer can be understood as initial prototypes to help decide on the abstractions to be used. The complete rewriting of the successive versions by different persons avoided the accumulation of levels, corrections, translations which over time can make large programs tangled and inefficient.

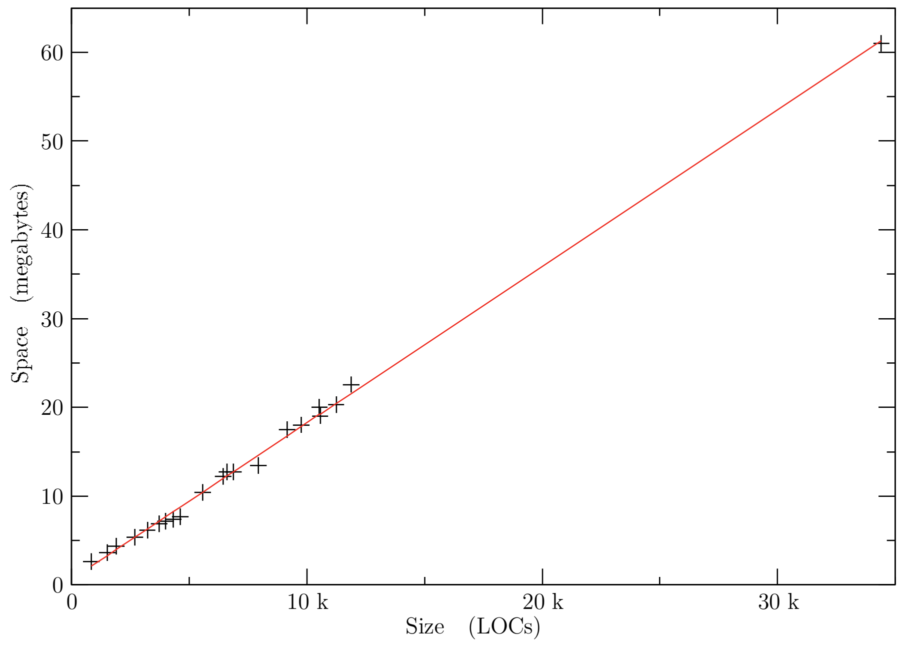


**Fig. 1.** Memory usage as a function of the subject program size.

## 8 Performances

The whole problem of static analysis is to find the right cost-performance balance. In program verification, the precision is fixed to zero (false-)alarm. When the zero alarm precision problem is solved, it remains to estimate the performance of the analysis. To estimate memory and timing performances, we made various analysis experiments with slices of the program as well as the synchronous product of the program several times with itself.

The memory used to store an abstract environment grows linearly with the number of variables, which is itself proportional, for the considered applications, to the size of the program itself $\ell$. Due to nested loops, loop unrolling, and trace partitioning, the analyzer may need to store several abstract environments during the analysis. However, thanks to the use of functional maps, a huge part of these environment is shared, thus reducing the memory consumption. We have found experimentally (Fig. 1[5]) that the peak memory consumption of the analyzer is indeed $\mathcal{O}(\ell)$. For our application, it is of the order of a few megabytes, which is not a problem for modern computers.

[5] The best curve fitting [3] with formula $y = a + bx$ and a tolerance of $10^{-6}$ yields $a = 0.623994$ and $b = 0.00176291$ with association gauged by Chi-square: 6.82461, Correlation coefficient: 0.951324, Regression Mean Square (RMS) per cent error: 0.0721343 and Theil Uncertainty (U) coefficient: 0.0309627.

Thanks to the use of functional maps (Sec. 6.2), the amortized cost of the elementary abstract operations can be estimated to be at most of the order of the time to access abstract values of variables, which is $\mathcal{O}(\ln v)$, where $v$ is the number of program variables. In the programs considered in our experiment, the number of program variables is itself proportional to the number $\ell$ of LOCs. It follows that the cost of the elementary abstract operations is $\mathcal{O}(\ln \ell)$. A fixpoint iteration sweeps over the whole program. Because the abstract analysis of procedures is semantically equivalent to an expansion (Sec. 5), each iteration step of the fixpoint takes $\mathcal{O}(\ell' \times \ln \ell \times p \times i')$ where $\ell'$ is the number of LOCs after procedure expansion, $p$ is a bound to the number of abstract environments that need to be handled at each given program point[6], and $i'$ is a bound to the number of inner fixpoint iterations. The fixpoint computation is then of the order $\mathcal{O}(i \times p \times i' \times \ell' \times \ln \ell)$ where $i$ is a bound to the number of iterations.

We now estimate the bounds $p$, $i$, and $i'$. The number $p$ only depends on end-user parameters. The numbers $i$ and $i'$ are at worst $\mathcal{O}(l \times t)$ where $t$ denotes the number of thresholds, but are constant in practice. So, the execution time is expected to be of the order of $\mathcal{O}(\ell' \times \ln \ell)$. Our hypotheses are confirmed experimentally by best curve fitting [3] the analyzer execution time on various experiments. The fitting formula[7] $y = ax$ yields $a = 0.000136364$, as shown in Fig. 2.

The procedure expansion factor giving $\ell'$ as a function of the program size $\ell$ has also been determined experimentally, see Fig. 3. The best curve fitting with formula[8] $y = a \times x \times (\ln x)^b$ yields $a = 0.927555$, $b = 0.638504$. This shows that, for the considered family of programs, the polyvariant analysis of procedures (equivalent to a call by copy semantics), which is known to be more precise than the monovariant analysis (where all calls are merged together), has a reasonable cost.

By composition, we get that the execution time of the analyzer is $\mathcal{O}(\ell(\ln \ell)^a)$ where $\ell$ is the program size. This is confirmed experimentally by curve fitting the analyzer execution time for various experiments. The non-linear fitting formula[9] $y = a + bx + cx(\ln x)^d$ yields $a = 2.2134 \times 10^{-11}$, $b = 5.16024 \times 10^{-08}$, $c = 0.00015309$ and $d = 1.55729$, see Fig. 4.

The memory and time performances of the analyzer, as extrapolated in Fig. 5, show that extreme precision (no alarm in the experiment) is not incompatible with efficiency. Therefore we can expect such specific static analyzers to be routinely usable for absence of run-time errors verification during the program de-

[6] This number only depends on loop unrolling and trace partitioning.

[7] with association gauged by Chi-square: 239.67, Correlation coefficient: 0.941353, RMS per cent error: 0.515156 and Theil U coefficient: 0.0628226 for a tolerance of $10^{-6}$.

[8] with association gauged by Chi-square: $7.00541 \times 10^{07}$, Correlation coefficient: 0.942645, RMS per cent error: 0.113283 and Theil U coefficient: 0.0464639 at tolerance $10^{-6}$.

[9] with association gauged by Chi-square: 40.1064, Correlation coefficient: 0.956011, RMS per cent error: 0.0595795 and Theil U coefficient: 0.0248341 for a tolerance of $10^{-6}$.

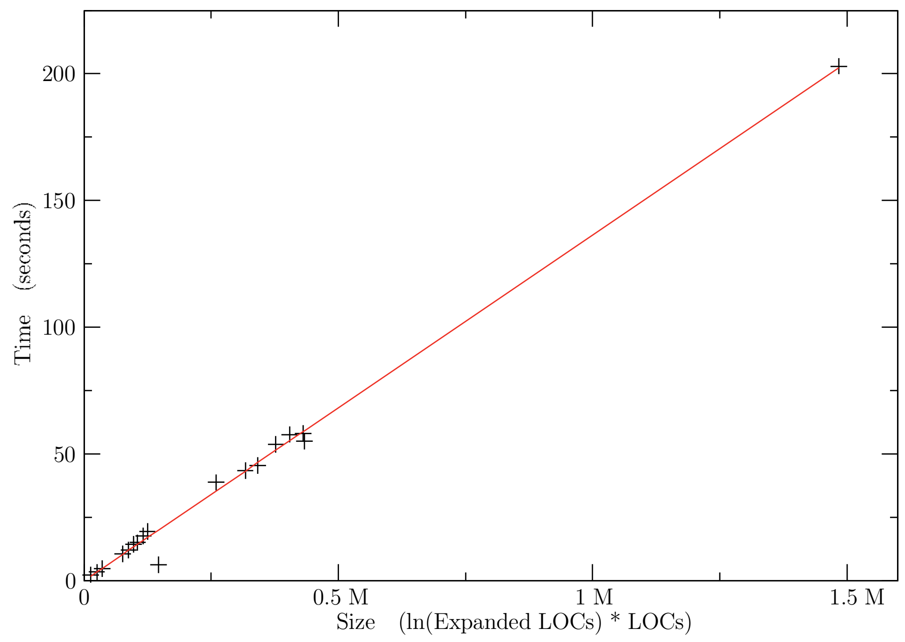


**Fig. 2.** Execution time as a function of $\ell' \times \ln \ell$, where $\ell'$ is the expanded subject program size and $\ell$ its size.

velopment, test, and maintenance processes. Thanks to parameterization, the end-user can easily adjust the analysis to cope with small modifications of the program.

## 9 Conclusion

When first reading the program, we were somewhat pessimistic on the chances of success of the zero false alarm objective since the numerical computations which, not surprisingly for a non-linear control program, represent up to 80% of the program, looked both rather complex and completely incomprehensible for the neophyte. The fact that the code is mostly machine-generated did not help. Using complex numerical domains (such as polyhedral domains) would have been terribly costly. So, the design criterion was always the simpler, i.e., the most abstract, the better, i.e., the most efficient.

Because of undecidability, human hinting is necessary to analyze programs without false alarm:

– in deductive methods this is done by providing inductive arguments (e.g. invariants) as well as hints for the proof strategy;
– in model-checking, this is done by providing the finite model of the program to be checked;

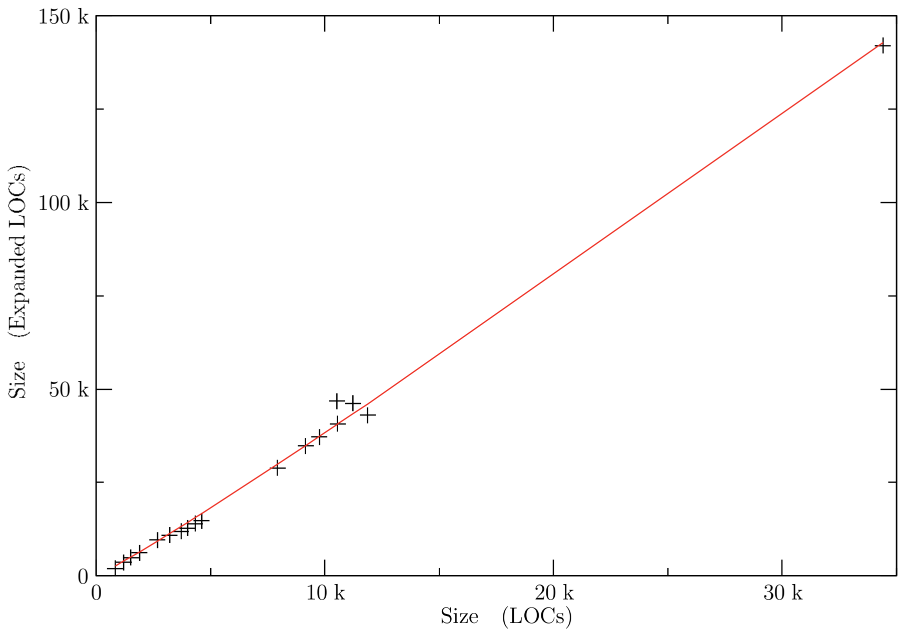


**Fig. 3.** Procedure-expanded program size $\ell'$ as a function of the subject program size $\ell$.

- in static program analysis, we have shown on a non-trivial example that this can be done by providing hints on the local choice of the abstract domains and widenings.

In all cases, some understanding of the verification technique is necessary. We have the feeling that hints to a parameterized analyzer are much easier to provide than correct invariants or program models. Once specialists have designed the domain-specific static analyzer in a parameterized way, the local refinement process is very easy to specify by end-users who are not specialists in static program analysis.

We have serious doubts on the fact that this refinement process can be fully automated. A counter-example based refinement to handle false alarms would certainly be able only to refine abstract domains, abstract element by abstract element, where these abstract elements directly refer to concrete values. In such an approach, the size of the refined analysis would grow exponentially. Clearly, a non-obvious inference step and a significant rewriting of the analyzer are required to move from examples to abstraction techniques such as partitioning or the relational domain handling the loop counters.

So, our approach to get zero false alarm was to design a special purpose analyzer which is parameterized to allow for casual end-users to choose for the specific refinements which must be applied for any program in the considered family.

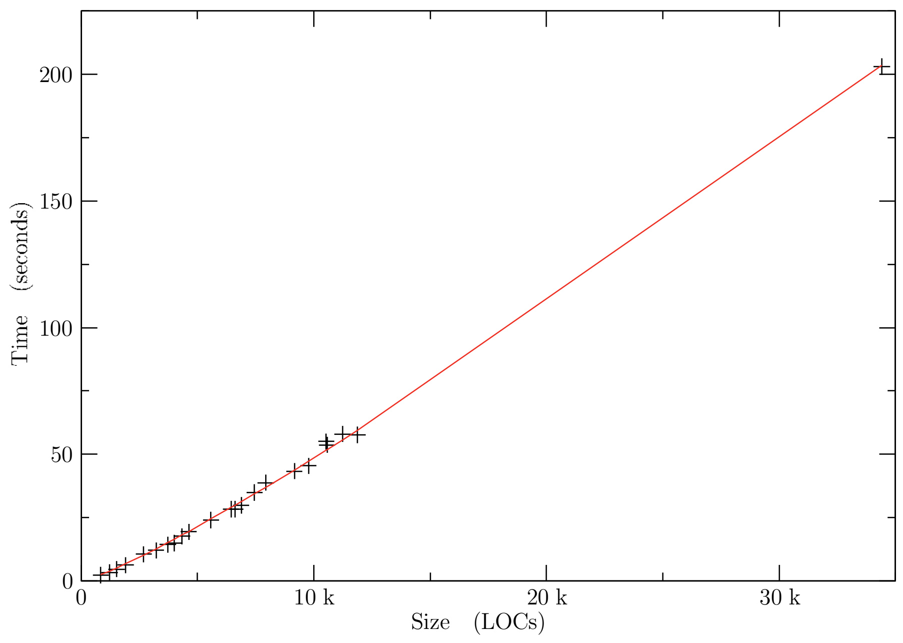


**Fig. 4.** Execution time as a function of the subject program size.

The project is now going on with real-life much larger programs (over 250,000 LOCs) . The resource usage estimates of Figures 1 and 5 were confirmed. Not surprisingly, false alarms showed up since the floating-point numerical computations in these programs are much involved than in the reported first experimentation. A new refinement cycle is therefore restarted to design appropriate abstract domains which are definitely necessary to reach the zero alarm objective at a low analysis cost.

## References

1. J.-R. Abrial. On B. In D. Bert, editor, *Proc. 2nd Int. B Conf. , B'98: Recent Advances in the Development and Use of the B Method*, Montpellier, FR, LNCS 1393, pages 1–8. Springer-Verlag, 22–24 Apr. 1998.
2. American National Standards Institute, Inc. IEEE standard for binary floating-point arithmetic. Technical Report 754-1985, ANSI/IEEE, 1985. `http://grouper.ieee.org/groups/754/`.
3. P. R. Bevington and D. K. Robinson. *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill, 1992.
4. P. Cousot. The Marktoberdorf'98 generic abstract interpreter. `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml`, Nov. 1998.
5. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2000.

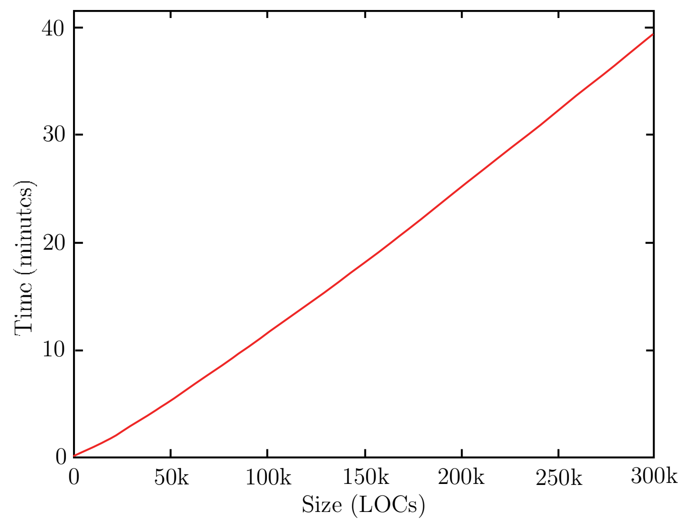


**Fig. 5.** Extrapolated execution time as a function of the subject program size ($y = 2.2134 \times 10^{-11} + 5.16024 \times 10^{-8} \times x + 0.00015309 \times x \times (\ln x)^{1.55729}$).

6. P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 1–25. Springer-Verlag, 26–29 Jul. 2000.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130. Dunod, 1976.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.
10. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.
11. M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 200–214. Springer-Verlag, 1998.
12. G.J. Holzmann. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Int. Conf. CAV '2002*, Copenhagen, DK, LNCS 2404, pages 1–16. Springer-Verlag, 27–31 Jul. 2002.
13. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Int. Series in Computer Science. Prentice-Hall, June 1993.
14. JTC 1/SC 22. Programming languages — C. Technical report, ISO/IEC 9899:1999, 16 Dec. 1999.
15. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual (release 3.04). Technical report, INRIA, Rocquencourt, FR, 10 Dec. 2001. `http://caml.inria.fr/ocaml/`.
16. A. Miné. A new numerical abstract domain based on difference-bound matrices. In 0. Danvy and A. Filinski, editors, *Proc. 2nd Symp. PADO '2001*, Århus, DK,

21–23 May 2001, LNCS 2053, pages 155–172. Springer-Verlag, 2001. `http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf`.

17. S. Owre, N. Shankar, and D.W.J. Stringer-Calvert. PVS: An experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *PROC Applied Formal Methods - FM-Trends'98, International Workshop on Current Trends in Applied Formal Method*, Boppard, DE, LNCS 1641, pages 338–345. Springer-Verlag, 7–9 Oct. 1999.

# Meta-circular Abstract Interpretation in Prolog

Michael Codish[1] and Harald Søndergaard[2]

[1] Dept. of Computer Science, Ben-Gurion Univ. of the Negev, Beer-Sheva, Israel
[2] Dept. of Computer Science and Software Eng., Univ. of Melbourne, Australia

**Abstract.** We give an introduction to the meta-circular approach to the abstract interpretation of logic programs. This approach is particularly useful for prototyping and for introductory classes on abstract interpretation. Using interpreters, students can immediately write, adapt, and experiment with interpreters and working dataflow analysers. We use a simple meta-circular interpreter, based on a "non-ground $T_P$" semantics, as a generic analysis engine. Instantiating the engine is a matter of providing an appropriate domain of approximations, together with definitions of "abstract" unification and disjunction. Small changes of the interpreter let us vary both what can be "observed" by an analyser, and how fixed point computation is done. Amongst the dataflow analyses used to exemplify this approach are a parity analysis, groundness dependency analysis, call patterns, depth-$k$ analysis, and a "pattern" analysis to establish most specific generalisations of calls and success sets.

## 1 Introduction

The seminal work on abstract interpretation [22,23] was presented in the setting of flowchart programs. By the mid 1980s, abstract interpretation had also been established as a powerful tool for the analysis of functional programs [51,1], and in the process, the theory had been popularised. Around that time it was also realised that abstract interpretation could provide a uniform approach to a wide range of program analyses for Prolog. In fact, logic programming became a busy laboratory for abstract interpretation, when it was found that the theory offered a disciplined approach to such diverse problems as the exposure of the independence, aliasing, or freeness of variables, the patterns or types of terms, the calling modes and determinacy of predicates, and the freedom from occur-check errors in logic programs (for example [10,25,38,48,49,56]).

To harness the wide range of analyses, a number of analysis "frameworks" were suggested. These frameworks aimed at isolating what was common to all analyses from the details that were specific to a given analysis. A framework would provide a definition of what abstract interpretation of logic programs was about. It would be parameterised: one would create an instance by providing an "abstract domain" together with definitions of a few (abstract) primitive operations, such as unification and variable projection. The seed for much of this work was a note [36] (now lost) circulated by Neil Jones when an informal workshop on abstract interpretation was organised by Samson Abramsky and Richard Sykes in 1985.

Some frameworks [38,60] were presented using denotational semantics while others [8,49] had an operational semantics flavour. The frameworks turned out to be applicable to the analysis of constraint logic programs as well, and so they became vehicles for defining a truly wide range of program analyses. Importantly, they provided the blueprints for generic Prolog analysis "engines". These were working analysis tools that could be instantiated fairly easily to solve specific analysis problems for full (constraint) logic programming languages. The engines that appeared over the next ten years include PLAI [50], GAIA [40], CHINA [3], the analyser in the CLP(R) compiler by Kelly et al. [39], and Fecht's analyser [29].

In this paper we illustrate a generic approach to the abstract interpretation of Prolog. It is based on meta-circular interpretation. The interpreters and abstract interpreters that we present are working Prolog programs. The method (and applications) that we present continue a line of work pioneered by Codish and Demoen [14]. Codish and Demoen's interpreter provided a concise formalisation of an idea that was current at the time: Bottom-up evaluation could be implemented by repeatedly solving clause bodies (using the facts derived so far) and then asserting the corresponding instances of their heads until the system stabilised.

"Meta-interpreter based" abstract interpretation is easy to explain and understand. It provides the reader with a ready-to-use implementation that is immediately accessible, as opposed to the more sophisticated analysis engines that are typically faster and more general, but also much more complex. Meta-interpreters can be useful starting points for the development of debuggers, tracers, profilers, and they are convenient vehicles for language extensions or for changing aspects of semantics. We argue that they offer also a versatile platform for the specification and development of applications for abstract interpretation.

The meta-circular approach can be particularly successful in the teaching of abstract interpretation, as it encourages immediate hands-on learning, giving students an opportunity to appreciate the strong synergy between the theory and practice of program analysis. Students can learn about semantics and the principles of abstract interpretation (of logic programs) and at the same time follow the development of a platform which they will subsequently use to experiment with different logic program analyses.

In teaching meta-circular interpretation it is common to start with a "vanilla" version and then show how to extend the source language or instrument the semantics, for example to count deduction steps, or to show a trace of the evaluation. These tasks are usually quite easy. The same kind of flexibility and ease of extension carries over to the abstract interpreter. Simple changes in the interpreter can determine the semantics that is being approximated. Relatively simple changes can determine whether the analyser collects information about successful queries, calls that occur at runtime, or bindings that apply at given program points. Moreover, a simple change of the "logic" component of the interpreter module can determine which kind of runtime property the analyser

```
prove([H|Goal]) :-
    my_clause(H,Body),
    prove(Body), prove(Goal).
prove([]).
```

**Fig. 1.** An Interpreter for Logic Programs

will investigate. And finally, the strategy used for finding fixed points can be changed, again quite easily.

Our aim is mainly practical and pedagogical. We focus on the programming aspects of the meta-circular approach and we offer many examples of program analysis. In the next section we outline the approach by giving a simple example of program analysis by meta-circular interpretation. Section 3 lays a foundation. The semantics that we use is presented together with its interpreter, and we discuss "observables" such as success sets and calls. Section 4 gives several examples of abstraction. We show how a range of different abstract domains can be implemented with little effort. In Section 5, we discuss how different strategies for fixed point iteration can be incorporated. Section 6 discusses related work and concludes. All the interpreters discussed in this paper are available via the world-wide web [19].

## 2 Interpretation and Abstraction

Figure 1 illustrates a Prolog program which implements an interpreter for logic programs. This is essentially the classic "vanilla" interpreter for logic programs (see for example Program 17.5 in Sterling and Shapiro [57]) except that queries (conjunctions of goals) are represented as lists of atoms. So a clause $h \leftarrow b_1, \ldots, b_n$ in a program to be interpreted is assumed to be represented by a fact of the form `my_clause(`$h$`,[`$b_1$`,`$\ldots$`,`$b_n$`])` and a unit clause $h \leftarrow true$ is represented as `my_clause(`$h$`,[])`. As a definition of the semantics of logic programs, the interpreter is clearly circular.

Consider now a program that implements Ackermann's function, $ack(i, j)$, using successor notation for the natural numbers ([57], Program 3.9):

```
ack(0,N,s(N)).                ack(s(M),s(N),Res) :-
ack(s(M),0,Res) :-                ack(s(M),N,Res1),
    ack(M,s(0),Res).              ack(M,Res1,Res).
```

We can represent this program as three facts, one per clause:

```
my_clause(ack(0,N,s(N)), []).
my_clause(ack(s(M),0,Res), [ack(M,s(0),Res)]).
my_clause(ack(s(M),s(N),Res), [ack(s(M),N,Res1),ack(M,Res1,Res)]).
```

To make the interpreter easier to use, we can arrange that it reads in the source program, asserting `my_clause` facts in the Prolog database along the way. In the code accompanying this paper [19] this is handled by the predicate `load_file` defined in module `input.pl`. We can then use the interpreter to answer queries:

```
?- prove([ack(s(s(0)),N,s(M))]).
M = s(s(0)),               N = 0 ;
M = s(s(s(s(0)))),         N = s(0) ;
M = s(s(s(s(s(s(0)))))), N = s(s(0))
```

and so on.

For some applications it is useful to make unification explicit in the interpreter. For example, one might want to facilitate changes to the semantics of unification, '`=`', so that it can stand for rational-tree unification or unification with or without the occur-check. For the purposes of dataflow analysis we shall want to approximate the behaviour of unification, so it will simplify our task if we normalise the program prior to its interpretation to make all unifications explicit. This can be done as the program is read by the predicate `load_file`. For example, the second clause for the `ack` predicate will be represented by

```
my_clause(ack(A,B,C),
    [ unify(A,s(M)), unify(B,0), unify(C,Res),
      unify(D,M), unify(E,s(0)), unify(F,Res), ack(D,E,F) ]).
```

In fact, in our approach, unification will be explicit *only* in user clauses like this. In the standard interpretation, `unify` is defined by

```
unify(X,Y) :- X=Y.
```

To support this explicit reference to the unification operation, the interpreter of Figure 1 is enhanced to treat `unify` as a built-in predicate whose definition is not part of the program being interpreted. Thus we add the following clause to the interpreter:

```
prove([unify(A,B)|Goal]) :- unify(A,B), prove(Goal).
```

For a particular analysis we then need to define `unify`, to specify how it should behave for a given domain of abstractions. As an example, let us choose a domain *Parity* for parity analysis, with four elements:

| | |
|---|---|
| `zero` | — denoting $\{0\}$ |
| `one` | — denoting $\{1\}$ |
| `even` | — denoting $\{n \in \mathbb{N} \mid n > 0 \text{ and } n \text{ is even}\}$ |
| `odd` | — denoting $\{n \in \mathbb{N} \mid n > 1 \text{ and } n \text{ is odd}\}$ |

The elements of *Parity* are used to approximate terms: $p(\delta_1, \ldots, \delta_n)$ approximates the atom $p(t_1, \ldots, t_n)$, if $t_i$ is a term in successor notation and $\delta_i \in$ *Parity* describes (contains) the corresponding natural number. Abstract unification for *Parity* is defined in Figure 2. The definition relies on the fact that normalisation introduces calls of the form `unify(X,term)` where `X` is a variable. Hence the first argument to `unify` will be a variable `X`, or, after unifications, an abstract

```
unify(X,Y) :-
    nonvar(Y), Y=s(T),
    unify(S,T),
    (  S=zero, X=one;
       S=one,  X=even;
       S=even, X=odd;
       S=odd,  X=even
    ).
```

```
unify(zero,Y) :-
    Y==0.
unify(X,Y) :-
    (Y==even; Y==odd; Y==zero; Y==one),
    X=Y.
unify(X,Y) :-
    var(Y),
    X=Y.
```

**Fig. 2.** Abstract Unification for the Extended Parity Domain

element. The second argument may be concrete, such as '`0`', a variable `Y`, or it may be an abstract element resulting from a constraint such as `X=Y`. So the definition of `unify(X,Y)` is given by four clauses, corresponding to the possible structures of the second argument which can be: a concrete non-variable term of the form `s(T)`, the concrete constant `0`, one of the abstract elements (`even`, `odd`, `zero` or `one`), or a variable.

The need for `nonvar` (and `var`) in the definition is an artifact of using the so-called non-ground representation for programs. With this representation, care must be taken not to instantiate meta-variables or terms accidentally. Also, we need to ensure that "abstract" symbols are chosen so that they are disjoint from "concrete" symbols.

We can now view our interpreter as a parity analyser. If we look at the set of all answers for an abstract query, we find that they describe correctly the set of all answers to a corresponding concrete query. This is due to the design of the abstract domain and the fact that the unification algorithm correctly mimics the concrete unification of natural numbers with respect to parity information. For example, if we consider the query

```
?- ack(even,one,R).
```

we obtain the answer `R = odd`. Indeed, this reflects the fact that for a positive even number $n$, the Prolog evaluation of the query `ack(n,1,R)` assigns to `R` an odd value, greater than 1. However, the resulting analyser is not entirely satisfactory. The answer `R = odd` will be produced infinitely often and so, strictly, we cannot know whether there is also an answer of the form `R = even`, say.

Instead of considering an interpreter which embodies a top-down semantics, one alternative is to consider an interpreter which corresponds to a bottom-up semantics. Such an interpreter generates new facts incrementally, in the style of the well-known $T_P$ operator [59] or its non-ground version, the $s$-semantics [26]. This type of interpreter does not require the user to specify a particular entry point, or shape of query. Rather it will produce information pertaining to all predicates in one go. This avoids the volatility of the sketched top-down approach.

For Ackermann's function, the success set of the `ack` predicate is clearly infinite, so generation of all ground atomic consequences is not possible in finite time. But the approximation that we considered above will work well and it provides useful information about the success set in finite time.

The interpreters presented later are based on various variants of a $T_P$ semantics (see below). Presenting the Ackermann program to any of the interpreters that approximate success sets yields the following answers (though not necessarily in that order):

```
ack(zero, zero,  one)          ack(even, zero, odd)
ack(zero,  one, even)          ack(even,  one, odd)
ack(zero, even,  odd)          ack(even, even, odd)
ack(zero,  odd, even)          ack(even,  odd, odd)
ack( one, zero, even)          ack( odd, zero, odd)
ack( one,  one,  odd)          ack( odd,  one, odd)
ack( one, even, even)          ack( odd, even, odd)
ack( one,  odd,  odd)          ack( odd,  odd, odd)
```

Note that this output exposes non-trivial parity dependencies amongst the arguments of `ack`. For example, the last column on the right tells us that $ack(i, j)$ is indeed odd (and greater than 1) for all $i > 1$.

## 3 Concrete Semantics and Interpreters

In the previous section we sketched the use of a (standard) meta-circular interpreter for logic programs as a step in obtaining an example analyser for the language. The key point in this approach is that the interpreter plays the role of a semantic definition. By replacing the basic operations in the interpreter with operations on data descriptions, we obtain an abstract interpreter for the language. However we have seen that this approach is not satisfactory. It is similar to the common statement that, intuitively, abstract interpretation is like "executing the program over a domain of data descriptions instead of over the data itself." But the object we want to approximate is not the execution of the program as such, but rather some aspect of its meaning—namely, a semantics which makes observable the property of interest, and preferably little else. So instead of abstracting the operations in a standard meta-interpreter, we should abstract the operations in an interpreter for a suitable concrete semantics.

### 3.1 A Semantics Based on Fixed Points

The $T_P$ "immediate consequences" operator [59] is the best known fixed point characterisation of the meaning of (definite) logic programs. Letting $\mathcal{H}$ denote the set of ground atomic formulas (over an appropriate alphabet), $T_P : \mathcal{P}(\mathcal{H}) \to \mathcal{P}(\mathcal{H})$ is defined by

$$T_P(I) = \left\{ A \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_k \text{ is a ground instance of a clause in } P, \\ \{B_1, \ldots, B_k\} \subseteq I \end{array} \right\}$$

In other words, for a given program $P$ and a set $I$ of ground facts (often called a Herbrand interpretation), $T_P(I)$ is the set of ground facts deducible from $I$

and some clause in $P$. Thus $lfp(T_P)$, the least fixed point of $T_P$, makes $P$'s least Herbrand model observable. In Section 2 we argued that this semantics is a better and simpler basis for abstract interpretation than the usual query-driven interpreter which sometimes requires careful crafting to ensure termination, even for finite abstract domains. Compared to the SLD semantics [42], $T_P$ is more abstract, hiding the resolvents that are created during SLD resolution. If needed, a $T_P$ style semantics can be devised to make a variety of runtime events observable. Moreover, unlike an SLD characterisation, it does not force us to select an entry point in the program, or a particular query instance. Instead it generates information about all predicates.

But the $T_P$ semantics is not without its problems. First, it expresses the meaning of a program relative to an alphabet. The meaning of a program fragment depends on the alphabet, which is determined by the symbols occurring in the larger program, and hence depends on the fragment's context. In this sense, the $T_P$ semantics is not modular. Worse, in the context of the non-ground meta-circular approach, an unintended interference occurs when program and interpreter are combined [47]. For example, let $P$ be the program

```
p(X).
q(a).
```

and let $P'$ denote Figure 1's interpreter together with the clauses

```
my_clause(p(X),[]).
my_clause(q(a),[]).
```

Then $lfp(T_{P'})$ includes `prove([p(q(a))])`, because `q(a)` occurs as a term in $P'$. But `p(q(a))` is not in $lfp(T_P)$, as `q` is not a term constructor in $P$.

Second, almost all the interesting program analyses for logic programs aim at detecting how logic variables are used or instantiated. The $T_P$ semantics, at least in its original form, provides ground atomic formulas, and hence cannot say anything about variables. An early objection to the use of $T_P$ as a basis for program analysis was therefore that, at least in its original form, it is of rather limited use [45,5]. It was later shown that the introduction of a class of "non-term" constants in the Herbrand universe can ameliorate this problem, so that many interesting program properties can be captured [30]. Even so, the $T_P$ semantics is unable to capture properties of the so-called computed answers [42], a set which, in general, is not closed under substitution.

## 3.2 Observing Answers

The $s$-semantics [6,27,28] is a non-ground version of the $T_P$ semantics. The $s$-semantics of a program $P$ is a set of possibly non-ground atoms which characterises both declarative as well as operational properties of a program. In particular: (a) the ground instances of the $s$-semantics gives precisely the least Herbrand model of $P$; and (b) the computed answer substitutions for any initial query $q$ with $P$ are determined by solving $q$ using the (potentially infinite) set of atoms in the $s$-semantics of $P$. Letting $\mathcal{H}_v$ denote the set of atomic formulas, the $s$-semantics of $P$ is defined as the least fixed point of an "immediate consequences" operator $T_P^v : \wp(\mathcal{H}_v) \to \wp(\mathcal{H}_v)$, similar to the standard least Herbrand model semantics. More precisely,[1]

$$T_P^v(I) = \left\{ h\theta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P, \\ \langle a_1, \ldots, a_n \rangle \ll_C I, \\ \theta = mgu(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\}$$

where $\langle a_1, \ldots, a_n \rangle \ll_C I$ expresses that $a_1, \ldots, a_n$ are variants of elements of $I$ renamed apart from $C$ and from each other, and $mgu$ gives the most general unifier of two (sequences of) expressions.

As a basis for designing program analyses, the $s$-semantics has proven particularly useful (see for example [5,12]). This is due, in part, to the fact that it captures both declarative and operational properties, and in part to the simplicity of its definition. The advantages of using a non-ground semantics become more tangible in cases where we wish to *compute* program meanings, as in the case of semantics-based program analysis.

As an example, for the append program,

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

we get as the least fixed point this set of atoms:

$$\{\ \texttt{append}([x_1, \ldots, x_n], ys, [x_1, \ldots, x_n | ys]) \mid n \geq 0\ \}$$

Logically, the variables $x_1, \ldots, x_n$ and $ys$ in these non-ground atoms are considered universally quantified. They obtain values from the underlying term algebra which is determined by the given alphabet. An important advantage of using a non-ground semantics is that the meaning of a program fragment is determined by the symbols it contains, without considering the entire underlying alphabet. This resolves the interference problem in the sense that the meaning $\mathit{lfp}(T_P)$ of a program $P$ contains the atom $p$ if and only if the meaning $\mathit{lfp}(T_{P'})$ of the corresponding meta-program $P'$ contains `prove([`$p$`])` [41,47].

If we consider the instances of the atomic formulas given by the $s$-semantics, we get the set of "correct answers" [42], representing all the logical consequences of $P$. This semantics is often referred to as the $c$-semantics [27] and is also useful as a basis for program analysis.

The fundamental operation in the bottom-up semantics of a program $P$ is the repeated application of facts derived so far, to solve the bodies of $P$'s rules, generating new instances of their heads. This process starts with an empty set of facts and proceeds (sometimes forever) until no new facts are derived. Codish and Demoen [14] captured this idea with the simple meta-circular interpreter shown in Figure 3.

[1] The formal definition involves equivalence classes of atoms (equal up to variable names). We ignore this detail in this presentation.

```
iterate :-
    operator, fail.
iterate :-
    retract(flag),
    iterate.
iterate.

record(F) :- cond_assert(F).

raise_flag :-
    ( flag ->  true
    ;   assert(flag)
    ).

cond_assert(F) :-
    \+ (numbervars(F,0,_), fact(F)),
    assert(fact(F)), raise_flag.
```

**(a) The control**

```
operator :-
    my_clause(H,B),
    prove(B),
    record(H).

prove([B|Bs]) :-
    fact(B),
    prove(Bs).
prove([unify(A,B)|Bs]) :-
    unify(A,B),
    prove(Bs).
prove([]).
```

**(b) The logic**

```
go(File) :-
    load_file(File),
    iterate,
    showfacts.
```

```
showfacts :-
    fact(F),
    numbervars(F,0,_),
    print(F), nl,
    fail ; true.
```

**(c) Using the interpreter**

**Fig. 3.** A Bottom-Up Prolog Interpreter

As for the standard interpreter of Figure 1, a clause $h \leftarrow b_1, \ldots, b_n$ in the program to be interpreted is represented as a fact `my_clause(`$h$`,[`$b_1, \ldots, b_n$`])`. While computing the semantics, whenever a new fact $F$ is derived, it is maintained in the dynamic Prolog database as a fact of the form `fact(`$F$`)`. This simplifies the specification of the interpreter making it easy to implement a mechanism for "solving a clause body using the facts derived so far". In particular, the operation of renaming apart the facts used to solve a clause body is obtained "for free" from the underlying Prolog platform.

The interpreter in Figure 3 is divided conceptually into three components. On the left **(a)** is the "control" component which triggers iteration of an `operator` until no new facts are derived. When a new fact gets inserted into the Prolog database, a `flag` is raised. Iteration stops when `retract(flag)` fails in the second clause for `iterate`, that is, when no new facts were asserted in the previous iteration. On the right **(b)**, the predicate `operator` provides the "logic" and the inner loop of the algorithm which for each `my_clause(`*Head*, *Body*`)` in $P$ proves the *Body* using facts derived so far and calls the predicate `record` which adds the *Head* to the set of facts derived so far, provided it is a new fact. In this version of the interpreter, a fact $F$ is considered new if it is not subsumed by a fact which is already recorded (and hence implements the $c$-semantics). The subsumes check for $F$ is implemented by application of the built-in predicate `numbervars(F,0,_)` which instantiates the variables in F (with fresh constant symbols) and then calling `fact`(F). This should fail if $F$ is to be considered new (and hence does not affect the variables in the recorded fact). The distinction between `record` and `cond_assert` is unnecessary at this point, but it will prove useful when we consider variations of the interpreter. The second clause in the definition of `prove` is for the case when unification is made explicit and should be considered as a built-in predicate by the interpreter.

Bottom-up evaluation is initiated by the query `?- iterate` which leaves the result of the evaluation in the Prolog database. In **(c)** the predicate `go` facilitates the use of the interpreter. This predicate loads the program to be interpreted, initiates iteration and finally prints the derived facts on the screen. Of course this interpreter will only work in case the semantics of the program is finite.

The interpreter in Figure 3 computes a non-ground variation of the $T_P$ semantics of the input program. The $c$-semantics and the $s$-semantics are variations (when `cond_assert` performs a *subsumes* check or a *variant* check, respectively) [6,26]. Both of these semantic variations are widely applied in the context of deductive databases where they are sometimes referred to as the set of *generated consequences* and *irredundant generated consequences* [43].

The practicality of using Prolog's dynamic database to store facts hinges on an efficient indexing mechanism in the underlying Prolog system. Alternatives are: (1) to maintain the set of derived facts using for example 2–3–4 trees, and provide this set as an additional argument in the interpreter; and (2) to use a tabling mechanism [15].

### 3.3 Observing Calls

A serious drawback of bottom-up evaluation is that it focuses only on the *success patterns* of the program (answers). Typically we are interested in how the predicates in the program are called at runtime. Another problem is that we cannot ask for just those facts that are relevant to processing a given query. It may well be that we are only interested in the analysis of the program for a specific set of initial queries, but bottom-up evaluation, as described so far, will produce the entire success set, or its approximation.

The essential idea in most bottom-up methods is to combine a top-down generation of goals with a bottom-up generation of facts. In the more common "bottom-up" approach, this is achieved through a source-to-source program transformation. The magic-sets transformation [4] and other related techniques originate as an optimisation technique in the context of deductive databases. The common principle underlying these techniques is a transformational approach in which a "magic program" $P_q^{\mathcal{M}}$ is derived from a given program $P$ and query $q$. The least model of the derived program is more efficient to compute (bottom-up) and contains the information from the least model of $P$ which is relevant for

```
operator :-                     prove([B|Bs]) :-
    my_clause(H,B),                 record(call(B)), fail.
    fact(call(H)),              prove([B|Bs]) :-
    prove(B),                       fact(ans(B)), prove(Bs).
    record(ans(H)).             prove([]).
```

**Fig. 4.** An Operator for Query-Directed Bottom-Up Evaluation

the query $q$. It is interesting to note that the idea behind the magic program is similar to the motivation for Jones and Mycroft's minimal function graphs [37].

Assuming that the body of a rule is evaluated from left to right as with Prolog's execution strategy, the magic-sets transformation is defined as follows: Let $P$ be a logic program and $q$ an initial query. The corresponding magic program is $P_q^{\mathcal{M}}$. For each $n$-ary predicate symbol $p/n$ in $P$ and $q$, $P_q^{\mathcal{M}}$ will contain two new predicate symbols: $p^c/n$ (capturing calls to $p$) and $p^a/n$ (capturing successes). In the following, if $b$ is an atom $p(t_1, \ldots, t_n)$, we let $b^c$ denote the atom $p^c(t_1, \ldots, t_n)$, and similarly for $b^a$. For each clause $h \leftarrow b_1, \ldots, b_n$ in $P$, $P_q^{\mathcal{M}}$ will contain $n$ clauses of the form

$$b_i^c \leftarrow h^c, b_1^a, \ldots, b_{i-1}^a, \text{ with } i = 1 \ldots n$$

(read: "$b_i$ is a call if $h$ is a call and $b_1, \ldots, b_{i-1}$ are answers") together with the clause

$$h^a \leftarrow h^c, b_1^a, \ldots, b_n^a$$

(read: "$h$ is an answer if $h$ is a call and $b_1, \ldots, b_n$ are answers"). In addition, $P_q^{\mathcal{M}}$ will contain the "seed" fact $q^c$ ("$q$ is a call") corresponding to the initial query $q$.

The magic-sets transformation has proven useful in the context of program analysis because the facts of the form $p^c$ in the (non-ground) least model of a transformed program $P_q^{\mathcal{M}}$ correspond to the *calls* for the predicate of $p$ in the original program which arise in the computations of $q$.

We mention the magic-sets transformation only as a device to explain the interpreter in Figure 4. In fact, no program transformation takes place in our scheme—our approach is simpler. Codish [11] shows how the bottom-up semantics and its interpreter (Figure 3) can be specialised for the case when it is applied to programs generated by the magic-sets transformation, and the result is an interpreter which provides information about the answers and calls which arise in the computations (of a program and a set of initial goals).

Its operator is shown in Figure 4. The control for the interpreter is the same as that shown in Figure 3**(a)** and the interpreter can be used as shown in Figure 3**(c)**, except that we need to add an argument `Query` to the predicate `go` and to make the "seed" assertion `assert(fact(call(Query)))` before calling `iterate`. On the whole, the interpreter is unchanged. Figure 5 shows the result of the query `ack(2,2,A)` (for readability we have avoided successor notation).

```
call(ack(2,2,A))        ans(ack(1,1,3))         ans(ack(2,1,5))
call(ack(2,1,A))        ans(ack(2,0,3))         call(ack(1,5,A))
call(ack(2,0,A))        call(ack(1,3,A))        call(ack(1,4,A))
call(ack(1,1,A))        call(ack(1,2,A))        call(ack(0,5,A))
call(ack(1,0,A))        call(ack(0,3,A))        ans(ack(0,5,6))
call(ack(0,1,A))        ans(ack(0,3,4))         ans(ack(1,4,6))
ans(ack(0,1,2))         ans(ack(1,2,4))         call(ack(0,6,A))
ans(ack(1,0,2))         call(ack(0,4,A))        ans(ack(0,6,7))
call(ack(0,2,A))        ans(ack(0,4,5))         ans(ack(1,5,7))
ans(ack(0,2,3))         ans(ack(1,3,5))         ans(ack(2,2,7))
```

**Fig. 5.** Calls and Answers in the Evaluation of an Ackermann Query

The facts are listed in the order they are derived which, for this example, corresponds to the events (calls and answers) which occur in a standard top-down evaluation.

## 3.4 Observing Program Points

It is often the case that analysis is required to characterise the state of the computation at a given program point. This type of information is more refined than that obtained by characterising the calls to the predicates in the program because the same predicate can be called in different ways from different points in the program.

To support analyses which focus on program points we can alter the procedure which inputs the source program and constructs its internal `my_clause` representation. The fact that $q$ is the $j^{th}$ atom in the $i^{th}$ clause is represented by an atom of the form pp($i$,$j$,$q$). For example, the following program to rotate the elements of a list ([57], page 312):

```
rotate(Xs,Ys) :-                      append([],Ys,Ys).
    append(As,Bs,Xs),                 append([X|Xs],Ys,[X|Zs]) :-
    append(Bs,As,Ys).                     append(Xs,Ys,Zs).
```

is represented internally as:

```
my_clause(rotate(Xs,Ys),[pp(1,1,append(As,Bs,Xs)),pp(1,2,append(Bs,As,
        Ys))]).
my_clause(append([],Ys,Ys),[]).
my_clause(append([X|Xs],Ys,[X|Zs]),[pp(3,1,append(Xs,Ys,Zs))]).
```

The changes to the interpreter are then minimal and shown in Figure 6. Similar solutions are applied in other analysers [31,15].

With the seed assertion `assert(fact(call(0,0,rotate([a,b,c],Ys))))` for an initial query, the result of the interpretation contains the following information for the three calls to `append` in this program at points $(1,1)$, $(1,2)$ and $(3,1)$:

```
operator :-                     prove([pp(I,J,B)|Bs]) :-
    my_clause(Head,Body),           record(call(I,J,B)), fail.
    fact(call(_,_,Head)),       prove([pp(I,J,B)|Bs]) :-
    prove(Body),                    fact(ans(B)), prove(Bs).
    record(ans(Head)).          prove([]).
```

**Fig. 6.** Query-Directed Bottom-Up Evaluation with Program Points

```
unify(X,Y) :-                                iff(true,[]).
   var(Y), X=Y.                              iff(true,[true|Vars]) :-
unify(X,Y) :-                                    iff(true,Vars).
   nonvar(Y),                                iff(false,[false|_]).
   ( (Y == true ; Y == false) ->             iff(false,[_|Vars]) :-
       X=Y                                       iff(false,Vars).
   ;   (term_vars(Y,Vars), iff(X,Vars))
   ).
```

**Fig. 7.** Abstract Unification for Groundness Dependency Analysis

```
call(1,1,append(A,B,[a,b,c]))       call(3,1,append([b,c],[],A))
call(1,2,append([a,b,c],[],A))      call(3,1,append(A,B,[]))
call(1,2,append([b,c],[a],A))       call(3,1,append([c],[],A))
call(1,2,append([c],[a,b],A))       call(3,1,append([c],[a],A))
call(1,2,append([],[a,b,c],A))      call(3,1,append([],[],A))
call(3,1,append(A,B,[b,c]))         call(3,1,append([],[a],A))
call(3,1,append(A,B,[c]))           call(3,1,append([],[a,b],A))
```

## 4 Meta-circular Abstract Interpretation

In the previous sections we illustrated how interpreters for logic programs can be enhanced to make unification explicit. We showed how this approach facilitates the design of program analyses. In order to obtain a dataflow analyser for parity, all we needed to do was replace the definition of the predicate `unify`.

### 4.1 Groundness Analysis

We now show how the same approach yields a groundness dependency analyser. The new definition of `unify` is shown in Figure 7. The atom `term_vars(T,V)` expresses that $V$ is the set of variables in the term $T$. This predicate is a variant of Sicstus Prolog's `term_variables`, adapted to the specific application [19]. The atom `iff`$(X, [Y_1, \ldots, Y_n])$ captures the Boolean function $X \leftrightarrow (Y_1 \wedge \cdots \wedge Y_n)$ (read: "$X$ is ground if and only if $Y_1, \ldots, Y_n$ are").

This analysis is very precise, effectively representing groundness dependencies as Boolean functions. It is pleasing to see how easily such a sophisticated program analysis fits into the meta-circular approach. Abstract unification is not hard to capture, and all the other components of the interpreter remain unchanged.

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

**(a) The append relation**

```
append(true,A,A)
append(false,A,false)
```

**(b) Groundness result for append**

```
append3(A,B,C,D) :-
    append(B,C,Temp),
    append(A,Temp,D).
```

**(c) Appending three lists**

```
call(append3(true,true,true,_))
call(append(true,true,_))
ans(append3(true,true,true,true))
ans(append(true,true,true))
```

**(d) Calls and answers for append3**

```
bubble_sort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    greater_than(X,Y),
    append(As,[Y,X|Bs],Xs1),
    bubble_sort(Xs1,Ys).
bubble_sort(Xs,Xs) :-
    sorted(Xs).
```

**(e) Bubble sort**

```
call(1,1,append(_,_,true))
call(1,3,append(true,true,_))
call(4,1,append(true,true,true))
call(4,1,append(false,true,true))
call(4,1,append(true,true,false))
call(4,1,append(true,false,true))
call(4,1,append(false,false,true))
```

**(f) Calls to append for bubble_sort**

**Fig. 8.** Examples for Groundness Analysis

Consider again the append predicate depicted as Figure 8**(a)**. The groundness analyser yields the two facts given as Figure 8**(b)**. These facts represent a certain Boolean function of three variables, say $(A,B,C)$, with set of models

$$\{(true, true, true), (false, true, false), (true, false, false), (false, false, false)\}$$

that is, $(A \wedge B) \leftrightarrow C$. We can conclude that for any element append$(A, B, C)$ $\in \mathcal{H}_v$ of the program's success set, $C$ is ground (true) if and only if $A$ and $B$ are. Now consider the relation append3 (to concatenate three lists) defined by the clauses in Figure 8**(c)** and consider a query-directed analysis which specifies that the call to append3 is of the form append3(true,true,true,_), indicating that the first three arguments are ground inputs. A query-directed analysis using the operator defined in Figure 4 indicates that during a computation of this query, all calls to append have the first two arguments ground. Notice that there are three places in the program in which there are calls to append (two in the definition of append3 and one in the definition of append itself). The result of the analysis is given as Figure 8**(d)**.

A similar example is the bubble sort program given as Figure 8**(e)**. The two calls to append in this program serve different purposes. The first breaks up a list of the form

$$Xs = [\ \underbrace{\ldots}_{As},\ X, Y,\ \underbrace{\ldots}_{Bs}\ ]$$

to find two elements $X$ and $Y$ that need to be interchanged; the second concatenates the two lists $As$ and $[Y, X|Bs]$ to get back the original list with the positions of $X$ and $Y$ interchanged. If no such $X/Y$ pair is found then the list is sorted.

Performing the query-directed analysis with query `bubble_sort(true,Ys)` (indicating that the first argument is ground) does not give a very useful result, because no distinction is made between the two calls to `append` (or the third call, in the definition of `append` itself). This problem can be remedied by tracking which calls are made at which program points. Assuming that the clauses for `append` occur after the clauses for `bubble_sort` (as clauses `3` and `4`), the result of a call pattern analysis that uses the program point interpreter is given as Figure 8**(f)** (only the results for `append` are shown). Notice that in the recursive call `append(Xs,Ys,Zs)` at point $(4, 1)$, either the first two arguments are ground or the third is (corresponding to $(Xs \wedge Ys) \vee Zs$) while in the two calls of the first clause, at points $(1, 1)$ and $(1, 3)$, the third and first two arguments are ground respectively.

## 4.2 Depth-k Analysis

For some well-known analyses, abstract unification is simply unification, and there is no need to come up with a new definition. We now derive two such analyses, depth-$k$ analysis in this sub-section, and pattern analysis in the next. In both cases, we shall need to redefine how facts are recorded.

For depth-$k$ analysis, the idea is that terms are allowed only a certain maximal depth [55,46]. The depth of a variable or a constant is 1, while otherwise, the depth of a term $t$ can be defined recursively:

$$depth(t) = 1 + max\ \{depth(t') \mid t' \text{ is a proper subterm of } t\}.$$

A term with depth greater than $k$ can be approximated by replacing subterms at that depth by fresh variables. For example, `g(f(f(V)),f(V))` is depth-3 approximated by `g(f(U),f(V))`. (This abstraction in fact falls outside classical abstract interpretation [22], as there is no *best* approximation in general. For example, `g(f(f(V)),f(V))` could equally well be approximated by `g(f(U),U)`, but the two approximations `g(f(U),f(V))` and `g(f(U),U)` are incomparable depth-3 terms, since neither is represented by the other. In terms of abstract interpretation, the former is in its own concretisation, but not in that of the latter, and *vice versa.* Still, depth-$n$ approximation is safe and the analysis is useful.)

Figure 9 shows how `record` is redefined to prune the depth of the terms in a fact before adding the fact to the set of facts derived so far. Observe that the call is changed to `record(K,F)`, so as to pass the desired depth to the analyser and that the approximation is performed by calling `depth(K,T,NewTerm)`

```
record(K,F) :-
    depth(K,F,NewF),
    cond_assert(NewF).

depth_list([],[],_K).
depth_list([T|Ts],[NT|NTs],K) :-
    depth(K,T,NT),
    depth_list(Ts,NTs,K).
```

```
depth(_K,Term,Term) :-
    var(Term), !.
depth(0,_Term,_NewTerm).
depth(K,Term,NewTerm) :-
    K>0,
    Term =.. [F|Args],
    K1 is K-1,
    depth_list(Args,NewArgs,K1),
    NewTerm =.. [F|NewArgs].
```

**Fig. 9.** Recording Facts in Depth-$k$ Analysis

which takes the desired depth $K$ and a term $T$ and produces the pruned version *NewTerm*.[2]

Consider the following program for the Towers of Hanoi problem ([57], Program 3.31). `A-B` indicates that we should move a disk from peg `A` to peg `B`.

```
/*  hanoi(N,A,B,C,Moves) :-
        Moves is a sequence of moves for solving the Towers of
        Hanoi problem with N disks and three pegs, A, B and C.
*/
    hanoi(s(0),A,B,C,[A-B]).
    hanoi(s(N),A,B,C,Moves) :-
        hanoi(N,A,C,B,Ms1),
        hanoi(N,C,B,A,Ms2),
        append(Ms1,[A-B|Ms2],Moves).
```

Here is the result of the analysis using a depth-10 abstraction:

```
hanoi(s(0),A,B,C,[A-B])
hanoi(s(s(0)),A,B,C,[A-C,A-B,C-B])
hanoi(s(s(s(0))),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B])
hanoi(s(s(s(s(0)))),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(0))))),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(0)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(s(s(0)...),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(s(s(0)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(s(s(s(_)...),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(s(s(_)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
```

Notice that the list of moves to solve the problem (the fifth argument) is precise for 3 disks or less. After that we get only the first part of the solution. It is apparent that for an odd number of disks the solution always starts with the move `A-B` and for an even number of disks, with the moves `A-C,A-B,C-B` (though this is not a proof). Notice that so long as the number of disks is not pruned

[2] $=..$ is Prolog's "univ" operator which gives access to the components of a term, including the term constructor: $f(t_1,\ldots,t_n) =.. [f,t_1,\ldots,t_n]$.

```
record(F) :-            keep_msg(F) :-
    keep_msg(F).            \+ (numbervars(F,0,_), fact(F)),
                            functor(F,N,A), functor(F1,N,A),
                            ( retract(fact(F1)) ->
                                term_subsumer(F1,F,MSG)
                            ;   MSG = F
                            ),
                            assert(fact(MSG)),
                            raise_flag.
```

**Fig. 10.** Recording Facts in Pattern Analysis

(and hence we can determine if it is odd or even) there is always a single solution for that number of disks in the fifth argument.

By analysing the program with a combined domain that abstracts numbers using the parity domain and lists using a depth-5 analysis we obtain a *proof* of an interesting property of the Towers of Hanoi problem. Namely, for an odd number of disks, the solution always starts with the move `A-B` and that for an even number of disks, with the moves `A-C,A-B,C-B`. With this domain the analysis gives:

```
hanoi(odd,A,B,C,[A-B])               hanoi(even,A,B,C,[A-C,A-B,C-B])
hanoi(odd,A,B,C,[A-B,A-C,B-C,_|_])   hanoi(even,A,B,C,[A-C,A-B,C-B,_|_])
```

## 4.3 Pattern Analysis

This analysis maintains a single concrete atom as the approximation for each predicate in the program. For a given predicate, any pair of abstract atoms is replaced by their most specific generalisation (using anti-unification [53]). The analysis can be viewed as a generalisation of constant propagation analysis, which uses a flat domain of constants, together with bottom and top elements. With constant propagation, if a variable may assume two different values $c_1$ or $c_2$ then it is approximated by *top*. In the pattern analysis, we have terms, or partial structures, instead of constants. Approximation maintains the information common to the terms approximated, that is, it provides their most specific generalisation. Approximation of the values that a variable may assume can be relaxed repeatedly, each time yielding a more general term. The number of times a value can be relaxed is finite, but cannot be bounded a priori. The abstract domain used here is therefore an example of a domain which has infinite chains, but no infinite ascending chain.

Figure 10 shows how `record` is redefined to yield a pattern analysis. The implementation of `keep_msg(F)` covers three cases ('`msg`' stands for "most specific generalisation"):

1. There is already a pattern at least as general as `F`. In this case the call to `\+ (numbervars(F,0,_), fact(F))` fails.

2. There is no previous pattern for the predicate of `F`. In this case the condition `retract(F1)` fails and `fact(MSG)` with `MSG=F` is asserted.
3. There is a pattern `F1` for the predicate of `F` (not as general as `F`). In this case `fact(MSG)`, with `MSG` the most specific term that generalises both `F1` and `F`, replaces `fact(F1)` in the database. The call to the Sicstus term library predicate `term_subsumer(F1,F,MSG)` computes the most specific term which generalises both `F1` and `F`.

For the goal dependent analyser, we need two specialised versions of `keep_msg(F)` (`keep_msg(call(F)` and `keep_msg(ans(F))`) to maintain most specific patterns for calls and answers respectively.

For example, consider this program which reverses a list ([57], Program 3.16):

```
reverse([],[]).                  append([],Ys,Ys).
reverse([X|Xs],Zs) :-            append([X|Xs],Ys,[X|Zs]) :-
    reverse(Xs,Ys),                  append(Xs,Ys,Zs).
    append(Ys,[X],Zs).
```

Goal dependent pattern analysis with the initial query `reverse(Xs,Ys)` yields:

```
call(reverse(A,B))               ans(reverse(A,B))
call(append(A,[B],C))            ans(append(A,[B],[C|D]))
```

indicating that for any initial query to `reverse`, in all subsequent calls to `append`, the second argument is a singleton list.

One application of pattern analysis is so-called *structure untupling* [18]. This transformation can improve the precision of other program analyses by making invariant data-structures explicit. For an example, consider the following Prolog program ([57], Program 15.3) which reverses a list efficiently, using difference lists.

```
reverse(Xs,Ys) :-                reverse_dl([X|Xs],Ys-Zs) :-
    reverse_dl(Xs,Ys-[]).            reverse_dl(Xs,Ys-[X|Zs]).
                                 reverse_dl([],Xs-Xs).
```

Here `Xs` will be ground if and only if `Ys` is ground, for any successful query `reverse(Xs,Ys)`. However, a groundness dependency analysis can not be expected to pick up this dependency. To do that, it would be necessary to infer the groundness dependency $(\mathtt{A} \wedge \mathtt{C}) \leftrightarrow \mathtt{B}$ for `reverse_dl(A,B-C)`. But groundness analysis typically considers dependencies amongst predicate arguments, not between sub-structures, so the desired information is not produced.

Pattern analysis for this program shows that all answers for `reverse_dl` are in fact of the form `reverse_dl(A,B-C)`. Hence the program can be specialised automatically to obtain

```
reverse(Xs,Ys) :-                reverse([X|Xs],Ys,Zs) :-
    reverse(Xs,Ys,[]).               reverse(Xs,Ys,[X|Zs]).
                                 reverse([],Xs,Xs).
```

Groundness analysis now gives the desired accurate result, for the original program.

## 5 Evaluation Strategies

Until now we have presented interpreters which calculate fixed points under a naive evaluation strategy. More efficient evaluation strategies exist and are well understood. These include semi-naive evaluation [52,58], eager evaluation [11,61], and techniques which take advantage of the structure of strongly connected components in the program's call graph.

For the presentation of the various abstractions, the naive approach is adequate, but it is pleasing to note that variations of the evaluation strategy are easily introduced to the implementation of an analysis by adapting the underlying interpreter.

### 5.1 Semi-naive Evaluation

Semi-naive evaluation is based on the observation that whenever a clause body is solved, in some iteration, at least one new fact, derived "one iteration ago", should be used. Otherwise, the resulting instance of the clause head will not be new. The classic algorithm for semi-naive evaluation (for example as presented by Ullman [58]) maintains the facts derived so far in two sets denoted $\Delta_{new}$ and $\Delta_{old}$. To facilitate the Prolog implementation, a new fact $F$ derived in the $n^{th}$ iteration is maintained in the dynamic Prolog database together with a "time-stamp" in the form `fact(n,F)`, where `n` is an iteration number.

Figure 11 illustrates a Prolog interpreter for semi-naive bottom-up evaluation. The interpreter is essentially the same as that presented in Figure 3. The important difference is that the `operator` predicate on the right carries two arguments, corresponding to the previous and current iteration numbers, and always selects from a clause body a fact derived in the previous iteration, before proving the rest of the body. For semi-naive iteration, evaluation is initiated by the query `iterate(0,1)` which leaves the result of the evaluation in the Prolog database.

Note that (a) to determine whether another iteration is required we check if there is a fact of the form `fact(Curr,_)`, instead of using a flag; (b) when recording a new fact we time stamp it using the additional argument in the predicate `record`; and (c) the definition of the `operator` predicate contains two clauses — one for the first iteration, and one for the subsequent iterations.

### 5.2 Eager Evaluation

Eager evaluation also aims to insure that at least one "new" atom is used when a clause body is solved. The difference between this strategy and the semi-naive strategy is in the book-keeping of "what is new". Eager evaluation uses a newly derived fact $F$ immediately, postponing other tasks while exploring which new facts can be derived now that $F$ has been established. In the process, new facts may be derived, and their investigation will then get higher priority. That is, tasks are handled in a first-in last-out manner.

```
iterate(Prev,Curr) :-
    operator(Prev,Curr),
    fail.
iterate(_,Curr) :-
    fact(Curr,_), !,
    Next is Curr + 1,
    iterate(Curr,Next).
iterate(_,_).

record(N,F) :-
    cond_assert(N,F).

cond_assert(N,F) :-
    \+ (numbervars(F,0,_), fact(_,F)),
    assert(fact(N,F)).
```

```
operator(_,1) :-
    my_clause(H,[]),
    record(1,H).
operator(Prev,Curr) :-
    Curr > 1,
    my_clause(H,B),
    fact(Prev,Atom),
    select(Atom,B,Rest),
    prove(Rest),
    record(Curr,H).

prove([B|Bs]) :-
    fact(_,B),
    prove(Bs).
prove([]).
```

**Fig. 11.** A Prolog Interpreter for Semi-Naive Evaluation

```
go(File) :-
    load_file(File),
    eager(true),
    fail.

prove([B|Bs]) :-
    fact(B),
    prove(Bs).
prove([]).
```

```
eager(true) :-
    my_clause(H,[]),
    eager(H).
eager(Atom) :-
    record(Atom),
    my_clause(H,B),
    select(Atom,B,Rest),
    prove(Rest),
    eager(H).
```

**Fig. 12.** A Prolog Interpreter for Eager Evaluation

An interpreter for this method was introduced by Wunderwald [61]. In contrast to semi-naive evaluation, eager evaluation does not need to distinguish between new and old facts. Instead, whenever a new instance of a clause head is derived it is used immediately to re-solve all clauses which contain a call to the corresponding predicate in their body. Consequently eager evaluation is simple to describe as a recursive depth first strategy. Figure 12 depicts an interpreter for eager bottom-up evaluation. Evaluation is triggered by a call of the form `eager(true)` and the control component is managed by the underlying control for recursive calls, the runtime stack.

Eager (and semi-naive) evaluation can be used also for query directed bottom-up evaluation as shown in Figure 13. Program lines have been numbered to facilitate an explanation. There are two cases to consider when a new fact is found:

1. If the new fact is an *answer* (line `14`), then for each clause `Head :- Body` (line `15`) that has been called (line `16`) and contains a matching call for the

```
/* 1*/  go(File,Goal) :-
/* 2*/     load_file(File),
/* 3*/     record(call(Goal)),
/* 4*/     eager(call(Goal)),
/* 5*/     fail.
/* 6*/  go(_,_) :- showfacts.

/* 7*/  prove_ind([B|_]) :-
/* 8*/     record(call(B)),
/* 9*/     eager(call(B)).
/*10*/  prove_ind([B|Bs]) :-
/*11*/     fact(ans(B)),
/*12*/     prove_ind(Bs).
/*13*/  prove_ind([]).
```

```
/*14*/  eager(ans(Atom)) :-
/*15*/     my_clause(Head,Body),
/*16*/     fact(call(Head)),
/*17*/     \+\+ member(Atom,Body),
/*18*/     prove_ind(Body),
/*19*/     record(ans(Head)),
/*20*/     eager(ans(Head)).

/*21*/  eager(call(Atom)) :-
/*22*/     my_clause(Atom,Body),
/*23*/     prove_ind(Body),
/*24*/     record(ans(Atom)),
/*25*/     eager(ans(Atom)).
```

**Fig. 13.** Induced Eager Evaluation

new fact (line `17`), we solve the `Body` (line `18`) and record the corresponding instance of the `Head` (line `19`). If it is new then there is a recursive call (line `20`) (otherwise we fail at line `19`).

2. If the new fact is a *call* (line `21`), then for each clause with a matching head (line `22`), we solve the body (line `23`) and record the corresponding instance of the `Head` (line `24`). If it is new then there is a recursive call (line `25`) (otherwise we fail at line `24`).

In both cases, solving a clause body involves recording and processing new calls (lines `7-9`) and matching body atoms with facts derived so far (lines `10-12`).

We can improve the code in Figure 13 for the case when a new answer is encountered (lines `14-20`). A new answer `Atom` that matches a call $b_i$ in the clause $h \leftarrow b_1, \ldots, b_i, \ldots, b_n$ may give rise to new call patterns for $b_{i+1}, \ldots, b_n$ but will not influence calls for $b_1, \ldots, b_i$. Hence the case covered by lines `7-9` in the code is redundant for atoms to the left of (and including) $b_i$. The optimised interpreter is obtained by replacing lines `17` and `18` by

```
/*17'*/   append(Left,[Atom|Right],Body),
/*18'*/   prove(Left), prove_ind(Right),
```

respectively and defining `prove` by

```
prove([B|Bs]) :-
    fact(ans(B)), prove(Bs).
prove([]).
```

## 6 Discussion

We have presented the case for the meta-circular approach to the analysis of logic programs. We have argued that, when taking this approach, it is natural to use interpreters based on $T_P$-style semantics, or more precisely, on the *s*-semantics [26]. This paper presents working Prolog interpreters for program analysis, and detailed code is available via the world-wide web [19]. We have avoided details about how to approximate Prolog builtins, as this is well understood.

The meta-circular approach is highly versatile, in the sense that simple changes to the interpreter allow us to change what is being observed, which properties are being extracted, and even the evaluation strategy used. It appears particularly useful for prototyping, but also for classes on semantics and abstract interpretation of logic programs. The versatility of the approach has again been confirmed to us during the writing of this survey: Both the domain *Parity* from Section 2 and its combination with the depth-$k$ domain used in Section 4.2 resulted from interactive experiments with the interpreter—they were created on the laboratory bench, so to speak.

A bottom-up formulation of abstract interpretation for logic programs was originally suggested by Marriott and Søndergaard [44,45,46]. The idea was further developed by many researchers, including Barbuti, Giacobazzi and Levi [5], Codish and Demoen [13,14], Codish, Dams and Yardeni [12], Gallagher and co-workers [32,30], as well as in the "pre-interpretation" approach [7,9]. An interesting approach has been the use of Toupie [21], which can be thought of as a finite-domain $\mu$-calculus interpreter allowing fixed point problems to be stated succinctly and solved efficiently. Toupie incorporates a finite-domain constraint solver and was designed and used for the abstract interpretation of logic programs.

In addition to the domains used as examples in this paper, several others can be utilised with the framework. Recent work considers $E$-unification (unification with an equality theory $E$ [2]) for `unify(A,B)`. It turns out that ACI unification leads to a notion of type dependency analysis [16]; adding a unit element **1** and using standard ACI**1** unification gives an analysis for groundness dependencies over the domain of *definite* Boolean functions [33]; while a non-standard ACI**1** unification algorithm gives set-sharing analysis [17].

Building on work by Dawson, Ramakrishnan and Warren [24], Codish, Demoen and Sagonas [15] demonstrated the utility of a logic program system with *tabulation* for the purpose of program analysis. To this end they use the XSB system [54]. XSB offers efficient fixed point evaluation through tabling implemented at the abstract machine level. The authors describe simple XSB interpreters which make answers and calls observable and give several abstract domains for analysis. The approach is similar to that presented here, but the emphasis is on techniques that make the analyses efficient, such as the tabling of abstract operations. In contrast we have focused on the specification of extensions *within* the interpreters. The evaluation strategy cannot be varied in the XSB approach, as it is fixed by the underlying implementation.

A shortcoming of a dataflow analysis approach based on meta-circular interpretation is that an interpreter inevitably incurs a cost which is felt as input programs grow. However, it is possible to apply partial evaluation to remove the level of interpretation [41]. This leads to a more efficient analyser and can be viewed as what is sometimes called abstract compilation [34].

Another shortcoming is that it is far from clear how to implement abstract domains for which abstract conjunction is not simply conjunction. Set sharing analysis, for example, falls in this class. The problem is not so much the fact that the analysis needs to keep track of relations that hold amongst variables, as opposed to properties of individual variables. The same can be said for the groundness dependency analysis given in Section 4.1, and it is possible to express sharing information much like groundness information, using propositional logic [20]. Rather, the problem is that sharing analysis uses an abstract conjunction operation which is rather complex, and not faithfully mimicked by the conjunction implicitly used by the interpreter.

However, making up for these shortcomings is the fact that the meta-circular approach can produce *fast* analysers (for example [35]) in very short development time. Moreover, as experimental or pedagogical tool, the meta-circular approach has considerable advantages, as we hope to have demonstrated in this paper. We are not aware of similar work for other programming language paradigms, and this is something that may be worth exploring.

**Acknowledgements**

We would like to thank John Gallagher and Peter Schachte for fruitful discussions, and Samir Genaim for help with the interpreters.

## References

1. S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
2. F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford University Press, 1994.
3. R. Bagnara. China: A data-flow analyzer for CLP languages. Survey at URL http://www.cs.unipr.it/China/, 19 February 2001.
4. F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. Fifth ACM SIGMOD/SIGACT Symp. Principles of Database Systems*, pages 1–15. ACM Press, 1986.
5. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based abstract interpretation. *ACM Transactions on Programming Languages*, 15(1):133–181, 1993.
6. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20:149–197, 1994.
7. D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming: Proc. Sixth Int. Symp.*, volume 844 of *Lecture Notes in Computer Science*, pages 432–446. Springer-Verlag, 1994.

8. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.
9. M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A freeness and sharing analysis of logic programs based on a pre-interpretation. In R. Cousot and D. A. Schmidt, editors, *Static Analysis: Proc. Third Int. Symp.*, volume 1145 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 1996.
10. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimisation of Prolog programs. In *Proc. 1987 Symp. Logic Programming*, pages 192–204. IEEE Comp. Soc., 1987.
11. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
12. M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.
13. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In P. Van Hentenryck, editor, *Static Analysis: Proc. First Int. Symp.*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.
14. M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
15. M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. *Springer Int. Journal of Software Tools for Technology Transfer*, 2(1):29–45, 1998.
16. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
17. M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. *Journal of Logic Programming*, 41(2):110–149, 2000.
18. M. Codish, K. Marriott, and C. Taboch. Improving program analyses by structure untupling. *Journal of Logic Programming*, 43(3):251–263, 2000.
19. M. Codish and H. Søndergaard. Meta-circular abstract interpretation in Prolog. http://www.cs.bgu.ac.il/~mcodish/Tutorial/, 11 June 2001.
20. M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
21. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 75–91. Springer-Verlag, 1993.
22. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
23. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282. ACM Press, 1979.
24. S. Dawson, R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems: A case study. In *Proc. ACM SIGPLAN 96 Conf. Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.

25. S. K. Debray and D. S. Warren. Automatic mode inference for Prolog programs. In *Proc. 1986 Symp. Logic Programming*, pages 78–88. IEEE Comp. Soc., 1986.
26. M. Falaschi, G. Levi, M. Gabbrielli, and C. Palamidessi. A new declarative semantics for logic languages. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 993–1005. MIT Press, 1988.
27. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
28. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 103:86–113, 1993.
29. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1997.
30. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. Lloyd, editor, *Logic Programming: Proc. 1995 Int. Symp.*, pages 351–365. MIT Press, 1995.
31. J. Gallagher and D. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation: Proc. LOPSTR 92*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.
32. J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Logic Programming: Proc. Eleventh Int. Conf.*, pages 599–613. MIT Press, 1994.
33. S. Genaim and M. Codish. The Def-inite approach to dependency analysis. In D. Sands, editor, *Proc. Tenth European Symp. Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 2001.
34. M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.
35. J. Howe and A. King. Implementing groundness analysis with definite Boolean functions. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 2000.
36. N. D. Jones. Concerning the abstract interpretation of Prolog. Draft paper, DIKU, Copenhagen, 1985. Cited in Mellish [49].
37. N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proc. Thirteenth ACM Symp. Principles of Programming Languages*, pages 296–306. ACM Press, 1986.
38. N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood, 1987.
39. A. Kelly, K. Marriott, H. Søndergaard, and P. Stuckey. A practical object-oriented analysis engine for constraint logic programs. *Software—Practice and Experience*, 28(2):199–224, 1998.
40. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
41. G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Logic Programming: Proc. Tenth Int. Conf.*, pages 354–373. MIT Press, 1993.

42. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.
43. M. J. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In E. Lusk and R. Overbeek, editors, *Logic Programming: Proc. North American Conf. 1989*, pages 963–980. MIT Press, 1989.
44. K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 733–748. MIT Press, 1988.
45. K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. X. Ritter, editor, *Information Processing 89*, pages 601–606. North-Holland, 1989.
46. K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(2 & 3):181–204, 1992.
47. B. Martens and D. De Schreye. Why untyped nonground metaprogramming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.
48. C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.
49. C. S. Mellish. Abstract interpretation of Prolog programs. In E. Shapiro, editor, *Proc. Third Int. Conf. Logic Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 1986.
50. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2&3):315–347, 1992.
51. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Scotland, 1981.
52. R. Ramakrishnan and J. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.
53. J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 135–152. 1970.
54. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 442–453. ACM Press, 1994.
55. T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.
56. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1986.
57. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, second edition, 1994.
58. J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.
59. M. van Emden and R. Kowalski. The semantics of logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.
60. W. Winsborough. *Automatic, Transparent Parallelization of Logic Programs at Compile Time.* PhD thesis, University of Wisconsin-Madison, Wisconsin, 1988.
61. J. Wunderwald. Memoing evaluation by source-to-source transformation. In M. Proietti, editor, *Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1995.

# Flow Analysis: Games and Nets

Chris Hankin[1], Rajagopal Nagarajan[2], and Prahladavaradan Sampath[3]

[1] Imperial College of Science, Technology and Medicine, London SW7 2BZ.
[2] University of Warwick, Coventry CV4 7AL.
[3] Teamphone.com, London W1D 7EQ.

**Abstract.** This paper presents a graph-based formulation of control-flow analysis using results from game semantics and proof-nets. Control-flow analysis aims to give a conservative prediction of the flow of control in a program. In our analysis, terms are represented by proof-nets and control-flow analysis amounts to the analysis of computation paths in the proof-net. We focus on a context free analysis known in the literature as 0-CFA, and develop an algorithm for the analysis. The algorithm for 0-CFA performs dynamic transitive closure of a graph that is based on the judgement associated with the proof-net. Correctness of the algorithm relies on the correspondence between proof-nets and certain kinds of strategies in game semantics.

## 1 Introduction

Twenty years ago Neil Jones published [Jon81] in the 8th International Colloquium on Automata, Languages and Programming. The paper opens with the words:

"*A method is described to extract from an untyped λ-expression information about the sequence of intermediate λ-expressions obtained during its evaluation. ... From a flow analysis viewpoint these results extend existing interprocedural analysis methods to include ... the use of functions both as arguments to other functions and as the results returned by them.*"

The paper goes on to present a technique called "control flow analysis" which results in safe descriptions of the states that an abstract machine reaches in evaluating an expression. The paper was a seminal contribution to the field of control flow analysis; this field was further developed by a number of the members of Neil's TOPPS group at the University of Copenhagen (notably [Ses91, Mos97a, Mos97b]). Moreover, the work has inspired a great deal of work on this topic in other research groups [Shi91, Deu92, Pal94, HM97b, NN97, MH98b] (which represents a very small sample!).

Most approaches in the literature specify control-flow analysis as the solution to a set of (in)equations derived from the syntax of the program. Soundness of the analysis is then demonstrated with respect to an operational semantics; i.e. the derived set of (in)equations describes, for each sub-term, at least the set of abstractions that it evaluates to in the operational semantics of the program. An example of such a soundness result is the proof in [NN97] for a constraint based formulation of control-flow analysis. That paper relates various flavours of control-flow analyses to a structural operational semantics (SOS), and the soundness results for the analysis employs sophisticated co-inductive proof techniques.

More recently, work has been reported on using game semantics to provide a framework for control-flow analysis [MH98b, MH98a, MH99, Sam99]. A feature of the game semantic approach is that control-flow analysis can be specified in a *semantics directed* manner; i.e. the analysis is specified directly in terms of the semantics rather than syntax of a program. This opens up the possibility of exploiting the rich structure of game semantics for specifying analyses. For instance, it might be possible to use the *factorization theorems* of game semantics [AM97, AHM98] to extend program analyses to handle new language features in a modular way.

The approach that we report on in this paper is based on game semantics [AJM94, HO94], and uses proof-nets as representations of $\lambda$-terms [Mac94, Sam99]. This is in contrast to earlier analyses [MH98b, MH98a, MH99] based on game semantics, which use a Böhm tree like representation of terms. As a result of using this representation, these analyses rely on pre-processing a program into a special form before analysing them. The use of proof-nets to represent terms in this paper removes the need for such pre-processing of programs.

Proof-nets play an important role in the analysis specified in this paper. The proof-net encoding of $\lambda$-terms gives rise to structures that are isomorphic to traditional graph representations of $\lambda$-terms. Moreover linear head reduction[1] of $\lambda$-terms can be expressed in terms of linear head cut-elimination of proof-nets [MP92, Sam99]. Proof-nets and games are also intimately related in many ways and share a common origin in Linear Logic [Gir87]. Proof-nets can be seen as encoding the strategies of game semantics, thereby providing a bridge between syntax and semantics.

We specify the control-flow analysis directly in terms of game semantics. It is based on the interpretation of *interacting* moves in game semantics as linear head-reduction steps of $\lambda$-terms [DHR96, Sam99]. The soundness proof of the analysis relies on the close connection between game semantics on one hand, and the syntax and linear head-reduction of $\lambda$-terms on the other hand. The soundness proof can be factored into two stages; first establishing the connection between game semantics and linear head reduction, and then demonstrating the soundness of the analysis with respect to game semantics. Only the second stage of the correctness proof is specific to the analysis — the connection between game semantics and linear head-reduction is a general semantic property of games and is expected to be applicable to a number of control-flow based analyses.

We also develop an algorithm for control-flow analysis, based on the game semantics directed specification. The algorithm performs *dynamic-transitive closure* of a *cost-resource* graph. The graph structure is extracted from the proof-net representation of the program, and the edges of the graph are labelled with cost and resource components obtained from the proof-net representation.

[1] a linearized version of head reduction which makes one substitution at a time.

As we will see later, the graph structure used in the algorithm is in fact a representation of the type of the program. In this sense our work is related to the type-based approaches of [Mos97b, HM97a]. These approaches perform (explicit or implicit) $\eta$-expansion (based on the type-structure) of the program, and then use a simple transitive closure procedure to obtain the analysis. The algorithm we present here does not require an $\eta$-expansion step, but on the other hand requires a dynamic-transitive closure procedure.

The worst-case complexity of the algorithm we present for control-flow analysis is $\mathcal{O}(n^3)$, and is in line with the state-of-the-art. We can in fact refine the complexity of the algorithm to $\mathcal{O}(n^2m)$, where $m$ is the size of the (partial) type information required by the algorithm, which itself is bounded above by $n$. By comparison, the algorithm presented in [Mos97b, HM97a], exhibits a better (quadratic) asymptotic complexity, but has to make the assumption that the size of the type information is linear in the size of the (untyped) term. Our algorithm utilizes type information in a similar manner to [Mos97b, HM97a], but without making any assumptions on the size of the type information.

The significance of the contributions of this paper are:

- The analysis framework using games is extended, using proof-nets, to cope with arbitrary (typed) $\lambda$-terms whereas previous work has been restricted to pre-processed $\lambda$-terms having a special form [MH98b, MH98a].
- The game semantic framework gives rise to a novel algorithm for control-flow analysis. The algorithm operates on the type structure of the program; this is in contrast to other approaches that operate on the syntactic structure of the program. The worst case complexity of the algorithm is $\mathcal{O}(n^3)$ where the term is of $\mathcal{O}(n)$; this is in line with other algorithms for control-flow analysis.
- The correctness of the control-flow analysis uses only standard semantic properties of games. Therefore there is no need to *instrument* the semantics to add control-flow information.
- The soundness proof can be factored into two stages, only one of which is specific to the analysis. It is conceivable to use the connection between game semantics and linear head-reduction as the basis for a number of other control-flow based analyses.

For the sake of simplicity we only consider terms of simply typed $\lambda$-calculus under call-by-name evaluation, but the framework and algorithm are equally applicable to PCF and need not be changed to handle conditionals, recursion etc.

The rest of the paper is organised as follows. Section 2 introduces the basic ideas behind game semantics. Section 3 presents proof-nets as syntactic representation for $\lambda$-terms. Section 4 outlines the correspondence between proof-nets, games and linear head-reduction. Section 5 describes how control-flow analysis can be performed in our framework. Section 6 presents an algorithm for the analysis. Section 7 contains concluding remarks.

## 2 Game Semantics

Game semantics [HO94, AJM94] is a relatively new field of semantics where semantic models of programming languages are built from two player games. Game semantics exhibits the compositional nature of standard denotational semantics while at the same time capturing the intensional nature of operational semantics. This is achieved in game semantics by denoting programs by strategies, which express *potential* for computation. Actual computation is then performed by *interacting* with another strategy (representing the environment). The steps of interaction intuitively represent the steps of an operational semantics. In this section we give a brief overview of some of the concepts of game semantics; we refer the reader to the literature [AJM94, HO94, McC98, Har99] for a more complete introduction to game semantics.

Game semantic models are presented as (two person) games between $\mathsf{P}$ (for player) and $\mathsf{O}$ (for opponent). Each person takes turns in making moves, which can be interpreted as steps of a computation. Roughly speaking, $\mathsf{P}$ models the behaviour of a program while $\mathsf{O}$ models the *environment* of the program. Each person ($\mathsf{O}$ and $\mathsf{P}$) can make two types of moves: Questions and Answers. A question can be interpreted as *calling* a procedure and an answer can be interpreted as *returning* from the procedure with a *value*. There is also a causality relation, called *enabling*, between moves in the games; for example questions cannot be answered before they are asked. At any point in the game, the *position* in the game refers to the sequence of moves made in the game up to that point. The set of moves in a particular game along with some *ground-rules* like enabling are together referred to as the *arena* of the particular game being played. More formally, an arena for type $A$ is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where $M_A$ is the set of moves that can be played in the arena,

$$\lambda_A : M_A \rightarrow \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$$

is the $\mathsf{O}/\mathsf{P}$ labelling function on moves, and $\vdash_A$ is the enabling relation on moves. It is usual to place restrictions on the enabling such that only player moves enable opponent moves (and vice versa), initial moves are always opponent questions ("the environment initiates evaluation") and answer moves are enabled by question moves.

*Example 1.* The arenas for base types have a very simple structure; for example in the case of booleans, the arena consists of an *initial* Opponent move $\mathsf{q}$ and two Player moves $\mathsf{true}$ and $\mathsf{false}$. The enabling relation then states that the Player moves cannot be played before the initial Opponent move is played.

$$M_{\mathbf{B}} = \{\mathsf{q}, \mathsf{true}, \mathsf{false}\}$$

and

$$\begin{aligned} \lambda_{\mathbf{B}}(\mathsf{q}) &= \mathsf{OQ} \\ \lambda_{\mathbf{B}}(\mathsf{true}) &= \mathsf{PA} \\ \lambda_{\mathbf{B}}(\mathsf{false}) &= \mathsf{PA} \end{aligned}$$

and

$$\mathsf{q} \vdash_{\mathbf{B}} \mathsf{true}$$
$$\mathsf{q} \vdash_{\mathbf{B}} \mathsf{false}$$

□

Having defined arenas for base types, those for composite types are constructed inductively on the structure of the type. For example, given two arenas, we can combine them to model a *function space.* Given arenas $A$ and $B$ we construct the function space $A \Rightarrow B$ by switching the O/P polarity of moves in $A$. Also, the initial moves in $A$ are now enabled by the initial moves in $B$; intuitively, we do not want to consult an input unless an output is requested in the game[2].

**Definition 1 (Function space).** *Given two arenas $A$ and $B$, we can define the arena $A \rightarrow B$ as:*

$$\begin{aligned} M_{A\rightarrow B} &= M_A + M_B \\ \lambda_{A\rightarrow B} &= [\bar{\lambda}_A, \lambda_B] \\ \vdash_{A\rightarrow B} &= a \vdash_{A\Rightarrow B} b \text{ iff } (a \vdash_B b) \vee (a \vdash_A b \wedge a \neq b) \vee (a \in I_B \wedge b \in I_A) \end{aligned}$$

□

Given a program, *prog*, the denotation of the program in game semantics, $[\![prog]\!]$, is a *strategy* for P to play a game – formally this is a set of sequences of moves that conform to certain rules (for example, all of the sequences are of even length – Player has the last move; but see the cited papers for examples of additional possible rules).

*Example 2.* We illustrate the example of the $[\![iszero]\!]$ strategy, played on the arena $\mathbb{N} \rightarrow \mathbb{B}$, that returns true if the input is zero and false if the input is of the form succ($n$). One possible play is shown below

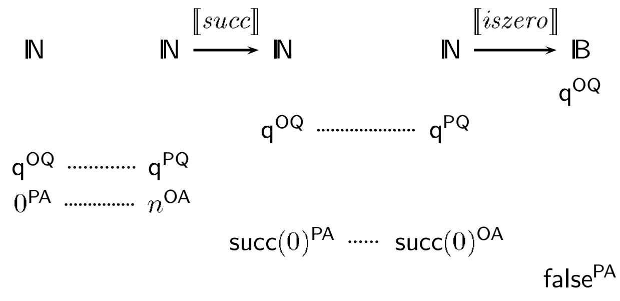
$$\begin{tikzcd} \mathbb{N} \arrow[r] & \mathbb{B} \\ & \mathsf{q}^{\mathsf{OQ}} \\ \mathsf{q}^{\mathsf{PQ}} & \\ \mathsf{succ}(n)^{\mathsf{OA}} & \\ & \mathsf{false}^{\mathsf{PA}} \end{tikzcd}$$

The labels on the moves indicate the Player/Opponent and Question/Answer nature of moves. Intuitively, the first move requests the result of the function; the Player responds by requesting an argument — this move is played in the (sub) arena corresponding to the type of the argument. When Opponent supplies an argument (succ(0) say), the player replies to the initial question with *iszero*(succ(0)), i.e. false. □

[2] This models call-by-name computation.

Given a strategy denoting a program, executing the program is represented by pitting the strategy for the program against a strategy that represents the context of the program. This corresponds to providing the arguments, and environment for interpreting the free variables of the program. In game semantics, this pairing of program and context is represented by the *composition* of two strategies, for the program and context respectively. The composition operation sets up an *interaction* between strategies whereby some P-moves in one strategy *interact* with O-moves in the other and vice-versa. This is similar to parallel composition of processes in, for example, CCS [Mil89]. In traditional game semantics it is usual for the resulting interactions to be hidden, analogous to the parallel composition with hiding of CCS. However, for the purposes of control flow analysis, it is important that the interaction be exposed.

*Example 3.* We illustrate interaction between strategies by considering the composition ($[\![0]\!]$; $[\![succ]\!]$); $[\![iszero]\!]$ of three strategies.



$\mathbb{N}$
$\mathbb{N} \xrightarrow{[\![succ]\!]} \mathbb{N}$
$\mathbb{N} \xrightarrow{[\![iszero]\!]} \mathbb{B}$
$q^{OQ}$
$q^{OQ}$ ........ $q^{PQ}$
$q^{OQ}$ ........ $q^{PQ}$
$0^{PA}$ ........ $n^{OA}$
$succ(0)^{PA}$ ...... $succ(0)^{OA}$
$false^{PA}$

□

In this example, the dotted lines indicate the interaction of between moves in the arenas for the strategies $[\![succ]\!]$, $[\![iszero]\!]$, and $[\![0]\!]$. Note that interacting moves possess opposite (O/P) polarities. Intuitively this corresponds to the fact that the "output" of *succ* is the "input" of *iszero*.

Formally, a program makes a request to the context by playing a P move in its arena. This move then interacts with an O move in the context-arena; playing the interacting O move passes "control" to the strategy denoting the context. Similarly, any response from the context strategy is returned to the program strategy along with "control".

Another game semantic notion that will be used in this paper is that of the Player-view (P-view) of a position in a game. The Player-view of a position is defined as a particular subsequence of moves in the position. Strategies where the P-moves made at any position are fully determined by the P-view of the position are of independent interest — they are known as the *innocent* strategies and can be represented as a function from a P-view to a P-move. Innocent strategies have been used for constructing fully abstract models of functional languages [HO94, McC98].

Lying behind our informal presentation of game semantics is a category of call-by-name games. The interaction between player and opponent emulates the typical demand-driven evaluation of interpreters for call-by-name languages.

## 3 Proof-Nets

In this section we introduce proof-nets as representations of $\lambda$-terms. Proof-nets also support the notion of a *computation path* through a program; i.e. computation can be viewed as visiting a sequence of nodes in the proof-net representation of a program. In this sense proof-nets are analogous to the higher-order flow graphs of [Mos97a, Mos97b], and to the generalized flow-charts of [MH98a].

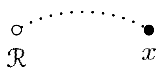
Proof-nets are graphical proof-representations for linear-logic. They were proposed in [Gir87] as a technique of performing multi-conclusion natural-deduction style proofs for linear logic. In this paper, we consider proof-net representations of *intuitionistic propositional* logic. Proof-nets can be considered as representations of terms in the simply-typed $\lambda$-calculus via a Curry-Howard correspondence: given a proof-net representation of a term, the *type* of the term can be obtained as the *judgement* proved by the proof-net. In fact, proof-net representations of $\lambda$-terms exhibit an isomorphism with a more conventional graph-representation [Wad71] of the terms.

In this paper we draw the proof-nets as forest structures. We associate a positive or negative polarity to the nodes of the forest. The nodes with positive polarity are drawn as $\circ$, and the nodes with negative polarity are drawn as $\bullet$. The axiom links of the proof-net are represented as connections (dotted-lines) between nodes of opposite polarities in the forest. The cut links are represented as connections (solid lines) between the roots of two trees in the forest structure having opposite polarities (cf. Example 5). The proof-nets we consider have a unique root node of positive polarity that is not connected by a cut-link: This corresponds to the root of the term represented by the proof-net. We will use $\mathfrak{R}$ to refer to this node. In a traditional graph representation of $\lambda$-terms, the node labelled $\mathfrak{R}$ corresponds to the unique root of the graph representation of a term.

We refer the reader to the literature [Gir87, Dan90, Lam94, Sam99] for a formal definition of proof-nets; we just give a translation from $\lambda$-terms in context to proof-nets. The proof-nets we use in this paper are described in [Sam99], and are simplified versions of general proof-nets, designed with the $\lambda$-calculus in mind.

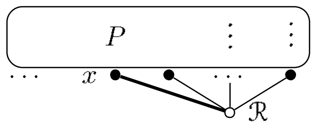
The translation $\mathfrak{T}[\![\_]\!]$ from simply-typed $\lambda$-terms in context to proof-nets is defined by induction on the structure of a (sequent calculus) derivation of a typed term. The translation presents explicit constructions for contraction and weakening — these are essentially vestiges of the linear logic origins of proof-nets.

**Definition 2 (Variable).** *We define $\mathfrak{T}[\![x : A \vdash x : A]\!]$, where $x$ is of type $A$, as the proof-net*



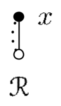
*which is an axiom-link in proof-net terminology. Axiom links in proof-nets represent the axioms of the sequent calculus presentation of the proof.* □

**Definition 3 (Abstraction).** *Let* $\mathcal{T}[\![\Gamma, x : A \vdash M : B]\!] = P$*, then we define* $\mathcal{T}[\![\Gamma \vdash (\lambda x.M) : A \Rightarrow B]\!]$ *from the proof-net* $P$ *by making the* $\bullet$*-node corresponding to the variable* $x$ *an immediate successor of the root node.*

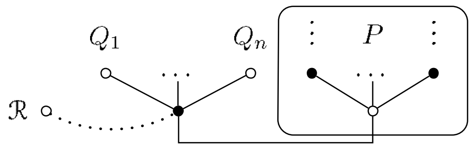


□

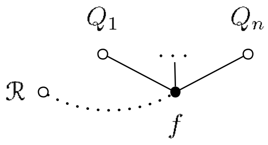
*Example 4.* Consider the term $\lambda x.x$. We have already seen the representation of the term $x$ as a single axiom link. Abstracting the variable $x$ is now represented by making the $\bullet$-node corresponding to $x$ an immediate successor of the root node ($\mathcal{R}$):



**Definition 4 (Application).** *Let* $\mathcal{T}[\![\Gamma \vdash M : A_1 \ldots A_n \Rightarrow B]\!] = P$ *and* $\mathcal{T}[\![\Delta \vdash N_i : A_i]\!] = Q_i$*, then we define* $\mathcal{T}[\![\Gamma, \Delta \vdash MN_1 \ldots N_n : B]\!]$ *to be the proof-net*



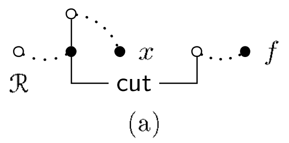
*We also give a translation of the term* $MN_1 \ldots M_n$ *for the special case where* $M$ *is a* variable*,* $f$ *say. In this case the term can be translated into a proof-net as:*

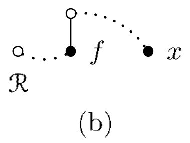


□

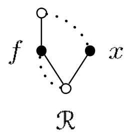
In fact, the special case above can be obtained from the more general translation given above by a series of *axiom-cut elimination* steps on proof-nets. Axiom-cuts are cuts between two proof-nets where at least one of them is a simple axiom link. In general, elimination of cuts in a proof-net is considered to be *computation.* However elimination of Axiom-cuts is considered to be mere bureaucratic rearrangement of the structure of the proof-net, and no computational content is ascribed to these cut-elimination steps [Sam99].

*Example 5.* Consider the term $\lambda f, x.f(x)$. We have already seen how the variable occurrences $f$ and $x$ can be translated into axiom links. The translation of application terms gives the proof-net (a) below; and the translation for the special case where the operator is a variable occurrence gives the proof-net (b) below. Note the presence of the cut-link in the first translation; this cut-link is elided in the second translation as the cut is an axiom-cut and has no computational significance.

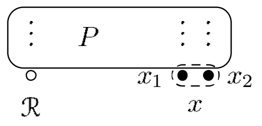




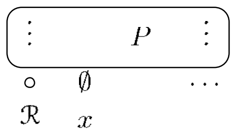
Now applying the translation for abstraction, the translation of the term is given as:



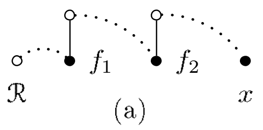
**Definition 5 (Contraction and Weakening).** *Let $\mathfrak{T}[\![\Gamma, x_1 : A, x_2 : A \vdash M : B]\!]=P$, then the contraction of variables $x_1$ and $x_2$ as $x$, $\mathfrak{T}[\![\Gamma, x : A \vdash M : B]\!]$is given by the following proof-net where the $\bullet$-nodes corresponding to $x_1$ and $x_2$ are combined into a single set, representing $x$.*

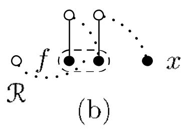


*Let $\mathfrak{T}[\![\Gamma \vdash M : B]\!]=P$, then we define $\mathfrak{T}[\![\Gamma, x : A \vdash M : B]\!]$ where $x$ does not occur in $\Gamma$ as the following proof-net where $x$ is represented by the empty set, $\emptyset$.*

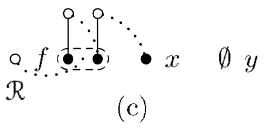


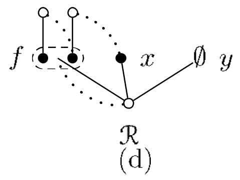
*Example 6.* Consider the term $\lambda f, x, y.f(f(x))$. This term abstracts the variable $y$ that does not occur in the body, and also contains two occurrences of the variable $f$. Let us first translate the term $f_1(f_2(x))$, where $f_1$ and $f_2$ are distinct. This corresponds to the translation $[\![f_1 : A \to A, f_2 : A \to A, x : A \vdash f_1(f_2(x)) : A]\!]$ and is given by the proof-net (a) below. Now applying contraction to the variable occurrences $f_1$ and $f_2$ gives the proof-net (b) below. This corresponds to the translation $[\![f : A \to A, x : A \vdash f(f(x)) : A]\!]$.





Applying weakening for the variable $y$ gives the proof-net (c) and finally applying abstraction for the variables $f$, $x$, and $y$ gives the proof-net (d).





**Lemma 1.** *Given a simply-typed λ-term, the translation of the term using* $\mathcal{T}[\![\_]\!]$ *results in a proof-net; moreover this proof-net is isomorphic to a traditional graph representation of the λ-term [Wad71].*

*Proof.* By induction on the structure of the λ-term. The isomorphism with traditional graph-representation of λ-terms maps sequences of abstraction nodes to ∘-nodes and sequence of application nodes to •-nodes. In the rest of the paper we will refer to terms and proof-nets interchangeably in light of the isomorphism between them. □

The isomorphism between proof-nets and λ-terms gives rise to a correspondence between nodes in the proof-net and pieces of syntax of the λ-term. In the inductive translation of a λ-term, $T$, every ∘-node, $a$, in the proof-net is the root ($\mathcal{R}$) of the translation of some sub-term, $T'$, of $T$. The node $a$ in the proof-net then corresponds to the sub-term $T'$. A similar correspondence can be established for •-nodes and variable occurrences. Loosely, each •-node represents an *occurrence* of a variable and each ∘-node in the proof-net represents a *sub-term*.

Given the proof-net representation of a term, the type of the term can be obtained as the judgement proved by the proof-net. The judgement itself can be obtained by simply considering the forest structure of the proof-net and *forgetting* the contractions and weakenings occurring in the proof net. The resulting forest structure represents the intuitionistic sequent proved by the proof-net: the unique tree with ∘-polarity root represents the unique conclusion of an intuitionistic sequent.

The judgement proved by the proof-net is the type of the term represented by the proof-net; this can be seen as a manifestation of the Curry-Howard correspondence. Note that the judgement obtained from the proof-net contains pairs of cut-formulae. By including the pairs of cut-formulae in the type of the term, we obtain a notion of the *intensional type* of a term. Therefore the intensional type of a term is a judgement of the form $\vdash \Gamma; [\cdots]$; where the square brackets enclose a list of pairs of cut-formulae. Intuitively, the intensional type of the term captures some *internal* structure of the term in the form of the cut-formulae.

There is a natural map from the nodes of the proof-net to the nodes in the forest representation of its intensional type, obtained by *forgetting* the effects of weakening and contraction in the proof-net. We will refer to this mapping in the rest of the paper as $\rho_{\mathcal{N},\vdash\Gamma}$, where $\mathcal{N}$ stands for the proof-net and $\vdash \Gamma$ stands for the intensional type of $\mathcal{N}$. Where the context is clear, we will refer to this mapping as simply $\rho$.

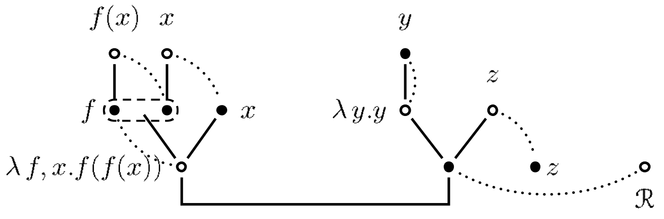
*Example 7.* We construct the proof-net representation of the term

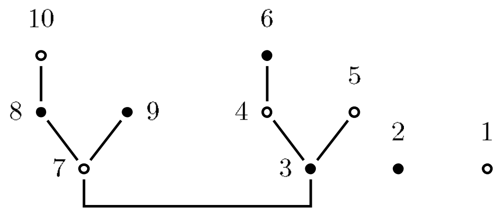
$$(\lambda f, x.f(fx))(\lambda y.y)z$$

We also give the intensional type consisting of the judgement proved by the proof-net along with pairs of cut formulae. Note that the forest structure of the intensional type represents the sequent

$$\vdash \alpha^2 \Rightarrow \alpha^1; [(\alpha^{10} \Rightarrow \alpha^8) \Rightarrow \alpha^9 \Rightarrow \alpha^7 \ \mathsf{cut}\ (\alpha^6 \Rightarrow \alpha^4) \Rightarrow \alpha^5 \Rightarrow \alpha^3]$$

where the superscripts correspond to the labelling on the nodes in the forest representation of the intensional type.





□

We can use the proof-net representation of a term to define a reduction strategy for $\lambda$-terms — *linear head-reduction.* This is based on linear head cut-elimination of proof-nets [MP92]. Linear head-reduction, follows the same normalizing strategy as head-reduction, but substitutes for only the *head-occurrence* of the head-variable; it is therefore a *hyper-lazy* evaluation strategy. Linear head-reduction has a very simple characterisation in terms of proof-nets. The linearisation step *separates* the occurrence of the head-variable from the other occurrences, and the cut-elimination step performs the *computation*; i.e. substitution.

If we interpret the cut links in the proof-nets as encoding substitutions, then the linearisation step can be interpreted as the reduction

$$(\lambda x_1 \dots x_m . y E(y))[y \mapsto t] \longrightarrow (\lambda x_1 \dots x_m . y_1 E(y_2))[y_1 \mapsto t][y_2 \mapsto t]$$

where $E(y)$ stands for any expression which possibly contains (non-head) occurrences of $y$. The cut-elimination step can be interpreted as the reduction

$$\lambda x_1 \dots x_m . x t_1 \dots t_n [x \mapsto \lambda y_1 \dots y_n . t] \longrightarrow \lambda x_1 \dots x_m . t[y_1 \mapsto t_1 \dots y_n \mapsto t_n]$$

In this case linear head-reduction is said to *visit*, in sequence, the nodes in the proof-net corresponding to the variable occurrence $x$ (a $\bullet$-node) and the term $\lambda y_1 \dots y_n . t$ (a $\circ$-node). The linear head-reduction procedure shown above has the attributes of an abstract-machine; for example substitution is performed on occurrences of variables rather than on all occurrences at once (the linearisation step). In this sense the reduction step described above is similar to a *stack-free* Krivine machine.

## 4 Proof-Nets and Games

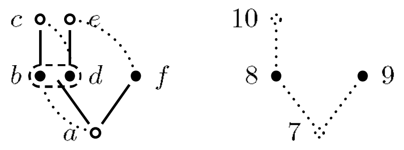
Following [Lam96, Sam98, Sam99], we can view proof-nets as encodings of innocent strategies. The moves in the game are the nodes in the proof-net, modulo the natural map $\rho$ from the proof-net to its intensional type. The P-moves are the $\bullet$-nodes and the O-moves are the $\circ$-nodes in the proof-net(intensional type). Now given the correspondence between proof-nets and $\lambda$-calculus presented in Sec. 3, P-moves correspond to occurrences of variables and O-moves correspond to sub-terms.

Given a proof-net $\mathcal{N}$, we can construct a strategy, $\sigma_{\mathcal{N}}$, corresponding to the game semantic denotation of the proof-net by tracing paths in the proof-net ([Sam99, Theorem 4.23]). The basic idea is to obtain a sequence of alternating $\circ$, and $\bullet$ nodes by

– starting with the root ($\mathcal{R}$) of the proof-net
– following axiom-links (dotted lines) for moving to a $\bullet$ node from a $\circ$ node
– and only visiting a $\circ$ node whose predecessor in the forest structure of the proof-net has already been visited.

We can view these paths as *paths of computation* through the proof-net.

*Example 8.* Consider the term $\lambda f, x.f(f(x))$. The proof-net representation and the intensional type (extracted from Example 7) are given below. Here the nodes of the proof-net are labelled for clarity.



The map $\rho$ from proof-net nodes to nodes in the intensional type is given as

$$\rho = \{ \ a \mapsto 7, b \mapsto 8, d \mapsto 8, c \mapsto 10, e \mapsto 10, f \mapsto 9 \ \}$$

The paths traced through the proof-net corresponding to game semantics are

$$\{ \ a.b, a.b.c.d, a.b.c.d.e.f \ \}$$

These paths, via $\rho$, give rise to the set of positons:

$$\{ \ 7.8, 7.8.10.8, 7.8.10.8.10.9 \ \}$$

that fully determines the strategy, $[\![\lambda f, x.f(f(x))]\!]$, on the arena corresponding to the intensional type. $\square$

In the above interpretation of proof-nets as strategies, the *arena* of the game is the intensional type of the term. However, the proof-net and therefore the intensional type could also include cut-links. A cut-link between two trees in the intensional type identifies pairs of *interacting moves* in the corresponding arena. Two nodes (moves) in the intensional type (arena) *interact* with one another if they are the "dual" nodes, across the cut, of a cut-formula. We also constrain the paths traced throught the proof-net by specifying that whenever a $\bullet$-node is reached that has a "dual" $\circ$-node across a cut, the next node to be visited is this "dual" node. This in fact corresponds to a natural notion of execution in game semantics that we outline below.

Game semantics has a natural notion of *execution* of the semantics. We start off the game with the initial O-move. The strategy then plays a P-move in response. This P-move may then *interact* (cf. Example 3) with some O-move in the arena and O plays this move. This is then followed by the strategy's response to this move and so on, until a P-move is played that does not interact with any other O-move. The trace produced by *executing* a strategy has the form

$$\circ \quad \bullet \bowtie \circ \quad \bullet \bowtie \circ \quad \ldots \quad \bullet \bowtie \circ \quad \bullet \bowtie \circ \quad \bullet$$

where we use $\circ$ to represent an O-move and $\bullet$ to represent a P-move, and where $\bowtie$ represents an interaction between a P and an O-move. The position constructed by *executing* the strategy as above exhibits a close connection with linear-head reduction of $\lambda$-terms.

Each pair of *interacting* P and O moves in the execution of a strategy corresponds to a single step of linear head-reduction ([Sam99, Theorem 5.14]). In terms of the syntax, each interacting pair of moves in the execution of a strategy can be interpreted as the *substitution* of a sub-term for an occurrence of a variable. It is this interpretation of *execution of games*, and its connection with linear head-reduction that forms the basis of control-flow analysis of a functional language using game semantics.

## 5 Control-Flow Analysis Using Games

Control-flow analysis (CFA) is a generic term for a range of analyses investigating the dynamics of executing a program. By control-flow, we mean that we are not interested in the flow of data in a program; rather, we are interested in how *control* flows from one function to another during the execution of a program. There are a number of characterisations of control-flow for higher-order functional languages [Shi91, Ses91, Deu92, Jon81, NN97]. We follow the approach to control-flow used in [Ses91]; this approach is also known as *closure analysis* and approximates the set of abstractions that can be bound to a given variable.

In this paper, we abstract the execution trace of game semantics to perform a variant of control-flow analysis generally known as 0-CFA. 0-CFA analysis is essentially a context-free analysis where we do not differentiate between different occurrences of variables. Therefore in the framework of game semantics we can ignore the differences between different occurrence of P-moves. Given a variable in the term, it corresponds to a P-move ($\alpha$ say) in the game semantics of the term. 0-CFA for the variable can now be formalised as follows:

> Capture the set of terms corresponding to all occurrences of O-moves that *interact* with any occurrence of the move $\alpha$ in the execution trace of game semantics.

## 6 Algorithm for Closure Analysis

The 0-CFA analysis computes a mapping, 0CFA, from variables to the set of *abstractions* in the term that get substituted for the variable. Note in particular that the range of 0CFA contains only abstraction sub-terms.

The idea of the analysis itself is quite simple. The first step is to translate the term to be analysed into its proof-net representation. We can then obtain the intensional type of the the proof-net from the forest structure of the proof-net (cf. Sec. 3). We also compute the map $\rho$ from the proof-net nodes to the intensional type. This mapping basically gives the move in the arena that corresponds to a given variable/sub-term.

Given a variable, $x$, the analysis first computes the P-move, $\rho(x)$, corresponding to the variable. The analysis then requires the computation of the O-move that interacts with $\rho(x)$. The interaction function, which is named $\bowtie$, can be easily computed from the cut-formulae in the forest structure of the intensional type: given a P-move, $\alpha$, the O-move interacting with it (if any) is the 'dual' node across a cut link in the forest structure of the intensional type. The control-flow analysis then returns the set of sub-terms that are mapped by $\rho$ to this interacting O-move; i.e.

$$\mathsf{0CFA}(x) = (\rho^{-1} \circ \bowtie \circ \rho)(x) \ .$$

*Example 9.* Consider the analysis of the term $(\lambda\, f, x.f(fx))(\lambda\, y.y)z$, referring to the proof-net representation in Example 7. We have already labelled the nodes in the intensional type.

Now given a variable, $f$ say, we have $\rho(f) = 8$. Now from the intensional type, we can see that $\bowtie(8) = 4$ and finally $\rho^{-1}(4) = \{\, \lambda\, y.y \,\}$. Therefore, we can conclude that $\mathsf{0CFA}(f) = \{\, \lambda\, y.y \,\}$.

Similarly, for the variables $x$ and $y$, we have $\rho(x) = 9$ and $\rho(y) = 6$. Also, from the intensional type, $\bowtie(9) = 5$ and $\bowtie(6) = 10$. Finally, $\rho^{-1}(5) = \{\, z \,\}$, and $\rho^{-1}(10) = \{\, f(x), x \,\}$. Therefore, we can conclude that $\mathsf{0CFA}(x) = \emptyset = \mathsf{0CFA}(y)$ since neither of these sets contain abstraction terms. □

It should be emphasised that the crux of the analysis is the computation of the interaction function, $\bowtie$, on the intensional type of the proof-net. The actual term is mainly used to compute the *type assignment* $\rho$, and its inverse $\rho^{-1}$, which are required for the *read-back* of the analysis. Computing the $\bowtie$ relation effectively performs the same task as the *closing-rules* of [Mos97a] and the conditional constraints of [Pal94], but 0CFA performs the computation on the type structure (intensional type) rather than on the term (proof-net).

In general, the procedure given above is not sufficient to compute 0-CFA. Given a variable $x$, of higher type, $\bowtie(\rho(x)) = a$ is a $\circ$-node in the intensional type. In general, the sub-term given by $\rho^{-1}(a)$ may not be in the suitable form of an abstraction: $\rho^{-1}(a)$ may be an application term (of higher-order type). In this case we need to perform a step of transitive closure, basically traversing computation paths, to obtain the set of abstraction sub-terms that $\rho^{-1}(a)$ can evaluate to. Transitive closure itself may require new computation paths to be added corresponding to the closing rules of [Mos97a]; this can be represented as a dynamic component of transitive closure.

Let us first consider the case when $a' = \rho^{-1}(a)$ is an application sub-term. Then, in general, the term corresponding to $a'$ is of the form

$$yM_1 \ldots M_m$$

Let the node in the proof-net corresponding to the head-variable $y$ be $b'$. From the translation of terms to proof-nets (Sec. 3), $b'$ is connected by an *axiom link* to $a'$ in the proof-net. Then if the O-move $a$ is played in the execution trace, the strategy will play the P-move $\rho(b') = b$. Now let the strategy play the move $\rho(c') = c$ *interacting* with the move $b$. This situation is represented as

$$\begin{array}{ccccc} \ldots & \circ & \bullet \bowtie \circ & \ldots \\ & a & b \bowtie c & \end{array}$$

The sub-term corresponding to the node $c'$ in the proof-net has the general-form

$$\lambda\, y_1 \ldots y_n.M$$

Then according to the interpretation of interacting moves, $\lambda\, y_1 \ldots y_n.M$ is substituted for the head-occurrence $y$. If $n > m$, then the resulting term after reduction is an abstraction of the form

$$\lambda\, y_{(m+1)} \cdots y_n.M$$

and we can conclude that the variable $x$ is bound to this abstraction. Otherwise, if $n \leq m$, then we have to perform the above steps again, starting with the node $c'$, until we reach an abstraction. This procedure is effectively the same as the procedure in [MH98b] to follow computation paths.

This procedure can be rephrased as a graph transitive-closure. The nodes of the graph are the nodes of the *intensional type*; and the edges of the graph are *projections* onto the intensional type, of computation paths in the proof-net. In the above example, there would be an edge from the node $a$ to the node $c$, indicating that the closure with body $a'$ evaluates to a closure with body $c'$.

We also need to capture the information concerning the number of abstractions or applications in a term. This can be done by associating a *cost* and *resource* to each edge. Given an edge from $a$ to $c$, as above, the cost of the edge is the number of *abstractions* in the proof-net node $a'$, and the resource of the edge is the number of *applications* in the node $b'$ in the proof-net (corresponding to the head variable $y$ above). Therefore, in the above example we would have an edge $a \xrightarrow{(0,m)} b$. A similar procedure is carried out for edges between $\bullet$-nodes in the intensional type. Intuitively, the cost and resource information is required to identify sub-terms of interest; since we are considering n-ary abstractions and applications, the cost and resource indicate the sub-term containing "cost" number of abstractions or "resource" number of applications. Effectively, cost-resource (CR) edges emulate the use of *labels* to identify sub-terms.

We can now compose two CR edges,

$$a \xrightarrow{(c_1,r_1)} b \xrightarrow{(c_2,r_2)} c$$

to give an edge $a \xrightarrow{(c,r)} b$ where

$$\begin{aligned} (c,r) &= (c_1, (r_1 - c_2) + r_2) && \text{if } r_1 > c_2 \\ (c,r) &= (c_1 + (c_2 - r_1), r_2) && \text{otherwise} \end{aligned}$$

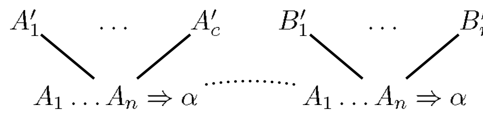
Transitive closure is performed using this definition of composition. The definition of CR edges and the operation of composition bear a remarkable resemblance to the KSL algorithm of [BKKS87]; the cost and resource correspond to the 'K' and 'L' components respectively and the composition operation is a simplified version of the $\oplus$ operation on KSL triples.

Now if there is an edge $a \xrightarrow{(0,r)} b$, then we can conclude that there is a *computation path* from $\rho^{-1}(a) = a'$ to $\rho^{-1}(b) = b'$. In particular, we can conclude that a closure with body $a'$ *may* compute to a closure with body $\mathsf{cut}(r, b')$, where

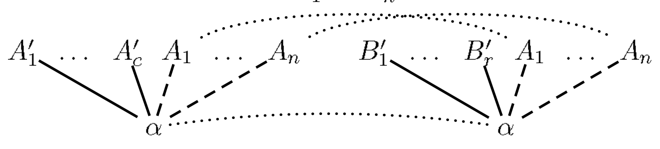
$$\mathsf{cut}(r, \lambda\, y_1 \ldots y_n.M) = \lambda\, y_{(r+1)} \cdots y_n.M$$

if $n \geq r$ and is undefined otherwise. This is essentially *read-back* of the analysis. Therefore, given a variable $x$, such that $\bowtie (\rho(x)) = a$, if $\rho^{-1}(a) = a'$ is an application sub-term, we need to compute all the *zero-cost* paths from $a$ to obtain the abstractions required for the analysis of $x$.

As we mentioned earlier, new CR edges computed during transitive closure may require additional edges to be added dynamically. These dynamically computed edges correspond to performing $\eta$-expansion on the term. Consider an axiom link between nodes $a$ and $b$ of type $A_1 \dots A_n \Rightarrow \alpha$


$A'_1$
$\dots$
$A'_c$
$B'_1$
$\dots$
$B'_r$
$A_1 \dots A_n \Rightarrow \alpha$
$A_1 \dots A_n \Rightarrow \alpha$

The root of the trees (of type $A_1 \dots A_n \Rightarrow \alpha$) can now be expanded and we can infer axiom links between the nodes $A_1 \dots A_n$ as



This corresponds to the $\eta$-expansion of the term

$$\lambda x_1 \dots x_c \,.\, y e_1 \dots e_r$$

where $y e_1 \dots e_r$ is of type $A_1 \dots A_n \Rightarrow \alpha$, as

$$\lambda x_1 \dots x_c x_1 \dots x_n \,.\, y e_1 \dots e_r x_1 \dots x_n$$

These inferred axiom links dynamically give rise to CR edges. If there is a CR edge $x \xrightarrow{(c,r)} y$ and suppose $z$ is the $m$th node immediately above $x$, where $m > c$, then following the above diagram we add an edge between nodes $z$ and $w$ where $w$ is the $((m-c)+r)th$ node immediately above $y$. We also set the cost and resource of this edge to be 0; i.e. $z \xrightarrow{(0,0)} w$.

We can now perform 0-CFA as follows:

- Encode terms into proof-nets.
- Compute the intensional type of the proof-net, and the mappings $\rho$ and $\rho^{-1}$.
- Compute the interaction relation $\bowtie$.
- Project the axiom links of the proof-net onto the intensional type as CR edges.
- Perform dynamic transitive closure ($\mathsf{DT}$), using a standard work-list algorithm [NNH99].
- Read back the analysis. This roughly corresponds to the function

  $$\mathsf{0CFA} = \mathsf{cut} \circ \rho^{-1} \circ \mathsf{DT} \circ \bowtie \circ \rho$$

  Here $\mathsf{DT}$ computes the CR edges from a given node, $a$, by performing dynamic transitive closure. The $\rho^{-1}$ operation considers nodes ($b$) connected to $a$ by *zero*-cost edges ($a \xrightarrow{(0,r)} b$); and the $\mathsf{cut}$ operation removes $r$ number of leading $\lambda$s from $\rho^{-1}(b)$.

**Theorem 1 (Soundness of analysis).** *The algorithm for performing 0-CFA analysis is sound with respect to linear head-reduction.*

*Proof.* This theorem essentially follows from the fact that the dynamic transitive closure only follows paths that are specified by game semantics; i.e. each step of the dynamic transitive closure is sound with respect to the game semantics [Sam99, Theorem 7.6]. □

We note that the soundness proof above only captures soundness of the algorithm with respect to game semantics. The soundness with respect to linear head-reduction follows from the connection between game semantics and linear head-reduction ([Sam99, Theorem 5.14]), which is a general semantic property of game semantics.

*Example 10.* We illustrate the algorithm on the term $(\lambda x.x)(\lambda y.y)(\lambda z.z)$. Note that this term is not fully $\eta$-expanded; variable $x$ of $\lambda x.x$ is of higher-order type. Similarly with $\lambda y.y$. Fig. 1 gives the proof-net representation and intensional type for this term. The CR edges obtained by projecting the axiom-links of the

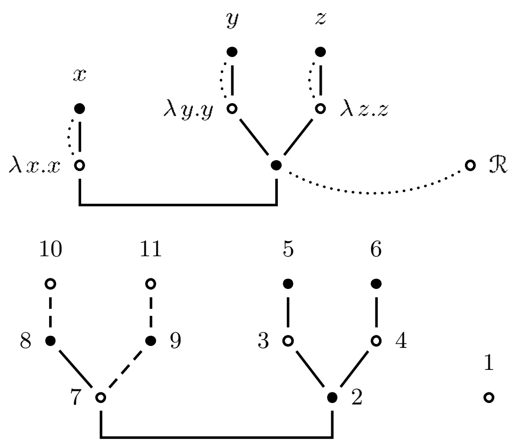


**Fig. 1.** Proof-net representation of $(\lambda x.x)(\lambda y.y)(\lambda z.z)$

proof-net are

$$7 \xrightarrow{(1,0)} 3 \quad 3 \xrightarrow{(1,0)} 10 \quad 4 \xrightarrow{(1,0)} 11 \quad 1 \xrightarrow{(0,2)} 7$$
$$8 \xrightarrow{(0,1)} 2 \quad 5 \xrightarrow{(0,1)} 8 \quad 6 \xrightarrow{(0,1)} 9$$

Performing dynamic-transitive closure on these edges gives rise to the edges

$$9 \xrightarrow{(0,0)} 5 \quad 1 \xrightarrow{(0,1)} 3 \quad 7 \xrightarrow{(2,0)} 10 \quad 10 \xrightarrow{(0,0)} 4$$
$$5 \xrightarrow{(0,2)} 2 \quad 9 \xrightarrow{(0,1)} 8 \quad 1 \xrightarrow{(0,0)} 10 \quad 3 \xrightarrow{(1,0)} 4$$
$$7 \xrightarrow{(2,0)} 4 \quad 9 \xrightarrow{(0,2)} 2 \quad 1 \xrightarrow{(0,0)} 4$$

Now computing the function $\mathsf{DT} \circ \bowtie \circ \rho$ for the variables $x$, $y$ and $z$ gives us the mapping

$$\begin{aligned} x &\mapsto \{\, 3, \xrightarrow{(1,0)} 4, \xrightarrow{(1,0)} 10 \,\} \\ y &\mapsto \{\, 10, \xrightarrow{(0,0)} 4 \,\} \\ z &\mapsto \{\, 11 \,\} \end{aligned}$$

Now performing $\mathsf{cut} \circ \rho^{-1}$ for the zero-cost reachable nodes in the sets, we obtain the analysis

$$\begin{aligned} x &\mapsto \{\, \lambda y.y \,\} \\ y &\mapsto \{\, \lambda z.z \,\} \\ z &\mapsto \emptyset \end{aligned}$$

□

*Complexity* The algorithm for 0-CFA is computed by a dynamic-transitive closure on the intensional-type structure of the term. The complexity of the analysis is $\mathcal{O}(n^3)$ in the size of the term. Intuitively, the algorithm considers $\mathcal{O}(n^2)$ number of edges[3] and each step of the dynamic-transitive closure performs $\mathcal{O}(n)$ amount of work; this is comparable with similar results in the literature [Pal94, MH98b].

Our algorithm is also related to the type-based algorithms developed in [Mos97b, HM97a]. These approaches perform (explicit or implicit) $\eta$-expansion (based on the type-structure) of the program, and then use a simple transitive closure procedure to obtain the analysis. The algorithm we present here does not require an $\eta$-expansion step, but on the other hand requires a dynamic-transitive closure procedure. As a result of this tradeoff, the algorithm presented here does not have some of the pleasant properties of the algorithms in [Mos97b, HM97a]. For example, it is not possible to obtain control-flow information for individual variables independently of the rest of the analysis; this is a result of performing dynamic-transitive closure as opposed to simple transitive closure.

The runtime complexity of our algorithm is $\mathcal{O}(n^2 m)$, where $n$ is the size of the program and $m$ is the size of the partial intensional type required by the algorithm, and where $m$ is bounded above by $n$ in the worst case. This is in line with state-of-the-art 0-CFA algorithms that exhibit a $\mathcal{O}(n^3)$ complexity. Compared to the type-based 0-CFA algorithms that exhibit a quadratic complexity [Mos97b, HM97a], our algorithm does not make the assumption that the size of the type-annotated term is linear in the size of the untyped term[4].

[3] The $\mathcal{O}(n^2)$ figure is not obvious from the formulation of the CR edges in the algorithm, but it can be shown that the number of edges considered in the algorithm can be bounded by $\mathcal{O}(n^2)$ [Sam99].

[4] In general, the size of the type-annotated term is arbitrarily larger than the size of the untyped term

*Interpretation as constraints* The algorithm for 0-CFA given above has a close relation (as expected) with the constraint based formulations of the analysis [Pal94]. Each of the CR edges considered in the algorithm can be interpreted as an inclusion constraint on the λ-term. Each edge, $a \xrightarrow{(c,r)} b$, between ∘-nodes $a$ and $b$, can be interpreted as a constraint

$$\mathsf{cut}(r, \rho^{-1}(b)) \subseteq \mathsf{cut}(c, \rho^{-1}(a))$$

and each edge $l \xrightarrow{(c,r)} m$, between •-nodes $l$ and $m$, can be interpreted as a constraint

$$\mathsf{take}(c, \rho^{-1}(l)) \subseteq \mathsf{take}(r, \rho^{-1}(m))$$

where the function take is defined as

$$\mathsf{take}(r, xM_1 \ldots M_n) = xM_1 \ldots M_r$$

if $n \geq r$ and is undefined otherwise.

## 7 Conclusion and Future Work

We have presented a technique of performing control-flow analysis for a functional language. The analysis is based on a game semantic model of the language. We have also presented an algorithm for the analysis, using proof-nets as the syntactic representation of programs. The algorithm presented has state-of-the-art performance, and the proof of correctness essentially follows from the game semantic model.

Although we have focussed on the simply-typed λ-calculus, the techniques presented in this paper can easily be applied to a richer language like PCF containing base-type constants, conditionals and recursion.The game-semantic model we have used in this paper is actually a model for PCF [AJM94, HO94]. Therefore the specification of the analysis itself need not be changed to accommodate PCF.

The proof-net representation of λ-terms can also be extended to include PCF constructs [Mac94]; in particular we can still obtain a isomorphism between the graph and proof-net representations of PCF terms. The proof-net representation of PCF terms will include special nodes for constants, a special form of axiom-link for conditionals and a "looping" cut-link to represent recursion. From the point of view of the analysis algorithm, these graphs are exactly the same as the graphs for simply typed λ-terms, and only a minimal change is required in the algorithm to deal with PCF terms.

There is no necessity to stop at PCF. The game semantic models of programming languages often exhibit very elegant *factorisation results*, which allow language features to be added in a modular fashion. We expect to be able to use the semantic machinery of games to extend the analysis to languages with additional features like side-effects, continuations, non-determinism, etc. In fact the specification of the analysis is expected to be very similar to the ones in this paper; the challenging issue is obtaining representations analogous to proof-nets for developing efficient algorithms.

## References

[Hy 9] Hypatia papers archive. `http://hypatia.dcs.qmw.ac.uk`, 1996–9.

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Proceedings of LICS'98*. IEEE Press, 1998.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In *Proceedings of TACS'94*, volume 789 of *Lecture Notes in Computer Science*, pages 1 – 15. Springer-Verlag, 1994.

[AM97] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. In Peter W O'Hearn and R D Tennent, editors, *Algol-like Languages*, volume 2. Birkhaüser, 1997.

[BKKS87] Henk Barendregt, J R Kennaway, J W Klop, and M R Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 74:191–231, 1987.

[Dan90] Vincent Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (Principalement du Lambda-calcul)*. PhD thesis, Université Paris VII, 1990.

[Deu92] Alain Deutsch. *Modeles Operationnels de Langage de Programmation et Representations de Relations Sur des Langages Rationnels avec Application a la Determination Statique de Proprietes de Partages Dynamiques de Donnees*. PhD thesis, Universite Paris VI, 1992.

[DHR96] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics and abstract machines. In *Proceedings of LICS'96*. IEEE Press, 1996.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1 – 102, 1987.

[Har99] Russell Harmer. *Games and full abstraction for non-deterministic languages*. PhD thesis, University of London, 1999.

[HM97a] Nevin Heintze and David McAllester. Linear time subtransitive control flow analysis. In *Proceedings of PLDI'97*, pages 261–272. ACM SIGPLAN, June 1997.

[HM97b] Nevin Heintze and David McAllester. On the cubic-bottleneck of subtyping and flow analysis. In *Proceedings of LICS'97*. IEEE Press, 1997.

[HO94] J. Martin E. Hyland and C.-H L. Ong. On full abstraction for PCF: Parts I, II and III. Available from [Hy 9], 1994.

[Jon81] Neil D Jones. Flow analysis of lambda expressions. In S Even and O Kariv, editors, *Proceedings of ICALP'81*, number 115 in Lecture Notes in Computer Science, pages 114–128, Acre (Akko), Israel, July 1981. Springer-Verlag.

[Lam94] François Lamarche. Proof nets for intuitionistic linear logic I: Essential nets. Available from [Hy 9], 1994.

[Lam96] François Lamarche. From proof nets to games: extended abstract. *ENTCS*, 1996.

[Mac94] Ian Mackie. *The Geometry of Implementation*. PhD thesis, University of London, 1994.

[McC98] Guy McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Distinguished dissertations. Springer-Verlag, 1998.

[MH98a] Pasquale Malacaria and Chris Hankin. Generalised flowcharts and games: Extended abstract. In Kim G Larsen, Sven Skyum, and Glynn Winskel, editors, *25th International Colloquium, ICALP'98*, Aalborg, Denmark, July 1998.

[MH98b] Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In Koskimies Kai, editor, *Proceedings of CC'98*, volume 1383 of *Lecture Notes in Computer Science*, Lisbon, Portugal, 1998. Springer.

[MH99] Pasquale Malacaria and Chris Hankin. Non-deterministic games and program analysis: An application to security. In *Proceedings of LICS'99*. IEEE Press, 1999.

[Mil89] Robin Milner. *Communication and Concurrency*. International series in computer science. Prentice Hall, 1989.

[Mos97a] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, 1997.

[Mos97b] Christian Mossin. Higher-order value flow graphs. In *Proceedings of 9th International Symposium on Programming Languages, Implementation, Logics, and Programming (PLILP'97)*, 1997.

[MP92] Gianfranco Mascari and Marco Pedicini. Head linear reduction and pure proof net extraction. In *Theoretical Computer Science*, volume 135, pages 111 – 137, 1992.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of POPL'97*. ACM Press, 1997.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Pal94] Jens Palsberg. Global program analysis in constraint form. In Sophie Tison, editor, *Proceedings CAAP'94*, volume 787 of *Lecture Notes in Computer Science*, pages 276–290. Springer-Verlag, 1994.

[Sam98] Prahladavaradan Sampath. On abstract machines and games. In Valeria de Paiva and Eike Ritter, editors, *Proceedings of the ESSLLI'98 Workshop on Logical Abstract Machines*, Technical Report CSR-98-8. University of Birmingham, 1998.

[Sam99] Prahladavaradan Sampath. *Program Analysis using Game Semantics*. PhD thesis, University of London, 1999.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1991.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages (or) Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Wad71] C Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.

# On Extracting Static Semantics

John Hannan[⋆]

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA

**Abstract.** We examine the problem of automatically extracting a static semantics from a language's semantic definition. Traditional approaches require manual construction of static and dynamic semantics, followed by a proof that the two are consistent. As languages become more complex, the static analyses also become more complex, and consistency proofs have typically been challenging. We need to find techniques for automatically constructing static analyses that are provably correct. We study the issues of developing such techniques and propose avenues of research in this area. We find that significant advances are required before realizing the practical extraction of static semantics from language definitions.

## 1 Introduction

A common approach to programming language design and implementation is to identify static (compile time) and dynamic (run time) operations or properties of the language and then construct a static semantics and a dynamic semantics. Even if formal language definitions do not consist of such explicit phases, compilers for these languages almost always do make such distinctions, performing semantic analysis before generating code.

A simple compiler-design principle might be summarized by the following phases:

1. Lexical Analysis
2. Semantics Analysis
3. Code Generation

Alternatively, the approach to language design mentioned above can be summarized by:

1. Static Semantics
2. Dynamic Semantics

These two phases correspond to the latter two phases of a compiler. Just as code generation uses information provided by the semantics analysis, the dynamic semantics can use information or properties provided by the static semantics.

[⋆] This work is supported in part by NSF Award #CCR-9900918.

In a strongly typed language, this information consists of a well-typed result that allows the dynamic semantics to avoid any typechecking, as in the language Standard ML [11].

Because types play a significant role in most static analyses, the static phase of a language's definition is typically given as a type system. Types characterize the meaning of expressions. To ensure that the static semantics provides valid information to the dynamic semantics, some kind of consistency result is required. This result states that an expression's value will have the same type as the expression, and hence, type errors will not occur at runtime. Constructing such proofs by hand can be a difficult task, especially as languages become more complex.

An alternative approach to semantics defines a single semantics that contains both the static and dynamic checks. One way of viewing such a combined semantics is that it manipulates both *syntactic objects* (available at compile time) and *semantic objects* (available at run time). Splitting and separating such a semantics into its static and dynamic parts is a challenging problem. A few previous efforts in this area have demonstrated the possibilities and the limitations of this idea, and they can be categorized based on the kinds of semantics they use and degree of automation. An early approach tried to manually construct typing semantics from a given semantics and then prove consistency [1,12]. Additional manual transformations are then used to construct the dynamic semantics.

Towards automating this process, some researchers have applied techniques from partial evaluation [7]. When a partial evaluator is applied to a semantics definition, it produces a compiler. This compiler hopefully evaluates the static parts of an input program, performing some of the static analysis. Unfortunately there is no guarantee that the semantics contains a kind of static typing that can be extracted.

The more general problem of deriving a compiler from a semantics has been studied by several researchers over the past 25 years or so [10,13,15,16,18]. While these efforts focus on code generation, not static typing, they do demonstrate how a semantics, taken as a definitional interpreter, can be decomposed into compile-time and run-time phases.

Doh and Schmidt [3] demonstrate how typing rules can be derived from action semantics [14] by taking semantic equations (expressed in action semantics notation), constructing corresponding typing equations, and then translating these equations into inference rules. The corresponding dynamic semantics can then be constructed by specializing the general equations using the typing semantics. They demonstrate their techniques on a simple first-order expression language.

In the next section we consider the application of partial evaluation to this problem, as pioneered by Neil Jones. Then we move on to consider a new proposal based on our previous research results and demonstrate its promise and limitations.

## 2 A Partial-Evaluation Approach

In 1991, Neil Jones considered the problem of constructing a type checker from a semantic specification [6]. His focus is specifically on type systems, not the more general problem of static properties. But as types are the most important static property in use today, starting with them is sensible.

Since the problem of constructing a type checker from a semantics is one of introducing stages of computation - compile time and run time - it's only natural to consider traditional staging transformations [8], particularly partial evaluation, as a technique for separating the static and dynamic parts of a semantics. Jones did this and we reproduce here some of his notation and observations.

First we should understand exactly what the problem is. Given an operational semantics that manipulates both syntactic objects and semantic objects, we want to (automatically) separate compile-time operations and runtime operations. We are specifically concerned with error checking. But how exactly do we define and identify compile-time operations occurring in a semantics?

Let $\mathbf{L}$ be a programming language. Then $[\![p]\!]_{\mathbf{L}} v$ is the result of running $\mathbf{L}$-program $p$ on input $v$. A *Definitional interpreter int* for language **S** written in **L** has the following property:

$$[\![int]\!]_{\mathbf{L}}(source, input) \;=\; [\![source]\!]_{\mathbf{S}}\, input$$

Errors in **S**-programs are realized by error-detecting code in *int*. Given this setting we are able to describe static semantics:

> *Static Semantics* is a mechanism for detecting whether or not a given **S**-program can cause *int* to execute a "static error" on some input.

The notion of static error is interesting. Do we take static semantics as providing the definition of what the static errors are? Or is there some other definition of static error from which our definition of static semantics can be proved sound and complete? Considering this further, we see that we really have to address two problems in general:

1. Determine the static properties of a semantics
2. Construct a static semantics that captures these properties

The question of whether static semantics define the static errors is analogous to existing views on type systems:

1. *Prescriptive* (Church): types are predefined conditions to ensure meaningfulness;
2. *Descriptive* (Curry): types describe the values that a program manipulates.

We consider a semantics that contains a descriptive use of types and attempt to construct a semantics that uses the prescriptive view.

Following Jones [6], we can make the concept of static error more precise, using the concept of program specialization called partial evaluation. Assume we have a partial evaluator *mix*. Recall that a partial evaluator takes a program $p$ and part of the program's input $s$ (the static part) and yields a new program that when applied to the dynamic part $d$ behaves as the original program does:

$$mix(p, s)\,(d) \;=\; p(s, d)$$

Partial evaluation provides a staging of computation into two parts, typically described as the static part (the computation $mix(p, s)$) and the dynamic part (the application of the residual program to $d$).

Of interest to us in when the program is a definitional interpreter. In this case the static input is the source program to be interpreted, (the dynamic input is the input to the source program) and the above equation can be rewritten as

$$int_{source} \;=\; [\![mix]\!]_{\mathbf{L}}(int, source)$$

If the interpreter contains error checking (including typechecking) then we might hope that partial evaluation performs these tests and that the residual program $p_s = mix(p, s)$ contains no such error checking. In this case, we have solved the problem of staging typechecking, though without constructing an explicit type system. Unfortunately, this idea doesn't work in practice because the restriction of no error checking in the residual program is too strong, as this rules out any kind of errors, even dynamic ones (e.g., divide by zero), from ever occurring in programs.

The problem that arises with this approach is that we fail to distinguish between static and dynamic errors. To solve this problem Jones observes that a static errors should be ones that can only be performed by the partial evaluator (operating on *int*) and code for recognizing these errors will never be generated in a residual program. Such a property can be detected by a binding-time analysis (BTA). BTA divides every basic function call or data constructor into one of two classes: those that can be computed at program specialization time, and those that must be performed at run time (by a specialized program $p_s$) [6]. Thus the problem of identifying static errors in an interpreter reduces to performing a binding-time analysis on it when we consider the source program as static input.

Jones demonstrates how BTA, combined with partial evaluation, can be used to identify static properties (type checking) and to perform them separately from the evaluation of the program.

## 3 A Pass-Separation Approach

A fundamental limitation of using partial evaluation to specify static semantics is that we do not explicitly construct a type system or some other specification of the static operations. Instead, a general partial evaluator implicitly performs these operations. An alternative to partial evaluation is another staging transformation called pass separation [8]. Recall that staging transformations are, in general, methods of separating stages of computations based on the availability of data, with the most common application being developing compilers from interpreters. Partial evaluation is perhaps the most widely known and used staging transformation, but others exist, including traditional compiler optimizations (e.g., constant folding) and pass separation.

Consider the general problem addressed by staging transformations: Given a program $p$ with two inputs $x, y$, *stage* the computations performed by $p(x, y)$ into two sequential phases:

– one taking $x$ as input
– one taking $y$ as input

Partial evaluation accomplishes this via a single program *mix* such that if $mix(p, x) = p_x$ then $p(x, y) = p_x(y)$ for all inputs $x$ and $y$. In the example from above, $p$ is an interpreter, $x$ is the source program to be interpreted, and $y$ is the input to the program. Pass separation requires a technique to construct two new programs, $p_1$ and $p_2$ such that $p(x, y) = p_2(p_1(x), y)$ for all inputs $x$ and $y$. In this case, if $p$ is an interpreter and $x$ is a source program, then we might expect $p_1$ to be a compiler and $p_2$ to be the evaluator for the target language of the compiler. The drawback of pass separation is that we do not have a general way of taking an arbitrary program $p$ and constructing the required new programs $p_1$ and $p_2$ such that $p_1$ performs some non-trivial computations. (We could always let $p_1$ be the identity function and let $p_2 = p$.) Arbitrary functions are just too general, and the possible divisions into two functions too numerous to support a simple means for performing pass separation.

In previous work we addressed this problem by considering a restricted form of interpreter, namely abstract machines [4]. In this work, an abstract machine is represented by a set of rewrite rules $st \Rightarrow st'$ in which $st$ and $st'$ are machine states. Typically, machine states consist of program and data. Each rule of the machine specifies how a given program state manipulates data to yield a new program state. Rewriting in this system is particularly simple as we only allow rewriting at the top level, i.e., the entire term. This simple form of computation is expressive, but also extremely convenient to reason about because we only have a single form of computation (machine-state rewrite) to consider.

As an example we studied the CLS machine [5], a variant of Landin's SECD machine [9]. This is a machine that performs call-by-value reduction on the lambda calculus. The machine state consists of a configuration $\langle C, (L, S)\rangle$ consisting of

***Code*** - a sequence of expressions to be evaluated
***L*** - a sequence of environments
***S*** - a stack of intermediate values

The terms of the language are given by the following grammar:

$$e ::= \; 1 \mid e + 1 \mid \lambda\, e \mid e\, e \mid \mathsf{ap}$$

Observe that we use de Bruijn indices for variables. Code or a program is given by a list of terms. An environment is a list of values. Values are simply closures

$$\langle (e_1\, e_2)\,;C,\ \ (\rho\,;L,\ S)\rangle \Rightarrow \langle e_1\,;e_2\,;\mathsf{ap}\,;C,\ \ (\rho\,;\rho\,;L,\ S)\rangle$$
$$\langle (\lambda\, e)\,;C,\ \ (\rho\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ \{\rho,\lambda\, e\}\,;S)\rangle$$
$$\langle 1\,;C,\ \ ((\rho;v)\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ v\,;S)\rangle$$
$$\langle (e+1)\,;C,\ \ ((\rho;v)\,;L,\ S)\rangle \Rightarrow \langle e\,;C,\ \ (\rho\,;L,\ S)\rangle$$
$$\langle \mathsf{ap}\,;C,\ \ (L,\ v\,;\{\rho,\lambda\, e\}\,;S)\rangle \Rightarrow \langle e\,;C,\ \ ((\rho;v)\,;L,\ S)\rangle$$

**Fig. 1.** The CLS Machine

consisting of a term and an environment. A stack is also a list of values. The rules for the machine are given in Figure 1.

The idea behind our pass-separation technique is to consider abstract machines with rules of the form

$$\langle s, d\rangle \longrightarrow \langle s', d'\rangle$$

in which $s$ and $s'$ represent static (compile-time) components and $d$ and $d'$ represent dynamic (runtime) components. For the CLS machine we take the $C$ component to be the static part and the pair $(L, S)$ to be the dynamic part. We then decompose each rule into two parts: one operating only on the static part and one operating on the dynamic part. For example, a rule of the form

$$\langle s, d\rangle \longrightarrow \langle s', d'\rangle$$

would be decomposed into a pair of rules

$$\langle s, Y\rangle \longrightarrow \langle s_1, Y\rangle \ \text{ and } \ \langle s_2, d\rangle \longrightarrow \langle s', d'\rangle.$$

The terms $s_1$ and $s_2$ are constructed to ensure that rewriting proceeds in lock step with the original system. Applying this simple transformation yields two sets of rewrite rules (abstract machines):

– a compiler (introducing a new machine language)
– an interpreter for the new language

The construction of these new rules is based on the form of the original rules. Because the rules have such a simple structure, we can identify a few distinct cases that give rise to meta-rules - rules on how to construct new rules. These meta-rules are based on the dependencies among $s$, $d$, $s'$, and $d'$. In particular, we consider how the terms $s'$ and $d'$ are constructed from $s$ and $d$. (See [5] for a detailed description of these meta rules.)

Applying the meta-rules to the CLS machine we arrive at the two machines given in Figure 2. The new machine language consists of instructions push, lam, fst, snd and ap. These instructions are generated by the meta-rules and the

$$\begin{aligned}
\mathsf{ev}(e_1\, e_2) \,;\, C &\Rightarrow \mathsf{push} \,;\, \mathsf{ev}(e_1) \,;\, \mathsf{ev}(e_2) \,;\, \mathsf{ap} \,;\, C \\
\mathsf{ev}(\lambda\, e) \,;\, C &\Rightarrow \mathsf{lam}(\mathsf{ev}(e) \,;\, \mathsf{nil}) \,;\, C \\
\mathsf{ev}(1) \,;\, C &\Rightarrow \mathsf{fst} \,;\, C \\
\mathsf{ev}(e+1) \,;\, C &\Rightarrow \mathsf{snd} \,;\, \mathsf{ev}(e) \,;\, C
\end{aligned}$$

$$\begin{array}{rcl}
\langle \mathsf{push}; C,\ \rho \,;\, L,\ S\rangle & \Rightarrow & \langle C,\ \rho \,;\, \rho \,;\, L,\ S\rangle \\
\langle (\mathsf{lam}\, C'); C,\ \rho \,;\, L,\ S\rangle & \Rightarrow & \langle C,\ L,\ (\rho, \mathsf{lam}\, C') \,;\, S\rangle \\
\langle \mathsf{fst}; C,\ (v,\rho) \,;\, L,\ S\rangle & \Rightarrow & \langle C,\ L,\ v \,;\, S\rangle \\
\langle \mathsf{snd}; C,\ (v,\rho) \,;\, L,\ S\rangle & \Rightarrow & \langle C,\ \rho \,;\, L,\ S\rangle \\
\langle \mathsf{ap}; C,\ L,\ v \,;\, (\rho, \mathsf{lam}\, C') \,;\, S\rangle & \Rightarrow & \langle C'C,\ (v,\rho) \,;\, L,\ S\rangle
\end{array}$$

**Fig. 2.** The Separated Machines

rules giving meaning to them provide a definition for this new abstract machine language.

The resulting machines can be proved correct with respect to the original machine. Interestingly, this new machine is essentially the Categorical Abstract Machine [2]. The compiler decomposes the structure of a lambda term into simpler components (new machine language), so we see a separation between the traversal of the original term's structure and the actual evaluation of the term.

## 4 Pass Separation for Static Semantics

Can pass separation be used to separate the static semantics inherent in an abstract machine? We explore this idea by considering the issues involved and give an example constructed through some reverse engineering. Three questions immediately must be considered:

- Can we identify an appropriate form of the operational semantics? Even restricting ourselves to abstract machines, we still want want to understand the best form that will allow us to separate out a static semantics.
- Can we identify static errors? A more general problem is simply identifying the static properties of a semantics. This is analogous to defining what static errors are.
- Can we identify a set of meta-rules for decomposing static and dynamic properties? Our previous work using pass separation identified a small set of meta-rules for constructing new rewrite rules. These meta-rules were justified with correctness proofs. We would like to find a general set for the current problem

We do not give any definitive answers to these questions. Instead we will work through an example to demonstrate that the ideas of pass separation can be applied to the problem of separating static semantics.

We begin by modifying the abstract machine providing the semantics for our language. The CLS machine is an untyped machine: no typechecking is done in the machine; terms have no associated types. To provide the most information possible, we initially assume that terms are explicitly typed: every term (and subterm) is explicitly tagged with its type (written as a superscript). While this is perhaps not realistic (as it assumes some pre-existing type system that inserts these types), this assumption is a useful starting point allowing us to see how we might separate static and dynamic operations. We modify the CLS machine to manipulate such explicitly typed terms and to include certain type checks. In particular, the machine checks that the value of an argument in a function call has the same type as the formal parameter bound to it. The machine also checks that the value of a variable (found in an environment) has the same type as the variable. The modified machine is given in Figure 3. An environment $\delta$ maps variables (de Bruijn indices) to typed values (i.e., values and their types).

$$\begin{aligned}
&\langle (m\,n)^{\tau}\,;C,\ \ (\delta\,;L,\ S)\rangle \Rightarrow \langle m^{\tau_2\to\tau}\,;n^{\tau_2}\,;\mathsf{ap}\,;C,\ \ (\delta\,;\delta\,;L,\ S)\rangle\\
&\langle (\lambda\,m)^{\tau_1\to\tau_2}\,;C,\ \ (\delta\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ \{\delta,\lambda\,m\}^{\tau_1\to\tau_2}\,;S)\rangle\\
&\langle 1^{\tau}\,;C,\ \ ((\delta;v^{\tau})\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ v\,;S)\rangle\\
&\langle 1^{\tau}\,;C,\ \ ((\delta;v^{\tau'})\,;L,\ S)\rangle \Rightarrow \mathit{error}\ \mathsf{if}\ \tau\neq\tau'\\
&\langle (m+1)^{\tau}\,;C,\ \ ((\delta;v)\,;L,\ S)\rangle \Rightarrow \langle m^{\tau}\,;C,\ \ (\delta\,;L,\ S)\rangle\\
&\langle \mathsf{ap}\,;C,\ \ (L,\ v^{\tau_2}\,;\{\delta,\lambda\,m\}^{\tau_2\to\tau}\,;S)\rangle \Rightarrow \langle m^{\tau}\,;C,\ \ ((\delta;v^{\tau_2})\,;L,\ S)\rangle\\
&\langle \mathsf{ap}\,;C,\ \ (L,\ v\,;\{\delta,\lambda\,m\}\,;S)\rangle \Rightarrow \mathit{error}\ \mathsf{otherwise}
\end{aligned}$$

**Fig. 3.** Explicitly Typed Machine

We can restructure this machine by making some simple syntactic changes. Specifically, we decompose $\delta$ into a pair of environments: $\Gamma$ (mapping variables to types) and $\rho$ (mapping variables to values). The resulting machine is given in Figure 4. We take as given that the static property we want to extract is the association of an expression and a type. Automatically identifying this property and structuring the machine in such a way as to make this property evident is a challenging problem, and one for which we currently offer no solution. But once we have the machine in Figure 4, we can consider how to achieve our goal. Ultimately, our goal is not simply to produce a static semantics, but to provide some static checking that when satisfied indicates an absence of certain kinds of runtime errors.

$$\langle [\Gamma, (m\, n)]^{\tau} ; C, \quad \rho ; L, \ S\rangle \Rightarrow \langle [\Gamma, m]^{\tau_2 \to \tau} ; [\Gamma, n]^{\tau_2} ; \mathsf{ap} ; C, \quad \rho ; \rho ; L, \ S\rangle$$

$$\langle [\Gamma, \lambda\, m]^{\tau_1 \to \tau_2} ; C, \quad \rho ; L, \ S\rangle \Rightarrow \langle C, \quad L, \ \{\rho, [\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}\} ; S\rangle$$

$$\langle [(\Gamma; \tau), 1]^{\tau} ; C, \quad (\rho; v)\ L, \ S\rangle \Rightarrow \langle C, \quad L, \ v ; S\rangle$$

$$\langle [(\Gamma; \tau'), 1]^{\tau} ; C, \quad (\rho; v)\ L, \ S\rangle \Rightarrow \mathit{error} \text{ if } \tau \neq \tau'$$

$$\langle [(\Gamma; \tau'), (m+1)]^{\tau} ; C, \quad (\rho; v)\ L, \ S\rangle \Rightarrow \langle [\Gamma, m]^{\tau} ; C, \quad \rho\ L, \ S\rangle$$

$$\langle \mathsf{ap} ; C, \quad L, \ v^{\tau_2} ; \{\rho, [\Gamma, \lambda\, m]^{\tau_2 \to \tau}\} ; S\rangle \Rightarrow \langle [(\Gamma; \tau_2), m]^{\tau} ; C, \quad (\rho; v)\ L, \ S\rangle$$

$$\langle \mathsf{ap} ; C, \quad L, \ v ; \{\rho, \Gamma, \lambda\, m\} ; S\rangle \Rightarrow \mathit{error} \text{ otherwise}$$

**Fig. 4.** A Restructured Machine

Because our goal is to avoid error states in the abstract machine we will attempt to identify states that can never lead to an error state. Fortunately, we have explicit descriptions of error states in our machine. How do we know if a particular machine state can ever lead to an error state? If we wish to avoid reaching an error state, then forward reasoning tells us very little. Since we have no a priori definition of a safe state, we cannot reason that one safe state leads to another (as forward reasoning would do). Backwards reasoning, however, lets us reason from assumed safe states to other safe states. If a machine is in a safe state, then the previous state it was in led to that safe state. Consider the following rules for safe states:

1. Any final state is a safe state;
2. if $s$ is a safe state and $s' \Rightarrow s$ is an instance of some rule in the machine then $s'$ is also a safe state.

If we take the least-fixed-point (inductive) interpretation of these two rules (operating over the space of finite and infinite computation sequences) then we capture only safe states leading to final states (finite computations). If instead we take the greatest-fixed-point (co-inductive) interpretation of the rules, then we capture safe states of both terminating and non-terminating computations (that avoid non-safe states). This definition seems more appropriate as many well-typed programs in strongly-typed languages can fail to terminate. Note that condition (2) requires that the machine be deterministic: the only step possible from state $s'$ must be to state $s$. This observation is part of a formal theory that must be developed to formally justify the steps we are taking to construct a static semantics.

Now that we have a general strategy for determining safe states we would like to extract the static properties. Since the type errors we wish to avoid are based on the property of type associated with expressions, we can understand error states to be states in which an incorrect type is associated with an expression. This observation leads us to a simple meta-rule for constructing inference rules for a static semantics:

For each rewrite rule $s \Rightarrow s'$ in the machine, let $A$ and $A'$ be the set of static properties occurring in $s$ and $s'$, respectively. If $A$ is not a singleton set, then the method fails; otherwise construct the inference rule

$$\frac{A'}{A}$$

If we apply this meta-rule to the rules in Figure 4, then we get the set of inference rules in Figure 5. The second rule provides no information so it can be deleted. The remaining rules are the traditional rules for typechecking the lambda calculus.

$$\frac{[\Gamma, m]^{\tau_2 \to \tau} \qquad [\Gamma, n]^{\tau_2}}{[\Gamma, (m\,n)]^{\tau}}$$

$$\frac{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}$$

$$\frac{}{[(\Gamma; \tau), 1]^{\tau}}$$

$$\frac{[\Gamma, m]^{\tau}}{[(\Gamma; \tau'), (m+1)]^{\tau}}$$

$$\frac{[(\Gamma; \tau_2), m]^{\tau}}{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}$$

**Fig. 5.** Inference Rules for Static Typechecking

While we have informally justified the construction of these rules, arguing that the original rewrite rules provide enough information to extract inference rules, we have not provided a formal proof that the inference system provides safety against runtime type errors from occurring. Neither have we characterized a general form of abstract machines for which this technique applies. Still, we hope this example sheds some light on the problem of extracting a static semantics from an operational one.

## 5 From Scheme to Simple Types?

Where did the types in the previous section come from? We assumed that terms came with their types already attached and that the semantics used those types to perform checks. These assumptions stacked the deck in our favor, helping us to achieve our goal of constructing a type system. As a first step, we think this is fair, providing us with as much information as possible. One might argue that what our example really shows is part of the proof of type soundness, telling us that we can erase the types at runtime. But where did these types come from in the first place?

A more realistic starting point for our work would be the original CLS machine. It contains no types. Error conditions are implicit: if a machine state does not match the left-hand side of some rule, then an error occurs (unless the machine is in a final state). A more familiar starting point may be a language like Scheme. Scheme has no static typing of expressions but it does have dynamic typechecking on values. For example, given an application $(e_1\, e_2)$ Scheme only requires that the value of $e_1$ be a function (of one argument). There is no check that the type of the value of $e_2$ match the type of the parameter of this function.

Suppose our goal is to start with a language like Scheme but then add static checking to ensure that certain kinds of runtime errors cannot occur. Essentially, how can we get from a language like Scheme (with dynamic typing of values) to one like ML (with static typing of expressions)? We start with the very simple notion of type that Scheme employs. We then identify a notion of subject reduction that preserves this property.

First, let us slightly modify our language. We switch to a syntax using variable names for simplicity. We also add integer constants and associated operations:

$$e ::\ \cdots \mid z \mid s\,e \mid \mathsf{pred}\,e \mid \mathsf{ifz}\,e\,e\,e$$

The term $z$ represents 0, $s$ is the successor constructor, pred is the predecessor operator, and ifz is a conditional that tests for 0. We give a traditional big-step semantics for the entire language in Figure 6. There are three essential cases in these rules in which the form of a value is significant. They occur in the antecedents of the rules for application, predecessor, and the two conditional rules:

– the rule for application requires that $e_1$ evaluate to a function;
– the rule for pred requires that $e$ evaluate to a value built from the $s$ constructor;
– the two conditional rules require that $e_0$ evaluate to either $z$ or a value built from the $s$ constructor.

We can characterize these requirements in the form of simple properties:

$$\begin{aligned}\lambda x.e &: \mathsf{function}\\ s\,v &: \mathsf{successor}\\ z &: \mathsf{zero}\end{aligned}$$

Given that the two conditional rules allow $e_0$ to have either the zero or successor property, we might consider their union as another property. We'll call this property nat.

$$\frac{}{\lambda x.e \hookrightarrow \lambda x.e}$$

$$\frac{e_1 \hookrightarrow \lambda e \quad e_2 \hookrightarrow v_2 \quad \rhd e[v_2/x] \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

$$\frac{}{z \hookrightarrow z} \qquad \frac{e \hookrightarrow v}{s\, e \hookrightarrow s\, v} \qquad \frac{e \hookrightarrow s\, v}{\mathsf{pred}\, e \hookrightarrow v}$$

$$\frac{e_0 \hookrightarrow z \quad e_1 \hookrightarrow v}{\mathsf{ifz}\, e_0\, e_1\, e_2 \hookrightarrow v} \qquad \frac{e_0 \hookrightarrow s\, v \quad e_2 \hookrightarrow v}{\mathsf{ifz}\, e_0\, e_1\, e_2 \hookrightarrow v}$$

**Fig. 6.** Big Step Semantics

These are really the only properties of values that we need to check during the evaluation of expressions. Implicit in the given operational semantics are error conditions that we can describe as situations in which values do not have the required property. For example, if the operator subexpression of an application evaluates to a value other than a function, than an error occurs. Similar statements can be made for predecessor and conditional expressions. Note that we don't require a stronger property for functions, e.g., the usual simple function types specifying the type of argument and return value. This additional information is not required to know that an application can proceed as a function call.

In the example in the previous section we considered state transitions and the notion of preserving safe states via transitions. Now we will consider a notion of subject reduction, hoping that the properties of values above can be extended to expressions and shown to be preserved by the evaluation relation. To assist this, we use an operational semantics based on substitution instead of environments. This semantics uses values that are also expressions (no function closures).

Looking at the rules and following the idea of subject reduction, we can conclude the following:

– the operator $e_1$ in an application should have the function property
– the expression $e_0$ in a conditional should have the nat property
– the argument $e$ in a predecessor expression should have the successor property

The first two conclusions are ones we expect to have as part of a static type system, while the last is not. We typically expect an operation like predecessor to possibly generate runtime errors, even in a statically typed language. How we do we distinguish between these two situations? What additional information do we need to include?

The additional information comes from the requirement that subject reduction apply to all terms. We need to consider every rule and argue that the property of the value (right-hand side of the "evalsto" relation) is true of the expression (left-hand side). In the cases for the axioms, this result holds immediately. Consider the case for application. The property of the value $v$ should also hold of the expression $(e_1\, e_2)$. We can conclude that the property of $e_1$ should be function. The property of $v$ should also be that of the property of $e[v_2/x]$. The property of $v_2$ should be the same as the property of $e_2$. Hence the property of $e$ is determined under the assumption that the variable $x$ has the same property. Summing this up (with plenty of reverse engineering and insight), the essential property of $e_1$ (whose value is $\lambda x.e$) is not only function, but the function has the additional property that its body have a certain property under an assumption about its bound variable. What we have here is the notion of simple function types:

$$\frac{e_1 : \mathsf{function}(Px, P) \quad e_2 : Px}{e_1\, e_2 : P}$$

Now consider the rules for conditionals. If the value of a conditional expression has property $P$, then it's because either $e_1$ or $e_2$ has type $P$. So the property of a conditional expression should be the union of the properties $e_1$ and $e_2$. Furthermore, the property of $e_0$ should be nat (the union of properties zero and successor).

$$\frac{e_0 : \mathsf{nat} \quad e_1 : P_1 \quad e_2 : P_2}{\mathsf{ifz}\, e_0\, e_1\, e_2 : P_1 \cup P_2}$$

(We are being very informal here about the use of union.) The operational rule for successor presents some problems. All we really need is the following static rule:

$$\frac{}{s\, e : \mathsf{successor}}$$

Just looking at the operational rule for successor, we don't need any further condition on the property of $e$.

Finally the rule for predecessor also presents problems:

$$\frac{e : \mathsf{successor}}{\mathsf{pred}\, e : ???}$$

We don't have enough information in either case to determine how to proceed.

These frustrations should really not come as a surprise because the language, as given, does not require any stronger properties regarding successor and predecessor. This situation is related to the fact that some ill-typed programs (in a strongly typed language) can execute without type errors. For example, consider the expression

$$(\mathsf{pred}\, (s\, (\lambda x.x)))\, z$$

This expression would probably not be well typed in any reasonable type system, but evaluates to $z$ without error. To impose a static type system that eliminates all possibility of run-time type errors, we are faced with the problem of changing the language, i.e., eliminating programs that run correctly. How do we do this? We again are faced with the problem identified by Jones of distinguishing between static type errors and run-time errors. Additionally, we are faced with the problem of restricting the language in a sensible way.

With further insight (and reverse engineering) we might come to the right conclusions about restricting the language via a static type system, but it is not clear how to automatically deduce this information from the operational rules. The operational semantics just does not provide explicitly the information that we need to construct properties of expressions. Perhaps this is because, in general, there could be more than one set of properties that could satisfy the operational semantics. This situation is certainly possible, with the result being more than one static semantics for a given language. This seems to be an inherent situation with operational semantics.

Another issue that arises is the kinds of types that might be useful for reasoning about static analyses. Even if we hope to engineer a simple type system, we might find intermediate uses for more sophisticated types, such as union types and intersection types [17]. We have already suggested how union types might come into play, as we find an expression that must satisfy more than one static property. Alternatively, we might find use for intersection types which allow us to express that an expression must have multiple properties. Considering these kinds of types and type systems that manipulate them greatly increases the facilities we have for extracting static properties from semantics.

An open question of research, then, is to consider specifications of operational semantics that provide sufficient information for constructing a static semantics. Associated with this task is the identification of such semantics via a required form or set of properties. In our work on pass separation of abstract machines we gave a definition of abstract machine that supported separation into compile-time and run-time operations. A similar, restrictive definition of operational semantics, ensuring the ability to separate compile-time error checking and run-time error checking, if found, would significantly enhance our ability to automatically construct static semantics from operational semantics.

## 6 Conclusions

The study of type systems and static analyses done by Neil Jones provided a starting point for studying the problem of extracting static semantics. He identified key problems, including definitions of static errors, helping to pave the way for the current and future work. Still, significant obstacles stand in the way of reaching practical solutions for real languages. Our experiments here hopefully illuminated some of them, including the definition of a style of semantics that supports pass separation into static and dynamic error checking, as well as the identification of language restrictions to support such separation.

The simple question that Neil Jones asked years ago continues to hold our interest and asks us to consider new problems.

## References

1. Barbuti, R. and A. Martelli, *A structured approach to static semantics correctness*, Science of Computer Programming **3** (1983), pp. 279–311.
2. Cousineau, G., P.-L. Curien and M. Mauny, *The categorical abstract machine*, The Science of Programming **8** (1987), pp. 173–202.

3. Doh, K.-G. and D. A. Schmidt, *Extraction of strong typing laws from action semantics definitions*, in: *European Symposium on Programming*, Lecture Notes in Computer Science **582** (1992), pp. 151–166.
4. Hannan, J., *Staging transformations for abstract machines*, in: P. Hudak and N. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation* (1991), pp. 130–141.
5. Hannan, J. and D. Miller, *From operational semantics to abstract machines*, Mathematical Structures in Computer Science **2** (1992), pp. 415–459, appears in a special issue devoted to the 1990 ACM Conference on Lisp and Functional Programming.
6. Jones, N., *Static semantics, types, and binding time analysis*, Theoretical Computer Science **90** (1991), pp. 95–118, also appeared in Images of Programming, eds. D. Bjørner and V. Kotov, North-Holland.
7. Jones, N. D., C. K. Gomard and P. Sestoft, "Partial evaluation and automatic program generation," Prentice Hall,, 1993.
8. Jørring, U. and W. Scherlis, *Compilers and staging transformations*, in: *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986, pp. 86–96.
9. Landin, P. J., *The mechanical evaluation of expressions*, Computer Journal **6** (1964), pp. 308–320.
10. Lee, P. and U. F. Pleban, *A realistic compiler generator based on high-level semantics*, in: *Conference Record of the 14th ACM Symposium on the Principles of Programming Languages*, 1987, pp. 284–295.
11. Milner, R., M. Tofte, R. Harper and D. MacQueen, "The Definition of Standard ML (Revised)," MIT Press, 1997.
12. Montenyohl, M. and M. Wand, *Correct flow analysis in continuation semantics*, in: *Conf. Rec. of the 15th ACM Symposium on Principles of Programming Languages*, 1988, pp. 204–218.
13. Mosses, P. D., *Compiler generation using denotational semantics*, in: *Mathematical Foundations of Computer Science*, number 45 in Lecture Notes in Computer Science (1976), pp. 436–441.
14. Mosses, P. D., "Action Semantics," Tracts in Theoretical Computer Science, Cambridge University Press, 1992.
15. Nielson, F. and H. R. Nielson, *Two-level semantics and code generation*, Theoretical Computer Science **56** (1988), pp. 59–133.
16. Palsberg, J., *A provably correct compiler generator*, in: *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science **582** (1992), pp. 418–434.
17. Pierce, B. C., "Types and Programming Languages," MIT Press, 2002.
18. Tofte, M., "Compiler Generators," Springer-Verlag, 1990.

# Foundations of the Bandera Abstraction Tools*

John Hatcliff[1], Matthew B. Dwyer[1], Corina S. Păsăreanu[2], and Robby[1]

[1] Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA.
{hatcliff,dwyer,robby}@cis.ksu.edu
[2] Kestrel Technology, NASA Ames Research Center
Moffet Field, CA, 94035-1000, USA.
pcorina@email.arc.nasa.gov

**Abstract.** Current research is demonstrating that model-checking and other forms of automated finite-state verification can be effective for checking properties of software systems. Due to the exponential costs associated with model-checking, multiple forms of abstraction are often necessary to obtain system models that are tractable for automated checking.

The Bandera Tool Set provides multiple forms of automated support for compiling concurrent Java software systems to models that can be supplied to several different model-checking tools. In this paper, we describe the foundations of Bandera's data abstraction mechanism which is used to reduce the cardinality (and the program's state-space) of data domains in software to be model-checked. From a technical standpoint, the form of data abstraction used in Bandera is simple, and it is based on classical presentations of abstract interpretation. We describe the mechanisms that Bandera provides for declaring abstractions, for attaching abstractions to programs, and for generating abstracted programs and properties. The contributions of this work are the design and implementation of various forms of tool support required for effective application of data abstraction to software components written in a programming language like Java which has a rich set of linguistic features.

## 1 Introduction

Current research is demonstrating that model-checking and other techniques for automated finite-state verification can be applied directly to software in written in widely used programming languages like C and Java [1, 3, 4, 15, 28]. Although

* This work was supported in part by NSF under grants CCR-9703094, CCR-9708184, CCR-9896354, and CCR-9901605, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, by DARPA/IXO's PCES program through AFRL Contract F33615-00-C-3044, by NASA under grant NAG-02-1209, by Intel Corporation under grant 11462, and was performed for the Formal Verification of Integrated Modular Avionics Software Cooperative Agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

they may vary substantially in the specifics, in essence each of these techniques exhaustively checks a finite-state model of a system for violations of a system requirement formally specified by some assertion language or in some temporal logic (e.g., LTL [22]). Finite-state verification is attractive because it automatically and exhaustively checks all behaviors captured in the system model against the given requirement. A weakness of this approach is that it is computationally very expensive (especially for concurrent systems) due to the huge number of system states, and this makes it difficult to scale the approach to realistic systems.

The widespread adoption of Java with its built-in concurrency constructs and emphasis on event-based programming has led to a state of affairs where correct construction of multi-threaded reactive systems, which was previously a domain for experienced programmers, must be tackled by novice programmers. Moreover, Java is being used increasingly in embedded systems where it is more important to detect and remove errors before initial deployment. Thus, there is substantial motivation for building model-checking tools to assess the effectiveness of applying software model-checking to Java. Central to any such tool should be *abstraction mechanisms* that are employed to reduce the number of states encountered during the exploration of software models.

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin [16], dSpin [9], and JPF [3]). Both program slicing and user extensible data abstraction components are applied to form abstract program models customized the to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

Various forms of predicate abstraction [1, 3, 28] and data abstraction [8, 10, 27] have been used in model-checking, and there is a wide body of literature on these techniques. Given this rich resource upon which to build our abstraction facilities in Bandera, our particular choice of abstraction techniques was influenced by multiple requirements outlined below.

**(I)** *The abstraction facilities should be easy to understand and apply by software engineers with little technical background in formal semantics:* This is a basic requirement if Bandera is to be applied effectively by a broad spectrum of users.

**(II)** *The abstraction capabilities should integrate well with the dynamic features found in Java:* In Java programs, features such as dynamic thread/object creation and traversal of heap-allocated data are ubiquitous. Existing work on predicate abstraction and automated refinement [1, 28] has focused on software that relies on integer computation and pointers that are restricted to referencing static data. Automated counter-example-driven abstraction refinement for programs with dynamically allocated data/threads is still an open research area.

**(III)** *The abstraction process should scale to realistic software systems:* Methods for selecting abstractions and methods for constructing abstract programs should not degrade dramatically as the size of programs considered increases.

**(IV)** *The abstraction process should be decoupled from model-checker engines:* Bandera encapsulates existing model-checkers and does not modify their functionality. Thus, any abstraction process implemented by Bandera needs to be accomplished outside of any encapsulated model-checkers.

Bandera addresses Requirement I by providing a easy-to-use and flexible mechanism for defining abstractions. Complex domain structures are avoided by embracing only domains that can be represented as powersets of finite sets of "abstract tokens". These domains (along with appropriate abstract versions of operators) form *abstract types* that users can associate with program variables. An abstract type inference takes as input an abstract type environment that gives abstraction selections for a small set of relevant program variables and then propagates this information through the entire program. This significantly reduces the amount of effort required by users to specify how a system should be abstracted.

Bandera addresses Requirement II by taking advantage of the fact that the type-based data abstraction process described above can be applied in a component-wise manner to fields of different classes. This allows components of heap-allocated data to be abstracted without considering more complicated forms of, for example, shape analysis or predicate abstraction.

Bandera addresses Requirement III by precomputing definitions of abstract operations (using a theorem prover) and then compiling those definitions into the Java source code to form an abstract program. Thus, the repeated (expensive) calls to a theorem prover used in predicate abstraction approaches are not needed during abstract program construction or the model-checking process.

Bandera addresses Requirement IV by transforming the Java source code of program such that concrete operations on types to be abstracted are replaced by corresponding calls to generated abstraction operations. Since the abstraction process takes places at the source level, existing model-checking engines do not have to be modified to incorporate abstraction mechanisms.

In summary, we have arrived at the form of abstraction used in Bandera by balancing a number of competing goals. Bandera's abstraction facilities are less automatic than the automated predicate abstraction and refinement techniques of other tools, but they are much less expensive and can be used immediately in the presence of concurrency and dynamically allocated data without any technical extensions. Moreover, we have found the facilities to be effective in reasoning about a variety of real Java systems.

The tool-oriented aspects of Bandera's abstraction facilities have been described in detail elsewhere [10]. In this paper, we focus on technical aspects of the facilities. Section 2 reviews the methodology that users typically follow when applying Bandera. Section 3 describes the program/property syntax and semantics for a simple flowchart language which we will use to present technical aspects of Bandera's abstraction facilities. Section 4 presents a formal view of Bandera's abstraction definitions and how decision procedures are used to automatically construct definitions of abstract operators and tests. Section 5 describes Bandera's abstract type inference mechanism that is used to bind abstraction declarations to program components. Section 6 outlines how Bandera uses the abstraction bindings calculated above to generate an abstracted program. Section 7 describes how chosen program abstractions should also give rise to appropriate abstractions of the property being checked. Section 8 summarizes related work, and Section 9 concludes.

## 2 The Bandera Abstraction Facilities

Bandera is designed to support a semi-automated abstraction methodology. The goal of this methodology is to minimize the amount of information that user's need to supply to perform an abstract model check of a given property on a Java system. The main steps in applying Bandera are:

1. Identify the portion of the system to be analyzed;
2. Formalize the property in the Bandera Specification Language;
3. Compile the system and property;
4. Define and select the abstractions to be used;
5. Generate an abstracted system model and property;
6. Execute a model check on the abstracted system model; and
7. Analyze any generated counter-examples for feasibility.

Bandera provides various forms of automated support for Step 1. Once the system has been closed, in Step 2 the user formalizes properties to be checked using the Bandera Specification Language (BSL) [5] — a language for defining propositions/predicates on a program's control points and data state and for expressing temporal relationships between declared propositions. In Step 3, Bandera compiles a closed Java unit and the property specification down to a three-address intermediate form called Jimple – part of the Soot [29] Java compiler infrastructure. After transformations in Steps 4 and 5 and other transformations such as slicing have completed, Jimple is transformed to a lower-level intermediate representation called the Bandera Intermediate Representation (BIR). A detailed presentation of BIR's semantics and the translation of Java to BIR and BIR to Promela, the Spin model-checker's input language, is available in [18].

In the remainder of this section, we provide a brief overview of Steps 4–7. We emphasize the toolset components related to the definition of abstractions and the encoding of abstract system models and properties.

### 2.1 Defining and Selecting Abstractions

Users select abstractions by considering the semantics of predicates appearing in the property to be checked and the program expressions that can exert either

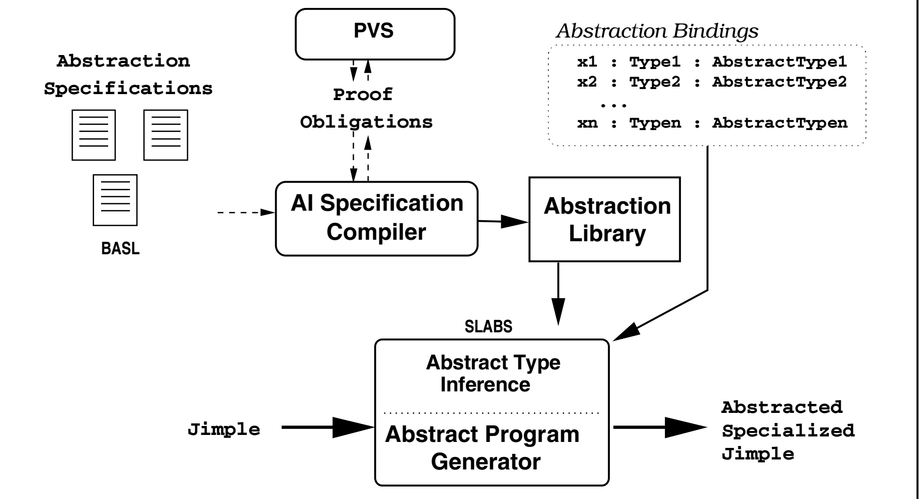

PVS
Abstraction Bindings
x1 : Type1 : AbstractType1
x2 : Type2 : AbstractType2
...
xn : Typen : AbstractTypen
Abstraction Specifications
Proof Obligations
AI Specification Compiler
Abstraction Library
BASL
SLABS
Abstract Type Inference
Abstract Program Generator
Jimple
Abstracted Specialized Jimple

**Fig. 1.** Architecture of Bandera's Abstraction Facilities

control or date influences on those predicates. In [10] we describe tool support in Bandera that allows users to query and explore the program dependence graph that is generated by using a property's predicates to derive a slicing criterion. While strategies for exploiting this tool support to identify the program variables that should be abstracted and the semantics that should be preserved by such an abstraction are relevant for users of Bandera, in this paper, we assume that such a determination has been made. The user carries out the balance of the abstraction process by interacting with the Source Level Abstraction (SLABS) Bandera tool components displayed in Figure 1.

Bandera includes an *abstraction library* containing definitions of common abstractions for Java base types from which users can select. If necessary, the user specifies new abstractions using the rule-based Bandera Abstraction Specification Language (BASL). For Java base types, the user need only define an abstract domain, and abstract versions of concrete operations are generated automatically using the decision procedures of the PVS theorem prover [24]. The abstraction definitions are then compiled to a Java representation and added to the library.

The user declares how program components should be abstracted by binding class fields to entries from the abstraction library. It is usually only necessary to attach abstraction to a relatively few variables since the toolset contains an abstract type-inference phase that automatically propagates the abstract type information to remaining program components. When abstract type inference is complete, the concrete Jimple program representation is transformed to abstract Jimple by replacing concrete constants and operations by abstract tokens and operations drawn from the compiled abstraction representations in the abstraction library. We describe these steps in more detail in the subsections below.

```
abstraction EvenOdd abstracts int
  begin
   TOKENS = {EVEN, ODD};

   abstract(n)              operator * mul           operator > gt
    begin                    begin                    begin
      n % 2 == 0 -> {EVEN};   (ODD, ODD) -> {ODD} ;     (_, _) -> {true, false}
      n % 2 == 1 -> {ODD};    (_, _) -> {EVEN} ;       end
   end                      end
  end
```

**Fig. 2.** BASL definition of EvenOdd AI (excerpts)

**Defining Abstractions** Bandera provides BASL – a simple declarative specification language that allows users to define the three components of an AI described above.

Figure 2 illustrates the format of BASL for abstracting base types by showing excerpts of the even-odd AI specification. The specification begins with a definition of a set of tokens — the actual abstract domain will be the powerset of the token set. Although one can imagine allowing users to define arbitrary lattices for abstract domains, BASL currently does not provide this capability because we have found powersets of finite token sets to be easy for users to understand and quite effective for verification. Following the token set definition, the user specifies the abstraction function which maps concrete values (in this case, integers) to elements of the abstract domain. After the abstraction function, the BASL specification for base types must contain a definition of an abstract operator for each corresponding basic concrete operator.

Abstract operator definitions can be *generated automatically* from the BASL token set and abstraction function definitions for integer abstractions using the elimination method based on weakest pre-conditions from [2]. Using this approach makes it extremely easy for even novice users to construct new AIs for integers. Given a binary concrete operator *op*, generation of the abstract operator $op_{abs}$ applied to a particular pair of abstract tokens $a_1$ and $a_2$ proceeds as follows. The tool starts with the most general definition (i.e., it assumes that $op_{abs}(a_1, a_2)$ can output any of the abstract tokens – which trivially satisfies the safety requirement). Then, for each token in the output, it checks to see (using the theorem prover PVS [24]) if the safety property would still hold if the token is eliminated from the output. An abstract token can be safely eliminated from the output token set if the result of the concrete operation applied to concrete values cannot be abstracted to that abstract value.

BASL also includes formats for specifying AIs for classes and arrays. Class abstractions are defined component-wise: the BASL format allows the user to assign AIs to each field of the class. BASL's array format allow specification of an integer abstraction for the array index and an abstraction for the component type [10].

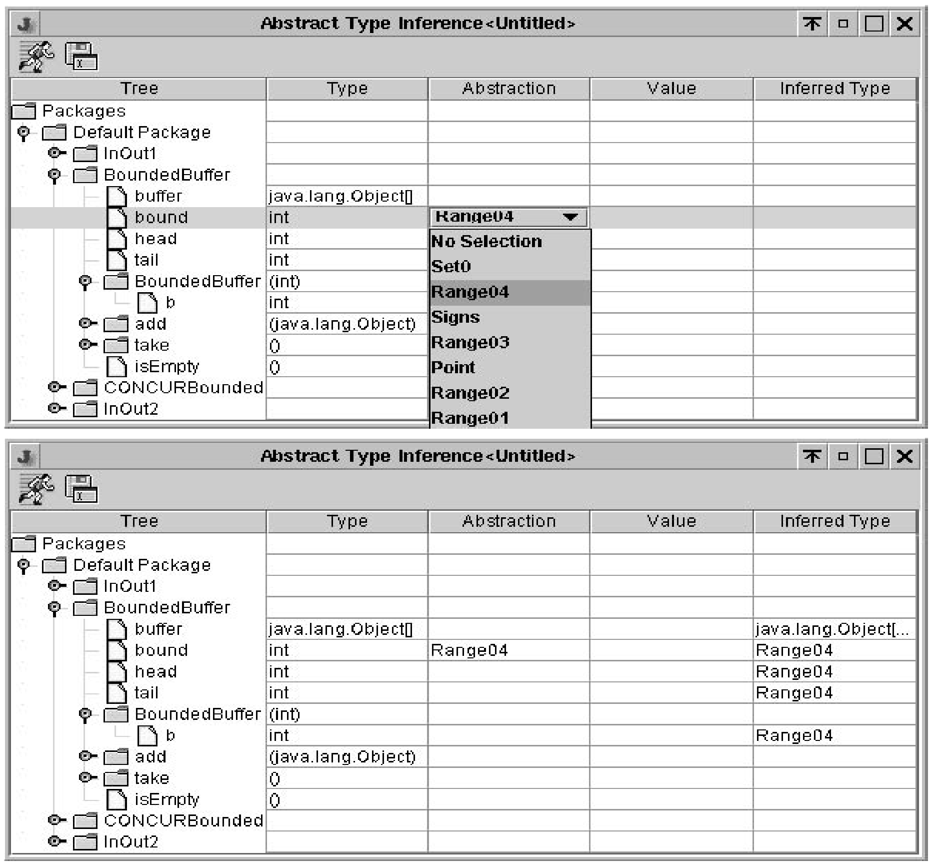
**A Library of Abstractions** Since they are so widely applied, abstractions for integers are organized into several different families including the *concrete* (or *identity*), *range*, *set*, *modulo* and *point* families which we discuss below.

A *concrete* AI (also known as an *identity* AI) performs no abstraction at all, but rather preserves all concrete values and uses the original concrete operations on these. A *range* AI tracks concrete values between lower and upper bounds $l$ and $u$ but abstracts values less than $l$ and greater than $u$ by using a token set of the form $\{belowl, l, ..., u, aboveu\}$; an abstraction that preserves the *sign* of values is a range-$(0, 0)$ abstraction. A *set* AI can often be used instead of a range AI when no operations other than equality are performed (e.g., when integers are used to simulate an enumerated type). For example, a set AI that tracks the concrete values 3 and 5 would have the token set $\{three, five, other\}$. A *modulo-k* AI merges all integers that have the same remainder when divided by $k$. The EvenOdd abstraction with token set $\{EVEN, ODD\}$ is a modulo-2 abstraction. Finally, the token set for the *point* AI includes a single token *unknown*. The point abstraction function maps all concrete values to this single value; this has the effect of throwing away all information about the data domain.

**Defining Field Abstractions** Bandera includes tool support to ease the process of binding class fields to abstractions. Abstractions are indexed by type, thus when the user considers a field, such as `BoundedBuffer.bound`, the type, `int`, can be used to present the candidate abstractions from the library as illustrated in Figure 3. The user selects from these abstractions and repeats the process for other variables that have been determined to require abstraction. Once all such bindings have been made the tools calculate abstractions for all other fields in the program. The resulting inferred abstract types are displayed for the user to view as illustrated at the bottom of Figure 3. Conflicts in the inferred type for a given field are presented to the user for resolution. Fields which are unconstrained by the type inference can be set to a default type which is usually either the concrete type or the *point* abstraction.

## 2.2 Generating an Abstracted System Model and Property

Generating an abstract program involves three separate steps. First, given a selection of AIs for a program's data components, the BASL specification for each selected AI is retrieved from the abstraction library and compiled into a Java class that implements the AI's abstraction function and abstract operations. Second, the given concrete Java program is traversed, and concrete literals and operations are replaced with calls to classes from the first step that implement the corresponding abstract literals and operations. The resulting abstract program yields an *over-approximation* of the concrete program's behavior. An over-approximation ensures that every behavior in the concrete program that violates a given property is also present in the abstract program. To ensure the soundness of verification results, the third step abstracts the property to be checked so as to *under-approximate* the set of behaviors described by the original property. An under-approximation ensures that any program behavior that


Abstract Type Inference<Untitled>
No Selection
Set0
Range04
Signs
Range03
Point
Range02
Range01
Abstract Type Inference<Untitled>

| Tree | Type | Abstraction | Value | Inferred Type |
|---|---|---|---|---|
| Packages | | | | |
| Default Package | | | | |
| InOut1 | | | | |
| BoundedBuffer | | | | |
| buffer | java.lang.Object[] | | | |
| bound | int | Range04 | | |
| head | int | | | |
| tail | int | | | |
| BoundedBuffer | (int) | | | |
| b | int | | | |
| add | (java.lang.Object) | | | |
| take | () | | | |
| isEmpty | () | | | |
| CONCURBounded | | | | |
| InOut2 | | | | |

| Tree | Type | Abstraction | Value | Inferred Type |
|---|---|---|---|---|
| Packages | | | | |
| Default Package | | | | |
| InOut1 | | | | |
| BoundedBuffer | | | | |
| buffer | java.lang.Object[] | | | java.lang.Object[... |
| bound | int | Range04 | | Range04 |
| head | int | | | Range04 |
| tail | int | | | Range04 |
| BoundedBuffer | (int) | | | |
| b | int | | | Range04 |
| add | (java.lang.Object) | | | |
| take | () | | | |
| isEmpty | () | | | |
| CONCURBounded | | | | |
| InOut2 | | | | |

**Fig. 3.** Abstraction selection and abstract type inference

is contained in the set of behaviors described by the abstract property will also be contained in the set of behaviors described by the original property (these issues are discussed in greater detail below and in Section 7).

**Compiling Abstractions to Java** Figure 4 shows excerpts of the Java representation of the BASL even-odd specification in Figure 2. Abstract tokens are implemented as integer values (constructed by shifting `1` into the position indicated by the bit mask declarations), and the abstraction function and operations have straightforward implementations as Java methods. The most noteworthy aspect of the implementation is the modeling of the approximation that arises due to abstraction. The approximate nature of the even-odd abstraction means that a "greater than" comparison of any pair of abstract values could be `true` or `false`. Instead of representing such sets directly (e.g., as a bit vector), a single value is chosen non-deterministically from the set of possible values. This is valid when the meaning of a particular program is taken to be the collection of all

```
public class EvenOdd {
  public static final int EVEN  = 0; // bit mask
  public static final int ODD = 1; // bit mask

  public static int abs(int n) {
    if (n % 2 == 0) return (1 << EVEN);
    if (n % 2 == 1 || n % 2 == -1) return (1 << ODD);
    throw new RuntimeException();
  }

  public static int mul(int arg1, int arg2) {
    if (arg1==(1 << ODD)  && arg2==(1 << ODD))  return (1 << ODD);
    return (1 << EVEN);
  }

  public static boolean gt(int arg1, int arg2) {
    return Bandera.choose();
  }
}
```

**Fig. 4.** Compilation of BASL EvenOdd AI (excerpts)

possible traces or executions of the program. In Figure 4, the `Bandera.choose()` method denotes a non-deterministic choice between `true` and `false` values. This method has no Jimple implementation; instead, when Bandera compiles the abstracted program down to the input of given a model-checker, the method is implemented in terms of the model-checker's built-in constructs for expressing non-deterministic choice. Since the model-checker will generate all paths leading out of a non-deterministic choice, this ensures that all appropriate behaviors are represented in the model. This approach has some interesting implications compared to more tradition presentations of abstract interpretation. Using non-determinism to model imprecision in this manner (in essence, by transforming data imprecision into extra control paths), means that the abstract interpretation is maximally polyvariant, and there is never any merging of information using, for example, least-upper-bound operators. This approach can be effective since abstract domains in Bandera are finite, of finite height, and typically quite small. An alternative approach would be to use a set of abstract tokens to represent imprecision and to represent the set as a bit-vector. However, splitting sets into single tokens using non-determinism as described above yields a much simpler implementation.

**Replacing Concrete Operators** Traversing a given concrete program and replacing each operation with a call to a corresponding abstract version is relatively straightforward. The only challenge lies in resolving *which* abstract version of an operation should be used when multiple AI's are selected for a program. This problem is solved by the abstract type inference phase outlined in the previous section: in addition to propagating abstract type information to each of the program variables, type inference also attaches abstract type information to each node in the program's syntax tree. For example, consider the code fragment `(x + y) + 2` where the user selected variable `x` to have type even-odd and `y` was not selected for abstraction. This code fragment will be transformed into:

```
EvenOdd.add(EvenOdd.add(x, Coerce.IntToEvenOdd(y)),
            EvenOdd.Even);
```

For the innermost concrete `+` operation, the user selection of even-odd for `x` forces the abstract version of `+` to be `EvenOdd.add`. Assuming no other contexts force `y` to be abstracted, `y` will hold a concrete value, and thus a coercion (`Coerce.IntToEvenOdd`) is inserted that converts `y`'s concrete value to an even-odd abstract value. For the outermost `+`, since the left argument has an abstract type of even-odd, the constant 2 in the right argument is "coerced" at translation time to an even-odd abstract constant.

**Property Abstraction** Bandera's program abstraction approach yields a model in which execution states safely over-approximate the values of program variables. For example, a concrete state where variable `x` has the value 2 may be approximated by a modulo-2 abstracted value of *even*. When abstracting properties, this can be problematic if the abstractions do not preserve the ability to exactly decide the predicates in the property. Consider a predicate `x==4` evaluated in the example state described above. This predicate would appear to be true in the abstract state, since 4 is clearly abstracted by *even*, but the predicate evaluates to false in the concrete state.

Bandera abstracts the predicates appearing in the property being checked using an approach that is similar to [20]. Consider an AI for a variable `x` (*e.g.*, signs) that appears in a predicate (*e.g.*, `(x<1)`). Bandera converts this to a disjunction of predicates of the form `x==`*a*, where *a* are the abstract values that correspond to values that imply the truth of the original predicate (*e.g.*, `x==`*neg* implies `x<1` as does `x==`*zero*, but `x==`*pos* does not). Thus, this abstract disjunction, `x==`*zero* `&& x==`*neg*, under-approximates the concrete predicate insuring that the property holds on the original program if it does on the abstract program.

## 2.3 Abstract Model Checking

The resulting abstracted program and property are converted to BIR from their Jimple form and then to the input language of the selected model checker. Bandera runs the model checker and displays the results to the user. Counter-examples are mapped back to the unabstracted source program.

In addition to supporting exhaustive and sound verification of properties, Bandera provides a number of useful bounded state-space exploration strategies. Bounds can be placed on resources such as the size of integers and arrays, on the number of threads allocated, and on the number of object instances allocated at particular allocator sites. Bandera can construct models for existing model checkers, such as Spin, that perform *resource-bounded searches* [18] that can often yield effective bug-finding without performing any abstraction. These searches can be thought of as depth-bounded searches where the depth of the search is controlled by the bounds placed on different resources. When the bound is exceeded for a particular resource along a particular execution path, the model-checker simply aborts the search along that path and continues searching other unexplored paths.

### 2.4 Counter-Example Feasibility

Model checking an abstracted program may produce a counter-example that is infeasible with respect to the program's concrete semantics. Since counter-examples may be very long and complex, user's require tool support to assist in the determination of feasibility. Bandera includes both an on-line technique for biasing the state-space search to find guaranteed feasible counter-examples and an off-line for simulating a counter-example on the abstract and concrete programs and checking their correspondence. The former, while unsound, has the advantage of being fast and surprisingly effective. A detailed presentation of these techniques is given in [27].

## 3 Program Syntax and Semantics

We noted in the previous section that Bandera translates Java programs into the Jimple intermediate form. To capture the essence of the Jimple structure for our formal overview of Bandera abstraction, we use the simple flowchart language FCL of Gomard and Jones [13, 14, 19].

Since our abstraction framework involves presenting abstraction definitions as types, we present a formalization of the framework using multi-sorted algebras. This allows new abstractions to be introduced as new sorts/types.

### 3.1 Signatures and Algebras

A signature $\Sigma$ is a structure containing a set $\mathsf{Types}[\Sigma]$ of types (which must include the distinguished type $\mathsf{Bool}$), a non-reflexive subtyping relation $\ll_\Sigma$ between the types of $\mathsf{Types}[\Sigma]$ that forms an upper semi-lattice and for each $(\tau_1, \tau_2) \in \ll_\Sigma$ a coercion symbol $[\tau_1 \ll_\Sigma \tau_2]$, a set $\mathsf{Ops}[\Sigma]$ of typed operation (*e.g.*, +), a set $\mathsf{Tests}[\Sigma]$ of typed test symbols (*e.g.*, =, >), and a set $\mathsf{Cons}[\Sigma]$ of typed constant symbols (*e.g.*, 2, 3). For notational simplicity, we will only consider binary operations and tests. The type of an operation $o \in \mathsf{Ops}[\Sigma]$ is denoted $[o]_\Sigma = \tau_1 \times \tau_2 \rightarrow \tau$ (similarly for tests and constants). For simplicity, for operation types $[o]_\Sigma = \tau_1 \times \tau_2 \rightarrow \tau$ we assume $\tau_1 = \tau_2 = \tau$ and for test types $[o]_\Sigma = \tau_1 \times \tau_2 \rightarrow \mathsf{Bool}$ we assume $\tau_1 = \tau_2$. This corresponds to the type structure of most of the built in base type operations in Java.

A $\Sigma$-algebra is a structure containing for each $\tau \in \mathsf{Types}[\Sigma]$ a semantic domain $[\![\tau]\!]^A_\Sigma$, for each pair $(\tau_1, \tau_2) \in \ll_\Sigma$ a total coercion relation $[\![[\tau_1 \ll_\Sigma \tau_2]]\!]^A_\Sigma \subseteq [\![\tau_1]\!]^A_\Sigma \times [\![\tau_2]\!]^A_\Sigma$, for each operation symbol $o \in \mathsf{Ops}[\Sigma]$ with type $\tau \times \tau \rightarrow \tau$ a relation $[\![o]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma$, for each test symbol $t \in \mathsf{Tests}[\Sigma]$ with type $\tau \times \tau \rightarrow \mathsf{Bool}$ a total relation $[\![t]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma \times [\![\mathsf{Bool}]\!]$ where $[\![\mathsf{Bool}]\!] = \{\mathit{true}, \mathit{false}\}$, and for each constant symbol $c \in \mathsf{Cons}[\Sigma]$ with type $\tau$ a set $[\![c]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma$ (we will drop the super/sub-scripts $\Sigma$ and $A$ when these are clear from the context). Using relations instead of functions to model the semantics of operations and tests (and sets instead of a single value for constants) provides a convenient way to capture the imprecision of abstractions.

In Bandera, the abstraction process begins by considering the concrete semantics of a program which we will model using a *basis configuration* – a basis signature $\Sigma_{basis}$ with

$$\begin{aligned}
\mathsf{Types}[\Sigma_{basis}] &= \{\mathsf{Int}, \mathsf{Bool}\},\\
\ll_{\Sigma_{basis}} &= \emptyset,\\
\mathsf{Ops}[\Sigma_{basis}] &= \{+, -, *, \ldots\},\\
\mathsf{Tests}[\Sigma_{basis}] &= \{>, =, \&\&, ||, \ldots\},\\
\mathsf{Cons}[\Sigma_{basis}] &= \{\ldots, -1, 0, 1, \ldots, \mathsf{true}, \mathsf{false}\}
\end{aligned}$$

and a basis algebra $A_{basis}$ with the usual carrier sets for $[\![\mathsf{Int}]\!]$ and $[\![\mathsf{Bool}]\!]$, the usual functional interpretation for operations and tests $[\![+]\!]$, $[\![-]\!]$, $[\![>]\!]$, $[\![=]\!]$, *etc.*, and singleton set interpretations for integer and boolean constants, *e.g.*., $[\![1]\!] = \{1\}$, $[\![\mathsf{true}]\!] = \{true\}$. The subtyping relation is empty in the basis signature, because we use subtyping to express refinement relationships between abstractions, and no abstractions appear in as yet unabstracted concrete programs.

## 3.2 Program and Property Syntax

**Program Syntax** Figure 5 presents the definition of FCL syntax. An FCL program consists of a list of parameters $x^*$, a label $l$ of the initial block to be executed, and a non-empty list $b^+$ of *basic blocks*. Each basic block consists of a *label* followed by a (possibly empty) list of *assignments*. Each block concludes with a *jump* that transfers control from that block to another one. Even though the syntax specifies prefix notation for operators, we will often use infix notation in examples with deeply nested expressions to improve readability. As noted earlier, the basis signature contains an empty subtyping relation, so coercion expressions $[\tau \ll_{\Sigma} \tau']\ e$ will not appear in concrete programs to be abstracted.

Figure 6 presents an FCL program that computes the power function. The input parameters to the program are `m` and `n`, and the initial block is specified by the line `(init)`. The parameters can be referenced and assigned to throughout the program. Other variables such as `result` can be introduced without explicit declaration. The initial value of a non-parameter variable is 0. The output of program execution is the state of memory when the **return** construct is executed.

In the presentation of FCL semantics, we need to reason about nodes in a *statement-level control-flow graph* (CFG),*i.e.*, a graph where there is a separate node $n$ for each assignment and jump in a given program $p$. We will assume that each statement (CFG node) has an identifier that is unique within the program, and we will annotate each statement in the source code with its unique identifier. For example, the first assignment in the `loop` block has the unique identifier (or node number) [`loop`.1].

To access code at particular program points within a given FCL-program, $p$, we use the functions $code[p]$, $first[p]$, $succ[p]$, defined below. We will drop the $[p]$ argument from the functions when it is clear from the context.

*Syntax Domains*

| | |
|---|---|
| $p \in \text{Programs}[\Sigma]$ | $x \in \text{Variables}[\Sigma]$ |
| $b \in \text{Blocks}[\Sigma]$ | $T \in \text{Type-Identifiers}[\Sigma]$ |
| $l \in \text{Block-Labels}[\Sigma]$ | $e \in \text{Expressions}[\Sigma]$ |
| $a \in \text{Assignments}[\Sigma]$ | $j \in \text{Jumps}[\Sigma]$ |
| $al \in \text{Assignment-Lists}[\Sigma]$ | $o \in \mathsf{Ops}[\Sigma]$ |
| | $t \in \mathsf{Tests}[\Sigma]$ |
| | $c \in \mathsf{Cons}[\Sigma]$ |

*Grammar*

$$\begin{array}{ll} p ::= (x^*)\ (l)\ b^+ & a ::= x \mathbin{\mathbf{:=}} e;\ \mid\ \mathbf{skip}; \\ b ::= l :\ al\ j & e ::= c \mid x \mid T\ e \mid o(e_1, e_2) \mid t(e_1, e_2) \\ al ::= a^* & j ::= \mathbf{goto}\ l; \mid \mathbf{return}; \mid \\ & \quad\ \mathbf{if}\ e\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2; \end{array}$$

*Typing Rules (expressions)*

$$\frac{}{\Gamma \vdash_\Sigma c : [c]_\Sigma} \qquad \frac{}{\Gamma \vdash_\Sigma x : \Gamma(x)}$$

$$\frac{\Gamma \vdash_\Sigma e_i : \tau \qquad [o]_\Sigma = \tau \times \tau \rightarrow \tau}{\Gamma \vdash_\Sigma o(e, e) : \tau}$$

$$\frac{\Gamma \vdash_\Sigma e_i : \tau \qquad [t]_\Sigma = \tau \times \tau \rightarrow \mathsf{Bool}}{\Gamma \vdash_\Sigma t(e_1, e_2) : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad (\tau, \tau') \in \ll_\Sigma}{\Gamma \vdash_\Sigma [\tau \ll_\Sigma \tau']\ e : \tau'}$$

**Fig. 5.** Syntax of the Flowchart Language FCL for a signature $\Sigma$

– The code map function *code*[$p$] maps a CFG node $n$ to the code for the statement that it labels. Taking the program of Figure 6 as an example, *code*([`loop`.2]) yields the assignment `n := -(n 1);`.
– The function *first*[$p$] maps a label $l$ of $p$ to the first CFG node occurring in the block labeled $l$. For example, *first*(`loop`) = [`loop`.1].
– The function *succ*[$p$] maps each node to the set of nodes which are immediate successors of that node. For examples, *succ*([`test`.1]) = {[`loop`.1], [`end`.1]}.

**Property Syntax** LTL [22] is a rich formalism for specifying state and action sequencing properties of systems. An LTL specification describes the intended behavior of a system on all possible executions.

The syntax of LTL in Figure 7 includes primitive propositions $P$ with the usual propositional connectives, and three temporal operators. Bandera distinguishes logical connectives (e.g., $\wedge$, $\vee$) in the specification logic from logical program operations, and it automatically transforms property specifications to

```
(m n)
(init)

  init: result := 1;                       [init.1]
        goto test;                         [init.2]

  test: if <(n, 1) then end else loop;     [test.1]

  loop: result := *(result, m);            [loop.1]
        n := -(n, 1);                      [loop.2]
        goto test;                         [loop.3]
  end:  return;                            [end.1]
```

**Fig. 6.** Power FCL program

*Syntax Domains*

$$\psi \in \text{Formulas}[\Sigma] \qquad e \in \text{Expressions}[\Sigma]$$
$$P \in \text{Propositions}[\Sigma]$$

*Grammar*

$$\begin{aligned} \psi ::= {} & P \mid \neg P \mid \\ & \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \\ & \Box\psi \mid \Diamond\psi \mid \psi_1 \;\mathcal{U}\; \psi_2 \end{aligned} \qquad P ::= [n] \mid e$$

**Fig. 7.** Syntax of the FCL property language for a signature $\Sigma$

Negation Normal Form (NNF) [17] to simplify the property abstraction process. Accordingly, the syntax of Figure 7 only generates formulas in NNF, and we will assume but not encode explicitly the fact that expressions $e$ in propositions do not contain logical operators.

When specifying properties of software systems, one typically uses LTL formulas to reason about execution of particular program points (*e.g.*, entering or exiting a procedure) as well as values of particular program variables. To capture the essence of this for FCL, we use the following primitive propositions.

- Intuitively, $[n]$ holds when execution reaches the statement with unique identifier $n$ (*i.e.*, the statement at node $n$ will be executed next). We call propositions of this form *node propositions*.
- Intuitively, $e$ holds when the evaluation of $e$ at the current node is not false. We call propositions of this form *variable propositions*.

For example, a program requirement that says if `m` is odd initially, then when the program terminates `result` is odd, can be written as

$$\Diamond[end] \Rightarrow \Box([init] \wedge =(\%(\texttt{m},2),1) \Rightarrow \Diamond([end] \wedge =(\%(\texttt{result},2),1))).$$

Converting the property to NNF yields

$$\Box\neg[\mathit{end}] \lor \Box(\neg[\mathit{init}] \lor \neg{=}(\%(\mathtt{m},2),1) \lor \Diamond([\mathit{end}] \land {=}(\%(\mathtt{result},2),1))).$$

This particular property is an instance of a global response property [11] under the assumption[26] that the program eventually terminates (i.e., reach `end`). We need the assumption to work with since our abstractions cannot preserve liveness property. That is, an abstracted program may violate some liveness properties even though the the original program does not. This is due to the imprecision introduced in our abstraction process, for example, when an abstracted loop condition cannot be decided this gives rise to infinite traces in the program that may violate the liveness property.

Given an LTL formula $\psi$ where $\mathcal{P}$ is the of set of primitive propositions appearing in $\psi$, we will write Nodes[$\mathcal{P}$] for the set of CFG nodes mentioned in node propositions in $\mathcal{P}$, and Vars[$\mathcal{P}$] for the set of variables mentioned in variable propositions in $\mathcal{P}$.

## 3.3 Program and Property Semantics

**Program Semantics** Figure 8 presents the semantics of FCL programs. The interpretation of a $\Sigma$-program $p$ is parameterized on a $\Sigma$-algebra $A$. Given a $\Sigma$, $A$, and a type-assignment $\Gamma$ mapping $\Sigma$-program variables to $\Sigma$-types, a store $\sigma$ is $(\Sigma, A, \Gamma)$-compatible when $\mathit{domain}(\sigma) = \mathit{domain}(\Gamma)$ and for all $x \in \mathit{domain}(\Gamma)$ . $\sigma(x) \in [\![\Gamma(x)]\!]$. The set of $(\Sigma, A, \Gamma)$-compatible stores is denoted $[\![\Gamma]\!]^A_\Sigma$. The semantics of a program is expressed via transitions on program states $(n, \sigma)$ where $n$ is a CFG node identifier from $p$ and $\sigma$ is a $(\Sigma, A, \Gamma)$-compatible store. A series of transitions gives an *execution trace* through $p$'s statement-level control flow graph. It is important to note that when execution is in state $(n_i, \sigma_i)$, the code at node $n_i$ has not yet been executed. Intuitively, the code at $n_i$ is executed on the transition from $(n_i, \sigma_i)$ to successor state $(n_{i+1}, \sigma_{i+1})$.

Figure 8 gives a simple operational semantics that formalizes the transition relation on states. In contrast to the small-step formalization of transition relation, a big-step semantics is used to formalize expression evaluation since we consider expression evaluation to be atomic. The top of the figure gives the definition of expression, assignment, and jump evaluation. The intuition behind the rules for these constructs is as follows.

- $\sigma \vdash_{expr} e \Rightarrow v$ means that under store $\sigma$, expression $e$ evaluates to value $v$.
- $\sigma \vdash_{assign} a \Rightarrow \sigma'$ means that under store $\sigma$, the assignment $a$ yields the updated store $\sigma'$.
- $\sigma \vdash_{jump} j \Rightarrow l$ means that under the store $\sigma$, jump $j$ will cause a transition to the block labeled $l$.

The three transition rules describe small-step transitions caused by assignment evaluation, jump evaluation leading to a new block, and jump evaluation leading to termination. We assume that the set of node labels Nodes[FCL] used in the semantics contains a distinguished node halt which we use to indicate a terminal state.

*Expressions and Assignments*

$$\frac{v \in [\![c]\!]^A_\Sigma}{\sigma \vdash_{expr}\ c\ \Rightarrow\ v} \qquad \frac{}{\sigma \vdash_{expr}\ x\ \Rightarrow\ \sigma(x)} \qquad \frac{\sigma \vdash_{expr}\ e\ \Rightarrow\ v \qquad (v, v') \in [\![ [\tau \ll_\Sigma \tau'] ]\!]^A_\Sigma}{\sigma \vdash_{expr}\ [\tau \ll_\Sigma \tau']\ e\ \Rightarrow\ v'}$$

$$\frac{\sigma \vdash_{expr}\ e_i\ \Rightarrow\ v_i \qquad [\![o]\!]^A_\Sigma(v_1, v_2, v)}{\sigma \vdash_{expr}\ o(e_1, e_2)\ \Rightarrow\ v} \qquad \frac{\sigma \vdash_{expr}\ e_i\ \Rightarrow\ v_i \qquad [\![t]\!]^A_\Sigma(v_1, v_2, v)}{\sigma \vdash_{expr}\ t(e_1, e_2)\ \Rightarrow\ v}$$

$$\frac{\sigma \vdash_{expr}\ e\ \Rightarrow\ v}{\sigma \vdash_{assign}\ x := e\ \Rightarrow\ \sigma[x \mapsto v]} \qquad \frac{}{\sigma \vdash_{assign}\ \textbf{skip}\ \Rightarrow\ \sigma}$$

*Jumps*

$$\frac{}{\sigma \vdash_{jump}\ \textbf{goto}\ l\ \Rightarrow\ l} \qquad \frac{}{\sigma \vdash_{jump}\ \textbf{return}\ \Rightarrow\ \mathsf{halt}}$$

$$\frac{\sigma \vdash_{expr}\ e\ \Rightarrow\ true}{\sigma \vdash_{jump}\ \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2\ \Rightarrow\ l_1}$$

$$\frac{\sigma \vdash_{expr}\ e\ \Rightarrow\ false}{\sigma \vdash_{jump}\ \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2\ \Rightarrow\ l_2}$$

*Transitions*

$$\frac{\sigma \vdash_{assign}\ a\ \Rightarrow\ \sigma'}{(n, \sigma)\ \longmapsto\ (n', \sigma')} \qquad \begin{array}{l}\text{if} \quad code(n) = a\\ \text{where } n' = succ(n)\end{array}$$

$$\frac{\sigma \vdash_{jump}\ j\ \Rightarrow\ l}{(n, \sigma)\ \longmapsto\ (n', \sigma)} \qquad \begin{array}{l}\text{if} \quad code(n) = j\\ \text{where } n' = first(l)\end{array}$$

$$\frac{\sigma \vdash_{jump}\ j\ \Rightarrow\ \mathsf{halt}}{(n, \sigma)\ \longmapsto\ (\mathsf{halt}, \sigma)} \qquad \text{if } code(n) = j$$

*Semantic Values*

$$\begin{array}{l} n \in \text{Nodes[FCL]} \\ s \in \text{States[FCL]} = \text{Nodes[FCL]} \times \text{Stores[FCL]} \end{array}$$

**Fig. 8.** Operational semantics of a $\Sigma$-FCL program with respect to $\Sigma$-algebra $A$

**Property Semantics** The semantics of a primitive proposition is defined with respect to states.

$$[\![ [m] ]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{true\} & \text{if } m = n \\ \{false\} & \text{otherwise} \end{cases}$$

$$[\![ \neg[m] ]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{true\} & \text{if } m \neq n \\ \{false\} & \text{otherwise} \end{cases}$$

$$[\![e]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{true\} & \text{if } \sigma \vdash_{expr}\ e\ \not\Rightarrow false \\ \{false\} & \text{otherwise} \end{cases}$$

$$[\![\neg e]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{true\} & \text{if } \sigma \vdash_{expr}\ e\ \not\Rightarrow true \\ \{false\} & \text{otherwise} \end{cases}$$

Note that the semantics of expression propositions defines an under-approximation, i.e., the proposition expression is not considered true if the expression evaluates to $\{true, false\}$.

The semantics of an LTL formula is defined with respect to traces, where each trace is a (possibly infinite) non-empty sequence of states written $\Pi = s_1, s_2, \ldots$. We write $\Pi^i$ for the suffix starting at $s_i$, *i.e.*, $\Pi^i = s_i, s_{i+1}, \ldots$. Thus, an execution trace of $p$ is a state sequence $\Pi = s_1, s_2, \ldots$ with the following constraints: $s_1$ is an initial state for $p$ and $s_i \longmapsto s_{i+1}$.

The temporal operator $\Box$ requires that its argument be true from the current state onward, the $\Diamond$ operator requires that its argument become true at some point in the future, and the $\mathcal{U}$ operator requires that its first argument is true up to the point where the second argument becomes true. We refer the reader to, e.g., [17], for a formal definition of the semantics of LTL.

## 4 Defining Abstractions

Section 2.1 noted that each Bandera abstraction is associated with a concrete type $\tau$ and that each abstraction definition has four components: an abstraction name, an abstract domain, an abstraction function/relation, and abstract versions of each concrete operation and test. Accordingly, if $\tau$ is a type from $\Sigma$, an $[\Sigma, A]$-compatible *$\tau$-abstraction* $\alpha$ is a structure containing an abstraction type identifier $\tau_\alpha$, a finite abstraction domain $[\![\tau_\alpha]\!]$, an abstraction relation $\rightsquigarrow_\alpha \subseteq [\![\tau]\!]^A_\Sigma \times [\![\tau_\alpha]\!]$, for each $\tau$ operation symbol $o$ in $\Sigma$ an operation $[\![o_\alpha]\!] \subseteq [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!]$, and for each $\tau$ test symbol $t$ in $\Sigma$ a test $[\![o_\alpha]\!] \subseteq [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!] \times [\![\mathsf{Bool}]\!]$.

To ensure that properties that hold true for the abstracted system also hold true in the original concrete system, one needs the standard notion of safety (denoted $\lhd$) as a simulation between operation/test relations.

**Definition 1.** *(Safe abstract operations and abstract tests)*
*Let $\tau_\alpha$ be a $\tau$-abstraction.*

- $[\![o]\!] \lhd [\![o_\alpha]\!]$ *iff for every* $c_1, c_2, c \in [\![\tau]\!]$, *and* $a_1, a_2 \in [\![\tau_\alpha]\!]$, *if* $c_1 \rightsquigarrow_\alpha a_1$, $c_2 \rightsquigarrow_\alpha a_2$, *and* $[\![o]\!](c_1, c_2, c)$ *then there exists* $a \in [\![\tau_\alpha]\!]$ *such that* $[\![o_\alpha]\!](a_1, a_2, a)$ *and* $c \rightsquigarrow_\alpha a$,
- $[\![t]\!] \lhd [\![t_\alpha]\!]$ *iff for every* $c_1, c_2 \in [\![\tau]\!]$, $a_1, a_2 \in [\![\tau_\alpha]\!]$, *and* $b \in [\![\mathsf{Bool}]\!]$, *if* $c_1 \rightsquigarrow_\alpha a_1$, $c_2 \rightsquigarrow_\alpha a_2$, *and* $[\![t]\!](c_1, c_2, b)$, *then* $[\![t_\alpha]\!](a_1, a_2, b)$.

As noted in Section 2.1, when defining an abstract type $\tau_\alpha$ for integers in Bandera, the user only needs to use BASL to specify its abstraction domain $[\![\tau_\alpha]\!]$ and its abstraction relation $\rightsquigarrow_\alpha$. Safe operations and tests involving the new abstract type are generated by Bandera automatically using a calculation similar in style to those used to calculate weakest-preconditions in predicate abstraction.

For example, suppose that the user wants to define a new integer abstraction $\tau_{signs}$. The user then can define the abstraction domain as $[\![\tau_{signs}]\!] =$

$\{neg, zero, pos\}$. In writing the abstraction function, the user would write simple predicates to create a covering of the integer domain (e.g., as shown for the EvenOdd BASL definition in Figure 2). In our formal notation, we capture this using the following predicates for each of the abstract tokens in the $\tau_{signs}$ domain: $neg? = \lambda x.\, x < 0, zero? = \lambda x.\, x = 0$, and $pos? = \lambda x.\, x > 0$. Given these predicates, we can define the associated abstraction relation $\rightsquigarrow_{signs}$ as

$$\forall x \in [\![\mathsf{Int}]\!] \,.\, \forall a \in [\![\tau_{signs}]\!] \,.\, x \rightsquigarrow_{signs} a \text{ iff } a?(x).$$

Given these definitions, Bandera automatically constructs a *safe* definition for each abstract operation and test essentially by (a) beginning with a worst case assumption that the relation defining the abstract operation is total (note that this is a safe definition since a total relation covers all possible behaviors of the concrete system), and then (b) calling the decision procedures of PVS to see if individual tuples in the relation can be eliminated.

For example, consider how the definition of $+_{signs}$ would be derived. Since the abstract domain $[\![\tau_{signs}]\!]$ contains 3 abstract tokens, we would initially have 27 tuples in the total relation associated with $[\![+_{signs}]\!]$. Now, for each tuple $(a_1, a_2, a) \in [\![+_{signs}]\!]$, Bandera would construct a purported theorem in the input syntax of PVS which we represent as follows

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!] \,.\, a_1?(n_1) \wedge a_2?(n_2) \implies \neg a?([\![+]\!](n_1, n_2)).$$

If PVS can prove the fact above, then the tuple $(a_1, a_2, a)$ can safely be eliminated relation defining $[\![+_{signs}]\!]$ because, since the theorem is true, $a$ is never needed to simulate the result of adding $n_1$ and $n_2$.

Specifically, consider the three tuples $(pos, pos, neg)$, $(pos, pos, zero)$, and $(pos, pos, pos)$. The decision procedure is able prove the two theorems

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!] \,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg neg?([\![+]\!](n_1, n_2)),$$
$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!] \,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg zero?([\![+]\!](n_1, n_2)),$$

but it fails to prove

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!] \,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg pos?([\![+]\!](n_1, n_2)),$$

so Bandera would remove the first two tuples from the relation and have $[\![+_{signs}]\!](pos, pos) = \{pos\}$ (depicting the relation as a set-valued function).

In summary, the definitions for abstract operations and tests for integer abstractions is as follows.

**Definition 2.** *Let $\tau_\alpha$ be an integer abstraction.*

– *For all $a_1, a_2, a \in [\![\tau_\alpha]\!]$, $[\![o_\alpha]\!](a_1, a_2, a)$ iff the decision procedure fails to decide*

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!] \,.\, a_1?(n_1) \wedge a_2?(n_2) \implies \neg a?([\![o]\!](n_1, n_2)).$$

– *For all $a_1, a_2 \in [\![\tau_\alpha]\!]$ and $b \in [\![\mathsf{Bool}]\!]$, $[\![t_\alpha]\!](a_1, a_2, b)$, iff the decision procedure fails to decide*

$$\forall n_1, n_2.\, a_1?(n_1) \wedge a_2?(n_2) \implies b \neq [\![t]\!](n_1, n_2).$$

This technique can also be used to infer coercions between two integer abstractions $\alpha$ and $\alpha'$. Specifically, $(a, a') \in [\![ [\tau_\alpha \ll \tau_{\alpha'}] ]\!]$ if the decision procedure fails to decide $\forall n.a?(n) \Longrightarrow \neg a'?(n)$.

## 5 Attaching Abstractions

Bandera's process for transforming concrete programs to abstract programs requires that each variable and each occurrence of a constant, operation, and test in the concrete program be bound to an abstraction type; these bindings indicate to the program transformer which versions of abstract program elements should be used in place of a particular concrete element (e.g., which abstract version of the + operation should be used in place of a particular concrete instance of +). Requiring the user to specify all of this binding information directly would put a tremendous burden on the user.

To construct the desired binding information while at the same time keeping user effort to a minimum, Bandera provides an abstract type inference facility. A user begins the type inference phase by selecting abstractions from the abstraction library for a small number of program variables that the user deems relevant. Bandera provides a default set of coercions between library abstractions for each concrete type that the user can override if desired. For example, for any Int abstraction $\tau_\alpha$ a coercion relation $[\![ [\mathsf{Int} \ll \tau_\alpha] ]\!]$ is automatically introduced by taking $[\![ [\mathsf{Int} \ll \tau_\alpha] ]\!] = \rightsquigarrow_\alpha$ (i.e., the coercion is just the abstraction function) and a coercion relation $[\![ [\tau_\alpha \ll \mathsf{Point}] ]\!]$ is automatically introduced where $[\![ [\tau_\alpha \ll \mathsf{Point}] ]\!]$ simply maps all elements of $[\![ \tau_\alpha ]\!]$ to the single element of the Point domain. The default coercion definitions plus any user-defined coercions give rise to a subtyping structure for each concrete type. This abstraction selection and subtyping/coercion information forms the input to the type inference component. In the Bandera methodology, boolean variables are never abstracted since they already have a small domain. We will model this in the definitions below by abstracting all boolean variables with an identity abstraction which has the effect of leaving boolean variables and values unchanged by the abstraction process.

Given the program, initial abstraction selection, and subtyping information, type inference proceeds in two steps.

1. Abstract types are propagated along value flows to determine abstraction bindings for as many constructs as possible. If there are any abstract type conflicts during this process, type inference is halted and the user is presented with an error message.
2. Some variables and constructs may not be assigned abstract types in the first step because they are *independent* of the variables in the initial abstraction selection. Abstractions for independent variables/constructs are determined according to default abstractions specified by the user. The most commonly used defaults are (a) to abstract all independent variables/constructs with the *point* abstraction which has the effect of discarding all information about values manipulated by these constructs, or (b) to abstract these constructs

```
Annotated Program                          Abstracted Program

(m n)                                      (m n)
(init)                                     (init)

  init: result := 1^1;                       init: result := odd;
        goto test;                                 goto test;

  test: if (<(n^2,1^3))^4                    test  if <(n, 1)
        then end else loop;                        then end else loop;

  loop: result := (*(result^5,m^6))^7;       loop: result := *_eo(result, m);
        n := (-(n^8,1^9))^10;                      n := -(n, 1);
        goto test;                                 goto test;

  end:  return;                              end:  return;
```

**Fig. 9.** Annotated and abstracted versions of the FCL power program

with the identity abstraction which has the effect of preserving all information about the values manipulated by these constructs.

As an example, consider the power program and the LTL formula from Section 3.2. Following the methodology for selecting abstractions in Section 2.1, the EvenOdd abstraction would be *appropriate* for the variable `m` of the power program in Figure 6. Intuitively, an abstraction is appropriate for a property when it is precise enough to decide all propositions appearing in the property.

Suppose that the abstraction library that is used contains the concrete (identity) Int, EvenOdd, Signs, and Point abstractions for integer types. The subtyping relation between abstractions must always form a lattice; the default subtyping relation has Int as the least element, Point as the greatest element, with the remaining abstractions being unrelated to each other. This lattice is augmented with an additional element $\perp$ which ends up being bound to variables/constructs whose type is unconstrained due to the fact that they are independent of the initial abstraction selection. The second phase of the type inference process described above involves replacing $\perp$ with one of the default options for unconstrained types. We write $\sqsubseteq$ to denoted the augmented subtyping ordering used in the type inference process. Intuitively, if $\tau_1 \sqsubseteq \tau_2$, then $\tau_1$ is at least as precise as $\tau_2$.

To represent the binding between program constructs and abstractions, we will assume that each expression abstract syntax tree node is annotated with an unique label as in the left side of Figure 9. Bindings are then captured by a *cache* structure $\hat{\mathcal{C}}$ which maps variables, labeled AST nodes, and operator/test instances to types in the augmented subtyping lattice. Following convention, type inference is phrased as a constraint-solving problem in which constraints

*Syntax Domain Extension*

$$\begin{array}{l} \bar{l} \in \text{Exp-Labels[FCL]} \\ \bar{t} \in \text{Terms[FCL]} \end{array}$$

*Grammar with Labeled Expression Extension*

$$e ::= \bar{t}^{\bar{l}} \qquad \bar{t} ::= c \mid x \mid o(e_1, e_2) \mid t(e_1, e_2)$$

*Constraints (Excerpts)*

$$\begin{array}{lll}
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models c^{\bar{l}} & \text{iff} & \bot \sqsubseteq \hat{\mathcal{C}}(\bar{l}) \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models x^{\bar{l}} & \text{iff} & \hat{\mathcal{C}}(x) = \hat{\mathcal{C}}(\bar{l}) \text{ and } x\mathcal{R}\bar{l} \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models (o(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2}))^{\bar{l}} & \text{iff} & \langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models \bar{t}_1^{\bar{l}_1} \text{ and } \langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models \bar{t}_2^{\bar{l}_2} \\
 & & \text{and } \hat{\mathcal{C}}(\bar{l}_1) \sqsubseteq \hat{\mathcal{C}}(o, \bar{l}) \text{ and } \hat{\mathcal{C}}(\bar{l}_2) \sqsubseteq \hat{\mathcal{C}}(o, \bar{l}) \\
 & & \text{and } \hat{\mathcal{C}}(o, \bar{l}) = \hat{\mathcal{C}}(\bar{l}) \\
 & & \text{and } \bar{l}_1\mathcal{R}(o, \bar{l}) \text{ and } \bar{l}_2\mathcal{R}(o, \bar{l}) \\
 & & \text{and } (o, \bar{l})\mathcal{R}\bar{l} \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models x := \bar{t}^{\bar{l}} & \text{iff} & \hat{\mathcal{C}}(\bar{l}) \sqsubseteq \hat{\mathcal{C}}(x) \text{ and } \bar{l}\mathcal{R}x
\end{array}$$

*Data Structures*

$$\begin{array}{lll}
\text{(graph nodes)} & \mathcal{N} = & \text{Variables[FCL]} \cup \text{Exp-Labels[FCL]} \cup \\
 & & ((\text{Operations[FCL]} \cup \text{Tests[FCL]}) \times \text{Exp-Labels[FCL]}) \\
\text{(cache)} & \hat{\mathcal{C}} = & \mathcal{N} \rightharpoonup \mathsf{Types}[\Sigma_\alpha]_\bot \\
\text{(dependencies)} & \mathcal{R} \subseteq & \mathcal{N} \times \mathcal{N}
\end{array}$$

**Fig. 10.** Type Inference

on cache entries are generated in a syntax-directed traversal of the program and then solved using a union-find data structure.

## 5.1 Type Inference: Generating Constraints

Figure 10 presents the specification of constraint generation in the style of [23]. The data structures used include a type dependency graph with nodes $\mathcal{N}$ that are either variables, labels of AST nodes, or operation/test occurrences which are identified by a pair consisting of the operation/test symbol and a label for the node in which the operation/test instance occurs. As described above, a cache $\hat{\mathcal{C}}$ maps each graph node to a type from the augmented lattice $\mathsf{Types}[\Sigma_\alpha]_\bot$. The relation $\mathcal{R}$ maintains dependency information associated with value flows. This information is used in the second phase of type inference (described above) where constructs that are independent of initial abstraction bindings are assigned types. Due to the manner in which constraints are constructed, $x\mathcal{R}y$ implies $\hat{\mathcal{C}}(x) \sqsubseteq \hat{\mathcal{C}}(y)$.

Constraints on $\hat{\mathcal{C}}$ and $\mathcal{R}$ are generated in a syntax-directed manner according to the following intuition.

- There are no constraints on constants except those imposed by the context (which are captured by other rules). Thus, the type assigned to a constant can be any value in the lattice at or above $\bot$.
- A variable reference expression (which yields the value of variable) should have the same type as the variable itself, and $x\mathcal{R}\bar{l}$ captures the fact that the type of $x$ must be at least as precise as that of $\bar{l}$.
- The abstraction associated with the arguments of an operator application must lie at or below the abstraction associated with the operator itself. Note that this will allow the type of an argument to be coerced to the type of the operator.
- The abstraction associated with the right-hand side of an assignment must lie at or below the abstraction associated with the variable being assigned. Note that this will allow the type of the right-hand side to be coerced to the type of the left-hand side.

In addition to the constraints generated from the rules above, for every variable $x$ appearing in the user's initial abstraction selection, a constraint $\hat{\mathcal{C}}(x) = \tau_x$ is added where $\tau_x$ is the abstraction type chosen for $x$. For each remaining variable $y$, a constraint $\bot \sqsubseteq \hat{\mathcal{C}}(y)$ is added.

Generating the least solution for a system of constraints yields abstraction bindings that are as precise as possible (with respect to the subtyping rules). In particular, the ability to use coercions at operation/test arguments, etc. avoids having argument abstractions determined by the context (i.e., having to receive the same type as the operation), and thus allows abstraction assignments at such positions to be as precise as possible.

For the power program example on the left in Figure 9, the following constraints are generated (ignoring $\mathcal{R}$ for now).

$$\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{1}) = \bot & \hat{\mathcal{C}}(1) \sqsubseteq \hat{\mathcal{C}}(\mathtt{result}) & \hat{\mathcal{C}}(\mathtt{n}) = \hat{\mathcal{C}}(2) & \hat{\mathcal{C}}(2) \sqsubseteq \hat{\mathcal{C}}(<,4) \\
\hat{\mathcal{C}}(\mathtt{3}) = \bot & \hat{\mathcal{C}}(3) \sqsubseteq \hat{\mathcal{C}}(<,4) & \hat{\mathcal{C}}(<,4) = \hat{\mathcal{C}}(4) & \hat{\mathcal{C}}(\mathtt{result}) = \hat{\mathcal{C}}(5) \\
\hat{\mathcal{C}}(5) \sqsubseteq \hat{\mathcal{C}}(*,7) & \hat{\mathcal{C}}(\mathtt{m}) = \hat{\mathcal{C}}(6) & \hat{\mathcal{C}}(6) \sqsubseteq \hat{\mathcal{C}}(*,7) & \hat{\mathcal{C}}(*,7) = \hat{\mathcal{C}}(7) \\
\hat{\mathcal{C}}(7) \sqsubseteq \hat{\mathcal{C}}(\mathtt{result}) & \hat{\mathcal{C}}(\mathtt{n}) = \hat{\mathcal{C}}(8) & \hat{\mathcal{C}}(8) \sqsubseteq \hat{\mathcal{C}}(-,10) & \hat{\mathcal{C}}(9) = \bot \\
\hat{\mathcal{C}}(9) \sqsubseteq \hat{\mathcal{C}}(-,10) & \hat{\mathcal{C}}(-,10) = \hat{\mathcal{C}}(10) & \hat{\mathcal{C}}(10) \sqsubseteq \hat{\mathcal{C}}(\mathtt{n}) &
\end{array}$$

Given a user selection of EvenOdd for m, the following additional constraints are generated.

$$\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} \text{ and } \bot \sqsubseteq \hat{\mathcal{C}}(\mathtt{n}) \text{ and } \bot \sqsubseteq \hat{\mathcal{C}}(\mathtt{result})$$

## 5.2 Type Inference: Solving Constraints

Once constraints are generated as described above, Bandera finds the least solution with respect to $\sqsubseteq$. For example, the least solution for the constraints from the power example is as follows.

$$\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(\mathtt{n}) = \bot & \hat{\mathcal{C}}(\mathtt{result}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(1) = \bot \\
\hat{\mathcal{C}}(2) = \bot & \hat{\mathcal{C}}(3) = \bot & \hat{\mathcal{C}}(4) = \bot & \hat{\mathcal{C}}(5) = \mathsf{EvenOdd} \\
\hat{\mathcal{C}}(6) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(8) = \bot & \hat{\mathcal{C}}(9) = \bot \\
\hat{\mathcal{C}}(10) = \bot & \hat{\mathcal{C}}(<,4) = \bot & \hat{\mathcal{C}}(*,7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(-,10) = \bot
\end{array}$$

As illustrated by the presence of $\perp$ in some of the bindings above, this step may leave the type of some variables/AST-nodes unconstrained. In general, $\perp$-bindings such as those shown above actually fall into two categories.

The first category contains variables/AST-nodes that produce values that can flow into a context that *is* constrained. For example, this is the case with the AST node labeled 1: the constant `1` flows into the variable `result` (which is bound to EvenOdd) as a consequence of the assignment. Thus, to obtain an abstraction assignment that is precise as possible (i.e., one that does not "bump up" the abstraction assigned to `result` to a higher value in the lattice of abstractions), the abstraction chosen for such nodes should not be greater than that of any constrained context into which the values produced by such nodes can flow. The dependency information provided by the $\mathcal{R}$ structure is used to determine if a unconstrained node falls into this category (i.e., if a constrained context be reached by following the dependency arcs of $\mathcal{R}$).

The second category contains variables/AST-nodes that produce values that do *not* flow into constrained contexts. There are several reasonable views as to what the abstraction bindings should be for items in this category. One view is that one should generate models that are as abstract as possible in order to reduce the size of the state space as much as possible. Following this view, one might bind the Point abstraction to each item in this category. On the other hand, this could result in such an over-approximation that infeasible counter-examples are introduced. Thus, one might want to generate models that are as precise as possible. Following this view, one might bind the Int abstraction to each item in this category. Note that although such a choice might lead to an unbounded state-space (since integers are left unabstracted in the program), this is still quite useful in practice since model-checkers such as Spin allow arbitrary integer values with bounds imposed only by the number of bits used in the storage class (e.g., `byte`, `int`, etc.). Bandera actually provides a flexible mechanism for declaring upper and lower-bounds on individual integer variables.

In any case, the two categories above are currently treated as follows in Bandera. For the first category, Bandera binds items to the concrete Int abstraction. This always satisfies the constraints since Int is the least element in the non-augmented abstraction lattice, and it follows the heuristic of keeping abstractions as precise as possible. At the point where concrete integers flow into abstracted contexts, an appropriate coercion will be introduced in the model. Since items in the second category are completely unconstrained, Bandera allows the user to select a default abstraction $\tau_{\mathsf{def}}$ (typically, Int or Point) for these items.

Capturing this in our formal notation, Bandera proceeds by building a new $\hat{\mathcal{C}}'$ from $\hat{\mathcal{C}}$ as follows.

$$\hat{\mathcal{C}}'(x) = \begin{cases} \hat{\mathcal{C}}(x), & \text{if } \hat{\mathcal{C}}(x) \neq \perp \\ \mathsf{Int}, & \text{if } \hat{\mathcal{C}}(\mathcal{R}^*(x)) \neq \{\perp\} \\ \tau_{\mathsf{def}}, & \text{if } \hat{\mathcal{C}}(\mathcal{R}^*(x)) = \{\perp\} \end{cases}$$

That is, already assigned an abstraction keep the same abstraction in $\hat{\mathcal{C}}'$, items from the first category above get assigned $\mathsf{Int}$, and items from the second category get assigned the chosen default abstraction. In the example program, this results in the following final bindings.

$$\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(\mathtt{n}) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(\mathtt{result}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(1) = \mathsf{Int} \\
\hat{\mathcal{C}}(2) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(3) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(4) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(5) = \mathsf{EvenOdd} \\
\hat{\mathcal{C}}(6) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(8) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(9) = \tau_{\mathsf{def}} \\
\hat{\mathcal{C}}(10) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(<, 4) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(*, 7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(-, 10) = \tau_{\mathsf{def}}
\end{array}$$

As a future improvement to the treatment of items from the first category (where $\hat{\mathcal{C}}(\mathcal{R}^*(x)) \neq \{\bot\}$), it may be desirable to give user the flexibility to replace $\mathsf{Int}$ with any less precise abstraction $\tau$ that still lies at or below the abstraction of any context that an item's value may flow into, i.e.,

$$\tau \sqsubseteq \sqcap\{\hat{\mathcal{C}}'(y) \mid y \in \mathcal{R}^*(x) \text{and } \hat{\mathcal{C}}'(y) \neq \bot\}.$$

From a usability standpoint, it is important to note that the type inference algorithm outlined above is efficient and scales well, and that the process of selecting abstractions and visualizing type inference results is interactive. Thus, the user can experiment with the abstraction selection with ease, e.g., by incrementally adding the abstraction selections and visualizing the effects of each selection.

Bandera provides feedback to the user if the abstraction selection is inconsistent. For example, suppose that the user selects `m` as $\mathsf{EvenOdd}$ abstraction and `result` as $\mathsf{Signs}$ abstraction. A conflict arises because of the following constraints cannot be satisfied.

$$\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(\mathtt{result}) = \mathsf{Signs} & \hat{\mathcal{C}}(\mathtt{result}) = \hat{\mathcal{C}}(5) & \hat{\mathcal{C}}(5) \sqsubseteq \hat{\mathcal{C}}(*, 7) \\
\hat{\mathcal{C}}(\mathtt{m}) = \hat{\mathcal{C}}(6) & \hat{\mathcal{C}}(6) \sqsubseteq \hat{\mathcal{C}}(*, 7) & \hat{\mathcal{C}}(*, 7) = \hat{\mathcal{C}}(7) & \hat{\mathcal{C}}(7) \sqsubseteq \hat{\mathcal{C}}(\mathtt{result})
\end{array}$$

## 6 Generating Abstract Programs

Once abstract type inference has been carried on a $\Sigma$-program interpreted with $A$, the set of $[\Sigma, A]$-compatible abstractions $\{\alpha_1, ..., \alpha_n\}$ chosen by the user and the final abstraction bindings from the type inference process are used to induce an abstract program based on a new signature and algebra $[\Sigma_\alpha, A_\alpha]$ that combines the selected abstractions. Section 2.2 noted that this process is implemented in Bandera by replacing primitive concrete Java operations in the program to be abstracted with calls to Java methods in abstraction library classes that implement semantics associated with abstract versions of operations.

We first formalize the notion of these library classes/methods by reifying the abstraction semantics from Section 4 into constants and symbols to be used in the signature for the abstract program. Specifically, given a *$\tau$-abstraction* $\alpha$, we form a new type named by $\alpha$'s abstraction type identifier $\tau_\alpha$. For this type, constants, operation symbols, and test symbols, are constructed as follows.

- $\mathsf{Cons}[\tau_\alpha] = \{\underline{a} \mid a \in [\![\tau_\alpha]\!]\}$. That is, $\tau_\alpha$ constants are formed by introducing a fresh symbol $\underline{a}$ for each element of the abstract domain. This corresponds to the use of constants such as `EvenOdd.Even` in abstracted Java programs (see Section 2.2).
- $\mathsf{Ops}[\tau_\alpha] = \{o_\alpha \mid o \in \mathsf{Ops}[\tau]\}$. That is, $\tau_\alpha$ operation symbols are formed by introducing a fresh symbol $o_\alpha$ for each operation symbol associated with the $\Sigma$-type $\tau$ being abstracted. This corresponds to the use of method calls such as `EvenOdd.add` (see Section 2.2).
- $\mathsf{Tests}[\tau_\alpha] = \{t_\alpha \mid t \in \mathsf{Ops}[\tau]\}$. That is, $\tau_\alpha$ test symbols are formed by introducing a fresh symbol $t_\alpha$ for each test symbol associated with the $\Sigma$-type $\tau$ being abstracted.

With these syntactic elements in hand, we now form a signature to be used for the abstracted program by combining the symbols introduced above. Given user-selected abstractions $\{\alpha_1, ..., \alpha_n\}$ along with default and user declared coercions, a new signature $\Sigma_\alpha$ representing this combination of abstractions is constructed as follows.

- $\mathsf{Types}[\Sigma_\alpha] = \{\tau_{\alpha_1}, ..., \tau_{\alpha_n}\}$ where $\tau_{\alpha_i}$ is the type identifier corresponding to each abstraction $\alpha_i$.
- $\mathsf{Ops}[\Sigma_\alpha] = \bigcup_{i\in\{1,\dots,n\}} \mathsf{Ops}[\alpha_i]$.
- $\mathsf{Tests}[\Sigma_\alpha] = \bigcup_{i\in\{1,\dots,n\}} \mathsf{Tests}[\alpha_i]$.
- $\ll_{\Sigma_\alpha} = \{(\tau_1, \tau_2) \mid$ a coercion exists from $\tau_1$ and $\tau_2\}$.

An appropriate abstract $\Sigma_\alpha$-algebra is now formed in a straightforward manner as follows.

- For all types $\tau_\alpha \in \Sigma_\alpha$, $[\![\tau_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![\tau_\alpha]\!]$ (i.e., the domain specified in the $\alpha$ abstraction).
- For all $o_\alpha \in \mathsf{Ops}[\Sigma_\alpha]$ where $o_\alpha$ is a $\tau_\alpha$ operation, $[\![o_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![o_\alpha]\!]$ (i.e., the operation interpretation specified in the $\alpha$-abstraction).
- For all $t_\alpha \in \mathsf{Tests}[\Sigma_\alpha]$ where $t_\alpha$ is a $\tau_\alpha$ test, $[\![t_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![t_\alpha]\!]$ (i.e., the test interpretation specified in the $\alpha$-abstraction).
- For each coercion symbol $[\tau_1 \ll \tau_2]$, the corresponding coercion relation is defined for default coercions as explained earlier or defined by the user.

Figure 11 presents rules that formalize the translation of concrete programs to abstract programs. The rules are guided by bindings of labeled AST nodes to abstract types as captured by the cache $\hat{\mathcal{C}}$.

The first group of rules in Figure 11 have the form $\vdash c \uparrow^{\tau_\alpha}_{\tau} e_\alpha$ and describe how constants of type source $\tau$ may be transformed (or coerced) to abstract constants of target type $\tau_\alpha$. If there is no difference between the source and target types, then the transformation is the identity transformation. If there is a single abstract constant associated with a concrete integer constant then the transformation yields that abstract constant. Otherwise, a coercion expression is introduced to carry out the transformation during model-checking. Recall that boolean program elements are never abstracted, so the presented rules cover all possible cases for boolean constants.

*Constant Coercion*

$$\vdash c \uparrow^{\mathsf{Bool}}_{\mathsf{Bool}} c \qquad\qquad \vdash c \uparrow^{\mathsf{Int}}_{\mathsf{Int}} c$$

$$\vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} \underline{a} \quad \begin{array}{l}\text{if } \mathsf{Int} \neq \tau_\alpha \\ \text{where } \rightsquigarrow_\alpha ([\![c]\!]) = \{a\}\end{array} \qquad \vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} [\mathsf{Int} \ll \tau_\alpha]\, c \quad \begin{array}{l}\text{if } \mathsf{Int} \neq \tau_\alpha \\ \text{where } \rightsquigarrow_\alpha ([\![c]\!]) \neq \{a\}\end{array}$$

*Expression Coercion*

$$\frac{\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha}{\hat{\mathcal{C}} \vdash \bar{t}^{l} \Uparrow^{\tau_\alpha}_{\tau_\alpha} e_\alpha}$$

$$\frac{\vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} e_\alpha}{\hat{\mathcal{C}} \vdash c^{l} \Uparrow^{\tau_\alpha}_{\mathsf{Int}} e_\alpha} \quad \text{where } \mathsf{Int} \neq \tau_\alpha \qquad \frac{\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha}{\hat{\mathcal{C}} \vdash \bar{t}^{l} \Uparrow^{\tau_{\alpha 2}}_{\tau_{\alpha_1}} [\tau_{\alpha_1} \ll \tau_{\alpha_2}]\, e_\alpha} \quad \begin{array}{l}\text{if } \tau_{\alpha_1} \neq \tau_{\alpha_2} \\ \text{and } \bar{t} \neq c\end{array}$$

*Expression Translation*

$$\frac{\vdash c \uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\tau} e_\alpha}{\hat{\mathcal{C}} \vdash c \Rightarrow e_\alpha} \quad \text{where } c \in \mathsf{Cons}[\tau] \qquad \hat{\mathcal{C}} \vdash x \Rightarrow x$$

$$\frac{\hat{\mathcal{C}} \vdash \bar{t}_1^{\bar{l}_1} \Uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\hat{\mathcal{C}}(\bar{l}_1)} e_{\alpha 1} \qquad \hat{\mathcal{C}} \vdash \bar{t}_2^{\bar{l}_2} \Uparrow^{\hat{\mathcal{C}}(\bar{l}_2)}_{\hat{\mathcal{C}}(\bar{l})} e_{\alpha 2}}{\hat{\mathcal{C}} \vdash o(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2})^{l} \Rightarrow o_{\hat{\mathcal{C}}(\bar{l})}(e_{\alpha 1}, e_{\alpha 2})} \qquad \frac{\hat{\mathcal{C}} \vdash \bar{t}_1^{\bar{l}_1} \Uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\hat{\mathcal{C}}(\bar{l}_1)} e_{\alpha 1} \qquad \hat{\mathcal{C}} \vdash \bar{t}_2^{\bar{l}_2} \Uparrow^{\hat{\mathcal{C}}(\bar{l}_2)}_{\hat{\mathcal{C}}(\bar{l})} e_{\alpha 2}}{\hat{\mathcal{C}} \vdash t(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2})^{l} \Rightarrow t_{\hat{\mathcal{C}}(\bar{l})}(e_{\alpha 1}, e_{\alpha 2})}$$

**Fig. 11.** Translating concrete programs to abstract programs (excerpts)

The second group of rules in Figure 11 have the form $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Uparrow^{\tau_{\alpha 2}}_{\tau_{\alpha 1}} e_\alpha$ and are similar in spirit to the rules above. If there is no difference between the source and target types, the result of the transformation is simply the result of recursive transforming the labeled term $\bar{t}^{\bar{l}}$. If a constant is being coerced, the constant coercion rules are used. On non-constant terms where the source type is different from the target type, a coercion is inserted after recursively transforming the argument of the translation.

The third group of rules in Figure 11 have the form $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha$. The constant coercion rules are used to transform a constant from its concrete type to a possibly abstract type. In the rules for operations and tests, the expression coercion rules are used to transform and possibly coerce the arguments. Then, the concrete operation is replaced by the abstract version indicated by the corresponding cache entry.

The remaining rules which are not displayed in Figure 11 are straightforward — remaining constructs such as conditions, returns, and gotos are preserved while transforming all subexpressions.

The rules of Figure 11 generate a syntactically correct abstract program (the proposition below captures this for expressions).

**Proposition 1 (Syntactically correct abstract expressions).**
*Let $\Gamma \vdash_\Sigma e : \tau$ and $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha$ where $\hat{\mathcal{C}}$ is compatible with $\Gamma$, $\bar{t}^{\bar{l}}$ is the labeled version of $e$. Then $\Gamma_\alpha \vdash_{\Sigma_\alpha} e_\alpha : \hat{\mathcal{C}}(\bar{l})$ where $domain(\Gamma_\alpha) = domain(\Gamma)$ and $\forall x \in domain(\Gamma_\alpha)\,.\, \Gamma_\alpha(x) = \hat{\mathcal{C}}(x)$.*

Applying the translation rules of Figure 11 to the power program in Figure 6 with context $\hat{\mathcal{C}}'$ from Section 5 gives an abstracted power program shown on the right in Figure 9.

In the definitions that follow, when $\Gamma_\alpha$ arises from $\Gamma$ due to the program abstraction process captured in the proposition above, we say that $\Gamma_\alpha$ is a abstract version of $\Gamma$.

We now consider some basic safety properties that we need to express the correctness of abstraction. If $\Sigma_\alpha$ and $A_\alpha$ represent the abstract signature and algebra generated from a basis $\Sigma$ and $A$, and $\Gamma_\alpha$ is an abstract version of $\Gamma$ built using a set $\alpha$ of abstractions, the safety relation between a $\Gamma$-compatible store $\sigma$ and a $\Gamma_\alpha$-compatible store $\sigma_\alpha$ (denoted $\sigma \lhd \sigma'$) holds iff for all $x \in$ *domain*$(\Gamma)$ . $\sigma(x) \rightsquigarrow_{\Gamma_\alpha(x)} \sigma_\alpha(x)$, i.e., the store values for each $x$ are related by the abstraction relation associated with $x$'s abstract type.

**Lemma 1.** *(Safety for expressions) Let $\Gamma \vdash_\Sigma e : \tau$ and let $\Gamma_\alpha \vdash_{\Sigma_\alpha} e_\alpha : \tau_\alpha$ be the abstract expression constructed by the type inference and program abstraction process described above. Let $\sigma \in [\![\Gamma]\!]^A_\Sigma$, $\sigma_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$, $v \in [\![\tau]\!]$, and $b \in [\![\mathsf{Bool}]\!]$:*

- *$\sigma \lhd \sigma_\alpha$ and $\sigma \vdash e \Rightarrow v$ implies $\exists v_\alpha \in [\![\tau_\alpha]\!]$ such that $\sigma_\alpha \vdash e_\alpha \Rightarrow v_\alpha$ and $v \rightsquigarrow_{\tau_\alpha} v_\alpha$*
- *$\sigma \lhd \sigma_\alpha$ and $\sigma \vdash e \Rightarrow b$ implies $\sigma_\alpha \vdash e_\alpha \Rightarrow b$*

**Lemma 2.** *(Safety for transitions) Let $\Sigma_\alpha$ and $A_\alpha$ represent the abstract signature and algebra generated from a basis $\Sigma$ and $A$, and let $\Gamma_\alpha$ be an abstract version of $\Gamma$ built using a set $\alpha$ of abstractions, then for every $\sigma, \sigma' \in [\![\Gamma]\!]^A_\Sigma$, and for every $\sigma_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$, and $n, n' \in \mathrm{Nodes[FCL]}$, $\sigma \lhd \sigma_\alpha$ and $(n, \sigma) \longmapsto (n', \sigma')$ implies $\exists \sigma'_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$ such that $(n, \sigma_\alpha) \longmapsto (n', \sigma'_\alpha)$ and $\sigma' \lhd \sigma'_\alpha$.*

Given these basic properties, the fact that a concrete program is simulated by its abstracted counterpart is established in a straightforward manner.

## 7 Generating Abstract Properties

When abstracting properties, we want to ensure that if an abstracted property holds for an abstracted program, then the original property holds for the original program. In order to achieve this goal, properties have to be *under-approximated.* This is the dual of the process of abstracting a program. A program is abstracted by over-approximating its behaviors, i.e., the abstracted program may contain more behaviors that are not present in the original program due to the imprecision introduced in the abstraction process. Thus, if the abstracted program satisfies a particular requirement, then we can safely conclude that the original program satisfies the requirement. When abstracting a property, however, the abstraction may introduce imprecision such that the abstracted property may allow more behaviors of the program that satisfies it. Thus, we only consider the cases where the abstracted property can precisely decide the original property, i.e., under-approximating it.

Property abstraction begins in Bandera by performing type-inference on and abstracting each expression $e$ in the property where property expressions are constructing following the grammar in Figure 7. Let $e$ be a property expression such that $\mathit{\Gamma} \vdash e : \mathsf{Bool}$ and $\mathit{domain}(\mathit{\Gamma}) = \text{Variables}[e]$ where $\text{Variables}[e]$ denotes the set of variables occurring in $e$. Furthermore, assume that $\mathit{\Gamma}_\alpha$ is an abstract version of $\mathit{\Gamma}$ and that $e_\alpha$ is an abstract version of $e$ (i.e., as generated by the transformation process described in the previous section where we have $\mathit{\Gamma}_\alpha \vdash e_\alpha : \mathsf{Bool}$).

Section 3.3 defined the semantics of expression propositions as an under-approximation (i.e., an expression is only considered to be true when it does not evaluate to false). Bandera represents this semantics by constructing explicitly a disjunctive normal form that encodes the cases of stored values that cause an expression proposition to be interpreted as *true*.

For an abstract property expression $e_\alpha$ such that $\mathit{\Gamma}_\alpha \vdash e_\alpha : \mathsf{Bool}$, we denote the set of $\mathit{\Gamma}_\alpha$-compatible stores that make $e_\alpha$ true as

$$\mathit{TrueStores}[\mathit{\Gamma}_\alpha](e_\alpha) \stackrel{\text{def}}{=} \{\sigma_\alpha \mid \sigma_\alpha \in [\![\mathit{\Gamma}_\alpha]\!] \; \mathit{and} \; \exists n. [\![e_\alpha]\!](n, \sigma_\alpha)\}.$$

Note that the semantics of expression propositions is independent of control points $n$.

Next, we denote a conjunction that specifies the bindings of a store $\sigma_\alpha$ as

$$\mathit{Bindings}(\sigma_\alpha) \stackrel{\text{def}}{=} \bigwedge \{=(x, a) \mid (x, a) \in \sigma_\alpha\}.$$

The following function $\mathcal{T}$ specifies the transformation that Bandera uses to generate abstracted properties (the transformation is structure preserving except for the case of proposition expressions which we give below).

$$\mathcal{T}(e_\alpha) = \bigvee \{\mathit{Bindings}(\sigma_\alpha) \mid \sigma_\alpha \in \mathit{TrueStores}[\mathit{\Gamma}_\alpha](e_\alpha)\}$$
$$\mathcal{T}(\neg e_\alpha) = \bigvee \{\mathit{Bindings}(\sigma_\alpha) \mid \sigma_\alpha \in \mathit{TrueStores}[\mathit{\Gamma}_\alpha](\neg e_\alpha)\}$$

For example, suppose that we want to abstract the property

$$\Box\neg[\mathit{end}] \vee \Box(\neg[\mathit{init}] \vee \neg{=}(\%(\mathtt{m}, 2), 1) \vee \Diamond([\mathit{end}] \wedge {=}(\%(\mathtt{result}, 2), 1))).$$

with `m` and `result` abstracted using the evenodd abstraction. After applying $\mathcal{T}$, the property becomes

$$\Box\neg[\mathit{end}] \vee \Box(\neg[\mathit{init}] \vee {=}(\mathtt{m}, \mathit{even}) \vee \Diamond([\mathit{end}] \wedge {=}(\mathtt{result}, \mathit{odd}))).$$

This is the case where the abstraction is precise enough to decide the original property.

However, suppose that now `m` is abstracted using the evenodd abstraction, and `result` is abstracted using the point abstraction. After applying $\mathcal{T}$, the property becomes

$$\Box\neg[\mathit{end}] \vee \Box(\neg[\mathit{init}] \vee {=}(\mathtt{m}, \mathit{even}) \vee \Diamond([\mathit{end}] \wedge \mathit{false})).$$

This is the case where an abstraction is not precise enough to decide a proposition, i.e., $=(\%(\texttt{result}, 2), 1)$ is under-approximated to *false*, because *point* is not precise enough. When submitted to a model checker, infeasible counter-examples would be generated as evidence of the imprecision. Various proofs of property under-approximation can be found in [25].

## 8 Related Work

There is a wide body of literature on abstract interpretation. In our discussions of related work, we confine ourselves to work on automated abstraction facilities dedicated to constructing abstract models suitable for model-checking from program source code or closely related artifacts.

The closest work to ours is that of Gallardo, et. al. [12] on *alpha SPIN* – a tool for applying data abstraction to systems described in Promela (the input language of SPIN [15]). Alpha SPIN collects abstractions in libraries and transforms both Promela models and properties following a strategy that is similar to Bandera's. Alpha SPIN does not include automated facilities such as those found in Bandera for deriving sound abstractions, finding appropriate program components to abstract using dependency information, nor automated support for attaching abstractions via type-inference.

A closely related project that focuses on data abstraction of C program source code is the work on the abC tool by Dams, Hesse, and Holzmann [7]. Rather than providing a variety of abstractions in a library, abC focuses on *variable hiding* – a conceptually simple and practically very useful form of data abstraction in model checking which amounts to suppressing all information about a given set of variables. abC uses an integrated demand-driven pointer analysis to deal effectively with C pointers, and it has been implemented as an extension of GCC. Functionality that is similar to what abC provides can be achieved using Bandera's slicing facility (which detects and removes irrelevant variables) and Bandera's Point abstraction. However, since abC is dedicated to variable hiding, it provides a more precise form of abstraction attachment (e.g., compared to Bandera's type inference) for pointer types.

The Automated Software Engineering group at NASA Ames has developed a flexible explicit-state model-checker Java Pathfinder (JPF) that works directly on Java byte-code [3]. JPF includes a number of interesting search heuristics that are proving effective in software model-checking. The Ames group has also produced a simple predicate abstraction tool and a distributed version of the model-checking engine. Due to the difficulties associated with dynamically created data, the JPF predicate abstraction tool applies to integer variables only and does not include support for automated refinement. In collaboration with researchers at NASA Ames, JPF has been incorporated as a back-end checker for Bandera.

The Microsoft Research SLAM Project [1] focuses on checking sequential C code using well-engineered predicate abstraction and abstraction refinement tools. As discussed in Section 1, the strengths of the SLAM abstraction tool compared to Bandera are its automated refinement techniques which can significantly reduce the effort required by the user of the tool. The tradeoffs are that such techniques are computationally more expensive than the "compiled abstraction" approach taken by Bandera, and they have not been scaled up to work with computational patterns often used in Java where programs iterate over dynamically created data structure.

The BLAST Project [28], inspired by the SLAM work, combines the three-steps of abstract-check-refine into a single phase. Like SLAM, BLAST also works on sequential C code, and tradeoffs between the BLAST and Bandera abstraction approach are the same as those between SLAM and Bandera.

Gerard Holzmann's Feaver tool extracts Promela programs from annotated C programs for checking with SPIN [15]. Feaver performs abstraction by consulting a user built lookup-table that maps textual patterns appearing the the source code to textual patterns that form pieces of the abstract program. This tool has been used in several substantial production telecommunications applications.

Eran Yahav has developed a tool for checking safety properties of Java programs [30] built on top of Lev-Ami and Sagiv's three-valued logic analysis tool (TVLA) [21].

## 9 Conclusion

We have given an overview of some of the technical issues associated Bandera's tools for constructing abstract models of Java software. These tools are based on classical abstract interpretation techniques [6], and aim to provide users with simple but effective mechanisms for generating tractable models suitable for verification using widely-applied model-checking engines. Bandera's abstraction techniques have been used effectively in case studies with researchers at NASA Ames involving checking properties of avionics systems.

The strength of the Bandera abstraction tools include their simplicity, their ability to scale to large programs, and the ease with which they can be applied to systems with dynamic allocation of data and threads. We believe the main contribution of our work is the integration of different techniques into a coherent program abstraction toolset that has the ability to greatly extend the range of programs to which model checking techniques can be effectively applied.

Weaknesses of the tool include the lack of automated refinement techniques and the lack of sophisticated heap abstractions. As noted earlier, work on projects such as SLAM [1] and BLAST [28] have demonstrated the effectiveness of automated refinement techniques when applied to sequential programs that do not manipulate dynamically created data. Scaling these techniques up to a language like Java is an open problem that could a long way toward addressing the lack of automated refinement techniques in Bandera. Sophisticated heap abstraction capabilities have been developed in work on shape analysis (e.g., the TVLA project [21]), but automated abstraction selection and refinement techniques have not be developed yet. Combining and scaling up the automated predicate abstraction refinement techniques and heap abstractions with automated refinement is a research direction that we are pursuing.

## References

[1] T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2000.

[2] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10th International Conference on Computer Aided Verification*, June 1998.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[5] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002. To appear.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] G.J. Holzmann D. Dams, W. Hesse. Abstracting C with abC. In *Proc. 14th International Conference on Computer Aided Verification*, July 2002.

[8] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[9] C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, September 1999.

[10] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[11] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[12] M. M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. aSPIN: Extending SPIN with abstraction. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 254–258. Springer-Verlag, 2002.

[13] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[14] John Hatcliff. An introduction to partial evaluation using a simple flowchart language. In John Hatcliff, Peter Thiemann, and Torben Mogensen, editors, *Proceedings of the 1998 DIKU International Summer School on Partial Evaluation*, Tutorials in Computer Science, Copenhagen, Denmark, June 1998.
[15] G. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer-Verlag, 2000.
[16] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
[17] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 1999.
[18] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating Java for multiple model checkers: the bandera back end. Technical Report 2002-1, SAnToS Laboratory Technical Report Series, Kansas State University, Department of Computing and Information Sciences, 2002.
[19] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
[20] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
[21] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene-based static analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, 2000.
[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
[23] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
[24] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 1th International Conference on Automated Deduction (LNCS 607)*, 1992.
[25] Corina S. Păsăreanu. *Abstraction and Modular Reasoning for the Verification of Software*. PhD thesis, Kansas State University, 2001.
[26] Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software : A comparative case study. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 16 80)*, September 1999.
[27] Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, April 2001.
[28] Rupak Majumdar Thomas A. Henzinger, Ranjit Jhala and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002.
[29] Raja Valle-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, November 1999.
[30] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 27–40, January 2001.

# Types in Program Analysis

Thomas Jensen

IRISA/CNRS
Campus de Beaulieu
F-35042 France

**Abstract** This paper surveys type-based program analysis with an emphasis on the polyvariant program analyses that can be obtained using conjunctive (or intersection) types and parametric polymorphism. In particular, we show 1) how binding-time analysis and strictness analysis are variations of a common framework based on conjunctive types, 2) that the standard abstract interpretation used for strictness analysis is equivalent to the type-based analysis, and 3) that the conjunctive strictness analysis can be made more modular by blending conjunctions with parametric polymorphism.

## 1 Introduction

1989 saw the publication of two pieces of research that gave momentum to a field which soon became known as *type-based program analysis*. The first was due to Neil Jones and his co-workers at DIKU and was concerned with partial evaluation of the lambda calculus [Gom89, JGB$^+$90]. Partial evaluations of higher-order programs had proven a tricky issue because of the difficulty to determine automatically what expressions are *static* ie., only depend on the input provided (and so can be reduced by the partial evaluator). The problem was solved by using type-like annotations of expressions to indicate whether they are static or not, and to use typing-like rules for defining when a program is well annotated. Inferring such annotations can then be done using a type inference-like algorithm. At around the same time, Tsung-Min Kuo and Prateek Mishra published their strictness type analysis [KM89]. This work recast strictness analysis as a type inference problem in which the types are strictness properties (more details below) and where strictness analysis becomes a problem of type inference. In both cases, standard types from functional programming were taken and modified to represent particular properties required for a specific program transformation.

Type-based program analysis quickly gained in popularity—to the extent that a speaker at the 1990 Lisp & Functional Programming conference proclaimed the death of abstract interpretation (the predominant analysis technique at that time). While these rumours turned out to be exaggerated, there remained the question of how these two analysis techniques could be related and compared. It was soon observed that some of the early type-based analyses were less powerful than a standard abstract interpretation. Could this shortcoming be fixed? Another, related question was how to exploit the substantial body of knowledge developed in the setting of type theory and type systems for programming languages in order to improve type-based analyses.

With this paper we hope to convey some intuition about some of the basic ideas and concepts underlying type-based program analysis. Most of the paper is about a particular analysis, viz., strictness analysis. Our excuse for bothering the reader with such an unfashionable analysis is 1) that its basic property ("the value of an expression is undefined") is easy to understand, 2) that the theory is well developed[1] and 3) that the analysis is sufficiently complex to exhibit most of the interesting problems and issues to which we want to draw attention. We will discuss other, more recent analyses towards the end of the paper.

## 2 Type-Based Strictness Analysis

In this and the following sections we will present a type system for reasoning about the strictness properties of functions defined in a small functional language. The type system is strongly influenced by earlier work on *intersection types* for the untyped lambda calculus [CD78, Sal78]. Intersection types allow to specify several properties of a function (expressed as the conjunction of these properties) which makes them well suited for defining *polyvariant* program analyses *i.e.*, analyses where a function can be ascribed different properties depending on the context in which it is used. The usefulness of intersection types for program analysis was independently observed by Nick Benton and the author who both came up with the strictness type system presented in Figure 2 [Ben92, Jen91].

We recall that a function is said to be *strict* if the result of applying the function to an undefined argument is undefined. For each standard type $\sigma$ such as integers or booleans we introduce a non-standard type $\mathtt{f}^{\sigma}$ with the idea that $\mathtt{f}^{\sigma}$ is the type given to a $\sigma$-expression that diverges. Similarly, the property of a function of type $\sigma \rightarrow \tau$ being strict is expressed with the non-standard type $\mathtt{f}^{\sigma} \rightarrow \mathtt{f}^{\tau}$, which is the type of those functions that map undefined arguments to undefined results.

The definition of the logic is divided into two parts:

1. first we axiomatise a set of strictness properties,
2. then we define a set of inference rules for inferring such properties of programs.

The judgments of the logic have the form $\Gamma \vdash e : \varphi$ where $\Gamma$ is an environment that associates strictness types with the free variables of expression $e$ and $\varphi$ is a strictness type that can be deduced for $e$ in such an environment. For example,

[1] The present paper is a synthesis of work that have appeared in various conference proceedings and journals [Jen91, Jen95, Jen97, Jen98]. See also the author's *thèse d'habilitation* from University of Rennes 1, 1999

we can prove the judgment $[x : \mathtt{f}^{\mathsf{Int}}] \vdash x+17 : \mathtt{f}^{\mathsf{Int}}$ that intuitively states that "if $x$ diverges, then so does $x+17$". The general format of the rules is

$$\frac{\mathit{Hyp}_1, \ldots, \mathit{Hyp}_n}{\mathit{Concl}}$$

where the hypotheses $\mathit{Hyp}_1, \ldots, \mathit{Hyp}_n$ are either judgments or conditions guaranteeing that the environments provides a strictness types for all the free variables in the expression. The rule for analysing $\lambda$-abstractions

$$\textbf{Abs} \quad \frac{\Delta[x \mapsto \phi] \vdash e : \psi}{\Delta \vdash \lambda x.e : (\phi \rightarrow \psi)}$$

is similar to what is found in most type systems and has a similar reading. With this rule we can deduce that

$$\frac{[x : \mathtt{f}^{\mathsf{Int}}] \vdash x+17 : \mathtt{f}^{\mathsf{Int}}}{\vdash \lambda x.x+17 : \mathtt{f}^{\mathsf{Int}} \rightarrow \mathtt{f}^{\mathsf{Int}}},$$

proving the function $\lambda x.x+17$ strict.

## 3 Strictness Types

The programming language of study is the simply typed lambda calculus with integer and boolean constants and operations (+,∗,=), a conditional construct if and a fixed point operator fix. The set of types and terms of the language is given by the grammar:

$$\begin{aligned} \sigma &= \mathit{Int} \mid \mathit{Bool} \mid \sigma_1 \rightarrow \sigma_2 \\ e &= x^\sigma \mid c \mid e_1 \; \mathit{op} \; e_2 \mid \lambda x^\sigma.e \mid e_1 e_2 \\ &\mid \mathsf{fix}(\lambda f.e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3. \end{aligned}$$

where *Int* is the type of integers and *Bool* that of booleans.

For each type $\sigma$ we define a set of conjunctive program properties $L(\sigma)$ as follows:

$$\mathtt{t}, \mathtt{f} \in L(\sigma) \qquad \frac{\varphi_1 \in L(\sigma_1) \quad \varphi_2 \in L(\sigma_2)}{\varphi_1 \rightarrow \varphi_2 \in L(\sigma_1 \rightarrow \sigma_2)}$$

$$\frac{\varphi_i \in L(\sigma) \quad i \in I}{\bigwedge_{i \in I} \varphi_i \in L(\sigma)}$$

where I is a finite, non-empty set. We write $\varphi^\sigma$ to indicate that $\varphi \in L(\sigma)$ ie., that $\varphi$ is a property of expressions of type $\sigma$. Somewhat sloppily, we will say that $\sigma$ is the type of property $\varphi^\sigma$. The property $\mathtt{t}^\sigma$ is the trivial property that holds for all values of type $\sigma$. The interpretation of $\mathtt{f}$ is "the value of this expression is undefined". In strictness analysis, $\mathtt{f} \rightarrow \mathtt{f}$ means that the function

- $\dfrac{\varphi \leq \psi_i, \quad i \in I}{\varphi \leq \bigwedge_{i\in I} \psi_i}$
- $\bigwedge\limits_{i\in I} \varphi_i \leq \varphi_i, \quad \forall i \in I$
- $\varphi \leq \mathtt{t}$
- $\mathtt{t}^{\sigma\to\tau} = \mathtt{t}^{\sigma} \to \mathtt{t}^{\tau}$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}^{\sigma} \to \mathtt{f}^{\tau} = \mathtt{f}^{\sigma\to\tau}$
- $\bigwedge\limits_{i\in I} (\varphi \to \psi_i) = \varphi \to \bigwedge\limits_{i\in I} \psi_i$
- $\dfrac{\psi_1 \leq \varphi_1, \varphi_2 \leq \psi_2}{\varphi_1 \to \varphi_2 \leq \psi_1 \to \psi_2}$

**Fig. 1.** Axiomatisation of conjunctive strictness properties

maps an undefined argument to an undefined result *i.e.* that the function is strict. Similarly, $\mathtt{t} \to \mathtt{f}$ means that all values are mapped to undefined.

On each set $L(\sigma)$ of properties is defined an implication ordering $\leq$ that extends the basic implication $\mathtt{f}^{\sigma} \leq \mathtt{t}^{\sigma}$ to conjunctions and function properties. The intuition is that if $\varphi \leq \psi$ then the property $\varphi$ implies the property $\psi$, or, in other words, that $\psi$ is a safe, but less precise approximation of the property $\varphi$. In the strictness analysis, where the properties talk about the definedness of an expression, the inequality $\mathtt{f}^{\sigma} \leq \mathtt{t}^{\sigma}$ can be read as saying that the property "will definitely diverge" implies the property "may converge". We define $=$ to mean that two properties $\varphi$ and $\psi$ are provably equivalent *i.e.* that $\varphi \leq \psi$ and $\psi \leq \varphi$. The rules defining $\leq$ are given in Figure 1. Most of the rules should be straightforward to understand. The axiom $\mathtt{t}^{\sigma} \to \mathtt{f}^{\tau} = \mathtt{f}^{\sigma\to\tau}$ expresses that the analysis identifies the undefined value of a function type with the function that maps all arguments to the undefined value.[2]

## 4 The Strictness Logic

The set of conjunctive strictness formulae forms the basis of a logic for reasoning about strictness properties of functions. The logic is defined by the inference rules in Figure 2. A judgment is of form $\Delta \vdash e : \phi$ where $e$ is an expression, $\phi$ is a formula describing a property of $e$ and $\Delta$ is an environment associating program variables in the expression $e$ with properties.

Most of the rules have the same reading as they have in a standard type system. The rule **App** for example expresses that if a function $e_1$ maps arguments satisfying $\varphi_1$ to results satisfying $\varphi_2$ and if we have an argument satisfying a property $\psi$ which implies $\varphi_1$ the the result of the application $e_1 e_2$ satisfies $\varphi_2$.

[2] In semantic terms, the analysis does not distinguish between $\bot^{\sigma\to\tau}$ and $\lambda v.\bot$. Finer analyses that maintain this distinction are possible.

$$\textbf{Taut}\quad \frac{FV(e) \subseteq Dom(\Delta)}{\Delta \vdash e : \mathtt{t}} \qquad \textbf{Var}\quad \Delta[x \mapsto \phi] \vdash x : \phi$$

$$\textbf{Conj}\quad \frac{\Delta \vdash e : \psi_1 \quad \Delta \vdash e : \psi_2}{\Delta \vdash e : \psi_1 \wedge \psi_2}$$

$$\textbf{Weak}\quad \frac{\Delta \vdash e : \phi \quad \phi \leq \psi}{\Delta \vdash e : \psi}$$

$$\textbf{Abs}\quad \frac{\Delta[x \mapsto \phi] \vdash e : \psi}{\Delta \vdash \lambda x.e : (\phi \rightarrow \psi)}$$

$$\textbf{Fix}\quad \frac{\Delta[f : \psi_1] \vdash e : \psi_2 \quad \psi_2 \leq \psi_1}{\Delta \vdash \mathsf{fix}(\lambda f.e) : \psi_2}$$

$$\textbf{App}\quad \frac{\Delta \vdash e_1 : (\phi_1 \rightarrow \phi_2) \quad \Delta \vdash e_2 : \psi \quad \psi \leq \phi_1}{\Delta \vdash e_1 e_2 : \phi_2}$$

$$\textbf{Base-L}\quad \frac{\Delta \vdash e_1 : \varphi_1 \quad \Delta \vdash e_2 : \varphi_2}{\Delta \vdash e_1 \; op \; e_2 : \varphi_1}$$

$$\textbf{Base-R}\quad \frac{\Delta \vdash e_1 : \varphi_1 \quad \Delta \vdash e_2 : \varphi_2}{\Delta \vdash e_1 \; op \; e_2 : \varphi_2}$$

$$\textbf{If-1}\quad \frac{FV(e_1) \cup FV(e_2) \subseteq Dom(\Delta) \quad \Delta \vdash b : \mathtt{f}}{\Delta \vdash \; \mathsf{if}\; b \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 : \mathtt{f}}$$

$$\textbf{If-2}\quad \frac{FV(b) \subseteq Dom(\Delta) \quad \Delta \vdash e_1 : \phi \quad \Delta \vdash e_2 : \phi}{\Delta \vdash \; \mathsf{if}\; b \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 : \phi}$$

**Fig. 2.** Conjunctive strictness logic

The rule **Taut** allows to deduce the trivial property $\mathtt{t}^\sigma$ of all expressions of type $\sigma$, **Conj** enables us to prove of an expression that satisfies $\varphi_1$ and $\varphi_2$ that it satisfies the conjunction $\varphi_1 \wedge \varphi_2$ and the rule for **Weak** allows to infer that if we can prove that $e$ satisfies a property $\varphi$ then it also satisfies any property that is implied by $\varphi$.

The rule **Fix** is an induction rule for fixed points. Intuitively, if we can prove a property $\psi_2$ of (the defining expression of) a function $f$ by making the assumption $\psi_1$ on the recursive calls to $f$ and if the assumption is implied by the $\psi_2$ that was deduced, then $\psi_2$ is a valid property of the recursively defined function. The last four rules are strictness specific. The rules **Base-L** and **Base-**

**R** deal with the basic binary arithmetic and logical operators whose arguments are of integer or boolean type. The rules state that these operators are strict in each of their arguments: as soon as one is undefined the result is undefined.[3] The rule for the conditional similarly states that it is strict in the boolean condition and that the the result of a conditional is undefined if both of the branches are undefined. Note also that when several rules can be used to analyse an expression we are free to choose which one to apply. Thus, in the case of an if-expression, even if we can prove $\Delta \vdash b : \texttt{f}$ we can still apply the rule **If-2**, but with a potential loss of precision in the property we derive.

The following proof demonstrates the use of the proof system. The function being analysed stems from Kuo and Mishra's paper [KM89], where it was used to demonstrate the limitations of a type system without conjunctive types. It is defined as:

$$f\,x\,y\,z = \mathit{if}\,(z=0)\ \mathit{then}\ x+y\ \mathit{else}\ f\ y\ x\,(z-1)$$

Written in our language it looks like $\mathsf{fix}\lambda f.E$ where $E$ is defined by

$$E \equiv \lambda x.\lambda y.\lambda z.\ \mathsf{if}\ z=0\ \mathsf{then}\ x+y\ \mathsf{else}\ f\ y\ x\,(z-1).$$

We want to show that this function is strict in $x$ and $y$ separately, which in our logic is expressed by the formula

$$\texttt{f} \to \texttt{t} \to \texttt{t} \to \texttt{f} \wedge \texttt{t} \to \texttt{f} \to \texttt{t} \to \texttt{f}$$

For notational convenience denote this formula by $\psi$ and let similarly $\psi_1$ and $\psi_2$ denote $\texttt{f} \to \texttt{t} \to \texttt{t} \to \texttt{f}$ and $\texttt{t} \to \texttt{f} \to \texttt{t} \to \texttt{f}$, respectively. Furthermore, let $\Gamma$ be the environment $[f : \psi, x : \texttt{t}, y : \texttt{f}, z : \texttt{t}]$. We then get the following proof tree (where we have omitted the hypotheses concerning the free variables and the environment).

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ [f:\psi]\vdash E:\psi_1\end{array}\qquad \dfrac{\dfrac{\dfrac{\Gamma\vdash x:\texttt{t}\quad \Gamma\vdash y:\texttt{f}}{\Gamma\vdash x+y:\texttt{f}}\qquad \dfrac{\dfrac{\Gamma\vdash f:\psi}{\Gamma\vdash f:\psi_1}\quad \Gamma\vdash y:\texttt{f}\quad \Gamma\vdash x:\texttt{t}\quad \Gamma\vdash (z-1):\texttt{t}}{\Gamma\vdash f\ y\ x\,(z-1):\texttt{f}}}{\Gamma\vdash\ \mathsf{if}\ z=0\ \mathsf{then}\ x+y\ \mathsf{else}\ f\ y\ x\,(z-1)):\texttt{f}}\ \textbf{If-2}}{[f:\psi]\vdash E:\psi_2}\ \textbf{Abs}}{[f:\psi]\vdash \lambda x.\lambda y.\lambda z.\ \mathsf{if}\ z=0\ \mathsf{then}\ x+y\ \mathsf{else}\ f\ y\ x\,(z-1):\psi}\ \textbf{Conj}}{\vdash \lambda f.\lambda x.\lambda y.\lambda z.\ \mathsf{if}\ z=0\ \mathsf{then}\ x+y\ \mathsf{else}\ f\ y\ x\,(z-1):\psi\to\psi}\ \textbf{Abs}}{\vdash \mathsf{fix}(\lambda f.\lambda x.\lambda y.\lambda z.\ \mathsf{if}\ z=0\ \mathsf{then}\ x+y\ \mathsf{else}\ f\ y\ x\,(z-1)):\psi}\ \textbf{Fix}$$

The point of this example is that in order to prove $\psi_2$ of the body $E$ of function $f$ we need to assume property $\psi_1$ of $f$ in the environment, and vice versa. This

[3] These rules can be merged into one rule using conjunctions, as is done in the polymorphic strictness analysis to be defined in Section 9.

means that in order to prove the conjunction of $\psi_1$ and $\psi_2$, we need to assume that same conjunction of $f$. There is no conjunction-free type that is equivalent to $\psi = \psi_1 \wedge \psi_2$ so this deduction would not be possible in a strictness type system like the one in [KM89].

## 5 A Variation: Binding-Time Analysis

Type-based program analysis does not limit itself to strictness analysis—other analyses can be cast in this framework. In this section we show how the conjunctive strictness logic can be modified to give a binding time analysis. The binding-time analysis obtained is derived from and will correspond to the abstract interpretation for binding-time analysis presented by Hunt and Sands [HS91].

Binding-time analysis aims at determining which part of a program can be evaluated given only partial knowledge about the input to the program. This information is essential for a *partial evaluator* in order to determine which part of the program can be evaluated during partial evaluation [JGS93]. The two basic properties that can be assigned to an expression are *static*, meaning that the value of the expression can be determined, and *dynamic*, meaning that there might not be enough information to compute the value of the expression. The abstract interpretation by Hunt and Sands [HS91] uses as abstract domain the two element domain

$$\begin{tikzcd} D \arrow[d, dash] & \text{Dynamic} \\ S & \text{Static} \end{tikzcd}$$

to model base types Int and Bool, *i.e.*, the same domain as in strictness analysis but with different interpretation of the elements. The type constructor $\rightarrow$ is interpreted exactly as in strictness analysis (i.e., as the monotone function space) so the abstract domains are all order-isomorphic. The analysis by Hunt and Sands also includes list types which we do not treat here.

The binding time interpretation of terms is equal to the strictness interpretation except for the if-expression (called *cond* in [HS91]) and for constants. The abstract interpretation of if in the Hunt-Sands binding time analysis is

$$[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho = \begin{cases} D & \text{if } [\![b]\!]\rho = D_{\mathsf{Bool}} \\ [\![e_1]\!]\rho \sqcup [\![e_2]\!]\rho & \text{if } [\![b]\!]\rho = S_{\mathsf{Bool}} \end{cases}$$

which should be understood as follows. In order to determine the value of a conditional the branching condition $b$ must be known, *i.e.*, have abstract value $S_{\mathsf{Bool}}$. In that case we see how much we can determine of the values of the branches and combine these results by taking the least upper bound. Since the value of constants can be determined regardless of what information is available the interpretation of constants is $S$, *i.e.*, the smallest element in the domain **2** as opposed to strictness analysis where constants are interpreted by the top element. Apart from this the abstract interpretations are identical.

The similarity in the abstract interpretations is reflected in the program logics. The type `f` now represents the property of being static. The changes that have to be made to the strictness logic concern the conditional, the constants and the built-in functions. Conditionals are handled by replacing the rules **If-1** and **If-2** with the following rule

$$(\textbf{If-BTA})\quad \frac{\Delta \vdash b : \texttt{f} \quad \Delta \vdash e_1 : \varphi \quad \Delta \vdash e_2 : \varphi}{\Delta \vdash \ \textsf{if}\ b\ \textsf{then}\ e_1\ \textsf{else}\ e_2 : \varphi}$$

Constants are always static so the rule for constants is changed to

$$(\textbf{Const-BTA})\quad \vdash c : \texttt{f}.$$

Finally, a function like addition requires that all elements are static in order for the result to be static which means that the rules for the basic operators must be replaced by the rule

$$(\textbf{Base-BTA})\quad \frac{\Delta \vdash e_1 : \texttt{f} \quad \Delta \vdash e_2 : \texttt{f}}{\Delta \vdash e_1\ op\ e_2 : \texttt{f}}$$

## 6 Relationship to Abstract Interpretation

The question of the relative power of conjunctive strictness analysis and strictness analysis by abstract interpretation as defined by Geoff Burn, Chris Hankin and Samson Abramsky [BHA86] can be settled: they are equivalent. In this section we show how to formulate this statement precisely and give some hints as to how to prove it (for detailed proofs, see [Jen95]). Our original approach to setting up this equivalence was inspired by Samson Abramsky's Domain Logic [Abr91] in which (very roughly speaking) the denotational semantics of an expression is shown to be equivalent to the set of formulae provable of this expression in an axiomatic semantics. Due to the finite nature of our abstract domains, proofs are considerably simpler than in Domain Logic. The proof proceeds in two steps:

– first show how the set of strictness types modulo equivalence is in one-to-one correspondence with elements of the abstract domains of the abstract interpretation,
– then prove by structural induction that the type $\varphi$ corresponding to the abstract interpretation of an expression $e$ (via this one-to-one correspondence) can be inferred in the strictness logic, and that all other types $\psi$ that can be inferred of $e$ are implied by $\varphi$ ($\varphi \leq \psi$).

### 6.1 Lindenbaum Algebras

The implication ordering $\leq$ from the formal systems $(L_{ST}(\sigma), \leq)$ defined in the previous sections is only a preorder, since there are syntactically distinct formulae that imply each other. Logically equivalent formulae denote the same *property* so we can obtain the structure of properties axiomatized by the formal systems as the quotient structures

$$\mathcal{LA}(\sigma) = (L_{ST}(\sigma)/\!=, \leq/\!=),$$

called the Lindenbaum algebra of the formal system. The main result about this axiomatisation is that for each type $\sigma$ the Lindenbaum algebra is a finite lattice of properties and this lattice is exactly the lattice used in the strictness analysis for higher-order functional languages defined by Burn, Hankin and Abramsky [BHA86]. In this analysis, the types are interpreted as follows:

$$\begin{aligned}
[\![Int]\!] &= \mathbf{2}\\
[\![Bool]\!] &= \mathbf{2}\\
[\![\sigma_1 \rightarrow \sigma_2]\!] &= [\![\sigma_1]\!] \rightarrow_m [\![\sigma_2]\!]
\end{aligned}$$

where $\mathbf{2}$ is the two-point lattice with carrier $\{0, 1\}$ ordered by $0 \sqsubseteq 1$ and $D_1 \rightarrow_m D_2$ stands for the domain of monotone functions from $D_1$ to $D_2$.

**Theorem 1.**

$$L_{ST}(\sigma)/\!= \;\cong\; [\![\sigma]\!]$$

Concretely, this means that for each type we can exhibit a finite set $L_{NF}(\sigma)$ of "normal forms" such that to each formula $\varphi \in L_{ST}(\sigma)$ there exists a normal form $[\varphi] \in L_{NF}(\sigma)$ satisfying $\varphi = [\varphi]$. In particular, for the base types *Int* and *Bool*, the set of normal forms is the two-point lattice since all formulae are equivalent to either `t` or `f`.

## 6.2 Strictness Analysis by Abstract Interpretation

Strictness analysis by abstract interpretation interprets a term of type $\sigma$ as an element of the lattice $\sigma^{\#}$. The abstract interpretation of a term $f$ of type $\mathsf{Int} \rightarrow \mathsf{Int}$ is a function $[\![f]\!] : \mathbf{2} \rightarrow \mathbf{2}$. The function $[\![\_]\!]$ takes as arguments a term in $\Lambda_T$ and an environment mapping variables to values and returns as result the abstract value of the term. It is defined as shown in Figure 3.

The main theorem linking the abstract interpretation with the strictness logic states that all properties provable in the strictness logic are implied by the property found by the abstract interpretation, and, furthermore, that the property found by the abstract interpretation can be proved in the logic. For closed terms (i.e. terms with no free variables) the theorem can be stated as follows

**Theorem 2.** *Let $e$ be a term of type $\sigma$ and let $\Phi = \{\varphi \mid\vdash e : \varphi\}$ be the set of formulae provable of $e$ in the strictness logic. Let furthermore $\varphi_e \in L_{ST}(\sigma)$ denote the formula corresponding to the element $[\![e]\!]$ in $[\![\mathbf{2}]\!]$ via the isomorphism in Theorem 1. Then*

1. $\forall \varphi \in \Phi \,.\, \varphi_e \leq \varphi$
2. $\vdash e : \varphi_e$.

$$
\begin{array}{ll}
[\![x]\!]\rho & = \rho(x) \\
[\![c]\!]\rho & = 1 \\
[\![(e_1, e_2)]\!]\rho & = ([\![e_1]\!]\rho, [\![e_2]\!]\rho) \\
[\![\lambda x.e]\!]\rho & = \lambda v.[\![e]\!]\rho[x \mapsto v] \\
[\![e_1 e_2]\!]\rho & = [\![e_1]\!]\rho([\![e_2]\!]\rho) \\
[\![\mathsf{fix}\lambda x.e]\!]\rho & = \bigsqcup_{n\in\omega} ([\![\lambda x.e]\!]\rho)^n(\bot) \\
[\![\ \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho & = \begin{cases} 0 & \text{if } [\![b]\!]\rho = 0_{\mathsf{Bool}} \\ [\![e_1]\!]\rho \sqcup [\![e_2]\!]\rho & \text{otherwise} \end{cases}
\end{array}
$$

**Fig. 3.** Strictness analysis by abstract interpretation

## 7 Parametrically Polymorphic Program Properties

The conjunctive strictness analysis presented in the previous section has the virtue of clarifying the relationship between type-based analysis and abstract interpretation. However, it is a long shot away from an inference algorithm. A central problem here is to calculate a principal strictness type for an expression. It can be argued that expressions *do* have principal typings in the conjunctive strictness type system (obtained by tabulating the abstract interpretation of the expression using conjunctions) but there is nothing canonical about them and it is unclear how they can be inferred.

Another issue is that of modularity. Static analysis of program fragments with free variables usually either requires an environment associating a property with each free variable (as with the conjunctive strictness logic) or assumes that only trivial properties hold of the free variables. For programs structured into modules the latter approach is often the only one feasible and leads to poor analysis results.

Hints at a solution to these problems came from polymorphic type inference and in particular Trevor Jim's analysis of the notion of principal typing [Jim96]. The basic idea is to define analyses such that analysis of a program fragment results in a property *and* environment that together describes the relation between the properties of the free variables and the result of the program. One way of representing such relations is by adding *variables* to our language of conjunctive strictness types. Such variables can be instantiated with (ground) strictness types to obtain several different strictness types. The combination of variables (and universal quantification) with conjunctions yields compact representations of strictness properties: the strictness property of the addition function (whose minimal representation in the conjunctive system was $\mathtt{f} \to \mathtt{t} \to \mathtt{f} \wedge \mathtt{t} \to \mathtt{f} \to \mathtt{f}$ can now be expressed by

$$\forall \alpha_1 \alpha_2 . \alpha_1 \to \alpha_2 \to (\alpha_1 \wedge \alpha_2)$$

from which all strictness properties of addition can be obtained by instantiation and reduction. In addition, this kind of strictness properties can be combined with the strong induction principle of polymorphic recursion to obtain a precise analysis of recursively defined functions. Due to their similarity with the polymorphic types encountered in parametric polymorphism, we call this *polymorphic strictness properties.*

In the following we will show such a polymorphic strictness analysis. We will limit ourselves to defining the inference rules and show an example of their use. In particular, we will not give details about how the addition of polymorphism together with the restriction to *ranked* strictness types leads to an inference algorithm—the interested reader is referred to [Jen98].

## 8 Polymorphic and Conditional Program Properties

We extend the language of properties in two ways: for each type we add property variables and define a notion of property scheme and we introduce a new property constructor, ?, for building conditional properties (explained below). These features allow us to express program properties succinctly and leads to a syntax-directed inference system for inferring such properties. Both features are inspired by similar constructs for typing of programs [Mil78, AWL94].

The two base types *Int* and *Bool* have isomorphic sets of properties so it is convenient to identify these two sets. We do this by replacing *Int* and *Bool* with a common base type $\flat$ in the definition of the set of properties. A property of the set $L(\flat)$ can describe both an integer and a boolean value. Formally, we redefine the set of types to be

$$\sigma = \flat \mid \sigma_1 \to \sigma_2$$

Assume that for each type $\sigma$ we are given a set of property variables, ranged over by $\alpha^\sigma$. We extend the set of properties by defining the type-indexed collection of properties $L(\sigma)$ as follows:

**Definition 1.** *For each type $\sigma$ define $L(\sigma)$ to be the smallest set satisfying*

$$\mathtt{t}, \mathtt{f}, \alpha^\sigma \in L(\sigma) \qquad \frac{\varphi_1 \in L(\sigma_1) \quad \varphi_2 \in L(\sigma_2)}{\varphi_1 \to \varphi_2 \in L(\sigma_1 \to \sigma_2)}$$

$$\frac{\varphi_i \in L(\sigma) \quad i \in I}{\bigwedge_{i\in I} \varphi_i \in L(\sigma)} \qquad \frac{\varphi \in L(\sigma)}{\forall \alpha.\varphi \in L(\sigma)}$$

$$\frac{\varphi_2 \in L(\flat) \quad \varphi_1 \in L(\sigma)}{\varphi_1 ? \varphi_2 \in L(\sigma)}.$$

The variables $\alpha^\sigma$ range over the properties of type $\sigma$ (we shall frequently leave out the type annotation on these variables when we think it is possible). As an example, we have that the identity function on integers satisfies $\forall \alpha^\flat.\alpha^\flat \to \alpha^\flat$ from which we get as an instance that the identity satisfies $\mathtt{f} \to \mathtt{f}$ *i.e.* that it is strict. As a somewhat more interesting example that mixes polymorphism and conjunctions we have the addition operator that is bi-strict: as soon as one argument is undefined the result is undefined. This is expressed by the property

$$\forall\alpha_1^\flat\alpha_2^\flat.\alpha_1^\flat \to \alpha_2^\flat \to \alpha_1^\flat \wedge \alpha_2^\flat.$$

The notions of free and bound variables of a property $\varphi$ (written $FV(\varphi)$ and $BV(\varphi)$) and renaming of bound variables carry over to properties in the standard way. We do not distinguish between properties that only differ in the choice of bound variables. Substitution of $\psi$ for all free occurrences of variable $\alpha$ in $\varphi$ is written $\varphi[\psi/\alpha]$.

Conditional properties have the form $\varphi_1?\varphi_2$ where $\varphi_2$ is a property of base type. They were used for dataflow analysis by Reynolds [Rey69] and for soft (or partial) typing of functional programs by Aiken *et al.* [AWL94]. If $\varphi_2 = \mathtt{f}^\flat$ then $\varphi_1?\varphi_2$ is equal to $\mathtt{f}^\sigma$ and if $\varphi_2 = \mathtt{t}^\flat$ then $\varphi_1?\varphi_2$ is equal to $\varphi_1$ where $\varphi \in L(\sigma)$. This functionality could conveniently have been expressed as the conjunction of $\varphi_1$ and $\varphi_2$, were it not for the fact that $\varphi_1$ and $\varphi_2$ have different types. We need this "conjunction across types" to state that the result of an if-statement of type $\sigma$ is undefined (satisfies $\mathtt{f}^\sigma$) if the boolean condition in the expression is undefined (satisfies $\mathtt{f}^\flat$).

## 9 Polymorphic Strictness Analysis

The polymorphic strictness analysis defined in this section operates on judgements of the form $\Gamma \vdash e : \varphi$ where the environment $\Gamma$ is a finite set of assumptions of the form $x_i^\sigma : \varphi_i^\sigma$ associating properties to program variables. The set of assumptions $\Gamma$ will always contain exactly one assumption for every free program variable in $e$. In order to maintain this invariant, we have to define a special operator $+$ on assumption sets that allow to combine several assumption sets into one. More precisely, we define $\Gamma_1 + \Gamma_2$ to be $\Gamma_1 \cup \Gamma_2$ except that if $x : \varphi_1 \in \Gamma_1$ and $x : \varphi_2 \in \Gamma_2$ then these are replaced by $x : \varphi_1 \wedge \varphi_2$ in $\Gamma_1 + \Gamma_2$.

A (property) variable $\alpha^\sigma$ is said to be free in $\Gamma$ if it is free in one or more of the $\varphi_i^\sigma$. The set of free variables in $\Gamma$ is denoted by $FV(\Gamma)$. For given environment $\Gamma$ and property $\varphi$ we define the property $Gen(\Gamma, \varphi) \equiv \forall\overline{\alpha}.\varphi$ obtained by quantifying $\varphi$ over all variables free in $\varphi$ and not free in $\Gamma$. Polymorphic type systems usually contain rules for introducing and eliminating quantified types, *viz.*,

$$Gen\frac{\Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau} \qquad Int\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\tau'/\alpha]}.$$

However, these rules breaks the syntax-directed property of the inference system and it is therefore common to modify the other rules in the system to incorporate *Gen* and *Int*. We adhere to this latter approach. Our set of inference rules will not contain *Gen* and *Int* and thus does not assign quantified types to expressions. Quantified types appear in the rule for recursion and in the definition of the implication ordering $\leq$ on strictness types that has to be extended with the rules

$$\frac{\psi \in L(\sigma)}{\forall \alpha^{\sigma}.\varphi \leq \varphi[\psi/\alpha^{\sigma}]} \qquad \text{and} \qquad \frac{\varphi \leq \psi \quad \alpha \notin FV(\varphi)}{\varphi \leq \forall \alpha.\psi}.$$

In order to deal with recursive definitions it is important that the rule for fix allows that different occurrences of the recursive function variable $f$ in fix $\lambda f.e$ can be ascribed different properties. In the conjunctive system this was naturally achieved by combining the different properties using conjunctions and then select the relevant property at each occurrence using the weakening rule. However, weakening breaks the syntax-directedness of a system so it must be eliminated and built into the other rules. One way of achieving this is to use the principle of polymorphic recursion proposed by Mycroft [Myc84] for typing ML programs. This principle, that was first exploited in type-based program analysis by Dirk Dussart, Fritz Henglein and Christian Mossin [DHM95], can be expressed by the inference rule

$$\frac{\Gamma \cup \{f : \bigwedge_{i\in I} \varphi_i\} \vdash e : \varphi \quad \forall i \in I : Gen(\Gamma, \varphi) \leq \varphi_i}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}.$$

Informally, it states that each hypothesis $\varphi_i$ made on $f$ to analyse the body $e$ must be implied by the generalisation of the property deduced for $e$. Notice that we can generalise over variables that are free in $\varphi_i$ as long as they are not free in $\Gamma$. For example, having proved that the body of a function satisfies $\alpha_2 \to \alpha_1 \to (\alpha_1 \wedge \alpha_2)$ under the assumption $\{f : \alpha_1 \to \alpha_2 \to (\alpha_1 \wedge \alpha_2)\}$, we can generalise over all free variables $(\alpha_1, \alpha_2)$ to obtain $\forall \alpha_1\alpha_2.\alpha_2 \to \alpha_1 \to (\alpha_1 \wedge \alpha_2)$ of which the assumption on $f$ is an instance. Another point to notice is that the rule for application has been rewritten so that the rule only applies when the property of the function has a particular form. The rule furthermore requires us to prove as many properties (the $\psi_i$) of the actual argument to the function as there are hypotheses (the $\varphi_i$) about the argument in the property the function. This modification facilitates the definition of an inference algorithm but we will not go further into this issue here. The inference rules defining the polymorphic strictness logic are given in Figure 4. We give an example of a deduction in the system in Figure 5.

As opposed to the conjunctive strictness logic in Figure 2, we now have that the variables in the environment are exactly the free variables in $e$; hence the need for two rules for $\lambda$-abstraction and fix. This eliminates the arbitrariness in the conjunctive system that allowed to add irrelevant properties to the environment. When the rule for basic operators *op* is instantiated to e.g. $=$ we take advantage of having a common set of properties for integer and boolean values since we otherwise would have to convert the conjunction $\varphi_1 \wedge \varphi_2$ of integer properties to the equivalent property for booleans.

The example in Figure 5 illustrates the analysis by showing the deduction of property

$$\alpha_1 \to \alpha_2 \to \alpha_3 \to ((\alpha_1 \wedge \alpha_2)?\alpha_3)$$

$$\{x : \varphi\} \vdash x : \varphi \qquad \emptyset \vdash c : \mathtt{t}$$

$$\frac{\Gamma \vdash e : \varphi \quad x \notin FV(e)}{\Gamma \vdash \lambda x.e : \mathtt{t} \rightarrow \varphi} \qquad \frac{\Gamma \cup \{x : \varphi\} \vdash e : \psi}{\Gamma \vdash \lambda x.e : \varphi \rightarrow \psi}$$

$$\frac{\Gamma \vdash e_1 : ((\bigwedge_{i\in I} \varphi_i) \rightarrow \varphi)?\varphi_b \quad \Gamma_i \vdash e_2 : \psi_i \quad \forall i \in I : \psi_i \leq \varphi_i}{\Gamma + \sum_{i\in I} \Gamma_i \vdash e_1 e_2 : \varphi?\varphi_b}$$

$$\frac{\Gamma_i \vdash e_i : \varphi_i \;, i = 1,2,3 \quad \varphi_2 \leq \varphi \quad \varphi_3 \leq \varphi}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \varphi?\varphi_1}$$

$$\frac{\Gamma \cup \{f : \bigwedge_{i\in I} \varphi_i\} \vdash e : \varphi \quad \forall i \in I : Gen(\Gamma, \varphi) \leq \varphi_i}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}$$

$$\frac{\Gamma_1 \vdash e_1 : \varphi_1 \quad \Gamma_2 \vdash e_2 : \varphi_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ op\ e_2 : \varphi_1 \wedge \varphi_2} \qquad \frac{\Gamma \vdash e : \varphi \quad f \notin FV(e)}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}$$

**Fig. 4.** Polymorphic strictness analysis.

for the function $f$, recursively defined by

$$\begin{aligned}\mathsf{fix}\ \lambda f.\lambda xyz.\ \mathsf{if}\ z = 0\ \mathsf{then}\ x - y\ \mathsf{else}&\\ \mathsf{if}\ z < 0\ \mathsf{then}\ f\ x\ y\ (1 - z)&\\ \mathsf{else}\ f\ y\ x\ (z - 1).&\end{aligned}$$

The function is an elaboration of an example due to Kuo and Mishra [KM89]. The function $f$ is strict in each of the arguments $x$ and $y$ separately *i.e.*, it satisfies the property $\mathtt{f} \rightarrow \mathtt{t} \rightarrow \mathtt{t} \rightarrow \mathtt{f} \ \wedge \ \mathtt{t} \rightarrow \mathtt{f} \rightarrow \mathtt{t} \rightarrow \mathtt{f}$. In a purely conjunctive logic, two sub-deductions with the same structure are needed to prove that the function satisfies each of the conjuncts, see [Jen95]. These sub-deductions are combined into one in our analysis by using quantified property variables and polymorphic recursion; more precisely, we prove that $f$ satisfies

$$\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow ((\alpha_1 \wedge \alpha_2)?\alpha_3).$$

The two recursive calls to $f$ will need two different properties of $f$ that both are implied by the generalisation of the one above *viz.*,

$$\alpha_2 \rightarrow \alpha_1 \rightarrow \alpha_3 \rightarrow (\alpha_2 \wedge \alpha_1)$$

and

$$\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow (\alpha_1 \wedge \alpha_2).$$

Analysis of the function

$$\mathsf{fix}\ \lambda f.\lambda xyz.\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \ \mathsf{if}\ z<0\ \mathsf{then}\ f\ x\ y\ (1-z)\ \mathsf{else}\ f\ y\ x\ (z-1).$$

Let

$\Gamma_1=\{x:\alpha_1\}, \Gamma_2=\{y:\alpha_2\}, \Gamma_3=\{z:\alpha_3\}$,
$\Gamma_4=\{f:\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2)\}, \Gamma_5=\{f:\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)\}$ and
$\Gamma=\sum_{i=1}^{5}\Gamma_i=\{x:\alpha_1, y:\alpha_2, z:\alpha_3,$
$\quad f:(\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2))\wedge(\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2))\}$.

The analysis of the inner if-expression consists of combining the three deductions:

$$\dfrac{\Gamma_5\vdash f:\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)\quad \Gamma_1\vdash x:\alpha_1\quad \Gamma_2\vdash y:\alpha_2\quad \dfrac{\emptyset\vdash 1:\mathtt{t}\quad \Gamma_3\vdash z:\alpha_3}{\Gamma_3\vdash(1-z):\alpha_3}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_5\vdash f\,x\,y\,(1-z):\alpha_1\wedge\alpha_2},$$

$$\dfrac{\Gamma_3\vdash z:\alpha_3\quad \emptyset\vdash 0:\mathtt{t}}{\Gamma_3\vdash z<0:\alpha_3},$$

$$\dfrac{\Gamma_4\vdash f:\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2)\quad \Gamma_2\vdash y:\alpha_2\quad \Gamma_1\vdash x:\alpha_1\quad \begin{array}{c}\vdots\\ \Gamma_3\vdash(z-1):\alpha_3\end{array}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_4\vdash f\,y\,x\,(z-1):\alpha_1\wedge\alpha_2}$$

in order to deduce $\Gamma\vdash\ \mathsf{if}\ z<0\ \mathsf{then}\ f\ x\ y\ (1-z)\ \mathsf{else}\ f\ y\ x\ (z-1):(\alpha_1\wedge\alpha_2)?\alpha_3$. The analysis then proceeds as follows, writing $e_{\mathit{ff}}$ as an abbreviation for the expression $\mathsf{if}\ z<0\ \mathsf{then}\ f\ x\ y\ (1-z)\ \mathsf{else}\ f\ y\ x\ (z-1)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_3\vdash z:\alpha_3\quad \emptyset\vdash 0:\mathtt{t}}{\Gamma_3\vdash z=0:\alpha_3}\quad \dfrac{\Gamma_1\vdash x:\alpha_1\quad \Gamma_2\vdash y:\alpha_2}{\Gamma_1+\Gamma_2\vdash x-y:\alpha_1\wedge\alpha_2}\quad \begin{array}{c}\vdots\\ \Gamma\vdash e_{\mathit{ff}}:(\alpha_1\wedge\alpha_2)?\alpha_3\end{array}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_4+\Gamma_5\vdash\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ e_{\mathit{ff}}:(\alpha_1\wedge\alpha_2)?\alpha_3}}{\Gamma_1+\Gamma_2+\Gamma_4+\Gamma_5\vdash\lambda z.\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ e_{\mathit{ff}}:\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\Gamma_1+\Gamma_4+\Gamma_5\vdash\lambda yz.\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ e_{\mathit{ff}}:\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\Gamma_4+\Gamma_5\vdash\lambda xyz.\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ e_{\mathit{ff}}:\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\vdash\ \mathsf{fix}\ \lambda f.\lambda xyz.\ \mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ e_{\mathit{ff}}:\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{}$$

During the deduction we have used that $\alpha_3\wedge\mathtt{t}=\alpha_3$ and that $(\alpha_1\wedge\alpha_2)?\alpha_3\le\alpha_1\wedge\alpha_2$. The last step of the deduction used that by instantiating $\alpha_1$ with $\alpha_2$ and $\alpha_2$ with $\alpha_1$ we get

$$\begin{aligned}\mathit{Gen}(\emptyset,\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)?\alpha_3)&=\forall\alpha_1\alpha_2\alpha_3.\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)\\&\le\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2).\end{aligned}$$

In a similar fashion, we obtain the other conjunct of the hypothesis on $f$:

$$\begin{aligned}\mathit{Gen}(\emptyset,\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)?\alpha_3)&=\forall\alpha_1\alpha_2\alpha_3.\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)\\&\le\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2).\end{aligned}$$

**Fig. 5.** An example deduction

*Soundness.* The soundness of this system is first proved with respect to the conjunctive strictness logic presented in Figure 2, ie., all facts deduced with the polymorphic strictness logic can be shown to be deducible in the conjunctive logic as well. Since the conjunctive logic is known to be sound, soundness in general follows. We will not set up the technical machinery needed to state this formally; the interested reader is referred to [Jen98].

## 10 Variants of Type-Based Analysis

The story told here is only a small fragment of what has been achieved in type-based program analysis. Here, we very briefly list a number of related issues that we haven't treated.

**Union types** are natural addenda to a conjunctive type system [BDC91]. Union types allow to deduce a set of formulae of an expression with the intention that the expression will satisfy (at least) one of them. The advantage of these types become apparent with the if-construct where we can return the set consisting of the types derived for the then- and for the else-branch This is in contrast to the conjunctive types where we had to return one type that could cover (ie that is weaker) than the types of the branches. See [Jen97] for a longer discussion and for other references. The same reference can also provide a starting point for an exploration of how to deal with *data structures* such as lists and trees whose unboundedness must be tamed in order to provide useful analyses.

**Polymorphic invariance** In this paper we have focused on a simply-typed functional language without polymorphism. A pertinent question is how the uniformity guaranteed by parametric polymorphism can be exploited when analysing polymorphic functions, that otherwise would require analysing an infinite number of monomorphic instances. Abramsky identified the notion of polymorphic invariance which intuitively means that a property holds of all (monomorphic) instances of a polymorphic function if and only if it holds of one of the instances. See Benton [Ben92] for a direct prof of the invariance of the conjunctive strictness logic.

**Other analyses** A variety of other analyses have been cast as type system or the related constraint-based analyses. Control flow analysis is one such example. Control flow analysis aims at determining the order in which functions call each other during evaluation. The problem was studied early on by Neil Jones [Jon81] and was then picked up by Olin Shivers [Shi91] and Peter Sestoft [Ses88]. More recent accounts include the work of Jens Palsberg [Pal95], Christian Mossin [Mos97], Anindya Banerjee [Ban97]. The region analysis of Mads Tofte and Jean-Pierre Talpin [TT94] is aimed at imposing a stack-like memory-management discipline in higher-order functional programs. It is an example of a type-and-effect system where information about the operational behaviour of functions are included in the types. Another such example (this time for a concurrent language) is the type and effect system of Torben Amtoft, Flemming Nielson and Hanne Riis Nielson for analysing communication behaviour [ANN99]. More recently, type systems have been used in security analysis of software to detect illicit information flow, see the work of Dennis Volpano, Geoffrey Smith and Cynthia Irvine [VSI96] and by Fran cois Pottier and Sylvain Conchon [PC00].

**Types from abstract interpretation** We have focused on the equivalence between conjunctive type systems and the abstract interpretation version of strictness analysis. Other analysis formats have been investigated by Patrick and Radhia Cousot who addressed the problem of relating set constraint-based analyses to abstract interpretation [CC95]. Finally, the whole notion of type systems can be studied in the setting of abstract interpretation as demonstrated by Patrick Cousot in [Cou97] in which a considerable number of type systems are given a uniform presentation and compared using the theory of abstract interpretation.

**Inference algorithms** for implementing these analyses are of course an essential and difficult problem that must be solved for these analyses to come to practical use. The literature on this issue itself is vast and deserves its own detailed treatment so we'll say nothing about this in this survey.

**An essential reference** for the domain is the web page maintained by Jens Palsberg (`http://www.cs.purdue.edu/homes/palsberg/tba`) that contains references to a large part of the work on type-based program analysis.

*Acknowledgments:* Many thanks are due to Dave Schmidt for providing extensive comments on an earlier version of this paper.

## References

[Abr91] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[ANN99] T. Amtoft, F. Nielson, and H. Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency.* IC Press, 1999.

[AWL94] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1994.

[Ban97] A. Banerjee. A modular, polyvariant and type-based closure analysis. In *Proc. ACM Int. Conf. on Functional Programming (ICFP'97)*, pages 1–10. ACM Press, 1997.

[BDC91] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Software.* Springer LNCS 526, 1991.

[Ben92] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proc. of the 2. International Symposium on Logical Foundations of Computer Science, Tver, Russia*, LNCS vol. 620. Springer, 1992.

[BHA86] G. Burn, C. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set constraint-based program analysis by abstract interpretation. In *Proc. of the ACM*

*Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 170–181. ACM Press, 1995.

[CD78] M. Coppo and M. Dezani–Ciancaglini. A new type-assignment for lambda terms. *Archiv für Mathematische Logik*, 19:139–156, 1978.

[Cou97] P. Cousot. Types as abstract interpretations. In *Proc. of 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, 1997.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proc. 2nd Int. Static Analysis Symposium (SAS), Glasgow, Scotland*, LNCS vol. 983. Springer, 1995.

[Gom89] C. Gomard. Higher order partial evaluation - hope for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark., September 1989.

[HS91] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM Symposium on Partial Evaluation and Semantics–Based Program Manipulation*. ACM Press,, New York, 1991.

[Jen91] T. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523, pages 352–366. Springer, 1991.

[Jen95] T. Jensen. Conjunctive type systems and abstract interpretation of higher-order functional programs. *Journal of Logic and Computation*, 5(4):397–421, 1995.

[Jen97] T. Jensen. Disjunctive program analysis for algebraic data types. *ACM Transactions on Programming Languages and Systems*, 19(5):752–804, 1997.

[Jen98] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, pages 209–221. ACM Press, 1998.

[JGB$^+$90] N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *Proc. of 3. International Conference on Computer Languages*. IEEE Computer Society, 1990.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Series in Computer Science. Prentice-Hall International, 1993.

[Jim96] T. Jim. What are principal typings and what are they good for? In *Proc. of 23rd Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 1996.

[Jon81] N. Jones. Flow analysis of lambda expressions. In S. Even and O. Kariv, editors, *Proc. of 8th International Colloquium on Automata, Languages and Programming*, pages 114–128. Springer LNCS vol. 115, 1981.

[KM89] T.-M. Kuo and P. Mishra. Strictness analysis : A new perspective based on type inference. In *Proc. 4th. Int. Conf. on Functional Programming and Computer Architecture*. ACM Press, 1989.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mos97] C. Mossin. Exact flow analysis. In P. v. Hentenryck, editor, *Proc. of 4th Int. Static Analysis Symposium (SAS'97)*, pages 250–264. Springer LNCS vol. 1302, 1997.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. of Int. Symposium on Programming, Toulouse, France*, pages 217–239. Springer LNCS vol. 167, 1984.

[Pal95] J. Palsberg. Closure analysis in constraint form. *ACM Trans. on Programming Languages and Systems*, 17(1):47–62, 1995.

[PC00] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, 2000.

[Rey69] J. C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68, 1969.

[Sal78] P. Sallé. Une extension de la théorie des types en $\lambda$-calculus. In *Proc. of Int. Coll. on Automata, Language and Computation (ICALP'78)*, pages 398–410. Springer LNCS vol. 62, 1978.

[Ses88] P. Sestoft. Replacing function parameters by global variables. Master's thesis, Univ. of Copenhagen, 1988.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University, 1991.

[TT94] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 188–201, 1994.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

# Flow Logic: A Multi-paradigmatic Approach to Static Analysis

Hanne Riis Nielson and Flemming Nielson

Informatics and Mathematical Modelling
Technical University of Denmark
{riis,nielson}@@imm.dtu.dk

**Abstract.** Flow logic is an approach to static analysis that separates the *specification* of when an analysis estimate is acceptable for a program from the actual *computation* of the analysis information. It allows one not only to combine a variety of programming paradigms but also to link up with state-of-the-art developments in classical approaches to static analysis, in particular data flow analysis, constraint-based analysis and abstract interpretation. This paper gives a tutorial on flow logic and explains the underlying methodology; the multi-paradigmatic approach is illustrated by a number of examples including functional, imperative, object-oriented and concurrent constructs.

## 1 Introduction

Computer systems often combine different programming paradigms and to support the software development phase it is desirable to have techniques and tools available for reasoning about and validating the behaviour of multi-paradigmatic programs. One such set of techniques and tools are based on static analysis. Traditionally, static analysis has been developed and used in the construction of optimising compilers but recent developments show promise of a much wider application area that includes validating that programs adhere to specific protocols and that certain performance guarantees can be ensured.

Flow logic is a formalism for static analysis that is based on logical systems. It focusses on specifying what it means for an analysis estimate to be acceptable for a program. The specification can be given at different levels of abstraction depending on the interest in implementation details but there will always be a clear distinction between specifying *what* is an acceptable analysis estimate and *how* to compute it. Flow logic is not a "brand new" technique for static analysis; rather it is firmly rooted upon existing technology and insights, in particular from the classical areas of data flow analysis [9], constraint-based analysis [10] and abstract interpretation [7]. In contrast to the traditional presentations of these approaches, flow logic applies to programs expressed in mixed programming paradigms and allows the existing techniques and insights to be integrated thereby giving a wider perspective on their applicability.

**Flow logic and operational semantics.** Flow logic shares many of its properties with structural operational semantics [23]. One reason for the success of structural operational semantics is that it is applicable to a wide range of programming paradigms. Another is that within the same formalism, definitions can be given at different levels of abstraction reflecting the interest in implementation details; subsequently one definition can be refined into another by a transformation process or, alternatively, the relationship between different definitions can be established using various proof techniques.

The ability to specify the analysis at different abstraction levels has lead to the identification of *different styles of flow logic* just as there are different styles of operational semantics – as for example small-step versus big-step transitions, environments versus substitutions, and evaluation contexts versus explicit rules for evaluation in context. In Section 2, which takes the form of a short tutorial, we introduce two sets of styles that so far have emerged for flow logic: one being *abstract* versus *compositional* and the other being *succinct* versus *verbose*.

**Flow logic and type systems.** The overall methodology of flow logic has much in common with that of type systems [11]. As already mentioned we have a clear distinction between the judgements *specifying* the acceptability of an analysis estimate and the algorithms *computing* the information. Also, we are interested in the following properties that all have counterparts in type systems:

– *Semantic correctness*: this property ensures that the analysis estimates always "err on the safe side" and relates to the slogan "well-typed programs do not go wrong" (see Subsection 3.2).
– *Existence of best analysis estimates*: this property ensures that there is a most precise analysis estimate for all programs; the analogy is here to the existence of principal types in type systems (see Subsection 3.3).
– *Efficient implementations*: these typically take the form of constraint generation algorithms working in conjunction with constraint solving algorithms; as for type systems the correctness is established via syntactic soundness and completeness results (see Subsection 3.4).

However, there are also important differences between flow logic and type systems. One is that the clauses defining the judgements of a flow logic in principle have to be interpreted *co-inductively* rather than inductively: an analysis estimate for a program is acceptable if it does *not* violate any of the conditions imposed by the specification. It turns out that for the compositional specifications the co-inductive and inductive interpretations coincide whereas for the abstract specifications additional care needs to be taken to ensure well-definedness (see Subsection 3.1).

Another important difference between flow logic and type systems is that the former allows the exploitation of classical techniques for static analysis; we shall not go further into this in the present paper.

**Flow logic for multiple paradigms.** Flow logic was originally developed for a pure *functional* language (resembling the core of Standard ML) [14]; then it was further developed to handle *imperative* constructs (as in Standard ML) [17,21] and *concurrent* constructs (as in Concurrent ML) [8]. Later the approach has been used for various calculi of computation including *object-oriented* calculi (as the imperative object calculus) [15], the pi-calculus (modeling communication in a pure form) [3], the ambient calculus (generalising the notion of mobility found in Java) [19] and the spi-calculus (modeling cryptographic primitives) [4].

We conclude in Section 4 by illustrating the flexibility of flow logic by giving a number of examples for different programming paradigms thereby opening up for the exploitation of static analysis techniques in a software development process integrating several paradigms.

## 2 A Tutorial on Flow Logic

A flow logic specification consists of a set of clauses defining a judgement expressing the acceptability of an analysis estimate for a program fragment. In principle, the specification has to be interpreted *co-inductively*: an analysis estimate can only be rejected if it does not fulfill the conditions expressed by the definition. Two sets of criteria have emerged for classifying flow logic specifications:

- *abstract* versus *compositional*, and
- *succinct* versus *verbose*.

A *compositional* specification is syntax-directed whereas an *abstract* specification is not. In general an abstract specification will be closer to the standard semantics and a compositional specification will be closer to an implementation. In the earlier stages of designing an analysis it may be useful to work with an abstract specification as it basically relies on an understanding of the semantics of the language and one can therefore concentrate on designing the abstract domains used by the analysis; this is particularly pertinent for programming languages allowing programs as data objects. Thinking ahead to the implementation stage one will often transform the specification into a compositional flow logic. Another reason for working with compositional specifications is that the associated proofs tend to be simpler as the coinductive and inductive interpretation of the defining clauses coincide.

A *verbose* specification reports all of the internal flow information as is normally the case for data flow analysis and constraint-based analysis; technically, this is achieved by having appropriate data structures (often called caches) holding the required analysis information. A specification that is not verbose is called *succinct* and it will often focus on the top-level parts of the analysis estimate in the manner known from type systems.

To give a feeling for these different styles we shall in this section present all four combinations. To keep things manageable we shall focus on a single programming paradigm; in Section 4 we extend the ideas to other paradigms.

**The $\lambda$-calculus.** The example language will be the $\lambda$-calculus with expressions $e \in \mathbf{Exp}$ given by

$$e ::= c \mid x \mid \lambda x_0.e_0 \mid e_1\, e_2$$

where $c \in \mathbf{Const}$ and $x \in \mathbf{Var}$ are unspecified sets of first-order constants and variables.

We choose an environment-based call-by-value big-step operational semantics [6]. The values $v \in \mathbf{Val}$ are either constants $c$ or closures of the form $\langle \lambda x_0.e_0, \rho_0 \rangle$

where $\rho_0 \in \mathbf{Env} = \mathbf{Var} \rightarrow_{\mathsf{fin}} \mathbf{Val}$ is an environment mapping (a finite set of) variables to values. The semantics is formalised by a transition relation $\rho \vdash e \longrightarrow v$ meaning that evaluating $e$ in the environment $\rho$ gives rise to the value $v$; it is defined by the following axioms and rules:

$$\begin{array}{ll} \rho \vdash c \longrightarrow c & \\ \rho \vdash x \longrightarrow \rho(x) & \dfrac{\rho \vdash e_1 \longrightarrow \langle \lambda x_0.e_0, \rho_0\rangle \quad \rho \vdash e_2 \longrightarrow v_2 \quad \rho_0[x_0 \mapsto v_2] \vdash e_0 \longrightarrow v_0}{\rho \vdash e_1\, e_2 \longrightarrow v_0} \\ \rho \vdash \lambda x_0.e_0 \longrightarrow \langle \lambda x_0.e_0, \rho\rangle & \end{array}$$

*Example 1.* The expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ evaluates to $\langle \lambda z.y, [y \mapsto 3]\rangle$ as shown by the derivation tree

$$\dfrac{[\,] \vdash \lambda x.x\,3 \longrightarrow \mathsf{C}_x \quad [\,] \vdash \lambda y.\lambda z.y \longrightarrow \mathsf{C}_y \quad \dfrac{\rho_x \vdash x \longrightarrow \mathsf{C}_y \quad \rho_x \vdash 3 \longrightarrow 3 \quad \rho_y \vdash \lambda z.y \longrightarrow \mathsf{C}_z}{\rho_x \vdash x\,3 \longrightarrow \mathsf{C}_z}}{[\,] \vdash (\lambda x.x\,3)(\lambda y.\lambda z.y) \longrightarrow \mathsf{C}_z}$$

where we write $[\,]$ for the environment with empty domain and additionally make use of the abbreviations $\mathsf{C}_x = \langle \lambda x.x\,3, [\,]\rangle$, $\mathsf{C}_y = \langle \lambda y.\lambda z.y, [\,]\rangle$, $\rho_x = [x \mapsto \mathsf{C}_y]$, $\rho_y = [y \mapsto 3]$ and $\mathsf{C}_z = \langle \lambda z.y, \rho_y\rangle$. □

The aim of the analysis is to statically predict which values an expression may evaluate to. The analysis works with abstract representations of the semantic values. All constants are represented by the entity $\diamond$ so the analysis will record the presence of a constant but not its actual value. A closure $\langle \lambda x_0.e_0, \rho_0\rangle$ is represented by an abstract closure $\{\!|\lambda x_0.e_0|\!\}$; a representation of the environment $\rho_0$ will be part of a global abstract environment $\hat\rho \in \widehat{\mathbf{Env}} = \mathbf{Var} \rightarrow \widehat{\mathbf{Val}}$. The judgements of the analysis will be specified in terms of *sets* $\hat v \in \widehat{\mathbf{Val}}$ of such abstract values and will be relative to an abstract environment. The abstract environment is *global* meaning that it is going to represent *all* the environments that may arise during the evaluation of the expression — thus, in general, a more precise analysis result can be obtained by renaming bounding occurrences of variables apart. More details of the analysis will be provided shortly.

*Example 2.* Continuing the above example, the analysis may determine that during the evaluation of the expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ the variable $x$ may be bound to $\{\!|\lambda y.\lambda z.y|\!\}$, $y$ may be bound to $\diamond$, $z$ will never be bound to anything (as the function $\lambda z.y$ is never applied) and, furthermore, the expression may evaluate to a value represented by the abstract closure $\{\!|\lambda z.y|\!\}$. □

## 2.1 An Abstract Succinct Specification

Our first specification is given by judgements of the form

$$\hat\rho \models_{\mathsf{as}} e : \hat v$$

and expresses that the set $\hat v$ is an acceptable analysis estimate for the expression $e$ in the context specified by the abstract environment $\hat\rho$. The analysis is defined by the clauses:

$$
\begin{array}{rll}
\hat{\rho} \models_{\mathsf{as}} c : \hat{v} & \text{iff} & \diamond \in \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} x : \hat{v} & \text{iff} & \hat{\rho}(x) \subseteq \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} \lambda x_0.e_0 : \hat{v} & \text{iff} & \{\!|\lambda x_0.e_0|\!\} \in \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} e_1\, e_2 : \hat{v} & \text{iff} & \hat{\rho} \models_{\mathsf{as}} e_1 : \hat{v}_1 \;\wedge\; \hat{\rho} \models_{\mathsf{as}} e_2 : \hat{v}_2 \;\wedge \\
& & \forall \{\!|\lambda x_0.e_0|\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow\; [\hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \hat{\rho} \models_{\mathsf{as}} e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}]
\end{array}
$$

The first clause expresses that the analysis estimate must contain the abstract value $\diamond$, the second clause requires that the analysis estimate includes all the values that $x$ can take in the abstract environment $\hat{\rho}$ and the third clause states that the abstract closure $\{\!|\lambda x_0.e_0|\!\}$ must be included in the analysis estimate.

The clause for application expresses that in order for $\hat{v}$ to be an acceptable analysis estimate then it must be possible to find acceptable analysis estimates for the operator as well as the operand. Furthermore, whenever some abstract closure $\{\!|\lambda x_0.e_0|\!\}$ is a possible value of the operator and whenever the operand may evaluate to something at all (i.e $\hat{v}_2 \neq \emptyset$) then it must be the case that the potential actual parameters $\hat{v}_2$ are included in the possible values for the formal parameter $x_0$; also, there must exist an acceptable analysis estimate $\hat{v}_0$ for the body $e_0$ that is included in the analysis estimate $\hat{v}$ of the application itself. Note that $\hat{v}_0$, $\hat{v}_1$ and $\hat{v}_2$ only occur on the right hand side of the clause; it is left implicit that they are *existentially quantified*.

*Example 3.* Returning to the expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ of the previous examples let us verify that

$$\hat{\rho} \models_{\mathsf{as}} (\lambda x.x\,3)(\lambda y.\lambda z.y) : \{\{\!|\lambda z.y|\!\}\}$$

is an acceptable analysis estimate in the context given by:

$$\hat{\rho} = \begin{array}{|c|c|c|} x & y & z \\ \hline \{\{\!|\lambda y.\lambda z.y|\!\}\} & \{\diamond\} & \emptyset \\ \hline \end{array}$$

First we observe that the clause for $\lambda$-abstraction gives:

$$\hat{\rho} \models_{\mathsf{as}} \lambda x.x\,3 : \{\{\!|\lambda x.x\,3|\!\}\} \text{ and } \hat{\rho} \models_{\mathsf{as}} \lambda y.\lambda z.y : \{\{\!|\lambda y.\lambda z.y|\!\}\}$$

This establishes the first two conditions of the clause for application. To verify the third condition we only have to consider $\{\!|\lambda x.x\,3|\!\}$ and we have to *guess* an analysis estimate for the body $x\,3$. One such guess gives us the proof obligations:

$$\{\{\!|\lambda y.\lambda z.y|\!\}\} \subseteq \hat{\rho}(x) \text{ and } \hat{\rho} \models_{\mathsf{as}} x\,3 : \{\{\!|\lambda z.y|\!\}\} \text{ and } \{\{\!|\lambda z.y|\!\}\} \subseteq \{\{\!|\lambda z.y|\!\}\}$$

The first and the third of these are trivial and to verify the second condition we apply the clause for application once again; we dispense with the details.

Note that $\hat{\rho}(z) = \emptyset$. This reflects that $z$ is never bound to anything, or equivalently, that $\lambda z.y$ is never applied. □

The above specification is *abstract* because the body of a $\lambda$-abstraction is only required to have an acceptable analysis estimate if the $\lambda$-abstraction might be applied somewhere. This follows the tradition of data flow analysis and abstract interpretation where only reachable code it analysed and it is in contrast to the more type theoretic approaches where all subexpressions are required to be typable. Note that we may be required to find several analysis estimates for the body of a single $\lambda$-abstraction as it may be applied in many places; in the terminology of constraint-based analysis we may say that the analysis is polyvariant.

The specification is *succinct* because the occurrence of $\hat{v}$ in $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ expresses the overall analysis estimate for $e$; if we want details about the analysis information for subexpressions of $e$ then we have to inspect the reasoning leading to the judgement $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$.

It is interesting to note that the specification applies to *open systems* as well as closed systems since $\lambda$-abstractions are analysed when they are applied; hence they are not required to be part of the program of interest but may for example be part of library routines.

## 2.2 A Compositional Succinct Specification

A compositional specification of the same analysis will analyse the body of $\lambda$-abstractions at their definition point rather than at their application point. This means that we need some way of linking information available at these points and for this we introduce a global cache $\hat{C}$ that will record the analysis estimates for the bodies of the $\lambda$-abstractions. In order to refer to these bodies we shall extend the syntax with labels $\ell \in \mathbf{Lab}$

$$e ::= c \mid x \mid \lambda x_0.e_0^{\ell_0} \mid e_1\, e_2$$

and we take $\hat{C} \in \widehat{\mathbf{Cache}} = \mathbf{Lab} \to \widehat{\mathbf{Val}}$. The labels allow us to refer to specific program points in an explicit way and they allow us to use the cache as a global data structure but they play no role in the semantics; for optimal precision the subexpressions should be uniquely labelled. The judgements of our analysis now have the form

$$(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$$

and expresses that $\hat{v}$ is an acceptable analysis estimate for $e$ in the context specified by $\hat{C}$ and $\hat{\rho}$. The analysis is defined as follows:

$$\begin{array}{rcl}
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} c : \hat{v} & \mathsf{iff} & \diamond \in \hat{v} \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} x : \hat{v} & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{v} \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} \lambda x_0.e_0^{\ell_0} : \hat{v} & \mathsf{iff} & \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{v} \land \\
& & \hat{\rho}(x_0) \neq \emptyset \Rightarrow [(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_0^{\ell_0} : \hat{v}_0 \land \hat{v}_0 \subseteq \hat{C}(\ell_0)] \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_1\, e_2 : \hat{v} & \mathsf{iff} & (\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_1 : \hat{v}_1 \ \land\ (\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_2 : \hat{v}_2 \land \\
& & \forall \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{v}_1 : \hat{v}_2 \subseteq \hat{\rho}(x_0) \land \hat{C}(\ell_0) \subseteq \hat{v}
\end{array}$$

As before the third clause expresses that $\{\!|\lambda x_0.e_0^{\ell_0}|\!\}$ must be included in the analysis estimate but it additionally requires that *if* the $\lambda$-abstraction is ever applied to some value *then* the body $e_0$ has an acceptable analysis estimate $\hat{v}_0$ that is included in the cache for $\ell_0$. The check of whether or not the $\lambda$-abstraction is applied is merely a check of whether the abstract environment specifies that some actual parameters may be bound to $x_0$. The clause for application is as before with the exception that we consult the appropriate entry in the cache whenever we need to refer to an analysis estimate for the body of a $\lambda$-abstraction.

*Example 4.* We shall now annotate the example as $(\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2)$ and show that

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} (\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2) : \{\{\!|\lambda z.y^3|\!\}\}$$

is an acceptable analysis estimate in the context where:

$$\hat{C} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \{\{\!|\lambda z.y^3|\!\}\} & \{\{\!|\lambda z.y^3|\!\}\} & \emptyset \\ \hline \end{array} \quad \text{and} \quad \hat{\rho} = \begin{array}{|c|c|c|} \hline x & y & z \\ \hline \{\{\!|\lambda y.(\lambda z.y^3)^2|\!\}\} & \{\diamond\} & \emptyset \\ \hline \end{array}$$

The clause for application requires that we *guess* acceptable analysis estimates for the operator and operand; one such guess leaves us with the proof obligations

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda x.(x\,3)^1 : \{\{\!|\lambda x.(x\,3)^1|\!\}\} \text{ and } (\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda y.(\lambda z.y^3)^2 : \{\{\!|\lambda y.(\lambda z.y^3)^2|\!\}\}$$

together with $\{\{\!|\lambda y.(\lambda z.y^3)^2|\!\}\} \subseteq \hat{\rho}(x)$ and $\hat{C}(1) \subseteq \{\{\!|\lambda z.y^3|\!\}\}$. Let us concentrate on the second judgement above where we will use the clause for $\lambda$-abstraction. Clearly $\{\!|\lambda y.(\lambda z.y^3)^2|\!\} \in \{\{\!|\lambda y.(\lambda z.y^3)^2|\!\}\}$ and since $\hat{\rho}(y) \neq \emptyset$ we have to *guess* an acceptable analysis estimate for the body of the $\lambda$-abstraction. It is here sufficient to show:

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda z.y^3 : \{\{\!|\lambda z.y^3|\!\}\} \text{ and } \{\{\!|\lambda z.y^3|\!\}\} \subseteq \hat{C}(2)$$

The latter is trivial and since $\hat{\rho}(z) = \emptyset$ we immediately get the former from the clause for $\lambda$-abstraction. Note that $\hat{C}(3) = \emptyset$; this reflects that the body of the function $\lambda z.y^3$ is never evaluated as the function is never called. □

Clearly the specification is *compositional*; it still has *succinct* components since the $\hat{v}$ of $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$ gives the analysis information for the overall expression $e$; to get hold of the analysis information of subexpressions we have to inspect the reasoning leading to the judgement $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$. In contrast to the abstract specification, we only have to find one acceptable analysis estimate for each subexpression; in the terminology of constraint-based analysis we may say that the analysis is monovariant.

**Relationship between the specifications.** The above specification is restricted to *closed* systems since the bodies of $\lambda$-abstractions only are analysed at their definition points. If we restrict our attention to closed systems then the above specification can be seen as a refinement of the one of Subsection 2.1 in the following sense:

> Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$ and $\hat{C}$ and $\hat{\rho}$ only mention $\lambda$-abstractions occurring in $e$ then $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ (modulo the labelling of expressions).

This means that an implementation that is faithful to the compositional specification also will be faithful to the abstract specification. The converse result does not hold in general; this is analogous to the insight that polyvariant analyses often are more precise than monovariant ones. In [8,18] we present proof techniques for establishing the relationship between abstract and compositional specifications.

We may obtain an analysis closer in spirit to type systems by replacing the clause for $\lambda$-abstraction with

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda x_0.e_0^{\ell_0} : \hat{v} \quad \mathsf{iff} \quad \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{v} \ \wedge\ (\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e_0^{\ell_0} : \hat{v}_0 \ \wedge\ \hat{v}_0 \subseteq \hat{C}(\ell_0)$$

where the body must have an acceptable analysis estimate even in the case where the $\lambda$-abstraction is never applied. This will give a less precise (but still semantically correct) analysis.

*Example 5.* With the above clause the proof of $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda z.y^3 : \{\{\!|\lambda z.y^3|\!\}\}$ breaks down: we have to show that $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} y^3 : \hat{v}_0$ and $\hat{v}_0 \subseteq \hat{C}(3)$ for some $\hat{v}_0$ and this requires that $\diamond \in \hat{\rho}(y) \subseteq \hat{v}_0$ which contradicts $\hat{C}(3) = \emptyset$. For the slightly less precise analysis result where $\hat{\rho}$ and $\hat{C}$ are as above except that $\hat{\rho}(z) = \hat{C}(3) = \{\diamond\}$ we can indeed show that $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} (\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2) : \{\{\!|\lambda z.y^3|\!\}\}$. □

## 2.3 A Compositional Verbose Specification

To get an even more implementation oriented specification we shall extend our use of the cache to record the analysis estimates of *all* the subexpressions; we shall call the analysis *verbose* since the analysis domain captures all the analysis information of interest for the complete program. To formalise this we extend our notion of labelling to

$$e^\ell ::= c^\ell \mid x^\ell \mid (\lambda x_0.e_0^{\ell_0})^\ell \mid (e_1^{\ell_1}\, e_2^{\ell_2})^\ell$$

and as before the labels play no role in the semantics; they only serve as explicit reference points for the analysis and take a form resembling addresses in the syntax tree. The judgements now have the form

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^\ell$$

and the intension is that $\hat{C}(\ell)$ is the analysis estimate for $e$. The analysis is defined by the following clauses that are obtained by a simple rewriting of those of Subsection 2.2:

$$\begin{array}{rcl}
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} c^\ell & \mathsf{iff} & \diamond \in \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} x^\ell & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (\lambda x_0.e_0^{\ell_0})^\ell & \mathsf{iff} & \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{C}(\ell) \wedge \\
& & \hat{\rho}(x_0) \neq \emptyset \Rightarrow (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_0^{\ell_0} \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (e_1^{\ell_1}\, e_2^{\ell_2})^\ell & \mathsf{iff} & (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_1^{\ell_1} \ \wedge\ (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_2^{\ell_2} \ \wedge \\
& & \forall \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{C}(\ell_1) : \hat{C}(\ell_2) \subseteq \hat{\rho}(x_0) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(\ell)
\end{array}$$

Note that in contrast to the previous two specifications all entities occurring on the right hand sides of the clauses also occur on the left hand side; hence there are no implicitly quantified variables.

*Example 6.* We now write the expression as $(\lambda x.(x^4\, 3^5)^1)^6(\lambda y.(\lambda z.y^3)^2)^7)^8$ and an acceptable analysis estimate is

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (\lambda x.(x^4\, 3^5)^1)^6(\lambda y.(\lambda z.y^3)^2)^7)^8$$

where $\hat{\rho}$ is as in Example 3 and:

$\hat{C} =$

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| $\{\lbag\lambda z.y^3\rbag\}$ | $\{\lbag\lambda z.y^3\rbag\}$ | $\emptyset$ | $\{\lbag\lambda y.(\lambda z.y^3)^2\rbag\}$ |
| 5 | 6 | 7 | 8 |
| $\{\diamond\}$ | $\{\lbag\lambda x.(x^4\,3^5)^1\rbag\}$ | $\{\lbag\lambda y.(\lambda z.y^3)^2\rbag\}$ | $\{\lbag\lambda z.y^3\rbag\}$ |

It is straightforward to verify that this is indeed an acceptable analysis estimate; in particular there is no need to make any guesses of intermediate judgements as all the required information is available in the cache. We omit the details. $\square$

The specification is clearly *compositional* and we say that it is *verbose* since the cache $\hat{C}$ of a judgement $(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^\ell$ contains all the analysis information of interest for the program.

**Relationship between the specifications.** The verbose specification is a refinement of the specification of Subsection 2.2 in the sense that:

Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^\ell$ then $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{C}(\ell)$ (modulo the labelling of expressions).

Thus an implementation of the verbose specification will also provide solutions to the abstract (succinct as well as compositional) specification. The systematic transformation of a compositional succinct specification into a compositional verbose specification is studied in [21]. One may note that the converse of the lemma also holds due to the compositionality of the specifications.

## 2.4 An Abstract Verbose Specification

The verbose specification style is characterised by the explicit caching of analysis results for all subexpressions and we shall now combine it with the abstract specification style thereby generalising the previous specification to open systems. The new judgements have the form

$$(\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^\ell$$

where the labelling scheme is as in Subsection 2.3. The analysis is specified by the clauses:

$$\begin{array}{rcl}
(\hat{C},\hat{\rho}) \models_{\mathsf{av}} c^\ell & \mathsf{iff} & \diamond \in \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{av}} x^\ell & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{av}} (\lambda x_0.e_0^{\ell_0})^\ell & \mathsf{iff} & \lbag\lambda x_0.e_0^{\ell_0}\rbag \in \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{av}} (e_1^{\ell_1}\, e_2^{\ell_2})^\ell & \mathsf{iff} & (\hat{C},\hat{\rho}) \models_{\mathsf{av}} e_1^{\ell_1} \,\wedge\, (\hat{C},\hat{\rho}) \models_{\mathsf{av}} e_2^{\ell_2} \,\wedge \\
 & & \forall \lbag\lambda x_0.e_0^{\ell_0}\rbag \in \hat{C}(\ell_1) : \hat{C}(\ell_2) \neq \emptyset \Rightarrow \\
 & & \qquad [\hat{C}(\ell_2) \subseteq \hat{\rho}(x_0) \wedge (\hat{C},\hat{\rho}) \models_{\mathsf{av}} e_0^{\ell_0} \wedge \hat{C}(\ell_0) \subseteq \hat{C}(\ell)]
\end{array}$$

Note again that $\lambda$-abstractions that are never applied need not have acceptable analysis estimates.

*Example 7.* For our example expression we now have

$$(\hat{C},\hat{\rho}) \models_{\mathsf{av}} (\lambda x.(x^4\,3^5)^1)^6(\lambda y.(\lambda z.y^3)^2)^7)^8$$

where $\hat{C}$ and $\hat{\rho}$ are as in the previous example. It is straightforward to verify that this is indeed a valid judgement and as in Subsection 2.3 no guessing of intermediate judgements are needed as all the required information is available in the cache. We omit the details. $\square$

Clearly the specification is *abstract* as the clause for application demands the analysis of all bodies of $\lambda$-abstractions that result from the operator part. It is also *verbose* since the abstract domains contain all the analysis information of interest for the program.

**Relationship between the specifications.** In analogy with the result of Subsection 2.2 (and those of [8,18]) we have:

> Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^{\ell}$ and $\hat{C}$ and $\hat{\rho}$ only mention $\lambda$-abstractions occurring in $e$ then $(\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^{\ell}$ (modulo the labelling of expressions).

We can also express a relationship between the two abstract specifications (analogously to the result stated in Subsection 2.3):

> Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^{\ell}$ then $\hat{\rho} \models_{\mathsf{as}} e : \hat{C}(\ell)$ (modulo the labelling of expressions).

The converse results do not hold in general due to the polyvariant nature of the abstract specifications.

The four results are summarised in the "flow logic square":

$$\begin{tikzcd}
(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{C}(\ell) \arrow[d] & (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^{\ell} \arrow[l] \arrow[d] \\
\hat{\rho} \models_{\mathsf{as}} e : \hat{C}(\ell) & (\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^{\ell} \arrow[l]
\end{tikzcd}$$

## 3 The Methodology of Flow Logic

A flow logic *specification* defines:

- the universe of discourse for the analysis estimates;
- the format of the judgements; and
- the defining clauses.

The universe of discourse usually is given by complete lattices and hence follows the approaches of data flow analysis, constraint-based analysis and abstract interpretation. The formulation of the judgements and their clauses focuses on *what* the analysis does and not *how* it does it. There are several benefits from this: (*i*) it is not necessary to think about the design and the implementation of the analysis at the same time; (*ii*) one can concentrate on specifying the analysis, i.e. on how to collect analysis information and link it together and (*iii*) the problem of trading efficiency for precision can be studied at the specification level and hence independently of implementation details.

Having specified a flow logic the next step is to show that the analysis enjoys a number of desirable properties:

– the judgements are well-defined;
– the judgements are semantically correct;
– the judgements have a Moore family (or model intersection) property; and
– the judgements have efficient implementations.

The Moore family property is important because it ensures not only that each program can be analysed but also that it has a best or most precise analysis result. We now survey the appropriate techniques and illustrate them on the example analyses of Section 2.

## 3.1 Well-Definedness of the Analysis

It is straightforward to see that the compositional specifications are well-defined; a simple induction on the syntax of the programs will do. However, it is not so obvious that the abstract specifications are well-defined; for the analysis specified in Subsection 2.1 it is the following clause that is problematic:

$$\begin{array}{lll}\hat{\rho} \models_{\mathsf{as}} e_1\, e_2 : \hat{v} & \text{iff} & \hat{\rho} \models_{\mathsf{as}} e_1 : \hat{v}_1 \;\wedge\; \hat{\rho} \models_{\mathsf{as}} e_2 : \hat{v}_2 \;\wedge \\ & & \forall \{\!|\lambda x_0.e_0|\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow [\hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \hat{\rho} \models_{\mathsf{as}} e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}]\end{array}$$

To ensure that an abstract specification is well-defined we shall turn the defining clauses into an appropriate functional over some complete lattice — in doing so we shall impose some demands on the form of the clauses such that monotonicity of the functional is enforced. The benefit is that then Tarski's fixed point theorem ensures that the functional has fixed points, in particular that it has a least as well as a greatest fixed point. Since we are giving meaning to a specification, the analysis is defined as the *greatest* fixed point (i.e. co-inductively) — intuitively, this means that only judgements that violates the conditions imposed by the specification are excluded. Actually, if we had insisted on defining the analysis as the least fixed point of the functional then there are important parts of the development that would fail; in particular, the Moore family property would not hold in general so there are programs that cannot be analysed [18].

The approach of defining the analysis coinductively works for abstract as well as compositional specifications; however, in the latter case it turns out that the functional has *exactly one* fixed point so the least and the greatest fixed points coincide and we may safely say that the analysis is defined *inductively* as mentioned above.

We shall illustrate the technique for the *abstract succinct specification* of Subsection 2.1. Write $(\mathbf{Q}, \sqsubseteq)$ for the complete lattice $\widehat{\mathbf{Env}} \times \mathbf{Exp} \times \widehat{\mathbf{Val}} \to \{\mathsf{tt}, \mathsf{ff}\}$ with the ordering

$$Q_1 \sqsubseteq Q_2 \;\; \text{iff} \;\; \forall(\hat{\rho}, e, \hat{v}) : (Q_1(\hat{\rho}, e, \hat{v}) = \mathsf{tt}) \Rightarrow (Q_2(\hat{\rho}, e, \hat{v}) = \mathsf{tt})$$

expressing that $Q_2$ may give true more often than $Q_1$; it is easy to check that this indeed defines a complete lattice. The clauses for $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ define the following functional $\mathcal{Q}_{\mathsf{as}} : \mathbf{Q} \to \mathbf{Q}$ (i.e. $\mathcal{Q}_{\mathsf{as}} : \mathbf{Q} \to (\widehat{\mathbf{Env}} \times \mathbf{Exp} \times \widehat{\mathbf{Val}} \to \{\mathsf{tt}, \mathsf{ff}\})$):

$$\begin{aligned}
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, c, \hat{v}) &= (\diamond \in \hat{v})\\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, x, \hat{v}) &= (\hat{\rho}(x) \subseteq \hat{v})\\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, \lambda x_0.e_0, \hat{v}) &= (\{\!\mid \lambda x_0.e_0 \mid\!\} \in \hat{v})\\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, e_1\, e_2, \hat{v}) &= \exists \hat{v}_1, \hat{v}_2 : Q(\widehat{\rho}, e_1, \hat{v}_1) \wedge Q(\widehat{\rho}, e_2, \hat{v}_2) \wedge\\
&\qquad \forall \{\!\mid \lambda x_0.e_0 \mid\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow\\
&\qquad\qquad [\exists \hat{v}_0 : \hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge Q(\widehat{\rho}, e_0, \hat{v}_0) \wedge \hat{v}_0 \subseteq \hat{v}]
\end{aligned}$$

It is straightforward to check that $\mathcal{Q}_{\mathsf{as}}$ is monotonic: when $Q$ gives true more often then so does $\mathcal{Q}_{\mathsf{as}}(Q)$ — intuitively, the reason why $\mathcal{Q}_{\mathsf{as}}$ is monotonic is that all occurrences of $Q$ on the right hand sides of the equations occur in positive positions: they are prefixed by an even (in particular none) number of negations. It then follows from Tarski's fixed point theorem that $\mathcal{Q}_{\mathsf{as}}$ has a complete lattice of fixed points in $\mathbf{Q}$ and we define $\models_{\mathsf{as}}$ to be the *greatest* fixed point.

## 3.2 Semantic Correctness

Flow logic is a *semantics based* approach to static analysis meaning that the information obtained from the analysis can be proved correct with respect to a semantics. The correctness can be established with respect to many different kinds of semantics (e.g. big-step or small-step operational semantics or denotational semantics); however, as pointed out in [15,16] the actual choice of semantics may significantly influence the style of the proof.

To establish semantic correctness we must define a *correctness relation* between the entities of the semantics and the analysis estimates. For our example both the semantics and the analysis operate with values and environments so we shall define two relations $\mathfrak{R}_{\mathsf{Val}} \subseteq \mathbf{Val} \times (\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}})$ and $\mathfrak{R}_{\mathsf{Env}} \subseteq \mathbf{Env} \times \widehat{\mathbf{Env}}$. The formal definitions amount to nothing but a formalisation of the intuition behind how the semantic entities are represented in the analysis:

$$\begin{aligned}
c\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho}) \quad &\text{iff} \quad \diamond \in \hat{v}\\
\langle \lambda x_0.e_0, \rho_0 \rangle\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho}) \quad &\text{iff} \quad \{\!\mid \lambda x_0.e_0 \mid\!\} \in \hat{v}\ \wedge\ \rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho}\\
\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho} \quad &\text{iff} \quad \forall x \in \mathsf{dom}(\rho) : \rho(x)\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{\rho}(x), \hat{\rho})
\end{aligned}$$

It is immediate to show that the relations are well-defined by induction on the size of the semantic entities.

We shall now exploit that the abstract values and environments can be viewed as complete lattices in the following way: The abstract values $\widehat{\mathbf{Val}}$ constitutes a powerset and can naturally be equipped with the subset ordering; this ordering is then lifted in a pointwise manner to the abstract environments of $\widehat{\mathbf{Env}}$ and to the pairs of $\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}$. Following the tradition of the denotational approach to abstract interpretation it is then desirable that the correctness relations satisfy admissibility conditions like

$$\begin{aligned}
v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho})\ \wedge\ (\hat{v}, \hat{\rho}) \sqsubseteq (\hat{v}', \hat{\rho}') \quad &\Rightarrow \quad v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}', \hat{\rho}')\\
(\forall (\hat{v}, \hat{\rho}) \in Y \subseteq (\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}) : v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho})) \quad &\Rightarrow \quad v\ \mathfrak{R}_{\mathsf{Val}}\ (\textstyle\bigsqcap Y)
\end{aligned}$$

and similarly for $\mathfrak{R}_{\mathsf{Env}}$. The first condition expresses that a small abstract value is more precise than a larger one whereas the second condition expresses that there always is a least abstract value for describing a value. In Subsection 3.3 we show that these properties can be lifted to the judgements.

Given a semantics that relates pairs of configurations, the correctness result is often expressed as a *subject reduction result*: if the first configuration is described by the analysis estimate then so is the second configuration. We shall first illustrate this for the abstract succinct analysis of Subsection 2.1 against the big-step operational semantics of Section 2 and subsequently we illustrate it against a small-step operational semantics presented below.

**Big-step operational semantics.** Recall that the transitions of the semantics have the form $\rho \vdash e \longrightarrow v$ meaning that executing $e$ in the environment $\rho$ will give the value $v$. The correctness result for the analysis is expressed by:

Theorem: If $\rho \vdash e \longrightarrow v$, $\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat\rho$ and $\hat\rho \models_{\mathsf{as}} e : \hat v$ then $v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat v, \hat\rho)$.

So if $\hat\rho$ describes $\rho$ then the result $v$ of evaluating $e$ in $\rho$ is described by $\hat v$. In the special case where $v$ is a closure $\langle \lambda x_0.e_0, \rho_0\rangle$ this amounts to $\{\!|\lambda x_0.e_0|\!\} \in \hat v$ (and $\rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat\rho$) and in the case where $v$ is a constant $c$ it amounts to $\diamond \in \hat v$. The proof of the theorem is by induction in the inference $\rho \vdash e \longrightarrow v$.

**Small-step operational semantics.** An environment-based small-step operational semantics for the $\lambda$-calculus uses values $v \in \mathbf{Val}$ together with intermediate expressions containing closures $\langle \lambda x_0.e_0, \rho_0\rangle$ and special constructs of the form $\mathsf{bind}\,\rho_0\,\mathsf{in}\,e_0$; the role of the latter is to stack the environments arising in applications. The transitions are written as $\rho \Vdash e \longrightarrow e'$ and are defined by:

$$\rho \Vdash x \longrightarrow \rho(x) \qquad \frac{\rho \Vdash e_1 \longrightarrow e_1'}{\rho \Vdash e_1\, e_2 \longrightarrow e_1'\, e_2}$$

$$\rho \Vdash \lambda x_0.e_0 \longrightarrow \langle \lambda x_0.e_0, \rho\rangle$$

$$\rho \Vdash (\langle \lambda x_0.e_0, \rho_0\rangle)\, v \longrightarrow \mathsf{bind}\,\rho_0[x_0 \mapsto v]\,\mathsf{in}\,e_0 \qquad \frac{\rho \Vdash e_2 \longrightarrow e_2'}{\rho \Vdash e_1\, e_2 \longrightarrow e_1\, e_2'}$$

$$\frac{\rho_0 \Vdash e \longrightarrow e'}{\rho \Vdash \mathsf{bind}\,\rho_0\,\mathsf{in}\,e \longrightarrow \mathsf{bind}\,\rho_0\,\mathsf{in}\,e'} \qquad \frac{\rho_0 \Vdash e \longrightarrow v}{\rho \Vdash \mathsf{bind}\,\rho_0\,\mathsf{in}\,e \longrightarrow v}$$

The correctness result will still be a subject reduction result and since some of the configurations may contain intermediate expressions we shall need to *extend the analysis* to handle these:

$$\hat\rho \models_{\mathsf{as}} \langle \lambda x_0.e_0, \rho_0\rangle : \hat v \quad \text{iff} \quad \langle \lambda x_0.e_0, \rho_0\rangle\ \mathfrak{R}_{\mathsf{Val}}\ (\hat v, \hat\rho)$$

$$\hat\rho \models_{\mathsf{as}} \mathsf{bind}\,\rho_0\,\mathsf{in}\,e_0 : \hat v \quad \text{iff} \quad \hat\rho \models_{\mathsf{as}} e_0 : \hat v \ \wedge\ \rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat\rho$$

In both cases we use the correctness relations $\mathfrak{R}_{\mathsf{Val}}$ and $\mathfrak{R}_{\mathsf{Env}}$ to express the relationship between the entities of the semantics and the analysis. As before the analysis and the correctness relations are easily shown to be well-defined.

We can now formulate the semantic correctness of the analysis as

Theorem: If $\rho \Vdash e \longrightarrow e'$, $\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat\rho$ and $\hat\rho \models_{\mathsf{as}} e : \hat v$ then $\hat\rho \models_{\mathsf{as}} e' : \hat v$.

expressing that the analysis estimate remains acceptable as the computation proceeds; the proof is by induction on the transitions of the semantics. This correctness result is slightly stronger than the previous one since it also applies to non-terminating computations.

Similar results can be obtained for the other three formulations of the analysis; naturally the correctness relations have to be extended to take the cache into account. We refer to [16] for a discussion of how the choice of semantics may influence the required proof techniques.

## 3.3 Moore Family Result

Having specified an analysis it is natural to ask whether every expression admits an acceptable analysis estimate and whether every expression has a best or most informative analysis estimate. To answer both of these questions in the affirmative it is sufficient to prove that the set of acceptable analysis estimates enjoys a Moore family (or model intersection) property: A *Moore family* is a subset $\widehat{V}$ of a complete lattice satisfying that whenever $Y \subseteq \widehat{V}$ then $\bigsqcap Y \in \widehat{V}$.

Let us once again consider the abstract succinct specification of Subsection 2.1. The complete lattice of interest is $\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}$ equipped with the pointwise subset ordering and the Moore family result can be formulated as follows:

Theorem: The set $\{(\hat\rho, \hat v) \mid \hat\rho \models_{\mathsf{as}} e : \hat v\}$ is a Moore family for all $e$.

Taking $Y = \{(\hat\rho, \hat v) \mid \hat\rho \models_{\mathsf{as}} e : \hat v\}$ it follows that each expression has a least and acceptable analysis estimate.

Similar results can be formulated for the other styles of specification; the proof will be by co-induction or induction depending on whether the acceptability judgement is defined by co-induction or induction [18].

## 3.4 Efficient Implementation

The compositional verbose specification of Subsection 2.3 is a good starting point for actually obtaining an implementation. The idea will be to turn it into an algorithm for computing a set of conditional constraints that subsequently can be solved using standard constraint solvers as available in for example the BANE system [2].

For our example analysis we shall introduce two arrays $\mathsf{C}$ and $\mathsf{R}$ indexed by the labels and variables occurring in the expression $e_\star$ of interest and corresponding to the mappings $\hat C$ and $\hat\rho$ of the specification. The conditional constraints have the form $\mathit{lst}$ => $\mathit{lhs} \subseteq \mathit{rhs}$ and express that if the conjunction of the conditions of the list $\mathit{lst}$ holds then one set is included in another. The list may contain conditions of the form $\mathit{set} \neq \emptyset$ and $\mathit{elm} \in \mathit{set}$ with their obvious interpretation.

Initially, the algorithm $\mathcal{C}_{\mathsf{cv}}$ is applied to $e_\star$ and an empty list $\epsilon$ of conditions and it will construct a set of conditional constraints. The general algorithm is as follows:

$$\begin{aligned}
\mathcal{C}_{\mathsf{cv}}[\![c^\ell]\!]\mathit{lst} &= \{\ \mathit{lst} \texttt{ => } \{\diamond\} \subseteq \mathsf{C}[\ell]\ \}\\
\mathcal{C}_{\mathsf{cv}}[\![x^\ell]\!]\mathit{lst} &= \{\ \mathit{lst} \texttt{ => } \mathsf{R}[x] \subseteq \mathsf{C}[\ell]\ \}\\
\mathcal{C}_{\mathsf{cv}}[\![(\lambda x_0.e_0^{\ell_0})^\ell]\!]\mathit{lst} &= \{\ \mathit{lst} \texttt{ => } \{\langle\!|\lambda x_0.e_0^{\ell_0}|\!\rangle\} \subseteq \mathsf{C}[\ell]\ \} \ \cup\ \mathcal{C}_{\mathsf{cv}}[\![e_0^{\ell_0}]\!](\mathit{lst}\,\hat{}\,(\mathsf{R}[x_0] \neq \emptyset))\\
\mathcal{C}_{\mathsf{cv}}[\![(e_1^{\ell_1}\ e_2^{\ell_2})^\ell]\!]\mathit{lst} &= \mathcal{C}_{\mathsf{cv}}[\![e_1^{\ell_1}]\!]\mathit{lst}\ \cup\ \mathcal{C}_{\mathsf{cv}}[\![e_2^{\ell_2}]\!]\mathit{lst}\\
&\quad \cup \{\ \mathit{lst}\,\hat{}\,(\langle\!|\lambda x_0.e_0^{\ell_0}|\!\rangle \in \mathsf{C}[\ell_1]) \texttt{ => } \mathsf{C}[\ell_2] \subseteq \mathsf{R}[x_0]\ \mid \lambda x_0.e_0^{\ell_0} \text{ is in } e_\star\}\\
&\quad \cup \{\ \mathit{lst}\,\hat{}\,(\langle\!|\lambda x_0.e_0^{\ell_0}|\!\rangle \in \mathsf{C}[\ell_1]) \texttt{ => } \mathsf{C}[\ell_0] \subseteq \mathsf{C}[\ell]\ \mid \lambda x_0.e_0^{\ell_0} \text{ is in } e_\star\}
\end{aligned}$$

For each $\lambda$-abstraction a new condition is appended to the list; the current list is then used as the condition whenever a constraint is generated. For each application we construct a *set* of conditional constraints, one for each of the

$\lambda$-abstractions of $e_\star$. In this simple case, the set of generated constraints can be solved by a simple worklist algorithm.

One can formally prove a *syntactic soundness and completeness result* expressing that any solution to the above constraint system is also an acceptable analysis estimate according to the specification of Subsection 2.3 and vice versa:

Lemma: $(\hat{C}, \hat{\rho}) \models_{\mathsf{cv}} e_\star$ if and only if $(\hat{\rho}, \hat{C})$ satisfies the constraints of $\mathcal{C}_{\mathsf{cv}}[\![e_\star]\!]\epsilon$.

Hence it follows that a solution $(\hat{\rho}, \hat{C})$ to the constraints also will be an acceptable analysis result for the other three specifications in Section 2. We refer to [18] for further details.

## 4 Pragmatics

The aim of this section is to illustrate that the flow logic approach scales up to other programming paradigms. Space does not allow us to give full specifications so we shall only present fragments of specifications and refer to other papers for more details. We first consider examples from the imperative, the object-oriented and the concurrent paradigm. Then we show how the ideas can be combined for multi-paradigmatic languages. We shall mainly consider abstract succinct specifications; similar developments can be performed for the other styles of specifications.

### 4.1 Imperative Constructs

Imperative languages are characterised by the presence of a *state* that is updated as the execution proceeds; typical constructs are assignments and sequencing of statements:

$$S ::= x := e \mid S_1; S_2 \mid \cdots$$

Here $x \in \mathbf{Var}$ and $e \in \mathbf{Exp}$ are unspecified sets of variables and expressions. We assume a standard semantics operating over states $\sigma \in \mathbf{St}$.

**The analysis.** We shall consider a forward analysis using a complete lattice $(\widehat{\mathbf{St}}, \sqsubseteq)$ of abstract states and with transfer functions $\phi_{x:=e} : \widehat{\mathbf{St}} \to \widehat{\mathbf{St}}$ specifying how the assignments $x := e$ modify the abstract states; as an example we may have $\widehat{\mathbf{St}} = \mathbf{Var} \to \widehat{\mathbf{Val}}$ and $\phi_{x:=e}(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma}]$ where $\widehat{\mathcal{E}}$ defines the analysis of expressions. The judgements of the analysis have the form

$$\models S : \widehat{\sigma} \rhd \widehat{\sigma}'$$

and expresses that $\widehat{\sigma} \rhd \widehat{\sigma}'$ is an acceptable analysis estimate for $S$ meaning that if $\widehat{\sigma}$ describes the initial state then $\widehat{\sigma}'$ will describe the possible final states. For the two constructs above we have the clauses:

$$\begin{array}{rll} \models x := e : \widehat{\sigma} \rhd \widehat{\sigma}' & \text{iff} & \phi_{x:=e}(\widehat{\sigma}) \sqsubseteq \widehat{\sigma}' \\ \models S_1; S_2 : \widehat{\sigma} \rhd \widehat{\sigma}'' & \text{iff} & \models S_1 : \widehat{\sigma} \rhd \widehat{\sigma}' \wedge \models S_2 : \widehat{\sigma}' \rhd \widehat{\sigma}'' \end{array}$$

The first clause expresses that the result $\phi_{x:=e}(\widehat{\sigma})$ of analysing the assignment has been captured by $\widehat{\sigma}'$. The second clause expresses that in order for $\widehat{\sigma} \triangleright \widehat{\sigma}''$ to be an acceptable analysis estimate for $S_1; S_2$ there must exist an abstract state $\widehat{\sigma}'$ such that $\widehat{\sigma} \triangleright \widehat{\sigma}'$ is an acceptable analysis estimate for $S_1$ and $\widehat{\sigma}' \triangleright \widehat{\sigma}''$ is an acceptable analysis estimate for $S_2$.

**A coarse analysis.** For later reference we shall present a coarser analysis that does not distinguish between the program points and hence uses a single abstract state $\widehat{\sigma} \in \widehat{\mathbf{St}} = \mathbf{Var} \rightarrow \widehat{\mathbf{Val}}$ to capture the information about *all* the states that may occur during execution. The judgements of this (verbose) analysis have the form

$$\widehat{\sigma} \models' S$$

and for our two constructs it is defined by the fairly straightforward clauses:

$$\begin{array}{lll} \widehat{\sigma} \models' x := e & \text{iff} & \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} \sqsubseteq \widehat{\sigma}(x) \\ \widehat{\sigma} \models' S_1; S_2 & \text{iff} & \widehat{\sigma} \models' S_1 \ \wedge\ \widehat{\sigma} \models' S_2 \end{array}$$

## 4.2 Object-Oriented Constructs

To illustrate the ideas on an imperative object-oriented language we use a fragment of Abadi and Cardelli's imperative object calculus [1]:

$$O ::= [m_i = \varsigma(x_i).O_i]_{i=1}^n \mid O.m \mid O.m := \varsigma(x_0).O_0 \mid \cdots$$

Here $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ introduces an object with ordered components defining methods with (distinct) names $m_1, \cdots, m_n \in \mathbf{Nam}$, formal parameters $x_1, \cdots, x_n \in \mathbf{Var}$ and bodies $O_1, \cdots, O_n$. Method invocation $O.m$ amounts to first evaluating $O$ to determine an object $o$ of the form $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ and then evaluating the body $O_j$ of the method $m_j$ associated with $m$ (i.e. $m = m_j$) while binding the formal parameter $x_j$ to $o$ so as to allow self-reference; the result of this will then be returned as the overall value. The construct $O.m := \varsigma(x_0).O_0$ will update the method $m$ of the object $o$ that $O$ evaluates to and hence it affects the global state; it is required that $o$ already has a method named $m$. We omit the formal semantics.

**The analysis.** The aim is to determine the set of objects that can reach various points in the program. An object $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ will be represented by a tuple $\overrightarrow{m} = (m_1, \cdots, m_n) \in \mathbf{Nam}^\star$ listing the names of its methods and a method will be represented by an abstract closure $\{\!|\varsigma(x_0).O_0|\!\} \in \mathbf{Mt}$. The judgements of the analysis have the form

$$(\hat{\rho}, \widehat{\sigma}) \models O : \hat{v}$$

and expresses the acceptability of the analysis estimate $\hat{v}$ for $O$ in the context described by $\hat{\rho}$ and $\widehat{\sigma}$. As in Section 2 the idea is that $\hat{v} \in \widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Nam}^\star)$ describes the set of abstract objects that $O$ may evaluate to and $\hat{\rho} : \mathbf{Var} \rightarrow \widehat{\mathbf{Val}}$ describes the abstract values associated with the variables. For simplicity we follow the coarser approach of Subsection 4.1 and represent *all* the states that may arise during the computation by a single abstract state $\widehat{\sigma} \in \widehat{\mathbf{St}} = (\mathbf{Nam}^\star \times \mathbf{Nam}) \to \mathcal{P}(\mathbf{Mt})$ that for each abstract object and method name determines a set of abstract closures.

The clauses for object definition and method call are very similar to the clauses for $\lambda$-abstraction and function application in Subsection 2.1:

$$\begin{array}{rcl} (\hat{\rho},\widehat{\sigma}) \models [m_i = \varsigma(x_i).O_i]_{i=1}^n : \hat{v} & \text{iff} & (m_1,\cdots,m_n) \in \hat{v} \wedge \\ & & \forall i \in \{1,\cdots,n\} : \{\!\!\{\varsigma(x_i).O_i\}\!\!\} \in \widehat{\sigma}((m_1,\cdots,m_n),m_i) \\ (\hat{\rho},\widehat{\sigma}) \models O.m : \hat{v} & \text{iff} & (\hat{\rho},\widehat{\sigma}) \models O : \hat{v}' \wedge \\ & & \forall \overrightarrow{m} \in \hat{v}', \forall \{\!\!\{\varsigma(x_0).O_0\}\!\!\} \in \widehat{\sigma}(\overrightarrow{m},m) : \\ & & \quad \overrightarrow{m} \in \hat{\rho}(x_0) \wedge (\hat{\rho},\widehat{\sigma}) \models O_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v} \end{array}$$

The first clause expresses that $(m_1,\cdots,m_n)$ represents a possible value of the construct and ensures that the abstract state $\widehat{\sigma}$ contains the relevant abstract closures. The second clause first requires that the object $O$ has an acceptable analysis estimate $\hat{v}'$. Then it inspects each of the abstract closures $\{\!\!\{\varsigma(x_0).O_0\}\!\!\}$ that may be associated with $m$ for some abstract object $\overrightarrow{m}$ of $\hat{v}'$: it is required that $\hat{\rho}$ records that the formal parameter $x_0$ may be bound to the actual object $\overrightarrow{m}$ and that the body $O_0$ of the method has an acceptable analysis estimate that is included in the overall analysis estimate.

The clause for method update is inspired by the clause for assignment in the coarser analysis of Subsection 4.1:

$$\begin{array}{rcl} (\hat{\rho},\widehat{\sigma}) \models O.m := \varsigma(x_0).O_0 : \hat{v} & \text{iff} & (\hat{\rho},\widehat{\sigma}) \models O : \hat{v}' \wedge \\ & & \forall \overrightarrow{m} \in \hat{v}' : \widehat{\sigma}(\overrightarrow{m},m) \neq \emptyset \Rightarrow \\ & & \qquad \overrightarrow{m} \in \hat{v} \ \wedge \ \{\!\!\{\varsigma(x_0).O_0\}\!\!\} \in \widehat{\sigma}(\overrightarrow{m},m) \end{array}$$

It expresses that in order for $\hat{v}$ to be an acceptable analysis estimate for the assignment then it must be the case that $O$ has an acceptable analysis estimate $\hat{v}'$. Furthermore, for each abstract object of $\hat{v}'$ that has a method named $m$ it is required that the abstract state $\widehat{\sigma}$ records the abstract closure $\{\!\!\{\varsigma(x_0).O_0\}\!\!\}$.

## 4.3 Concurrency

To illustrate how concurrency and communication can be handled we shall study a fragment of the $\pi$-calculus [12]:

$$P ::= \overline{u}t.P \mid u(x).P \mid (\nu\, n)P \mid P_1 \shortparallel P_2 \mid \cdots$$

Here $t, u$ are terms that can be either variables $x \in \mathbf{Var}$ or channel names $n \in \mathbf{Ch}$. The process $\overline{u}t.P$ will output the channel that $t$ evaluates to over the channel that $u$ evaluates to and then it will continue as $P$. The process $u(x).P$ will input a channel over the channel that $u$ evaluates to, bind it to the variable $x$ and then it will continue as the process $P$. The construct $(\nu\, n)P$ creates a new channel $n$ with scope $P$. The two processes $P_1$ and $P_2$ of the parallel composition $P_1 \shortparallel P_2$ act independently of one another but can also communicate when one of them performs an input and the other an output action over the same channel. We omit the detailed semantics.

**The analysis.** The aim will be to determine which channels may be communicated over which other channels; in [3] we show how this information can be used to validate certain security properties. The judgements of the analysis have the form

$$(\hat{\rho}, \widehat{\kappa}) \models P$$

and expresses that $P$ has an acceptable analysis estimate in the context described by $\hat{\rho}$ and $\widehat{\kappa}$. Here $\hat{\rho} : \mathbf{Var} \to \mathcal{P}(\mathbf{Ch})$ maps the variables to the sets of channels they may be bound to and $\widehat{\kappa} : \mathbf{Ch} \to \mathcal{P}(\mathbf{Ch})$ maps the channels to the sets of channels that may be communicated over them. When formulating the analysis below we shall extend $\hat{\rho}$ to operate on terms and define $\hat{\rho}(n) = \{n\}$ for all channel names $n$. For the above constructs we define the analysis by the following clauses:

$$\begin{array}{rcl}
(\hat{\rho}, \widehat{\kappa}) \models \overline{u}\,t.P & \textsf{iff} & (\hat{\rho}, \widehat{\kappa}) \models P \ \wedge\ \forall n \in \hat{\rho}(u) : \hat{\rho}(t) \subseteq \widehat{\kappa}(n) \\
(\hat{\rho}, \widehat{\kappa}) \models u(x).P & \textsf{iff} & (\hat{\rho}, \widehat{\kappa}) \models P \ \wedge\ \forall n \in \hat{\rho}(u) : \widehat{\kappa}(n) \subseteq \hat{\rho}(x) \\
(\hat{\rho}, \widehat{\kappa}) \models (\nu n)P & \textsf{iff} & (\hat{\rho}, \widehat{\kappa}) \models P \\
(\hat{\rho}, \widehat{\kappa}) \models P_1 \parallel P_2 & \textsf{iff} & (\hat{\rho}, \widehat{\kappa}) \models P_1 \ \wedge\ (\hat{\rho}, \widehat{\kappa}) \models P_2
\end{array}$$

The clause for output first ensures that the continuation $P$ has an acceptable analysis estimate and it then requires that all the channels that $y$ may be bound to also are recorded in the communication cache for all the channels that $x$ may evaluate to. Turning to input it first ensures that the continuation has an acceptable analysis estimate and then it requires that the set of channels possibly bound to $y$ includes those that may be communicated over all the channels that $x$ may evaluate to. The clause for channel creation just requires that the subprocess has an acceptable analysis estimate. The final clause expresses that in order to have an acceptable analysis estimate for parallel composition we must have acceptable analysis estimates for the two subprocesses.

The above specification does not take into account that the semantics of the $\pi$-calculus allows $\alpha$-renaming of names and variables; we refer to [3] for how to deal with this.

## 4.4 Combining Paradigms

We shall conclude by illustrating how the above techniques can be combined for multi-paradigmatic languages. Our example language will be Concurrent ML [24] that is an extension of Standard ML [13] and that combines the functional, imperative and concurrent paradigms. We shall consider the following fragment:

$$\begin{array}{rl}
e ::= & c \mid x \mid \texttt{fn}\ x_0 \texttt{=>} e_0 \mid \texttt{fun}\ f\,x_0 \texttt{=>} e_0 \mid e_1\,e_2 \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \cdots \\
\mid & e_1 \texttt{:=} e_2 \mid e_1; e_2 \mid \texttt{let}\ x\ \texttt{=ref}_r\ \texttt{in}\ e \mid \texttt{deref}\ e \mid \cdots \\
\mid & \texttt{send}\ e_1\ \texttt{on}\ e_2 \mid \texttt{receive}\ e \mid \texttt{let}\ x\ \texttt{=chan}_n\ \texttt{in}\ e \mid \texttt{spawn}\ e \mid \cdots
\end{array}$$

The functional paradigm is represented by the first line where we have an extension of the $\lambda$-calculus with constructs for defining recursive functions and conditional expressions. The imperative constructs extend those of Subsection 4.1 by adding constructs for the dynamic creation of reference cells and for dereferencing a reference cell. The third line adds concurrency constructs similar to those of Subsection 4.3 except that parallelism is introduced by an explicit `spawn`-construct. The labels $r \in \mathbf{Ref}$ and $n \in \mathbf{Ch}$ denote program points and are merely introduced to aid the analysis; they will be used to refer to the reference cells and channels created at the particular program points.

**The analysis.** As in the previous examples the aim will be to estimate which values an expression may evaluate to. The judgements have the form

$$(\hat\rho, \widehat\sigma, \widehat\kappa) \models e : \hat v$$

and expresses the acceptability of the analysis result $\hat v$ in the context of $\hat\rho$, $\widehat\sigma$ and $\widehat\kappa$. The abstract environment $\hat\rho : \mathbf{Var} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be bound to the variables, the abstract state $\widehat\sigma : \mathbf{Ref} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be bound to the reference cells and the mapping $\widehat\kappa : \mathbf{Ch} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be communicated over the channels. In addition to the constant $\diamond$ and the abstract closures $\{\!|\mathtt{fn}\ x_0 \,\mathtt{=>}\, e_0|\!\}$, the abstract values can now also be recursive abstract closures $\{\!|\mathtt{fun}\, f\, x_0 \,\mathtt{=>}\, e_0|\!\}$, reference cells $r$ and channels $n$.

For the functional constructs we adapt the clauses of Subsection 2.1:

$$\begin{array}{rcl}
(\hat\rho,\widehat\sigma,\widehat\kappa) \models c : \hat v & \text{iff} & \diamond \in \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models x : \hat v & \text{iff} & \hat\rho(x) \subseteq \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \mathtt{fn}\ x_0 \,\mathtt{=>}\, e_0 : \hat v & \text{iff} & \{\!|\mathtt{fn}\ x_0 \,\mathtt{=>}\, e_0|\!\} \in \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \mathtt{fun}\, f\, x_0 \,\mathtt{=>}\, e_0 : \hat v & \text{iff} & \{\!|\mathtt{fun}\, f\, x_0 \,\mathtt{=>}\, e_0|\!\} \in \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1\, e_2 : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 : \hat v_1 \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_2 : \hat v_2 \ \wedge \\
 & & \forall \{\!|\mathtt{fn}\ x_0 \,\mathtt{=>}\, e_0|\!\} \in \hat v_1 : \hat v_2 \neq \emptyset \Rightarrow \\
 & & \qquad [\hat v_2 \subseteq \hat\rho(x_0) \wedge (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_0 : \hat v_0 \wedge \hat v_0 \subseteq \hat v] \wedge \\
 & & \forall \{\!|\mathtt{fun}\, f\, x_0 \,\mathtt{=>}\, e_0|\!\} \in \hat v_1 : \hat v_2 \neq \emptyset \Rightarrow \\
 & & \qquad [\{\!|\mathtt{fun}\, f\, x_0 \,\mathtt{=>}\, e_0|\!\} \in \hat\rho(f) \wedge \hat v_2 \subseteq \hat\rho(x_0) \wedge \\
 & & \qquad\qquad (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_0 : \hat v_0 \wedge \hat v_0 \subseteq \hat v]
\end{array}$$

$$\begin{array}{rcl}
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_0 : \hat v_0 \wedge \\
 & & \diamond \in \hat v_0 \Rightarrow [(\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 : \hat v_1 \wedge \hat v_1 \sqsubseteq \hat v \wedge \\
 & & \qquad (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_2 : \hat v_2 \wedge \hat v_2 \sqsubseteq \hat v]
\end{array}$$

Recursion is handled much as ordinary function abstraction and the clause for application is extended to take care of recursion as well. The construct for conditional only requires that the branches have acceptable analysis estimates if the test could evaluate to a constant. Note that $\widehat\sigma$ and $\widehat\kappa$ play no role in this part of the specification.

For the imperative constructs we follow the approach of the coarser analysis of Subsection 4.1 and take:

$$\begin{array}{rcl}
(\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 \,\mathtt{:=}\, e_2 : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 : \hat v_1 \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_2 : \hat v_2 \ \wedge \\
 & & \diamond \in \hat v \ \wedge\ \forall r \in \hat v_1 : \hat v_2 \subseteq \widehat\sigma(r) \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \mathtt{deref}\ e : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e : \hat v' \ \wedge\ \forall r \in \hat v' : \widehat\sigma(r) \subseteq \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \mathtt{let}\ x \,\mathtt{=}\, \mathtt{ref}_r\ \mathtt{in}\ e : \hat v & \text{iff} & r \in \hat\rho(x) \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e : \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 ; e_2 : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 : \hat v' \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_2 : \hat v
\end{array}$$

The first clause reflects that an assignment in Standard ML evaluates to a dummy constant. In the second clause we consult the abstract state to determine the abstract values that the reference cell may contain. The `let`-construct evaluates to its body so we only ensure that the new reference cell (called $r$) is a potential value for the variable $x$. The final clause records that $e_1; e_2$ evaluates to the value of $e_2$.

Finally, for the concurrency constructs we take:

$$\begin{array}{rcl}
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \texttt{send}\ e_1\ \texttt{on}\ e_2 : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_1 : \hat v_1 \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_2 : \hat v_2 \ \wedge \\
 & & \diamond \in \hat v \ \wedge\ \forall n \in \hat v_2 : \hat v_1 \subseteq \widehat\kappa(n) \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \texttt{receive}\ e : \hat v & \text{iff} & (\hat\rho,\widehat\sigma,\widehat\kappa) \models e : \hat v' \ \wedge\ \forall n \in \hat v' : \widehat\kappa(n) \subseteq \hat v
\end{array}$$

$$\begin{array}{rcl}
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \texttt{let}\ x\ \texttt{=}\,\texttt{chan}_n\ \texttt{in}\ e : \hat v & \text{iff} & n \in \hat\rho(x) \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e : \hat v \\
(\hat\rho,\widehat\sigma,\widehat\kappa) \models \texttt{spawn}\ e : \hat v & \text{iff} & \diamond \in \hat v \ \wedge\ (\hat\rho,\widehat\sigma,\widehat\kappa) \models e : \hat v' \ \wedge \\
 & & \forall \{\!|\texttt{fn}\ x_0 \texttt{=>} e_0|\!\} \in \hat v' : \\
 & & \qquad \diamond \subseteq \hat\rho(x_0) \wedge (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_0 : \hat v_0 \\
 & & \forall \{\!|\texttt{fun}\ f\, x_0 \texttt{=>} e_0|\!\} \in \hat v' : \\
 & & \qquad \{\!|\texttt{fun}\ f\, x_0 \texttt{=>} e_0|\!\} \in \hat\rho(f) \wedge \diamond \subseteq \hat\rho(x_0) \wedge \\
 & & \qquad\qquad (\hat\rho,\widehat\sigma,\widehat\kappa) \models e_0 : \hat v_0
\end{array}$$

The first clause reflects that the abstract values of $e_1$ represent potential values to be communicated over the channels that $e_2$ may evaluate to; the `send`-construct itself evaluates to a dummy constant. The second clause then records that the `receive`-construct may evaluate to the abstract values that may be communicated over the channels that $e$ evaluates to. The `let`-construct is analogous to the `let`-construct for reference cells. The `spawn`-construct expects a function as parameter and, when executed, it will bind a dummy constant to the formal parameter and spawn a process for executing the body. In the analysis this is recorded using the same technique as we introduced for function application.

In [14,21,17] we present more precise analyses of the functional and imperative subset of the language and in [8] we give a more precise treatment of the functional and concurrent subset of the language.

## 5 Conclusion

We have shown that the flow logic approach applies to a variety of programming paradigms. The ease with which this can be done is mainly due to the *high abstraction level* supported by the approach: there is a clear separation between specifying an analysis and implementing it. Another important aspect is that even at the specification level one can combine different styles of specification for different aspects of the language and different aspects of the analysis.

The flow logic approach is also very flexible when it comes to integrating state-of-the-art techniques developed from data flow analysis, constraint-based analysis and abstract interpretation. In [14] we show how the approach of abstract interpretation can be combined with a constraint-based formulation of a control flow analysis for a functional language and the relationship to a number of standard approaches like the $k$-CFA approach is clarified. This work is carried one step further in [17,21] where we extend the language to have functional as well as imperative constructs and show how to integrate classical techniques for interprocedural data flow analysis (call strings and assumption sets) into the specification; we also show how the standard technique of reference counts can be incorporated.

In the context of the ambient calculus [5] we have shown how powerful abstraction techniques based on tree grammars can be used in conjunction with flow logic [22] and in [20] we show how the three valued logic approach originally developed for analysing pointers in an imperative language [25] can be integrated with the flow logic approach.

**Acknowledgement.** The paper was written while the authors were visiting Universität des Saarlandes and Max-Planck Institute für Informatik, Saarbrücken, Germany.

## References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. TIC'98*, 1998.
3. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the $\pi$-calculus with applications to security. *Information and Computation*, to appear, 2000.
4. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. PaCT'01*, number 2127 in Lecture Notes in Computer Science, pages 27–41. Springer-Verlag, 2001.
5. L. Cardelli and A. Gordon. Mobile ambients. In *Proc. FoSSaCS'98*, 1998.
6. D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *Proc. L & FP*, 1986.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
8. K.L.S. Gasser, F. Nielson, and H. Riis Nielson. Systematic realisation of control flow analyses for CML. In *Proc. ICFP'97*, pages 38–51. ACM Press, 1997.
9. M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.
10. N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
13. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
14. F. Nielson and H. Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.
15. F. Nielson and H. Riis Nielson. The flow logic of imperative objects. In *Proc. MFCS'98*, number 1450 in Lecture Notes in Computer Science, pages 220–228. Springer-Verlag, 1998.

16. F. Nielson and H. Riis Nielson. Flow logics and operational semantics. *Electronic Notes of Theoretical Computer Science*, 10, 1998.
17. F. Nielson and H. Riis Nielson. Interprocedural control flow analysis. In *Proc. ESOP'99*, number 1576 in Lecture Notes in Computer Science, pages 20–39. Springer-Verlag, 1999.
18. F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
19. F. Nielson, H. Riis Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *Proc. CONCUR'99*, number 1664 in Lecture Notes in Computer Science, pages 463–477. Springer-Verlag, 1999.
20. F. Nielson, H. Riis Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In *Proc. ESOP'00*, number 1782 in Lecture Notes in Computer Science, pages 304–319. Springer-Verlag, 2000.
21. H. Riis Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC'98*, number 1383 in Lecture Notes in Computer Science, pages 109–127. Springer-Verlag, 1998.
22. H. Riis Nielson and F. Nielson. Shape analysis for mobile ambients. In *Proc. POPL'00*, pages 142–154. ACM Press, 2000.
23. G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
24. J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL'99*, 1999.

# Structure-Preserving Binary Relations for Program Abstraction

David A. Schmidt[⋆]

Computing and Information Sciences Department
Kansas State University
Manhattan, KS 66506 USA.

**Abstract.** An *abstraction* is a property-preserving contraction of a program's model into a smaller one that is suitable for automated analysis. An abstraction must be sound, and ideally, complete. Soundness and completeness arguments are intimately connected to the abstraction process, and approaches based on homomorphisms and Galois connections are commonly employed to define abstractions and prove their soundness and completeness.
This paper develops Mycroft and Jones's proprosal that an abstraction should be stated as a form of structure-preserving binary relation. Mycroft-Jones-style relations are defined, developed, and employed in characterizations of the homomorphism and Galois-connection approaches to abstraction.

## 1 Introduction

In program analysis (*static analysis*) [6]), the term, "abstraction," describes a property-preserving contraction of a program's model into one that is smaller and more suitable for automated analysis. Abstraction-generated models are used by compilers to perform data-flow analysis [1, 16], by type-checkers to verify type safety [29], and by model checkers to validate safety properties of systems of processes [3].

An abstraction must be *sound*, in the sense that properties proved true of the abstracted model must hold true of the original program model. Ideally, the abstract model should be most precise, or *complete*, with regards to the properties of interest. Soundness and completeness arguments are intimately connected to the abstraction process.

Our intuitions tell us that a program's model is a kind of algebra, and the abstraction process is a kind of homomorphism, and indeed, the *homomorphism approach* provides a simple means of devising sound abstractions [4].

A crucial insight from the abstract interpretation research of Cousot and Cousot [7, 8, 9] is that an abstraction can be defined in terms of a homomorphism and its "inverse" function, giving a *Galois connection.* The inverse function gives us a tool for arguing the completeness of an abstraction.

[⋆] Supported by NSF CCR-9970679, INT-9981558, ITR-0085949, and ITR-0086154.

Galois connections possess a rich collection of properties that facilitate correctness proofs of static analyses. But as Mycroft and Jones noted in a seminal paper in 1985 [26], there is a price to be paid for possessing these properties: Program models employing data of compound and higher-order types require Galois connections that operate on powerdomains of overwhelming complexity. Mycroft and Jones suggest that an alternative framework, based on binary relations, can be used to avoid much of this complexity.[1] Then, reasoning techniques based on logical relations [24, 27, 28, 31, 32] apply to the compound and higher-typed values.

The intuition behind Mycroft and Jones's proposal is simple: Given the set of source program states, $C$, and the set of abstract program states, $A$, define a binary relation, $\mathcal{R} \subseteq C \times A$, based on the intuition that $c \; \mathcal{R} \; a$ if $a$ is an acceptable modelling of $c$ in the abstract model. For example, for a type-checking abstraction, where concrete data values are abstracted to data-type tags, we might assert that $3 \, \mathcal{R} \, int$ and also $3 \, \mathcal{R} \, real$, but not $3 \, \mathcal{R} \, bool$. In practice, it is relation $\mathcal{R}$ that is crucial to a proof of safety of abstraction.

But this seems too simple—can the homomorphism and Galois-connection approaches be discarded so readily? And can *any* binary relation between $C$ and $A$ define a meaningful property for a static analysis? Mycroft and Jones state that a useful binary relation, $\mathcal{R} \subseteq C \times A$, must possess *LU-closure*: For all $c, c' \in C$, $a, a' \in A$,

$$c' \sqsubseteq_C c, \; c \, \mathcal{R} \, a, \text{ and } a \sqsubseteq_A a' \text{ imply } c' \, \mathcal{R} \, a'$$

But is this strong enough for proving soundness and completeness? Is it too strong? Does LU-closure characterize homomorphisms or Galois connections?

This paper addresses these and other fundamental questions regarding the homomorphism, Galois connection, and binary-relations approaches to abstraction, relating all three in terms of "structure preserving" binary relations.

The paper's outline goes as follows: Section 2 defines the format of program model—it is a *Kripke structure*, which is an automaton whose states possess local properties of interest that can be preserved or discarded by abstraction. To develop intuition, the Section gives examples of models and abstractions based on Kripke structures.

Next, Section 4 demonstrates how to employ homomorphisms to abstract one Kripke structure by another. The underlying notion of *simulation* is used first to explain the homomorphism approach and next to relate homomorphisms to Galois connections, which are developed in Section 6. Applications of Galois connections to completeness proofs and synthesis of abstract Kripke structures are developed in detail.

Finally, Section 7 introduces Mycroft-Jones-style binary relations and lists crucial structure-preserving properties. These properties are used to state equivalences between structure-preserving forms of binary relations, homomorphisms, and Galois connections.

[1] And in response, Cousot and Cousot proposed a variant, *higher-order abstract interpretation* [10], that sidesteps most of the powerdomain machinery.

$$\mathcal{K} = \langle \{s_0, s_1, s_2\}, \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}, \mathcal{I}_K \rangle$$
$$\text{where } \mathcal{I}_K(s_0) = \{\}, \mathcal{I}_K(s_1) = \{own\}, \mathcal{I}_K(s_2) = \{own, use\}$$

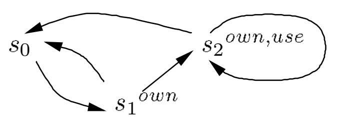


**Fig. 1.** Kripke structure of a process that uses a resource

## 2 Kripke Structures

We represent a program model as a *Kripke structure.* Simply stated, a Kripke structure is a finite-state automaton whose states are labelled with atomic, primitive "properties" that "hold true" at the states [3, 25]. Kripke structures neatly present program models based on operational semantics, such as flowcharts, data-flow frameworks [16], statecharts [14], and process-algebra processes [23].

**Definition 1.** *1. A* state-transition system *is a pair, $\langle \Sigma_K, \longrightarrow_K \rangle$, where*

*(a) $\Sigma_K$ is a set of states.*

*(b) $\longrightarrow_K \subseteq \Sigma_K \times \Sigma_K$ is the transition relation. We write $s \longrightarrow s'$ to denote $(s, s') \in \longrightarrow_K$.*

*2. A state-transition system is* labelled *if its second component is revised to $\longrightarrow_K \subseteq \Sigma_K \times \mathcal{L} \times \Sigma_K$, where $\mathcal{L}$ is some fixed, finite set. We write $s \xrightarrow{\ell} s'$ to denote $(s, \ell, s') \in \longrightarrow_K$.*

*3. A state-transition system is a* Kripke structure *if it is extended to a triple, $\langle \Sigma_K, \longrightarrow_K, \mathcal{I}_K \rangle$, where $\mathcal{I}_K : \Sigma_K \to \mathcal{P}(Atom)$ associates a set of atomic properties, $\mathcal{I}_K(s) \subseteq Atom$, to each $s \in \Sigma_K$, for some fixed, finite set, Atom.*

In this paper, we use (unlabelled) Kripke structures; Figure 1 presents a process that requests ownership of and uses a resource. The Kripke structure is presented as an automaton whose states are labelled by atomic properties from $Atom = \{use, own\}$.

A program's operational semantics can also be modelled as a Kripke structure; Figure 2 presents an infinite-state structure that results from a program that computes upon an input, `x`. A state, $(p, n)$, remembers the value of variable `x` on entry to program point, $p$. The program's transfer functions are coded as the transition relation.

*Atom* lists the properties of interest to the model. Here, it is helpful to visualize each $p \in Atom$ as naming some $S_p \subseteq \Sigma_K$ such that $p \in \mathcal{I}_K(s)$ iff $s \in S_p$, e.g., $isEven = \{(p, 2n) \mid n \geq 0\}$. Note that $\mathcal{P}(Atom)$ contains all possible combinations of the properties of interest—when we abstract the Kripke structure, we will use a subset of $\mathcal{P}(Atom)$ as the state space for an abstract model.

```
p0 : while (true) do
   p1 : if isEven(x)
      p2 : then x:= x div2
      p3 : else x:= x + 1
od
```

$\mathcal{K} = \langle \Sigma_K, \rightarrow_K, \mathcal{I}_K \rangle$
where $\Sigma_K = \mathit{ProgramPoint} \times \mathit{Nat}$
$\mathit{ProgramPoint} = \{p_0, p_1, p_2, p_3\}$
$\rightarrow_K = \{\ p_0, n \longrightarrow p_1, n,$
$p_1, 2n \longrightarrow p_2, 2n,$
$p_1, 2n+1 \longrightarrow p_3, 2n+1,$
$p_2, n \longrightarrow p_1, n/2,$
$p_3, n \longrightarrow p_1, n+1$
$\mid\ n \in \mathit{Nat}\}$
$\mathit{Atom} = \{\mathit{at}(p) \mid p \in \mathit{ProgramPoint}\}$
$\cup \{\mathit{isEven}, \mathit{isOdd}\}$
$\mathcal{I}_K(p, 2n) = \{\mathit{at}(p), \mathit{isEven}\}$
$\mathcal{I}_K(p, 2n+1) = \{\mathit{at}(p), \mathit{isOdd}\}$

**Fig. 2.** Infinite-state Kripke structure of a sequential program

$$\begin{tikzcd}
 & p_0 \arrow[d] & \\
 & p_1 \arrow[dl] \arrow[dr] & \\
p_2 \arrow[ur, bend left] & & p_3 \arrow[ul, bend right]
\end{tikzcd}$$

$\mathcal{I}_{A_1}(p_i) = \{\mathit{at}(p_i)\}$

$$\begin{tikzcd}
p_0, e \arrow[d] & p_0, o \arrow[d] \\
p_1, e \arrow[d] & p_1, o \arrow[d] \\
p_2, e \arrow[u, bend left] \arrow[ur] & p_3, o \arrow[ul]
\end{tikzcd}$$

$\mathcal{I}_{A_2}(p_i, e) = \{\mathit{at}(p_i), \mathit{isEven}\}$,
$\mathcal{I}_{A_2}(p_i, o) = \{\mathit{at}(p_i), \mathit{isOdd}\}$

**Fig. 3.** Two abstractions of a program

**Abstractions of a Kripke Structure** Because of its infinite state set, the structure defined in Figure 2 cannot be conveniently drawn pictorially nor can it be easily analyzed mechanically. We must abstract the structure into a manageably sized, finite-state Kripke structure, using the mapping, $\mathcal{I}_K$, to guide us. Figure 3 shows two possible abstractions, represented pictorially. The first abstraction, called a *control abstraction*, defines $\Sigma_{A_1} = \mathit{ProgramPoint}$—data values are forgotten, as are the properties, *isEven* and *isOdd*. Control abstraction generates the control-flow graph used by a compiler.

The second abstraction is a mixture of data and control abstraction: It simplifies the program's data domain (that is, x's value) to $\mathit{EvenOdd} = \{e, o\}$ and defines $\Sigma_{A_2} = \mathit{ProgramPoint} \times \mathit{EvenOdd}$. In consequence, more precise knowledge is presented about the outcomes of the test at program point $p_1$ (cf. [38]) in contrast to a control abstraction.

The state sets for both abstractions in Figure 3 are just subsets of $\mathcal{P}(\mathit{Atom})$ from Figure 2: $\Sigma_{A_2} = \mathcal{I}_K(\Sigma_K)$, and $\Sigma_{A_1} = \mathcal{I}_K(\Sigma_K) \setminus \{\mathit{isEven}, \mathit{isOdd}\}$, where $X \setminus Y$ denotes the sets of $X \subseteq \mathcal{P}(\mathit{Atom})$ with occurrences of $a \in Y$ removed.

The characterization of each $a \in \Sigma_{A_i}$ as $a \subseteq \mathit{Atom}$ induces the abstract structure's property map—it is simply, $\mathcal{I}_{A_i}(a) = a$.

**Derivation of Abstract Transitions** Given the abstract state set, $\Sigma_A$, we must derive a transition relation, $\rightarrow_A$; we begin with examples for needed intuition.

The intuition behind the second structure in Figure 3 is that *Nat*, the set of natural numbers, was abstracted to $\mathit{EvenOdd} = \{e, o\}$, such that even-valued naturals are represented by $e$ and odd-valued naturals are represented by $o$. The addition and division-by-two operations, encoded in the transition relation in Figure 2, must be restated to operate on the new set. By necessity, the semantics of division-by-two is nondeterministic:

$$\begin{array}{lll} e+1 \mapsto o & e/2 \mapsto e & o/2 \mapsto e \\ o+1 \mapsto e & e/2 \mapsto o & o/2 \mapsto o \end{array}$$

The operations above generate $\rightarrow_{A_2}$—in particular, we have these crucial transition rules for the decision point at $p_1$:

$$\begin{array}{l} p_1, e \longrightarrow p_2, e \\ p_1, o \longrightarrow p_3, o \end{array}$$

The rules retain information about the properties of the data that arrive at program points $p_2$ and $p_3$.

The above abstraction generates a finite set of states for the Kripke structure in Figure 2. But the finite cardinality might be too huge for practical analysis, and a more severe abstraction might be required. For this, one can partially order the abstract-data domain—the partial ordering helps us merge similar abstract states and so reduce the cardinality of the state set. For example, we might add the data values, $\bot$ ("no value at all") and $\top$ ("any value at all") to the *EvenOdd* set and partially order the result as follows:

$$\mathit{EvenOdd}_\bot^\top = \begin{tikzcd} & \top & \\ e \arrow[ur, dash] & & o \arrow[ul, dash] \\ & \bot \arrow[ul, dash] \arrow[ur, dash] & \end{tikzcd}$$

The ordering suggests precision-of-information-content—e.g., $e$ is more precise than $\top$ about the range of numbers it represents. (To rationalize this explanation, $\bot$ is the most "precise" of all, because it precisely notes no value at all.) This particular partial ordering is no accident: It comes from our belief that abstract values and abstract states are just sets of properties. Here, $\mathit{EvenOdd}_\bot^\top$ is isomorphic to $\mathcal{P}\{\mathit{isEven}, \mathit{isOdd}\}$, ordered by superset inclusion.

The arithmetic operations must be revised for the new domain:

$$\begin{array}{ll} e/2 \mapsto \top & e+1 \mapsto o \\ o/2 \mapsto \top & o+1 \mapsto e \\ \top/2 \mapsto \top & \top+1 \mapsto \top \end{array}$$

$$\begin{tikzcd}
p_0, \top \arrow[d] & \\
p_1, \top \arrow[r] \arrow[d] & p_3, o \arrow[d] \\
p_2, e \arrow[u, bend left] & p_1, e \arrow[l]
\end{tikzcd}$$

$\mathcal{I}_{A_3}(p_i, e) = \{at(p_i), isEven\}$
$\mathcal{I}_{A_3}(p_i, o) = \{at(p_i), isOdd\}$
$\mathcal{I}_{A_3}(p_i, \top) = \{at(p_i)\}$

**Fig. 4.** Kripke structure generated with partially ordered data domain

(Operations on $\bot$ are ignored.) In particular, division by 2 produces $\top$ as its result. Now, program states have the form, *ProgramPoint* $\times$ $\mathit{EvenOdd}_{\bot}^{\top}$, where $p, \bot$ denotes a program point that is never reached during execution ("dead code") and $p, \top$ represents a program point that may be reached with both even- and odd-valued numbers. The presence of $\top$ suggests these transition rules for the program's decision point, $p_1$:[2]

$$\begin{aligned} p_1, \top &\longrightarrow p_2, e \\ p_1, \top &\longrightarrow p_3, o \end{aligned}$$

We use $\mathit{EvenOdd}_{\bot}^{\top}$ with the above transition rules for $p_1$ to generate a new Kripke structure for the example in Figure 2; see Figure 4. The resulting structure has one fewer state than its counterpart in Figure 3.

**State Merging** We can generate a smaller structure from Figure 4 by merging the two states, $(p_1, e)$ and $(p_1, \top)$. This merge yields the result state, $(p_1, \top)$, because $e \sqcup \top = \top$ in the ordering on the data values. The transitions to and from $(p_1, e)$ are "folded" into $(p_1, \top)$, and the resulting structure looks like this:

$$\begin{tikzcd}
 & p_0, \top \arrow[d] & \\
 & p_1, \top \arrow[dl] \arrow[dr] & \\
p_2, e \arrow[ur, bend left] & & p_3, o \arrow[ul, bend right]
\end{tikzcd}$$

In data-flow analysis, it is common to perform state-merging "on the fly" while generating the Kripke structure: $\Sigma_K$ is defined as a subset of *ProgramPoint* $\times$ *DataFlowInformation* such that, if $(p_k, d_1) \in \Sigma_K$ and $(p_k, d_2) \in \Sigma_K$, then $d_1 = d_2$; that is, there is at most one state per program point. When the Kripke structure is generated by executing the source program with the data domain, *DataFlowInformation*, newly generated states of form $(p_k, d_i)$ are merged with the existing state, $(p_k, d)$, giving $(p_k, d_i \sqcup d)$.

[2] The following transition rules for $p_1$ would also be acceptable, but they lose too much precision: $p_1, \top \longrightarrow p_2, \top$ and $p_1, \top \longrightarrow p_3, \top$.

**Monotone Kripke Structures** The structure in Figure 4 has "related" states: $(p_1, \top)$ and $(p_1, e)$ are related because $e \sqsubseteq \top$. To formalize this, we partially order the state set, $\Sigma_{A_3}$, such that $p_i, d_i \sqsubseteq_{A_3} p_j, d_j$ iff $p_i = p_j$ and $d_i \sqsubseteq_{EvenOdd_\perp^\top} d_j$.

An expected consequence of this relationship should be that, if $p_1, e \longrightarrow q$ then also $p_1, \top \longrightarrow q'$, where $q \sqsubseteq_{A_3} q'$—the less precise state should be consistent with the more precise one. Similar thinking causes us to demand, if $p_1, e \sqsubseteq_{A_3} p_1, \top$, then $\mathcal{I}_{A_3}(p_1, e) \supseteq \mathcal{I}_{A_3}(p_1, \top)$. These are *monotonicity* properties:

**Definition 2.** *For Kripke structure, $\mathcal{K} = \langle \Sigma_K, \rightarrow_K, \mathcal{I}_K \rangle$ such that $\Sigma_K$ is partially ordered by $\sqsubseteq_K \subseteq \Sigma_K \times \Sigma_K$,*

1. *$\rightarrow_K$ is* monotone (with respect to $\sqsubseteq_K$) *iff for all $s, s' \in \Sigma_K$, $s \sqsubseteq_K s'$ and $s \longrightarrow s_1$ imply there exists $s_2 \in \Sigma_K$ such that $s' \longrightarrow s_2$ and $s_1 \sqsubseteq_K s_2$:*

$$\begin{tikzcd} s \arrow[r, phantom, "\sqsubseteq_K"] \arrow[d] & s' \arrow[d] \\ s_1 \arrow[r, phantom, "\sqsubseteq_K"] & s_2 \end{tikzcd}$$

2. *$\mathcal{I}_K$ is* monotone (with respect to $\sqsubseteq_K$) *iff $s \sqsubseteq_K s'$ implies $\mathcal{I}_K(s) \supseteq \mathcal{I}_K(s')$.*

*A Kripke structure whose state set is partially ordered is* monotone *if both its transition relation and property map are monotone.*

Since a Kripke structure's transitions encode a program's transfer functions, it is natural to demand that the transition relation be monotone. The property map must be monotone to ensure that the partial ordering on states is sensible. We work exclusively with monotone Kripke structures—the monotonicity is crucial to proofs of soundness and completeness of abstractions.

## 3 Relating Models

Abstraction must relate a "detailed" Kripke structure with a "less detailed" counterpart. We call the detailed Kripke structure the *concrete structure* and name it $\mathcal{C}$, and we call the less detailed structure the *abstract structure* and name it $\mathcal{A}$. When we compare two Kripke structures, $\mathcal{C}$ and $\mathcal{A}$, *we assume they use the same set, Atom, as the codomain of their respective property maps.* This facilities easy definitions of soundness and completeness.

## 4 Homomorphisms

The classic tool for relating one algebraic structure to another is the homomorphism; here is the variant for Kripke structures:

**Definition 3.** *For Kripke structures $\mathcal{C} = \langle \Sigma_C, \rightarrow_C, \mathcal{I}_C \rangle$ and $\mathcal{A} = \langle \Sigma_A, \rightarrow_A, \mathcal{I}_A \rangle$, a* homomorphism from $\mathcal{C}$ to $\mathcal{A}$ *is a function, $h : \Sigma_C \rightarrow \Sigma_A$, such that*

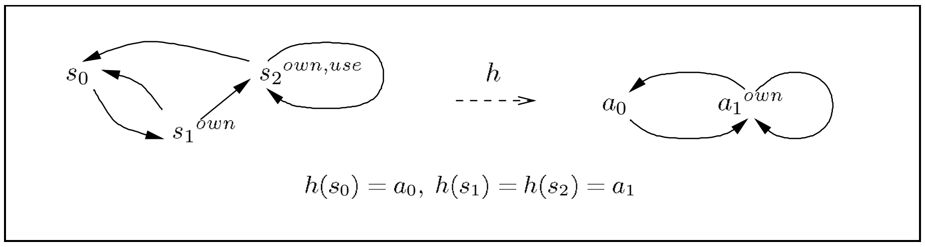


**Fig. 5.** Example homomorphism

1. *for all $c \in \Sigma_C$, $c \longrightarrow c'$ implies there exists a transition, $h(c) \longrightarrow a'$ such that $h(c') \sqsubseteq_A a'$:*

$$\begin{tikzcd} c \arrow[r, dotted, "h"] \arrow[d] & h(c) \arrow[d] \\ c' \arrow[r, dotted, "h"] & h(c') \sqsubseteq_A a' \end{tikzcd}$$

2. *for all $c \in \Sigma_C$, $\mathcal{I}_C(c) \supseteq \mathcal{I}_A(h(c))$.*

Figure 5 displays a simple example of two structures and their relationship by homomorphism. The Figure shows a homomorphism that preserves transitions up to equality. But the Definition does not require this—$h(c') \sqsubseteq a'$ is stated, rather than $h(c') = a'$, to handle the situation where $\Sigma_A$ is nontrivially partially ordered. For example, reconsider Figures 2 and 4, where $\Sigma_C$ is $ProgramPoint \times Nat$, $\Sigma_A$ is $ProgramPoint \times EvenOdd_\bot^\top$, and consider the transition, `x := x div 2`:

$$\begin{tikzcd} p_2, 6 \arrow[r, dotted, "h"] \arrow[d, "\texttt{x:= x div2}"'] & p_2, e \arrow[d, "\texttt{x:= x div2}"] \\ p_1, 3 \arrow[r, dotted, "h"] & p_1, o \sqsubseteq p_1, \top \end{tikzcd}$$

where $h(pgmpoint, n) = (pgmpoint, polarity(n))$ and $polarity(2n) = e$ and $polarity(2n+1) = o$.

Given a concrete (monotone) Kripke structure, $\mathcal{C} = \langle \Sigma_C, \rightarrow_C, \mathcal{I}_C \rangle$, we can use its property map, $\mathcal{I}_C : \Sigma_C \rightarrow \mathcal{P}(Atom)$, as a homomophism onto the monotone Kripke structure, $A_{\mathcal{I}_C} = \langle \Sigma_A, \rightarrow_A, \mathcal{I}_A \rangle$:

$$\begin{aligned} &\Sigma_A = \mathcal{I}_C(\Sigma_C) \\ &\rightarrow_A = \{a \rightarrow_A \mathcal{I}_C(c') \mid c \rightarrow_C c',\ \ a \subseteq \mathcal{I}_C(c)\} \\ &\mathcal{I}_A(a) = a \end{aligned}$$

As suggested earlier, we use the range of $\mathcal{I}_C$ as the states for the abstract structure; the definition can be proved to be "complete" [3] for state set, $\mathcal{I}_C(\Sigma_C)$ [8, 12].

[3] The formalization of "complete" must wait until the section on Galois connections, where needed technical machinery is provided.

Standard uses of homomorphisms to relate concrete and abstract structures can be found in papers by Clarke, Grumberg, and Long [5] and Clarke, et al. [2], among others.

## 5 Simulations

A homomorphism between Kripke structures preserves the ability to transit between states. This notion is better known as a *simulation.* Let $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$ be a binary relation on the states of structures $\mathcal{C}$ and $\mathcal{A}$, and write $c\,\mathcal{R}\,a$ when $(c, a) \in \mathcal{R}$.

**Definition 4.** *For Kripke structures* $\mathcal{C} = \langle \Sigma_C, \rightarrow_C, \mathcal{I}_C \rangle$ *and* $\mathcal{A} = \langle \Sigma_A, \rightarrow_A, \mathcal{I}_A \rangle$, *a binary relation,* $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$, *is a* simulation *of* $\mathcal{C}$ *by* $\mathcal{A}$, *written* $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$, *iff for every* $c \in \Sigma_C$, $a \in \Sigma_A$, *if* $c\,\mathcal{R}\,a$, *then*

1. *if* $c \longrightarrow c'$, *then there exists* $a' \in \Sigma_A$ *such that* $a \longrightarrow a'$ *and* $c'\,\mathcal{R}\,a'$*:*

$$\begin{tikzcd} c \arrow[r, dotted, no head, "\mathcal{R}"] \arrow[d] & a \arrow[d] \\ c' \arrow[r, dotted, no head, "\mathcal{R}"] & a' \end{tikzcd}$$

2. $\mathcal{I}_C(c) \supseteq \mathcal{I}_A(a)$.

In the above situation, we also say that $\mathcal{A}$ *simulates* $\mathcal{C}$.[4] When $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$, then $c\,\mathcal{R}\,a$ asserts that $a$ is an acceptable abstraction of $c$—properties true for $a$ hold true for $c$, and $a$'s transitions mimick all transitions made by $c$. More generally, any sequence of transitions in $\mathcal{C}$ can be mimicked by a sequence in $\mathcal{A}$.[5]

If we reexamine Figure 5, we readily see the underlying simulation between the concrete and abstract structures: $\mathcal{R} = \{(s_0, a_0), (s_1, a_1), (s_2, a_1)\}$. And, when we study the program flowgraph in Figure 2 and its two even-odd abstractions in Figures 3 and 4, we see this simulation relates states in the concrete structure to those in the abstract structures:

$$\begin{aligned} \mathcal{R} = \{\, & ((p_i, 2n), (p_i, e)), \\ & ((p_i, 2n+1), (p_i, o)), \\ & ((p_i, n), (p_i, \top)) \;\mid\; i \geq 0, n \in \mathit{Nat}\} \end{aligned}$$

We demand that simulations are *left total*:

**Definition 5.** *A binary relation,* $\mathcal{R} \subseteq S \times T$, *is* left total *if for all* $s \in S$, *there exists some* $t \in T$ *such that* $s\,\mathcal{R}\,t$. *The relation is* right total *if for all* $t \in T$, *there exists some* $s \in S$ *such that* $s\,\mathcal{R}\,t$.

[4] If labelled transitions are employed, the simulation must respect the labels: $c\,\mathcal{R}\,a$ and $c \xrightarrow{\ell} c'$ implies that $a \xrightarrow{\ell} a'$ and $c'\,\mathcal{R}\,a'$.

[5] Thus, properties that hold true for all *paths* starting at $a$ also hold true for all paths starting at $c$. This supports sound verification of LTL- and ACTL-coded properties of temporal-logic on the abstract Kripke structure [4, 12, 34, 37].

A left-total simulation ensures that every state in the concrete structure can be modelled in the abstract structure. Right totality ensures that there are no superfluous abstract states.

Simulations play a crucial role in equivalence proofs of interpreters [20]: Each execution step of interpreter, $\mathcal{C}$, for a source-programming language is mimicked by a (sequence of) execution steps of an interpreter, $\mathcal{A}$, for the target programming language, and mathematical induction justifies that sequences of transitions taken by $\mathcal{C}$ can be mimicked by $\mathcal{A}$. Coinductive reasoning [23, 30] extends the proof to sequences of infinite length. More recently, correctness proofs of hardware circuits, protocols, and systems of processes has been undertaken by means of (bi)simulations [23].

It is easy to extract the simulation embedded within a homomorphism: For $h : \Sigma_C \rightarrow \Sigma_A$, define

$$c\,\mathcal{R}_h\, a \text{ iff } h(c) \sqsubseteq_A a$$

As before, $h(c) \sqsubseteq_A a$ is stated, rather than $h(c) = a$, to take into account the partial ordering, if any, upon $\Sigma_A$.

Since we can extract a useful simulation from a homomorphism, we might ask if the dual is possible: The answer is “no”—a homomorphism is a function, and it takes little work to invent simulation relations that are not functions.

It is possible to synthesize a simulation for two Kripke structures, $\mathcal{C}$ and $\mathcal{A}$, from scratch: Define this hierarchy of binary relations, $\mathcal{R}_i \subseteq \Sigma_C \times \Sigma_A$, for $i \geq 0$:

$$\begin{aligned}
\mathcal{R}_0 &= \Sigma_C \times \Sigma_A \\
\mathcal{R}_{i+1} &= \{(c,a) \mid \mathcal{I}_C(c) \supseteq \mathcal{I}_A(a), \text{ and also} \\
&\qquad\qquad \text{if } c \longrightarrow c', \text{then there exists } a' \in \Sigma_A \text{ such that} \\
&\qquad\qquad a \longrightarrow a' \text{ and } c'\ \mathcal{R}_i\ a'\}
\end{aligned}$$

$c\ \mathcal{R}_i\ a$ asserts that $a$ mimicks $c$ for up to $i$ transitions, so we define the limit relation, $\mathcal{R}_\infty = \bigcap_{i \geq 0} \mathcal{R}_i$. As shown by Park [30] and Milner [22], for finite-image Kripke structures $\mathcal{C}$ and $\mathcal{A}$, $\mathcal{R}_\infty$ defines a simulation, and indeed, it is the largest possible simulation on $\mathcal{C}$ and $\mathcal{A}$: If $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$, then $\mathcal{R} \subseteq \mathcal{R}_\infty$.

Alas, the synthesis technique cannot produce a left-total simulation when one is impossible: Consider $\mathcal{C} = \langle \{c_0, c_1\}, \{c_0 \longrightarrow c_1\}, \mathcal{I}_C \rangle$ and $\mathcal{A}\langle \{a_0\}, \{\}, \mathcal{I}_A \rangle$—we calculate $\mathcal{R}_\infty = \{(c_1, a_0)\}$, which means there is no way to abstract $c_0$ within $\mathcal{A}$—the abstract structure is unsuitable. In this fashion, the synthesis technique can be used to decide whether one finite-state structure can be simulated by another.

Simulations give the foundation for reasoning about the relationships between structures. We next consider a tool that lets us reason about “complete” simulations.

## 6 Galois Connections

Widely used for abstraction studies [7, 8, 17, 21], Galois connections are a form of “invertible homomorphism” that come with powerful techniques for analyzing the precision of abstractions.

**Definition 6.** *For partially ordered sets $P$ and $Q$, a* Galois connection *is a pair of functions, $(\alpha: P \to Q, \gamma: Q \to P)$, such that for all $p \in P$ and $q \in Q$, $p \sqsubseteq_P \gamma(q)$ iff $\alpha(p) \sqsubseteq_Q q$:*

$$\begin{tikzcd}
\gamma(q) & q \arrow[l, dotted, "\gamma"'] \\
p \arrow[u, phantom, sloped, "\sqsubseteq_P"] \arrow[r, dotted, "\alpha"] & \alpha(p) \arrow[u, phantom, sloped, "\sqsubseteq_Q"]
\end{tikzcd}$$

*Equivalently, $\alpha, \gamma$ form a Galois connection iff*

1. *$\alpha$ and $\gamma$ are monotone*
2. *$\alpha \circ \gamma \sqsubseteq id_Q$*
3. *$id_P \sqsubseteq \gamma \circ \alpha$.*

*We say that $\alpha$ is the* lower adjoint *and $\gamma$ is the* upper adjoint *of the Galois connection.*

A plethora of properties can be proved about Galois connections; here are a few:

**Theorem 7.** *Say that $(\alpha: P \to Q, \gamma: Q \to P)$ is a Galois connection:*

1. *The adjoints uniquely determine each other, that is, if $(\alpha, \gamma')$ is also a Galois connection, then $\gamma = \gamma'$ (similarly for $\gamma$, $\alpha$, and $\alpha'$).*
2. *$\alpha(p) = \sqcap \gamma^{-1}(\uparrow p)$ and $\gamma(q) = \sqcup \alpha^{-1}(\downarrow q)$, where $\uparrow p = \{p' \in P \mid p \sqsubseteq_P p'\}$ and $\downarrow q = \{q' \in Q \mid q' \sqsubseteq_Q q\}$*
3. *$\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$.*
4. *$\alpha$ is one-one iff $\gamma$ is onto iff $\gamma \circ \alpha = id_P$; $\gamma$ is one-one iff $\alpha$ is onto iff $\alpha \circ \gamma = id_Q$.*
5. *$\alpha[P]$ and $\gamma[Q]$ are isomorphic partially ordered sets.*
6. *$\alpha$ preserves all joins, and $\gamma$ preserves all meets*
7. *If $P$ and $Q$ are complete lattices, then so are $\alpha[P]$ and $\gamma[Q]$, but they need not be sublattices.*

Proofs of these results can be found in [21] as well as in many other sources.

Result (ii) implies that one can validate when a function $\alpha$ (dually, $\gamma$) is a component of a Galois connection by merely checking the wellformedness of the definition of its partner: $\alpha$ is the lower adjoint of a Galois connection iff $\alpha^{-1}(\downarrow q)$ is a principal ideal in $P$, for every $q \in Q$. (A *principal ideal* is a set, $S \subseteq P$, such that $S = \downarrow p$, for some $p \in P$.)

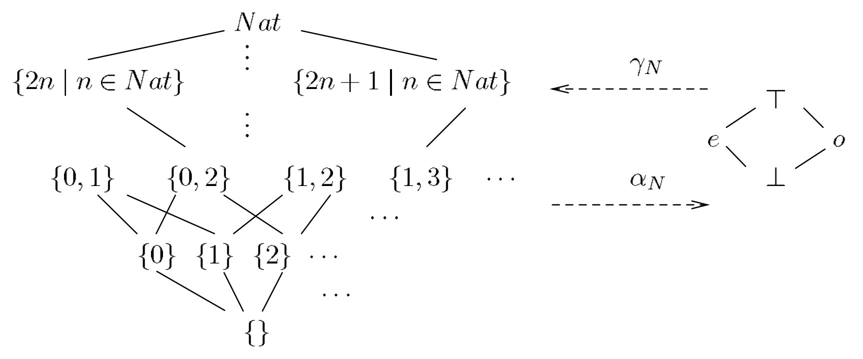
Say that an abstract Kripke structure, $\mathcal{A}$, has a partially ordered state set, $(\Sigma_A, \sqsubseteq_A)$. (For simplicity, say that it is a complete lattice.) We relate the concrete structure, $\mathcal{C}$, to $\mathcal{A}$, by means of a Galois connection of this form:[6]

$$\langle \alpha : (\mathcal{P}(\Sigma_C), \subseteq) \to (\Sigma_A, \sqsubseteq_A),\ \gamma : (\Sigma_A, \sqsubseteq_A) \to (\mathcal{P}(\Sigma_C), \subseteq) \rangle$$

[6] Assume that $\Sigma_C$ has no partial ordering of its own or that we ignore it [10].

We say that $\alpha$ is the *abstraction map* and $\gamma$ is the *concretization map* [7]—$\alpha(S)$ maps a set of states, $S \subseteq \Sigma_C$, to its most precise approximation in $\Sigma_A$—this is ideal for performing control abstraction, where several states must be merged into one. Of course, $\alpha\{c\}$ maps a single state, $c$, to its abstract counterpart, like a homomorphism would do. Dually, $\gamma(a)$ maps an abstract state to the set of concrete states that $a$ represents.

Figure 6 displays a Galois connection between the data values used in the structures in Figures 2 and 4. In the Figure, a Galois connection is first defined



A Galois connection between the above sets:

$$\begin{aligned}
&\alpha_N : \mathcal{P}(Nat) \to \mathit{EvenOdd}^{\top}_{\perp}\\
&\quad \alpha_N(S) = \bigsqcup(\{e \mid \text{exists } n \in Nat, 2n \in S\}\\
&\qquad\qquad\qquad \cup\{o \mid \text{exists } n \in Nat, 2n+1 \in S\})\\
&\gamma_N : \mathit{EvenOdd}^{\top}_{\perp} \to \mathcal{P}(Nat)\\
&\quad \gamma_N(a) = \{2n \mid n \in Nat, e \sqsubseteq a\} \cup \{2n+1 \mid n \in Nat, o \sqsubseteq a\}
\end{aligned}$$

Examples: $\alpha_N\{2,6\} = e, \alpha_N\{2,3,4\} = \top, \gamma_N(e) = \{0,2,4,\cdots\}, \gamma_N(\top) = Nat$.
Using $(\alpha_N, \gamma_N)$ to define a Galois connection between $\mathcal{P}(\Sigma_C)$ and $\Sigma_A$:

$$\begin{aligned}
&\Sigma_C = \mathit{ProgramPoint} \times Nat \qquad \Sigma_A = (\mathit{ProgramPoint} \times \mathit{EvenOdd}^{\top}_{\perp})^{\top}_{\perp}\\
&\alpha : \mathcal{P}(\Sigma_C) \to \Sigma_A\\
&\quad \alpha(T) = \begin{cases} \perp, & \text{if } T = \{\}\\ (p_k, \alpha_N(T \downarrow 2)), & \text{if } T \downarrow 1 = \{p_k\}\\ \top, & \text{if } p_i, p_j \in T \downarrow 1, i \neq j \end{cases}\\
&\quad \text{where } T \downarrow 1 = \{p \mid (p,n) \in T\}, \quad T \downarrow 2 = \{n \mid (p,n) \in T\}\\
&\gamma : \Sigma_A \to \mathcal{P}(\Sigma_C)\\
&\quad \gamma(p,a) = \{(p,n) \mid n \in \gamma_N(a)\}\\
&\quad \gamma(\perp) = \{\}\\
&\quad \gamma(\top) = \Sigma_C
\end{aligned}$$

**Fig. 6.** Example Galois connection

between the sets of data values, $\mathcal{P}(Nat)$ and $EvenOdd_{\bot}^{\top}$, used by the concrete and abstract structures, respectively. Then, this Galois connection is used to define a Galois connection between the state sets of the two structures. (To make a complete lattice, $\Sigma_A$ is augmented by $\bot$ and $\top$. As a technicality, we assume $(\top, \top) \in \rightarrow_A$.)

The Galois connection in the Figure makes clear that the abstract structure can remember at most one program point in its state. One might devise an abstract structure that remembers more than one program point at a time, e.g., $\Sigma_A = \mathcal{P}(ProgramPoint) \times EvenOdd_{\bot}^{\top}$, or better still, $\Sigma_A = \mathcal{P}(ProgramPoint \times EvenOdd_{\bot}^{\top})$. Such state sets would be useful for defining abstract structures that model nondeterministic computations.

**Extracting the Simulation within a Galois Connection** Thanks to Section 5, we know that we relate Kripke structures by simulations. Given the concrete Kripke structure, $\mathcal{C}$, an abstract structure, $\mathcal{A}$, and Galois connection, $(\alpha: \mathcal{P}(\Sigma_C) \rightarrow \Sigma_A, \gamma: \Sigma_A \rightarrow \mathcal{P}(\Sigma_C))$[7], we must prove a simulation exists, in the sense of Definition 4. To do this, we extract the following relation, $\mathcal{R}_\gamma \subseteq \Sigma_C \times \Sigma_A$, from the Galois connection:

$$c \, \mathcal{R}_\gamma \, a \text{ iff } c \in \gamma(a)$$

(Or, equivalently stated, $c \, \mathcal{R}_\gamma \, a$ iff $\alpha\{c\} \sqsubseteq_A a$.) For example, it is easy to prove that the relation extracted from the Galois connection in Figure 6 is a simulation.

**Synthesizing an Abstract Kripke Structure from a Galois Connection** In practice, we use a Galois connection to *synthesize* a "best" abstract Kripke structure for a concrete one: Say that we have $\mathcal{C} = \langle \Sigma_C, \rightarrow_C, \mathcal{I}_C \rangle$, and a complete lattice $A$, and a monotone property map, $\mathcal{I}_A : A \rightarrow \mathcal{P}(Atom)$.[8] We synthesize a Galois connection, and from it, an abstract Kripke structure that is sound (there exists a simulation) and complete (any other abstract structure that uses $A$ as its state set and simulates $\mathcal{C}$ computes less precise properties).

First, consider which element in $A$ may approximate some $c \in \Sigma_c$; the element must come from this set:

$$S_c = \{a' \in A \mid \mathcal{I}_C(c) \supseteq \mathcal{I}_A(a')\}$$

Define $\beta(c) = \sqcap S_c$; now, *if for every $c \in \Sigma_C$, $\beta(c)$ is an element of $S_c$, we can define a Galois connection* between $\mathcal{P}(\Sigma_C)$ and $A$ (c.f. Theorem 7(2)):

$$\begin{aligned} \alpha(S) &= \sqcup_{s \in S} \beta(s) \\ \gamma(a) &= \{c \mid \mathcal{I}_C(c) \supseteq \mathcal{I}_A(a)\} \end{aligned}$$

We use the Galois connection to define this transition relation on $A$:

$$\rightarrow_A = \{a \rightarrow_A \alpha\{c'\} \mid \text{there exists } c \rightarrow_C c' \text{ and } c \in \gamma(a)\}$$

[7] We omit the partial orderings to save space.

[8] Of course, if $A$ is a sublattice of $\mathcal{P}(Atom)$, we take $\mathcal{I}_A(a) = a$.

The abstract Kripke structure is defined as $\mathcal{A}_\gamma = \langle A, \rightarrow_A, \mathcal{I}_A \rangle$. The definition of $\rightarrow_A$ ensures that $\mathcal{A}_\gamma$ can simulate $\mathcal{C}$, that is, $\mathcal{C} \lhd_{\mathcal{R}_\gamma} \mathcal{A}_\gamma$, where $c\ \mathcal{R}_\gamma\ a$ iff $c \in \gamma(a)$, as usual. Note that $\mathcal{I}_C(c) \supseteq \mathcal{I}_A(a)$ iff $\beta(c) \sqsubseteq_A a$.

We can prove that $\mathcal{A}_\gamma$ is complete in the following sense: Say that there is another Kripke structure, $\mathcal{A}' = \langle A, \rightarrow'_A, \mathcal{I}_A \rangle$ such that $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}'$, where $c\ \mathcal{R}\ a$ iff $\mathcal{I}_C(c) \supseteq \mathcal{I}_A(a)$. *But* $\mathcal{R}$ *is just* $\mathcal{R}_\gamma$ . The Galois connection between $\mathcal{P}(\Sigma_C)$ and $A$ lets us prove that $\mathcal{A}_\gamma \lhd_{\mathcal{R}_{\sqsubseteq_A}} \mathcal{A}'$, where $a\ \mathcal{R}_{\sqsubseteq_A}\ a'$ iff $a \sqsubseteq_A a'$, implying $\mathcal{I}_A(a) \supseteq \mathcal{I}_A(a')$. Hence, $\mathcal{A}_\gamma$ is most precise.[9]

**Defining a Concrete Kripke Structure with a Galois Connection** Given a state-transition system, $\langle \Sigma_C, \rightarrow_C \rangle$, we can use a Galois connection to define an appropriate mapping, $\mathcal{I}_C$, to make the state-transition system into a Kripke structure.

Say we have a Galois connection, $(\alpha: \mathcal{P}(\Sigma_C) \rightarrow A, \gamma: A \rightarrow \mathcal{P}(\Sigma_C))$, where $A$ is a complete lattice. $A$ serves as the collection of expressible properties—use $\mathcal{I}_C(c) = \alpha\{c\}$, for $c \in \Sigma_C$. Furthermore, we use the techniques from the previous section to define a best abstraction of $\mathcal{C}$:

$$\mathcal{A} = \langle A, \{a \rightarrow_A \alpha(c') \mid c \rightarrow_C c', c \in \gamma(a)\}, id_A \rangle$$

**Analyzing Nondeterminism with Galois Connections** Galois connections are well suited for studying the transitions that might be taken from a *set* of concrete states. We can readily synthesize the appropriate Kripke structure that represents the nondeterministic version of $\mathcal{C}$: For $S, S' \subseteq \Sigma_C$, write $S \xrightarrow{\bullet} S'$ to assert that for every $c' \in S'$, there exists some $c \in S$ such that $c \longrightarrow c'$. Next, we define the monotone Kripke structure, $\mathcal{PC}$, into which $\mathcal{C}$ embeds:

$$\mathcal{PC} = \langle \mathcal{P}(\Sigma_C), \xrightarrow{\bullet}, \mathcal{I}_{PC} \rangle, \text{ where } \mathcal{I}_{PC}(S) = \cap\{\mathcal{I}_C(c) \mid c \in S\}$$

Given a Galois connection, $(\alpha : \mathcal{P}(\Sigma_C) \rightarrow \Sigma_A, \gamma : \Sigma_A \rightarrow \mathcal{P}(\Sigma_C))$, we readily define this relation, $\mathcal{R}_{(\alpha,\gamma)} \subseteq \mathcal{P}(\Sigma_C) \times \Sigma_A$:

$$S\ \mathcal{R}_{(\alpha,\gamma)}\ a \text{ iff } \alpha(S) \sqsubseteq_A a$$

or equivalently,

$$S\ \mathcal{R}_{(\alpha,\gamma)}\ a \text{ iff } S \subseteq \gamma(a)$$

To verify that $\mathcal{R}_{(\alpha,\gamma)}$ is a simulation, we must "complete" the abstract structure, $\mathcal{A}$, with all the abstract transitions it needs to model transitions from sets of concrete states to sets of concrete states:

**Definition 8.** *Given a monotone Kripke structure,* $\mathcal{A} = \langle \Sigma_A, \rightarrow_A, \mathcal{I}_A \rangle$*, such that* $\Sigma_A$ *is a complete lattice, we define its* closure *as follows:*

$$\begin{aligned} &closure(\mathcal{A}) = \langle \Sigma_A, \rightarrow_{CA}, \mathcal{I}_A \rangle \\ &\quad where\ \rightarrow_{CA} = \{a \rightarrow \sqcup T \mid T \subseteq \{a' \mid a \rightarrow_A a'\}\} \end{aligned}$$

[9] This reasoning is entirely analogous to the reasoning done with Galois connections on abstractions of functional definitions to prove that the abstract transfer function, $\alpha \circ f \circ \gamma$, is the most precise abstraction of the concrete transfer function, $f$.

For example, if $a_0 \rightarrow_A a_1$ and $a_0 \rightarrow_A a_2$ in $\mathcal{A}$, then these transitions, as well as $a_0 \rightarrow_{CA} (a_1 \sqcup a_2)$, appear in $closure(\mathcal{A})$. The closure construction builds a monotone Kripke structure and shows us how the relation on elements of $\Sigma_C$ naturally lifts into a relation on *subsets* of $\Sigma_C$:

**Theorem 9.** *If $\mathcal{A}$ is monotone, then $\mathcal{C} \lhd_{\mathcal{R}_\gamma} \mathcal{A}$ implies $\mathcal{PC} \lhd_{\mathcal{R}_{(\alpha,\gamma)}} closure(\mathcal{A})$.*

**Lifting a Homomorphism into a Galois Connection** An elegant alternative to writing a Galois connection directly is using a homomorphism as an intermediary: Given Kripke structures $\mathcal{C}$ and $\mathcal{A}$, prove that $h : \Sigma_C \rightarrow \Sigma_A$ is a homomorphism, so that we have $\mathcal{C} \lhd_{\mathcal{R}_h} \mathcal{A}$. Next, "lift" $h$ into the Galois connection, $(\alpha_h : \mathcal{P}(\Sigma_C) \rightarrow \Sigma_A, \gamma_h : \Sigma_A \rightarrow \mathcal{P}(\Sigma_C))$, as follows [27]:

$$\begin{aligned} \alpha_h(S) &= \sqcup\{h(c) \mid c \in S\} \\ \gamma_h(a) &= \{c \mid h(c) \sqsubseteq_A a\} \end{aligned}$$

Practitioners often find it convenient to map each state in $\Sigma_C$ to the state in $\Sigma_A$ that best approximates it. Once function $h$ is defined this way and proved a homomorphism, then it "lifts" into a Galois connection.

Of course, one recovers $h : \Sigma_C \rightarrow \Sigma_A$ from the Galois connection, $(\alpha, \gamma)$, by defining $h(c) = \alpha\{c\}$.

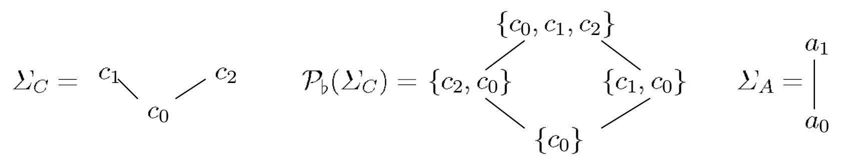
**Galois Connections on Lower Powerdomains** Stepwise abstractions might require that $\Sigma_C$ be nontrivially partially ordered and that its ordering be incorporated in the construction of $\mathcal{P}(\Sigma_C)$. In such a case, we suggest employing the *lower powerdomain* construction, $\mathcal{P}_\flat$ [13]: For a complete partially ordered set, $(P, \sqsubseteq_P)$,

$$\begin{aligned} &\mathcal{P}_\flat(P, \sqsubseteq_P) = (\{ScottClosure(S) \mid S \subseteq P, S \neq \{\}\ \}, \subseteq) \\ &\text{where } ScottClosure(S) = \downarrow S \cup \{\sqcup C \mid C \subseteq S \text{ is a chain}\} \\ &\text{and } \downarrow S = \{c \mid \text{exists } c' \in S, c \sqsubseteq_P c'\} \end{aligned}$$

(Recall that a *chain* is a sequence, $c_0 \sqsubseteq_P c_1 \sqsubseteq_P \cdots \sqsubseteq_P c_i \sqsubseteq_P \cdots$.) The elements of the lower powerdomain are nonempty *Scott-closed* sets, which are those nonempty subsets of $P$ that are closed downwards and closed under least-upper bounds of chains. An example of a lower powerdomain and its appearance in a Galois connection appears in Figure 7.

When using a lower powerdomain with the constructions in this section, we require that a homomorphism, $h : \Sigma_C \rightarrow \Sigma_A$, be a *Scott-continuous function* (that is, it preserves limits of chains: $h(\sqcup C) = \sqcup_{c \in C} h(c)$ for all chains, $C \subseteq \Sigma_C$).

Also, all the set-theoretic expressions in this section must be understood as Scott-closed sets—as necessary, apply *ScottClosure* to a set to make it Scott-closed.



$$\alpha : \mathcal{P}_\flat(\Sigma_C) \to \Sigma_A \qquad\qquad \gamma : \Sigma_A \to \mathcal{P}_\flat(\Sigma_C)$$

$$\alpha(S) = \begin{cases} a_0, & \text{if } S = \{c_0\} \\ a_1, & \text{otherwise} \end{cases} \qquad \begin{aligned} \gamma(a_0) &= \{c_0\} \\ \gamma(a_1) &= \{c_0, c_1, c_2\} \end{aligned}$$

**Fig. 7.** Lower powerdomain and Galois connection

## 7 Properties of Binary Relations

The previous sections showed that there are natural binary relations that underlie homomorphisms and Galois connections: In the case of a homomorphism, $h : \Sigma_C \to \Sigma_A$, we extract the relation, $\mathcal{R}_h \subseteq \Sigma_C \times \Sigma_A$:

$$c\, \mathcal{R}_h\, a \text{ iff } h(c) \sqsubseteq_A a$$

and in the case of a Galois connection, $(\alpha{:}\, \mathcal{P}(\Sigma_C) \to \Sigma_A, \gamma{:}\, \Sigma_A \to \mathcal{P}(\Sigma_C))$, we can extract the relation, $\mathcal{R}_\gamma \subseteq \Sigma_C \times \Sigma_A$:

$$c\, \mathcal{R}_\gamma\, a \text{ iff } c \in \gamma(a)$$

In both cases, the intuition behind $c\, \mathcal{R}\, a$ is that $a$ is an approximation of $c$ or that $c$ is a possible refinement of $a$.

It is worthwhile to "disassemble" these and other binary relations and examine the properties that make the relations well behaved. Initial efforts in this direction were taken by Hartmanis and Stearns [15], Mycroft and Jones [26], Schmidt [33], and Loiseaux, et al. [19].[10] In the developments that follow, we employ the usual monotone Kripke structures, $\mathcal{C} = \langle \Sigma_C, \to_C, \mathcal{I}_C \rangle$ and $\mathcal{A} = \langle \Sigma_A, \to_A, \mathcal{I}_A \rangle$.

Here are four fundamental properties on relations on partially ordered sets:

**Definition 10.** *For partially ordered sets $P$ and $Q$, a binary relation, $\mathcal{R} \subseteq P \times Q$, is*

- L-closed *("lower closed") iff for all $p \in P$, $q \in Q$, $p\, \mathcal{R}\, q$ and $p' \sqsubseteq_P p$ imply $p'\, \mathcal{R}\, q$*
- U-closed *("upper closed") iff for all $p \in P$, $q \in Q$, $p\, \mathcal{R}\, q$ and $q \sqsubseteq_Q q'$ imply $p\, \mathcal{R}\, q'$*
- G-closed *("greatest-lower-bound closed") iff for all $p \in P$, $\sqcap\{q \mid p\, \mathcal{R}\, q\}$ exists, and $p\, \mathcal{R}\, (\sqcap\{q \mid p\, \mathcal{R}\, q\})$*

[10] P. Cousot has remarked that the groundwork was laid by Shmuelli, but the appropriate references have not been located as of this date.

$$\begin{tikzcd}
\mathcal{C} = & c_1^{p,q} \arrow[dr] & & c_2^{p,r} \arrow[dl] \arrow[loop above] & & \mathcal{A} = & a_1^{p} \arrow[loop above] \arrow[dr] & \\
& & c_0 & & & & & a_0 \arrow[loop below]
\end{tikzcd}$$

$$\mathcal{R} = \{(c_0, a_0), (c_0, a_1), (c_1, a_1), (c_2, a_1)\}$$

**Fig. 8.** Binary simulation relation

– inclusive *iff for all chains* $\{p_i\}_{i \geq 0} \subseteq P$, $\{q_i\}_{i \geq 0} \subseteq Q$, *if for all* $i \geq 0$, $p_i \mathrel{\mathcal{R}} q_i$, *then both* $\sqcup_{i \geq 0} p_i$ *and* $\sqcup_{i \geq 0} q_i$ *exist, and* $\sqcup_{i \geq 0} p_i \mathrel{\mathcal{R}} \sqcup_{i \geq 0} q_i$.

L- and U-closure define monotonicity and antimonotonicity properties of the binary relation [26]. G-closure states that the relation determines a function from $P$ to $Q$, and inclusive-closure states that the relation is Scott-continuous.

For the concrete and abstract state sets in Figure 7, Figure 8 displays two structures that use these state sets and defines a binary relation between them. One can readily verify that the binary relation in the Figure is a simulation that is LUG-(and trivially inclusive)-closed. Homomorphisms and Galois connections supply exactly these properties:

**Proposition 11.** *Let* $h : \Sigma_C \rightarrow \Sigma_A$ *be a function, and define* $c \mathrel{\mathcal{R}_h} a$ *iff* $h(c) \sqsubseteq_A a$. *Then,*

1. $\mathcal{R}_h$ *is UG-closed;*
2. *if* $h$ *is monotonic, then* $\mathcal{R}_h$ *is L-closed;*
3. *if* $h$ *is Scott-continuous, then* $\mathcal{R}_h$ *is inclusive-closed.*

**Proposition 12.** *Let* $(\alpha\colon \mathcal{P}(\Sigma_C) \rightarrow \Sigma_A, \gamma\colon \Sigma_A \rightarrow \mathcal{P}(\Sigma_C))$ *be a Galois connection, and define* $c \mathrel{\mathcal{R}_\gamma} a$ *iff* $c \in \gamma(a)$. *Then* $\mathcal{R}_\gamma$ *is LUG-inclusive-closed.*

Momentarily, we will see that these properties let one lift a binary relation into a function or Galois connection, but first we examine the properties one by one and learn their value to proving simulations.

First, we must emphasize that *the properties in Definition 10 do not ensure that a relation is a simulation.* Here is a trivial counterexample:

$$\begin{aligned}
\mathcal{C} &= \langle \{c_0\}, \{c_0 \longrightarrow c_0\}, \mathcal{I}_C \rangle \\
\mathcal{A} &= \langle \{a_0\}, \{\}, \mathcal{I}_A \rangle
\end{aligned}$$

The relation, $c_0 \mathrel{\mathcal{R}} a_0$, is LUG-inclusive-closed, but $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$ does *not* hold. Regardless of the Definition 10-properties of a relation, $\mathcal{R}$, we must perform the usual simulation proof with the relation.[11]

[11] But a relation that lifts to a Galois connection *defines* a simulation that is sound and complete.

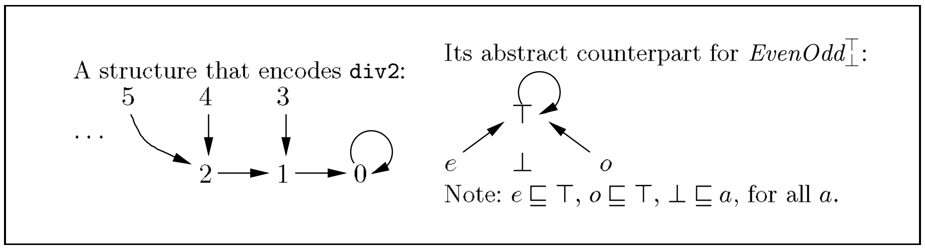

A structure that encodes div2:
5
4
3
...
2
1
0
Its abstract counterpart for $EvenOdd_{\perp}^{\top}$:
$\top$
$e$
$\perp$
$o$
Note: $e \sqsubseteq \top$, $o \sqsubseteq \top$, $\perp \sqsubseteq a$, for all $a$.

**Fig. 9.** Why U-closure is helpful for proving a simulation

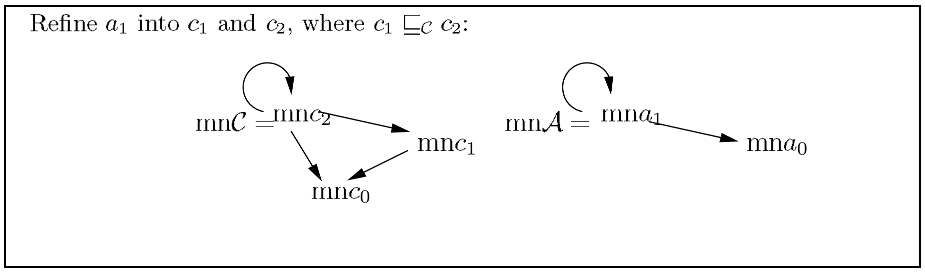

Refine $a_1$ into $c_1$ and $c_2$, where $c_1 \sqsubseteq_C c_2$:
mn$\mathcal{C}$ = mn$c_2$
mn$c_1$
mn$c_0$
mn$\mathcal{A}$ = mn$a_1$
mn$a_0$

**Fig. 10.** Why L-closure is helpful for proving a simulation

So, we must ask: How do the properties in Definition 10 aid us? To begin, consider an arbitrary relation, $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$, where we hope to prove that $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$. The guiding intuition behind $c\,\mathcal{R}\,a$ is that $a$ can act as an approximation of $c$ and that $c$ can act as a refinement of $a$.

If it is the case that $\Sigma_A$ is nontrivially partially ordered, the simulation proof will almost certainly require that $\mathcal{R}$ be U-closed, as suggested by the example in Figure 9. Within its transitions, the Figure's first structure encodes the `div2` operation on the natural numbers, and similarly, the second structure encodes the abstraction of `div2` on the tokens $e$ ("even"), $o$ ("odd'), and $\top$ ("any value"). To complete the proof that the second structure simulates the first, the obvious binary relation, $2n\,\mathcal{R}\,e$, $2n+1\,\mathcal{R}\,o$, must be made U-closed by adding $n\,\mathcal{R}\,\top$, for all $n \in \mathit{Nat}$.

We have this simple but useful property:

**Proposition 13.** *If $\mathcal{R}$ is U-closed, then $\gamma_{\mathcal{R}}(a) = \{c \mid c\,\mathcal{R}\,a\}$ is monotonic.*

The intuition behind $\gamma_{\mathcal{R}}(a)$ is that it identifies the elements in $\Sigma_C$ that are represented by $a \in \Sigma_A$. Monotonicity ensures that, as abstract states become less precise in $\sqsubseteq_A$, they represent more and more concrete states in $\Sigma_C$.

In an obvious, dual way, if one considers the refinement of a structure, $\mathcal{A}$, into an implementation, $\mathcal{C}$, where $\Sigma_C$ is nontrivially partially ordered, then L-closure is likely to be needed. Figure 10 gives one example, where the abstract state, $a_1$, is refined into the pair of states, $c_1$ and $c_2$, such that $c_1 \sqsubseteq_C c_2$. The refinement directs that the simulation relation be $c_0\,\mathcal{R}\,a_0$, and $c_i\,\mathcal{R}\,a_1$, for $i \in 1..2$. This relation is L-closed, and crucially so, in order to prove the simulation.

**Proposition 14.** *If $\mathcal{R}$ is L-closed, then $\beta_{\mathcal{R}}(c) = \{a \mid c\,\mathcal{R}\,a\}$ is antimonotonic.*

The intuition is that $\beta_{\mathcal{R}}(c)$ identifies the states in $\Sigma_A$ that might approximate $c$. The antimonotoncity result ensures that the more precisely defined concrete states possess smaller, "more precise" sets of possible approximations from $\Sigma_A$.

The $\beta_{\mathcal{R}}$ map gives good intuition about the behaviour of $\mathcal{R}$, but practitioners desire to map a concrete state, $c \in \Sigma_C$ to the "most precise" state that approximates it. G-closure yields these important results:

**Proposition 15.** *For partially ordered sets $P$ and $Q$, and binary relation, $\mathcal{R} \subseteq P \times Q$, define $\hat{\beta}_{\mathcal{R}}(c) = \sqcap\{a \mid c\,\mathcal{R}\,a\}$:*

1. *if $\mathcal{R}$ is G-closed, then $\hat{\beta}_{\mathcal{R}} : P \to Q$ is a well-defined function;*
2. *if $\mathcal{R}$ is LG-closed, then $\hat{\beta}_{\mathcal{R}}$ is monotonic;*
3. *if $\mathcal{R}$ is LG-inclusive-closed, then $\hat{\beta}_{\mathcal{R}}$ is Scott-continuous.*

This tells us that a G-closed relation identifies, for each $c \in \Sigma_C$, the element in $\Sigma_A$ that best approximates it, namely, $\sqcap\{a \mid c\,\mathcal{R}\,a\}$.

**Corollary 16.** *If $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$ is G-closed and $\mathcal{C} \lhd_{\mathcal{R}} \mathcal{A}$, then $\hat{\beta}_{\mathcal{R}}$ is a homomorphism from $\Sigma_C$ to $\Sigma_A$.*

The interaction of U- and G-closure brings us to Galois connections. For $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$, define $\gamma_{\mathcal{R}} : \Sigma_A \to \mathcal{P}(\Sigma_C)$ and $\alpha_{\mathcal{R}} : \mathcal{P}(\Sigma_C) \to \Sigma_A$ as follows:

$$\begin{aligned}
&\gamma_{\mathcal{R}}(a) = \{c \mid c\,\mathcal{R}\,a\}\\
&\alpha_{\mathcal{R}}(S) = \sqcup_{c \in S}\hat{\beta}_{\mathcal{R}}(c)\\
&\quad \text{where } \hat{\beta}_{\mathcal{R}}(c) = \sqcap\{a \mid c\,\mathcal{R}\,a\}
\end{aligned}$$

**Theorem 17.** *1. If $\mathcal{R}$ is UG-closed, then $(\alpha_{\mathcal{R}}, \gamma_{\mathcal{R}})$ form a Galois connection between $\mathcal{P}(\Sigma_C)$ and $\Sigma_A$;*
*2. If $\mathcal{R}$ is LUG-inclusive-closed and $\Sigma_C$ is nontrivially partially ordered, then $(\alpha_{\mathcal{R}}, \gamma_{\mathcal{R}})$ form a Galois connection between $\mathcal{P}_\flat(\Sigma_C)$ and $\Sigma_A$.*

Since Galois connections synthesize simulations that are sound and complete in the sense of Section 7, we conclude that *UG- (or LUG-)closed relations are best for abstraction studies.* If G-closure is impossible, then soundness can be ensured by U- (or LU-)closure, as noted by Mycroft and Jones [26].

## 8 Conclusion

We have seen how standard approaches for defining abstractions of program models can be understood in terms of structure-preserving binary relations. The crucial topics that follow the present work are: *(i)* the *extraction* of properties from an abstract Kripke structure and *(ii)* the *application* of properties to transformation of the corresponding concrete Kripke structure. Topic (i) can be best understood as validating temporal-logic properties by model checking [34, 35, 36, 37]; Topic (ii) is surprisingly underdeveloped, although Lacey, Jones, Van Wyk, and Frederiksen [18] and Cousot and Cousot [11] give recent proposals.

*Acknowledgements:* I thank Neil Jones for many years of inspiring work. The current paper evolved from research presented at the Workshop on Refinement and Abstraction, Amagasaaki, Japan, in 1999. I thank Yoshiki Kinoshita and the other workshop organizers for their encouragement. Anindya Banerjee, Michael Huth, Radha Jagadeesan, Markus Müller-Olm, and Hiroshi Watanabe supplied useful comments and corrections.

## References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Computer-Aided Verification 2000*, Lecture Notes in Computer Science. Springer, 2000.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[4] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In J.W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, number 803 in Lecture Notes in Computer Science, pages 124–175. Springer, 1993.

[5] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[6] P. Cousot, editor. *Static Analysis, 8th International Symposium.* Lecture Notes in Computer Science 2126, Springer, Berlin, 2001.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[9] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[10] P. Cousot and R. Cousot. Higher-order abstract interpretation. In *Proc. IEEE Int'l. Conf. Programming Languages.* IEEE Press, 1994.

[11] P. Cousot and R. Cousot. Systematic design of program transformations by abstract interpretation. In *Proc. 29th ACM Symp. on Principles of Prog. Languages.* ACM Press, 2002.

[12] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19:253–291, 1997.

[13] C. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1992.

[14] D. Harel. Statecharts: a visual formalization for complex systems. *Science of Computer Programming*, 8, 1987.

[15] J. Hartmanis and R. Streans. Pair algebras and their application to automata theory. *Information and Control*, 7:485–507, 1964.

[16] M. Hecht. *Flow Analysis of Computer Programs.* Elsevier, 1977.

[17] N. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 527–636. Oxford Univ. Press, 1995.

[18] D. Lacey, N.D. Jones, E. Van Wyk, and C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symp. on Principles of Prog. Languages.* ACM Press, 2002.

[19] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.
[20] C. McGowan. An inductive proof technique for interpreter equivalence. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 139–148. Prentice-Hall, 1972.
[21] A. Melton, G. Strecker, and D. Schmidt. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. Lecture Notes in Computer Science 240, Springer-Verlag, 1985.
[22] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Lecture Notes in Computer Science 92, 1980.
[23] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
[24] J.C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1996.
[25] M. Müller-Olm, D.A. Schmidt, and B. Steffen. Model checking: A tutorial introduction. In G. Filé and A. Cortesi, editors, *Proc. 6th Static Analysis Symposium*. Springer LNCS, 1999.
[26] A. Mycroft and N.D. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, pages 156–171. Lecture Notes in Computer Science 217, Springer-Verlag, 1985.
[27] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.
[28] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.
[29] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
[30] D. Park. Concurrency and automata in infinite strings. Lecture Notes in Computer Science 104, pages 167–183. Springer, 1981.
[31] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–374. Academic Press, 1980.
[32] J. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.
[33] D.A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.
[34] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. on Principles of Prog. Languages*. ACM Press, 1998.
[35] D.A. Schmidt. Binary relations for abstraction and refinement. *Workshop on Refinement and Abstraction, Amagasaki, Japan, Nov. 1999. Elsevier Electronic Notes in Computer Science*, to appear.
[36] D.A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *Proc. 5th Static Analysis Symposium*. Springer LNCS 1503, 1998.
[37] B. Steffen. Generating data-flow analysis algorithms for modal specifications. *Science of Computer Programming*, 21:115–139, 1993.
[38] B. Steffen. Property-oriented expansion. In R. Cousot and D. Schmidt, editors, *Static Analysis Symposium: SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, 1996.

# Principles of Inverse Computation and the Universal Resolving Algorithm

Sergei Abramov[1] and Robert Glück[2]*

[1] Program Systems Institute, Russian Academy of Sciences
RU-152140 Pereslavl-Zalessky, Russia,
abram@botik.ru

[2] PRESTO, JST & Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan,
glueck@acm.org

**Abstract.** We survey fundamental concepts in inverse programming and present the Universal Resolving Algorithm (URA), an algorithm for inverse computation in a first-order, functional programming language. We discuss the principles behind the algorithm, including a three-step approach based on the notion of a perfect process tree, and demonstrate our implementation with several examples. We explain the idea of a semantics modifier for inverse computation which allows us to perform inverse computation in other programming languages via interpreters.

## 1 Introduction

While computation is the calculation of the output of a program for a given input, *inverse computation* is the calculation of the possible input of a program for a given output. The only area that deals successful with a solution to inversion problems is logic programming [34, 36]. But solving inversion problems also makes sense outside of logic programming, for example, when an algorithm for one direction is easier to write than an algorithm for the direction in which we are interested, or when both directions of an algorithm are needed (such as encoding/decoding). Ideally, the inverse program can be derived from the forward program by *program inversion*. Given the importance of program inversion, relatively few papers have been devoted to the topic. We regard program inversion, beside program specialization and program composition, as one of the three fundamental tasks for metacomputation [16].

The idea of inverse computation can be traced back to reference [38]. We found that inverse computation and program inversion can be related conveniently [4] using the *Futamura projections* [12] which are well-known from *partial evaluation* [30]: a program inverter is a generating extension of an inverse interpreter. This insight is useful because inverse computation is simpler than program inversion. In this paper we shall focus on inverse computation.

* On leave from DIKU, Department of Computer Science, University of Copenhagen.

We survey fundamental concepts in inverse programming and present the *Universal Resolving Algorithm* (URA), an algorithm for inverse computation in a first-order functional language, by means of examples. The emphasis in this paper is on presenting principles and foundations of inverse computation. We cover the topic starting from theoretical considerations to actual computer experiments. More details and a formal treatment of the material presented in this paper can be found in the literature [2, 4, 5]. We also draw upon results of references [15, 17, 18]. This work belongs to a line of research on supercompilation and metacomputation (*e.g.*, [48, 50, 32, 1, 20, 29, 40, 46, 43]). It is the first paper in this series that brings together, in a unified form, the results of inverse computation. We assume the reader is familiar with partial evaluation and programming languages, for example as presented in reference [30, Parts I&II] or [28].

The paper is organized as follows: Sect. 2 summarizes fundamental concepts in inverse programming; Sect. 3 explains the principles behind the Universal Resolving Algorithm; Sect. 4 illustrates the algorithm with a complete example, and Sect. 5 demonstrates our implementation; Sect. 6 discusses related work; and Sect. 7 concludes the presentation.

## 2 Fundamental Concepts in Inverse Programming

This section surveys fundamental concepts in inverse programming from a language-independent perspective. We present two tools, an inverse interpreter and a program inverter, for solving inversion problems, explain important properties of inverse computation and how some of the constructions can be optimized.

*Notation* For any program text $p$, written in language $L$, we let $[\![p]\!]_L\ d$ denote the application of $L$-program $p$ to its input $d$ (when the index $L$ is unspecified, we assume that a language $L$ is intended). The notation is strict in its arguments. When we define a program using $\lambda$-abstraction, we assume that the definition is translated into the corresponding programming language. We assume that we are dealing with universal programming languages and, thus, this translation is always possible. Equality between applications shall always mean strong (computational) equivalence: either both sides of an equation are defined and equal, or both sides are undefined. Values are S-expressions known from Lisp. We write 'A for an atom A (for numbers we omit the quote), $d_1$:$d_2$ for a pair of values $d_1$ and $d_2$, and $[d_1, d_2, ..., d_n]$ for a list $d_1$:($d_2$:(...($d_n$:'Nil)...)) .

**Tools for Solving Inversion Problems** We distinguish between two tools for solving inversion problems: an inverse interpreter that performs *inverse computation* and a program inverter that performs *program inversion*. Let $p$ be a program that computes $y$ from $x$. For simplicity, we assume $p$ is injective in $x$. Computation of output $y$ by applying $p$ to $x$ is described by:

$$[\![p]\!]\, x = y \tag{1}$$

1. **Inverse interpreter**: The determination, for a given program $p$ and output $y$, of an input $x$ of $p$ such that $[\![p]\!]\,x = y$ is inverse computation. A program *invint* that performs inverse computation, is an *inverse interpreter*.

$$[\![\mathit{invint}]\!]\,[p, y] = x \tag{2}$$

2. **Program inverter**: Let $p^{-1}$ be a program that performs inverse computation for a given program $p$. A program *invtrans* that produces $p^{-1}$, is a *program inverter* (also called an inverse translator). Program $p^{-1}$ will often be significantly faster in computing $x$ than inverse interpretation of $p$ by *invint*. Inversion in two stages is described by:

$$[\![\mathit{invtrans}]\!]\,p = p^{-1} \tag{3}$$

$$[\![p^{-1}]\!]\,y = x \tag{4}$$

Inverse computation of a program $p$ can be performed in one stage with *invint* and in two stages with *invtrans*. Using equations (2, 3, 4) we obtain the following functional equivalence between *invint* and *invtrans*:

$$\underbrace{[\![\mathit{invint}]\!]\,[p, y]}_{\text{one stage}} = \underbrace{[\![\,[\![\mathit{invtrans}]\!]\,p\,]\!]\,y}_{\text{two stages}} \tag{5}$$

It is easy to see that an inverse interpreter can be defined in terms of a program inverter, and vice versa (where *sint* is a self-interpreter[1]):

$$\mathit{invint}' \equiv \lambda[p, y].[\![\mathit{sint}]\!]\,[[\![\mathit{invtrans}]\!]\,p, y] \tag{6}$$

$$\mathit{invtrans}' \equiv \lambda p.\lambda y.[\![\mathit{invint}]\!]\,[p, y] \tag{7}$$

Both definitions can be optimized using program composition as in the degeneration projections [18] (*invtrans* $\rightarrow$ *invint*), and program specialization as in the Futamura projections [12] (*invint* $\rightarrow$ *invtrans*). This will be explained below.

The computability of the solution is not always guaranteed, even with sophisticated inversion strategies. Some inversions are too resource consuming, while others are undecidable. For particular values $p$ and $y$, no value $x$ need exist. In fact, the existence of a value $x$ is an undecidable question: there is no program that will always terminate if such a value does not exist.

In spite of this, an inverse interpreter exists [38] that will compute value $x$ if it exists. Essentially, that program computes $[\![p]\!]\,x$ for all values $x$ until it comes to an $x$ such that $[\![p]\!]\,x = y$. Since that computation may not terminate for some $x$, the program needs to perform the search for $x$ in a diagonal manner. For instance, it examines $\forall k \geq 0$ and $\forall x$ with $\mathit{size}(x) \leq k$ the result of computation $[\![p]\!]^k\,x$ where $[\![p]\!]^k\,x = [\![p]\!]\,x$ if the computation of $[\![p]\!]\,x$ stops in $k$ steps; undefined otherwise. This *generate-and-test approach* will correctly find all solutions, but is too inefficient to be useful.

[1] A self-interpreter *sint* for $L$ is an $L$-interpreter written in $L$: $[\![\mathit{sint}]\!]_L\,[p, x] = [\![p]\!]_L\,x$ .

Even when it is certain that an efficient inverse program $p^{-1}$ exists, the derivation of such a procedure from $p$ by a program inverter may be difficult. Most of the work on program inversion (*e.g.*, [7, 8, 10, 24, 25, 39, 51]) has been on program transformation by hand: specify a problem as the inverse of an easy computation, and then derive an efficient algorithm by manual application of transformation rules. Note that we can define a *trivial inverse program* $p_{triv}^{-1}$ for any program $p$ using an inverse interpreter *invint*:

$$p_{triv}^{-1} \equiv \lambda y. [\![invint]\!]\, [p, y] \tag{8}$$

**Inverse Programming** Programs are usually written for forward computation. In *inverse programming* [1] one writes programs so that their backwards computation produces the desired result. Inverse programming relies on the existence of an inverse interpreter or a program inverter. Given a program $p$ that computes $y$ from $x$, one solves the inversion problem for a given $y$:

$$[\![p^{-1}]\!]\, y = x \tag{9}$$

This gives rise to a *declarative style* of programming where one specifies the result rather than describes how it is computed. An example is a program $r$ that checks whether two values $x$ and $y$ are in relation, and returns the corresponding Boolean value $bool \in \{\text{'True}, \text{'False}\}$ :

$$[\![r]\!]\, [x, y] = bool \tag{10}$$

Although checking whether $x$ and $y$ are in relation is only a particular case of forward computation, it is worth considering it separately because of the large number of problems that can be formulated this way. Relations are often used in specifications as they are not directional and give preference neither to $x$ nor to $y$. For example, in logic programming one writes a relation $r$ in a subset of *first-order logic* (Horn clauses) and solves a restricted inversion problem for a given $y$ and output 'True by an inverse interpreter. The resolution principle [41] reduced the combinatorial growth of the search space of the early generate-and-test methods used in automated theorem proving [14, 9], and became the basis for the use of first-order logic as a tool for inverse programming [23, 34].

The importance of inverting relations stems from the fact that it is often easier to check the correctness of a solution than to find it. But solving inversion problems also makes sense outside of logic programming, for example, when an algorithm for one direction is easier to write than an algorithm for the direction in which we are interested, or when both directions of an algorithm are needed. In fact, the problem of inversion does not depend on a particular programming language. Later in this paper we will study a small functional language (Sect. 4), and show inverse computation in a small imperative language (Sect. 5.3).

**A Semantics Modifier for Inverse Computation** We now explain an important property of inverse computation, namely that inverse computation can be performed in any language via an interpreter for that language [2]. Suppose we have two functionally equivalent programs $p$ and $q$ written in languages $P$ and $Q$, respectively. For the sake of simplicity, let $p$ and $q$ be injective. Let *invintP* and *invintQ* be two inverse interpreters for $P$ and $Q$, respectively. Since $p$ and $q$ are functionally equivalent, inverse computation of $p$ and $q$ using the corresponding inverse interpreter produces the same result (we say the inverse semantics is *robust* [2]):

$$(\forall x \,.\, [\![p]\!]_P\ x = [\![q]\!]_Q\ x) \;\Rightarrow\; (\forall y \,.\, [\![\mathit{invintP}]\!]_L\ [p, y] = [\![\mathit{invintQ}]\!]_M\ [q, y]) \quad (11)$$

Suppose we have a $Q$-interpreter written in $P$, but no inverse interpreter for $Q$. How can we perform inverse computation in $Q$ without writing an inverse interpreter *invintQ*? The following functional equivalence holds for the $Q$-interpreter:

$$[\![\mathit{intQ}]\!]_P\ [q, x] = [\![q]\!]_Q\ x \quad (12)$$

We can immediately define a $P$-program $q'$ (by fixing the program argument of *intQ*) such that $q'$ and $q$ are functionally equivalent, so $[\![q']\!]_P\ x = [\![q]\!]_Q\ x$, where

$$q' \equiv \lambda x.[\![\mathit{intQ}]\!]_P\ [q, x] \quad (13)$$

Then the inversion problem of $q$ can be solved by applying *invintP* to $q'$:

$$[\![\mathit{invintP}]\!]_L\ [q', y] = x \quad (14)$$

The result $x$ is a correct solution for the inversion problem $y$ because $q$ and $q'$ are functionally equivalent and inverse computation is robust (as stated for injective programs in Eq. 11). It is noteworthy that we obtained a solution for inverse computation of a $Q$-program using an inverse interpreter for $P$ and an interpreter for $Q$. This property of inverse computation is convenient because writing an interpreter is usually easier than writing an inverse interpreter. We show an example in Sect. 5.3 where we perform inverse computation in an imperative language using an inverse interpreter for a functional language. This indicates that the essence of an inverse semantics can be captured in a language-independent way.

Finally, we define a *semantics modifier for inverse computation* [4]. The program takes a $Q$-interpreter *intQ* written in $L$, a $Q$-program $q$, and an output $y$, and computes a solution $x$ for the given inversion problem $(q, y)$:

$$[\![\mathit{invmod}]\!]_L[\mathit{intQ}, q, y] = x \quad (15)$$

The inversion modifier *invmod* makes it easy to define an inverse interpreter for any language $Q$, given a standard interpreter *intQ* for that language:

$$\mathit{invintQ}' \equiv \lambda[q, y].[\![\mathit{invmod}]\!]_L[\mathit{intQ}, q, y] \quad (16)$$

Such programs are called *semantics modifiers* because they modify the standard semantics of a language $Q$ (given in the form of an interpreter *intQ*). The inversion modifier can be defined using the expressions in (13, 14). The construction is a generalization of the classical two-level interpreter hierarchy (a theoretical exposition of non-standard interpreter hierarchies can be found in reference [2]).

$$\begin{aligned} \mathit{invmod} &\equiv \lambda[\mathit{intQ}, q, y].[\![\mathit{invintP}]\!]_L\,[q', y] \\ &\quad \text{where } q' \equiv \lambda x.[\![\mathit{intQ}]\!]_P\,[q, x] \end{aligned} \tag{17}$$

**Optimizing the Constructions** As shown in [4], inverse interpreters and program inverters can be related conveniently using the Futamura projections known from partial evaluation [30]: a program inverter is a generating extension of an inverse interpreter. Similarly, an inverse interpreter is the composition of a self-interpreter and a program inverter [17, 18]. Given one of the two tools, we can obtain, in principle, an efficient implementation of its companion tool by program specialization or by program composition. We will describe three possibilities (a, b, c) for optimizing these constructions.

Before we show these transformations, we specify the semantics of a program composer *komp* and a program specializer *spec*. For notational convenience we assume that both programs are $L$-to-$L$ transformers written in $L$; for multi-language composition and specialization see [17].

$$[\![\mathit{komp}]\!]\,[p, q] = pq \;\text{ such that }\; [\![pq]\!]\,[x, y] = [\![p]\!]\,[\,[\![q]\!]\,x, y] \tag{18}$$

$$[\![\mathit{spec}]\!]\,[p, m, x_1...x_m] = p_x \;\text{ such that }\; [\![p_x]\!]\,[y_1...y_n] = [\![p]\!]\,[x_1...x_m, y_1...y_n] \tag{19}$$

*a) From program inversion to inverse computation* Let *sint* be a self-interpreter for $L$. Consider the composition of *sint* and *invtrans* in Eq. 6. Inverse computation is performed in two stages: first, program inversion of $p$; second, interpretation of $p$'s inverse program with $y$.

$$[\![\mathit{sint}]\!]\,[\,[\![\mathit{invtrans}]\!]\,p, y] = x \tag{20}$$

When we apply a program composer to this composition, we obtain a new program which we call *invint′* (it has the functionality of an inverse interpreter):

$$[\![\mathit{komp}]\!]\,[\mathit{sint}, \mathit{invtrans}] = \mathit{invint}' \tag{21}$$

$$[\![\mathit{invint}']\!]\,[p, y] = x \tag{22}$$

This application of a program composer is known as *degeneration* [17, 18]. The transformation converts a two-stage computation into a one-stage computation (here, a program inverter into an inverse interpreter).

*b) From inverse computation to program inversion* Consider an inverse interpreter *invint* and apply it to a program $p$ and a request $y$ as shown in Eq. 7:

$$[\![\mathit{invint}]\!]\,[p, y] = x \tag{23}$$

When we specialize the inverse interpreter *wrt* its first argument $p$ using a program specializer, we obtain a new program which we call $p'^{-1}$ (it has the functionality of an inverse program of $p$):

$$[\![\mathit{spec}]\!]\,[\mathit{invint}, 1, p] = p'^{-1} \tag{24}$$

$$[\![p'^{-1}]\!]\,[y] = x \tag{25}$$

We can continue the transformation and specialize the program specializer *spec* in Eq.24 *wrt* its arguments *invint* and 1. This is known as *self-application* [12] of a program specializer. Self-application is possible because a specializer, like any other program, can be subjected to program transformation. We obtain a new program which we call *invtrans′* (it has the functionality of a program inverter):

$$[\![spec]\!]\,[spec, 2, invint, 1] = invtrans' \tag{26}$$
$$[\![invtrans']\!]\; p = p'^{-1} \tag{27}$$

This transformation of an inverse interpreter into a program inverter is interesting because inverse computation is conceptually simpler than program inversion (*e.g.*, no code generation). The applications of program specialization in Eq. 24 and Eq. 26 are a variation of the 1st and 2nd Futamura projection [12], which were originally proposed for transforming interpreters into translators (and pioneered in the area of partial evaluation [30]). We see that the same principle applies to the transformation of inverse interpreters into program inverters.

*c) Porting inverse interpreters* Let *invmod* be an inversion modifier for $P$, and use it to perform inverse computation of a $Q$-program $q$ via a $Q$-interpreter *intQ* written in $L$ (as shown in Eq. 15):

$$[\![invmod]\!]\,[intQ, q, y] = x \tag{28}$$

When we specialize the modifier *wrt* its argument *intQ*, we obtain a new program which we call *invintQ″* (it has the functionality of an inverse interpreter for $Q$):

$$[\![spec]\!]\,[invmod, 1, intQ] = invintQ'' \tag{29}$$
$$[\![invintQ'']\!]\,[q, y] = x \tag{30}$$

We see that program specialization has the potential to make inversion modifiers more efficient. This is important for a practical application of the modifiers because inverse interpretation of a $Q$-program using an interpreter *intQ* as mediator between languages $P$ and $Q$ adds additional computational overhead. The equations can be carried further [4], for instance, when we specialize *spec* in Eq. 29 *wrt* its arguments *invmod* and 1, we obtain a program which converts an interpreter *intQ* into an inverse interpreter *invintQ″*. Or, when we specialize *invmod* in Eq. 28 *wrt* its arguments *intQ* and $q$, we obtain an $L$-program $q^{-1}$ (it has the functionality of an inverse program of $Q$-program $q$).

*About the transformers* The equations shown above say nothing about the efficiency of the transformed programs. This depends on the transformation power of each transformer. Indeed, each equation can be realized using a trivial program composer or a trivial program specializer:

$$komp_{triv} \equiv \lambda[p, q].\lambda[x, y].[\![p]\!]\,[\,[\![q]\!]\,x, y] \tag{31}$$
$$spec_{triv} \equiv \lambda[p, m, x_1, ..., x_m].\lambda[y_1, ..., y_n].[\![p]\!]\,[x_1, ..., x_m, y_1, ..., y_n] \tag{32}$$

In practice, we expect the transformers to be non-trivial. A non-trivial program composer generates an efficient composition of $p$ and $q$ by removing intermediate data structures, redundant computations, and other interface code (*e.g.*, [50, 52]). A non-trivial program specializer recognizes which of $p$'s computations can be precomputed so as to yield an efficient residual program $p_x$ (*e.g.*, [30]).

The question raised by the equations above is operational: to what extent can the constructions be optimized by existing transformers? If it is not possible with current methods, then this is a challenge for future work on program transformation.

## 3 Principles of Inverse Computation

This section presents the concepts behind the Universal Resolving Algorithm. We discuss the inverse semantics of programs and the key concepts of the algorithm.

**Inverse Semantics of Programs** The determination, for a program $p$ written in programming language $L$ and output $d_{out}$, of an input $ds_{in}$ such that $[\![p]\!]_L\ ds_{in} = d_{out}$ is inverse computation. When a program $p$ is not injective, or additional information about the input is available, we may want to restrict the search space of the input for a given output. Similarly, we may also want to specify a set of output values, instead of fixing a particular value. We do so by specifying the input and output domains using an *input-output class* $cls_{io}$. A class is a finite representation of a possibly infinite set of values. Let $\lceil cls_{io} \rceil$ be the set of values represented by $cls_{io}$, then a correct solution $Inv$ to an inversion problem is specified by

$$Inv(L, p, cls_{io}) = \{\ (ds_{in}, d_{out})\ \mid\ (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil,\ [\![p]\!]_L\ ds_{in} = d_{out}\ \} \quad (33)$$

where $L$ is a programming language, $p$ is an $L$-program, and $cls_{io}$ is an input-output class. The *universal solution* $Inv(L, p, cls_{io})$ for the given inversion problem is the largest subset of $\lceil cls_{io} \rceil$ such that $[\![p]\!]_L\ ds_{in} = d_{out}$ for all elements $(ds_{in}, d_{out})$ of this subset. An *existential solution* picks one of the elements of the universal solution as an answer. Note that computing an existential solution is a special case of computing a universal solution (the search stops after finding the first solution).

**Inverse Computation** In general, inverse computation using an inverse interpreter *invint* for $L$ takes the form

$$[\![invint]\!]\ [p, cls_{io}]\ =\ ans \quad (34)$$

where $p$ is an $L$-program and $cls_{io}$ is an input-output class. We say, $cls_{io}$ is a *request* for inverse computation of $L$-program $p$. When designing an algorithm for inverse computation, we need to choose a concrete representation of the input-output class $cls_{io}$ and the solution set *ans*. In this paper we use S-expressions known from Lisp as the value domain, and represent the search space $cls_{io}$ by

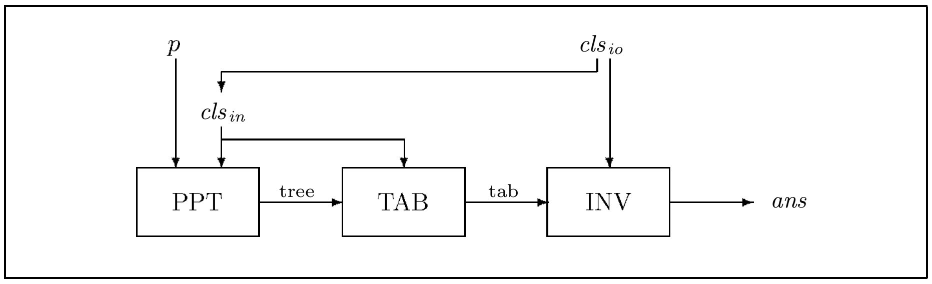


**Fig. 1.** Conceptual approach: three steps to inverse computation

*expressions with variables and restrictions*. This is a simple and elegant way to represent subsets of our value domain. (Other algorithms for inverse computation may choose other representations.)

The *Universal Resolving Algorithm* (URA) [5, 3] is an algorithm for inverse computation in a first-order functional language. The answer produced by URA is a set of substitution-restriction pairs $ans = \{(\theta_1, \widehat{r}_1), \ldots\}$ which represents set *Inv* for the given inversion problem. More formally, the correctness of the answer produced by URA is given by

$$\bigcup_i \lceil (cls_{io}/\theta_i)/\widehat{r}_i \rceil \;=\; Inv(L, p, cls_{io}) \tag{35}$$

where $(cls_{io}/\theta_i)/\widehat{r}_i$ narrows the pairs of values represented by $cls_{io}$ by applying substitution $\theta_i$ to $cls_{io}$ and adding restriction $\widehat{r}_i$ to the domains of the free variables. The set representation and the operations on it will be explained in Sect. 4. Our algorithm produces a universal solution, hence the first word of its name.

**An Approach to Inverse Computation** Inverse computation can be organized into three steps: walking through a perfect process tree (PPT), tabulating the input and output (TAB), and extracting the answer to the inversion problem from the table (INV). This organization is shown in Fig. 1. In practice, the three steps can be carried out in a single phase. However, we shall not be concerned with different implementation techniques in this paper.

Our approach is based on the notion of a *perfect process tree* [15] which represents the computation of a program with *partially specified input* (class $cls_{in}$ taken from $cls_{io}$) by a tree of all possible computation traces. Each fork in a perfect tree partitions the input class $cls_{in}$ into disjoint and exhaustive subclasses. The algorithm then constructs, breadth-first and lazily, a perfect process tree for a given program $p$ and input class $cls_{in}$. Note that we first construct a forward trace of the computation given $p$ and $cls_{in}$, and then extract the solution to the backward problem using $cls_{io}$. The construction of a process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input (*cf.* [28]).

In the next section we present each of the three steps in more detail:

1. **Perfect Process Tree**: tracing program $p$ under standard computation with input class $cls_{in}$ taken from $cls_{io}$.
2. **Tabulation**: forming the table of input-output pairs from the perfect process tree and class $cls_{in}$.
3. **Inversion**: extracting the answer for the desired output given by $cls_{io}$ from the table of input-output pairs.

Since our method for inverse computation is sound and complete [5], and since the source language of our algorithm is a universal programming language, which follows from the fact that the Universal Turing Machine can be programmed in it, we can apply inverse computation to any computable function. Thus our method for inverse computation has full generality.

## 4 A Complete Example

As an example, consider a program that replaces symbols in a symbol list by a new value given in a replacement list. Program 'findrep' in Fig. 2 is tail-recursive and returns the new symbol list in reversed order. For instance, applying the program to symbol list s = [1, 2] and replacement list[2] rr = [2, 3] returns the reversed symbol list [3, 1] where 2 has been replaced by 3:

$$[\![\text{findrep}]\!]\ [[1,2],[2,3]] \;=\; [3,1]\ .$$

The program is written in TSG, a first-order functional language. The language is a typed dialect of S-Graph [15]. The body of a function consists of a term which is either a function call, a conditional or an expression. The calls are restricted to tail-recursion. Values can be tested and/or decomposed in two ways. Test **eqa?** checks the equality of two atoms, and (**cons?** x h t a) works like pattern matching: if the value of variable x is a pair $d_1$**:**$d_2$, then variable h is bound to head $d_1$ and variable t to tail $d_2$; otherwise the value (an atom) is bound to variable a. By '_' we denote anonymous variables.

**An Inversion Problem** Suppose we have output [3, 1], and we want to find all possible inputs which produce this output. To show a complete example of inverse computation, let us limit the search space to lists of length two. We specify the symbol list as a list of two $\underline{a}$toms, $[Xa_1, Xa_2]$, and the replacement list as a list of two S-$\underline{e}$xpressions, $[Xe_3, Xe_4]$. Placeholders $Xa_i, Xe_i$ are called *configuration variables* (c-variables). We specify the input and output domains by the input-output class

$$cls_{io} = \langle (\, \underbrace{[[Xa_1, Xa_2], [Xe_3, Xe_4]]}_{\widehat{ds}_{in}}\,,\ \underbrace{[3,1]}_{\widehat{d}_{out}}\,),\ \emptyset\ \rangle$$

[2] In general, rr = $[u_1, v_1, \ldots, u_n, v_n]$ where $u_i$ is a symbol and $v_i$ its replacement.

where $\widehat{ds}_{in}$ is the partially specified input, $\widehat{d}_{out}$ is the desired output, and the restriction on the domain of c-variables is empty ($\emptyset$). Inverse computation with URA takes the form:

$$[\![ura]\!]\ [\text{findrep}, cls_{io}] = ans\ .$$

As the answer *ans* of inverse computation, we expect a (possibly infinite) sequence of substitution-restriction pairs for the c-variables occurring in $cls_{io}$ (*cf.* Eq. 35).

**Set Representation** Expressions with variables, called *c-expressions*, represent sets of values by means of two types of variables: *ca-variables* $Xa_i$ which range over atoms, and *ce-variables* $Xe_i$ which range over S-expressions.

For instance, consider the input class $cls_{in} = \langle \widehat{ds}_{in}, \emptyset \rangle$ which we take from $cls_{io}$ above. The set of values represented by $cls_{in}$, denoted $\lceil cls_{in} \rceil$, is

$$\lceil cls_{in} \rceil = \{\, [[da_1, da_2], [d_3, d_4]] \ \mid\ da_1, da_2 \in \text{DAval}, d_3, d_4 \in \text{Dval}\,\}$$

where DAval is the set of all atoms and Dval is the set of all values (S-expressions). In our example, $cls_{io}$ and $cls_{in}$ contain an empty restriction ($\emptyset$). In general, a *restriction* is a finite set of non-equalities on ca-variables. A *non-equality* takes the form $(Xa_i \,\#\, Xa_j)$ or $(Xa_i \,\#\, \text{'A})$, where 'A is an arbitrary atom. It specifies a set of values to which a ca-variable must not be equal. Suppose we add restriction $\{\, (Xa_1 \,\#\, Xa_2) \,\}$ to $cls_{in}$, then the set of values represented by the new class is narrowed to

$$\lceil cls'_{in} \rceil = \{\, [[da_1, da_2], [d_3, d_4]] \ \mid\ da_1, da_2 \in \text{DAval}, d_3, d_4 \in \text{Dval}, da_1 \neq da_2\,\}\ .$$

To restrict a class to represent an empty set of values, we add to it a *contradiction* $(Xa_1 \,\#\, Xa_1)$: there exist no value which satisfies this non-equality. Later, when we construct a perfect process tree, we shall see how restrictions are used. We need only non-equalities on ca-variables because our language can only test atoms for non-equality.[3] C-expressions and restrictions are our main method for manipulating infinite sets of values using a finite representation.

**1. Perfect Process Trees** A computation is a linear sequence of states and transitions. Each state and transition in a deterministic computation is fully defined. Tracing a program with partially specified input ($cls_{in}$) may confront us with tests that depend on unspecified values (represented by c-variables), and we have to consider the possibility that either branch is entered with some input value. This leads to traces that branch at conditionals that depend on unspecified values. Tracing a computation is called *driving* in supercompilation [50]; the method used below is *perfect driving* [15]. (A variant is *positive driving* [46].)

To trace a program with partially specified input we use *configurations* of the form $\langle (t, \widehat{\sigma}), \widehat{r} \rangle$ where $t$ is a program term, $\widehat{\sigma}$ an environment binding program

[3] An extension to express non-equalities between ce-variables can be found in [43].

```
    (define findrep [s, rr]
t1    (call scan [s, rr, [ ]]))

    (define scan [s, rr, out]
t2    (if (cons? s h s _)
t3        (if (cons? h _ _ a)
t4            'Error:atom_expected
t5            (call find [a, s, rr, rr, out]))
t6        out))
```

```
    (define find [a, s, r, rr, out]
t7    (if (cons? r h r _)
t8        (if (cons? h _ _ b)
t9            'Error:atom_expected
t10           (if (cons? r c r _)
t11               (if (eqa? a b)
t12                   (call scan [s, rr, c:out])
t13                   (call find [a, s, r, rr, out]))
t14               'Error:pair_expected))
t15       (call scan [s, rr, a:out])))
```

**Fig. 2.** TSG-program 'findrep'

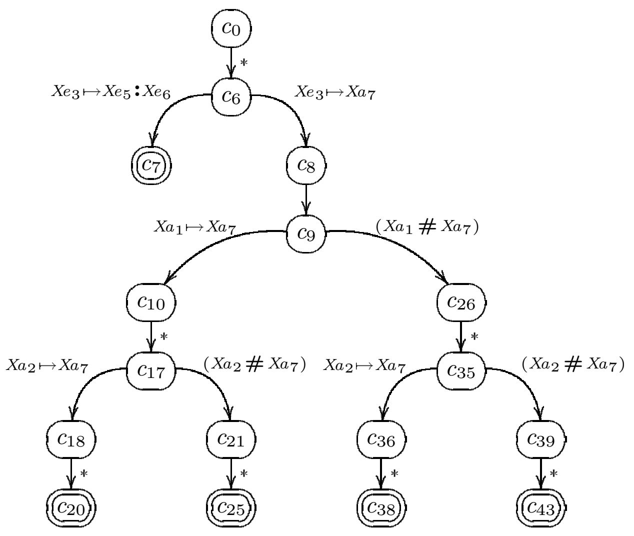


**Fig. 3.** Perfect process tree for program 'findrep'

variables to c-expressions, and $\widehat{r}$ is a restriction on the domain of c-variables. We call $\widehat{\sigma}$ a *c-environment* to stress that this environment binds program variables to c-expressions instead of values.

Let us trace our example program. We begin with a tree whose single node is labeled with the initial configuration $c_0$. The initial term $t_0$ is a call to function findrep, the initial c-environment binds program variables s and rr to the corresponding c-expressions, and the initial restriction is empty.

$$c_0 = \langle \underbrace{(\textbf{call}\ \text{findrep}\ [\text{s}, \text{rr}])}_{\text{term } t_0},\ \underbrace{[\text{s} \mapsto [Xa_1, Xa_2], \text{rr} \mapsto [Xe_3, Xe_4]]}_{\text{c-environment}}),\ \underbrace{\ \emptyset\ }_{\text{restr.}} \rangle$$

| $c$ | $t$ | a | s | b | h | c | r | rr | out | $\widehat{r}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $c_0$ | $t_0$ | | $[Xa_1, Xa_2]$ | | | | | $[Xe_3, Xe_4]$ | | $\emptyset$ |
| $c_6$ | $t_8$ | $Xa_1$ | $[Xa_2]$ | | $Xe_3$ | | $[Xe_4]$ | $[Xe_3, Xe_4]$ | $[\,]$ | $\emptyset$ |
| $\underline{c_7}$ | $t_9$ | $Xa_1$ | $[Xa_2]$ | | $Xe_5$:$Xe_6$ | | $[Xe_4]$ | $[(Xe_5$:$Xe_6), Xe_4]$ | $[\,]$ | $\emptyset$ |
| $c_8$ | $t_{10}$ | $Xa_1$ | $[Xa_2]$ | $Xa_7$ | $Xa_7$ | | $[Xe_4]$ | $[Xa_7, Xe_4]$ | $[\,]$ | $\emptyset$ |
| $c_9$ | $t_{11}$ | $Xa_1$ | $[Xa_2]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[\,]$ | $\emptyset$ |
| $c_{10}$ | $t_{12}$ | $Xa_7$ | $[Xa_2]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[\,]$ | $\emptyset$ |
| $c_{17}$ | $t_{11}$ | $Xa_2$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xe_4]$ | $\emptyset$ |
| $c_{18}$ | $t_{12}$ | $Xa_7$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xe_4]$ | $\emptyset$ |
| $\underline{c_{20}}$ | $t_6$ | | $[\,]$ | | | | | $[Xa_7, Xe_4]$ | $[Xe_4, Xe_4]$ | $\emptyset$ |
| $c_{21}$ | $t_{13}$ | $Xa_2$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xe_4]$ | $\widehat{r}_{21}$ |
| $\underline{c_{25}}$ | $t_6$ | | $[\,]$ | | | | | $[Xa_7, Xe_4]$ | $[Xa_2, Xe_4]$ | $\widehat{r}_{21}$ |
| $c_{26}$ | $t_{13}$ | $Xa_1$ | $[Xa_2]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[\,]$ | $\widehat{r}_{26}$ |
| $c_{35}$ | $t_{11}$ | $Xa_2$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xa_1]$ | $\widehat{r}_{26}$ |
| $c_{36}$ | $t_{12}$ | $Xa_7$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xa_1]$ | $\widehat{r}_{26}$ |
| $\underline{c_{38}}$ | $t_6$ | | $[\,]$ | | | | | $[Xa_7, Xe_4]$ | $[Xe_4, Xa_1]$ | $\widehat{r}_{26}$ |
| $c_{39}$ | $t_{13}$ | $Xa_2$ | $[\,]$ | $Xa_7$ | $Xa_7$ | $Xe_4$ | $[\,]$ | $[Xa_7, Xe_4]$ | $[Xa_1]$ | $\widehat{r}_{39}$ |
| $\underline{c_{43}}$ | $t_6$ | | $[\,]$ | | | | | $[Xa_7, Xe_4]$ | $[Xa_2, Xa_1]$ | $\widehat{r}_{39}$ |

where $t_0$ = (**call** findrep [s, rr]) $\quad \widehat{r}_{21} = \{(Xa_2 \# Xa_7)\}$
$\widehat{r}_{26} = \{(Xa_1 \# Xa_7)\} \quad \widehat{r}_{39} = \{(Xa_1 \# Xa_7), (Xa_2 \# Xa_7)\}$

**Fig. 4.** Selected configurations for program ‘findrep’

All subsequent configurations at branching nodes are listed in Fig. 4. Each row shows the name of the configuration, the term, the contents of the c-environment and the restriction. Underlined names denote terminal nodes. If a program variable does not occur in a c-environment then its entry is empty.

Tracing starts in the root, and then proceeds with one of the successor configurations depending on the shape of the values in the c-environment. The first test we encounter in our program is (**cons?** s h s _) in term $t_2$. It tests whether the value of program variable s is a pair. Since s is bound to list $[Xa_1, Xa_2]$, the test can be decided. After the first few steps we obtain a linear sequence of configurations from $c_0$ to $c_6$:

$$\begin{tikzcd} c_0 \arrow[d, "*"] \\ c_6 \end{tikzcd}$$

The next test is (**cons?** h _ _ b) in term $t_8$ of configuration $c_6$. Since program variable h is bound to c-variable $Xe_3$ (Fig. 4, $c_6$), which represents an arbitrary value, we have to consider two possibilities: $Xe_3$ is a pair of the form $Xe_5$:$Xe_6$ or an atom $Xa_7$. These assumptions lead to two new configurations, $c_7$ and $c_8$. They are expressed by attaching substitutions $Xe_3 \mapsto Xe_5$:$Xe_6$ and $Xe_3 \mapsto Xa_7$ to the corresponding edges. We call such a pair a *split*.

$$\begin{tikzcd}
& c_0 \arrow[d, "*"] & \\
& c_6 \arrow[dl, bend right, "Xe_3 \mapsto Xe_5 : Xe_6"'] \arrow[dr, bend left, "Xe_3 \mapsto Xa_7"] & \\
c_7 & & c_8
\end{tikzcd}$$

Configuration $c_7$ is a terminal node with output $t_9$ = 'Error:atom_expected. Tracing configuration $c_8$ one step leads to configuration $c_9$. The test (**eqa?** a b) in term $t_{11}$ of configuration $c_9$ depends on two unknown atoms represented by $Xa_1$ and $Xa_7$ (Fig. 4, $c_9$). Assuming that both atoms are identical, expressed by substitution $Xa_1 \mapsto Xa_7$, leads to configuration $c_{10}$. Assuming that the atoms are not equal, expressed by restriction $(Xa_1 \mathbin{\#} Xa_7)$, leads to configuration $c_{26}$:

$$\begin{tikzcd}
& c_0 \arrow[d, "*"] & & \\
& c_6 \arrow[dl, bend right, "Xe_3 \mapsto Xe_5 : Xe_6"'] \arrow[dr, bend left, "Xe_3 \mapsto Xa_7"] & & \\
c_7 & & c_8 \arrow[d] & \\
& & c_9 \arrow[dl, bend right, "Xa_1 \mapsto Xa_7"'] \arrow[dr, bend left, "(Xa_1 \mathbin{\#} Xa_7)"] & \\
& c_{10} & & c_{26}
\end{tikzcd}$$

Repeating these steps leads to a finite tree (in general, the tree may be infinite). Informally, the perfect process tree represents all computation traces of program findrep with $cls_{in}$, where branchings in the tree correspond to different assumptions about the c-variables in $cls_{in}$. Each configuration in the tree represents a set of computation states, and each edge corresponds to a set of transitions. In each branching configuration, two contractions are performed which split the set of states into two disjoint sets (see [29]). The complete tree is shown in Fig. 3.

We use only *perfect splits* in the construction of a process tree, hence its name. A perfect split guarantees that no elements will be lost, and no elements will be added when partitioning a set. The four perfect splits needed for tracing TSG-programs are

1. $(\kappa_{\mathsf{id}}, \kappa_{\mathsf{contra}})$
2. $([Xa_1 \mapsto \text{'A}], \{(Xa_1 \mathbin{\#} \text{'A})\})$
3. $([Xa_1 \mapsto Xa_2], \{(Xa_1 \mathbin{\#} Xa_2)\})$
4. $([Xe_3 \mapsto Xa^{\diamond}], [Xe_3 \mapsto Xe^{\diamond}_h : Xe^{\diamond}_t])$

where $\kappa_{\mathsf{id}}$ is an empty substitution, $\kappa_{\mathsf{contra}}$ is a restriction with a contradiction and 'A is an arbitrary atom. C-variables $Xa_1$, $Xa_2$ and $Xe_3$ occur in the c-expression we split, and $Xa^{\diamond}$, $Xe^{\diamond}_h$ and $Xe^{\diamond}_t$ are "fresh" c-variables.

**2. Tabulation** To build a table of input-output pairs, we follow each path from the root of the tree to a terminal node. All contractions encountered on such a path are applied to $cls_{in}$, and the new class $cls_i$ is entered in the table (Fig. 5) together with the output expression $\widehat{d_i}$ of the corresponding terminal node.

Each class $cls_i$ in the table represents a set of input values all of which lead to an output value that lies in $\widehat{d_i}$. Since all splits in the tree are perfect, $\lceil cls_{in} \rceil$ is partitioned into disjoint sets of input values: $\lceil cls_i \rceil \cap \lceil cls_j \rceil = \emptyset$ where $0 < i < j$. This means that each input value lies, at most, in one class $cls_i$. Note that the partitioning can also be carried out while constructing the tree. In general, when the tree has infinitely many terminal nodes, the table contains infinitely many entries.

| Input class $cls_i = \langle \widehat{ds_i}, \widehat{r_i} \rangle$ | Output expression $\widehat{d_i}$ |
|---|---|
| $\langle [[Xa_1, Xa_2], [(Xe_5{:}Xe_6), Xe_4]], \emptyset \rangle$ | 'Error:atom_expected |
| $\langle [[Xa_7, Xa_7], [Xa_7, Xe_4]], \emptyset \rangle$ | $[Xe_4, Xe_4]$ |
| $\langle [[Xa_7, Xa_2], [Xa_7, Xe_4]], \{(Xa_2 \# Xa_7)\} \rangle$ | $[Xa_2, Xe_4]$ |
| $\langle [[Xa_1, Xa_7], [Xa_7, Xe_4]], \{(Xa_1 \# Xa_7)\} \rangle$ | $[Xe_4, Xa_1]$ |
| $\langle [[Xa_1, Xa_2], [Xa_7, Xe_4]], \{(Xa_1 \# Xa_7), (Xa_2 \# Xa_7)\} \rangle$ | $[Xa_2, Xa_1]$ |

**Fig. 5.** Tabulation of program 'findrep' for $cls_{in} = \langle [[Xa_3, Xa_4], [Xe_3, Xe_4]], \emptyset \rangle$

**3. Inversion** Finally, we extract the solution to our inversion problem by intersecting each entry in the table with the given io-class $cls_{io}$. This can be done by unifying each pair $(\widehat{ds_i}, \widehat{d_i})$ with $(\widehat{ds_{in}}, \widehat{d_{out}})$, where $cls_{io} = \langle (\widehat{ds_{in}}, \widehat{d_{out}}), \widehat{r_{io}} \rangle$. If the unification succeeds with the most general unifier $\theta$, and the restriction $\widehat{r} = (\widehat{r_i} \cup \widehat{r_{io}})/\theta$ contains no contradiction, then pair $(\theta, \widehat{r})$ is in the answer *ans*.

Three examples are given in Fig. 6. They show the application of URA to three input-output classes and the corresponding solutions. In the first case, given output $\widehat{d_{out}} = [3, 1]$, we obtain three substitution-restriction pairs from the five entries in the table in Fig. 6 (unification of $[3, 1]$ with $[Xe_4, Xe_4]$ does not succeed, neither does unification of $[3, 1]$ with 'Error:atom_expected).

In the second case, given output $\widehat{d_{out}}$ = 'Error:pair_expected, the answer tells us which input values lead to the undesired error (namely, when $Xe_3$ in the replacement list is a pair of values). In the last case, given output $\widehat{d_{out}} = [1]$, the answer is empty. There exists no input in $\lceil cls_{in} \rceil$ which leads to output [1].

**Two Important Operations** Two key operations in the development of a process tree are:

1. Applying perfect splits at branching configurations.
2. Cutting infeasible branches in the tree.

Let us discuss these two operations. The second operation, *cutting infeasible branches*, is important because an infeasible branch in a process tree is either non-terminating, or terminating in an unreachable node. When infeasible branches

1. $[\![ura]\!]\,[\text{findrep}, \langle([[Xa_1, Xa_2], [Xe_3, Xe_4]], [3,1]), \emptyset\rangle\,] = [$
   $([Xa_1 \mapsto Xa_7, Xa_2 \mapsto 3, Xe_3 \mapsto Xa_7, Xe_4 \mapsto 1],\ \{(3 \,\#\, Xa_7)\}),$
   $([Xa_1 \mapsto 1, Xa_2 \mapsto Xa_7, Xe_3 \mapsto Xa_7, Xe_4 \mapsto 3],\ \{(1 \,\#\, Xa_7)\}),$
   $([Xa_1 \mapsto 1, Xa_2 \mapsto 3, Xe_3 \mapsto Xa_7, Xe_4 \mapsto Xe_4],\ \{(1 \,\#\, Xa_7), (3 \,\#\, Xa_7)\})\ \ ]$
2. $[\![ura]\!]\,[\text{findrep}, \langle([[Xa_1, Xa_2], [Xe_3, Xe_4]], \text{'Error:atom\_expected}), \emptyset\rangle\,] = [$
   $([Xa_1 \mapsto Xa_1, Xa_2 \mapsto Xa_2, Xe_3 \mapsto Xe_5{:}Xe_6, Xe_4 \mapsto Xe_4],\ \emptyset)\ \ ]$
3. $[\![ura]\!]\,[\text{findrep}, \langle([[Xa_1, Xa_2], [Xe_3, Xe_4]], [1]), \emptyset\rangle\,] = [\,]$

**Fig. 6.** Solutions for program ‘findrep’ and three requests

are not detected at branching points, the correctness of the solution can be guaranteed, but inverse computation becomes less terminating and less efficient. The risk of entering non-terminating branches which are infeasible makes inverse computation less terminating (but completeness of the solution can be preserved). A terminal node reached via an infeasible branch can only be associated with an empty set of input (but soundness of the solution is preserved). In both cases unnecessary work is performed.

The correctness of the solution cannot be guaranteed without the first operation, *applying perfect splits*, because the missing information can lead to a situation where an empty set of input cannot be detected, neither during the development of the tree nor in the solution. Suppose we reach a terminal node, then the input class $cls_i$ associated with that terminal node may tell us that there exists an input which computes a certain output, even though this is not true.

In short, the first operation is essential to guarantee the correctness of the solution and the second operation improves the termination and efficiency of the algorithm. When building a process tree, we perform both operations (see example ‘findrep’ above). This is possible because the set representation we use expresses structural properties of values (S-expressions) and all tests on it which occur during the development of a process tree are decidable.

Extensions to other value domains may involve the use of constraint systems [37] or theorem proving as in [13]. When the underlying logic of the set representation is not decidable for certain logic formulas (or too time consuming to prove), infeasible branches cannot always be detected. For example, this is the case when we use a perfect tree developer for Lisp based on generalized partial computation (GPC) [13]. In this case, the solution of inverse computation may contain elements which represent empty sets of input, but the solution is correct.

**Termination** In general, inverse computation is undecidable, so an algorithm for inverse computation cannot be sound, complete, and terminating at the same time. Our algorithm is sound and complete with respect to the solutions defined by a given program, but not always terminating. If a source program terminates on a given input and produces the desired output, our algorithm will find that input. Each such input will be found in finite time [5].

```
(define inorder [t]                      (define center [rest, out]
   (call tree [t, [ ], [ ]]))               (if (cons? rest cr rest _)
                                               (if (cons? cr center r _)
(define tree [t, rest, out]                       (call tree [r, rest, (center:out)])
   (if (cons? t l cr leaf)                        'Error:tree_expected)
      (call tree [l, (cr:rest), out])          out))
      (call center [rest, (leaf:out)])))
```

**Fig. 7.** TSG-program 'inorder'

Inverse computation does not always terminate because the search for inputs can continue infinitely, even when the number of inputs that lead to the desired output is finite (*e.g.*, the search for a solution continue along an infinite branch in the process tree). Our algorithm terminates *iff* the process tree is finite [5].

**Algorithmic Aspects** The algorithm for inverse computation is fully implemented [5] in Haskell and serves our experimental purposes quite well. The program constructs, breadth-first and lazily, a perfect process tree. In particular, Haskell's lazy evaluation strategy allowed us to use a modular approach very close to the organization shown in Fig. 1 (where the development of the perfect process tree, the tabulation, and the inversion of the table are conveniently separated). A more efficient implementation may merge these modules, and other designs may make the algorithm 'more' terminating (*e.g.*, by detecting certain finite solution sets or cutting more infinite branches).

In general, since the Universal Resolving Algorithm explores sets of program traces, inverse computation of a program $p$ using it will be more efficient than the generate-and-test approach [38] (which generates all possible ground input and tests the corresponding output), but less efficient than running the corresponding (non-trivial) inverse program $p^{-1}$. This corresponds to the familiar trade-off between interpretation and translation of programs.

## 5 Demonstration

This section shows three examples of inverse computation including inverse computation of an interpreter for a small imperative programming language.

### 5.1 Inorder Traversal of a Binary Tree

Consider a program that traverses a binary tree and returns a list of its nodes. Given a list of nodes, inverse computation then produces all binary trees that lead to that list. The classic example is to construct a binary tree given its inorder and preorder traversal [8, 51]. Our example uses the inorder traversal [25].

The program 'inorder' is written in TSG (Fig. 7). It performs an inorder traversal of a binary tree and, for simplicity, returns the list of nodes in reversed

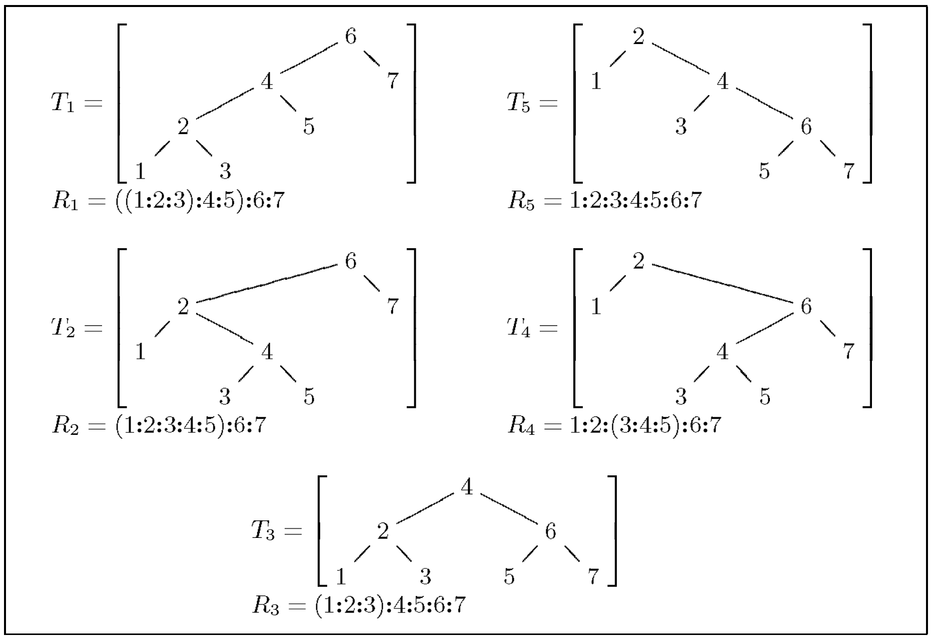


**Fig. 8.** Program 'inorder': representation of binary trees

order. Binary trees are represented by S-expressions. For example, 1:2:3 represents a binary tree with left child 1, root 2 and right child 3. The representation ($R_i$) of all five binary trees ($T_i$) that can be built from seven nodes is shown in Fig. 8. For all $R_i, i = 1...5$, we have:

$$[\![\text{inorder}]\!]\, R_i = [7, 6, 5, 4, 3, 2, 1]$$

In other words, the inorder traversal of all binary trees $T_i$ returns the same node list. Given this node list, inverse computation of the program 'inorder' by URA returns all five binary trees which can be built from that node list. URA finds all answers, but does not terminate. It continues the search after all answers have been found. The time[4] to find the five binary trees is 0.13 sec.

$$
\begin{array}{l}
[\![ura]\!]\, [\text{inorder}, \langle([Xe_1], [7,6,5,4,3,2,1]), \emptyset\rangle] = [\\
\quad ([Xe_1 \mapsto ((1{:}2{:}3){:}4{:}5){:}6{:}7],\ \emptyset), \quad ([Xe_1 \mapsto (1{:}2{:}3{:}4{:}5){:}6{:}7],\ \emptyset),\\
\quad ([Xe_1 \mapsto (1{:}2{:}3){:}4{:}5{:}6{:}7],\ \emptyset), \quad ([Xe_1 \mapsto 1{:}2{:}(3{:}4{:}5){:}6{:}7],\ \emptyset),\\
\quad ([Xe_1 \mapsto 1{:}2{:}3{:}4{:}5{:}6{:}7],\ \emptyset),\\
\quad \ldots
\end{array}
$$

[4] All running times for Intel Pentium-IV-1.4GHz, RAM 512MB, MS Windows Me, Red Hat Cygwin 1.3.4-2 and The Glorious Glasgow Haskell Compilation System 5.02.1.

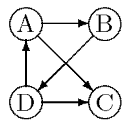
```
(define isWalk[w, g]
  (call wloop[w, g, [ ], w]))

(define wloop[w, g, pw, cc]
  (if (cons? w wh wt a_)
    (if (cons? wh e_ e_ ac)
      'Error:atom_expected
      (call check_pw[pw, ac, wt, g, pw, cc]))
    'True))

(define check_cc[x, ac, w, g, pw, cc]
  (if (cons? x xh xt a_)
    (if (cons? xh e_ e_ ax)
      'Error:bad_graph_definition
      (if (eqa? ax ac)
        (call new_cc[g, ac, w, g, pw])
        (call check_cc[xt, ac, w, g, pw, cc])))
    'False))

(define check_pw[x, ac, w, g, pw, cc]
  (if (cons? x xh xt a_)
    (if (cons? xh e_ e_ ax)
      'Error:internal
      (if (eqa? ax ac)
        'False
        (call check_pw[xt, ac, w, g, pw, cc])))
    (call check_cc[cc, ac, w, g, pw, cc])))

(define new_cc[x, ac, w, g, pw]
  (if (cons? x xh xt a_)
    (if (cons? xh xhh cc a_)
      (if (cons? xhh e_ e_ ax)
        'Error:bad_graph_definition
        (if (eqa? ax ac)
          (call wloop[w, g, (ac:pw), cc])
          (call new_cc[xt, ac, w, g, pw])))
      'Error:bad_graph_definition)
    'False))
```

**Fig. 9.** TSG-program 'isWalk'

## 5.2 Acyclic Walk in a Directed Graph

As another example, consider a program that checks whether a walk can be realized without cycles in a directed graph. The program is written as a predicate that takes a walk and a graph as input, and returns 'True, 'False, or an error atom as output. The program 'isWalk' is written in TSG (Fig. 9). It takes a walk as a list of nodes, and a directed graph as an association list where the first element is the node and the remaining elements are the reachable nodes:

gr = [ ['A, 'B, 'C],
       ['B, 'D],
       ['C],
       ['D, 'A, 'C] ]



⟦isWalk⟧ [['A], gr] = 'True
⟦isWalk⟧ [['A, 'B, 'D], gr] = 'True
⟦isWalk⟧ [['A, 'B, 'D, 'A], gr] = 'False
⟦isWalk⟧ [[('B: 'D)], gr] = 'Error:atom_expected

Given a graph and one of the outputs, inverse computation finds all walks in the graph which produce the desired output. We consider two tasks for inverse computation. Here the perfect process tree is always finite.

**Task 1**: Find all cycle-free walks in graph gr. URA enumerates all 17 answers and terminates. The time to find the answers is 0.15 sec. By convention a walk $Xe_1$ is terminated by an atom 'Nil. The answers show that, in fact, it can be any atom without changing the output of the program.

$[\![ura]\!]$ [isWalk, ⟨([$Xe_1$, gr], 'True), ∅⟩] = [
    ([$Xe_1 \mapsto Xa_4$], ∅),        ([$Xe_1 \mapsto$ 'A:$Xa_{10}$], ∅),

$$\begin{array}{ll}
([Xe_1 \mapsto \text{'B:}Xa_{10}],\ \emptyset), & ([Xe_1 \mapsto \text{'C:}Xa_{10}],\ \emptyset), \\
([Xe_1 \mapsto \text{'D:}Xa_{10}],\ \emptyset), & ([Xe_1 \mapsto \text{'A:'B:}Xa_{16}],\ \emptyset), \\
([Xe_1 \mapsto \text{'A:'C:}Xa_{16}],\ \emptyset), & ([Xe_1 \mapsto \text{'D:'A:}Xa_{16}],\ \emptyset), \\
([Xe_1 \mapsto \text{'B:'D:}Xa_{16}],\ \emptyset), & ([Xe_1 \mapsto \text{'D:'C:}Xa_{16}],\ \emptyset), \\
([Xe_1 \mapsto \text{'A:'B:'D:}Xa_{22}],\ \emptyset), & ([Xe_1 \mapsto \text{'B:'D:'A:}Xa_{22}],\ \emptyset), \\
([Xe_1 \mapsto \text{'D:'A:'B:}Xa_{22}],\ \emptyset), & ([Xe_1 \mapsto \text{'D:'A:'C:}Xa_{22}],\ \emptyset), \\
([Xe_1 \mapsto \text{'B:'D:'C:}Xa_{22}],\ \emptyset), & ([Xe_1 \mapsto \text{'A:'B:'D:'C:}Xa_{28}],\ \emptyset), \\
([Xe_1 \mapsto \text{'B:'D:'A:'C:}Xa_{28}],\ \emptyset)\ ] &
\end{array}$$

**Task 2**: Find all values which do not represent cycle-free walks in the directed graph gr. URA enumerates 54 answers (classes not shown) and, interestingly, also terminates. These classes represent all (infinitely many) values which do not represent cycle-free walks in gr. The time to find the answers is 0.17 sec.

$$[\![ura]\!]\ [\text{isWalk}, \langle(([Xe_1, \text{gr}], \text{'False}), \emptyset\rangle] = [\ ...\textit{omitted}...\ ]$$

## 5.3 Interpreter for an Imperative Language

Consider the small imperative programming language MP [45] with assignments, conditionals, and while-loops. An MP-program consists of a parameter list, a variable declaration and a sequence of statements. The value domain is the set of S-expressions. An MP-program operates over a global store. The semantics is conventional Pascal-style semantics.

We implemented an MP-interpreter, intMP, in TSG (309 lines of pretty-printed program text; 30 functions in TSG) and rewrote all three example programs in MP (findrep, inorder, isWalk). The three MP-programs are shown in Figs. 10, 11, and 12. The text of the MP-interpreter is too long to be included.

In order to compare the application of URA to the three MP-programs (via the MP-interpreter) with the direct application of URA to the corresponding TSG-programs, we repeated the six experiments from the sections above.

The direct application of URA to a TSG-program *prog* takes the form:

$$[\![ura]\!]\ [prog, \langle(\widehat{ds}_{in}, \widehat{d}_{out}), \emptyset\rangle] = ans\,.$$

Two-level inverse computation of the corresponding MP-program *progMP* via the MP-interpreter intMP is performed by the following application which is an implementation of Eq. 15:

$$[\![ura]\!]\ [\text{intMP}, \langle(([progMP, \widehat{ds}_{in}], \widehat{d}_{out}), \emptyset\rangle] = ans'\,.$$

The values for *progMP*, $\widehat{ds}_{in}$, $\widehat{d}_{out}$, and the running times (in seconds) for one-level ($t_1$) and two-level ($t_2$) inverse computations are as follows:

```
program
    parameters input, find_rep;
    variables f_r, what, head, result;
begin
    while (input) begin
        head := car(input);
        input := cdr(input);
        f_r := find_rep;
        while (f_r) begin
            what := car(f_r);
            f_r := cdr(f_r);
            if (isEqual?(head, what)) begin
                head := car(f_r);
                f_r := []; /* break f_r-loop */
            end else f_r := cdr(f_r);
        end;
        result := cons(head, result);
    end;
    return result;
end.
```

**Fig. 10.** MP-program 'findrepMP'

| *progMP* | $\widehat{ds}_{in}$ | $\widehat{d}_{out}$ | $t_1$ | $t_2$ | $t_2/t_1$ |
|---|---|---|---|---|---|
| findrep | $[[Xa_1, Xa_2], [Xe_3, Xe_4]]$ | $[3, 1]$ | 0.15 | 0.36 | 2.40 |
| findrep | $[[Xa_1, Xa_2], [Xe_3, Xe_4]]$ | 'Error:atom_expected | 0.11 | 0.33 | 3.00 |
| findrep | $[[Xa_1, Xa_2], [Xe_3, Xe_4]]$ | $[1]$ | 0.08 | 0.33 | 4.13 |
| inorder | $[Xe_1]$ | $[7, 6, 5, 4, 3, 2, 1]$ | 0.13 | 3.55 | 27.31 |
| isWalk | $[Xe_1, \text{gr}]$ | 'True | 0.15 | 5.00 | 33.33 |
| isWalk | $[Xe_1, \text{gr}]$ | 'False | 0.17 | 5.12 | 30.12 |

In each case, the answer $ans'$ obtained via the MP-interpreter was identical to the answer $ans$ obtained by the direct application of URA – modulo reordering of elements in the printed list. Such a close correspondence between the answers may not always be the case, but as our experiments indicate, it is not infeasible either. As our results show, the answer of two-level inverse computation can have the same quality as the answer of one-level inverse computation. On the other hand, and this is what can be expected, the extra level of interpretation increases the running time by an order of magnitude. Optimization by program specialization outlined in Sect. 2 has the potential to reduce the running times (*e.g.*, as suggested in Eq. 29). This will be a task for future work.

```
program
    parameters t;
    variables rest, out;
begin
    while (true) begin
        if (t) begin
            rest := cons(cdr(t), rest); /* center and right */
            t := car(t); /* left */
        end else begin /* t is leaf */
            out := cons(t, out);
            if (rest)
                if (car(rest)) begin
                    out := cons(car(car(rest)), out); /* center */
                    t := cdr(car(rest)); /* right */
                    rest := cdr(rest);
                end else return 'Error:tree_expected
            else return out;
        end;
    end;
end.
```

**Fig. 11.** MP-program 'inorderMP'

## 6 Related Work

An early result [48] of inverse computation in a functional language was obtained in 1972 when *driving* [50], a unification-based program transformation technique, was shown to perform subtraction given an algorithm for binary addition (see [1, 22]). The Universal Resolving Algorithm presented in this paper is derived from *perfect driving* [15] and is combined with a mechanical extraction of the answers (*cf.* [1, 42]) giving our algorithm the power comparable to SLD-resolution, but for a first-order, functional language (*cf.* [20]). The complete algorithm is given in [5]. A program analysis for inverting programs which makes use of *positive driving* [46] was proposed in [44]. The use of driving for theorem proving is studied in [49] and its relation to partial evaluation in [29]. Reference [21] contains a detailed bibliography on driving and supercompilation.

Logic programming inherently supports inverse computation [34]. The use of an appropriate inference procedure permits the determination of any computable answer [36]. It is not surprising that the capabilities of logic programming provided the foundation for many applications [47] in artificial intelligence, program verification and logical reasoning. Connections between logic programming, inverse computation and metacomputation are discussed in [49, 1]; see also [21, 4]. Driving and partial deduction, a technique for program specialization in logic programming, were related in [20].

```
program
    parameters w, g;
    variables c, cc, pw, x, cont;
begin
    pw := []; /* prefix of w */
    cc := w;
    while (w) begin
        c := car(w); /* current node */
        x := pw;
        while (x) /* check loops: c ‘elem‘ pw */
            if (isEqual?(car(x), c)) then return ’False
            else x := cdr(x);
        x := cc;
        cont := true;
        while (cont) /* check: c ‘elem‘ cc */
            if (x) then
                if (isEqual?(car(x),c)) then cont := false
                else x := cdr(x);
            else return ’False;
        x := g;
        cont := true;
        while (cont) /* search c in g, compute next cc */
            if (x) then
                if (isEqual?(car(car(x)), c) then begin
                    cc := cdr(car(x)); /* next c must be ‘elem‘ cc */
                    cont := false;
                end
                else x := cdr(x);
            else return ’False;
        pw := cons(c, pw);
        w := cdr(w);
    end;
    return ’True;
end.
```

**Fig. 12.** MP-program ‘isWalkMP’

Similar to ordinary programming, there exists no single programming paradigm that would satisfy all needs of inverse programming. New languages emerge as new problems are approached. It is therefore important to develop inversion methods also outside the domain of logic programming. Recently, work in this direction has been done on the integration of the functional and logic programming paradigm using narrowing, a unification-based goal-solving mechanism [26]; for a survey see [6].

Most of the work on program inversion deals with imperative languages (*e.g.*, [8, 10, 24, 25, 39, 51]). Inversion of functional programs has been studied in [7, 11, 27, 31, 33, 39, 42]. Experiments with program inversion using specialization systems are reported in [22, 40, 19, 35].

## 7 Conclusion

We presented two tools for inverse programming (inverse interpreter, inverse translator) and showed how they can be related by program specialization and program composition. We described an algorithm for inverse computation in a first-order functional language, discussed its organization and structure, and illustrated our implementation with several examples. Our results show that inverse computation can be performed in a functional programming language and ported to other programming languages via interpreters. We demonstrated this idea for a fragment of an imperative programming language.

The Universal Resolving Algorithm is fully implemented in Haskell which serves our experimental purposes quite well. In particular, Haskell's lazy evaluation strategy allowed us to use a modular approach very close to the theoretical definition of the algorithm (where the development of a perfect process trees, the tabulation, and the inversion of the table are conveniently separated). Clearly, more efficient implementations exist.

Methods for detecting finite solution sets and cutting infinite branches can make the algorithm 'more' terminating which will be useful for a practical application of the method. Techniques from program transformation and logic programming may prove to be useful in this context. Since it is impossible to design an algorithm for inverse computation that always terminates, even when a terminating inverse program is known to exist, we can only expect approximate solutions that do 'better' in terms of efficiency and termination. More work is needed to study the portability of the algorithm to realistic programming languages as suggested by the semantics modifier approach.

*Acknowledgements* The authors thank the reviewers for thorough and careful reading of the submitted paper and for providing many helpful comments.

## References

[1] S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).

[2] S. M. Abramov, R. Glück. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor, S. Prasad (eds.), *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 1974, 201–213. Springer-Verlag, 2000.

[3] S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 1837, 187–212. Springer-Verlag, 2000.

[4] S. M. Abramov, R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171–211, 2001.

[5] S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.

[6] E. Albert, G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.

[7] R. Bird, O. de Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.

[8] W. Chen, J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15:1–13, 1990.

[9] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[10] E. W. Dijkstra. EWD671: Program inversion. In *Selected Writings on Computing: A Personal Perspective*, 351–354. Springer-Verlag, 1982.

[11] D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, 219–221. Morgan Kaufmann, Inc., 1985.

[12] Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[13] Y. Futamura, K. Nogi. Generalized partial computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 133–151. North-Holland, 1988.

[14] P. C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of Research and Development*, 4:28–35, 1960.

[15] R. Glück, A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, A. Rauzy (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.

[16] R. Glück, A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl (ed.), *Cybernetics and Systems '94*, Vol. 2, 1563–1570. World Scientific, 1994.

[17] R. Glück, A. V. Klimov. A regeneration scheme for generating extensions. *Information Processing Letters*, 62(3):127–134, 1997.

[18] R. Glück, A. V. Klimov. On the degeneration of program generators by program composition. *New Generation Computing*, 16(1):75–95, 1998.

[19] R. Glück, M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification (extended abstract). In D. Bjørner, M. Broy, A. V. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 93–100. Springer-Verlag, 2000.

[20] R. Glück, M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, 165–181. Springer-Verlag, 1994.

[21] R. Glück, M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 137–160. Springer-Verlag, 1996.

[22] R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.

[23] C. Green. Application of theorem proving to problem solving. In D. E. Walker, L. M. Norton (eds.), *Int. Joint Conference on Artificial Intelligence (IJCAI-69)*, 219–239. William Kaufmann, Inc., 1969.

[24] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[25] D. Gries, J. L. A. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E. W. Dijkstra (ed.), *Formal Development of Programs and Proofs*, 37–42. Addison Wesley, 1990.

[26] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[27] P. G. Harrison, H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.

[28] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.

[29] N. D. Jones. The essence of program transformation by partial evaluation and driving. In N. D. Jones, M. Hagiya, M. Sato (eds.), *Logic, Language and Computation*, LNCS 792, 206–224. Springer-Verlag, 1994.

[30] N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[31] H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications (RTA'89)*, LNCS 355, 564–568. Springer-Verlag, 1989.

[32] A. V. Klimov, S. A. Romanenko. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples). Preprint 71, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987. (in Russian).

[33] R. E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81)*, 1007–1009. William Kaufmann, Inc., 1981.

[34] R. Kowalski. Predicate logic as programming language. In J. L. Rosenfeld (ed.), *Information Processing 74*, 569–574. North-Holland, 1974.

[35] M. Leuschel, T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi (ed.), *Logic-Based Program Synthesis and Transformation. Proceedings*, LNCS 1817, 62–81. Springer-Verlag, 2000.

[36] J. W. Lloyd. *Foundations of Logic Programming. Second, extended edition*. Springer-Verlag, 1987.

[37] K. Marriott, P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[38] J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon, J. McCarthy (eds.), *Automata Studies*, 177–181. Princeton University Press, 1956.

[39] S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.

[40] A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 249–260. Springer-Verlag, 1996.

[41] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[42] A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.

[43] J. P. Secher, M. H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, A. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 113–127. Springer-Verlag, 2000.

[44] J. P. Secher, M. H. Sørensen. From checking to inference via driving and DAG grammars. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 41–51. ACM Press, 2002.
[45] P. Sestoft. The structure of a self-applicable partial evaluator. Technical Report 85/11, DIKU, University of Copenhagen, Denmark, Nov. 1985.
[46] M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
[47] L. Sterling, E. Shapiro. *The Art of Prolog. Second Edition.* MIT Press, 1994.
[48] V. F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods)*, 31–42, 1972. (In Russian).
[49] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker, J. van Leeuwen (eds.), *Automata, Languages and Programming*, LNCS 85, 645–657. Springer-Verlag, 1980.
[50] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
[51] J. L. A. van de Snepscheut. *What Computing is All About.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
[52] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

# A Symmetric Approach to Compilation and Decompilation

Mads Sig Ager[1], Olivier Danvy[1], and Mayer Goldberg[2]

[1] BRICS*
Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
({mads,danvy}@brics.dk)
[2] Department of Computer Science, Ben Gurion University
Be'er Sheva 84105, Israel
(gmayer@cs.bgu.ac.il)

**Abstract.** Just as an interpreter for a source language can be turned into a compiler from the source language to a target language, we observe that an interpreter for a target language can be turned into a compiler from the target language to a source language. In both cases, the key issue is the choice of whether to perform an evaluation or to emit code that represents this evaluation.
We substantiate this observation with two source interpreters and two target interpreters. We first consider a source language of arithmetic expressions and a target language for a stack machine, and then the $\lambda$-calculus and the SECD-machine language. In each case, we prove that the target-to-source compiler is a left inverse of the source-to-target compiler, i.e., that it is a decompiler.
In the context of partial evaluation, the binding-time shift of going from a source interpreter to a compiler is classically referred to as a Futamura projection. By symmetry, it seems logical to refer to the binding-time shift of going from a target interpreter to a compiler as a Futamura embedding.

*To Neil Jones, for his 60th birthday.*

## 1 Introduction

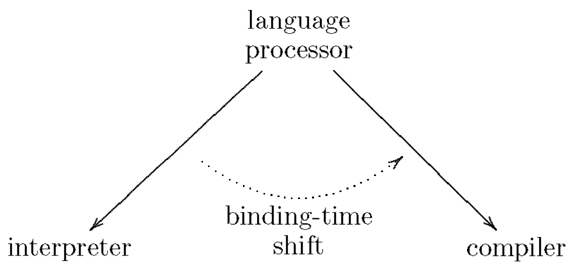
Intuitions run strong when it comes to connecting interpreters and compilers, be it by calculation [51], by derivation [47,61] or by partial evaluation [7,11,18,19,20, 31,38,42,43,39,44]. These intuitions have provided a fertile ground for two-level languages [56] and code generation [8,59,60,62]. Common to these approaches is the idea, in two-level programs, of shifting from a semantic model to a syntactic model in order to generate code: Rather than performing an evaluation in a

* Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

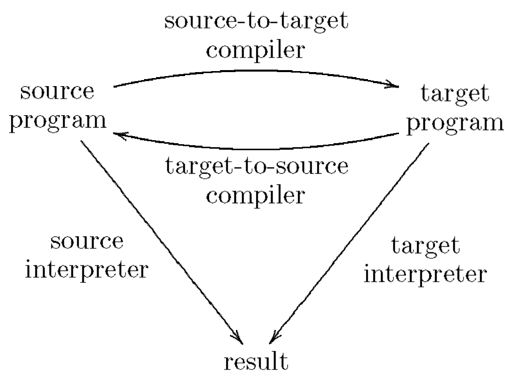
source-language interpreter, one emits target-language code that represents this evaluation, as in a compiler.

We observe that this binding-time shift directly applies to decompiling: Rather than performing an evaluation in a target-language interpreter, one can emit source-language code that represents this evaluation, as in a decompiler.

In the rest of this article, we illustrate both instances of this binding-time shift with a source language of arithmetic expressions and a target language for a stack machine (Section 2), and then with the λ-calculus and the SECD-machine language (Section 3). We stage each language processor as a core semantics, represented as an ML functor, and as interpretations, represented as ML structures. This staging corresponds to the factorized semantics of Jones and Nielson [41]. We show how instantiating each functor with elementary evaluation functions yields an interpreter and how instantiating it with elementary code-generating functions yields a compiler:



In one case, the compiler maps a source program to a target program, and in the other, it maps a target program to a source program:



In each of Sections 2 and 3, we formally prove that the target-to-source compiler is a left inverse of the source-to-target compiler, i.e., that it is a decompiler.

## 2 Arithmetic Expressions and a Stack Machine

We consider a simplified source language of arithmetic expressions and a simplified target language for a stack machine. It is straightforward to extend both languages with more arithmetic operators.

### 2.1 Staged Specification of the Source Language

The source language is as follows.

```
structure Source
= struct
    datatype exp = LIT of int
                 | PLUS of exp * exp
    type program = exp
  end
```

**The Core Semantics**

Recursive traversal of programs in the source language can be expressed generically as follows, using an ML functor. In this functor, the function `process` implements the fold function associated with the data type of the source language [6,16]. This fold function is parameterized by a structure of type `INTEGER`, which packages two types and a collection of operators corresponding to each constructor of the data type.

```
signature INTEGER
= sig
    type integer
    type result
    val lit : int -> integer
    val plus : integer * integer -> integer
    val compute : integer -> result
  end;

signature SOURCE_PROCESSOR
= sig
    type result
    val process : Source.program -> result
  end

functor Make_source_processor (structure I : INTEGER)
: SOURCE_PROCESSOR
= struct
    type result = I.result

    fun process p
        = let fun walk (Source.LIT n)
                  = I.lit n
```

```
                | walk (Source.PLUS (e1, e2))
                  = I.plus (walk e1, walk e2)
          in I.compute (walk p)
          end
  end
```

**A Code-Generation Instantiation: Source Identity**
As an example of the use of `Make_source_processor`, this functor can be instantiated to obtain the identity transformation over source programs. To this end, we specify a structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as the type of source expressions, `result` as the type of source programs, and the operators as the corresponding code-generating functions:

```
structure Integer_source_syntax : INTEGER
= struct
    type integer = Source.exp
    type result = Source.program

    fun lit n
        = Source.LIT n
    fun plus (e1, e2)
        = Source.PLUS (e1, e2)
    fun compute e
        = e
  end

structure Source_identity
= Make_source_processor (structure I = Integer_source_syntax)
```

## 2.2 Staged Specification of the Target Language

A target program is a list of instructions for a stack machine:

```
structure Target
= struct
    datatype instr = PUSH of int
                   | ADD
    type program = instr list
  end
```

**The Core Semantics**
Recursive traversal of programs in this language can be expressed generically as follows, again using an ML functor. In this functor, the function `process` implements the fold function associated with the target data type. This fold function is parameterized by a structure of type `TARGET_PARAMETERS`, which packages two types and a collection of operators corresponding to each constructor of the data type.

```
signature TARGET_PARAMETERS
= sig
    type computation
    type result
    val terminate : computation
    val push : int * computation -> computation
    val add : computation -> computation
    val compute : computation -> result
  end

signature TARGET_PROCESSOR
= sig
    type result
    val process : Target.program -> result
  end

functor Make_target_processor (structure P : TARGET_PARAMETERS)
: TARGET_PROCESSOR
= struct
    type result = P.result

    fun process p
        = let fun walk nil
                    = P.terminate
                | walk ((Target.PUSH n) :: is)
                    = P.push (n, walk is)
                | walk (Target.ADD :: is)
                  = P.add (walk is)
          in P.compute (walk p)
          end
  end
```

**A Code-Generation Instantiation: Target Identity**
As in Section 2.1, `Make_ target_processor` can be instantiated to obtain the identity transformation over target programs by defining `computation` as the type of lists of target instructions, `result` as the type of target programs, and the operators as the corresponding code-generating functions:

```
structure Target_parameters_identity : TARGET_PARAMETERS
= struct
    type computation = Target.instr list
    type result = Target.program

    val terminate = nil
    fun push (n, is)
        = (Target.PUSH n) :: is
    fun add is
        = Target.ADD :: is
    fun compute is
        = is
  end

structure Target_identity
= Make_target_processor (structure P = Target_parameters_identity)
```

## 2.3 Interpretation and Compilation for the Source Language

We instantiate `Make_source_processor` into an interpreter for the source language and into a compiler from the source language to the target language.

**An Evaluation Instantiation: Source Interpretation**
It is straightforward to instantiate the functor of Section 2.1 to obtain an interpreter. To this end, we define a structure of type `INTEGER` containing a semantic representation of integers. In this structure, both `integer` and `result` are defined as the type `int`, and the operators as the standard arithmetic operators:

```
structure Integer_semantics : INTEGER
= struct
    type integer = int
    type result = int

    fun lit n
        = n
    fun plus (n1, n2)
        = n1 + n2
    fun compute n
        = n
  end
```

We can now instantiate the functor of Section 2.1 to obtain an interpreter for the source language:

```
structure Source_int
= Make_source_processor (structure I = Integer_semantics)
```

For example, applying `Source_int.process` to the source program

```
PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40))
```

yields the integer `100`.

Compared to the identity instantiation of Section 2.1, rather than choosing a syntactic model and emitting source-language code, we choose a semantic model and carry out evaluation. The two instantiations illustrate a simple binding-time shift.

**A Code-Generation Instantiation: Source Compilation (Version 1)**
It is also straightforward to instantiate `Make_source_processor` to obtain a compiler to the target language. To this end, we implement a binding-time shift of going from an interpreter to a compiler with another structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as the type of lists of target instructions, `result` as the type of target programs, and operators as first-order code-generating functions:

```
structure Integer_target_syntax1 : INTEGER
= struct
    type integer = Target.instr list
    type result = Target.program

    fun lit n
        = [Target.PUSH n]
    fun plus (is1, is2)
        = is1 @ is2 @ [Target.ADD]
    fun compute is
        = is
  end

structure Source_cmp1
= Make_source_processor (structure I = Integer_target_syntax1)
```

For example, applying `Source_cmp1.process` to the source program

```
PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40))
```

yields the following target program:

```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

**A Code-Generation Instantiation: Source Compilation (Version 2)**
We can also instantiate `Make_source_processor` to obtain a less trivial but equivalent compiler that uses an accumulator instead of concatenating intermediate lists of instructions. To this end, we define yet another structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as a transformer of lists of target instructions, `result` as the type of target programs, and the operators as second-order code-generating functions:

```
structure Integer_target_syntax2 : INTEGER
= struct
    type integer = Target.instr list -> Target.instr list
    type result = Target.program

    fun lit n
        = (fn is => (Target.PUSH n) :: is)
    fun plus (c1, c2)
        = (fn is => c1 (c2 (Target.ADD :: is)))
    fun compute c
        = c nil
  end
```

In passing, let us stress the relation between Version 1 and Version 2 of the compiler with the following equivalent definition of Version 2, using a curried version of list construction and function composition instead of list construction and list concatenation, respectively:

```
structure Integer_target_syntax2' : INTEGER
= struct
    type integer = Target.instr list -> Target.instr list
    type result = Target.program

    fun cons x
        = (fn xs => x :: xs)

    fun lit n
        = cons (Target.PUSH n)
    fun plus (c1, c2)
        = c1 o c2 o (cons Target.ADD)
    fun compute c
        = c nil
  end
```

Either of `Integer_target_syntax2` or `Integer_target_syntax2'` can be used to obtain a compiler from the source language to the target language:

```
structure Source_cmp2
= Make_source_processor (structure I = Integer_target_syntax2)

structure Source_cmp2'
= Make_source_processor (structure I = Integer_target_syntax2')
```

## 2.4 Interpretation and Compilation for the Target Language

A target program is processed using a stack. This process is partial in that it expects the stack to be well-formed. We make it total in ML using an `option` type:

```
datatype 'a option = NONE
                   | SOME of 'a
```

When interpreting programs, the stack contains integers. According to the binding-time shift of going from an interpreter to a compiler, when compiling programs, the stack should contain representations of integers. We thus further parameterize the parameters of the target-language processor by a structure of type `INTEGER`:

```
functor Make_target_parameters (structure I : INTEGER)
: TARGET_PARAMETERS
= struct
    type computation = I.integer list -> I.integer list option
    type result = I.result option

    val terminate = (fn s => SOME s)
    fun push (n, c)
         = (fn s => c ((I.lit n) :: s))
    fun add c
        = (fn (x2 :: x1 :: xs) => c ((I.plus (x1, x2)) :: xs)
            | _ => NONE)
    fun compute c
        = (case c nil
             of (SOME (x :: nil)) => SOME (I.compute x)
              | _ => NONE)
  end
```

**An Evaluation Instantiation: Target Interpretation**
It is straightforward to instantiate `Make_target_parameters` to obtain the target parameters for an interpreter. To this end, we use the semantic representation of the integers specified in Section 2.3:

```
structure Target_parameters_semantics
= Make_target_parameters (structure I = Integer_semantics)
```

We can now instantiate the functor of Section 2.2 to obtain an interpreter for the target language:

```
structure Target_int
= Make_target_processor (structure P = Target_parameters_semantics)
```

For example, applying `Target_int.process` to the target program

```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

yields the optional integer `SOME 100`.

**A Code-Generation Instantiation: Target Compilation**
It is also straightforward to instantiate `Make_target_parameters` to obtain the target parameters for a compiler to the source language. To this end, we use the syntactic representation of the integers specified in Section 2.1:

```
structure Target_parameters_source_syntax
= Make_target_parameters (structure I = Integer_source_syntax)
```

We can now instantiate the functor of Section 2.2 to obtain a compiler from the target language to the source language:

```
structure Target_cmp
= Make_target_processor(structure P = Target_parameters_source_syntax)
```

For example, applying `Target_cmp.process` to the target program

```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

yields the following optional source program:

```
SOME (PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40)))
```

## 2.5 Properties

We successively consider the total correctness of the source compiler with respect to the source interpreter and the target interpreter, the partial correctness of the target compiler with respect to the target interpreter and the source interpreter, and the left inverseness of the source compiler and of the target compiler:

$$\begin{tikzcd}
\text{source program} \arrow[rr, bend left, "\text{source compiler}"] \arrow[dr, "\text{source interpreter}"'] & & \text{target program} \arrow[ll, bend left, "\text{target compiler}"] \arrow[dl, "\text{target interpreter}"] \\
& \text{result} &
\end{tikzcd}$$

In particular, we prove that the target compiler is a left inverse of the source compiler and therefore that it is a decompiler.

*Terminology and notation:*

- `Source_int.process` is the `process` function of the functor `Make_source_processor` of Section 2.1 instantiated with the structure `Integer_semantics` of Section 2.3. We refer to the corresponding `walk` function as `w_s_int` (for "walk function of the source interpreter").
- `Source_cmp2.process` is the `process` function of the functor `Make_source_processor` of Section 2.1 instantiated with the structure `Integer_target_syntax2` of Section 2.3. We refer to the corresponding `walk` function as `w_s_cmp` (for "walk function of the source compiler").
- `Target_int.process` is the `process` function of the functor `Make_target_processor` of Section 2.2 instantiated with the structure `Target_parameters_semantics` of Section 2.4. We refer to the corresponding `walk` function as `w_t_int` (for "walk function of the target interpreter").

– `Target_cmp.process` is the `process` function of the functor `Make_target_processor` of Section 2.2 instantiated with the structure `Target_parameters_source_syntax` of Section 2.4. We refer to the corresponding `walk` function as `w_t_cmp` (for "walk function of the target compiler").

All the functions above are pure and total, i.e., they are side-effect free and they always terminate, since they only use primitive recursion (the `process` functions in the functors `Make_source_processor` and `Make_target_processor` are fold functions).

We reason equationally on the ML syntax of the interpreters and compilers, using observational equivalence. We say that two expressions `e1` and `e2` are observationally equivalent, which we write as

$$\texttt{e1} \cong \texttt{e2}$$

whenever evaluating `e1` and `e2` in the same context yield the same result. Our equational reasoning involves unfolding function calls, which is sound for pure and total functions.

## Total Correctness of the Source Compiler

The compiler is correct if composing `Target_int.process` and `Source_cmp.process` yields the same function as `Source_int.process`. We use the following lemma as a stepping stone for proving this correctness.

**Lemma 1.** *For all ML values* `e : Source.exp`*,* `is : Target.instr list`*, and* `s : int list`*, the following observational equivalence holds:*

$$\texttt{w\_t\_int (w\_s\_cmp e is) s} \cong \texttt{w\_t\_int is ((w\_s\_int e) :: s).}$$

*Proof.* The proof is by structural induction on the source syntax (see extended version [4]).

**Theorem 1.** *For ML values* `sp : Source.program`*, the following observational equivalence holds:*

$$\texttt{Target\_int.process (Source\_cmp2.process sp)} \cong \texttt{SOME (Source\_int.process sp).}$$

*Proof.* Follows from Lemma 1 (see extended version [4]).

## Partial Correctness of the Target Compiler

As in Section 2.5, the compiler is correct if composing `Source_int.process` and `Target_cmp.process` yields the same function as `Target_int.process`. The issue, however, is more murky here because not all values of type `Target.program` are well-formed programs, as indicated by the `option` type in Section 2.4. Such ill-formed target programs are the reason why `Target_int.process` and `Target_cmp.process` may yield `NONE`. On the other hand, it is a corollary of Theorem 1 that compiling a source expression yields a well-formed target program and that interpreting a well-formed target program yields `SOME n`, for some integer `n`.

We leave the issue of partial correctness aside, and instead we turn to proving that the target compiler is a left inverse of the source compiler.

**Left Inverseness**
We use the following lemma as a stepping stone for proving that `Target_cmp.process` is a left inverse of `Source_cmp2.process` for all source expressions.

**Lemma 2.** *For all ML values* `e : Source.exp`*,* `is : Target.instr list`*, and* `s : Source.exp list`*, the following observational equivalence holds:*

$$\texttt{w\_t\_cmp (w\_s\_cmp e is) s} \cong \texttt{w\_t\_cmp is (e :: s)}.$$

*Proof.* The proof is by structural induction on the source syntax (see extended version [4]).

**Theorem 2.** *For all ML values* `sp : Source.program`*, the following observational equivalence holds:*

$$\texttt{Target\_cmp.process (Source\_cmp2.process sp)} \cong \texttt{SOME sp}.$$

*Proof.* For all ML values `sp : Source.program` and `tp : Target.program`, the following observational equivalences holds:

```
Source_cmp2.process sp
≅ (unfolding Source_cmp2.process)
Integer_target_syntax2.compute (w_s_cmp sp)
≅ (unfolding Integer_target_syntax2.compute)
w_s_cmp sp nil

Target_cmp.process tp
≅ (unfolding Target_cmp.process)
Target_parameters_source_syntax.compute (w_t_cmp tp)
≅ (unfolding Target_parameters_source_syntax.compute)
case w_t_cmp tp nil
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
```

For all ML values `sp : Source.program`, we therefore have to prove that the following observational equivalence holds:

$$\left(\begin{array}{l}\texttt{case w\_t\_cmp (w\_s\_cmp sp nil) nil} \\ \quad\texttt{of (SOME(x :: nil))=>SOME(Integer\_source\_syntax.compute x)} \\ \quad\;\texttt{| \_ => NONE}\end{array}\right) \cong \texttt{SOME sp}$$

This observational equivalence, however, follows from Lemma 2. Indeed, for all ML values `e : Source.exp`, `nil : Target.instr list`, and `nil : Source.exp list`, the observational equivalence of Lemma 2 reads as

$$\texttt{w\_t\_cmp (w\_s\_cmp e nil) nil} \cong \texttt{w\_t\_cmp nil (e :: nil)}$$

In particular,

```
w_t_cmp nil (e :: nil)
≅ (unfolding w_t_cmp)
Target_parameters_source_syntax.terminate (e :: nil)
≅ (unfolding Target_parameters_source_syntax.terminate)
(fn s => SOME s) (e :: nil)
≅ (function application)
SOME (e :: nil)
```

Since source programs are expressions and target programs are lists of instructions,

```
case w_t_cmp (w_s_cmp sp nil) nil
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
≅ (using the observational equivalence just above in context)
case SOME (sp :: nil)
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
≅ (reducing the case expression)
SOME (Integer_source_syntax.compute sp)
≅ (unfolding Integer_source_syntax.compute)
SOME sp
```

which concludes the proof.

## 2.6 Summary

We have systematically parameterized a source-language processor and a target-language processor and instantiated them into identity transformations, interpreters, and compilers. We also have shown that the target compiler is a left-inverse of the source compiler, and thus a decompiler.

Most of our instantiations hinge on a particular representation of integers—syntactic or semantic. The exception is the identity transformation over target programs, in Section 2.2, which hinges on a syntactic instantiation of target parameters. We can, however, instantiate `Make_target_parameters` with either of the syntactic interpretations of the integers in Sections 2.3 or 2.3:

```
structure Target_parameters_target_syntax1
= Make_target_parameters (structure I = Integer_target_syntax1)

structure Target_parameters_target_syntax2
= Make_target_parameters (structure I = Integer_target_syntax2)
```

We can now instantiate the functor of Section 2.2:

```
structure Target_identity1
= Make_target_processor (structure P=Target_parameters_target_syntax1)
```

```
structure Target_identity2
= Make_target_processor (structure P=Target_parameters_target_syntax2)
```

In this instantiation, the target program is processed with a stack and each component is mapped to a representation of an integer in either `Integer_target_syntax1` or `Integer_target_syntax2`, i.e., to target code. The instantiation yields a `process` function of type `Target.program -> Target.program option`. This `process` function reflects the partial correctness mentioned in Section 2.5 in that it maps any well-formed target program `p` into `SOME p` and all the other target programs into `NONE`.

Overall, we have shown that just as specializing a source-language processor can achieve compilation to a target language, specializing a target-language processor can achieve decompilation to a source language. This observation is very simple, but the authors have not seen it stated elsewhere. For example, specific efforts have been dedicated to decompiling compiled arithmetic expressions, independently of their interpretation, compilation, and execution [13, 14, 49].

## 3 Lambda-Terms and the SECD Machine

In this section we show that the symmetric approach to compilation and decompilation scales to an expression language with binding, namely the $\lambda$-calculus. We consider Henderson's version of the SECD machine [46, 35, 53].

### 3.1 Staged Specification of the Source Language

The source language is the untyped $\lambda$-calculus with integers and a plus operator. A program is a closed term.

```
structure Source
= struct
    type ide = string

    datatype term = LIT of int
                  | PLUS of term * term
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
    type program = term
  end
```

**The Core Semantics**
Recursive traversal of programs in this language can be expressed generically using an ML functor as in Section 2.1.

```
signature SOURCE_PARAMETERS
= sig
    type computation
    type result
    val lit : int -> computation
    val plus : computation * computation -> computation
    val var : int -> computation
    val lam : computation -> computation
    val app : computation * computation -> computation
    val compute : computation -> result
  end

signature SOURCE_PROCESSOR
= sig
    type result
    val process : Source.program -> result
  end

functor Make_source_processor (structure P : SOURCE_PARAMETERS)
: SOURCE_PROCESSOR
= struct
    type result = P.result

    fun process p
        = let fun walk (Source.LIT n) xs
                  = P.lit n
                | walk (Source.PLUS (t1, t2)) xs
                  = P.plus (walk t1 xs, walk t2 xs)
                | walk (Source.VAR x) xs
                  = P.var (Index.establish (x, xs))
                | walk (Source.LAM (x, t)) xs
                  = P.lam (walk t (x :: xs))
                | walk (Source.APP (t0, t1)) env
                  = P.app (walk t0 env, walk t1 env)
          in P.compute (walk p nil)
          end
  end
```

In order to account for bindings, the `walk` function threads a lexical environment `xs`. This environment is extended for each $\lambda$-abstraction and consulted for each occurrence of a variable. The lexical offset of each occurrence of a variable is established using `Index.establish`. (Given two ML values `x : Source.ide` and `xs : Source.ide list` where `x` occurs, applying `Index.establish` to `x` and `xs` yields the index of the first occurrence of `x` in `xs`.)

**A Code-Generation Instantiation: Source Identity Modulo Renaming** As an example of the use of `Make_source_processor`, this functor can be instantiated as follows to obtain the identity transformation over source programs, modulo renaming. To this end, we define a structure of type `SOURCE_PARAMETERS` where `computation` is a mapping from a list of identifiers to a source term, `result` is the type of source programs, and the operators are the corresponding code-generating functions:

```
structure Source_parameters_identity : SOURCE_PARAMETERS
= struct
    type computation = Source.ide list -> Source.term
    type result = Source.program

    fun lit n
        = (fn xs => Source.LIT n)
    fun plus (c1, c2)
        = (fn xs => Source.PLUS (c1 xs, c2 xs))
    fun var i
        = (fn xs => Source.VAR (Index.fetch (xs, i)))
    fun lam c
        = (fn xs => let val x = "x" ^ Int.toString (length xs)
                    in Source.LAM (x, c (x :: xs))
                    end)
    fun app (c0, c1)
        = (fn xs => Source.APP (c0 xs, c1 xs))
    fun compute c
        = c nil
  end

structure Source_identity
= Make_source_processor (structure P = Source_parameters_identity)
```

Fresh identifiers are needed to construct source λ-abstractions. We obtain them from the current de Bruijn level. These fresh identifiers are grouped in a list `xs` in the reverse order of their declaration. For each λ-abstraction, the list is extended, and for each occurrence of a variable, the corresponding fresh identifier is fetched using `Index.fetch`. (Given two values `i : int` and `xs : Source.ide list`, applying `Index.fetch` to `xs` and `i` fetches the corresponding identifier in `xs`.) A computation is a mapping from lists of fresh identifiers to source terms.

For example, the source term

```
LAM ("a", LAM ("b", APP (APP (LAM ("x", VAR "x"),
                              LAM ("y", VAR "y")),
                         APP (VAR "a", VAR "b"))))
```

is mapped into the following source term:

```
LAM ("x0", LAM ("x1", APP (APP (LAM ("x2",VAR "x2"),
                                LAM ("x2",VAR "x2")),
                           APP (VAR "x0",VAR "x1"))))
```

## 3.2 Staged Specification of the Target Language

A target program is a list of instructions for the SECD machine [35]:

```
structure Target
= struct
    datatype instr = PUSH of int
                   | ADD
                   | ACCESS of int
                   | CLOSE of instr list
                   | CALL
   type program = instr list
  end
```

Unlike the other interpreters considered in this article, the SECD machine is not directly defined by induction over the structure of target programs. For the sake of familiarity, we follow the canonical definition to write the target-language interpreter in Section 3.4. (Therefore, we stay away from the gymnastics of using a functor implementing a recursive descent, as in the appendix of the extended version of this article [4].)

## 3.3 Interpretation and Compilation for the Source Language

We instantiate `Make_source_processor` into an interpreter for the source language and into a compiler from the source language to the target language.

**An Evaluation Instantiation: Source Interpretation**
In order to instantiate the functor of Section 3.1 to obtain a call-by-value interpreter for the source language, we define a data type of values containing integers and functions from values to values. The computation type is then defined to be a mapping from environments, represented by lists of values, to values. The result type is defined to be values.

```
structure Source_parameters_std : SOURCE_PARAMETERS
= struct
    datatype value = INT of int
                   | FUN of value -> value option

    type computation = value list -> value option
    type result = value option

    fun lit n
        = (fn vs => SOME (INT n))
    fun plus (c1, c2)
        = (fn vs => (case (c1 vs, c2 vs)
                       of (SOME (INT n1), SOME (INT n2))
                          => SOME (INT (n1 + n2))
                        | (_, _)
                          => NONE))
    fun var i
        = (fn vs => SOME (Index.fetch (vs, i)))
    fun lam c
        = (fn vs => SOME (FUN (fn v => c (v :: vs))))
    fun app (c0, c1)
        = (fn vs => (case (c0 vs, c1 vs)
                       of (SOME (FUN f), SOME v)
                          => f v
                        | _
                          => NONE))
    fun compute c
        = c nil
  end

structure Source_int
= Make_source_processor (structure P = Source_parameters_std)
```

For example, applying `Source_int.process` to the source program

```
APP (APP (LAM ("a", LAM ("b", VAR "a")), LIT 10), LIT 20)
```

yields the optional value `SOME (INT 10)`.

**A Code-Generation Instantiation: Source Compilation (Version 1)**
It is also straightforward to instantiate `Make_source_processor` to obtain a compiler for the source language, by defining both `computation` and `result` as lists of instructions, and by defining the operators as first-order code-generating functions, as in Section 2.3:

```
structure Source_parameters_cogen1 : SOURCE_PARAMETERS
= struct
    type computation = Target.instr list
    type result = Target.program

    fun lit n
        = [Target.PUSH n]
    fun plus (is1, is2)
        = is1 @ is2 @ [Target.ADD]
    fun var n
        = [Target.ACCESS n]
    fun lam is
        = [Target.CLOSE is]
    fun app (is0, is1)
        = is0 @ is1 @ [Target.CALL]
    fun compute is
        = is
  end

structure Source_cmp1
= Make_source_processor (structure P = Source_parameters_cogen1)
```

The resulting compiler is a subset of Henderson's compiler for the SECD machine [35].

**A Code-Generation Instantiation: Source Compilation (Version 2)** As in Section 2.3, we can also instantiate `Make_source_processor` to obtain a less trivial but equivalent compiler that uses an accumulator instead of concatenating intermediate lists of instructions. To this end, we define `computation` as a transformer of lists of instructions, `result` as a program, and the operators as second-order code-generating functions:

```
structure Source_parameters_cogen2 : SOURCE_PARAMETERS
= struct
    type computation = Target.instr list -> Target.instr list
    type result = Target.program

    fun lit n
        = (fn is => (Target.PUSH n) :: is)
    fun plus (f1, f2)
        = (fn is => f1 (f2 (Target.ADD :: is)))
    fun var n
        = (fn is => (Target.ACCESS n) :: is)
    fun lam f
        = (fn is => (Target.CLOSE (f nil)) :: is)
    fun app (f1, f2)
        = (fn is => f1 (f2 (Target.CALL :: is)))
    fun compute f
        = f nil
  end

structure Source_cmp2
= Make_source_processor (structure P = Source_parameters_cogen2)
```

For example, applying `Source_cmp2.process` to the source program

```
APP (APP (LAM ("a", LAM ("b", VAR "a")), LIT 10), LIT 20)
```

yields the following target program:

```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

## 3.4 Interpretation and Compilation for the Target Language

We now turn to defining an interpreter and a compiler for SECD machine code. As already mentioned, for clarity, we refrain from factoring the two definitions through an ML functor. Instead, we present each of them on its own.

**An Evaluation Instantiation: Target Interpretation**

The interpreter for the target language is a scaled-down version of Henderson's interpreter [35], which is itself an implementation of the SECD machine [46,53]. As before, given two ML values `e : 'a list` and `i : int`, applying `Index.fetch` to `e` and `i` fetches the corresponding entry in `e`.

```
structure Target_int
= struct
    datatype value = INT of int
                   | CLOSURE of Target.instr list * value list

    (*  process : Target.program -> value option  *)
    fun process p
        = let fun walk (v :: nil, e, nil, nil)
                = SOME v
            | walk (v :: nil, e, nil, (s', e', c') :: d)
                = walk (v :: s', e', c', d)
            | walk (s, e, (Target.PUSH n) :: c, d)
                = walk ((INT n) :: s, e, c, d)
            | walk ((INT n2) :: (INT n1) :: s, e, Target.ADD :: c, d)
                = walk ((INT (n1 + n2)) :: s, e, c, d)
            | walk (s, e, (Target.ACCESS i) :: c, d)
                = walk ((Index.fetch (e, i)) :: s, e, c, d)
            | walk (s, e, (Target.CLOSE c') :: c, d)
                = walk ((CLOSURE (c', e)) :: s, e, c, d)
            | walk (a :: (CLOSURE (c', e')) :: s, e, Target.CALL :: c, d)
                = walk (nil, a :: e', c', (s, e, c) :: d)
            | walk (_, _, _, _)
                = NONE
          in walk (nil, nil, p, nil)
          end
  end
```

For example, applying `Target_int.process` to the target program

```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

yields the optional value `SOME (INT 10)`.

**A Code-Generation Instantiation: Target Compilation**

We obtain a compiler for the target language by instrumenting the SECD machine to build source terms (on the stack) instead of calculating values. Fresh identifiers are needed to construct residual $\lambda$-abstractions, and we obtain them by threading an integer.

```
structure Target_cmp
= struct
    type value = Source.term
    (*  process : Target.program -> value option  *)
    fun process p
        = let fun walk (v :: nil, e, nil, nil, g)
                = SOME v
            | walk (t :: nil, x :: e, nil, (s', e', c') :: d, g)
              = walk (Source.LAM (x, t) :: s', e', c', d, g)
            | walk (s, e, (Target.PUSH n) :: c, d, g)
              = walk ((Source.LIT n) :: s, e, c, d, g)
            | walk (t2 :: t1 :: s, e, Target.ADD :: c, d, g)
              = walk ((Source.PLUS (t1, t2)) :: s, e, c, d, g)
            | walk (s, e, (Target.ACCESS i) :: c, d, g)
              = walk ((Source.VAR (Index.fetch (e, i))) :: s, e, c, d, g)
            | walk (s, e, (Target.CLOSE c') :: c, d, g)
              = let val x = "x" ^ Int.toString g
                in walk (nil, x :: e, c', (s, e, c) :: d, g+1)
                end
            | walk (t1 :: t0 :: s, e, Target.CALL :: c, d, g)
              = walk ((Source.APP (t0, t1)) :: s, e, c, d, g)
            | walk (_, _, _, _, _)
              = NONE
          in walk (nil, nil, p, nil, 0)
          end
  end
```

– `PUSH n` and `ADD`: Pushing a number and adding two numbers implement the binding-time shift between an interpreter and a compiler: instead of treating the integers numerically, we treat them symbolically.
– `CALL`: Both the function and the argument occur on the stack; we construct the corresponding residual application and we store it on the stack.
– `CLOSE c'`: We residualize `c'` into the body of a $\lambda$-abstraction in an environment with a fresh identifier `x`. When residualization completes (second clause in the definition of `walk`), `x` is available in the environment to manufacture the complete $\lambda$-abstraction, which we store on the stack.

For example, applying `Target_cmp.process` to the target program

```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

yields the following optional source program:

```
SOME (APP (APP (LAM ("x0", LAM ("x1", VAR "x0")), LIT 10), LIT 20))
```

## 3.5 Properties

As in Section 2.5, we successively consider the total correctness of the source compiler with respect to the source interpreter and the target interpreter, the partial correctness of the target compiler with respect to the target interpreter and the source interpreter, and the left inverseness of the source compiler and of the target compiler:

$$\begin{tikzcd}
\text{source program} \arrow[rr, bend left, "\text{source compiler}"] \arrow[dr, swap, "\text{source interpreter}"] & & \text{target program} \arrow[ll, bend left, "\text{target compiler}"] \arrow[dl, "\text{target interpreter}"] \\
& \text{result} &
\end{tikzcd}$$

In particular, we prove that the target compiler is a left inverse of the source compiler and therefore that it is a decompiler.

*Terminology and notation:*

- `Source_int.process` is the `process` function of the functor `Make_source_processor` of Section 3.1 instantiated with the structure `Source_parameters_std` of Section 3.3. We refer to the corresponding `walk` function as `w_s_int` (for "walk function of the source interpreter").
- `Source_cmp2.process` is the `process` function of the functor `Make_source_processor` of Section 3.1 instantiated with the structure `Source_parameters_cogen2` of Section 3.3. We refer to the corresponding `walk` function as `w_s_cmp` (for "walk function of the source compiler").
- `Target_int.process` is the `process` function of the structure `Target_int` of Section 3.4. We refer to the corresponding `walk` function as `w_t_int` (for "walk function of the target interpreter").
- `Target_cmp.process` is the `process` function of the structure `Target_cmp` of Section 3.4. We refer to the corresponding `walk` function as `w_t_cmp` (for "walk function of the target compiler").

Among the functions above, `Source_cmp2.process` (and thus `w_s_cmp`) and `Target_cmp.process` (and thus `w_t_cmp`) are pure and total. They are pure because they have no side effects, and they terminate because they recursively traverse finite source and target programs.

As in Section 2.5, we reason equationally on the ML syntax of the interpreters and compilers, using observational equivalence.

## Total Correctness of the Source Compiler

**Theorem 3.** *For all ML values* `sp : Source.program`*, the following observational equivalence holds:*

`Target_int.process (Source_cmp2.process sp)` $\cong$ `SOME (Source_int.process sp).`

The proof of this theorem (i.e., of the correctness of Henderson's compiler for the SECD machine) is more involved than the proof of Theorem 1 and is beyond the scope of the present article. Therefore we omit it.

**Partial Correctness of the Target Compiler**

The situation is the same as in Section 2.5, i.e., not all values of type `Target.program` are well-formed programs, as indicated by the option type in Section 3.4 and Section 3.4. As in Section 2.5, we leave the issue of partial correctness aside, and instead we turn to proving that the target compiler is a left inverse of the source compiler.

**Left Inverseness**

In this section we prove that `Target_cmp.process` is a left inverse of `Source_cmp2.process` modulo $\alpha$-renaming. Our proof uses structural induction on source terms, and therefore we need to treat open terms together with their environment:

– the environment of a term, in the source compiler, is a list of identifiers;
– the environment of a term, in the target compiler, is a list of identifiers, all distinct.

We therefore relate the terms together with their environments as follows.

**Definition 1 (Left equivalence).** *For all ML values* `t : Source.term`*,* `xs : Source.ide list` *containing the identifiers free in* `t` *in reverse order of their declaration,* `t' : Source.term`*, and* `e : Source.ide list` *with the same length as* `xs` *and containing distinct identifiers, we say that* `t` *and* `t'` *are* left-equivalent with respect to `xs` and `e` *whenever the relation*

$$\langle \texttt{t}, \texttt{xs} \rangle \approx \langle \texttt{t'}, \texttt{e} \rangle$$

*is satisfied. This relation is defined inductively as follows:*

$$\frac{\texttt{n} \cong \texttt{n'}}{\langle \texttt{LIT n}, \texttt{xs} \rangle \approx \langle \texttt{LIT n'}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t1}, \texttt{xs} \rangle \approx \langle \texttt{t1'}, \texttt{e} \rangle \quad \langle \texttt{t2}, \texttt{xs} \rangle \approx \langle \texttt{t2'}, \texttt{e} \rangle}{\langle \texttt{PLUS (t1, t2)}, \texttt{xs} \rangle \approx \langle \texttt{PLUS (t1', t2')}, \texttt{e} \rangle}$$

$$\frac{\texttt{Index.fetch (e, Index.establish (x, xs))} \cong \texttt{x'}}{\langle \texttt{VAR x}, \texttt{xs} \rangle \approx \langle \texttt{VAR x'}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t}, \texttt{x :: xs} \rangle \approx \langle \texttt{t'}, \texttt{x' :: e} \rangle}{\langle \texttt{LAM (x, t)}, \texttt{xs} \rangle \approx \langle \texttt{LAM (x', t')}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t0}, \texttt{xs} \rangle \approx \langle \texttt{t0'}, \texttt{e} \rangle \quad \langle \texttt{t1}, \texttt{xs} \rangle \approx \langle \texttt{t1'}, \texttt{e} \rangle}{\langle \texttt{APP (t0, t1)}, \texttt{xs} \rangle \approx \langle \texttt{APP (t0', t1')}, \texttt{e} \rangle}$$

For closed terms that contain no $\lambda$-abstractions, left equivalence reduces to structural equality. For all closed terms, left equivalence implies $\alpha$-equivalence.

In Lemma 3 and Theorem 4, we use left equivalence to establish left inverseness.

**Lemma 3.** *For all ML values* `t : Source.term`*,* `xs : Source.ide list` *containing all the identifiers free in* `t`*,* `e : Source.ide list` *with the same length as* `xs` *and containing fresh (and all distinct) identifiers,* `s : Source.term list`*,* `c : Target.instr list`*,* `d : (Source.term list * Source.ide list * Target.instr list) list`*, and* `g : int`*, there exist two ML values* `t' : Source.term` *and* `g' : int` *such that the following conjunction holds:*

$$\langle \texttt{t}, \texttt{xs}\rangle \approx \langle \texttt{t'}, \texttt{e}\rangle \ \wedge\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp t xs c, d, g)} \\ \cong \texttt{w\_t\_cmp (t' :: s, e, c, d, g').}$$

*Proof.* The proof is by structural induction on the source syntax.

Base case: `LIT n`

For all ML values `xs : Source.ide list`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int`, we want to show that the following conjunction holds:

$$\langle \texttt{LIT n}, \texttt{xs}\rangle \approx \langle \texttt{LIT n}, \texttt{e}\rangle \ \wedge\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp (LIT n) xs c, d, g)} \\ \cong \texttt{w\_t\_cmp ((LIT n) :: s, e, c, d, g')}$$

for some ML value `g' : int`.

The left conjunct holds by definition of $\approx$. As for the right conjunct,

`w_t_cmp (s, e, w_s_cmp (LIT n) xs c, d, g)`
$\cong$ (unfolding `w_s_cmp`)
`w_t_cmp (s, e, Source_parameters_cogen2.lit n c, d, g)`
$\cong$ (unfolding `Source_parameters_cogen2.lit`)
`w_t_cmp (s, e, (fn is => (PUSH n) :: is) c, d, g)`
$\cong$ (function application)
`w_t_cmp (s, e, (PUSH n) :: c, d, g)`
$\cong$ (unfolding `w_t_cmp`)
`w_t_cmp ((LIT n) :: s, e, c, d, g)`

Induction case: `PLUS (t1, t2)`

For all ML values `xs : Source.ide list` containing all the identifiers free in `t1` and `t2`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int`, we want to show that the following conjunction holds:

$$\langle \texttt{PLUS (t1, t2)}, \texttt{xs}\rangle \approx \langle \texttt{PLUS (t1', t2')}, \texttt{e}\rangle \\ \wedge \\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp (PLUS (t1, t2)) xs c, d, g)} \\ \cong \texttt{w\_t\_cmp ((PLUS (t1', t2')) :: s, e, c, d, g'')}$$

for some ML value `g'' : int` and for all ML values `t1 : Source.term` and `t1' : Source.term` satisfying the induction hypothesis and for all ML values `t2 : Source.term` and `t2' : Source.term` satisfying the induction hypothesis. The left conjunct holds because of the induction hypotheses and by definition of $\approx$. As for the right conjunct,

```
w_t_cmp (s, e, w_s_cmp (PLUS (t1, t2)) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, w_s_cmp (Source_parameters_cogen.plus
                           (w_s_cmp t1 xs, w_s_cmp t2 xs) c, d, g)
≅ (unfolding Source_parameters_cogen2.plus)
w_t_cmp (s, e, (fn is =>
                  w_s_cmp t1 xs (w_s_cmp t2 xs (ADD :: is))) c, d, g)
≅ (function application)
w_t_cmp (s, e, w_s_cmp t1 xs (w_s_cmp t2 xs (ADD :: c)), d, g)
≅ (induction hypothesis on t1, for some ML value g' : int)
w_t_cmp (t1' :: s, e, w_s_cmp t2 xs (ADD :: c), d, g')
≅ (induction hypothesis on t2, for some ML value g'' : int)
w_t_cmp (t2' :: t1' :: s, e, ADD :: c, d, g'')
≅ (unfolding w_t_cmp)
w_t_cmp ((PLUS (t1', t2')) :: s, e, c, d, g'')
```

Base case: `VAR x`

For all ML values `xs : Source.ide list` containing `x`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int` we want to show that the following conjunction holds:

$$\langle \texttt{VAR x}, \texttt{xs}\rangle \approx \langle \texttt{VAR x'}, \texttt{e}\rangle \ \wedge\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp (VAR x) xs c, d, g)} \\ \cong \texttt{w\_t\_cmp ((VAR x') :: s, e, c, d, g')}$$

for some ML values `x' : Source.ide` and `g' : int`.

We reason equationally:

```
w_t_cmp (s, e, w_s_cmp (VAR x) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, Source_parameters_cogen.var
                 (Index.establish (x, xs)) c, d, g)
≅ (unfolding Source_parameters_cogen2.var)
w_t_cmp (s, e, (fn is =>
                 (ACCESS (Index.establish (x, xs))) :: is) c, d, g)
≅ (function application)
w_t_cmp (s, e, ((ACCESS (Index.establish (x, xs))) :: c), d, g)
≅ (unfolding w_t_cmp)
w_t_cmp ((VAR (Index.fetch (e, Index.establish (x, xs)))) :: s, e, c, d, g)
```

There are no unbound identifiers in source programs and by assumption all identifiers are accounted for in `xs`. Since `xs` and `e` have the same length, there exists an ML value `x' : Source.ide` in `e` satisfying

$$\texttt{Index.fetch (e, Index.establish (x, xs))} \cong \texttt{x'}$$

Given this `x'`, by definition of $\approx$,

$$\langle \texttt{VAR x}, \texttt{xs} \rangle \approx \langle \texttt{VAR x'}, \texttt{e} \rangle$$

holds and furthermore the following observational equality holds:

```
w_t_cmp ((VAR (Index.fetch (e, Index.establish (x, xs)))) :: s, e, c, d, g)
≅ w_t_cmp ((VAR x') :: s, e, c, d, g)
```

Induction case: `LAM (x, t)`
For all ML values `xs : Source.ide list` containing all the identifiers free in `t`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int` we want to show that the following conjunction holds:

```
                ⟨LAM (x, t), xs⟩ ≈ ⟨LAM (x', t'), e⟩
                                 ∧
        w_t_cmp (s, e, w_s_cmp (LAM (x, t)) xs c, d, g)
        ≅ w_t_cmp (LAM (x', t') :: s, e, c, d, g')
```

for some ML value `g' : int` and for all ML values `t : Source.term` and `t' : Source.term` satisfying the induction hypothesis.
We reason equationally:

```
w_t_cmp (s, e, w_s_cmp (LAM (x, t)) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, Source_parameters_cogen2.lam (w_s_cmp t (x :: xs)) c, d, g)
≅ (unfolding Source_parameters_cogen2.lam)
w_t_cmp (s, e, (fn is =>
               (CLOSE (w_s_cmp t (x :: xs) nil)) :: is) c, d, g)
≅ (function application)
w_t_cmp (s, e, (CLOSE t (w_s_cmp (x :: xs) nil)) :: c, d, g)
≅ (unfolding w_t_cmp)
w_t_cmp (nil, x' :: e, w_s_cmp t (x :: xs) nil, (s, e, c) :: d, g+1)
where x' = "x" ^ Int.toString g and is a fresh identifier.
≅ (induction hypothesis on t since xs' and e' have the same length,
   for some ML value t' : Source.term satisfying ⟨t, x :: xs⟩ ≈ ⟨t', x' :: e⟩
   for some ML value g' : int)
w_t_cmp (t' :: nil, x' :: e, nil, (s, e, c) :: d, g')
≅ (unfolding w_t_cmp)
w_t_cmp (LAM (x', t') :: s, e, c, d, g')
```

By induction hypothesis on `t`, $\langle$`t,x :: xs`$\rangle \approx \langle$`t',x' :: e`$\rangle$ holds, and therefore $\langle$`LAM (x, t),x :: xs`$\rangle \approx \langle$`LAM (x', t'),x' :: e`$\rangle$ also holds, by definition of $\approx$.

Induction case: `APP (t0, t1)`
This case is similar to the `PLUS` case above.

**Theorem 4.** *For each ML value* `sp : Source.program`*, there exists an ML value* `sp' : Source.program` *that is $\alpha$-equivalent to* `sp` *and that satisfies the following observational equivalence:*

`Target_cmp.process (Source_cmp2.process sp)` $\cong$ `SOME sp'`.

*Proof.* For all ML values `sp : Source.program` and `tp : Target.program`, the following observational equivalences hold:

`Source_cmp2.process sp`
$\cong$ (unfolding `Source_cmp2.process`)
`Source_cmp2.compute (w_s_cmp sp nil)`
$\cong$ (unfolding `Source_cmp2.compute`)
`w_s_cmp sp nil nil`

`Target_cmp.process tp`
$\cong$ (unfolding `Target_cmp.process`)
`w_t_cmp (nil, nil, tp, nil, 0)`

For all ML values `sp : Source.program`, we therefore have to prove the following observational equivalence:

`w_t_cmp (nil, nil, w_s_cmp sp nil nil, nil, 0)` $\cong$ `SOME sp'`

for a program `sp'` that is $\alpha$-equivalent to `sp`. This observational equivalence, however, follows from Lemma 3. Indeed, for all ML values `t : Source.term` that are closed `nil : Source.ide list`, `nil : Source.term list`, `nil : Target.instr list`, `nil : (Source.term list * Source.ide list * Target.instr list) list`, and `0 : int`, Lemma 3 reads as

$\langle$`t,nil`$\rangle \approx \langle$`t',nil`$\rangle \wedge$ `w_t_cmp (nil, nil, w_s_cmp t nil nil, nil, 0)`
$\cong$ `w_t_cmp (t' :: nil, nil, nil, nil, g')`

for some ML values `t' : Source.term` and `g' : int`. Therefore `t` and `t'` are left-equivalent. Since `t` is a closed term, `t'` is a closed term too, i.e., a program. Since they are left-equivalent, they are also $\alpha$-equivalent.

Finally,

`w_t_cmp (t' :: nil, nil, nil, nil, g')`
$\cong$ (unfolding of `w_t_cmp`)
`SOME t'`

and the result follows.

### 3.6 Summary

We have shown that the symmetric approach to compilation and decompilation scales to the $\lambda$-calculus and the SECD-machine language. We have not seen this approach to decompilation described elsewhere. For example, specific efforts have been dedicated to decompiling terms for abstract machines in the literature, independently of interpreting them and of compiling them [32, 34].

## 4 Related Work

This section situates our symmetric approach to compilation and decompilation with respect to compilation, decompilation, partial evaluation, and parsing.

### 4.1 Compilation and Decompilation

We consider in turn the construction, correctness, and derivation of compilers and decompilers.

**Construction**
Compilation and decompilation technologies have been around for over five decades. While many authors note that compilation is an inverse of decompilation [17, 33], in practice these technologies have evolved independently.

The area of compilation is well established and well mapped today, with a number of subdivisions—e.g., syntactic analysis, semantic analysis, and code generation. In contrast, the area of decompilation is not in the main stream and it is less well defined and less well-mapped. The general problem of decompilation is known to be unsolvable [17, 28, 36, 37], or requiring "human", i.e., "manual" intervention.

In general, writing a real decompiler is an engineering challenge which is documented comprehensively at ⟨`http://www.program-transformation.org/twiki/bin/view/Transform/DeCompilation`⟩.

In an imperative setting, the strategy is to establish control-flow and data-flow graphs to build high-level constructs [17]. For an example closer to our work here, Proebsting and Watterson decompile Java expressions by symbolically executing JVM instructions [57].

In a logical setting, decompiling by executing compiled programs is a standard technique that directly builds on a relational specification such as the one in Section 4.1 [12, 15].

**Correctness**
In their work on the lambda-sigma calculus [34], Hardin, Maranget, and Pagano consider a compiler to Cardelli's functional abstract machine and the corresponding compiler, and they prove an inverseness property. Similarly, in their work on strong reduction [32], Grégoire and Leroy also consider a compiler and

the corresponding decompiler. We are not aware of any other work addressing inverseness properties for a compiler and a decompiler. Also, we are aware of only few semantic approaches to decompilation, including Mycroft's type-based strategy and Katsumata and Ohori's proof-directed strategy [45, 54, 55].

Since McCarthy and Painter's first correctness proof of a compiler [50], correctness proofs for compilers typically use structural induction on the source syntax. Alternatively to defining two functions `Source_cmp.process` and `Target_cmp.process`, however, one can define a relation $\sim$ between source and target programs. For example, for the arithmetic expressions of Section 2, one can define the following relation between source expressions and lists of target instructions:

$$\frac{}{\texttt{Source.LIT n} \;\sim\; \texttt{[Target.PUSH n]}}$$

$$\frac{\texttt{e1} \sim \texttt{is1} \qquad \texttt{e2} \sim \texttt{is2}}{\texttt{Source.PLUS (e1, e2)} \;\sim\; \texttt{is1 @ is2 @ [Target.ADD]}}$$

This specification is the relational counterpart of the compiler of Section 2.3 and proving properties about it is done relationally.

In general, compilation and decompilation form yet another example of Galois connections in computer science, as outlined by Melton, Schmidt, and Strecker [52]. Indeed in general the image of each transformation is a sublanguage over which the composition of the two transformations acts as the identity, whereas it acts as a normalizer for programs in the annulus. The two examples presented here do not illustrate this normalization, but adding let expressions to the arithmetic expressions of Section 2 is enough to make it appear: target programs are decompiled into source programs where all the let expressions have been lifted out. More concretely, if the source language is

```
datatype exp = LIT of int
             | PLUS of exp * exp
             | LET of ide * exp * exp
             | VAR of ide
```

then the source sublanguage of normal forms is

```
datatype operation_nf = LIT_nf of int
                      | VAR_nf of ide
                      | PLUS_nf of operation_nf * operation_nf

datatype exp_nf = LET_nf of ide * operation_nf * exp_nf
                | BODY_nf of operation_nf
```

Another way to illustrate normalization by compilation and decompilation is to consider an optimizing compiler—e.g., one that includes constant propagation, constant folding, and common sub-expression elimination. In principle, the decompiler yields a correspondingly optimized source program, if one is expressible in the source language. The issue then is that of completeness. The phenomenon could be referred to as *normalization by staged evaluation*, in reference to normalization by evaluation [5, 9, 21, 22, 23, 24, 30].

**Derivation**
Much literature has been devoted to deriving a compiler from an interpreter, up to and including undergraduate textbooks [1,25]. We single out Morris's 700 follow-up paper for its observation that massaging a $\lambda$-interpreter can yield a compiler for the SECD machine [53] and Wand's article *Deriving target code as a representation of continuation semantics* for its compelling title that precisely characterizes the binding-time shift of going from evaluation to code generation [60].

In principle decompilation could be achieved by program inversion over a compiler [2, 29]. Abramov, Glück, and Klimov have recently reported ongoing efforts in this direction [3].

Our work is a study of a simultaneous derivation of a compiler and decompiler. The two are related by left-inverseness properties (Theorems 2 and 4). The relations between the compiler, the decompiler, and the two interpreters for the source and target languages are given by the the standard commuting diagram displayed in Section 1.

## 4.2 Partial Evaluation

In some sense, we are doing offline partial evaluation by hand. In particular, the factorizations into functors and structures of our language processors manifest a binding-time separation between the static (compile-time) and the dynamic (run-time) components of the language—what Lee refers to as macro- and micro-semantics [47] and as identified by Jones and Muchnick [40]. Given an unfactorized language processor, the binding-time analysis of an offline partial evaluator could achieve this binding-time division provided the language processor is well-written [38]. Specialization then corresponds to the instantiation of a functor with a code-generating structure.

Specializing interpreters is a popular application of partial evaluation, one that was discovered by Futamura in the early 1970s [26,27].

**The First Futamura Projection for Compiling**
Given an interpreter for a defined language written in a defining language and given a program written in the defined language, specializing the interpreter with respect to the program gives a residual program written in the defining language. In conjunction with a self-applicable partial evaluator, the first Futamura projection has been a major source of inspiration in the area of partial evaluation [39, 43].

In practice, specializing an interpreter with respect to a program yields a residual program that includes all the idiosyncrasies of the interpreter. For example, the residual program shown in Futamura's original article reveals that his interpreter represents environments as association lists [26, page 390]. Against this backdrop, the notion of Jones-optimality has been developed [48, 58].

**The First Futamura Projection for Decompiling**
In principle, the first Futamura projection directly applies for decompiling, given an interpreter for a target language written in a source language and given a program written in the target language. In practice, specializing this interpreter with respect to this program does give a residual program written in the source language but this residual program in general includes all the idiosyncrasies of the interpreter. In that sense, decompiling using the first Futamura projection is far from Jones-optimal.

In contrast, doing partial evaluation by hand as we do here gives us some extra flexibility regarding the target language in which to express residual programs, up to the point of left inverseness. For symmetry, it seems logical to refer to our methodology as a *Futamura embedding.*

### 4.3 Parsing

In some sense, and as agreed upon in the decompilation community, decompiling arithmetic expressions in reverse Polish form is akin to parsing [10]. More generally, a parser generator such as Yacc makes it possible to generate a compiler as well as an interpreter. A Yacc user parameterizes the core parsing engine by semantic actions, and these semantic actions can either carry out computations and construct intermediate results or they can build abstract-syntax trees. In that sense, we could use Yacc to decompile and to interpret arithmetic expressions in reverse Polish notation and also to decompile and to interpret programs for the SECD machine.

## 5 Conclusion

At the heart of turning an interpreter into a (front-end) compiler, there is a binding-time shift: Where the interpreter performs an evaluation, the compiler emits code representing this evaluation. In this article, we have shown how this binding-time shift can be used not only to construct a compiler from a source language to a target language but also to construct a compiler from a target language to a source language. We have treated two examples and we have proven that in each case the target compiler is a left inverse of the source compiler—i.e., formally, that the target compiler is a decompiler.

The source languages we have considered are a canonical language of expressions and its functional extension, the $\lambda$-calculus. Independently, we have also considered several other languages of expressions:

– a source language of boolean expressions, a target language of conditional expressions, and a compiler that implements short-cut boolean evaluation;
– a language of expressions with block structure and the language of a register-stack machine, as in Section 4.1; and
– another abstract machine for the $\lambda$-calculus.

In each case, we were able to apply the methodology of specifying each language processor with a functor implementing the corresponding fold function and instantiating this functor into an interpreter (with elementary evaluation functions) or a compiler (with elementary code-generation functions). To this end, we took advantage of the correspondence between source expressions and expressible values in functional languages. For imperative languages, however, it seems unavoidable to use some form of control-flow graph, as in traditional decompilation. At any rate, the methodology is not an end in itself; we see it as a systematic means to explore the binding-time shift between an interpreter and a (de)compiler.

*Acknowledgments:* We are grateful to Neil Jones for his generous inspiration, both direct and indirect, which includes coining the term 'binding-time shift'. This article has also benefited from comments by anonymous reviewers, by the editors of the Festschrift, and by Kenichi Asai, Robert Glück, Julia L. Lawall, Karoline Malmkjær, Lasse R. Nielsen, Henning Korsholm Rohde, and Ulrik P. Schultz. Thanks are also due to Jon Erickson, from Dr. Dobb's Journal, for a timely communication with the second author in March 2002.

This work was partly carried out while the third author was visiting BRICS.

## References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[2] Sergei M. Abramov and Robert Glück. The universal resolving algorithm: inverse computation in a functional language. In Roland Backhouse and José N. Oliveira, editors, *Mathematics of Program Construction. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 187–212. Springer-Verlag, 2000.

[3] Sergei M. Abramov, Robert Glück, and Yury Klimov. Faster answers and improved termination in inverse computation of non-flat languages. Technical report, Program Systems Institut, Russian Academy of Sciences, Pereslavl-Zalessky, March 2002 2002.

[4] Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. A symmetric approach to compilation and decompilation (extended version). Technical Report BRICS RS-02-37, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2002.

[5] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[6] Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: an introduction. In S. Doaitse Swierstra, Pedro Rangel Henriques, and José N. Oliveira, editors, *Advanced functional programming, Third International School*, number 1608 in Lecture Notes in Computer Science, pages 28–115, Braga, Portugal, September 1998. Springer-Verlag.

[7] Guntis Barzdins. Mixed computation and compiler basis. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 15–26. North-Holland, 1988.

[8] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[10] M. N. Bert and L. Petrone. Decompiling context-free languages from their Polish-like representations. *Calcolo*, XIX(1):35–57, March 1982.

[11] Anders Bondorf. Compiling laziness by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in Computing, pages 9–22, Glasgow, Scotland, 1990. Springer-Verlag.

[12] Jonathan Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, 1994.

[13] Peter J. Brown. Re-creation of source code from reverse Polish. *Software—Practice and Experience*, 2:275–278, 1972.

[14] Peter J. Brown. More on the re-creation of source code from reverse Polish. *Software—Practice and Experience*, 7(8):545–551, 1977.

[15] Kevin A. Buettner. Fast decompilation of compiled Prolog clauses. In Ehud Y. Shapiro, editor, *Proceedings of the third international conference on logic programming*, number 225 in Lecture Notes in Computer Science, pages 663–670, London, United Kingdom, July 1986. Springer-Verlag.

[16] Rod M. Burstall and Peter J. Landin. Programs and their proofs: An algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[17] Cristina Cifuentes. *Reverse Compilation Techniques.* PhD thesis, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, July 1994.

[18] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[19] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[20] Charles Consel and Siau-Cheng Khoo. Semantics-directed generation of a Prolog compiler. *Science of Computer Programming*, 21:263–291, 1993.

[21] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[22] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in

Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[23] Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

[24] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[25] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[26] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[27] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[28] R. Stockton Gaines. On the translation of machine language programs. *Communications of the ACM*, 8(12):736–741, 1965.

[29] Robert Glück and Andrei V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl, editor, *Cybernetics and Systems Research '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.

[30] Mayer Goldberg. Gödelization in the $\lambda$-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[31] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[32] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, Pittsburgh, Pennsylvania, September 2002. ACM Press. To appear.

[33] Maurice H. Halstead. *Machine Independent Computer Programming*. Spartan Books, Washington, D.C., 1962.

[34] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[35] Peter Henderson. *Functional Programming – Application and Implementation*. Prentice-Hall International, 1980.

[36] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[37] Barron C. Housel and Maurice H. Halstead. A methodology for machine language decompilation. In *Proceedings of the 27th ACM Annual Conference*, pages 254–260, 1974.

[38] Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 216–237, Dagstuhl, Germany, February 1996. Springer-Verlag.

[39] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[40] Neil D. Jones and Steven S. Muchnick. Some thoughts towards the design of an ideal language. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 77–94. ACM Press, January 1976.

[41] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *The Handbook of Logic in Computer Science*. North-Holland, 1992.

[42] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 70–93, Aarhus, Denmark, 1980. Springer-Verlag.

[43] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[44] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.

[45] Shin-ya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 352–366, Genova, Italy, April 2001. Springer-Verlag.

[46] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[47] Peter Lee. *Realistic Compiler Generation*. The MIT Press, 1989.

[48] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Proceedings of the First Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, number 1924 in Lecture Notes in Computer Science, pages 129–148, Montréal, Canada, September 2000. Springer-Verlag.

[49] William May. A simple decompiler – recreating source code without token resistance. *Dr. Dobb's Journal*, 50:50–52, June 1988.

[50] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings of the 1967 Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, Rhode Island, 1967.

[51] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, Nijmegen, The Netherlands, 1992.

[52] Austin Melton, David A. Schmidt, and George Strecker. Galois connections and computer science applications. In David H. Pitt et al., editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 299–312, Guildford, UK, September 1986. Springer-Verlag.

[53] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

[54] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 208–223, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[55] Alan Mycroft, Shin-ya Katsumata, and Atsushi Ohori. Comparing type-based and proof-directed decompilation. In Peter Aiken and Elizabeth Burd, editors, *Pro-*

*ceedings of the Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001. `http://reengineer.org/wcre2001/`.

[56] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[57] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In Steve Vinoski, editor, *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 185–197, Portland, Oregon, June 1997. The USENIX Association.

[58] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, number 2053 in Lecture Notes in Computer Science, pages 257–275, Aarhus, Denmark, May 2001. Springer-Verlag.

[59] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[60] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[61] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.

[62] Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

# The Abstraction and Instantiation of String-Matching Programs*

Torben Amtoft[1], Charles Consel[2], Olivier Danvy[3], and Karoline Malmkjær[4]

[1] Department of Computing and Information Sciences, Kansas State University
216 Nichols Hall, Manhattan, KS 66506-2302, USA
[2] INRIA/LaBRI/ENSEIRB
1 avenue du docteur Albert Schweitzer
Domaine universitaire – BP 99, F-33402 Talence Cedex, France
[3] BRICS[‡], Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
[4] Ericsson Telebit A/S
Skanderborgvej 232, DK-8260 Viby J., Denmark

**Abstract.** We consider a naive, quadratic string matcher testing whether a pattern occurs in a text; we equip it with a cache mediating its access to the text; and we abstract the traversal policy of the pattern, the cache, and the text. We then specialize this abstracted program with respect to a pattern, using the off-the-shelf partial evaluator Similix.
Instantiating the abstracted program with a left-to-right traversal policy yields the linear-time behavior of Knuth, Morris and Pratt's string matcher. Instantiating it with a right-to-left policy yields the linear-time behavior of Boyer and Moore's string matcher.

*To Neil Jones, for his 60th birthday.*

## 1 Background and Introduction

To build the first self-applicable partial evaluator [41, 42], Neil Jones, Peter Sestoft and Harald Søndergaard simplified its domain of discourse to an extreme: polyvariant specialization of symbolic first-order recursive equations. Source programs were expressed as recursive equations, data were represented as S-expressions, and the polyvariant-specialization strategy amounted to an interprocedural version of Kildall's constant-propagation algorithm [4, 17]: for each of its calls, a recursive equation is specialized with respect to its known arguments and the resulting specialized equation is indexed with these known arguments and cached in a worklist. Specialization terminates once the worklist is complete, and the specialized equations in the worklist form the specialized program.

* Corresponding authors: Torben Amtoft (`tamtoft@cis.ksu.edu`) and Olivier Danvy (`danvy@brics.dk`). Extended version available as the BRICS technical report RS-01-12.

‡ Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

Since then, this basic framework has been subjected to many developments. For example, not all calls need to be specialized and give rise to a specialized recursive equation: many of them can merely be unfolded. Data need not be only symbolic either—numbers, characters, strings, pairs, higher-order functions, etc. can also be handled, and so can side effects. Finally, source programs need not be mere recursive equations—e.g., block-structured programs are amenable to partial evaluation as well.

Nevertheless, the basic strategy of polyvariant program-point specialization remains the same: generating mutually recursive residual equations, each of them corresponding to a source program point indexed by known values. In 1987, at the workshop on Partial Evaluation and Mixed Computation [11, 29], this very simple specialization strategy was challenged. Yoshihiko Futamura said that he had needed to design a stronger form of partial evaluation—Generalized Partial Computation [32]—to be able to specialize a naive quadratic-time string matcher into Knuth, Morris, and Pratt's (KMP) linear-time string matcher [45]. He therefore asked the DIKU group whether polyvariant specialization was able to obtain the KMP.

## 1.1 Obtaining the KMP

The naive matching algorithm tests whether the pattern is the prefix of one of the suffixes of the text. Every time the pattern does not match a prefix of a suffix of the text, the pattern is shifted by one position to try the next suffix. In contrast, the KMP algorithm proceeds in two phases:

1. given the pattern, it constructs a 'failure table' in time linear in the size of the pattern; and
2. it traverses the text linearly, using the failure table to determine how much the pattern should be shifted in case of mismatch.

The first thing one could try is to specialize the traversal of the text with respect to a failure table. Unsurprisingly, the result is a linear-time residual program. In their article [45, page 330], Knuth, Morris and Pratt display a similar program where the failure table has been "compiled" into the control flow.

More in the spirit of Futamura's challenge, one can consider the naive specification, i.e., a (known) pattern string occurs in an (unknown) text if it is the prefix of one of its suffixes. On the left of each mismatch, the pattern and the text are equal:

```
              ---->
    Pattern  +---------o-----------+
              |||||||||* mismatch
       Text  +---------o-----------------------------------------+
              ---->
```

Yet, the naive specification shifts the pattern one place to the right and re-scans it together with the text up to the mismatch point and beyond:

```
           ---->    ---->
Pattern   +---------o-----------+
           ||||||||||||||
   Text  +---------o-----------------------------------------+
           ---->    ---->
```

Instead, one can match two instances of the pattern (the pattern and a prefix of the pattern equal to the prefix of the text matched so far, with one character chopped off) up to the point of mismatch and then resume scanning the pattern and the text:

```
           ---->
Pattern  +---------o
           ||||||||---->
Pattern   +---------o-----------+
           ---->   ||||||
   Text            o-----------------------------------------+
                   ---->
```

This staged program is equivalent to the original program. In particular, its complexity is the same. But if one specializes it with respect to a pattern, matching the pattern against itself is carried out at partial-evaluation time. Furthermore, since the index on the text only increases, specializing this staged program yields a string matcher that does not back up on the text and thus operates in time linear in the text. Therefore part of the challenge is met: polyvariant specialization can turn a (staged) quadratic string matcher into a linear one.

But how does this linear string matcher compare to the KMP?

Short of a formal proof, one can compare, for given patterns, (1) the corresponding residual string matchers and (2) the results of specializing the second phase of the KMP with respect to the corresponding failure tables produced by the first phase. It turns out that the resulting string matchers are indeed isomorphic, suggesting that mere polyvariant specialization can obtain the KMP if the naive string matcher is staged as pictured above [22].

This experiment clarifies that linearity requires one to keep a static view of the dynamic prefix of the text during specialization. This static view can be kept explicitly in the source program, as above. Alternatively, a partial evaluator could implicitly memoize the outcome of every dynamic test. (For example, given a dynamic variable `x`, a conditional expression `(if (odd? x) e1 e2)`, and some knowledge about `odd?`, such a partial evaluator would know that `x` is odd when processing `e1` and that `x` is even when processing `e2`.) Futamura's Generalized Partial Computation is such a partial evaluator. Therefore, it automatically keeps a static view of the dynamic prefix of the text during specialization and thus obtains the KMP out of a naive, unstaged string matcher. So the key to linearity is precisely the static view of the dynamic prefix of the text during specialization.

To summarize, we need a static view of the dynamic prefix of the text during specialization. Whether this view is managed implicitly in the partial evaluator (as in Generalized Partial Computation) or explicitly in the source program (as in basic polyvariant specialization) is a matter of convenience. If the management is implicit, the program is simpler but the partial evaluator is more complex. If the management is explicit, the program is more complicated but the partial evaluator is simpler.

That said, there is more to the KMP than linearity over the text—there is also linearity over the pattern. Indeed, the KMP algorithm operates on a failure table that is constructed in time linear in the size of the pattern, whereas in general, a partial evaluator does not operate in linear time.

## 1.2 This Work

We extend the construction of Section 1.1 to obtain linear string matchers in the style of Knuth, Morris, and Pratt as well as string matchers in the style of Boyer and Moore from the same quadratic string matcher.

To this end, we instrument the naive string matcher with a *cache* that keeps a trace of what is known about the text, similarly to a memory cache on a hardware processor. Any access to the text is mediated by an access to the cache. If the information is already present in the cache, the text is not accessed. If the information is not present in the cache, the text is accessed and the cache is updated.

The cache has the same length as the pattern and it gives us a static view of the text. We make it keep track both of what is known to occur in the text and of what is known not to occur in the text. Each entry of the cache thus contains either some *positive* information, namely the corresponding character in the text, or some *negative* information, namely a list of (known) characters that do not match the corresponding (unknown) character in the text. The positive information makes it possible to test characters statically, i.e., without accessing the text. The negative information makes it possible to detect statically when a test would fail. Initially, the cache contains no information (i.e., each entry contains an empty list of negative information).

By construction, the cache-based string matcher consults an entry in the text either not at all or repeatedly until the entry is recognized. An entry in the text can be left unconsulted because an adjacent character does not match the pattern. Once an entry is recognized, it is never consulted again because the corresponding character is stored in the cache. The entry can be consulted repeatedly, either until it is recognized or until the pattern has shifted. Each unrecognized character is stored in the cache and thus the number of times an entry is accessed is at most the number of different characters in the pattern. Therefore, each entry in the text is consulted a bounded number of times. The specialized versions of the cache-based string matcher inherit this property, and therefore their matching time is linear with respect to the length of the text.

On the other hand, the size of the specialized versions can be large: polyvariant specialization yields residual equations corresponding to source program points indexed by known values—here, specifically, the cache. In order to reduce the size of the residual programs, we allow ourselves to *prune* the cache in the source string matcher. For example, two extreme choices are to remove all information and to remove no information from the cache, but variants are possible.

Finally, since the KMP compares the pattern and the text from left to right whereas the Boyer and Moore compares the pattern and the text from right to left, we also parameterize the string matcher with a traversal policy. For example, two obvious choices are left to right and right to left, but variants are possible.

Our results are that traversing the pattern from left to right yields the KMP behavior, with variants, and that traversing the pattern from right to left yields the Boyer and Moore behavior [15, 16], also with variants. (The variants are determined by how we traverse the pattern and the text and how we prune the cache.) These results are remarkable (1) because they show that it is possible to obtain both the KMP linear behavior and the Boyer and Moore linear behavior from the *same* source program, and (2) because even though it is equipped with a cache, this source program is a naive quadratic string-matching program.

The first author is aware of these results since the early 1990s [6] and the three other authors since the late 1980s [24]. Since then, these results have been reproduced and extended by Queinnec and Geffroy [52], but otherwise they have only been mentioned informally [23, Section 6.1].

## 1.3 A Note on Program Derivation and Reverse Engineering

We would like to point out that we are not using partial evaluation to reverse-engineer the KMP and the Boyer and Moore string matchers. What we are attempting to do here is to exploit the simple observation that, in the naive quadratic string matcher, and as depicted in Section 1.1, rematching can be optimized away by partial evaluation.

In 1988, we found that the resulting specialized programs behave like the KMP, for a left-to-right traversal. We then conjectured that for a right-to-left traversal, the resulting specialized programs should behave like Boyer and Moore—which they do. In fact, for any traversal, partial evaluation yields a linear-time residual program (which may be sizeable), at least as long as no pruning is done. We have observed that the method scales up to patterns with wild cards and variables as well as to pattern matching in trees à la Hoffman and O'Donnell [38]. For two other examples, the method has led Queinnec and Geffroy to derive Aho and Corasick's as well as Commentz-Walter's string matchers [52].

## 1.4 Overview

The rest of this article is organized as follows. Section 2 presents the core program and its parameters. Section 3 specifies the action of partial evaluation on the core program. Section 4 describes its KMP instances and Section 5 describes its Boyer and Moore instances. Related work is reviewed in Section 6 and Section 7 concludes. A correctness proof of the core program can be found in the extended version of this article [7].

Throughout, we use the programming language Scheme [43], or, more precisely, Petite Chez Scheme (`www.scheme.com`) and the partial evaluator Similix [13, 14].

## 2 The Abstracted String Matcher

We first specify the operations on the cache (Section 2.1). Then we describe the cache-based string matcher (Section 2.2).

### 2.1 The Cache and Its Operations

The cache has the same size as the pattern. It holds a picture of what is currently known about the text: either we know a character, or we know nothing about a character, or we know that a character is not equal to some given characters. Initially, we know nothing about any entry of the text. In the course of the algorithm, we may get to know some characters that do not match an entry, and eventually we may know this entry. We then disregard the characters that do not match the entry, and we only consider the character that matches the entry. Accordingly, each entry of the cache contains either some positive information (one character) or some negative information (a list of characters, possibly empty). We test this information with the following predicates.

```
(define pos? char?)                    (define (neg? e)
                                         (or (null? e) (pair? e)))
```

These two predicates are mutually exclusive.

We implement the cache as a list and we operate on it as follows.

– Given a natural number specifying the length of the pattern, `cache-init` constructs an initial cache containing empty lists of characters.

```
(define (cache-init n)
  (if (= n 0)
      '()
      (cons '() (cache-init (- n 1)))))
```

– In the course of string matching, the pattern slides to the right of the text by one character. Paralleling this shift, `cache-shift` maps a cache to the corresponding shifted cache where the right-most entry is negative and empty.

```
(define (cache-shift c)
  (append (cdr c) (list '())))
```

– Given an index, the predicates `cache-ref-pos?` and `cache-ref-neg?` test whether a cache entry is positive or negative.

```
(define (cache-ref-pos? c i)          (define (cache-ref-neg? c i)
  (pos? (list-ref c i)))                (neg? (list-ref c i)))
```

– Positive information is fetched by `cache-ref-pos` and negative information by `cache-ref-neg`.

```
(define cache-ref-pos list-ref)    (define cache-ref-neg list-ref)
```

– Given a piece of positive information at an index, `cache-extend-pos` extends the cache with this positive information at that index. The result is a new cache where the positive information supersedes any negative information at that index.

```
(define (cache-extend-pos c i pos)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons pos (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1)))))])
    (walk c i)))
```

– Given more negative information at an index, `cache-extend-neg` extends the cache with this negative information at that index. The result is a new cache.

```
(define (cache-extend-neg c i neg)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons (cons neg (car c)) (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1)))))])
    (walk c i)))
```

– Given the cache, `schedule-pc` and `schedule-pt` respectively yield the series of indices (represented as a list) through which to traverse the pattern and the cache, and through which to traverse the pattern and the text, respectively. The formal requirements on `schedule-pc` and `schedule-pt` are stated elsewhere [7, Appendix A].

1. The following definition of `schedule-pc` specifies a left-to-right traversal of the pattern and the cache:

```
(define (schedule-pc c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (cons i (walk (+ i 1) (cdr c)))]
                [else
                 (if (null? (car c))
                     (walk (+ i 1) (cdr c))
                     (cons i (walk (+ i 1) (cdr c))))]))])
    (walk 0 c)))
```

For example, applying `schedule-pc` to an empty cache yields the empty list. This list indicates that there is no need to compare the pattern and the cache.
For another example, applying `schedule-pc` to a cache containing two pieces of positive information, and then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(0 1 2)`. This list indicates that the pattern and the cache should be successively compared at indices 0, 1, and 2, and that the rest of the cache should be ignored.

2. The following definition of `schedule-pt` specifies a left-to-right traversal of the pattern and of the text:

```
(define (schedule-pt c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (walk (+ i 1) (cdr c))]
                [else
                 (cons i (walk (+ i 1) (cdr c)))]))])
    (walk 0 c)))
```

For example, applying `schedule-pt` to an empty cache of size 3 (as could be obtained by evaluating `(cache-init 3)`) yields the list `(0 1 2)`. This list indicates that the pattern and the text should be successively compared at indices 0, 1, and 2.
For another example, applying `schedule-pt` to a cache containing two pieces of positive information, then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(2 3 4 5)`. This list indicates that the pattern and the text already agree at indices 0 and 1, and should be successively compared at indices 2, 3, 4, and 5. (The negative information at index 2 in the cache is of no use here.)

– Finally, we give ourselves the possibility of pruning the cache with `cache-prune`. Pruning the cache amounts to shortening lists of negative information and resetting positive information to the empty list of negative information. For example, the identity function prunes nothing at all:

```
(define (cache-prune c)
  c)
```

## 2.2 The Core Program

The core program, displayed in Figure 1, is a cache-based version of a naive quadratic program checking whether the pattern occurs in the text, i.e., if the pattern is a prefix of one of the successive suffixes of the text, from left to right.

```
(define (match p t)
  (let ([lp (string-length p)])         ;;; lp is the length of p
    (if (= lp 0)
        0
        (let ([lt (string-length t)])   ;;; lt is the length of t
          (if (< lt lp)
              -1
              (match-pt p lp (cache-init lp) t 0 lt))))))

(define (match-pc p lp c t bt lt)
  ;;; matches p[0..lp-1] and c[0..lp-1]
  (letrec ([loop-pc
            (lambda (z c is)
              (if (null? is)
                  (let ([bt (+ bt z)] [lt (- lt z)])
                    (if (< lt lp)
                        -1
                        (match-pt p lp c t bt lt)))
                  (let ([i (car is)])
                    (if (if (cache-ref-pos? c i)
                            (equal? (string-ref p i)
                                    (cache-ref-pos c i))
                            (not (member (string-ref p i)
                                         (cache-ref-neg c i))))
                        (loop-pc z c (cdr is))
                        (let ([c (cache-shift c)])
                          (loop-pc (+ z 1) c (schedule-pc c)))))))])
    (loop-pc 1 c (schedule-pc c))))

(define (match-pt p lp c t bt lt)
  ;;; matches p[0..lp-1] and t[bt..bt+lp-1]
  (letrec ([loop-pt                     ;;; bt is the base index of t
            (lambda (c is)
              (if (null? is)
                  bt
                  (let* ([i (car is)] [x (string-ref p i)])
                    (if (equal? (string-ref t (+ bt i)) x)
                        (loop-pt (cache-prune
                                   (cache-extend-pos c i x))
                                 (cdr is))
                        (match-pc p lp
                                  (cache-shift
                                    (cache-extend-neg c i x))
                                  t bt lt)))))])
    (loop-pt c (schedule-pt c))))
```

**Fig. 1.** The core quadratic string-matching program

The program is composed of two mutually recursive loops: a static one checking whether the pattern matches the cache and a dynamic one checking whether the pattern matches the rest of the text.

*The main function,* `match`*:* Initially, `match` is given a pattern `p` and a text `t`. It computes their length (`lp` and `lt`, respectively), and after checking that there is room in the text for the pattern, it initiates the dynamic loop with an empty cache. The result is either $-1$ if the pattern does not occur in the text, or the index of the left-most occurrence of the pattern in the text.

*The static loop,* `match-pc`*:* The pattern is iteratively matched against the cache using `loop-pc`. For each mismatch, the cache is shifted and `loop-pc` is resumed. Eventually, the pattern agrees with (i.e., matches) the cache—if only because after repeated iterations of `loop-pc` and shifts of the cache, the cache has become empty. Then, if there is still space in the text for the pattern, the dynamic loop is resumed.

In more detail, `match-pc` is passed the pattern `p`, its length `lp`, the cache `c`, and also the text `t`, its base index `bt`, and the length `lt` of the remaining text to match in the dynamic loop. The static loop checks iteratively (with `loop-pc`) whether the pattern and the cache agree. The traversal takes place in the order specified by `schedule-pc` (e.g., left to right or right to left). For each index, `loop-pc` tests whether the information in the cache is positive or negative. In both cases, if the corresponding character in the pattern agrees (i.e., either matches or does not mismatch), `loop-pc` iterates with the next index. Otherwise the cache is shifted and `loop-pc` iterates. In addition, `loop-pc` records in `z` how many times the cache has been shifted (initially 1 since `match-pc` is only invoked from the dynamic loop). Eventually, the pattern and the cache agree. The new base and the new length of the text are adjusted with `z`, and if there is enough room in the text for the pattern to occur, the dynamic loop takes over.

When the dynamic loop takes over, the pattern and the cache completely agree, i.e., for all indices $i$:

- either the cache information is positive at index $i$ and it contains the same character as the pattern at index $i$;
- or the cache information is negative at index $i$ and the character at index $i$ in the pattern does not occur in the list of characters contained in this negative information.

*The dynamic loop,* `match-pt`*:* The pattern is iteratively matched against the parts of the text that are not already in the cache, using `loop-pt`. At each iteration, the cache is updated. If a character matches, the cache is updated with the corresponding positive information before the next iteration of `loop-pt`. If a character does not match, the cache is updated with the corresponding negative information and the static loop (i.e., `match-pc`) is resumed with a shifted cache.

In more detail, `match-pt` is passed the pattern `p`, its length `lp`, the cache `c`, and the text `t`, its base index `bt`, and the length `lt` of the remaining text to match. The traversal takes place in the order specified by `schedule-pt` (e.g., left to right or right to left). For each index, `loop-pt` tests the corresponding character in the text against the corresponding character in the pattern. If the two characters are equal, then `loop-pt` iterates with an updated and pruned cache. Otherwise, the static loop takes over with an updated and shifted cache. If `loop-pt` runs out of indices, pattern matching succeeds and the result is the base index in the text.

*The cache:* The cache is only updated (positively or negatively) during the dynamic loop. Furthermore, it is only updated with a character *from the pattern* and never with one from the text. Considering that the pattern is known (i.e., static) and that the text is unknown (i.e., dynamic), constructing the cache with characters from the pattern ensures that it remains known at partial-evaluation time. This selective update of the cache is the key for successful partial evaluation.

We allow `schedule-pt` to return indices of characters that are already in the cache and to return a list of indices with repetitions. These options do not invalidate the correctness proof of the core program, but they make it possible to reduce the size of residual programs, at the expense of redundant tests. This information-theoretic weakening is common in published studies of the Boyer-Moore string matcher [8, 39, 51]: the challenge then is to show that the weakened string matcher still operates in linear time [53].

In the current version of Figure 1, we only prune the cache at one point: after a call to `cache-extend-pos`. This lets us obtain a number of variants of string matchers in the style of Boyer and Moore. But as noted in Section 5.2, at least one remains out of reach.

## 3 Partial Evaluation

Specializing the core string matcher with respect to a pattern takes place in the traditional three steps: preprocessing (binding-time analysis), processing (specialization), and postprocessing (unfolding).

### 3.1 Binding-Time Analysis

Initially, `p` is static and `t` is dynamic.

In `match`, `lp` is static since `p` is static. The first let expression and the first conditional expression are thus static. Conversely, `lt` is dynamic since `t` is dynamic. The second let expression and the second conditional expression are thus dynamic. After binding-time analysis, `match` is annotated as follows.

```
(define (match p^s t^d)
  (let^s ([lp^s (string-length p^s)])
    (if^s (= lp^s 0^s)
          0^d
          (let^d ([lt^d (string-length t^d)])
            (if^d (< lt^d lp^s)
                  -1^d
                  (match-pt p^s lp^s (cache-init lp^s) t^d 0^d lt^d))))))
```

In `match-pc` and in `match-pt`, `p`, `lp` and `c` are static, and `t`, `bt`, and `lt` are dynamic. (Therefore, `0` is generalized to be dynamic in the initial call to `match-pt`. Similarly, since `match` returns a dynamic result, `0` and `-1` are also generalized to be dynamic.)

In `loop-pc`, `c` and `is` are static and—due to the automatic poor man's generalization of Similix [40]—`z` is dynamic. Except for the conditional expression testing `(< lt lp)` and its enclosing let expression, which are dynamic, all syntactic constructs are static.

After binding-time analysis, `match-pc` and `loop-pc` are annotated as follows.

```
(define (match-pc p^s lp^s c^s t^d bt^d lt^d)
  (letrec ([loop-pc
            (lambda (z^d c^s is^s)
              (if^s (null? is^s)
                   (let^d ([bt^d (+ bt^d z^d)] [lt^d (- lt^d z^d)])
                     (if^d (< lt^d lp^s)
                          -1^d
                          (match-pt p^s lp^s c^s t^d bt^d lt^d)))
                   (let^s ([i^s (car is^s)])
                     (if^s (if^s (cache-ref-pos? c^s i^s)
                                (equal? (string-ref p^s i^s)
                                        (cache-ref-pos c^s i^s))
                                (not (member (string-ref p^s i^s)
                                             (cache-ref-neg c^s i^s))))
                          (loop-pc z^d c^s (cdr is^s))
                          (let^s ([c^s (cache-shift c^s)])
                            (loop-pc (+ z^d 1^d) c^s (schedule-pc c^s)))))))])
    (loop-pc 1^d c^s (schedule-pc c^s))))
```

In `loop-pt`, `c` and `is` are static. Except for the conditional expression performing an equality test, all syntactic constructs are static.

After binding-time analysis, `match-pt` and `loop-pt` are annotated as follows.

```
(define (match-pt p^s lp^s c^s t^d bt^d lt^d)
  (letrec ([loop-pt
            (lambda (c^s is^s)
              (if^s (null? is^s)
                   bt^d
                   (let*^s ([i^s (car is^s)] [x^s (string-ref p^s i^s)])
                     (if^d (equal? (string-ref t^d (+ bt^d i^s)) x^s)
                          (loop-pt (cache-prune
                                     (cache-extend-pos c^s i^s x^s))
                                   (cdr is^s))
                          (match-pc p^s lp^s
                                    (cache-shift
                                      (cache-extend-neg c^s i^s x^s))
                                    t^d bt^d lt^d)))))])
    (loop-pt c^s (schedule-pt c^s))))
```

### 3.2 Polyvariant Specialization

In Similix [14], the default specialization strategy is to unfold all function calls and to treat every conditional expression with a dynamic test as a specialization point. Here, since there is exactly one specialization point in `loop-pc` as well as in `loop-pt`, without loss of generality, we refer to each instance of a specialization point as *an instance of* `loop-pc` and *an instance of* `loop-pt`, respectively.

Each instance of `loop-pc` is given a base index in the text, the remaining length of the text, and a natural number. The body of this instance is a clone of the dynamic let expression and of the dynamic conditional expression in the original `loop-pc`: it increments `bt` and decrements `lt` with the natural number, checks whether the remaining text is large enough, and calls an instance of `loop-pt`, as specified by the following template (where `<lp>` is a natural number denoting the length of the pattern).

```
(define (loop-pc-... t bt lt z)
  (let ([bt (+ bt z)] [lt (- lt z)])
    (if (< lt <lp>)
        -1
        (loop-pt-... t bt lt))))
```

Each instance of `loop-pt` is given a base index in the text and the remaining length of the text (their sum is always equal to the length of the text). It tests whether a character in the text is equal to a fixed character. If so, it either returns an index or branches to another instance of `loop-pt`. If not, it branches to an instance of `loop-pc` with a fixed shift. (Below, `<i>` stands for a non-negative integer, `<n>` stands for a natural number, and `<c>` stands for a character.)

```
(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      bt
      (loop-pc-... t bt lt <n>)))

(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      (loop-pt-... t bt lt)
      (loop-pc-... t bt lt <n>)))
```

Therefore, a residual program consists of mutually recursive specialized versions of the two specialization points, and of a main function computing the length of the text, checking whether it is large enough, and calling a specialized version of `loop-pt`. The body of each auxiliary function is constructed out of instances of the dynamic parts of the programs, i.e., it fits one of the templates above [25, 48].

We have written the holes in the templates between brackets. The name of each residual function results from concatenating `loop-pc-` and `loop-pt-` to a fresh index. Each residual function is closed, i.e., it has no free variables.

Overall, the shape of a residual program is as follows.

```
(define (match-0 t)
  (let ([lt (string-length t)])
    (if (< lt ...)
        -1
        (loop-pt-1 t 0 lt))))

(define (loop-pc-1 t bt lt z) ...)

(define (loop-pc-2 t bt lt z) ...)

...

(define (loop-pt-1 t bt lt) ...)

(define (loop-pt-2 t bt lt) ...)

...
```

This general shape should make it clear how residual programs relate to (and how their control flow could be "decompiled" into) a KMP-like failure table, as could be obtained by data specialization [9, 20, 44, 47].

### 3.3 Post-unfolding

To make the residual programs more readable, Similix post-unfolds residual functions that are called only once. It also defines the remaining functions locally to the main residual function. (By the same token, it could lambda-drop the variable `t`, as we do in the residual programs displayed in Sections 4, 5, and 7, for readability [27].)

So all in all, a specialized program is defined as a main function (an instance of `match`) with many locally defined and mutually recursive auxiliary functions (instances of `loop-pc` and `loop-pt`), each of which is called more than once.

## 4 The KMP Instances

In the KMP style of string matching, the pattern and the corresponding prefix of the text are traversed from left to right. Therefore, we define `schedule-pt` so that `match-pt` proceeds from left to right. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned. Furthermore, all residual programs are independent of `schedule-pc` since it is applied at partial-evaluation time. Since Consel and Danvy's work [22], this traversal has been consistently observed to coincide with the traversal of Knuth, Morris and Pratt's string matcher. (We are not aware of any formal proof of this coincidence, but we expect that we will be able to show that the two matchers operate in lock step.)

```
(define (match-aaa t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 0)) #\a)
        (if (equal? (string-ref t (+ bt 1)) #\a)
            (if (equal? (string-ref t (+ bt 2)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (loop-pc-1 bt lt 2))
        (loop-pc-1 bt lt 1)))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))
```

– For all `t`, `(match-aaa t)` equals `(match "aaa" t)`.
– For all `z`, `(loop-pc-1 bt lt z)` equals `(loop-pc z '(() () ()) '())` evaluated in the scope of `bt` and `lt`.
– `(loop-pt-1 bt lt)` equals `(loop-pt '(() () ()) '(0 1 2))` evaluated in the scope of `bt` and `lt`.

**Fig. 2.** Specialized version of Fig. 1 wrt. `"aaa"` à la Knuth, Morris and Pratt

This 'KMP behavior' has been already described in the literature (see Section 6), so we keep this section brief. In this instantiation, the cache is naturally composed of three parts: a left part with positive information, a right part with empty negative information, and between the left part and the right part, at most one entry with non-empty negative information. This entry may or may not be pruned away, which affects both the size of the residual code and its runtime, as analyzed by Grobauer and Lawall [36].

Figure 2 displays a specialized version of the core program in Figure 1 with respect to the pattern string `"aaa"`. This specialized version is renamed for readability. We instantiated `cache-prune` to the identity function. The code for `loop-pt-1` reveals that a left-to-right strategy is indeed employed, in that the text is addressed with offsets 0, 1, and 2.

## 5 The Boyer and Moore Instances

### 5.1 The Boyer and Moore Behavior

In the Boyer and Moore style of string matching, the pattern and the corresponding prefix of the text are traversed from right to left. Therefore, we define `schedule-pt` so that `match-pt` proceeds from right to left. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned and again independently of `schedule-pc`, which is executed at partial-evaluation time. We have consistently observed that this traversal coincides with the traversal of string matchers à la Boyer and Moore,[1] provided that `schedule-pt` processes the non-empty negative entries in the cache before the empty ones. Then there will always be at most one entry with non-empty negative information.

Except for the first author's PhD dissertation [6], this "Boyer and Moore behavior" has not been described in the literature, so we describe it in some detail here. Initially, in Figure 1, `loop-pt` iteratively builds a suffix of positive information. This suffix is located on the right of the cache, and grows from right to left. In case of mismatch, `loop-pt` punctuates the suffix with one (non-empty) negative entry, shifts the cache to the right, and resumes `match-pc`. The shift transforms the suffix into a segment, which drifts to the left of the cache while `loop-pt` and `loop-pc` continue to interact. Subsequently, if a character match is successful, the negative entry in the segment becomes positive, and the matcher starts building up a new suffix. The cache is thus composed of zero or more segments of positive information, the right-most of which may be punctuated on the left by one negative entry. Each segment is the result of an earlier unsuccessful call to `loop-pt`. The right-most segment (resp. the suffix), is the witness of the latest (resp. the current) call to `loop-pt`. In the course of string matching, adjacent segments can merge into one.

## 5.2 Pruning

**No Pruning:** Never pruning the cache appears to yield Knuth's optimal Boyer and Moore string matcher [45, page 346].

**Pruning Once:** Originally [15], Boyer and Moore maintained two tables. The first table indexes each character in the alphabet with the right-most occurrence (if any) of this character in $p$. The second table indexes each position in $p$ with the rightmost occurrence (if any) of the corresponding suffix, preceded by a different character, elsewhere in $p$.

We have not obtained Boyer and Moore's original string matcher because it exploits the two tables in a rather unsystematic way. Nevertheless, the following instantiation appears to yield a simplified version of Boyer and Moore's original string matcher, where only the first table is used (and hence where linearity does not hold):

– `cache-prune` empties the cache;
– after listing the entries with non-empty negative information, `schedule-pt` lists *all* negative entries, even those (zero or one) that are already listed.

[1] Again, we expect to be able to prove this coincidence by showing that the two traversals proceed in lock step.

(This redundancy is needed in order to keep the number of residual functions small, at the expense of redundant dynamic (residual) tests.)

Partsch and Stomp also observed that Boyer and Moore's original string matcher uses the two tables unsystematically [51]. A more systematic take led them to formally derive the alternative string matcher hinted at by Boyer and Moore in their original article [15, page 771]. It appears that this string matcher can be obtained as follows:

– `cache-prune` removes all information except for the right-most suffix of positive information (i.e., the maximal set of the form $\{j, \ldots, lp-1\}$ with all the corresponding entries being positive);
– after having listed the non-empty negative information, `schedule-pt` lists *all* indices (including the positive ones).

**More Pruning:** The pruning strategy of Figure 1 is actually sub-optimal. A better one is to prune the cache before reaching each specialization point. Doing so makes it possible to obtain what looks like Horspool's variant of Boyer and Moore's string matcher [39].

## 5.3 An Example

Figure 3 displays a specialized version of the program in Figure 1 with respect to the pattern string `"abb"`. This specialized version is renamed for readability. We used instantiations yielding Partsch and Stomp's variant. The code for `loop-pt-1` reveals that a right-to-left strategy is indeed employed, in that the text is addressed with offsets 2, 1, and 0. (NB. Similix has pretty-printed successive `if` expressions into one `cond` expression.)

## 6 Related Work

In his MS thesis [55, Sec. 8.2], Sørensen has observed that once the pattern is fixed, the naive string matcher is de facto linear—just with a factor proportional to the length of this pattern, *lp*. The issue of obtaining the KMP behavior is therefore to produce a specialized program that runs in linear time with a factor independent of *lp*. Since Consel and Danvy's original solution [22], obtaining the KMP behavior by partial evaluation (i.e., specialized programs that do not backtrack on the text) has essentially followed two channels: managing static information explicitly vs. managing static information implicitly.

### 6.1 Explicit Management of Static Information

A number of variants of the naive string matcher have been published that keep a static trace of the dynamic prefix explicitly [6, 40, 56, 57]. However, as the first author put it in his PhD thesis, "it is not obvious that [the transformation]

```
(define (match-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
                [(< lt 3)
                 -1]
                [(equal? (string-ref t (+ bt 0)) #\a)
                 (loop-pt-1 bt lt)]
                [else
                 (loop-pc-1 bt lt 2)])))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
            [(< lt 3)
             -1]
            [(equal? (string-ref t (+ bt 0)) #\a)
             (loop-pt-1 bt lt)]
            [else
             (loop-pc-1 bt lt 1)]))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))
```

– For all `t`, `(match-abb t)` equals `(match "abb" t)`.
– For all `z`, `(loop-pc-1 bt lt z)` equals `(loop-pc z '(() () ()) '())` evaluated in the scope of `bt` and `lt`,
– `(loop-pt-1 bt lt)` equals `(loop-pt '(() () ()) '(2 1 0))` evaluated in the scope of `bt` and `lt`.

**Fig. 3.** Specialized version of Fig. 1 wrt. `"abb"` à la Boyer and Moore

preserves semantics" [6, page 176]. Indeed, experience has shown that the transformation is error-prone. Therefore, we have spelled out the correctness proof of the generic string matcher [7, Appendix A]. Also, recently, Grobauer and Lawall have revisited how to obtain the KMP behavior by explicitly managing the static prefix, and have proven its correctness [36].

Another concern is the size of specialized programs. As Consel and Danvy put it, "there are no guarantees about the size of these programs, nor about the time taken by the partial evaluator to produce them" [23]. But Grobauer and Lawall have shown that specialized programs with the KMP behavior have a size linear in the length of the pattern [36]. As for the resources (time and space) taken by the partial evaluator, polyvariant specialization does not work in linear time in general. Therefore, it does not produce the specialized program in linear time, in contrast to Knuth, Morris and Pratt's algorithm, which first constructs a failure table in time linear to the pattern string.

### 6.2 Implicit Management of Static Information

A number of partial evaluators have been developed that keep a static trace of the dynamic prefix implicitly, making them able to pass the "KMP test" [55], i.e., to specialize the original quadratic string-matching program into a KMP-like residual program. Such partial evaluators include Futamura's Generalized Partial Computation [32], Smith's partial evaluator for constraint logic programming languages [54], Queinnec and Geffroy's intelligent backtracking [52], supercompilation [34, 35, 55, 56, 57], partial deduction [50], partial evaluators for functional logic programs [5, 46], and the composition of a memoizing interpreter and a standard partial evaluator [33].

Like Similix, none of these partial evaluators has been proven correct and there are no guarantees about the resources required to produce residual programs and the size of the residual programs. Nevertheless, all of them have been tested to produce an output similar to the output of a simple partial evaluator over a naive string matcher that keeps a static trace of the dynamic prefix explicitly.

### 6.3 Other Derivations of Knuth, Morris, and Pratt's String Matcher

Reconstructing the KMP appears to be a favorite in the program-transformation community, in some sense following Knuth's steps since he obtained the behavior of the KMP by calculating it from Cook's construction [45, page 338]. Examples include Dijkstra's use of invariants [28], Bird's tabulation technique [10], Takeichi and Akama's equational reasoning [59], Colussi's use of Hoare logic [21], and just recently Hernández and Rosenblueth's logic-program derivation [37]. Further variants of the KMP can be found, e.g., in Watson's PhD thesis [60] and in Crochemore and Hancart's chapter in the *Handbook of Algorithms and Theory of Computation* [26].

### 6.4 Other Derivations of Boyer and Moore's String Matcher

We have only found two reconstructions of Boyer and Moore's string matcher in the literature: Partsch and Stomp's formal derivation [51] and just recently

Hernández and Rosenblueth's logic-program derivation [37]. But as reviewed in Aho's chapter in the *Handbook of Theoretical Computer Science* [2], several variants of Boyer and Moore's string matcher exist (such as Sunday's variant [58] and Baeza-Yates, Choffrut, and Gonnet's variant [8]), with recurrent concerns about linearity in principle (e.g., Schaback's work [53]) and in practice (e.g., Horspool's work [39]).

## 7 Conclusion and Issues

We have abstracted a naive quadratic substring program with a static cache, and illustrated how specializing various instances of it with respect to a pattern gives rise to KMP-like and Boyer-Moore-like linear-time string matchers. This use of partial evaluation amounts to preprocessing the pattern by program specialization. This preprocessing is not geared to be efficient, since we use an off-the-shelf partial evaluator, but we find it illuminating as a guide for exploring pattern matching in strings.

Generalizing, one can extend the core string matcher to yield the list of all the matches instead of yielding (the index of) the left-most match. One can easily show that the sizes of the specialized programs do not change, as witnessed by Figure 4 that displays the match-all counterpart of the match-leftmost Figure 3.

Shifting perspective, one can also consider specializing the naive quadratic substring program with respect to a text instead of with respect to a pattern. One can then equip this program with a static cache, mutatis mutandis, and specialize various instances of it with respect to a text, obtaining programs that traverse the pattern in linear time. In the late 1980s [47, 49], Malmkjær and Danvy observed that traversing the pattern and the text from left to right yields a program representation of suffix trees [61], i.e., position trees [3], and that suitably pruning the cache yields the smallest automaton recognizing the subwords of a text [12]. Traversing the pattern and the text from right to left, however, is not possible, since we are not given the pattern and thus we do not know its length. Yet we can use the trick of enumerating its possible lengths, or again, more practically, we can be given its length as an extra static information. Partial evaluation then gives rise to a program representation of trees that are to Boyer and Moore what position trees are to Knuth, Morris and Pratt.

Getting back to the original motivation of this article, we would like to state two conclusions.

1. The vision that partial evaluation ought to be able to produce the KMP from a naive string matcher is due to Yoshihiko Futamura, who used it as a guiding light to develop Generalized Partial Computation. We observe that this vision also encompasses other linear string matchers.
2. Neil Jones envisioned that polyvariant specialization ought to be enough to implement a self-applicable partial evaluator. We observe that this vision also applies to the derivation of linear string matchers by partial evaluation.

```
(define (match-all-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          '()
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                (cons bt (loop-pc-1 bt lt 3))      ; collection
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
                [(< lt 3)
                 '()]
                [(equal? (string-ref t (+ bt 0)) #\a)
                 (loop-pt-1 bt lt)]
                [else
                 (loop-pc-1 bt lt 2)])))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
            [(< lt 3)
             '()]
            [(equal? (string-ref t (+ bt 0)) #\a)
             (loop-pt-1 bt lt)]
            [else
             (loop-pc-1 bt lt 1)]))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        '()
        (loop-pt-1 0 lt))))
```

**Fig. 4.** Match-all counterpart of Fig. 3

## 8 Postlude

This section is added in October 2002, as the Festschrift is finally going to press. Since this article was written, the connection between string matching and partial evaluation has been explored further:

– Futamura, Konishi, and Glück have continued to explore the connection between string matching and generalized partial computation, not only in the style of Knuth, Morris and Pratt but also in the style of Boyer and Moore [30, 31].
– Ager, Danvy, and Rohde have taken the next step and have formalized and characterized a class of KMP-like string matchers that are obtained by par-

tial evaluation [1]. They have found that a source matcher using only positive information precisely gives rise to Morris and Pratt's string matcher and that a source matcher using positive information and one character of negative information precisely gives rise to Knuth, Morris, and Pratt's string matcher. Using any further negative information gives rise to other string matchers.

We have also become aware of Charras and Lecroq's comprehensive handbook on exact string matching [18].

*Acknowledgments:* Each of the authors have benefited in one way or another from the research environment established by Neil Jones at DIKU and from his scientific approach to problem solving. Thanks are also due to Yoshihiko Futamura for proposing the initial problem, in 1987; to David Wise for his interest and encouragements, in 1989; and to Andrzej Filinski, Bernd Grobauer, Julia Lawall, and Henning Korsholm Rohde for comments. This article has also benefited from Torben Mogensen's proof-reading.

This work is supported by the NSF grant EIA-9806745 and by the ESPRIT Working Group APPSEM (`www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`). It was carried out while the first author was working at Boston University.

## References

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Chin [19], pages 32–46. Extended version available as the technical report BRICS-RS-02-32.

[2] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. The MIT Press, 1990.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[5] Maria Alpuente, Moreno Falaschi, Pascual Juliàn, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.

[6] Torben Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Department of Computer Science, University of Aarhus, 1993. Technical report PB-453.

[7] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Technical Report BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2001.

[8] Ricardo A. Baeza-Yates, Christian Choffrut, and Gaston H. Gonnet. On Boyer-Moore automata. *Algorithmica*, 12(4/5):268–292, 1994.

[9] Guntis J. Barzdins and Mikhail A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791, Computing Centre of Siberian Division of USSR Academy of Sciences, Novosibirsk, Siberia, 1988.

[10] Richard S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, November 1977.

[11] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[12] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[13] Anders Bondorf. Similix 5.1 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1993. Included in the Similix 5.1 distribution.

[14] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[15] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[16] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[17] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[18] Christian Charras and Thierry Lecroq. Exact string matching algorithms. `http://www-igm.univ-mlv.fr/~lecroq/string/`, 1997.

[19] Wei-Ngan Chin, editor. *ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Aizu, Japan, September 2002. ACM Press.

[20] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, 1999.

[21] Livio Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95:225–251, 1991.

[22] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[23] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[24] Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Unpublished manuscript, December 1989, and talks given at Stanford University, Indiana University, Kansas State University, Northeastern University, Harvard, Yale University, and INRIA Rocquencourt.

[25] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996. ACM Press.

[26] Max Crochemore and Christophe Hancart. Pattern matching in strings. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 11. CRC Press, Boca Raton, 1998.

[27] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[28] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[29] Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, New Generation Computing, Vol. 6, No. 2-3. Ohmsha Ltd. and Springer-Verlag, 1988.

[30] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Automatic generation of efficient string matching algorithms by generalized partial computation. In Chin [19], pages 1–8.

[31] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, 2002.

[32] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Bjørner et al. [11], pages 133–151.

[33] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 183–194, Toulouse, France, May 1994. IEEE Computer Society Press.

[34] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[35] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

[36] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[37] Manuel Hernández and David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001. ACM Press. To appear.

[38] Christoph M. Hoffman and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[39] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[41] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 124–140, Dijon, France, May 1985. Springer-Verlag.

[42] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[43] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[44] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 215–225. ACM Press, June 1996.

[45] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[46] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th International Workshop on Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

[47] Karoline Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1989.

[48] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms.* PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.

[49] Karoline Malmkjær and Olivier Danvy. Preprocessing by program specialization. In Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory*, pages 266–268, Department of Computer Science, University of Aarhus, October 1994. BRICS NS-94-4.

[50] Jonathan Martin and Michael Leuschel. Sonic partial deduction. In Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, number 1755 in Lecture Notes in Computer Science, pages 101–112, Akademgorodok, Novosibirsk, Russia, July 1999. Springer-Verlag.

[51] Helmuth Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2(2):109–122, 1990.

[52] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 109–117, Bordeaux, France, September 1992. IRISA, Rennes, France.

[53] Robert Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM Journal on Computing*, 17(4):648–658, 1988.

[54] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 62–71, New Haven, Connecticut, June 1991. ACM Press.

[55] Morten Heine Sørensen. Turchin's supercompiler revisited. an operational theory of positive information propagation. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, April 1994. DIKU Rapport 94/17.

[56] Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, Scotland, April 1994. Springer-Verlag.

[57] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
[58] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.
[59] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm. *Journal of Information Processing*, 13(4):522–528, 1990.
[60] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.
[61] Peter Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, New York, 1973.

# The Abstraction and Instantiation of String-Matching Programs*

Torben Amtoft[1], Charles Consel[2], Olivier Danvy[3], and Karoline Malmkjær[4]

[1] Department of Computing and Information Sciences, Kansas State University
216 Nichols Hall, Manhattan, KS 66506-2302, USA
[2] INRIA/LaBRI/ENSEIRB
1 avenue du docteur Albert Schweitzer
Domaine universitaire – BP 99, F-33402 Talence Cedex, France
[3] BRICS‡, Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
[4] Ericsson Telebit A/S
Skanderborgvej 232, DK-8260 Viby J., Denmark

**Abstract.** We consider a naive, quadratic string matcher testing whether a pattern occurs in a text; we equip it with a cache mediating its access to the text; and we abstract the traversal policy of the pattern, the cache, and the text. We then specialize this abstracted program with respect to a pattern, using the off-the-shelf partial evaluator Similix.
Instantiating the abstracted program with a left-to-right traversal policy yields the linear-time behavior of Knuth, Morris and Pratt's string matcher. Instantiating it with a right-to-left policy yields the linear-time behavior of Boyer and Moore's string matcher.

*To Neil Jones, for his 60th birthday.*

## 1 Background and Introduction

To build the first self-applicable partial evaluator [41, 42], Neil Jones, Peter Sestoft and Harald Søndergaard simplified its domain of discourse to an extreme: polyvariant specialization of symbolic first-order recursive equations. Source programs were expressed as recursive equations, data were represented as S-expressions, and the polyvariant-specialization strategy amounted to an interprocedural version of Kildall's constant-propagation algorithm [4, 17]: for each of its calls, a recursive equation is specialized with respect to its known arguments and the resulting specialized equation is indexed with these known arguments and cached in a worklist. Specialization terminates once the worklist is complete, and the specialized equations in the worklist form the specialized program.

* Corresponding authors: Torben Amtoft (`tamtoft@cis.ksu.edu`) and Olivier Danvy (`danvy@brics.dk`). Extended version available as the BRICS technical report RS-01-12.

‡ Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

Since then, this basic framework has been subjected to many developments. For example, not all calls need to be specialized and give rise to a specialized recursive equation: many of them can merely be unfolded. Data need not be only symbolic either—numbers, characters, strings, pairs, higher-order functions, etc. can also be handled, and so can side effects. Finally, source programs need not be mere recursive equations—e.g., block-structured programs are amenable to partial evaluation as well.

Nevertheless, the basic strategy of polyvariant program-point specialization remains the same: generating mutually recursive residual equations, each of them corresponding to a source program point indexed by known values. In 1987, at the workshop on Partial Evaluation and Mixed Computation [11, 29], this very simple specialization strategy was challenged. Yoshihiko Futamura said that he had needed to design a stronger form of partial evaluation—Generalized Partial Computation [32]—to be able to specialize a naive quadratic-time string matcher into Knuth, Morris, and Pratt's (KMP) linear-time string matcher [45]. He therefore asked the DIKU group whether polyvariant specialization was able to obtain the KMP.

## 1.1 Obtaining the KMP

The naive matching algorithm tests whether the pattern is the prefix of one of the suffixes of the text. Every time the pattern does not match a prefix of a suffix of the text, the pattern is shifted by one position to try the next suffix. In contrast, the KMP algorithm proceeds in two phases:

1. given the pattern, it constructs a 'failure table' in time linear in the size of the pattern; and
2. it traverses the text linearly, using the failure table to determine how much the pattern should be shifted in case of mismatch.

The first thing one could try is to specialize the traversal of the text with respect to a failure table. Unsurprisingly, the result is a linear-time residual program. In their article [45, page 330], Knuth, Morris and Pratt display a similar program where the failure table has been "compiled" into the control flow.

More in the spirit of Futamura's challenge, one can consider the naive specification, i.e., a (known) pattern string occurs in an (unknown) text if it is the prefix of one of its suffixes. On the left of each mismatch, the pattern and the text are equal:

```
           ---->
 Pattern  +---------o-----------+
           |||||||||* mismatch
    Text  +---------o----------------------------------------+
           ---->
```

Yet, the naive specification shifts the pattern one place to the right and re-scans it together with the text up to the mismatch point and beyond:

```
           ---->    ---->
Pattern   +---------o-----------+
           ||||||||||||||
   Text  +---------o----------------------------------------+
           ---->    ---->
```

Instead, one can match two instances of the pattern (the pattern and a prefix of the pattern equal to the prefix of the text matched so far, with one character chopped off) up to the point of mismatch and then resume scanning the pattern and the text:

```
           ---->
Pattern  +---------o
           ||||||||---->
Pattern   +---------o-----------+
           ---->   ||||||
   Text            o----------------------------------------+
                   ---->
```

This staged program is equivalent to the original program. In particular, its complexity is the same. But if one specializes it with respect to a pattern, matching the pattern against itself is carried out at partial-evaluation time. Furthermore, since the index on the text only increases, specializing this staged program yields a string matcher that does not back up on the text and thus operates in time linear in the text. Therefore part of the challenge is met: polyvariant specialization can turn a (staged) quadratic string matcher into a linear one.

But how does this linear string matcher compare to the KMP?

Short of a formal proof, one can compare, for given patterns, (1) the corresponding residual string matchers and (2) the results of specializing the second phase of the KMP with respect to the corresponding failure tables produced by the first phase. It turns out that the resulting string matchers are indeed isomorphic, suggesting that mere polyvariant specialization can obtain the KMP if the naive string matcher is staged as pictured above [22].

This experiment clarifies that linearity requires one to keep a static view of the dynamic prefix of the text during specialization. This static view can be kept explicitly in the source program, as above. Alternatively, a partial evaluator could implicitly memoize the outcome of every dynamic test. (For example, given a dynamic variable `x`, a conditional expression `(if (odd? x) e1 e2)`, and some knowledge about `odd?`, such a partial evaluator would know that `x` is odd when processing `e1` and that `x` is even when processing `e2`.) Futamura's Generalized Partial Computation is such a partial evaluator. Therefore, it automatically keeps a static view of the dynamic prefix of the text during specialization and thus obtains the KMP out of a naive, unstaged string matcher. So the key to linearity is precisely the static view of the dynamic prefix of the text during specialization.

To summarize, we need a static view of the dynamic prefix of the text during specialization. Whether this view is managed implicitly in the partial evaluator (as in Generalized Partial Computation) or explicitly in the source program (as in basic polyvariant specialization) is a matter of convenience. If the management is implicit, the program is simpler but the partial evaluator is more complex. If the management is explicit, the program is more complicated but the partial evaluator is simpler.

That said, there is more to the KMP than linearity over the text—there is also linearity over the pattern. Indeed, the KMP algorithm operates on a failure table that is constructed in time linear in the size of the pattern, whereas in general, a partial evaluator does not operate in linear time.

### 1.2 This Work

We extend the construction of Section 1.1 to obtain linear string matchers in the style of Knuth, Morris, and Pratt as well as string matchers in the style of Boyer and Moore from the same quadratic string matcher.

To this end, we instrument the naive string matcher with a *cache* that keeps a trace of what is known about the text, similarly to a memory cache on a hardware processor. Any access to the text is mediated by an access to the cache. If the information is already present in the cache, the text is not accessed. If the information is not present in the cache, the text is accessed and the cache is updated.

The cache has the same length as the pattern and it gives us a static view of the text. We make it keep track both of what is known to occur in the text and of what is known not to occur in the text. Each entry of the cache thus contains either some *positive* information, namely the corresponding character in the text, or some *negative* information, namely a list of (known) characters that do not match the corresponding (unknown) character in the text. The positive information makes it possible to test characters statically, i.e., without accessing the text. The negative information makes it possible to detect statically when a test would fail. Initially, the cache contains no information (i.e., each entry contains an empty list of negative information).

By construction, the cache-based string matcher consults an entry in the text either not at all or repeatedly until the entry is recognized. An entry in the text can be left unconsulted because an adjacent character does not match the pattern. Once an entry is recognized, it is never consulted again because the corresponding character is stored in the cache. The entry can be consulted repeatedly, either until it is recognized or until the pattern has shifted. Each unrecognized character is stored in the cache and thus the number of times an entry is accessed is at most the number of different characters in the pattern. Therefore, each entry in the text is consulted a bounded number of times. The specialized versions of the cache-based string matcher inherit this property, and therefore their matching time is linear with respect to the length of the text.

On the other hand, the size of the specialized versions can be large: polyvariant specialization yields residual equations corresponding to source program points indexed by known values—here, specifically, the cache. In order to reduce the size of the residual programs, we allow ourselves to *prune* the cache in the source string matcher. For example, two extreme choices are to remove all information and to remove no information from the cache, but variants are possible.

Finally, since the KMP compares the pattern and the text from left to right whereas the Boyer and Moore compares the pattern and the text from right to left, we also parameterize the string matcher with a traversal policy. For example, two obvious choices are left to right and right to left, but variants are possible.

Our results are that traversing the pattern from left to right yields the KMP behavior, with variants, and that traversing the pattern from right to left yields the Boyer and Moore behavior [15, 16], also with variants. (The variants are determined by how we traverse the pattern and the text and how we prune the cache.) These results are remarkable (1) because they show that it is possible to obtain both the KMP linear behavior and the Boyer and Moore linear behavior from the *same* source program, and (2) because even though it is equipped with a cache, this source program is a naive quadratic string-matching program.

The first author is aware of these results since the early 1990s [6] and the three other authors since the late 1980s [24]. Since then, these results have been reproduced and extended by Queinnec and Geffroy [52], but otherwise they have only been mentioned informally [23, Section 6.1].

## 1.3 A Note on Program Derivation and Reverse Engineering

We would like to point out that we are not using partial evaluation to reverse-engineer the KMP and the Boyer and Moore string matchers. What we are attempting to do here is to exploit the simple observation that, in the naive quadratic string matcher, and as depicted in Section 1.1, rematching can be optimized away by partial evaluation.

In 1988, we found that the resulting specialized programs behave like the KMP, for a left-to-right traversal. We then conjectured that for a right-to-left traversal, the resulting specialized programs should behave like Boyer and Moore—which they do. In fact, for any traversal, partial evaluation yields a linear-time residual program (which may be sizeable), at least as long as no pruning is done. We have observed that the method scales up to patterns with wild cards and variables as well as to pattern matching in trees à la Hoffman and O'Donnell [38]. For two other examples, the method has led Queinnec and Geffroy to derive Aho and Corasick's as well as Commentz-Walter's string matchers [52].

## 1.4 Overview

The rest of this article is organized as follows. Section 2 presents the core program and its parameters. Section 3 specifies the action of partial evaluation on the core program. Section 4 describes its KMP instances and Section 5 describes its Boyer and Moore instances. Related work is reviewed in Section 6 and Section 7 concludes. A correctness proof of the core program can be found in the extended version of this article [7].

Throughout, we use the programming language Scheme [43], or, more precisely, Petite Chez Scheme (`www.scheme.com`) and the partial evaluator Similix [13, 14].

## 2 The Abstracted String Matcher

We first specify the operations on the cache (Section 2.1). Then we describe the cache-based string matcher (Section 2.2).

### 2.1 The Cache and Its Operations

The cache has the same size as the pattern. It holds a picture of what is currently known about the text: either we know a character, or we know nothing about a character, or we know that a character is not equal to some given characters. Initially, we know nothing about any entry of the text. In the course of the algorithm, we may get to know some characters that do not match an entry, and eventually we may know this entry. We then disregard the characters that do not match the entry, and we only consider the character that matches the entry. Accordingly, each entry of the cache contains either some positive information (one character) or some negative information (a list of characters, possibly empty). We test this information with the following predicates.

```
(define pos? char?)
```

```
(define (neg? e)
  (or (null? e) (pair? e)))
```

These two predicates are mutually exclusive.

We implement the cache as a list and we operate on it as follows.

– Given a natural number specifying the length of the pattern, `cache-init` constructs an initial cache containing empty lists of characters.
```
(define (cache-init n)
  (if (= n 0)
      '()
      (cons '() (cache-init (- n 1)))))
```

– In the course of string matching, the pattern slides to the right of the text by one character. Paralleling this shift, `cache-shift` maps a cache to the corresponding shifted cache where the right-most entry is negative and empty.
```
(define (cache-shift c)
  (append (cdr c) (list '())))
```

– Given an index, the predicates `cache-ref-pos?` and `cache-ref-neg?` test whether a cache entry is positive or negative.
```
(define (cache-ref-pos? c i)
  (pos? (list-ref c i)))
```
```
(define (cache-ref-neg? c i)
  (neg? (list-ref c i)))
```

– Positive information is fetched by `cache-ref-pos` and negative information by `cache-ref-neg`.

```
(define cache-ref-pos list-ref)    (define cache-ref-neg list-ref)
```

– Given a piece of positive information at an index, `cache-extend-pos` extends the cache with this positive information at that index. The result is a new cache where the positive information supersedes any negative information at that index.

```
(define (cache-extend-pos c i pos)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons pos (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1)))))])
    (walk c i)))
```

– Given more negative information at an index, `cache-extend-neg` extends the cache with this negative information at that index. The result is a new cache.

```
(define (cache-extend-neg c i neg)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons (cons neg (car c)) (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1)))))])
    (walk c i)))
```

– Given the cache, `schedule-pc` and `schedule-pt` respectively yield the series of indices (represented as a list) through which to traverse the pattern and the cache, and through which to traverse the pattern and the text, respectively. The formal requirements on `schedule-pc` and `schedule-pt` are stated elsewhere [7, Appendix A].

1. The following definition of `schedule-pc` specifies a left-to-right traversal of the pattern and the cache:

```
(define (schedule-pc c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (cons i (walk (+ i 1) (cdr c)))]
                [else
                 (if (null? (car c))
                     (walk (+ i 1) (cdr c))
                     (cons i (walk (+ i 1) (cdr c))))]))])
    (walk 0 c)))
```

For example, applying `schedule-pc` to an empty cache yields the empty list. This list indicates that there is no need to compare the pattern and the cache.
For another example, applying `schedule-pc` to a cache containing two pieces of positive information, and then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(0 1 2)`. This list indicates that the pattern and the cache should be successively compared at indices 0, 1, and 2, and that the rest of the cache should be ignored.

2. The following definition of `schedule-pt` specifies a left-to-right traversal of the pattern and of the text:
```
(define (schedule-pt c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (walk (+ i 1) (cdr c))]
                [else
                 (cons i (walk (+ i 1) (cdr c)))]))])
    (walk 0 c)))
```
For example, applying `schedule-pt` to an empty cache of size 3 (as could be obtained by evaluating `(cache-init 3)`) yields the list `(0 1 2)`. This list indicates that the pattern and the text should be successively compared at indices 0, 1, and 2.
For another example, applying `schedule-pt` to a cache containing two pieces of positive information, then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(2 3 4 5)`. This list indicates that the pattern and the text already agree at indices 0 and 1, and should be successively compared at indices 2, 3, 4, and 5. (The negative information at index 2 in the cache is of no use here.)

– Finally, we give ourselves the possibility of pruning the cache with `cache-prune`. Pruning the cache amounts to shortening lists of negative information and resetting positive information to the empty list of negative information. For example, the identity function prunes nothing at all:
```
(define (cache-prune c)
  c)
```

## 2.2 The Core Program

The core program, displayed in Figure 1, is a cache-based version of a naive quadratic program checking whether the pattern occurs in the text, i.e., if the pattern is a prefix of one of the successive suffixes of the text, from left to right.

```
(define (match p t)
  (let ([lp (string-length p)])         ;;; lp is the length of p
    (if (= lp 0)
        0
        (let ([lt (string-length t)])   ;;; lt is the length of t
          (if (< lt lp)
              -1
              (match-pt p lp (cache-init lp) t 0 lt))))))

(define (match-pc p lp c t bt lt)
  ;;; matches p[0..lp-1] and c[0..lp-1]
  (letrec ([loop-pc
            (lambda (z c is)
              (if (null? is)
                  (let ([bt (+ bt z)] [lt (- lt z)])
                    (if (< lt lp)
                        -1
                        (match-pt p lp c t bt lt)))
                  (let ([i (car is)])
                    (if (if (cache-ref-pos? c i)
                            (equal? (string-ref p i)
                                    (cache-ref-pos c i))
                            (not (member (string-ref p i)
                                         (cache-ref-neg c i))))
                        (loop-pc z c (cdr is))
                        (let ([c (cache-shift c)])
                          (loop-pc (+ z 1) c (schedule-pc c)))))))])
    (loop-pc 1 c (schedule-pc c))))

(define (match-pt p lp c t bt lt)
  ;;; matches p[0..lp-1] and t[bt..bt+lp-1]
  (letrec ([loop-pt                     ;;; bt is the base index of t
            (lambda (c is)
              (if (null? is)
                  bt
                  (let* ([i (car is)] [x (string-ref p i)])
                    (if (equal? (string-ref t (+ bt i)) x)
                        (loop-pt (cache-prune
                                   (cache-extend-pos c i x))
                                 (cdr is))
                        (match-pc p lp
                                  (cache-shift
                                    (cache-extend-neg c i x))
                                  t bt lt)))))])
    (loop-pt c (schedule-pt c))))
```

**Fig. 1.** The core quadratic string-matching program

The program is composed of two mutually recursive loops: a static one checking whether the pattern matches the cache and a dynamic one checking whether the pattern matches the rest of the text.

*The main function,* `match`*:* Initially, `match` is given a pattern `p` and a text `t`. It computes their length (`lp` and `lt`, respectively), and after checking that there is room in the text for the pattern, it initiates the dynamic loop with an empty cache. The result is either $-1$ if the pattern does not occur in the text, or the index of the left-most occurrence of the pattern in the text.

*The static loop,* `match-pc`*:* The pattern is iteratively matched against the cache using `loop-pc`. For each mismatch, the cache is shifted and `loop-pc` is resumed. Eventually, the pattern agrees with (i.e., matches) the cache—if only because after repeated iterations of `loop-pc` and shifts of the cache, the cache has become empty. Then, if there is still space in the text for the pattern, the dynamic loop is resumed.

In more detail, `match-pc` is passed the pattern `p`, its length `lp`, the cache `c`, and also the text `t`, its base index `bt`, and the length `lt` of the remaining text to match in the dynamic loop. The static loop checks iteratively (with `loop-pc`) whether the pattern and the cache agree. The traversal takes place in the order specified by `schedule-pc` (e.g., left to right or right to left). For each index, `loop-pc` tests whether the information in the cache is positive or negative. In both cases, if the corresponding character in the pattern agrees (i.e., either matches or does not mismatch), `loop-pc` iterates with the next index. Otherwise the cache is shifted and `loop-pc` iterates. In addition, `loop-pc` records in `z` how many times the cache has been shifted (initially 1 since `match-pc` is only invoked from the dynamic loop). Eventually, the pattern and the cache agree. The new base and the new length of the text are adjusted with `z`, and if there is enough room in the text for the pattern to occur, the dynamic loop takes over.

When the dynamic loop takes over, the pattern and the cache completely agree, i.e., for all indices $i$:

– either the cache information is positive at index $i$ and it contains the same character as the pattern at index $i$;
– or the cache information is negative at index $i$ and the character at index $i$ in the pattern does not occur in the list of characters contained in this negative information.

*The dynamic loop,* `match-pt`*:* The pattern is iteratively matched against the parts of the text that are not already in the cache, using `loop-pt`. At each iteration, the cache is updated. If a character matches, the cache is updated with the corresponding positive information before the next iteration of `loop-pt`. If a character does not match, the cache is updated with the corresponding negative information and the static loop (i.e., `match-pc`) is resumed with a shifted cache.

In more detail, `match-pt` is passed the pattern `p`, its length `lp`, the cache `c`, and the text `t`, its base index `bt`, and the length `lt` of the remaining text to match. The traversal takes place in the order specified by `schedule-pt` (e.g., left to right or right to left). For each index, `loop-pt` tests the corresponding character in the text against the corresponding character in the pattern. If the two characters are equal, then `loop-pt` iterates with an updated and pruned cache. Otherwise, the static loop takes over with an updated and shifted cache. If `loop-pt` runs out of indices, pattern matching succeeds and the result is the base index in the text.

*The cache:* The cache is only updated (positively or negatively) during the dynamic loop. Furthermore, it is only updated with a character *from the pattern* and never with one from the text. Considering that the pattern is known (i.e., static) and that the text is unknown (i.e., dynamic), constructing the cache with characters from the pattern ensures that it remains known at partial-evaluation time. This selective update of the cache is the key for successful partial evaluation.

We allow `schedule-pt` to return indices of characters that are already in the cache and to return a list of indices with repetitions. These options do not invalidate the correctness proof of the core program, but they make it possible to reduce the size of residual programs, at the expense of redundant tests. This information-theoretic weakening is common in published studies of the Boyer-Moore string matcher [8, 39, 51]: the challenge then is to show that the weakened string matcher still operates in linear time [53].

In the current version of Figure 1, we only prune the cache at one point: after a call to `cache-extend-pos`. This lets us obtain a number of variants of string matchers in the style of Boyer and Moore. But as noted in Section 5.2, at least one remains out of reach.

## 3 Partial Evaluation

Specializing the core string matcher with respect to a pattern takes place in the traditional three steps: preprocessing (binding-time analysis), processing (specialization), and postprocessing (unfolding).

### 3.1 Binding-Time Analysis

Initially, `p` is static and `t` is dynamic.

In `match`, `lp` is static since `p` is static. The first let expression and the first conditional expression are thus static. Conversely, `lt` is dynamic since `t` is dynamic. The second let expression and the second conditional expression are thus dynamic. After binding-time analysis, `match` is annotated as follows.

```
(define (match p^s t^d)
  (let^s ([lp^s (string-length p^s)])
    (if^s (= lp^s 0^s)
        0^d
        (let^d ([lt^d (string-length t^d)])
          (if^d (< lt^d lp^s)
              -1^d
              (match-pt p^s lp^s (cache-init lp^s) t^d 0^d lt^d))))))
```

In `match-pc` and in `match-pt`, `p`, `lp` and `c` are static, and `t`, `bt`, and `lt` are dynamic. (Therefore, `0` is generalized to be dynamic in the initial call to `match-pt`. Similarly, since `match` returns a dynamic result, `0` and `-1` are also generalized to be dynamic.)

In `loop-pc`, `c` and `is` are static and—due to the automatic poor man's generalization of Similix [40]—`z` is dynamic. Except for the conditional expression testing `(< lt lp)` and its enclosing let expression, which are dynamic, all syntactic constructs are static.

After binding-time analysis, `match-pc` and `loop-pc` are annotated as follows.

```
(define (match-pc p^s lp^s c^s t^d bt^d lt^d)
  (letrec ([loop-pc
            (lambda (z^d c^s is^s)
              (if^s (null? is^s)
                   (let^d ([bt^d (+ bt^d z^d)] [lt^d (- lt^d z^d)])
                     (if^d (< lt^d lp^s)
                          -1^d
                          (match-pt p^s lp^s c^s t^d bt^d lt^d)))
                   (let^s ([i^s (car is^s)])
                     (if^s (if^s (cache-ref-pos? c^s i^s)
                                (equal? (string-ref p^s i^s)
                                        (cache-ref-pos c^s i^s))
                                (not (member (string-ref p^s i^s)
                                             (cache-ref-neg c^s i^s))))
                          (loop-pc z^d c^s (cdr is^s))
                          (let^s ([c^s (cache-shift c^s)])
                            (loop-pc (+ z^d 1^d) c^s (schedule-pc c^s)))))))])
    (loop-pc 1^d c^s (schedule-pc c^s))))
```

In `loop-pt`, `c` and `is` are static. Except for the conditional expression performing an equality test, all syntactic constructs are static.

After binding-time analysis, `match-pt` and `loop-pt` are annotated as follows.

```
(define (match-pt p^s lp^s c^s t^d bt^d lt^d)
  (letrec ([loop-pt
            (lambda (c^s is^s)
              (if^s (null? is^s)
                   bt^d
                   (let*^s ([i^s (car is^s)] [x^s (string-ref p^s i^s)])
                     (if^d (equal? (string-ref t^d (+ bt^d i^s)) x^s)
                          (loop-pt (cache-prune
                                     (cache-extend-pos c^s i^s x^s))
                                   (cdr is^s))
                          (match-pc p^s lp^s
                                    (cache-shift
                                      (cache-extend-neg c^s i^s x^s))
                                    t^d bt^d lt^d)))))])
    (loop-pt c^s (schedule-pt c^s))))
```

## 3.2 Polyvariant Specialization

In Similix [14], the default specialization strategy is to unfold all function calls and to treat every conditional expression with a dynamic test as a specialization point. Here, since there is exactly one specialization point in `loop-pc` as well as in `loop-pt`, without loss of generality, we refer to each instance of a specialization point as *an instance of* `loop-pc` and *an instance of* `loop-pt`, respectively.

Each instance of `loop-pc` is given a base index in the text, the remaining length of the text, and a natural number. The body of this instance is a clone of the dynamic let expression and of the dynamic conditional expression in the original `loop-pc`: it increments `bt` and decrements `lt` with the natural number, checks whether the remaining text is large enough, and calls an instance of `loop-pt`, as specified by the following template (where `<lp>` is a natural number denoting the length of the pattern).

```
(define (loop-pc-... t bt lt z)
  (let ([bt (+ bt z)] [lt (- lt z)])
    (if (< lt <lp>)
        -1
        (loop-pt-... t bt lt))))
```

Each instance of `loop-pt` is given a base index in the text and the remaining length of the text (their sum is always equal to the length of the text). It tests whether a character in the text is equal to a fixed character. If so, it either returns an index or branches to another instance of `loop-pt`. If not, it branches to an instance of `loop-pc` with a fixed shift. (Below, `<i>` stands for a non-negative integer, `<n>` stands for a natural number, and `<c>` stands for a character.)

```
(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      bt
      (loop-pc-... t bt lt <n>)))

(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      (loop-pt-... t bt lt)
      (loop-pc-... t bt lt <n>)))
```

Therefore, a residual program consists of mutually recursive specialized versions of the two specialization points, and of a main function computing the length of the text, checking whether it is large enough, and calling a specialized version of `loop-pt`. The body of each auxiliary function is constructed out of instances of the dynamic parts of the programs, i.e., it fits one of the templates above [25, 48].

We have written the holes in the templates between brackets. The name of each residual function results from concatenating `loop-pc-` and `loop-pt-` to a fresh index. Each residual function is closed, i.e., it has no free variables.

Overall, the shape of a residual program is as follows.

```
(define (match-0 t)
  (let ([lt (string-length t)])
    (if (< lt ...)
        -1
        (loop-pt-1 t 0 lt))))

(define (loop-pc-1 t bt lt z) ...)

(define (loop-pc-2 t bt lt z) ...)

...

(define (loop-pt-1 t bt lt) ...)

(define (loop-pt-2 t bt lt) ...)

...
```

This general shape should make it clear how residual programs relate to (and how their control flow could be "decompiled" into) a KMP-like failure table, as could be obtained by data specialization [9, 20, 44, 47].

### 3.3 Post-unfolding

To make the residual programs more readable, Similix post-unfolds residual functions that are called only once. It also defines the remaining functions locally to the main residual function. (By the same token, it could lambda-drop the variable `t`, as we do in the residual programs displayed in Sections 4, 5, and 7, for readability [27].)

So all in all, a specialized program is defined as a main function (an instance of `match`) with many locally defined and mutually recursive auxiliary functions (instances of `loop-pc` and `loop-pt`), each of which is called more than once.

## 4 The KMP Instances

In the KMP style of string matching, the pattern and the corresponding prefix of the text are traversed from left to right. Therefore, we define `schedule-pt` so that `match-pt` proceeds from left to right. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned. Furthermore, all residual programs are independent of `schedule-pc` since it is applied at partial-evaluation time. Since Consel and Danvy's work [22], this traversal has been consistently observed to coincide with the traversal of Knuth, Morris and Pratt's string matcher. (We are not aware of any formal proof of this coincidence, but we expect that we will be able to show that the two matchers operate in lock step.)

```
(define (match-aaa t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 0)) #\a)
        (if (equal? (string-ref t (+ bt 1)) #\a)
            (if (equal? (string-ref t (+ bt 2)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (loop-pc-1 bt lt 2))
        (loop-pc-1 bt lt 1)))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))
```

- For all `t`, `(match-aaa t)` equals `(match "aaa" t)`.
- For all `z`, `(loop-pc-1 bt lt z)` equals `(loop-pc z '(() () ()) '())` evaluated in the scope of `bt` and `lt`.
- `(loop-pt-1 bt lt)` equals `(loop-pt '(() () ()) '(0 1 2))` evaluated in the scope of `bt` and `lt`.

**Fig. 2.** Specialized version of Fig. 1 wrt. `"aaa"` à la Knuth, Morris and Pratt

This 'KMP behavior' has been already described in the literature (see Section 6), so we keep this section brief. In this instantiation, the cache is naturally composed of three parts: a left part with positive information, a right part with empty negative information, and between the left part and the right part, at most one entry with non-empty negative information. This entry may or may not be pruned away, which affects both the size of the residual code and its runtime, as analyzed by Grobauer and Lawall [36].

Figure 2 displays a specialized version of the core program in Figure 1 with respect to the pattern string `"aaa"`. This specialized version is renamed for readability. We instantiated `cache-prune` to the identity function. The code for `loop-pt-1` reveals that a left-to-right strategy is indeed employed, in that the text is addressed with offsets 0, 1, and 2.

## 5 The Boyer and Moore Instances

### 5.1 The Boyer and Moore Behavior

In the Boyer and Moore style of string matching, the pattern and the corresponding prefix of the text are traversed from right to left. Therefore, we define `schedule-pt` so that `match-pt` proceeds from right to left. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned and again independently of `schedule-pc`, which is executed at partial-evaluation time. We have consistently observed that this traversal coincides with the traversal of string matchers à la Boyer and Moore,[1] provided that `schedule-pt` processes the non-empty negative entries in the cache before the empty ones. Then there will always be at most one entry with non-empty negative information.

Except for the first author's PhD dissertation [6], this "Boyer and Moore behavior" has not been described in the literature, so we describe it in some detail here. Initially, in Figure 1, `loop-pt` iteratively builds a suffix of positive information. This suffix is located on the right of the cache, and grows from right to left. In case of mismatch, `loop-pt` punctuates the suffix with one (non-empty) negative entry, shifts the cache to the right, and resumes `match-pc`. The shift transforms the suffix into a segment, which drifts to the left of the cache while `loop-pt` and `loop-pc` continue to interact. Subsequently, if a character match is successful, the negative entry in the segment becomes positive, and the matcher starts building up a new suffix. The cache is thus composed of zero or more segments of positive information, the right-most of which may be punctuated on the left by one negative entry. Each segment is the result of an earlier unsuccessful call to `loop-pt`. The right-most segment (resp. the suffix), is the witness of the latest (resp. the current) call to `loop-pt`. In the course of string matching, adjacent segments can merge into one.

## 5.2 Pruning

**No Pruning:** Never pruning the cache appears to yield Knuth's optimal Boyer and Moore string matcher [45, page 346].

**Pruning Once:** Originally [15], Boyer and Moore maintained two tables. The first table indexes each character in the alphabet with the right-most occurrence (if any) of this character in $p$. The second table indexes each position in $p$ with the rightmost occurrence (if any) of the corresponding suffix, preceded by a different character, elsewhere in $p$.

We have not obtained Boyer and Moore's original string matcher because it exploits the two tables in a rather unsystematic way. Nevertheless, the following instantiation appears to yield a simplified version of Boyer and Moore's original string matcher, where only the first table is used (and hence where linearity does not hold):

– `cache-prune` empties the cache;
– after listing the entries with non-empty negative information, `schedule-pt` lists *all* negative entries, even those (zero or one) that are already listed.

[1] Again, we expect to be able to prove this coincidence by showing that the two traversals proceed in lock step.

(This redundancy is needed in order to keep the number of residual functions small, at the expense of redundant dynamic (residual) tests.)

Partsch and Stomp also observed that Boyer and Moore's original string matcher uses the two tables unsystematically [51]. A more systematic take led them to formally derive the alternative string matcher hinted at by Boyer and Moore in their original article [15, page 771]. It appears that this string matcher can be obtained as follows:

- `cache-prune` removes all information except for the right-most suffix of positive information (i.e., the maximal set of the form $\{j, \ldots, lp - 1\}$ with all the corresponding entries being positive);
- after having listed the non-empty negative information, `schedule-pt` lists *all* indices (including the positive ones).

**More Pruning:** The pruning strategy of Figure 1 is actually sub-optimal. A better one is to prune the cache before reaching each specialization point. Doing so makes it possible to obtain what looks like Horspool's variant of Boyer and Moore's string matcher [39].

### 5.3 An Example

Figure 3 displays a specialized version of the program in Figure 1 with respect to the pattern string `"abb"`. This specialized version is renamed for readability. We used instantiations yielding Partsch and Stomp's variant. The code for `loop-pt-1` reveals that a right-to-left strategy is indeed employed, in that the text is addressed with offsets 2, 1, and 0. (NB. Similix has pretty-printed successive `if` expressions into one `cond` expression.)

## 6 Related Work

In his MS thesis [55, Sec. 8.2], Sørensen has observed that once the pattern is fixed, the naive string matcher is de facto linear—just with a factor proportional to the length of this pattern, *lp*. The issue of obtaining the KMP behavior is therefore to produce a specialized program that runs in linear time with a factor independent of *lp*. Since Consel and Danvy's original solution [22], obtaining the KMP behavior by partial evaluation (i.e., specialized programs that do not backtrack on the text) has essentially followed two channels: managing static information explicitly vs. managing static information implicitly.

### 6.1 Explicit Management of Static Information

A number of variants of the naive string matcher have been published that keep a static trace of the dynamic prefix explicitly [6, 40, 56, 57]. However, as the first author put it in his PhD thesis, "it is not obvious that [the transformation]

```
(define (match-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
                [(< lt 3)
                 -1]
                [(equal? (string-ref t (+ bt 0)) #\a)
                 (loop-pt-1 bt lt)]
                [else
                 (loop-pc-1 bt lt 2)])))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
            [(< lt 3)
             -1]
            [(equal? (string-ref t (+ bt 0)) #\a)
             (loop-pt-1 bt lt)]
            [else
             (loop-pc-1 bt lt 1)]))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))
```

– For all `t`, `(match-abb t)` equals `(match "abb" t)`.
– For all `z`, `(loop-pc-1 bt lt z)` equals `(loop-pc z '(() () ()) '())` evaluated in the scope of `bt` and `lt`,
– `(loop-pt-1 bt lt)` equals `(loop-pt '(() () ()) '(2 1 0))` evaluated in the scope of `bt` and `lt`.

**Fig. 3.** Specialized version of Fig. 1 wrt. `"abb"` à la Boyer and Moore

preserves semantics" [6, page 176]. Indeed, experience has shown that the transformation is error-prone. Therefore, we have spelled out the correctness proof of the generic string matcher [7, Appendix A]. Also, recently, Grobauer and Lawall have revisited how to obtain the KMP behavior by explicitly managing the static prefix, and have proven its correctness [36].

Another concern is the size of specialized programs. As Consel and Danvy put it, "there are no guarantees about the size of these programs, nor about the time taken by the partial evaluator to produce them" [23]. But Grobauer and Lawall have shown that specialized programs with the KMP behavior have a size linear in the length of the pattern [36]. As for the resources (time and space) taken by the partial evaluator, polyvariant specialization does not work in linear time in general. Therefore, it does not produce the specialized program in linear time, in contrast to Knuth, Morris and Pratt's algorithm, which first constructs a failure table in time linear to the pattern string.

### 6.2 Implicit Management of Static Information

A number of partial evaluators have been developed that keep a static trace of the dynamic prefix implicitly, making them able to pass the "KMP test" [55], i.e., to specialize the original quadratic string-matching program into a KMP-like residual program. Such partial evaluators include Futamura's Generalized Partial Computation [32], Smith's partial evaluator for constraint logic programming languages [54], Queinnec and Geffroy's intelligent backtracking [52], supercompilation [34, 35, 55, 56, 57], partial deduction [50], partial evaluators for functional logic programs [5, 46], and the composition of a memoizing interpreter and a standard partial evaluator [33].

Like Similix, none of these partial evaluators has been proven correct and there are no guarantees about the resources required to produce residual programs and the size of the residual programs. Nevertheless, all of them have been tested to produce an output similar to the output of a simple partial evaluator over a naive string matcher that keeps a static trace of the dynamic prefix explicitly.

### 6.3 Other Derivations of Knuth, Morris, and Pratt's String Matcher

Reconstructing the KMP appears to be a favorite in the program-transformation community, in some sense following Knuth's steps since he obtained the behavior of the KMP by calculating it from Cook's construction [45, page 338]. Examples include Dijkstra's use of invariants [28], Bird's tabulation technique [10], Takeichi and Akama's equational reasoning [59], Colussi's use of Hoare logic [21], and just recently Hernández and Rosenblueth's logic-program derivation [37]. Further variants of the KMP can be found, e.g., in Watson's PhD thesis [60] and in Crochemore and Hancart's chapter in the *Handbook of Algorithms and Theory of Computation* [26].

### 6.4 Other Derivations of Boyer and Moore's String Matcher

We have only found two reconstructions of Boyer and Moore's string matcher in the literature: Partsch and Stomp's formal derivation [51] and just recently

Hernández and Rosenblueth's logic-program derivation [37]. But as reviewed in Aho's chapter in the *Handbook of Theoretical Computer Science* [2], several variants of Boyer and Moore's string matcher exist (such as Sunday's variant [58] and Baeza-Yates, Choffrut, and Gonnet's variant [8]), with recurrent concerns about linearity in principle (e.g., Schaback's work [53]) and in practice (e.g., Horspool's work [39]).

## 7 Conclusion and Issues

We have abstracted a naive quadratic substring program with a static cache, and illustrated how specializing various instances of it with respect to a pattern gives rise to KMP-like and Boyer-Moore-like linear-time string matchers. This use of partial evaluation amounts to preprocessing the pattern by program specialization. This preprocessing is not geared to be efficient, since we use an off-the-shelf partial evaluator, but we find it illuminating as a guide for exploring pattern matching in strings.

Generalizing, one can extend the core string matcher to yield the list of all the matches instead of yielding (the index of) the left-most match. One can easily show that the sizes of the specialized programs do not change, as witnessed by Figure 4 that displays the match-all counterpart of the match-leftmost Figure 3.

Shifting perspective, one can also consider specializing the naive quadratic substring program with respect to a text instead of with respect to a pattern. One can then equip this program with a static cache, mutatis mutandis, and specialize various instances of it with respect to a text, obtaining programs that traverse the pattern in linear time. In the late 1980s [47, 49], Malmkjær and Danvy observed that traversing the pattern and the text from left to right yields a program representation of suffix trees [61], i.e., position trees [3], and that suitably pruning the cache yields the smallest automaton recognizing the subwords of a text [12]. Traversing the pattern and the text from right to left, however, is not possible, since we are not given the pattern and thus we do not know its length. Yet we can use the trick of enumerating its possible lengths, or again, more practically, we can be given its length as an extra static information. Partial evaluation then gives rise to a program representation of trees that are to Boyer and Moore what position trees are to Knuth, Morris and Pratt.

Getting back to the original motivation of this article, we would like to state two conclusions.

1. The vision that partial evaluation ought to be able to produce the KMP from a naive string matcher is due to Yoshihiko Futamura, who used it as a guiding light to develop Generalized Partial Computation. We observe that this vision also encompasses other linear string matchers.
2. Neil Jones envisioned that polyvariant specialization ought to be enough to implement a self-applicable partial evaluator. We observe that this vision also applies to the derivation of linear string matchers by partial evaluation.

```
(define (match-all-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          '()
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                (cons bt (loop-pc-1 bt lt 3))      ; collection
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
                [(< lt 3)
                 '()]
                [(equal? (string-ref t (+ bt 0)) #\a)
                 (loop-pt-1 bt lt)]
                [else
                 (loop-pc-1 bt lt 2)])))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
            [(< lt 3)
             '()]
            [(equal? (string-ref t (+ bt 0)) #\a)
             (loop-pt-1 bt lt)]
            [else
             (loop-pc-1 bt lt 1)]))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        '()
        (loop-pt-1 0 lt))))
```

**Fig. 4.** Match-all counterpart of Fig. 3

## 8 Postlude

This section is added in October 2002, as the Festschrift is finally going to press. Since this article was written, the connection between string matching and partial evaluation has been explored further:

– Futamura, Konishi, and Glück have continued to explore the connection between string matching and generalized partial computation, not only in the style of Knuth, Morris and Pratt but also in the style of Boyer and Moore [30, 31].
– Ager, Danvy, and Rohde have taken the next step and have formalized and characterized a class of KMP-like string matchers that are obtained by par-

tial evaluation [1]. They have found that a source matcher using only positive information precisely gives rise to Morris and Pratt's string matcher and that a source matcher using positive information and one character of negative information precisely gives rise to Knuth, Morris, and Pratt's string matcher. Using any further negative information gives rise to other string matchers.

We have also become aware of Charras and Lecroq's comprehensive handbook on exact string matching [18].

*Acknowledgments:* Each of the authors have benefited in one way or another from the research environment established by Neil Jones at DIKU and from his scientific approach to problem solving. Thanks are also due to Yoshihiko Futamura for proposing the initial problem, in 1987; to David Wise for his interest and encouragements, in 1989; and to Andrzej Filinski, Bernd Grobauer, Julia Lawall, and Henning Korsholm Rohde for comments. This article has also benefited from Torben Mogensen's proof-reading.

This work is supported by the NSF grant EIA-9806745 and by the ESPRIT Working Group APPSEM (`www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`). It was carried out while the first author was working at Boston University.

## References

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Chin [19], pages 32–46. Extended version available as the technical report BRICS-RS-02-32.

[2] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. The MIT Press, 1990.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[5] Maria Alpuente, Moreno Falaschi, Pascual Juliàn, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.

[6] Torben Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Department of Computer Science, University of Aarhus, 1993. Technical report PB-453.

[7] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Technical Report BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2001.

[8] Ricardo A. Baeza-Yates, Christian Choffrut, and Gaston H. Gonnet. On Boyer-Moore automata. *Algorithmica*, 12(4/5):268–292, 1994.

[9] Guntis J. Barzdins and Mikhail A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791, Computing Centre of Siberian Division of USSR Academy of Sciences, Novosibirsk, Siberia, 1988.

[10] Richard S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, November 1977.

[11] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[12] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[13] Anders Bondorf. Similix 5.1 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1993. Included in the Similix 5.1 distribution.

[14] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[15] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[16] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[17] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[18] Christian Charras and Thierry Lecroq. Exact string matching algorithms. `http://www-igm.univ-mlv.fr/~lecroq/string/`, 1997.

[19] Wei-Ngan Chin, editor. *ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Aizu, Japan, September 2002. ACM Press.

[20] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, 1999.

[21] Livio Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95:225–251, 1991.

[22] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[23] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[24] Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Unpublished manuscript, December 1989, and talks given at Stanford University, Indiana University, Kansas State University, Northeastern University, Harvard, Yale University, and INRIA Rocquencourt.

[25] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996. ACM Press.

[26] Max Crochemore and Christophe Hancart. Pattern matching in strings. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 11. CRC Press, Boca Raton, 1998.

[27] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[28] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[29] Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, New Generation Computing, Vol. 6, No. 2-3. Ohmsha Ltd. and Springer-Verlag, 1988.

[30] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Automatic generation of efficient string matching algorithms by generalized partial computation. In Chin [19], pages 1–8.

[31] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, 2002.

[32] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Bjørner et al. [11], pages 133–151.

[33] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 183–194, Toulouse, France, May 1994. IEEE Computer Society Press.

[34] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[35] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

[36] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[37] Manuel Hernández and David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001. ACM Press. To appear.

[38] Christoph M. Hoffman and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[39] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[41] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 124–140, Dijon, France, May 1985. Springer-Verlag.

[42] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[43] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[44] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 215–225. ACM Press, June 1996.

[45] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[46] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th International Workshop on Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

[47] Karoline Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1989.

[48] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.

[49] Karoline Malmkjær and Olivier Danvy. Preprocessing by program specialization. In Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory*, pages 266–268, Department of Computer Science, University of Aarhus, October 1994. BRICS NS-94-4.

[50] Jonathan Martin and Michael Leuschel. Sonic partial deduction. In Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, number 1755 in Lecture Notes in Computer Science, pages 101–112, Akademgorodok, Novosibirsk, Russia, July 1999. Springer-Verlag.

[51] Helmuth Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2(2):109–122, 1990.

[52] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 109–117, Bordeaux, France, September 1992. IRISA, Rennes, France.

[53] Robert Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM Journal on Computing*, 17(4):648–658, 1988.

[54] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 62–71, New Haven, Connecticut, June 1991. ACM Press.

[55] Morten Heine Sørensen. Turchin's supercompiler revisited. an operational theory of positive information propagation. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, April 1994. DIKU Rapport 94/17.

[56] Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, Scotland, April 1994. Springer-Verlag.

[57] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
[58] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.
[59] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm. *Journal of Information Processing*, 13(4):522–528, 1990.
[60] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.
[61] Peter Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, New York, 1973.

# WSDFU: Program Transformation System Based on Generalized Partial Computation

Yoshihiko Futamura[1], Zenjiro Konishi[1], and Robert Glück[2]

[1] Waseda University, 3-4-1 Okubo, Shinjuku, Tokyo 169-8555 Japan
{futamura,konishi}@futamura.info.waseda.ac.jp
[2] PRESTO, JST & Waseda University,
3-4-1 Okubo, Shinjuku, Tokyo 169-8555 Japan
glueck@acm.org

**Abstract.** Generalized Partial Computation (GPC) is a program transformation method utilizing partial information about input data and auxiliary functions as well as the logical structure of a source program. GPC uses both an inference engine such as a theorem prover and a classical partial evaluator to optimize programs. Therefore, GPC is more powerful than classical partial evaluators but harder to implement and control. We have implemented an experimental GPC system called WSDFU (Waseda Simplify-Distribute-Fold-Unfold). This paper demonstrates the power of the program transformation system as well as its theorem prover and discusses some future works.

## 1 Introduction

Generalized Partial Computation (GPC) is a program transformation method utilizing partial information about input data, abstract data types of auxiliary functions and the logical structure of a source program. The basic idea of GPC was reported at PEMC'87 [6]. GPC uses a theorem prover to solve branching conditions including unknown variables. It also uses a theorem prover to decide termination conditions of unfolding and conditions for correct folding. The prover uses domain information about variables, abstract data types of auxiliary functions (knowledge database) and logical structures of programs to prove properties about variables and subexpressions.

Differences between classical partial evaluation [5,10,17] and GPC were described in [7]. In [7], *GPC trees* were used to describe the algorithm of GPC in detail. Although GPC was patented both in the US [8] and in Japan, it has not been fully implemented. A key element of GPC is the implementation of an efficient theorem prover.

We have tuned up Chang and Lee's classical TPU [4] for our experimental GPC system called WSDFU (Waseda Simplify-Distribute-Fold-Unfold). This paper reports program transformation methods in WSDFU, the structures of the theorem prover, benchmark tests and future works. The kernel of the system has been implemented in Common Lisp and its size is about 1500 lines of pretty-printed source code (60KB). This paper is a revision of the main part of our paper [12].

## 2 Outline of WSDFU

At present, WSDFU can optimize strict functional programs with call-by-value semantics. It can perform automatic recursion introduction and removal [2] using the program transformation system formulated by Burstall and Darlington [3,21]. We also use an algebraic manipulation system [16] to simplify mathematical expressions and to infer the number of recursive calls in a program. GPC controls all environments including knowledge database, theorem prover, recursion removal and algebraic manipulation system (Fig. 7). Program transformation in WSDFU can be summarized as follows. It proceeds in four steps:

1. Simplify expressions (programs) as much as possible (S).
2. If there is a conditional expression in a program, then distribute the context function over the conditional (D).
3. Try to fold an expression to an adequate function call (F).
4. If there is a recursive call which does not satisfy the termination conditions (W-redex), then unfold the call (U).

WSDFU iterates SDFU operations in this order until there remains no W-redex in a program. The system maintains properties of variables in the form of predicates such as $integer(u) \wedge u > 0$ or $x = 71$. Therefore, WSDFU can do all the partial evaluation so called online partial evaluators can do and more. The set of the predicates is called *environment*. The environment is modified by WSDFU in processes S and D using a theorem prover. Details of the S, D, F and U operations are described below using GPC trees. A *source program* is a program to be optimized by GPC (using WSDFU, here). Every source program becomes a node of a GPC tree with a new attached name to its node. Moreover, the GPC tree represents the *residual program* (i.e. the result of the GPC) of the source program. We now describe each process in more detail.

### 2.1 Simplification

Simplify a program (expression) using a theorem prover, algebraic manipulation [15,16,20] and recursion removal system [9,21]. Here, we use a knowledge database including mathematical formulas and abstract data types of auxiliary functions. Current WSDFU system has about 40 rules to solve all the examples described in this report. A simplification process is successful if the result entails the elimination of all recursive calls (to the user defined source program) through folding. Since the folding process comes after the simplification process, this operation is non-deterministic. That is, if a folding process is unsuccessful, we backtrack our SDFU process to the simplification process that caused the folding and undo the simplification. Then try another simplification if possible. Examples of simplification are shown below:

1. Let the current environment be $i$ and a source program be a conditional expression such as **if** $p(u)$ **then** $e_1$ **else** $e_2$. If $p(u)$ is provable from $i$ then the residual program is the result of GPC of $e_1$ with repect to $i \wedge p(u)$. If

$\neg p(u)$ is provable from $i$ then the residual program is the result of GPC of $e_2$ with repect to $i \wedge \neg p(u)$. If neither of the above cases holds within a predetermined time, then the residual program is **if** $p(u)$ **then** (residual progam of GPC of $e_1$ with repect to $i \wedge p(u)$) **else** (residual program of GPC of $e_2$ with repect to $i \wedge \neg p(u)$).

2. Let the current environment be $i$ and a source program be a conditional expression such as **if0** $p(u)$ **then** $e_1$ **else** $e_2$. The meaning of **if0** is the same as **if** in (1) total evaluation or (2) partial evaluation and $p(u)$ is provable or refutable. However, when $p(u)$ is neither provable nor refutable, the residual program is *residual program of GPC of* $e_2$ *with respect to* $i$. Similar to **if0**, we use **if1** the residual program of which is *residual program of GPC of* $e_1$ *with respect to* $i$. Use of **if0** or **if1** changes the semantics of residual programs and the correctness of the program transformation cannot be guaranteed. However, some times it is very useful for getting efficient residual programs. An example usage of these expressions is shown in [12].
3. Let $mod(x, d)$ be a function computing the remainder of $x/d$. Then $mod(x * y, d)$ is replaced by $mod(mod(x, d) * mod(y, d), d)$. The purpose of this replacement is to move the *mod* function into the multiplication. This makes the folding of composite function including *mod* easier (See Example 3.5).
4. If we know, for example, from our database or the properties of $p(x)$ and $d(x)$ that function $f$ terminates and $a$ is a constant, then
   $f(x) \equiv$ **if** $p(x)$ **then** $a$ **else** $f(d(x))$
   is replaced by $a$.
5. If $f(a) = a$, then $f^m(a)$ is replaced by $a$ for $m > 0$.

## 2.2 Distribution

Assume that a source program $e$ contains a conditional expression such as **if** $p(u)$ **then** $e_1$ **else** $e_2$. Let $e$ be $Cont$[**if** $p(u)$ **then** $e_1$ **else** $e_2$] for a function $Cont$. Then the operation to produce a program **if** $p(u)$ **then** $Cont[e_1]$ **else** $Cont[e_2]$ from $e$ is called *distribution*. Since WSDFU deals with only strict programs, the distribution operation preserves semantics of programs. GPC of $e$ wrt $i$ produces a GPC tree shown in Fig. 1. In the figure, $N_1$, $N_2$ and $N_3$ are new names attached to nodes. They also stand for function names defined by GPC subtrees. For example, $N_1(u) \equiv$ **if** $p(u)$ **then** $N_2(u)$ **else** $N_3(u)$, $N_2(u) = Cont[e_1]$ and $N_3(u) = Cont[e_2]$. Environments of nodes $N_2$ and $N_3$ are $i \wedge p(u)$ and $i \wedge \neg p(u)$, respectively.

Assume that a source program $e$ does not contain a conditional expression. Let $e$ be $Cont[e_1]$ and $e_1'$ be the result of GPC of $e_1$ wrt $i$. Then GPC of $e$ wrt $i$ produces a GPC tree shown in Fig. 2. In the figure, the environment of node $N_2$ is $i$. This process can be considered as a part of the distribution.

## 2.3 Folding

First, in a program $e$, search for a sub-expression $g(k(u))$ that has an ancestor in the current GPC tree whose body is $g(u)$ and the range of $k(u)$ is a subset of

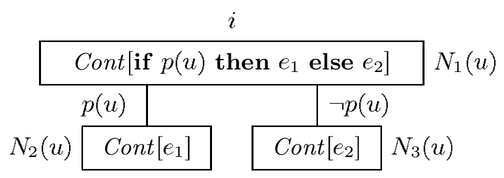


**Fig. 1.** GPC tree for distribution of a functional context over a conditional expression

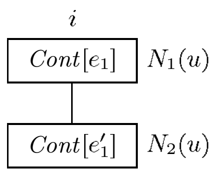


**Fig. 2.** GPC tree for distribution of a functional context over a simplified expression

the domain of $u$ (Here, $k$ is a primitive function. See Fig. 3). A theorem prover is used to check the inclusion which guarantees the correctness of our folding operation. Assume that the node name of $g(u)$ is $N$. Then $g(k(u))$ is replaced by $N(k(u))$.

If we find more than one $g(k(u))$'s in a predetermined time, we fold each of them one by one independently. Therefore, this process may produce more than one residual program. For example, we can choose one $g(k(u))$ which ends up with the shortest residual program. Since the shortest program is not necessarily the most efficient, users can choose the most efficient program from the residual programs if they want to have this fine level of control; otherwise a solution can be chosen automatically. We think the folding process is the most important operation in program transformation to obtain efficient residual programs [2].

## 2.4 Unfolding

In [7], we defined a *P-redex* as a procedure call with an actual parameter whose range is in the proper subset of the domain of the function. Among P-redexes, those who do not satisfy the termination condition (iii) or (iv) described later are called *W-redexes* (Waseda-redexes). Only W-redexes are unfolded in WSDFU. If there exists more than one W-redex in a program, we choose the leftmost-innermost W-redex first. This process can be conducted non-deterministically or in parallel like folding. Folding and unfolding are repeatedly applicable to parts of the GPC tree being processed by WSDFU.

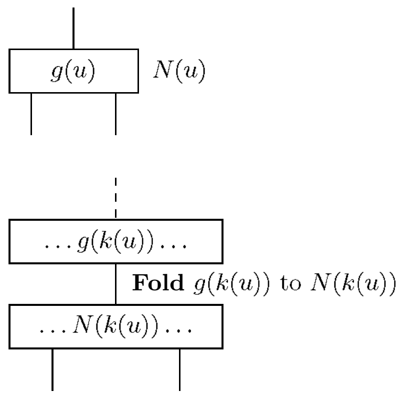


**Fig. 3.** Folding in a GPC tree

### 2.5 Example of GPC: 71-function

Here, we show the GPC of the 71-function [7] $f(u)$ wrt integer $u$ where

$f(u) \equiv$ **if** $u > 70$ **then** $u$ **else** $f(f(u+1))$.

Its GPC tree is shown in Fig. 4. Each $N_i$ in the figure is a label attached to each node. The $N_i$ stands for a function name defined by the GPC tree that has $N_i$ as a root. WSDFU terminates at leaves $N_5$ and $N_7$ because there is no recursive call at the nodes. WSDFU also terminates at node $N_4$ because the termination condition (i) described later applies. Since the range of $u+1$ in $N_2$ is integer, underlined $f$ can be folded to $N_1$. Note here that WSDFU unfold $N_1$ at $N_3$. Each edge of the tree has a predicate as a label. The conjunction of all predicates that appear on the path from the root to a node stands for the environment of variable $u$ in the node. The residual program defined by the GPC tree is:

$N_1(u) \equiv$ **if** $u > 70$ **then** $u$ **else** $N_3(u)$,
$N_3(u) \equiv$ **if** $u > 69$ **then** 71 **else**$f(N_3(u+1))$.

We eliminate recursion from $N_3$ utilizing our recursion removal system [19] and get a new node: $N_3(u) = f^{70-u}(71) = 71$. Then by simplification, $N_1$ is transformed to

$N_1(u) \equiv$ **if** $u > 70$ **then** $u$ **else** 71.

This is the final residual program for $f(u)$.

## 3 Termination Conditions

Since, in general the halting problem of computation is undecidable, termination conditions for WSDFU have to be heuristic. Here, we discuss halting problems

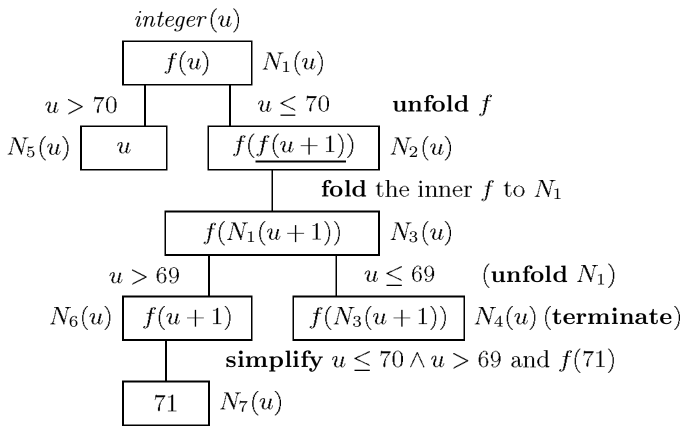


**Fig. 4.** GPC of 71-function

specific to GPC. Execution of programs based on partial evaluation has two phases [7,11,17], i.e. preprocessing (partial evaluation) and remaining processing (final evaluation). The first phase should evaluate as many portions of a source program as possible in order to save time during the final evaluation. However, it should avoid working on portions of the program not needed in the final evaluation in order to save the partial evaluation time. Moreover, it should terminate because it is a preprocessor. The termination can be guaranteed by setting up maximum execution time and residual program size. Errors concerning partial evaluation are:

1. Omission Error: We sometimes miss to evaluate a part of a source program that can be evaluated at the partial evaluation time in principle. This spoils the efficiency of a residual program.
2. Commission Error: We sometimes evaluate a part of a source program that is not evaluated in total computation. This causes a longer partial evaluation time and a larger residual program. This can also lead to transformation-time errors, such as **if** *too-long-to-prove-but-always-true-predicate* **then** $e$ **else** $1/0$ (division by zero).

Discussions here are not very formal but useful, we believe. Among the two errors above, the commission error is more dangerous. The termination conditions of unfolding a recursive call which we proposed in [7,23] tend to commit too much. They are:

(i) The range of the actual parameter $k(u)$ of a recursive call, e. g. $f(k(u))$, is the same as the one when $f$ was called before as a (direct or indirect) caller of this $f(k(u))$ in an ancestor node of a GPC tree.
(ii) The range of the actual parameter $k(u)$ of a recursive call, e. g. $f(k(u))$, is not included in the domain of $f$.

If condition (i) holds, then the GPC of the recursive call repeats the same computation as before. If condition (ii) holds, then the GPC of the recursive call does not make sense. It is not very difficult to see why we terminate GPC upon the two conditions.

### 3.1 New Termination Conditions

Here, we add two new termination conditions. Assume that function $f$ has $q$ parameters $x_1, \ldots, x_q$. Let the immediate caller of a recursive call $f(d_1)$ be $f(d_2)$ where $d_1$ and $d_2$ are expressions which do not include any recursive call to $f$. Let the ranges of $d_1$ and $d_2$ be $A = A'_1 \times \cdots \times A'_q$ and $B = B'_1 \times \cdots \times B'_q$, respectively. Note that each component of the product corresponds to the range of a variable $x_i$. Let the *recursion condition of* $f(d_2)$ *wrt* $f(d_1)$ stands for a condition in $f$ that caused the invocation of $f(d_1)$ from the GPC of $f(d_2)$. Let $x$ be a variable appearing in a simplified recursion condition of $f(d_2)$ wrt $f(d_1)$. Furthermore, let $A'$ be an infinite component of $A$ that is the range of variable $x$ and let $B'$ be corresponding component of $B$. If there exists $A'$ in $A$, then the two new termination conditions (iii) and (iv) are:

(iii) $A'$ is not a proper subset of $B'$.
(iv) Set difference $B' \setminus A'$ is finite.

Since the empty set is finite, conditions (i) and (iv) are not independent. Conditions (ii) and (iii) are not independent either because the domain of a function $f(u)$ and the range of its actual parameter $u$ is equal. Therefore, we check the conditions from (i) to (iv) in that order. These heuristics mean that the new domain of the recursive call is not small enough to become finite in the future. Of course, this is not always true. However, our experiments show that the new termination conditions prevent commission errors very well.

Here, we show five examples in order to look at the effectiveness of the new conditions. Without the new conditions, WSDFU would produce larger, but not more optimized residual programs for the examples. Especially for Example 3.2 to Example 3.4, it would not terminate and produced infinite residual programs.

*Example 3.1* Let $B$ be the set of all non-negative integers and $A$ be the set of all positive integers. Let

$$f(u) \equiv \textbf{if } u \leq 1 \textbf{ then } 1 \textbf{ else } f(u-1) + f(u-2).$$

Assume that $f$ is defined on $B$. Here, the range of $u - 1$ and $u - 2$ are $A$ and $B$ for $u > 1$, respectively. Then, $f(u - 2)$ satisfies condition (i). Since $A$ is infinite but the difference $B \setminus A$ is finite (i.e. $\{0\}$), then $f(u-1)$ satisfies (iv). Therefore, no unfolding occurs here and the residual program produced is the same as the source program. From the residual program, we can solve the recursion equation by an algebraic manipulation system [20] to get a closed form below. This gives us $O(\log(u))$ algorithm for $f(u)$.

$$f(u) = \frac{\phi^u - \hat{\phi}^u}{\sqrt{5}} \quad \text{where } \phi = \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

*Example 3.2* Let

$$f(m,n) \equiv \textbf{if } \mathit{mod}(m,n)=0 \textbf{ then } f(m/n,n+1)+1 \textbf{ else } 1$$

for non-negative integer $m$ and positive integer $n$. Let $D_m$ = {non-negative integers}. Now, we think about GPC of $f$ wrt $n=2$. Substituting $n$ by 2, we get a new function

$$f(m,2) \equiv \textbf{if } \mathit{mod}(m,2)=0 \textbf{ then } f(m/2,3)+1 \textbf{ else } 1.$$

Since variable $n$ does not appear in recursion condition $\mathit{mod}(m,2)$, we just consider $m/2$ on checking the termination of unfolding $f(m/2,3)$. Since $\mathit{mod}(m,2)=0$ for $f(m/2,3)$, the range of $m/2$ is $D_m$. Therefore, $f(m/2,3)$ satisfies condition (iv) and GPC stops unfolding (Here, condition (i) does not apply because $D_m \times \{2\}$ and $D_m \times \{3\}$ are not equal). The residual program produced here is the same as the source program. This means that WSDFU does not spoil the source.

*Example 3.3* Let

$$f(m,n) \equiv \textbf{if } m=0 \wedge n=n \textbf{ then } n \textbf{ else } f(m-1,3n)$$

for non-negative integers $m$ and $n$. Then the simplified recursion condition of $f(m,n)$ wrt $f(m-1,3n)$ is $m \neq 0$. Therefore, we just check the ranges of $m$ (i.e. non-negative integer) in $f(m,n)$ and $m-1$ (i.e. non-negative integer) in $f(m-1,3n)$. Then find that the termination condition (iv) holds for $f(m-1,3n)$. The residual program produced here is the same as the source program except that the recursion condition is simplified. The same as for Example 3.1, by using an algebraic manipulation system, we produce the final residual program $f(m,n) \equiv 3^m n$. This gives us $O(\log(m))$ algorithm for $f(m,n)$.

*Example 3.4* Let $h(u)$ be the Hailstorm function defined on positive integers:

```
$h(u) \equiv$
    **if** $u = 1$ **then** 1
    **else if** $odd(u)$ **then** $h(3u+1)$
    **else** $h(u/2)$.
```

Here, $h(u/2)$ satisfies the conditions (i) because $\mathit{range}(u/2)$ = {positive integers} for $u > 1 \wedge \mathit{even}(u)$. While, $h(3u+1)$ does not satisfy any of the termination conditions and the call is unfolded. Note here that $\mathit{range}(3u+1) = \{10, 16, 22, 28, \ldots\}$ and $\mathit{range}((3u+1)/2) = \{5, 8, 11, 14, \ldots\}$. Therefore, the call $h((3u+1)/2)$ satisfies (iii) and GPC terminates. See Fig. 5 for the complete GPC process. The residual program is

```
$N_1(u) \equiv$
    **if** $u = 1$ **then** 1
    **else if** $even(u)$ **then** $N_1(u/2)$
    **else** $N_1((3u+1)/2)$.
```

The residual program is a little bit more efficient than the source program.

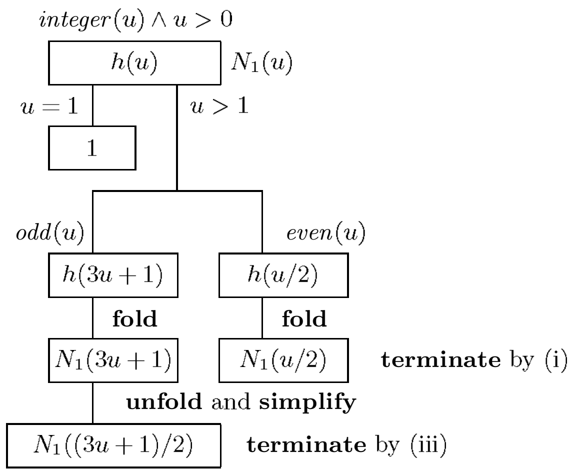


**Fig. 5.** GPC of Hailstorm function

*Example 3.5* Here, we think about GPC of $mod(exp(m, n), d)$ where

```
$exp(m, n) \equiv$
  **if** $n = 0$ **then** 1
  **else if** $odd(n)$ **then** $m * sqr(exp(m, (n-1)/2))$
  **else** $sqr(exp(m, n/2))$.
```

The drawback of the source program is to compute $exp(m, n)$ that cannot be tolerated when $m$ and $n$ are very large, say more than 100 digits each. Our residual program computes the result just using numbers less than $d^2$. The complete GPC process is shown in Fig. 6. We use the mathematical knowledge $mod(x * y, d) = mod(mod(x, d) * mod(y, d), d)$ in the transformation. The residual program is:

```
$N_1(m, n, d) \equiv$
  **if** $n = 0$ **then** 1
  **else if** $odd(n)$ **then** $mod(mod(m, d) * mod(sqr(N_1(m, (n-1)/2, d)), d), d)$
  **else** $mod(sqr(N_1(m, n/2, d)), d)$.
```

We think this program is almost optimal for our purpose. Note that if we do not have condition (iv), $N_1(m, n/2, d)$ in node $N_3$ is unfolded and the size of the residual program is more than doubled without any remarkable efficiency gain.

## 4 Theorem Prover

We have tuned up the classical TPU [4] theorem prover because it is powerful and the Lisp implementation is easy to modify. All goto's in the original program

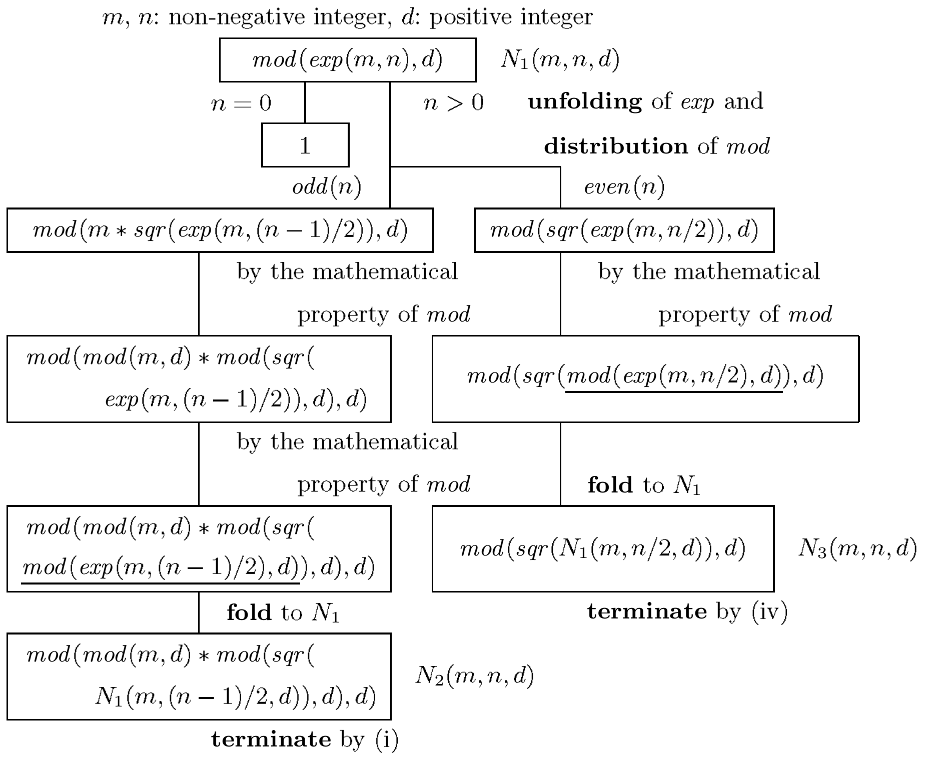

$m$, $n$: non-negative integer, $d$: positive integer
$mod(exp(m,n),d)$
$N_1(m,n,d)$
$n=0$
$n>0$
**unfolding** of $exp$ and
**distribution** of $mod$
$1$
$odd(n)$
$even(n)$
$mod(m*sqr(exp(m,(n-1)/2)),d)$
$mod(sqr(exp(m,n/2)),d)$
by the mathematical property of $mod$
by the mathematical property of $mod$
$mod(mod(m,d)*mod(sqr(exp(m,(n-1)/2)),d),d)$
$mod(sqr(\underline{mod(exp(m,n/2),d)}),d)$
by the mathematical property of $mod$
**fold** to $N_1$
$mod(mod(m,d)*mod(sqr(\underline{mod(exp(m,(n-1)/2),d)}),d),d)$
$mod(sqr(N_1(m,n/2,d)),d)$
$N_3(m,n,d)$
**fold** to $N_1$
**terminate** by (iv)
$mod(mod(m,d)*mod(sqr(N_1(m,(n-1)/2,d)),d),d)$
$N_2(m,n,d)$
**terminate** by (i)

**Fig. 6.** GPC of $mod(exp(m,n),d)$

have been removed because we want to partially evaluate TPU by GPC in the future. TPU is a uniform unit resolution theorem prover with set-of-support strategy, function depth test and subsumption test. We strengthened TPU with paramodulation so that theorems including equalities could be dealt with easily. Furthermore, term rewriting facility was added to the prover. Before submitting to the prover, theorems are transformed to simpler or shorter form by rewriting rules. For example, atomic formula $x+1>6$ is transformed to $x>5$.

The theorem prover plays the most important role in WSDFU (Fig. 7). The unfolding or folding of a recursive call terminates on the failure of proving a theorem (i.e., checking the termination or folding conditions) in a predetermined time [18,19]. This may cause omission errors but we'd be rather afraid of commission errors.

## 5 Benchmark Tests

This section describes 15 source programs and their residual programs produced automatically by WSDFU. All the residual programs are almost optimal, we

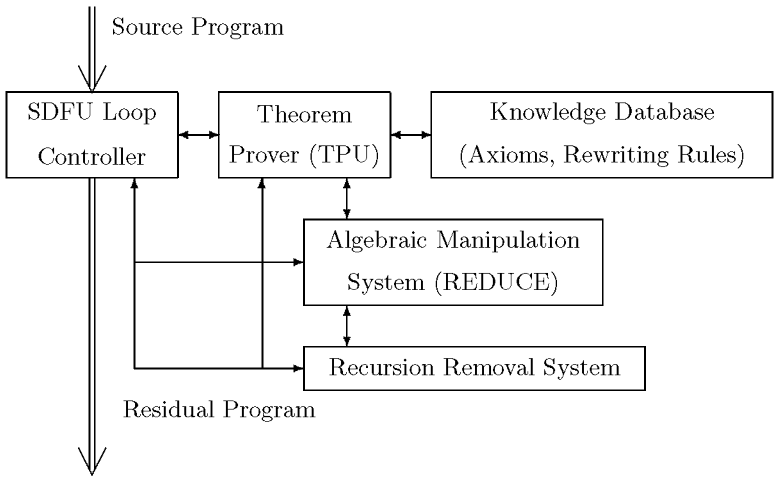


**Fig. 7.** Structure of WSDFU

believe. Every residual program was produced in less than 100 seconds on a notebook PC with Pentium II processor 366MHz and Allegro Common Lisp 5.0.1 for Windows 98. Although we have dealt with more test programs, those shown here are representatives and were chosen based on their simplicity and inefficiency. If a program is already optimal, it is impossible for WSDFU to produce a better one.

Here, we have three kinds of programs: In order to get almost optimal residual programs (1) WSDFU does not require any knowledge from users (5.1–5.10). (2) WSDFU needs some knowledge concerning source and subprograms from users (5.11–5.12). (3) WSDFU extensively uses an algebraic manipulation system and a recursion removal system (5.13–5.15).

Although, Lisp is the source language of the GPC system, here we use a more compact Pascal-like language to describe programs. Lists are written in Prolog style. For example, $[\,]$, $[a]$, $[a, b]$ and $[a, b|x]$ stand for *NIL*, $cons(a, NIL)$, $cons(a, cons(b, NIL))$ and $cons(a, cons(b, x))$, respectively.

## 5.1 Towers of Hanoi Problem (*mvhanoi*)

The source program $move(hanoi(n, a, b, c), n, m)$ is a naive version of computing the $m$-th move of $2^n - 1$ moves to solve $n$ disk Towers of Hanoi problem. Auxiliary functions *hanoi* and *move* are given below:

```
$hanoi(n, a, b, c) \equiv$
    **if** $n = 1$ **then** $[[a, c]]$
    **else** $[hanoi(n-1, a, c, b), [a, c]|hanoi(n-1, b, a, c)]$,
$move(s, n, m) \equiv$
    **if** $n = 1$ **then** $car(s)$
```

**Table 1.** Results of benchmark tests. The rightmost column shows the number of times that the theorem prover was called during GPC of a given program. The time shown is the overall transformation time used by GPC.

| # | Program | Time in sec. | TPU call |
|---|---|---|---|
| 5.1 | *mvhanoi* | 61.4 | 83 |
| 5.2 | *mvhanoi16* | 9.5 | 34 |
| 5.3 | *mvhanoi16a* | 8.8 | 32 |
| 5.4 | *mvhanoi3* | 43.5 | 103 |
| 5.5 | *mvhanoi3a* | 28.2 | 79 |
| 5.6 | *allonetwo* | 1.4 | 11 |
| 5.7 | *lengthcap* | 5.3 | 30 |
| 5.8 | *revapp1* | 2.9 | 22 |
| 5.9 | *revapp* | 6.0 | 40 |
| 5.10 | *matchaab* | 20.2 | 126 |
| 5.11 | *revrev* | 1.5 | 11 |
| 5.12 | *modexp* | 62.8 | 62 |
| 5.13 | *lastapp* | 4.4 | 31 |
| 5.14 | *f71* | 6.1 | 29 |
| 5.15 | *m91* | 94.0 | 305 |

```
    else if $m = 2^{n-1}$ then $car(cdr(s))$
    else if $m > 2^{n-1}$ then $move(cdr(cdr(s)), n-1, m-2^{n-1})$
    else $move(car(s), n-1, m)$.
```

The residual program is:

```
$N_1(n, m, a, b, c) \equiv$
    if $n = 1$ then $[a, c]$
    else if $m = 2^{n-1}$ then $[a, c]$
    else if $m > 2^{n-1}$ then $N_1(n-1, m-2^{n-1}, b, a, c)$
    else $N_1(n-1, m, a, c, b)$.
```

While the source is $O(2^n)$, the residual is $O(n)$ if we assume computation of $2^n$ costs $O(1)$.

## 5.2 Towers of Hanoi Problem (*mvhanoi16*)

We specialize $move(hanoi(n, a, b, c), n, m)$ wrt $m = 16$, here. The source program is $move(hanoi(n, a, b, c), n, 16)$. The residual is $N_1(n, a, b, c)$ below. The same as 5.1, while the source is $O(2^n)$, the residual is $O(n)$.

$N_1(n, a, b, c) \equiv$ **if** $5 = n$ **then** $[a, c]$ **else** $N_1(n-1, a, c, b)$.

## 5.3 Towers of Hanoi Problem (*mvhanoi16a*)

We want to perform the same specialization as 5.2 but we use the residual program of 5.1 instead of using naive *hanoi* and *move* programs. Therefore, the source program is $f(n, 16, a, b, c)$ where

```
$f(n, m, a, b, c) \equiv$
   **if** $n = 1$ **then** $[a, c]$
   **else if** $m = 2^{n-1}$ **then** $[a, c]$
   **else if** $m > 2^{n-1}$ **then** $f(n-1, m-2^{n-1}, b, a, c)$
   **else** $f(n-1, m, a, c, b)$.
```

The residual program is identical to that of 5.2 but produced in a shorter time.

## 5.4 Towers of Hanoi Problem (*mvhanoi3*)

We specialize *move*(*hanoi*($n, a, b, c$), $n, m$) wrt $n = 3$, here. The source program is *move*(*hanoi*($3, a, b, c$), $3, m$) and the residual program is $N_1(m, a, b, c)$ below. While the source is $O(2^3)$, the residual is $O(3)$.

```
$N_1(m, a, b, c) \equiv$
   **if** $m = 4$ **then** $[a, c]$
   **else if** $m > 4$ **then**
      **if** $m = 6$ **then** $[b, c]$
      **else if** $m > 6$ **then** $[a, c]$
      **else** $[b, a]$
   **else if** $m = 2$ **then** $[a, b]$
   **else if** $m > 2$ **then** $[c, b]$
   **else** $[a, c]$.
```

## 5.5 Towers of Hanoi Problem (*mvhanoi3a*)

We want to perform the same specialization as 5.4 but we use the residual program of 5.1 instead of using naive *hanoi* and *move* programs. The same as 5.3, we obtain the same residual program as 5.4 but in a shorter time.

## 5.6 Elimination of Intermediate Data (*allonetwo*)

Source program *allones*(*alltwos*($x$)) replaces every element of a given list by 2 and then replaces every 2 by 1. The residual program $N_1(x)$ directly replaces every element of a given list by 1. The efficiency gain here is that the residual does not produce an intermediate list consisting of 2'.

```
*allones*$(x) \equiv$ **if** *Null*$(x)$ **then** $[\,]$ **else** $[1|$*allones*$($*cdr*$(x))]$,
*alltwos*$(x) \equiv$ **if** *Null*$(x)$ **then** $[\,]$ **else** $[2|$*alltwos*$($*cdr*$(x))]$,
$N_1(x) \equiv$ **if** *Null*$(x)$ **then** $[\,]$ **else** $[1|N_1($*cdr*$(x))]$.
```

## 5.7 Elimination of Intermediate Data (*lengthcap*)

Source program *length*(*cap*($x, y$)) makes the intersection of two given lists $x$ and $y$ and then scan the intersection for its length. The residual program $N_1(x, y)$ does not make the intersection itself but just counts its length.

```
$length(x) \equiv$ **if** $Null(x)$ **then** $0$ **else** $1 + length(cdr(x))$,
$cap(x, y) \equiv$
    **if** $Null(x)$ **then** $[\,]$
    **else if** $Member(car(x), y)$ **then** $[car(x)|cap(cdr(x), y)]$
    **else** $cap(cdr(x), y)$,
$N_1(x, y) \equiv$
    **if** $Null(x)$ **then** $0$
    **else if** $Member(car(x), y)$ **then** $1 + N_1(cdr(x), y)$
    **else** $N_1(cdr(x), y)$.
```

## 5.8 Elimination of Intermediate Data (*revapp1*)

Source program $rev(app(x, [y]))$ appends a unit list $[y]$ to a given list $x$ then reverses the appended list. The residual program $N_1(x, y)$ produces the same results without building the intermediate appended list. Program *rev* is a naive reverse using append.

```
$rev(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $app(rev(cdr(x)), [car(x)])$,
$app(x, y) \equiv$ **if** $Null(x)$ **then** $y$ **else** $[car(x)|app(cdr(x), y)]$,
$N_1(x, y) \equiv$ **if** $Null(x)$ **then** $[y]$ **else** $app(N_1(cdr(x), y), [car(x)])$.
```

## 5.9 Elimination of Intermediate Data (*revapp*)

Source program $rev(app(x, y))$ appends a list $y$ to a list $x$ then reverse the appended list. The residual program $N_1(x, y)$ produces the same results without building the intermediate appended list.

```
$N_1(x, y) \equiv$ **if** $Null(x)$ **then** $N_2(x, y)$ **else** $app(N_1(cdr(x), y), [car(x)])$,
$N_2(x, y) \equiv$ **if** $Null(y)$ **then** $[\,]$ **else** $app(N_2(x, cdr(y)), [car(y)])$.
```

## 5.10 Pattern Matcher (*matchaab*)

Source program $matchaab(x)$ is a non-linear pattern matcher that check if there is pattern $[a, a, b]$ in a given text $x$. The residual program is a KMP-type linear pattern matcher. Note that the pattern matcher used here in the source program is as naive as the one in [6,14,22], but more naive than the one used in [1,7]. The generation of a Boyer-Moore type pattern matcher is discussed in [1,13].

```
$matchaab(x) \equiv f([a, a, b], x, [a, a, b], x)$,
$f(p, t, p_0, t_0) \equiv$
    **if** $Null(p)$ **then** *true*
    **else if** $Null(t)$ **then** *false*
    **else if** $car(p) = car(t)$ **then** $f(cdr(p), cdr(t), p_0, t_0)$
    **else if** $Null(t_0)$ **then** *false*
    **else** $f(p_0, cdr(t_0), p_0, cdr(t_0))$,
$N_1(x) \equiv$
  **if** $Null(x)$ **then** $false$
  **else if** $a = car(x)$ **then**
    **if** $Null(cdr(x))$ **then** $false$
    **else if** $a = cadr(x)$ **then** $N_8(x)$
    **else** $N_9(x)$
  **else** $N_5(x)$,
$N_5(x) \equiv N_1(cdr(x))$,
$N_8(x) \equiv$
  **if** $Null(cddr(x))$ **then** $false$
  **else if** $b = caddr(x)$ **then** $true$
  **else if** $a = caddr(x)$ **then** $N_8(cdr(x))$
  **else** $N_9(cdr(x))$,
$N_9(x) \equiv N_5(cdr(x))$.
```

### 5.11 List Reversal (*revrev*)

Source program $rev(rev(x))$ reverses a given list twice. The residual program $N_1(x)$ just copy a given list $x$. This time, we gave knowledge about *rev* and *append*, i.e. $rev(append(u, v)) = append(rev(v), rev(u))$ to WSDFU before starting GPC. The residual program $N_1$ below is equivalent to *copy* function.

$N_1(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $[car(x)|N_1(cdr(x))]$.

### 5.12 Example 3.5 in Section 3 (*modexp*)

See Example 3.5 in Section 3 for the problem, source and residual programs.

### 5.13 Last Element of an Appended List (*lastapp*)

Source program $last(app(x, [y]))$ appends a unit list $[y]$ to a given list $x$ and scan the appended list for the last element. The final residual program $N_1(x, y)$ directly returns $y$. This time we used a recursion removal system after getting a usual residual program to produce the final residual program.

```
$last(x) \equiv$ **if** $Null(cdr(x))$ **then** $car(x)$ **else** $last(cdr(x))$,
$N_1(x, y) \equiv$ **if** $Null(x)$ **then** $y$ **else** $N_1(cdr(x), y)$
  $\equiv y$   (by recursion removal [19]).
```

### 5.14 71-function (*f71*)

See 71-function example and Fig. 4 in Section 2 for the problem, source and residual programs.

### 5.15 McCarthy's 91-function (*m91*)

Source program $f(x)$ is McCarthy's 91-function that is a complex double recursive function. The final residual program $N_1(x)$ is $O(1)$. We used a recursion removal system after getting a usual residual program to produce the final one.

$f(x) \equiv$ **if** $x > 100$ **then** $x - 10$ **else** $f(f(x + 11))$.

First, GPC produces the following long residual program.

```
$N_1(x) \equiv$ if $x > 100$ then $x - 10$ else $N_3(x)$,
$N_3(x) \equiv$
   if $x > 89$ then
     if $x > 99$ then 91
     else if $x > 98$ then 91
     else if $x > 97$ then 91
     else if $x > 96$ then 91
     else if $x > 95$ then 91
     else if $x > 94$ then 91
     else if $x > 93$ then 91
     else if $x > 92$ then 91
     else if $x > 91$ then 91
     else if $x > 90$ then 91
     else 91
   else $f(N_3(x + 11))$
   $\equiv$ if $x > 89$ then 91 else $f(N_3(x + 11))$     (by simplification)
   $\equiv f^{(1+\lfloor(89-x)/11\rfloor)}(91)$     (by recursion removal [19])
   $\equiv 91$.     (by simplification)
```

Finally, the following residual program has been obtained.

$N_1(x) \equiv$ **if** $x > 100$ **then** $x - 10$ **else** 91.

## 6 Future Works

In the previous section, we have shown 15 GPC examples. Their residual programs are almost optimal. As shown in Table 1, transformation time is quite fast. However, we have not implemented such powerful program transformation methods as generalization of domains of functions, $\lambda$-abstraction and tupling. The methods are consistent with WSDFU and it is not a difficult task to build them in our current system. The rest of this section describes the idea of extending WSDFU to include generalization, $\lambda$-abstraction and tupling.

### 6.1 Generalization of Function Domain

As shown above, folding is one of the most useful operations in program transformation [2,3]. Without this operation, we will not produce an optimal residual

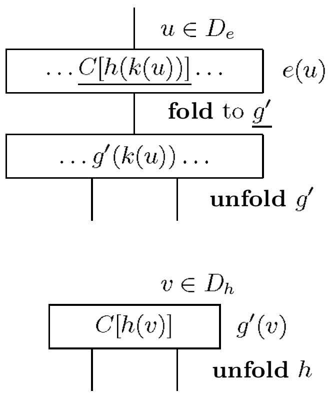


**Fig. 8.** Generalization and GPC Forest

program from a source program very often. A function is easier to be folded to if it has a larger domain. Therefore, when we define an auxiliary function (i.e. a new node in GPC tree) for residual programs, it is better for folding to extend the domain of the function as much as possible. Generalization in program transformation was first dicussed by Burstall and Darlington [3] and in partial evaluation by Turchin [24]. The generalization is discussed in detail in [21]. However, the extension of the domain is opposite to the specialization of programs. Therefore, we have to do both specialization and generalization in at the same time (or non-deterministically) to produce optimal residual programs. In our current implementation, the domain of a new function defined by WSDFU is given by the environment corresponding to the function. This domain is the most specialized one for the function in some sense. The outline of the modification to the current WSDFU to include a generalization strategy is described below.

Let $e$ be a node in a GPC tree, $e(u)$ be an expression in node $e$ and $D_e$ be the range of variable $u$ at $e$ (see Fig. 8).

First, we choose one subexpression $g(u)$ of $e(u)$ non-deterministically that includes $h(k(u))$ for a primitive function $k$ and a recursive function $h$. Let $g(u) = C[h(k(u))]$, $g'(v) = C[h(v)]$ and the domain of $h$ be $D_h$. Since $k(D_e) \subseteq D_h$, $g'$ is a generalized version of $g$ in some sense. Therefore, our generalization strategy is to perform GPC on $g'(v)$ instead of $g(u)$ (see Fig. 8). We add a GPC tree of $g'(v)$ to an existing GPC forest. If $g'(v)$ becomes a recursive function through a successful folding during its GPC process, generalization process is successful and $C[h(k(u))]$ in node $e$ is replaced by $g'(k(u))$. In this case, we call $g'$ the generalization of $g$. Otherwise, the generalization is unsuccessful and a GPC tree of $g'(v)$ is discarded.

## 6.2 Tupling and λ-abstraction

Tupling and $\lambda$-abstraction factorize similar computations in a source program and avoid to execute them more than once. The techniques were first used explicitly in program transformation in [3]. We can perform them in the simplification phase of WSDFU as follows.

Let $h_1(k_1(u))$ and $h_2(k_2(u))$ be procedure calls to recursive functions $h_1$ and $h_2$ in a program $e(u)$, and $k_1$ and $k_2$ be primitive functions. Moreover, assume that the domains of $h_1$ and $h_2$ are identical. Then we check the following two cases:

1. **$\lambda$-abstraction**: If $h_1 = h_2$ and $k_1 = k_2$, then replace all $h_i(k_i(u))$ in $e(u)$ by a fresh variable $y$ that does not appear in $e(u)$ and let the new program be $e'(u)$. Then replace $e(u)$ by $(\lambda y.e'(u))h_1(k_1(u))$. This corresponds to the insertion of **let**-expressions in Lisp.
2. **Tupling**: If $h_1 \neq h_2$ or $k_1 \neq k_2$, then let $h(u) \equiv \mathit{cons}(h_1(k_1(u)), h_2(k_2(u)))$ and check the following two cases:
   2.1 If $h(u)$ can be folded to any ancestor node, say $h'(u)$, then fold $h(u)$ to it (See node $N_5$ in Fig. 10).
   2.2 If we can find a generalization of $h(u)$, say $h'(v)$, in a predetermined time, then perform the following operation: replace all $h_1(k_1(u))$ and $h_2(k_2(u))$ by $\mathit{car}(y)$ and $\mathit{cdr}(y)$ for a fresh variable $y$ and let the new program be $e'(u)$. Note here that $h(u) = h'(k(u))$ for a primitive function $k$. Then replace $e(u)$ by $(\lambda y.e'(u))(h'(k(u)))$. If we cannot find a generalization of $h(u)$, then stop tupling (i.e. tupling is unsuccessful).

If the residual program of $h(u)$ itself is recursive, then $h'(u) \equiv h(u)$ in 2.2. Even when there are more than two functions involved, we can perform tupling the same way as above.

## 6.3 Example 6.1: List of Ascending Factorial Numbers

The source program $\mathit{afl}(n)$ lists factorial numbers from 0! to $n$!. The drawback of the program is to compute factorials $n+1$ times and costs $O(n^2)$ multiplications. The residual program $\mathit{new\_afl}(n)$ computes $n$! just once and costs $O(n)$ multiplications (see Fig. 9 and Fig. 10 for a complete GPC).

```
$\mathit{afl}(n) \equiv$ if $n = 0$ then $[1]$ else $\mathit{app}(\mathit{afl}(n-1), [\mathit{fact}(n)])$,
$\mathit{fact}(n) \equiv$ if $n = 0$ then $1$ else $n * \mathit{fact}(n-1)$,
$\mathit{new\_afl}(n) \equiv \mathit{cdr}(h'(n))$,
$h'(n) \equiv [\mathit{fact}(n)|\mathit{afl}(n)] \equiv$
    if $n = 0$ then $[1|[1]]$
    else $(\lambda y.((\lambda z.[z|\mathit{app}(\mathit{cdr}(y), [z])])(n * \mathit{car}(y))))(h'(n-1))$
```

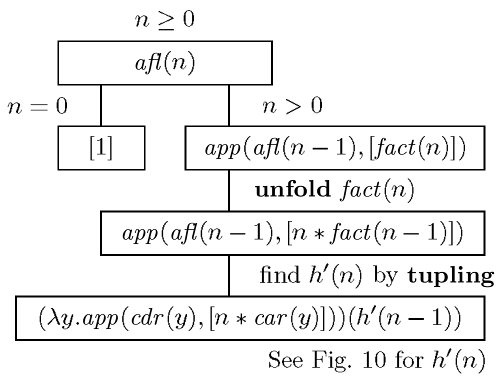

$n \geq 0$
$afl(n)$
$n = 0$
$n > 0$
$[1]$
$app(afl(n-1), [fact(n)])$
**unfold** $fact(n)$
$app(afl(n-1), [n * fact(n-1)])$
find $h'(n)$ by **tupling**
$(\lambda y.app(cdr(y), [n * car(y)]))(h'(n-1))$
See Fig. 10 for $h'(n)$

**Fig. 9.** Finding $h(n) \equiv [fact(n-1)|afl(n-1)]$ by tupling

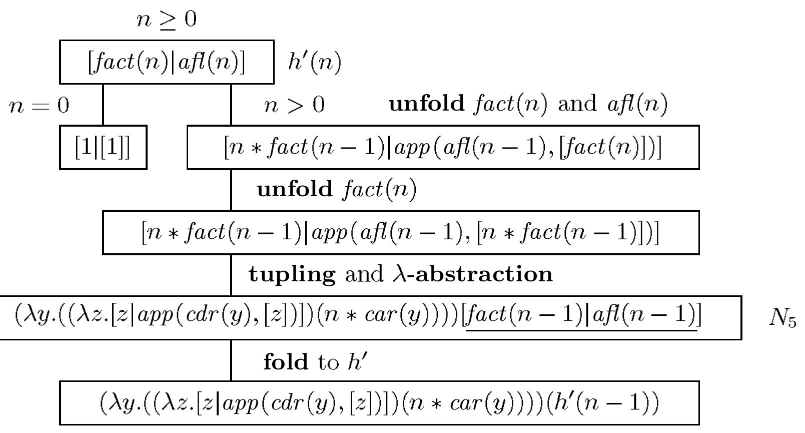

$n \geq 0$
$[fact(n)|afl(n)]$
$h'(n)$
$n = 0$
$n > 0$
**unfold** $fact(n)$ and $afl(n)$
$[1|[1]]$
$[n * fact(n-1)|app(afl(n-1), [fact(n)])]$
**unfold** $fact(n)$
$[n * fact(n-1)|app(afl(n-1), [n * fact(n-1)])]$
**tupling** and **$\lambda$-abstraction**
$(\lambda y.((\lambda z.[z|app(cdr(y), [z])])(n * car(y))))\underline{[fact(n-1)|afl(n-1)]}$
$N_5$
**fold** to $h'$
$(\lambda y.((\lambda z.[z|app(cdr(y), [z])])(n * car(y))))(h'(n-1))$

**Fig. 10.** GPC of $h'(n) \equiv [fact(n)|afl(n)]$

## 7 Conclusion

We have described GPC and WSDFU, a program transformation method using theorem proving combined with classical partial evaluation, and shown the results of a several benchmark tests using our present implementation. From the benchmark tests shown here, we can conclude that GPC is not far from being applicable to more practical problems. Also, we know how to strengthen WSDFU as described in Section 6.

We can say that our transformation method captures mathematical knowledge and formal reasoning. It allows to express source programs in a declarative, often inefficient style, and then to derive optimized versions exploiting that knowledge. As such, GPC can play an important role in different phases of software development.

A challenging problem is to implement a self-applicable GPC so that we can automatically generate generating extensions and compiler-compilers [10,11].

**Acknowledgments**

This research was partially supported by the Research for the Future Program of JSPS and New Technology Development Project. The authors are grateful to Torben Mogensen for valuable comments and discussions about the manuscript of this paper.

## References

1. Amtoft, T., Consel, C., Danvy, O., Malmkjaer, K.: The abstraction and instantiation of string-matching programs. BRICS RS-01-12, Department of Computer Science, University of Aarhus (2001)
2. Bird, R. S.: Improving programs by the introduction of recursion. Comm. ACM **20** (11) (1977) 856–863
3. Burstall, R. M., Darlington, J. A.: Transformation System for Developing Recursive Programs. JACM **24** (1) (1977) 44–67
4. Chang, C., Lee, R. C.: Symbolic Logic and Mechanical Theorem Proving. Academic Press (1973)
5. Danvy, O., Glück, R., Thiemann, P. (eds.): Partial Evaluation. Proceedings. Lecture Notes in Comp. Science **1110** Springer-Verlag (1996)
6. Futamura, Y., Nogi, K.: Generalized partial computation. in Bjørner, D., Ershov, A. P., Jones, N. D. (eds.): Partial Evaluation and Mixed Computation. North-Holland, Amsterdam (1988) 133–151
7. Futamura, Y., Nogi, K., Takano, A.: Essence of generalized partial computation. Theoretical Computer Science **90** (1991) 61–79
8. Futamura, Y., Nogi, K.: Program Transformation Based on Generalized Partial Computation. 5241678, US-Patent, Aug. 31, 1993
9. Futamura, Y., Otani, H.: Recursion removal rules for linear recursive programs and their effectiveness. Computer Software, JSSST **15** (3) (1998) 38–49 (in Japanese)
10. Futamura, Y.: Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. Higher-Order and Symbolic Computation **12** (4) (1999) 381–391
11. Futamura, Y.: Partial Evaluation of Computation Process Revisited. Higher-Order and Symbolic Computation **12** (4) (1999) 377–380
12. Futamura, Y., Konishi, Z., Glück, R.: Program Transformation System Based on Generalized Partial Computation. New Generation Computing **20** (1) (2001) 75–99
13. Futamura, Y., Konishi, Z., Glück, R.: Automatic Generation of Efficient String Matching Algorithms by Generalized Partial Computation. ASIA-PEPM 2002 (2002)

14. Glück, R., Klimov, A. V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., et al. (eds.): Static Analysis. Lecture Notes in Comp. Science **724** Springer-Verlag (1993) 112–123
15. Graham, R. L., Knuth, D. E., Patashnik, O.: Concrete Mathematics. Addison-Wesley (1989)
16. Hearn, A. C.: REDUCE — A Case Study in Algebra System Development. Lecture Notes in Comp. Science **144** Springer-Verlag, Berlin (1982)
17. Jones, N. D.: An Introduction to Partial Evaluation. ACM Computing Surveys **28** (3) (1996) 480–503
18. Konishi, Z., Futamura, Y.: A theorem proving system and a terminating process for Generalized Partial Computation (GPC). RIMS Workshop on Program Transformation, Symbolic Computation and Algebraic Manipulation, (1999) 59–64 (in Japanese)
19. Konishi, Z., Futamura, Y.: Recursion removal on generalized partial computation (GPC). Proc. of the 17th National Conference, C5-2, JSSST (2000) (in Japanese)
20. Matsuya, M., Futamura, Y.: Program transformation and algebraic manipulation. RIMS Workshop on Program Transformation, Symbolic Computation and Algebraic Manipulation, (1999) 115–122 (in Japanese)
21. Pettorossi, A., Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs. ACM Computing Surveys **28** (2) (1996) 360–414
22. Sørensen, M. H., Glück, R., Jones, N. D.: A positive supercompiler. Journal of Functional Programming **6** (6) (1996) 811–838
23. Takano, A., Nogi, K., Futamura, Y.: Termination Conditions of Generalized Partial Computation. Proceedings of the 7th National Conference, C8-1, JSSST (1990) (in Japanese)
24. Turchin, V. F.: The concept of a supercompiler. ACM TOPLAS **8** (3) (1986) 292–325

# Homeomorphic Embedding for Online Termination of Symbolic Methods

Michael Leuschel

Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ, UK
`mal@ecs.soton.ac.uk`

**Abstract.** Well-quasi orders in general, and homeomorphic embedding in particular, have gained popularity to ensure the termination of techniques for program analysis, specialisation, transformation, and verification. In this paper we survey and discuss this use of homeomorphic embedding and clarify the advantages of such an approach over one using well-founded orders. We also discuss various extensions of the homeomorphic embedding relation. We conclude with a study of homeomorphic embedding in the context of metaprogramming, presenting some new (positive and negative) results and open problems.

**Keywords:** Termination, Well-quasi orders, Program Analysis, Specialisation and Transformation, Logic Programming, Functional & Logic Programming, Metaprogramming, Infinite Model Checking.

## 1 Introduction

The problem of ensuring termination arises in many areas of computer science. It is especially important within all areas of automatic program analysis, synthesis, verification, specialisation, and transformation: one usually strives for methods which are guaranteed to terminate. It can also be an issue for other symbolic methods such as model checking, e.g., for infinite state systems. One can basically distinguish two kinds of techniques for guaranteeing termination:

- *offline* (or *static*) techniques (e.g., [12, 15, 16, 17, 64]), which *prove* termination of a program or process *beforehand* without any kind of execution, and
- *online* (or *dynamic*) techniques, which *ensure* termination of a process *during* its execution.

Offline approaches have less information at their disposal but do not require runtime intervention (which might be impossible). Which of the two approaches is taken depends entirely on the application area.

*Well-quasi orders* [10, 70] and in particular *homeomorphic embedding* [74, 53, 6, 2, 13] have become very popular to ensure online termination. In this paper we survey the now widespread use of these techniques and try to clarify (formally and informally) why these approaches have become so popular. We will also pay particular attention to the issue of metaprogramming.

## 2 Symbolic Methods and Online Termination

In this section we first introduce the general notion of a symbolic method which manipulates symbolic expressions.

**Symbolic Expressions** First, an *alphabet* consists of the following classes of symbols: 1) *variables*; and 2) *function symbols*. Function symbols have an associated *arity*, a natural number indicating how many arguments they take in the definitions below. *Constants* are function symbols with arity 0.

In the remainder, we suppose the set of variables is countably infinite and the set of function and predicate symbols is countable. In addition, alphabets with a finite set of function symbols will be called *finite*.

We will adhere as much as possible to the following syntactical conventions (stemming from logic programming [7, 56]):

- Variables will be denoted by upper-case letters like $X, Y, Z$, usually taken from the end of the (Latin) alphabet.
- Constants will be denoted by lower-case letters like $a, b, c$, usually taken from the beginning of the (Latin) alphabet.
- The other function symbols will be denoted by lower-case letters like $f, g, h$.

**Definition 1.** *The set of (symbolic)* expressions $\mathcal{E}$ *(over some given alphabet) is inductively defined as follows:*

- *a variable is an expression,*
- *a function symbol $f$ of arity $n \geq 0$ applied to a sequence $t_1, \ldots, t_n$ of $n$ expressions, denoted by $f(t_1, \ldots, t_n)$, is also an expression.*

For terms representing lists we will use the usual Prolog notation: e.g. $[]$ denotes the empty list, $[H|T]$ denotes a non-empty list with first element $H$ and tail $T$. As usual we can apply substitutions to expressions, and we will use the conventions of [7, 56].

**Symbolic Methods** Inspired by [67, 73], we now present a general definition of symbolic methods:

**Definition 2.** *A* configuration *is a finite tree whose nodes are labelled with expressions. Given two configurations $\tau_1, \tau_2$, we say that $\tau_1 \preceq \tau_2$ iff $\tau_2$ can be obtained from $\tau_1$ by attaching sub-trees to the leaves of $\tau_1$.*
*A* symbolic method *$sm$ is a map from configurations to configurations such that $\forall \tau$: $\tau \preceq sm(\tau)$. The symbolic method is said to* terminate *for some initial configuration $\tau_0$ iff for some $i \in \mathbb{N}$: $sm^i(\tau_0) = sm^{i+1}(\tau_0)$, where we define $sm^i$ inductively by $sm^0(\tau_0) = \tau_0$ and $sm^{i+1}(\tau_0) = sm(sm^i(\tau_0))$.*

This definition of a symbolic method is very general and naturally covers a wide range of interesting techniques, as shown in Fig. 1. For example, for partial evaluation [38, 36, 65], an expression is a partially specified call and the symbolic method will perform evaluation or unfolding. Another symbolic method is (conjunctive) partial deduction [57, 25, 48, 13, 49] of logic programs. Here the expressions are (conjunctions of) atoms and children are derived using resolution.[1] Other techniques are, e.g., supercompilation of functional programs [79, 30, 76, 75] and partial evaluation of functional logic programs [5, 6, 2, 42].

Also, a lot of program transformation techniques can be cast into the above form [66, 67]. Moreover, several algorithms for (infinite) model checking naturally follow the above form. Examples are the Karp-Miller procedure [40] for Petri nets, Finkel's minimal coverability procedure [18] for Petri nets, and also the backwards reachability algorithm for well-structured transition systems [1, 19]. Here, expressions in the tree are symbolic representations of sets of states of a system to be analysed and the symbolic method is either symbolically computing new successor or predecessor states. Probably, many other techniques (e.g., theorem proving) can be cast quite naturally into the above form.

| Symbolic Method | Expressions | $sm(.)$ |
|---|---|---|
| partial evaluation | terms with variables | evaluation + unfolding |
| supercompilation | terms with variables | driving |
| (conjunctive) partial deduction | conjunctions of atoms | resolution |
| Karp-Miller procedure | markings over $\mathbb{N} \cup \{\omega\}$ | firing transitions |

**Fig. 1.** Some symbolic methods

**Whistles and Online Termination** Quite often, symbolic methods do not terminate on their own, and we need some kind of supervisory process which spots potential non-termination (and then initiates some appropriate action). Formally, this supervision process can be defined as follows:

**Definition 3.** *A* whistle *$W$ is a subset of all configurations. The trees in $W$ are called* inadmissible *wrt $W$, the others are called* admissible *wrt $W$.*

Usually whistles are upwards-closed, i.e., $\tau_1 \preceq \tau_2 \wedge \tau_1 \in W \Rightarrow \tau_2 \in W$. In other words, if a whistle considers $\tau_1$ to be dangerous then it will also consider any extension of $\tau_1$ dangerous.

Now, a combination of a symbolic method $sm$ with a whistle $W$ is said to *terminate* iff either $sm$ terminates or for some $i$: $sm^i(\tau_0) \in W$.

Intuitively, we can view this as the whistle supervising the method $sm$ and "blowing" if it produces an inadmissible configuration ($sm^i(\tau_0) \in W$). What happens after the whistle blows depends on the particular application, and we will not go into great detail. In program specialisation or transformation one would usually generalise and restart. To ensure termination of the whole process one has to ensure that we cannot do an infinite number of restarts. This can basically be solved by viewing this generalisation and restart procedure as another

[1] One could argue that there are actually two symbolic methods: one relating to the so-called local control and the other to the global control.

symbolic method and applying the whistle approach to it as well; see [62, 73] for more details.

In a lot of cases, whistles are first defined on sequences of expressions, and then extended to configurations:

**Definition 4.** *A* whistle over sequences *$W$ is a subset of all finite sequences of expressions. The sequences in $W$ are called* inadmissible *wrt $W$.*

Given $W$, one will then often use the extension $W_c$ to configurations , defined as follows: $W_c = \{\tau \mid \exists$ a branch $\gamma$ of $\tau$ s.t. $\gamma \in W\}$. One can also use more refined extensions For example, one might focus on subsequences (see, e.g., the focus on selected literals and covering ancestors concept in [11, 61, 60, 59]).

In the remainder of the paper, we will mainly focus on whistles over sequences, and only occasionally discuss their extension to trees.

## 3 Subsumption, Depth Bounds, and Wfos

We now present some of the whistles used in early approaches to online termination.

**Subsumption and Variant Checking** The idea is to use subsumption testing to detect dangerous sequences. Formally, such a whistle is defined as follows: $W_{Sub} = \{e_1, e_2, \ldots \mid \exists i < j$ such that $e_i\theta = e_j\}$ In other words, a sequence is inadmissible if an element is an instance of a preceding expression. This approach was used, e.g., in early approaches to partial deduction (e.g., [78, 23, 9]). It fares pretty well on some simple examples, but, as shown in [11] (see also Ex. 1 below), it is not sufficient to ensure termination in the presence of accumulators. Sometimes variant testing is used instead of subsumption, making more sequences admissible but also worsening the non-termination problem.

**Depth Bounds** One, albeit ad-hoc, way to solve the local termination problem is to simply impose an arbitrary depth bound $D$ as follows: $W_D = \{e_1 e_2 \ldots e_k \ldots \mid k > D\}$. This approach is taken, e.g., in early partial deduction systems which unfold every predicate at most once. A depth bound is of course not motivated by any property, structural or otherwise, of the program or system under consideration, and is therefore both practically and theoretically unsatisfactory.

**Well-Founded Orders** A more refined approach to ensure termination is based on well-founded orders. Such an approach was first used for partial evaluation of functional programs in [69, 82] and for partial deduction in [11, 61, 60, 59]. These techniques ensure termination, while at the same time allowing symbolic manipulations related to the structural aspect of the program or system under consideration.

Formally, well-founded orders are defined as follows:

**Definition 5.** *A* strict partial order $<$ *is an irreflexive, transitive, and thus asymmetric, binary relation on* $\mathcal{E}$. *The whistle* $W_<$ *associated with* $<$ *is defined by* $W_< = \{e_1, e_2, \ldots \mid \exists i \text{ such that } e_{i+1} \not< e_i\}$. *We call* $<$ *a* well-founded order (wfo) *iff all infinite sequences of expressions are contained in* $W_<$.

In practice, wfos can be defined by so-called *norms*, which are mappings from expressions to $\mathbb{N}$ and thus induce an associated wfo.

*Example 1.* The following sequence of symbolic expressions arises when partially deducing the "reverse with accumulator" logic program [11]:

$$rev([a,b|T],[\,],R),\ rev([b|T],[a],R),\ rev(T,[b,a],R),\ rev(T',[H',b,a],R),\ \ldots$$

A simple well-founded order on expressions of the form $rev(t_1,t_2,t_3)$ might be based on comparing the *termsize* norm (i.e., the number of function and constant symbols) of the first argument. We then define the wfo for this example by:

$$rev(t_1,t_2,t_3) > rev(s_1,s_2,s_3) \text{ iff } termsize(t_1) > termsize(s_1).$$

Based on that wfo, the subsequence consisting of the first 3 expressions is admissible, but any further extension is not. At that point the partial deduction would stop and initiate a generalisation step.

In the above example we have fixed the wfo order beforehand. This is often quite difficult and (with the possible exception of [22] [63]) not very often done in practice. A more widely used approach for generating suitable wfos is to determine them while running the symbolic method. [11, 61, 60, 59], start off with a simple, but safe wfo and then refine this wfo during the unfolding process.

However, it has been felt by several researchers that well-founded orders are sometimes too rigid or (conceptually) too complex in an online setting. In the next section we introduce a more flexible approach which has gained widespread popularity to ensure online termination of symbolic techniques.

## 4 Wqos and Homeomorphic Embedding

Formally, well-quasi orders can be defined as follows.

**Definition 6.** *A* quasi order *is a reflexive and transitive binary relation on* $\mathcal{E}$.

Henceforth, we will use symbols like $<$, $>$ (possibly annotated by some subscript) to refer to strict partial orders and $\leq$, $\geq$ to refer to quasi orders and binary relations. We will use either "directionality" as is convenient in the context.

**Definition 7. (wbr,wqo)** *Let* $\leq$ *be a binary relation on* $\mathcal{E}$. *The whistle* $W_\leq$ *associated with* $\leq$ *is defined as follows* $W_\leq = \{e_1, e_2, \ldots \mid \exists i < j \text{ such that } e_i \leq e_j\}$. *We say that* $\leq$ *is a* well-binary relation (wbr) *iff all infinite sequences of expressions are contained in* $W_\leq$. *If* $\leq$ *is also a quasi order then* $\leq$ *is called a* well-quasi order (wqo).

Observe that, in contrast to wfos, non-comparable elements are allowed within admissible sequences. There are several other equivalent definitions of well-binary relations and well-quasi orders [33, 44, 81]. Traditionally, wqos have been used within *static* termination analysis to construct well-founded orders [15, 16]. The use of well-quasi orders in an *online* setting has only emerged quite recently. In that setting, transitivity of a wqo is usually not that interesting (because one does not have to generate wfos) and one can therefore drop this requirement, leading to the use of wbr's (see also Sect. 7).

An interesting wqo is the *homeomorphic embedding* relation $\trianglelefteq$. The following is the definition from [74], which adapts the pure[2] homeomorphic embedding from [16] by adding a simple treatment of variables.

**Definition 8.** *The* (pure) homeomorphic embedding *relation* $\trianglelefteq$ *on expressions is inductively defined as follows (i.e.* $\trianglelefteq$ *is the least relation satisfying the rules):*

1. $X \trianglelefteq Y$ *for all variables* $X, Y$
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ *if* $s \trianglelefteq t_i$ *for some* $i$
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ *if* $n \geq 0$ *and* $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

The second rule is sometimes called the *diving* rule, and the third rule is sometimes called the *coupling* rule (notice that $n$ is allowed to be 0 and we thus have $c \trianglelefteq c$ for all constants). When $s \trianglelefteq t$ we also say that $s$ is *embedded in* $t$ or $t$ is *embedding* $s$. By $s \triangleleft t$ we denote that $s \trianglelefteq t$ and $t \ntrianglelefteq s$.

The intuition behind the above definition is that $A \trianglelefteq B$ iff $A$ can be obtained from $B$ by removing some symbols, i.e., that the structure of $A$, split in parts, reappears within $B$. For instance, we have $p(a) \trianglelefteq p(f(a))$ because $p(a)$ can be obtained from $p(f(a))$ by removal of "$f(.)$" Observe that the removal corresponds to the application of rule 2 and that we also have $p(a) \triangleleft p(f(a))$. Another example is $f(a,b) \triangleleft p(f(g(a)), b)$ but $p(a) \ntrianglelefteq p(b)$ and $f(a,b) \ntrianglelefteq p(f(a), f(b))$. Finally, when adding variables, we have, e.g.: $p(X) \triangleleft p(f(Y))$, $p(X,X) \trianglelefteq p(X,Y)$, and $p(X,Y) \trianglelefteq p(X,X)$.

**Proposition 1.** *The relation* $\trianglelefteq$ *is a wqo on the set of expressions over a finite alphabet.*

In the presence of an infinite alphabet $\trianglelefteq$ is obviously not a wqo, (take, e.g, $0, 1, 2, \ldots$ where we then have $i \ntrianglelefteq j$ for $i \neq j$).

*Example 2.* Let us reconsider the sequence from Ex. 1:

$$rev([a,b|T],[\,],R),\ rev([b|T],[a],R),\ rev(T,[b,a],R),\ rev(T',[H',b,a],R),\ \ldots$$

The sequence consisting of the first three elements is admissible wrt $W_{\trianglelefteq}$, while the sequence consisting of the first four elements is not because $rev(T,[b,a],R) \trianglelefteq rev(T',[H',b,a],R)$. Hence, we have obtained the same power as the wfo in Ex. 1, without having to chose which arguments to measure and how to measure them. We further elaborate on the inherent flexibility of $\trianglelefteq$ in the next section.

[2] The full homeomorphic embedding makes use of an underlying wqo $\leq_F$ over the function symbols. The pure homeomorphic embedding uses equality for $\leq_F$, which is only a wqo for finite alphabets.

Also, $\trianglelefteq$ seems to have the desired property that often only "real" loops are detected and that they are detected at the earliest possible moment (see [58]). $\trianglelefteq$ also pinpoints where the dangerous growth has occurred; information which is vital for generalisation and restarting [74, 52, 53, 43].

Looking at Def. 8 one might think that the complexity of checking $s \trianglelefteq t$ for two expressions $s$ and $t$ is exponential. However, the complexity is actually linear [77, 31] (see also [58]); more precisely it is proportional to the size of $s$ and $t$ and to the maximum arity used within $s$ and $t$.

**History** $\trianglelefteq$ was first defined over strings by Higman [33] and later extended by Kruskal [41] to ordered trees (and thus symbolic expressions). Since then, $\trianglelefteq$ has been used for many applications. Arguably, the heaviest use of $\trianglelefteq$ within computer science was made in the context of term rewriting systems [15, 16], to automatically derive wfos for static termination analysis. The usefulness of $\trianglelefteq$ as a whistle for partial evaluation was probably first discovered and advocated in [58]. It was then later, independently, rediscovered and adapted for supercompilation by Sørensen and Glück in [74]. Neil Jones played an important role in that re-discovery. Indeed, Neil was tidying his collection of articles and among those articles was a worn copy of [15]. Neil brought this article, containing a definition of homeomorphic embedding, to the attention of Morten Sørensen. Morten immediately realised that this was what he was looking for to ensure termination of supercompilation in an elegant, principled way.

In autumn 1995, Bern Martens and myself were working on the problem of ensuring termination of partial deduction with characteristic trees [26]. Up to then only ad hoc approaches using depth bounds existed. Luckily, Bern Martens visited Neil's group at DIKU in Copenhagen, and came into contact with $\trianglelefteq$. Upon his return, we realised that $\trianglelefteq$ was also exactly the tool we needed to solve our problem, leading to [52, 53]. Later came the realisation that $\trianglelefteq$ provided a mathematically simpler and still more powerful way than wfos of ensuring termination of partial deduction in general, which lead to the development of [47], written during a stay at DIKU.

## 5 On the Power of Homeomorphic Embedding

It follows from Definitions 5 and 7 that if $\leq$ is a wqo then $<$ (defined by $e_1 < e_2$ iff $e_1 \leq e_2 \wedge e_1 \not\geq e_2$) is a wfo, but not vice versa. However, if $<$ is a well-founded order then $\preceq$, defined by $e_1 \preceq e_2$ iff $e_1 \not> e_2$, is a wbr. Furthermore, $<$ and $\preceq$ have the same set of admissible sequences. This means that, in an online setting, the approach based upon wbr's is in theory at least as powerful as the one based upon wfos. Let us now examine the power of $\trianglelefteq$ in more detail.

Let us examine the power of $\trianglelefteq$ on a few examples. For instance, $\trianglelefteq$ will admit all sequences in Fig. 2 (where, amongst others, Ex. 1 is progressively wrapped into so-called metainterpreters *solve* and *solve'*, counting resolution steps and keeping track of the selected predicates respectively):

| Sequence |
|---|
| $rev([a,b\|T],[\,],R) \rightsquigarrow rev([b\|T],[a],R)$ |
| $solve(rev([a,b\|T],[\,],R),0) \rightsquigarrow solve(rev([b\|T],[a],R),s(0))$ |
| $solve'(solve(rev([a,b\|T],[\,],R),0),[\,]) \rightsquigarrow solve'(solve(rev([b\|T],[a],R),s(0)),[rev])$ |
| $path(a,b,[\,]) \rightsquigarrow path(b,a,[a])$ |
| $path(b,a,[\,]) \rightsquigarrow path(a,b,[b])$ |
| $solve'(solve(path(a,b,[\,]),0),[\,]) \rightsquigarrow solve'(solve(path(b,a,[a]),s(0)),[path])$ |
| $solve'(solve(path(b,a,[\,]),0),[\,]) \rightsquigarrow solve'(solve(path(a,b,[b]),s(0)),[path])$ |

**Fig. 2.** Sequences admissible wrt $W_{\trianglelefteq}$

Achieving the above is very difficult for wfos and requires refined and involved techniques (of which to our knowledge no implementation in the online setting exists). For example, to admit the third sequence we have to measure something like the "termsize of the first argument of the first argument of the first argument." For the sequences 6 and 7, things are even more involved. We will return to the particular issue of metaprogramming in more detail in Sect. 8.

The above examples highlight the flexibility of $\trianglelefteq$ compared to wfos. But can one prove some "hard" results? It turns out that one *can* establish that — in the online setting — $\trianglelefteq$ is strictly more generous than a large class of refined wfos.

**Monotonic Wfos and Simplification Orderings** [47] establishes that $\trianglelefteq$ is strictly more powerful than the class of so-called *monotonic* wfos. In essence, a monotonic wfo $\succ$ has the property that if it considers the sequence $s_1, s_2$ to be inadmissible (i.e., $s_1 \not\succ s_2$) then it will also reject sequences such as $s_1, f(s_2)$ and $f(s_1), f(s_2)$. This is a quite natural requirement, which actually most wfos used in online practice satisfy. For instance, the wfo induced by the termsize norm is monotonic. Also, any linear norm induces a monotonic wfo [47]. Almost all of the refined wfos defined in [11, 61, 60, 59] are monotonic.[3]

Formally, monotonic wfos are defined as follows:

**Definition 9.** *A well-founded order $\prec$ on expressions is said to be* monotonic *iff the following rules hold:*

1. *$X \not\succ Y$ for all variables $X, Y$,*
2. *$s \not\succ f(t_1, \ldots, t_n)$ whenever $f$ is a function symbol and $s \not\succ t_i$ for some $i$ and*
3. *$f(s_1, \ldots, s_n) \not\succ f(t_1, \ldots, t_n)$ whenever $\forall i \in \{1, \ldots, n\} : s_i \not\succ t_i$.*

Note the similarity of structure with the definition of $\trianglelefteq$ (but, contrary to $\trianglelefteq$, $\not\succ$ does not have to be the least relation satisfying the rules), which means that $s \trianglelefteq t \Rightarrow s \not\succ t$.

Another interesting class of wfos are the so called simplification orderings, which we adapt from term rewriting systems (to cater for variables). It will turn out that the power of this class is also subsumed by $\trianglelefteq$.

[3] The only non-monotonic wfo in that collection of articles is the one devised for metainterpreters in Definition 3.4 of [11] (also in Section 8.6 of [59]) which allows to focus on subterms. We return to this approach below.

**Definition 10.** *A* simplification ordering *is a wfo $\prec$ which satisfies*

1. $s \prec t \Rightarrow f(t_1, \ldots, s, \ldots, t_n) \prec f(t_1, \ldots, t, \ldots, t_n)$ (replacement property),
2. $s \prec f(t_1, \ldots, s, \ldots, t_n)$ (subterm property) *and*
3. $s \prec t \Rightarrow s\sigma \prec t\gamma$ *for all variable only substitutions* $\sigma$ *and* $\gamma$ (invariance under variable replacement).

The third rule of the above definition is new wrt term-rewriting systems and implies that all variables must be treated like a unique new constant. It turns out that a lot of powerful wfos are simplification orderings [15, 64]: recursive path ordering, Knuth-Bendix ordering or lexicographic path ordering, to name just a few.

Observe that Def. 10 is also very similar to Def. 8 of $\trianglelefteq$. Indeed, for variable-free expressions we have that $s \trianglelefteq t$ implies $s \preceq t$ for all $\prec$ satisfying Def. 10 (by the Embedding Lemma from [14]). The following theorem is established in [47]. Transitivity of $\prec$ is required in the proof and Theorem 1 does not hold for well-founded relations. For simplification orderings on variable-free expressions, this theorem is a direct consequence of the Embedding Lemma from [14].

**Theorem 1.** *Let $\prec$ be a wfo on expressions which is either monotonic or a simplification ordering (or both). Then any admissible sequence wrt $\prec$ is also admissible wrt $\trianglelefteq$.*

This theorem implies that, no matter how much refinement we put into an approach based upon monotonic wfos or upon simplification orderings, we can only expect to approach $\trianglelefteq$ in the limit. But by a simple example we can even dispel that hope.

*Example 3.* Take the sequence $\delta = f(a), f(b), b, a$. This sequence is admissible wrt $\trianglelefteq$ as $f(a) \ntrianglelefteq f(b)$, $f(a) \ntrianglelefteq b$, $f(a) \ntrianglelefteq a$, $f(b) \ntrianglelefteq b$, $f(b) \ntrianglelefteq a$ and $a \ntrianglelefteq b$. However, there is no monotonic wfo $\prec$ which admits this sequence. More precisely, to admit $\delta$ we must have $f(a) \succ f(b)$ as well as $b \succ a$, i.e. $a \nsucc b$. Hence $\prec$ cannot be monotonic. This also violates rule 1 of Def. 10 and $\prec$ cannot be a simplification ordering.

**Non-monotonic Wfos** There are natural wfos which are neither simplification orderings nor monotonic. For such wfos, there can be sequences which are not admissible wrt $W_{\trianglelefteq}$ but which are admissible wrt the wfo. Indeed, $\trianglelefteq$ takes the whole term structure into account while wfos in general can ignore part of the term structure. For example, the sequence $[1, 2], [[1, 2]]$ containing two expressions, is admissible wrt the "listlength" measure but not wrt $\trianglelefteq$, where "listlength" measures a term as 0 if it is not a list and by the number of elements in the list if it is a list [60]. For that same reason the wfos for metainterpreters defined in Definition 3.4 of [11] are not monotonic, as they can focus on certain subterms, fully ignoring other subterms. It will require further work to automate that approach and to compare it with wqo-based approaches, both in theory and in practice.

Still there are some feats of $\trianglelefteq$ which *cannot* be achieved by a wfo approach (monotonic or not). Take the sequences $S_1 = p([\,],[a]), p([a],[\,])$ and $S_2 = p([a],[\,]), p([\,],[a])$. Both of these sequences are admissible wrt $\trianglelefteq$ but there exists *no* wfo which will admit *both* these sequences. By using a dynamic adjustment of wfos [11] it is possible to admit both sequences. However, for more complicated examples (e.g., from Fig. 2) the dynamic adjustment has to be very refined and one runs into the problem that infinitely many dynamic refinements might exist [60, 59], and to our knowledge no satisfactory solutions exists as of yet.

Finally, the above example also illustrates why, when using a wqo, one has to compare with every predecessor state of a process, whereas when using a wfo one has to compare only to the last predecessor. Hence, the online use of wqo is inherently more complex than the use of wfos.

## 6 Homeomorphic Embedding in Practice

In this section we survey the (now widespread) use of $\trianglelefteq$ for online termination, since [58, 74]. Works explicitly addressing metaprogramming will be discussed in Sect. 8.

**Functional Programming** As already mentioned, the first fully worked out online use of $\trianglelefteq$ was within *supercompilation* [79, 30]. [74, 76] presented, for the first time, a fully formal definition of positive supercompilation, for a first-order functional language with a lazy semantics. $\trianglelefteq$ was applied on expressions with variables and used to guide the generalisation and ensuring the construction of finite (partial) process trees. The approach was then later generalised in [71] to cover negative supercompilation, where negative constraints are propagated. [75] presents a largely language independent framework for proving termination of program transformers, where $\trianglelefteq$ is one of the possible mechanisms.

Recently, [72] uses $\trianglelefteq$ in the context of *generalized partial computation* (GPC) [24] and presents a refined termination criterion. The main idea consists in measuring the distance between the current expression and base cases for recursion (which have to be identifiable). Homeomorphic embedding is then applied to this sequence of distances, which results in a powerful termination condition, whose effectiveness in practice still needs to be evaluated.

**Logic Programming** First use of $\trianglelefteq$ for *partial deduction* of logic programs occurred in [52, 53]. Here, $\trianglelefteq$ was not only used on selected literals (to ensure termination of the so-called local control), but also on the atoms in the so-called global tree, as well as on characteristic trees. The latter characterise the computational behaviour of atoms to be specialised, which is often a better basis for controlling polyvariance than the syntactic structure of expressions.

In [29, 39, 13] $\trianglelefteq$ (and the generalisation process) was then extended to cope with conjunctions of atoms, to provide a terminating procedure for *conjunctive partial deduction*.

All of the above led to the development of the ECCE partial deduction system, which can achieve both deforestation and tupling [48]. Moreover, the refined way in which $\trianglelefteq$ ensures termination opened up new application areas, such as *infinite model checking* [54]. For some particular applications, such as coverability analysis of Petri nets, $\trianglelefteq$ is "fully precise," in the sense that it whistles *only* when a real infinite sequence is being constructed. As shown in [51, 50], one then obtains a decision procedure for these problems and one can establish some relatively surprising links with existing model checking algorithms such as the Karp-Miller procedure [40] or more recent techniques such as [18] and [1, 19].

In some cases, although $\trianglelefteq$ is fully precise, the associated generalisation operation of partial deduction is not. Hence, [43] defines a new generalisation operator which extrapolates the growth detected by $\trianglelefteq$. This enables to solve some new problems, such as the conjunctive planning problem for the fluent calculus.

In another line of work, one might use $\trianglelefteq$ as the basis for specialising and transforming *constraint logic programs*. First steps in that direction have been presented in [20, 21], where a wqo for constrained goals is developed and a generic algorithm is developed.

**Functional Logic Programming** Functional logic programming [32] extends both logic and functional programming. A lot of work has recently been carried out on partial evaluation of such programs [5, 4, 2, 6, 3], where $\trianglelefteq$ is used to ensure termination. This work has resulted in the INDY partial evaluation system, which has been successfully applied to a wide range of examples.

In another line of work, [42] has adapted $\trianglelefteq$ for constraint-based partial evaluation of functional logic programs.

## 7 Extensions of the Homemorphic Embedding

While $\trianglelefteq$ has a lot of desirable properties it still suffers from some drawbacks. First, the homeomorphic embedding relation $\trianglelefteq$ as defined in Def. 8 is rather unsophisticated when it comes to variables. In fact, all variables are treated as if they were identical, a practice which is often undesirable (namely when the same variable can appear multiple times within the same expression). Intuitively, $p(X,Y) \trianglelefteq p(X,X)$ could be justified, while $p(X,X) \trianglelefteq p(X,Y)$ can not. Indeed $p(X,X)$ could be seen as representing something like $and(p(X,Y), eq(X,Y))$, which embeds $p(X,Y)$, but not the other way around. Second, $\trianglelefteq$ behaves in quite unexpected ways in the context of generalisation, posing some subtle problems wrt the termination of a generalisation process [53, 46].

**Strict Homeomorphic Embedding $\trianglelefteq^+$** To remedy these problems, [53] introduced the so called strict homeomorphic embedding, which was then taken up, e.g., by [29, 42, 13]. The definition is as follows:

**Definition 11.** *Let $A, B$ be expressions. Then $B$* (strictly homeomorphically) embeds *$A$, written as $A \trianglelefteq^+ B$, iff $A \trianglelefteq B$ and $A$ is not a strict instance of $B$.*[4]

*Example 4.* We now still have that $p(X, Y) \trianglelefteq^+ p(X, X)$ but not $p(X, X) \trianglelefteq^+ p(X, Y)$. Note that still $X \trianglelefteq^+ Y$ and $X \trianglelefteq^+ X$.

The following is proven in [53].

**Theorem 2.** *The relation $\trianglelefteq^+$ is a wbr on the set of expressions over a finite alphabet.*

Unfortunately, $\trianglelefteq^+$ is not a wqo as it is not transitive. For example, we have

- $p(X, X, Y, Y) \trianglelefteq^+ p(X, Z, Z, X)$ and $p(X, Z, Z, X) \trianglelefteq^+ p(X, X, Y, Z)$
- but $p(X, X, Y, Y) \not\trianglelefteq^+ p(X, X, Y, Z)$.

One might still feel dissatisfied with $\trianglelefteq^+$ for another reason. Indeed, although going from $p(X)$ to $p(f(X))$ looks very dangerous, a transition from $p(X, Y)$ to $p(X, X)$ is actually not dangerous, as there are only finitely many new variable links that can be created. To remedy this, [46] develops the following refinement of $\trianglelefteq^+$, which is useful, for example, in the context of Datalog programs (logic programs who operate on constants only).

**Definition 12.** *We define $s \trianglelefteq_{var} t$ iff $s \lhd t$ or $s$ is a variant of $t$.*

It is obvious that $\trianglelefteq_{var}$ is strictly more powerful than $\trianglelefteq^+$ (if $t$ is strictly more general than $s$, then it is not a variant of $s$ and it is also not possible to have $s \lhd t$). For example, we have $p(X) \trianglelefteq_{var} p(f(X))$ as well as $p(X, Y) \trianglelefteq_{var} p(Z, X)$ but $p(X, X) \not\trianglelefteq_{var} p(X, Y)$ and $p(X, Y) \not\trianglelefteq_{var} p(X, X)$.

**Theorem 3.** *The relation $\trianglelefteq_{var}$ is a wqo on the set of expression over a finite alphabet.*

**The Extended Homeomorphic Embedding $\trianglelefteq^*$** Although $\trianglelefteq^+$ has a more refined treatment of variables than $\trianglelefteq$, it is still somewhat unsatisfactory. One point is the restriction to a finite alphabet. Indeed, for a lot of practical programs, using, e.g., arithmetic built-ins, a finite alphabet is no longer sufficient. Luckily, the fully general definition of homeomorphic embedding [41, 16] remedies this aspect. It also allows function symbols with variable arity (which can also be seen as associative operators). We will show below how this definition can be adapted to incorporate a more refined treatment of variables.

However, there is another unsatisfactory aspect of $\trianglelefteq^+$ (and $\trianglelefteq_{var}$). Indeed, we have $p(X, X) \not\trianglelefteq^+ p(X, Y)$ and $p(X, X) \trianglelefteq p(X, Y)$ as expected, but we have, e.g., $f(a, p(X, X)) \trianglelefteq^+ f(f(a), p(X, Y))$, which is rather unexpected. In other words, the more refined treatment of variables is only performed at the root of expressions, but not recursively within the structure of the expressions.

The following, new and more refined embedding relation remedies this somewhat ad hoc aspect of $\trianglelefteq^+$ and adds support for infinite alphabets.

[4] $A$ is a strict instance of $B$ iff there exists a substitution $\gamma$ such that $A = B\gamma$ and there exists no substitution $\sigma$ such that $B = A\sigma$.

**Definition 13.** *Given a wbr $\preceq_F$ on the function symbols and a wbr $\preceq_S$ on sequences of expressions, we define the* extended homeomorphic embedding *on expressions by the following rules:*

1. $X \trianglelefteq^* Y$ *if $X$ and $Y$ are variables*
2. $s \trianglelefteq^* f(t_1, \ldots, t_n)$ *if $s \trianglelefteq^* t_i$ for some $i$*
3. $f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n)$ *if $f \preceq_F g$ and $\exists 1 \leq i_1 < \ldots < i_m \leq n$ such that $\forall j \in \{1, \ldots, m\} : s_j \trianglelefteq^* t_{i_j}$ and $\langle s_1, \ldots, s_m \rangle \preceq_S \langle t_1, \ldots, t_n \rangle$*

Observe that for rule 3 both $n$ and $m$ are allowed to be 0, but we must have $m \leq n$. In contrast to Def. 8 for $\trianglelefteq$, the left- and right-hand terms in rule 3 do not have to be of the same arity. The above rule therefore allows to ignore $n-m$ arguments form the right-hand term (by selecting the $m$ indices $i_1 < \ldots < i_m$).

Furthermore, the left- and right-hand terms in rule 3 do not have to use the same function symbol: the function symbols are therefore compared using the wbr $\preceq_F$. If we have a finite alphabet, then equality is a wqo on the function symbols (one can thus obtain the pure homeomorphic embedding as a special case). In the context of, e.g., program specialisation or analysis, we know that the function symbols occurring within the program (text) and call to be analysed are of finite number. One might call these symbols *static* and all others *dynamic.* A wqo can then be obtained by defining $f \preceq g$ if either $f$ and $g$ are dynamic or if $f = g$. For particular types of symbols a natural wqo or wbr exists (e.g., for numbers) which can be used instead. Also, for associative symbols (such as the conjunction $\wedge$ in logic programming) one can represent $c_1 \wedge \ldots \wedge c_n$ by $\wedge(c_1, \ldots, c_n)$ and then use equality up to arities (e.g., $\wedge/2 = \wedge/3$) for $\preceq_F$.

*Example 5.* If we take $\preceq_F$ to be equality up to arities and ignore $\preceq_S$ (i.e., define $\preceq_S$ to be always true) we get all the embeddings of $\trianglelefteq$, for example, $p(a) \trianglelefteq^* p(f(a))$. But we also get

- $p(a) \trianglelefteq^* p(b, f(a), c)$ (while $p(a) \trianglelefteq p(b, f(a), c)$ does not hold),
- $\wedge(p(a), q(b)) \trianglelefteq^* \wedge(s, p(f(a)), r, q(b))$, and
- $\wedge(a, b, c) \trianglelefteq^* \wedge(a, b, c, d)$.

One can see that $\trianglelefteq^*$ provides a convenient way to handle associative operators such as the conjunction $\wedge$. (Such a treatment of $\wedge$ has been used in [29, 39, 13] to ensure termination of conjunctive partial deduction.) Indeed, in the context of $\trianglelefteq$ one has to use, e.g., a binary representation. But then whether $\wedge(a, b, c)$ is embedded in $\wedge(a, b, c, d)$ depends on the particular representation, which is not very satisfactory:

- $\wedge(a, \wedge(b, c)) \trianglelefteq \wedge(a, \wedge(\wedge(b, c), d))$, but
- $\wedge(a, \wedge(b, c)) \not\trianglelefteq \wedge(\wedge(a, b), \wedge(c, d))$.

In the above definition we can now instantiate $\preceq_S$ such that it performs a more refined treatment of variables, as done for $\trianglelefteq^+$. For example, we can define: $\langle s_1, \ldots, s_m \rangle \preceq_S \langle t_1, \ldots, t_n \rangle$ iff $\langle t_1, \ldots, t_n \rangle$ is not strictly more general than $\langle s_1, \ldots, s_m \rangle$. (Observe that this means that if $m \neq n$ then $\preceq_S$ will hold.) This relation is a wbr (as the strictly more general relation is a wfo [35]). Then, in contrast to $\trianglelefteq^+$ and $\trianglelefteq_{var}$, this refinement will be applied *recursively* within $\trianglelefteq^*$.

For example, we now not only have $p(X, X) \not\trianglelefteq^* p(X, Y)$ but also $f(a, p(X, X)) \not\trianglelefteq^* f(f(a), p(X, Y))$ whereas $f(a, p(X, X)) \trianglelefteq^+ f(f(a), p(X, Y))$.

The reason why a recursive "not strict instance" test was not incorporated in [53] (which uses $\trianglelefteq^+$) was that the authors were not sure that this would remain a wbr (no proof was found yet). In fact, at first sight it looks like recursively applying the "not strict instance" might endanger termination.[5] But the following result, proven in [46], shows that this is not the case:

**Theorem 4.** *$\trianglelefteq^*$ is a wbr on expressions. Additionally, if $\preceq_F$ and $\preceq_S$ are wqos then so is $\trianglelefteq^*$.*

**Other Extensions** Other extensions, in the context of static termination analysis of term rewriting systems, are proposed in [68] and [45]. Further research is required to determine their usefulness in an online setting.

[28] presents an extension of $\trianglelefteq$ which can handle multiple levels of encodings in the context of metaprogramming. We will examine the issue of metaprogramming in much more detail in the next section.

## 8 Metaprogramming: Some Results and Open Problems

As we have seen earlier in Sect. 5, $\trianglelefteq$ alone is already very flexible for metainterpreters, even more so when combined with characteristic trees [53] (see also [80]). This section we will study the issue of metaprogramming in more detail, present some new results, and show that some subtle problems still remain. This section is slightly more biased towards partial deduction of logic programs. The discussions should nonetheless be valid for most of the other application areas, namely when the symbolic method is not applied to a system/program directly but to an encoding of it.

**Ground versus Non-ground Representation** Let us first discuss one of the main issues in the context of metaprogramming, namely the representation of object-level expressions at the metalevel. In setting with logical variables, there are basically two different approaches to representing an object level expression, say $p(X, a)$, at the metalevel. In the first approach one uses the expression itself as the object level representation. This is called a *non-ground* representation, because it represents an object level variable by a metalevel variable. In the second approach, one uses something like $struct(p, [var(1), struct(a, [\,])])$ to represent the object level expression. (Usually one does not use the representation $p(var(1), a)$, because then one cannot use the function symbol $var/1$ at the object level.) This is called a *ground* representation, as it represents an object level

[5] If we slightly strengthen point 3 of Def. 13 by requiring that $\langle s_1, \ldots, s_m \rangle$ is not a strict instance of the selected subsequence $\langle t_{i_1}, \ldots, t_{i_m} \rangle$, we actually no longer have a wbr [46].

| Object level | Ground representation |
|---|---|
| $X$ | $var(1)$ |
| $c$ | $struct(c, [\,])$ |
| $f(X, a)$ | $struct(f, [var(1), struct(a, [\,])])$ |

**Fig. 3.** A ground representation

variable by a ground term. Fig. 3 contains some further examples of the particular ground representation which we will use in this section. For a more detailed discussion we refer the reader to [34, 8].

Of course, one is not restricted to just one level of metainterpretation; one can have a whole hierarchy of metainterpretation [27] where each layer adds its own functionality.

In this section we want to study the relationship between admissible sequences at the object and metalevel. The simplest setting is when one just uses metainterpreters which mimic the underlying execution and do not add any functionality (in functional programming, such interpreters are often called self-interpreters [37]). Now, the first question that comes to mind is: "If a sequence of evaluation steps at the object level is admissible wrt some well-quasi order, what about the corresponding sequence of evaluation steps at the meta level ?" Ideally, one would want a well-quasi order which is powerful enough to also admit the sequence at the metalevel. In the context of partial deduction this would ensure that if an object program and query can be fully unfolded then the same holds at the metalevel, no matter how many layers of interpretation we put on top of each other. Unfortunately, as we will see below, finding such a wqo turns out to be a daunting task.

## 8.1 The Representation Problem

In this subsection we will concentrate on the difficulties arising from the fact that object level expressions have to be represented in a different (and possibly more complex) manner at the metalevel. We will ignore for the moment that sequences of expressions at the metalevel might actually be even more involved (i.e., there might be intermediate expressions which have no counterpart at the object level; or a sequence at the object level might correspond to multiple sequences at the metalevel). We will return to some of these issues in Sect. 8.2 below.

To abstract from the number of layers of metainterpretation and the particular representation employed at each layer, we define a function *enc*(.) which maps object level expressions to corresponding metalevel expressions. For example, if we just use the logic programming, (non-ground) vanilla metainterpreter [34, 8] depicted in Fig. 4 we have $enc(p(X)) = solve(p(X))$. If we use two nested vanilla metainterpreters we will get $enc(p(X))= solve(solve(p(X)))$. More involved metainterpreters might actually have additional arguments, such as a debugging trace. For the ground representation of Fig. 3 we will get $enc(p(X))$

$solve(true) \leftarrow$
$solve(A\&B) \leftarrow solve(A) \wedge solve(B)$
$solve(H) \leftarrow clause(H,B) \wedge solve(B)$

**Fig. 4.** The vanilla metainterpreter

$= solve(struct(p, [var(1)]), CAS)$, where $CAS$ is an output variable for the computed answer substitution, which has to be returned explicitly.

We say that a wfo or wbr is *invariant under a particular encoding* $enc(.)$ if whenever it admits a sequence of expressions $o_1, o_2, \ldots, o_n$ then it also admits $enc(o_1), enc(o_2), \ldots, enc(o_n)$, and vice-versa. Solving the representation problem then amounts to finding an adequate wfo or wbr which is invariant under a given encoding and still powerful enough (obviously the total relation $\preceq_\top$ with $\forall s,t : s \preceq_\top t$ is a wqo which is invariant under any encoding $enc(.)$).

We now show that $\trianglelefteq$ solves the representation problem in the context of the vanilla metainterpreter of Fig. 4. In the following we use $f^n(t)$ as a shorthand for the expression $\underbrace{f(\ldots(f(}_{n}t)\ldots)$.

**Proposition 2.** *Let $o_1, o_2, \ldots, o_n$ be a sequence of expressions. $o_1, o_2, \ldots, o_n$ is admissible wrt $\trianglelefteq$ iff $solve^n(o_1), solve^n(o_2), \ldots, solve^n(o_n)$ is admissible wrt $\trianglelefteq$.*

*Proof.* It is sufficient to show that $o_i \trianglelefteq o_j$ iff $solve(o_i) \trianglelefteq solve(o_j)$. Obviously $o_i \trianglelefteq o_j$ implies $solve(o_i) \trianglelefteq solve(o_j)$ by applying the coupling rule 3 of Def. 8. Now suppose that $solve(o_i) \trianglelefteq solve(o_j)$. Either rule 3 of Def. 8 was applied and we can conclude that $o_i \trianglelefteq o_j$. Or the diving rule 2 was applied. This means that $solve(o_i) \trianglelefteq o_j$. At that point it is possible that the diving rule 2 was further applied, but sooner or later rule 1 or 3 must be applied and we can then conclude that $o_i \trianglelefteq t$ for some subterm $t$ of $o_j$. Hence we know that $o_i \trianglelefteq o_j$. □

The above result also holds for more involved vanilla-like encodings with extra-arguments provided that within every sequence under consideration an extra-argument does not use the *solve* function symbol and that it embeds all the earlier ones. This holds, e.g., for extra-arguments which contain constants, increasing counters, or histories. For example, the sequence $p, q$ is admissible and so are

- $solve(p, 0), solve(q, s(0))$ (where the interpreter counts evaluation steps) and
- $solve(p, [\,]), solve(q, [p])$ (where the interpreter keeps track of the evaluation history).

However, if the extra argument does not necessarily embed the earlier ones, then more sequences might be admissible at the metalevel. For example, if we add to Fig. 4 a resolution counter, counting downwards, we have that $p(a), p(f(a))$ is not admissible while $solve(p(a), s(0)), solve(p(f(a)), 0)$ is. Alternatively, if the extra argument can contain the *solve* symbol then we can have sequences admissible at the object level but not at the meta level: $p, q$ is admissible while $solve(p, 0), solve(q, solve(p, 0))$ is not.

One might hope that a similar invariance property holds for a ground representation. Unfortunately, this is not the case due to several problems, which we examine below.

**Multiple Arity** If the same predicate symbol can occur with multiple arity, then the following problem can arise. Take the expressions $p(X)$ and $p(X,X)$. We have that $p(X) \not\trianglelefteq p(X,X)$ ($\trianglelefteq$ inherently treats $p/1$ and $p/2$ as different symbols and the coupling rule 3 of Def. 8 cannot be applied). However, for the ground representation we have $\mathit{struct}(p,[X]) \trianglelefteq \mathit{struct}(p,[X,X])$ because all arguments of $p/n$ are put into a single argument of *struct* containing a list of length $n$.

A simple solution to this problem is to use a predicate symbol only with a single arity. Another one is to use $\trianglelefteq^*$ (instead of $\trianglelefteq$) with an underlying identity of function symbols up to arities. A third solution is to add the arity as an extra argument in the ground representation. For the above example, we then obtain $\mathit{struct}(p,1,[X]) \not\trianglelefteq \mathit{struct}(p,2,[X,X])$. This approach also solves more contrived examples such as $p(X) \not\trianglelefteq p(X,f(X))$, where we then have $\mathit{struct}(p,1,[X]) \not\trianglelefteq \mathit{struct}(p,2,[X,\mathit{struct}(f,1,[X])])$.

**Variable Encoding** If we represent variables as integers of the form $\tau = 0 \mid s(\tau)$, we can have that $X \trianglelefteq Y$ while $\mathit{var}(s(0)) \not\trianglelefteq \mathit{var}(0)$. (In that case $\trianglelefteq$ on the encoding is actually more admissible.) One solution is to use a different function symbol for each distinct variable (meaning we have an infinite alphabet) and then use $\trianglelefteq^*$ with an underlying wqo on these new function symbols, e.g., either treating all encodings of variables as one fresh constant or even incorporating refinements similar to $\trianglelefteq^+$ and $\trianglelefteq_{var}$.

**Multiple Embeddings in the Same Argument** Unfortunately, even in the absence of variables and even if every function symbol only occurs with a single arity, $\trianglelefteq$ is not invariant under the ground representation. For example, we have

$$f(a,b) \not\trianglelefteq f(g(a,b),c)$$

while

$$\mathit{struct}(f,[\text{“}a\text{”},\text{“}b\text{”}]) \trianglelefteq \mathit{struct}(f,[\mathit{struct}(g,[\text{“}a\text{”},\text{“}b\text{”}]),\text{“}c\text{”}])$$

where we have used “$c$” to denote the ground representation $\mathit{struct}(c,[\,])$ of a constant symbol $c$.

The reason for this odd behaviour is that the coupling rule 3 of Def. 8 checks whether the term $a$ is embedded in $g(a,b)$ (which holds) and $b$ is embedded in $c$ (which does not hold). The rule *disallows* to search for *both* $a$ and $b$ in the *same argument* $g(a,b)$ (which would hold). But this is exactly what *is allowed* when working with the ground representation, due to the fact that an argument tuple is translated into a list.

One might think that one possible solution to this problem would be to adapt $\trianglelefteq$ such that one is allowed to examine the *same* argument multiple times for embedding. In other words one would define a relation $\trianglelefteq^-$ by adapting rule 3 of Def. 8 to (and keeping rules 1 and 2):

3. $f(s_1, \ldots, s_m) \trianglelefteq^- f(t_1, \ldots, t_n)$ if $\exists 1 \leq i_1 \leq \ldots i_m \leq n$ such that $\forall j \in \{1, \ldots, m\} : s_j \trianglelefteq^- t_{i_j}$.

Note that, contrary to $\trianglelefteq$, $i_j$ can be equal to $i_{j+1}$ and we now have $f(a,b) \trianglelefteq^- f(g(a,b),c)$. Unfortunately, this solution is not invariant under the ground representation either. A counterexample is as follows: $f(a) \not\trianglelefteq^- g(f(b), g(a))$ while $\mathit{struct}(f, [\text{“}a\text{”}]) \trianglelefteq^- \mathit{struct}(g, [\mathit{struct}(f, [\text{“}b\text{”}]), \mathit{struct}(g, [\text{“}a\text{”}])])$.

So, despite the usefulness of $\trianglelefteq$ for metaprogramming exhibited earlier in the paper, there still remains the open problem: Can we find a strengthening or useful adaptation of $\trianglelefteq$ which is invariant under the ground representation? It might be possible to achieve this by using (a refinement of) [68], which extends Kruskal's theorem. A pragmatic solution would of course be to simple decode data as much as possible in, e.g., the program specialiser, and then apply $\trianglelefteq$ (or $\trianglelefteq^*$) on the de-encoded data only. This, however, requires knowledge about the particular encodings that are likely to appear. A more refined and promising approach for handling multiple levels of encodings within $\trianglelefteq$ is presented in [28].

## 8.2 The Parsing Problem

In the context of metaprogramming. we also encounter the so-called *parsing problem* [59]. Below we provide another view of the parsing problem, in terms of invariance under representation.

In partial deduction of logic programs, nobody has found it useful to compare complete goals, only the *selected* atoms within the goals are compared. Also, it was quickly realised that it was difficult to define an order relation on the full sequence that was giving good results and that it was sufficient and easier to do so on certain subsequences containing the so-called covering ancestors [11].

Take, for example, the program $P$ consisting of the two clauses:

$p(f(X)) \leftarrow p(b) \wedge p(f(X))$
$p(a) \leftarrow$

Let us unfold the goal $\leftarrow p(f(Y))$ by using $\trianglelefteq$ on the selected atoms (selected atoms are underlined):

$$\begin{array}{c} \leftarrow \underline{p(f(Y))} \\ \downarrow \\ \leftarrow \underline{p(b)} \wedge p(f(Y)) \\ \downarrow \\ \mathit{fail} \end{array}$$

The covering ancestor sequence for $p(b)$ is $p(f(Y)), p(b)$ which is admissible wrt $\trianglelefteq$. We were thus successful in fully unfolding $\leftarrow p(f(Y))$ and detecting finite failure. If we had looked at the entire goals, we would not have been able to fully unfold $\leftarrow p(f(Y))$, because $p(f(Y)) \trianglelefteq p(b) \wedge p(f(Y))$.

Let us examine how this refinement fares in the context of metaprogramming. For this we take the standard vanilla metainterpreter of Fig. 4, together with the following encoding of the above program $P$:

$clause(p(f(X)),(p(b)\&p(f(X)))) \leftarrow$
$clause(p(a),true) \leftarrow$

One would hope that, by using $\trianglelefteq$ on the selected atoms and comparing with the covering ancestors, it would be possible to fully unfold $\leftarrow solve(p(f(Y)))$ in a similar manner as for $\leftarrow p(f(Y))$ above. Let us examine the sequence of goals, needed to detect finite failure:

$$\begin{array}{c}
\leftarrow \underline{solve(p(f(Y)))} \\
\downarrow \\
\leftarrow \underline{clause(p(f(Y)),B)} \wedge solve(B) \\
\downarrow \\
\leftarrow \underline{solve(p(b)\&p(f(Y)))} \\
\downarrow \\
\leftarrow \underline{solve(p(b))} \wedge solve(p(f(Y))) \\
\downarrow \\
\leftarrow \underline{clause(p(b),B')} \wedge solve(B') \wedge solve(p(f(Y))) \\
\downarrow \\
fail
\end{array}$$

Unfortunately, even if we ignore the intermediate goals containing *clause* atoms (they have no counterpart at the object level) we have a problem: at the third step we select $solve(p(b)\&p(f(Y)))$ who has the covering ancestor $solve(p(f(Y)))$ with $solve(p(f(Y))) \trianglelefteq solve(p(b)\&p(f(Y)))$. The same embedding holds for $\trianglelefteq^{+}$ or $\trianglelefteq^{*}$. We are thus unable to fully unfold $\leftarrow solve(p(f(Y)))$. The problem is that, in the metainterpreter, multiple atoms can be put together in a single term, and the refinement of looking only at the selected atoms and their covering ancestors is not even invariant under the non-ground representation!

Solving this problem in a general manner is a non-trivial task: one would have to know that *solve* will eventually decompose $p(b)\&p(f(Y))$ into its constituents $p(b)$ and $p(f(Y))$ giving us the opportunity to continue with $solve(p(b))$ while stopping the unfolding of $solve(p(f(Y)))$. [80] presents a solution to this problem for the particular vanilla metainterpreter above, but unfortunately it does not scale up to other, more involved metainterpreters.

## 9 Discussion and Conclusion

*Critical Evaluation and Future Work* In theory, existing online systems, such as INDY and ECCE, based on $\trianglelefteq$, ensure termination in a fully automatic manner and can thus be used even by a naïve user. However, for more involved tasks, these systems can lead to substantial code explosion, meaning that some user expertise is still required to prevent such cases. Also, these systems might fail to provide a specialised program that is more efficient in practice, because existing control techniques fail to take certain pragmatic issues into account [49]. Indeed, although $\trianglelefteq$ has proven to be extremely useful superimposed, e.g., on determinate unfolding, on its own it will sometimes allow too much unfolding than desirable for efficiency concerns: more unfolding does not always imply a better specialised program and it can also negatively affect the efficiency of the specialisation process itself. Moreover, $\trianglelefteq$ can be used to fully evaluate the non primitive recursive Ackerman function (this is a corollary of Theorem 1; see also [58, p. 186–187]). Hence, $\trianglelefteq$ on its own can lead to a worst case complexity for the transformation/specialisation process which is not primitive recursive. Although cases of bad complexity seem to be relatively rare in practice, this still means that $\trianglelefteq$ should better not be used as is in a context (such as within a compiler) where a tight upper-bound on memory and time requirements is essential. However, we hope that it is going to be possible to engineer an efficient approximation of $\trianglelefteq$ (or $\trianglelefteq^*$), which takes more pragmatic issues into account.

On the other hand, for some applications, $\trianglelefteq$ as well as $\trianglelefteq^+$ and $\trianglelefteq^*$ remain too restrictive. As we have discussed in Sect. 8, they do not always perform satisfactorily in the context of arbitrary metainterpretation tasks. Only future research can tell whether one can solve these problems, while not (substantially) deteriorating the complexity. A completely different approach to termination has very recently been presented in [55]. It will be interesting to see how it compares to $\trianglelefteq$ and its derivatives.

*Conclusion* In summary, we have shed new light on the relation between wqos and wfos and have formally shown why wqos are more interesting, at least in theory, than wfos for ensuring termination in an online setting. We have illustrated the inherent flexibility of $\trianglelefteq$ and shown that, despite its simplicity, it is strictly more generous than a large class of wfos. We have surveyed the usage of $\trianglelefteq$ in existing techniques, and have touched upon some new application areas such as infinite model checking.

We have also discussed extensions of $\trianglelefteq$, which inherit all the good properties of $\trianglelefteq$ while providing a refined treatment of (logical) variables. We believe that these refinements can be of value in contexts such as partial evaluation of (functional and) logic programs or supercompilation of functional programming languages, where — at specialisation time – variables also appear. One can also simply plug $\trianglelefteq^*$ into the language-independent framework of [73]. We also believe that $\trianglelefteq^*$ provides both a theoretically and practically more satisfactory basis than $\trianglelefteq^+$ or $\trianglelefteq$.

Finally, we have discussed the use of $\trianglelefteq$ in the context of metaprogramming, proving some positive results for the non-ground representation, but also some negative results for both the ground and non-ground representation.

In summary, $\trianglelefteq$ is an elegant and very powerful tool to ensure online termination, but a lot of research is still needed to make it efficient enough for full practical use and powerful enough to cope with arbitrary metalevel encodings.

## Acknowledgements

First, I would like to thank Maurice Bruynooghe, Danny De Schreye, Robert Glück, Jesper Jørgensen, Neil Jones, Helko Lehmann, Bern Martens, Maurizio Proietti, Jacques Riche, Jens Peter Secher, and Morten Heine Sørensen for all the discussions and joint research which led to this chapter. I also wish to thank anonymous referees for their feedback. Finally, I would especially like to thank Neil Jones for kindling and re-kindling my enthusiasm for partial evaluation and also for his support and continuing interest in my work. Without his substantial research achievements and his continous stream of new ideas the area of computer science would have been a poorer place.

## References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 313–321, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[2] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving control in functional logic program specialization. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 262–277, Pisa, Italy, September 1998. Springer-Verlag.

[3] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A partial evaluation framework for curry programs. In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, LNCS 1705, pages 376–395. Springer-Verlag, 1999.

[4] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Spezialisation of lazy functional logic programs. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, 1997. ACM Press.

[5] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H. Riis Nielson, editor, *Proceedings of the 6th European Symposium on Programming, ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.

[6] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[7] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[8] K. R. Apt and F. Turini. *Meta-logics and Logic Programming.* MIT Press, 1995.

[9] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.

[10] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[11] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[12] D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[13] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[14] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.
[15] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
[16] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.
[17] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
[18] A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.
[19] A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proceedings of LATIN'98*, LNCS 1380, pages 102–118. Springer-Verlag, 1998.
[20] F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and strategies for contextual specialization of constraint logic programs. *Electronic Notes in Theoretical Computer Science*, 30(2), December 1999.
[21] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Logic Based Program Synthesis and Transformation. Proceedings of Lopstr'2000*, LNCS 1207, pages 125–146, 2000.
[22] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.
[23] D. A. Fuller and S. Abramsky. Mixed computation of Prolog programs. *New Generation Computing*, 6(2 & 3):119–141, June 1988.
[24] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
[25] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
[26] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
[27] R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 234–251, Utrecht, The Netherlands, September 1995. Springer-Verlag.
[28] R. Glück, J. Hatcliff, and J. Jørgensen. Generalization in hierarchies of online program specialization systems. In P. Flener, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'98*, LNCS 1559, pages 179–198, Manchester, UK, June 1998. Springer-Verlag.
[29] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.
[30] R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.
[31] J. Gustedt. *Algorithmic Aspects of Ordered Structures.* PhD thesis, Technische Universität Berlin, 1992.
[32] M. Hanus. The integration of functions into logic programming. *The Journal of Logic Programming*, 19 & 20:583–628, May 1994.

[33] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

[34] P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.

[35] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[36] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[37] N. D. Jones. *Computability and Complexity: From a Programming Perspective.* MIT Press, 1997.

[38] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[39] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

[40] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[41] J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[42] L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 168–188, Leuven, Belgium, July 1997.

[43] H. Lehmann and M. Leuschel. Solving planning problems by partial deduction. In M. Parigot and A. Voronkov, editors, *Proceedings of the International Conference on Logic for Programming and Automated Reasoning (LPAR'2000)*, LNAI 1955, pages 451–468, Reunion Island, France, 2000. Springer-Verlag.

[44] P. Lescanne. Rewrite orderings and termination of rewrite systems. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, LNCS 520, pages 17–27, Kazimierz Dolny, Poland, September 1991. Springer-Verlag.

[45] P. Lescanne. Well rewrite orderings and well quasi-orderings. Technical Report N$^o$ 1385, INRIA-Lorraine, France, January 1991.

[46] M. Leuschel. Improving homeomorphic embedding for online termination. In P. Flener, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'98*, LNCS 1559, pages 199–218, Manchester, UK, June 1998. Springer-Verlag.

[47] M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[48] M. Leuschel. Logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, Copenhagen, Denmark, 1999. Springer-Verlag.

[49] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[50] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings*

*of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

[51] M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[52] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.

[53] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[54] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.

[55] C. S. Lii, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*. ACM Press, January 2001.

[56] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[57] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[58] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, December 1994.

[59] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, K.U. Leuven, February 1994.

[60] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[61] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.

[62] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.

[63] J. Martin and M. Leuschel. Sonic partial deduction. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 101–112, Novosibirsk, Russia, 1999. Springer-Verlag.

[64] A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997.

[65] T. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, pages 247–279. Marcel Decker, 270 Madison Avenue, New York, New YOrk 10016, 1997.

[66] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

[67] A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 355–385, Schloß Dagstuhl, 1996. Springer-Verlag.

[68] L. Puel. Using unavoidable set of trees to generalize Kruskal's theorem. *Journal of Symbolic Computation*, 8:335–382, 1989.

[69] E. Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, March 1993.

[70] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[71] J. P. Secher and M. H. Sørensen. On perfect supercompilation. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 113–127, Novosibirsk, Russia, 1999. Springer-Verlag.

[72] L. Song and Y. Futamura. A new termination approach for specialization. In W. Taha, editor, *Proceedings of SAIG'00*, LNCS 1924, pages 72–91. Springer-Verlag, 2000.

[73] M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction, Proceedings of MPC'98*, LNCS 1422, pages 315–337. Springer-Verlag, 1998.

[74] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[75] M. H. Sørensen and R. Glück. Introduction to supercompilation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation — Practice and Theory*, LNCS 1706, pages 246–270, Copenhagen, Denmark, 1999. Springer-Verlag.

[76] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[77] J. Stillman. *Computational Problems in Equational Theorem Proving.* PhD thesis, State University of New York at Albany, 1988.

[78] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[79] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[80] W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.

[81] A. Weiermann. Complexity bounds for some finite forms of Kruskal's theorem. *Journal of Symbolic Computation*, 18(5):463–488, November 1994.

[82] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pages 165–191, Harvard University, 1991. Springer-Verlag.

# Simple Driving Techniques[⋆]

Mads Rosendahl

University of Roskilde, Dept. of Computer Science
Building 42.1, P.O. Box 260, DK-4000 Roskilde, Denmark
madsr@ruc.dk

**Abstract.** Driving was introduced as a program transformation technique by Valentin Turchin in some papers around 1980. It was intended for the programming language REFAL and used in metasystem transitions based on super compilation. In this paper we present one version of driving for a more conventional lisp-like language. Our aim is to extract a simple notion of driving and show that even in this tamed form it has much of the power of more general notions of driving. Our driving technique may be used to simplify functional programs which use function composition and will often be able to remove intermediate data structures used in computations.

A supercompiler (= supervised compiler) is a special program transformation system which, while evaluating the program, supervises its operation and uses the information to reconstruct the program in a more efficient way. A supercompiler may work on expressions with free variables and will then drive it through the interpretation process and produce a tree of states and transitions. The tree is made into a finite graph by generalizing patterns or configurations of expressions. A configuration is an expression without free variables but with "holes" in which one may insert other expressions. A supercompiler will have a strategy for generalizing configurations (*i.e.* make the holes bigger or make new holes). In order to guarantee termination the driving algorithm should generalize configurations into a finite set of simpler configurations called *basic configurations.*

The concept of super compilation is due to Valentin Turchin [10], [11]. The work and concepts are closely tied to the language Refal and not easily rephrased for other programming languages.

The strategy for generalizing configurations may be viewed as an abstraction function on expressions, or as an upper bound operation when several configurations should be described by a more general configuration. Different strategies will define different sets of possible configurations. To ensure that the driving terminates we need a strategy that does not produce infinitely many configurations for a program. As in abstract interpretation this may be achieved if the set of configurations is finite or has some finite-height property.

Our aim is to use driving to optimize functional programs where composition of user-defined functions plays a central role. We want to specialize functions

⋆ This research was partly sponsored by the IT-University in Copenhagen

when we know that the argument values are produced by certain other functions. We will use the driving technique and define a simple set of basic configurations. The result is a transformation system which is easy to implement and easy to use.

## 1 Background

The bulk of the work reported here was done in 1986 as part of the authors MSc. Thesis with Neil Jones as supervisor [4]. The abstract interpretation part of the thesis was later published in FPCA'89 [5] but much to the supervisor's regret the transformation part did not appear in journal form at that time.

The aim of the thesis was to construct time bound functions for programs - *i.e.* functions that return an upper bound of the execution time of a program given the size of input. The time bound function is constructed automatically in two stages. Using an abstract interpretation the program is transformed to a new program that computes the time bound function. The abstract interpretation is viewed as a different (abstract) semantics, thus defining a new language and we constructed a compiler from the language of the abstract semantics to the language of the standard semantics. This is reexamined in the final example of this paper.

In the second stage this generated program is simplified as much as possible, hopefully into a simple closed-form expression. The main challenge in the second stage is to remove various intermediate data structures used in the generated program. The driving technique described here is able to remove many such structures and more generally to optimize programs containing function compositions.

## 2 Language

The driving principle is here presented for a small lisp-like, first-order, eager, functional language. A program in the language consists of $n$-functions in a recursion equation system

$$\begin{array}{l} f_1(x_1, \ldots, x_{m_1}) = exp_1 \\ \ldots \\ f_n(x_1, \ldots, x_{m_n}) = exp_n \end{array}$$

Expressions are constructed from constants, parameter names, basic operations, conditional expressions and calls to functions in the program.

| | |
|---|---|
| $exp ::= c_i$ | numbers, strings, `true`, `false`, `nil` |
| $\mid\ x_i$ | parameters |
| $\mid\ op_i(exp_1, \ldots, exp_{i_n})$ | basic operations |
| $\mid$ `if` $exp_1$ `then` $exp_2$ `else` $exp_3$ | conditional |
| $\mid\ f_i(exp_1, \ldots, exp_{m_i})$ | function calls |

Operations include the functions `car`, `cdr`, `cons`, `atomp`, `null`, `eq` on dotted pairs and various arithmetic operations on numbers.

Running a program consists of calling the first function in the program with the input to the program. As examples of programs in the language consider

```
ff(x) = flip(flip(x))
flip(x) = if atomp(x) then x else cons(flip(cdr(x)),flip(car(x)))
```

and

```
sumn(n) = sum(fromn(n))
sum(x) = if null(x) then 0 else add(car(x),sum(cdr(x)))
fromn(n) = if eq(n,0) then nil else cons(n,fromn(sub(n,1)))
```

The first program flips a binary tree twice, thus returning a copy of the input as output. The second program computes the sum of the numbers from 1 to $n$, where $n$ is the input to the program.

## 3 Example

The example here is taken from Wegbreit [15].

Consider a program that appends three lists by the function

```
f(x,y,z)      = append(append(x,y),z)
append(x,y)   = if null(x) then y
                     else cons(car(x),append(cdr(x),y))
```

We try to improve this small program by analyzing the right hand side of the first function:

```
     append(append(x,y),z)
```

Here we immediately get the first function composition, and this is viewed as a *basic configuration* (or goal or procedure-expression). By functions we here mean the user defined functions, everything else is considered as basic expressions, since they cannot be unfolded. The configuration is represented using the notation `[append(append(_,_),_)]` where "_" denotes a hole which may be substituted for an arbitrary expression. The configuration is now viewed as a function and analyzed without its original context. This is in contrast to Turchin's work [11] where the configurations are recognized as being basic at their second appearance, hence the first reduction is done in the original context. The present method is chosen because of its simpler termination properties.

The configuration contains three free variables, and as with Turchin [11] we drive the free variables forcefully through the expressions of the program until the expression represents a *progressive* method for computing the configuration it defines (cp. [6]):

```
[append(append(_,_),_)] (a,b,c) =
    if null(a) then append(b,c)
     else cons(car(a),append(append(cdr(a),b),c)))
```

The driving is done by constructing a conditional expression with the first condition to be performed in a normal order evaluation of the expression. This method resembles the idea of outside-in reductions in driving REFAL [11].

This expression contains two configurations: The trivial one:

```
[append(_,_)]
```

and the original configuration:

```
[append(append(_,_),_)]
```

Hence we loop back and get the new function definition:

```
append3(a,b,c) = if null(a) then append(b,c)
                          else cons(car(a),append3(cdr(a),b,c))
```

Inserting in the original context the program is improved to

```
f(x,y,z)       = append3(x,y,z)
append(x,y)    = if null(x) then y
                        else cons(car(x),append(cdr(x),y))
append3(a,b,c) = if null(a) then append(b,c)
                          else cons(car(a),append3(cdr(a),b,c))
```

## 4 The Driving Algorithm

When unfolding function calls in program transformation systems certain criteria are needed to ensure termination. The most restrictive rule (after no unfolding) would be only to allow one unfolding per function in each branch. Here we use a somewhat extended principle by only allowing one unfolding per *basic configuration* in each branch. By only defining a finite number of basic configurations termination is easily secured. In this section we define the basic configurations and show how progressive computation schemes can be established. Afterwards the driving algorithm is described.

Unlike Turchin we define a fixed set of basic configurations independently of the program we transform. Our basic configurations are expressions with function compositions - *i.e.* where arguments to function calls are function calls. We will show that with this limited set of configurations we are able to make quite powerful transformations.

The set of basic configurations is defined as equivalence classes in the set of expressions, and two expressions are equivalent if they are function calls with call to the same functions as arguments. The configuration can be characterized by the first function name and a list of arguments with either a function name or an indication that the argument can be anything. We write basic configurations in square brackets `[`$f_i(u_1, \ldots, u_{m_i})$`]` where $u_k$ is either a hole: "_" or a function call where all arguments are unknown: $f_k(\_, \ldots, \_)$. As an example `[append(append(_,_),_)]` is a basic configuration but `[append(append(cdr(_),_),_)]` is not and will be generalized to the former.

A basic configuration where all arguments are holes (_) is called a trivial basic configuration. There is a natural ordering of basic configurations with the trivial configurations being more general than configurations with function names as arguments. Our aim is to locate as many expressions as possible based on non-trivial basic configurations in the program and construct better computational schemes for these configurations.

Given a (non-trivial) basic configuration we define a new function to compute expressions of this form. We use the most general expression described by a basic configuration as the right hand side and the free variables in the expression as parameter names.

1. Search the expression in normal order (outside-in) for the first basic condition to be evaluated. A condition (a test in an if-expression) is basic if it does not contain any calls to user defined functions.
2. In the context of this condition being respectively true and false the arguments to the outer function call are reduced (and unfolded) as much as possible without producing any conditional expressions.
3. If the two new argument lists are reduced to progressive forms, *i.e.* contain calls where arguments are smaller, then insert them in the function call resulting in two new expressions. They are reduced as much as possible without getting any conditional expressions, and *without removing any function compositions.* A conditional expression is now constructed by the condition found above and these two expressions. The last requirement is stronger than normally in driving, but it is due to our restricted set of basic configurations. The first requirement is equivalent to reduction of *transient* configurations in super compilation.
4. If the two argument sets could not use the information from the condition, then unfold the outer function call and insert the arguments. The expression is then reduced as much as possible as in (3).

## Reductions

"Reduce as much as possible" means the application of a number of standard reduction schemes (the names are taken from [15]):

Local simplification rules. This is also called symbolic evaluation (in [7]). For examples `null(cons(`$a$`,`$b$`))` is changed to `false` and `car(cons(`$a$`,`$b$`))` is changed to $a$. Such transformations are safe provided the sub-expressions do not contain non-terminating computations.

Distributing conditionals. This is also called reduction to normal form (in [14]). The conditionals are taken out of arguments to basic operations and functions, so they appear outermost in expressions. This makes the next step easier. For example

```
append(if null(a) then nil else cons(car(a),append(cdr(a),b)),c)
```

is change into

```
if null(a) then append(nil,c) else
append(cons(car(a),append(cdr(a),b)),c)
```

Evaluating in context. In the simplest form this consists in removing duplicate conditions in an expression. When a function call is expanded we know the result of all predicates in conditionals leading to the expression.

Expanding function bodies. This is also called application (in [6]) or unfolding. The method must be used with care, and we adopt a criteria similar to Wegbreit's [15]: Functions are only expanded if they give condition free-expressions and if they do not remove function compositions in the program.

## 5 Collecting Basic Configurations

The collection of basic configurations and simplification of their defining expressions is essentially a fixpoint problem. We only want to analyze the basic configurations that can be reached (called) from the first function in the program, and simplifying a right hand side for a basic configuration may result in other expressions being simplified. Our approach uses a depth-first solution to the problem. We start by analyzing the first basic configuration in the program. The analysis is suspended whenever we locate a new basic configuration to be analyzed. A single depth-first analysis is sufficient in the cases we have considered and we have not encountered examples where further recursion would improve the program.

The driving algorithm starts at the right hand side of the first function in the program. All basic configurations are examined from outside in, and progressive definition of the configurations (found in the last section) are analyzed in the same way recursively. To secure termination of the driving algorithm a basic configuration is only analyzed once in each branch, and to improve the residual program, three operations are performed: generalization of basic configurations, recognition of earlier defined function and solving difference equations. The algorithm is described in two parts: a local one dealing with the analysis of a single basic configuration, and the global one controlling generalization.

**Local part.** The driving algorithm treats basic configuration in the following five steps.

Given a basic configuration:

1. Construct a progressive computation scheme for it (see previous section).
2. Locate and drive basic configurations in sub-expressions by the global part of the driving algorithm.
3. Locate calls to the original basic configuration. If there are such recursive calls then the expression is a recursive definition of the basic configuration viewed as a function in its free variables. This definition is then examined by:
4. (a) Matching it with function definitions in the original program. If it is defined in advance then substitute the expression with a call to this function.
   (b) Solve difference equations by matching the definition with known functions with simpler computational form.

   (c) Otherwise construct a new recursive function and substitute the expression with a call to this function. (Folding).
5. If there are no recursive calls in the expression then do nothing.

**Generalization.**
Assume that, while driving a configuration `[f(g(_,...,_),h(_,...,_))]` we reach the configuration `[f(g(_,...,_),k(_,...,_))]` and we do not reach other calls to the configuration `[f(g(_,...,_),h(_,...,_))]`. In this situation the original configuration should be generalized to `[f(g(_,...,_),_)]`. When analyzing the defining expression for this configuration all appearances of more specific configurations (*i.e.* where the second argument is a function call) should also be generalized to this configuration. This idea is well known in the literature [6] [11] [15].

**Global part.** The global part of the driving algorithm controls the program transformation system.

1. Start with the right hand side of the first function in the program.
2. Search the expression from outside in for a basic configuration.
3. If the configuration is a generalization of an identified, but suspended configuration, then backtrack and generalize the first identified (outermost) configuration. If the configuration is either trivial or identical with a suspended configuration, then loop back by returning the expression.
4. Otherwise suspend the analysis of the current configuration while we analyze this new configuration with the local part of the driving algorithm.
5. If the resulting expression is a conditional expression then the basic configuration was not successful; continue driving with the arguments of the basic configuration (the original call).
6. Otherwise reduce the expression in its context and continue driving with this expression.

## 6 Finite Difference Equations

Certain recursive functions on integers may be simplified into well-known closed-form expressions. A quite general rule is the following:

$$\begin{aligned} F(x) &= \texttt{if}\, x = \ell\, \texttt{then}\, c_1\, \texttt{else}\, f(x) + c_2 \cdot F(x-1) \\ &= c_1 \cdot c_2^{x-\ell} + \sum_{i=\ell+1}^{x} f(i) \cdot c_2^{x-i} \end{aligned}$$

When viewed as a program transformation it may not, however, produce a simpler program since the summation and power operation will be done using a recursive function. The original program uses $(x-\ell)$ recursive calls whereas the transformed program will use in the order of $(x-\ell)^2$ calls. There are some special cases, however, where the transformation will produce simpler programs. We will consider four such cases.

$$\begin{aligned} F(x) &= \texttt{if}\, x = \ell\, \texttt{then}\, c_1\, \texttt{else}\, c_2 + F(x-1) \\ &= x \cdot c_2 + (c_1 - c_2 \cdot \ell) \end{aligned}$$

$$\begin{aligned} F(x) &= \texttt{if}\, x = \ell\, \texttt{then}\, c_1\, \texttt{else}\, c_2 + c_3 \cdot x + F(x-1) \\ &= \frac{c_3}{2} \cdot x^2 + \Big(c_2 + \frac{c_3}{2}\Big) \cdot x + \Big(c_1 - \frac{c_3}{2} \cdot \ell(\ell - 1)\Big) \end{aligned}$$

$$\begin{aligned} F(x) &= \texttt{if}\, x = \ell\, \texttt{then}\, c_1\, \texttt{else}\, c_2 + c_3 \cdot F(x-1) \qquad \text{where } c_3 \neq 1 \\ &= G(x) \cdot \left(c_1 - \frac{c_2}{1 - c_3}\right) + \frac{c_2}{1 - c_3} \end{aligned}$$
where $G(x) = \texttt{if}\, x = \ell\, \texttt{then}\, 1\, \texttt{else}\, c_3 \cdot G(x-1)$

$$\begin{aligned} F(x) &= \texttt{if}\, x = \ell\, \texttt{then}\, c_1\, \texttt{else}\, c_2 + c_3 \cdot x + c_4 \cdot F(x-1) \qquad \text{where } c_4 \neq 1 \\ &= G(x) \cdot \left(\frac{c_2 + \ell \cdot c_3}{c_4 - 1} + c_1 + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) - x \cdot \frac{c_3}{c_4 - 1} - \left(\frac{c_2}{c_4 - 1} + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) \end{aligned}$$
where $G(x) = \texttt{if}\, x = \ell\, \texttt{then}\, 1\, \texttt{else}\, c_4 \cdot G(x-1)$

**General decrement.**
There are some similar cases for numeric function on data structures. One simple example is the following which may be useful as a simplification rule when driving numeric programs.

$$\begin{aligned} \texttt{F(x)} &= \texttt{if null(x) then}\, c_1\, \texttt{else add(}\, c_2\, \texttt{,F(cdr(x)))} \\ &= \texttt{add(}\, c_1\, \texttt{,mul(}\, c_2\, \texttt{,length(x)))} \end{aligned}$$

## 7 Example: `flip`

Consider the program

```
ff(x) = flip(flip(x))
flip(x) = if atomp(x) then x else cons(flip(cdr(x)),flip(car(x)))
```

When we use the driving technique on this program the initial basic configuration is `[flip(flip(_))]`. The first basic condition to be evaluated in the expression `flip(flip(x))` is `atomp(x)`. If this condition is true then the expression `flip(flip(x))` is easily simplified to `x`. If the condition is false we attempt to expand the inner function call:

```
flip(if atomp(x) then x else cons(flip(cdr(x)),flip(car(x))))
```

which in this context may be simplified to

```
flip(cons(flip(cdr(x)),flip(car(x))))
```

Since the expression is condition-free we may proceed and expand the outer function call:

```
if atom(cons(flip(cdr(x)),flip(car(x))))
   then cons(flip(cdr(x)),flip(car(x)))
   else cons(flip(cdr(cons(flip(cdr(x)),flip(car(x))))),
             flip(car(cons(flip(cdr(x)),flip(car(x))))))
```

This may be reduced to

```
cons(flip(flip(car(x)),flip(flip(cdr(x)))
```

This is recognised as a successful reduction since no function compositions are removed and since it constitutes a progressive evaluation scheme. The resulting program is now

```
ff(x) = if atomp(x) then x else cons(ff(car(x)),ff(cdr(x)))
```

Although this is the identity function we cannot prove this using the driving technique. We have, however, removed the binary tree constructed as an intermediate data structure in the program.

## 8 Example: sumn

Let us also consider the program

```
sumn(n) = sum(fromn(n))
sum(x) = if null(x) then 0 else add(car(x),sum(cdr(x)))
fromn(n) = if eq(n,0) then nil else cons(n,fromn(sub(n,1)))
```

The initial basic configuration is `[sum(fromn(_))]`. The first basic condition to be evaluated in the expression `sum(fromn(n)` is `eq(n,0)`. If this condition is true then the expression is easily reduced to `0`. Otherwise we get the expression

```
sum(cons(n,fromn(sub(n,1))))
```

and unfolded to

```
add(car(cons(n,fromn(sub(n,1)))),sum(cdr(cons(n,fromn(sub(n,1))))))
```

The reduced definition for the basic configuration is now

```
[sum(fromn(_))] (n) = if eq(n,0) then 0 else
add(n,sum(fromn(sub(n,1)))
```

This is one of the finite difference equations listed above, and we obtain the result

```
sumn(x) = add(mul(div(1,2),mul(x,x)),mul(div(1,2),x))
```

This could also be written as

$$\texttt{sumn}(\texttt{x}) = \tfrac{1}{2}x^2 + \tfrac{1}{2}x$$

## 9 Time Bound Programs

A time bound program is a program which from information about the size of input computes an upper bound to the execution time of the program. As execution time we will here use the number of sub-expressions being evaluated. We will here outline how time bound programs are constructed. A fuller account in a semantic framework may be found in [5]. The motivation for our work on driving was to transform time bound programs into closed form (non-recursive) expressions.

Consider the reverse function in the language

```
reverse (x) =
     if null(x) then nil
        else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
     if null(x) then y
        else cons(car(x),append(cdr(x),y))
```

*Step counting version* The first part is to extend the program with profiling information so that the output of the program is the number of sub-expressions being evaluated. We call this the step counting version of the program

```
t-reverse (x) =
     if null(x) then 4
        else add(11,add(t-reverse(cdr(x))
                    t-append(reverse(cdr(x))) ))
t-append (x) =
    if null(x) then 4 else add(10,t-append(cdr(x)))
reverse (x) =
     if null(x) then nil
        else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
     if null(x) then y
        else cons(car(x),append(cdr(x),y))
```

This program is constructed fairly easily from the original program using a transformation function $\mathbf{T}$ on expressions

$$\begin{aligned}
&\mathbf{T}[c_i] &&= \texttt{1}\\
&\mathbf{T}[x_i] &&= \texttt{1}\\
&\mathbf{T}[op_i(e_1,\ldots,e_n)] &&= \texttt{add(1,}\mathbf{T}[e_1]\texttt{,}\cdots\texttt{,}\mathbf{T}[e_n]\texttt{)}\\
&\mathbf{T}[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3] &&= \texttt{if}\ e_1\ \texttt{then add(1,}\mathbf{T}[e_1]\texttt{,}\mathbf{T}[e_2]\texttt{) else add(1,}\mathbf{T}[e_1]\texttt{,}\mathbf{T}[e_3]\texttt{)}\\
&\mathbf{T}[f_i(e_1,\ldots,e_n)] &&= \texttt{add(1,t-}f_i(e_1,\ldots,e_n)\texttt{ , }\mathbf{T}[e_1]\texttt{ , }\cdots\texttt{ , }\mathbf{T}[e_n]\texttt{ )}
\end{aligned}$$

*Abstract semantics.* The arguments to a step counting version of a program are ordinary values in our lisp-like language: numbers, strings and dotted pairs of values and output is numbers. The next part is to extend the value domain with a special atom `all` that denotes any possible atom. The `all` may also appear in dotted pairs. We then construct an abstract interpretation which gives an upper bound of what the standard semantics would have given when `all` values are substituted with other atoms. The abstract interpretation is an extension of the standard semantics. The main difference is that conditions in if-expressions may have the value `all`. If the condition is `all` then the result should be an upper bound of the results in the two branches.

The usual approach in abstract interpretation is to implement it directly - typically using some fixpoint iteration strategy. We, however, view it as a new semantics. It defines a new language with the same syntax but different semantics. The construction now proceeds by translating the program in this special semantics into a new program in our usual semantics. The abstract semantics and the translation scheme is described in [5]. The translation changes basic operations and conditional expressions but leaves constants, parameters and function calls unchanged. For example the equality test `eq(a,b)` is translated into

```
if or(eq(a,"all"),eq(b,"all")) then "all" else eq(a,b)
```

The conditional expression `if a then b else c` in a step counting version of a function is translated into

```
if eq(a,all) then max(b,c) else if a then b else c
```

For other functions the translation is

```
if eq(a,all) then lub(b,c) else if a then b else c
```

where the `lub` function is defined as

```
lub(a,b) = if eq(a,b) then a else
      if and(consp(a),consp(b))
      then cons(lub(car(a),car(b)),lub(cdr(a),cdr(b)))
      else "all"
```

We refer to [5] for examples of programs after this translation.

*Inverted size measures.* When we want to construct a time bound program we need a size measure. For some programs it is the length of input lists but for other programs it the maximal depth of binary trees or something else. In the example here the length function is the natural choice. Our approach may, however, be used for other size measures.

We may use values with `"all"` atoms to represent sets of values. For example `("all".("all".("all".nil)))` represents all lists of length 3. This leads us to define the function

```
length-1 (x) =
     if eq(x,0) then nil
        else cons("all",length-1(sub(x,1))))
```

which has the property that `length(length-1(`$n$`))` $= n$ for any non-negative number. We call the function `length-1` an inverted size measure. The other composition `length(length-1(`$x$`))` is not, however, an identity map. It may therefore be more useful to think of the size measure as an abstraction function and the inverted size measure as a concretization function in a Galois connection.

*Time bound program.* We will now compose the translated step counting version of the program with the inverted size measure. After simplifications based on constant propagation the result is the following program.

```
time (x) = t-reverse(length-1(x))
length-1 (x) =
     if eq(x,0) then nil
        else cons("all",length-1(sub(x,1))))
t-reverse (x) =
     if null(x) then 4
        else add(11,add(t-reverse(cdr(x))
                    t-append(reverse(cdr(x))) ))
t-append (x) =
    if null(x) then 4 else add(10,t-append(cdr(x)))
reverse (x) =
     if null(x) then nil
        else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
     if null(x) then y
        else cons(car(x),append(cdr(x),y))
```

This program computes an upper bound to the execution time of the reverse program based on the length of its input. We can always construct such time bound programs but the time bound program may fail to terminate for some input sizes even if the original program terminates for all input of the given size. We cannot in general expect to be able to transform time bound programs into simple closed form expressions. The driving technique may, however, for a number of programs simplify the time bound programs to closed form.

## 10 Driving Time Bound Programs

In the following example we will use the driving technique on the `time` function from the previous section.

To start with we try to solve difference equations in the program. It is convenient to do this before using the driving algorithm because we might otherwise solve the same difference equation many times in the algorithm. There is only one function in the program to be simplified in this way:

```
t-append (x) =  add(4,mul(10,length(x)))
```

where `length` is the usual length function from Lisp.

```
length(x) =  if null(x) then 0 else add(1,length(cdr(x)))
```

The initial basic configuration is `[t-reverse(length-1(_))]` with the expression

```
[t-reverse(length-1(_))] (x) =
    if eq(x,0) then 4
       else add(15,
            add(t-reverse(length-1(sub(x,1)))
              mul(10,length(reverse(length-1(sub(x,1)))) ))
```

Here we may notice that `[length(reverse(length-1(_)))]` is not a basic configuration in our approach. We analyze the expression for outside in so the next basic configuration to be analyzed is `[length(reverse(_))]`.

```
[length(reverse(_))] (x) =
   if null(x) then 0
      else length(append(reverse(cdr(x)),cons(car(x),nil) ))
```

giving the new configuration

```
[length(append(_))] (x,y) =
    if null(x) then length(y)
        else add(1,length(append(cdr(x),y)))
```

This difference equation can be solved giving

```
[length(append(_))] (x,y) =
    add(length(x),length(y))
```

Inserted in the context from the previous configuration we get

```
[length(reverse(_))] (x) =
    if null(x) then 0
       else add(1,length(reverse(cdr(x))))
```

This function is matched against previously defined functions in the program thus giving

```
[length(reverse(_))] (x) = length(x)
```

Inserted in the original context we get

```
[t-reverse(length-1(_))] (x) =
    if eq(x,0) then 4
       else add(15,
            add(t-reverse(length-1(sub(x,1)))
                mul(10,(length(length-1(sub(x,1))))) ))
```

where we get the new configuration

```
[length(length-1(_))] (x) =
    if eq(x,0) then 0
       else add(1,length(length-1(sub(x,1))))
```

This difference equation is easily solved to

```
[length(length-1(_))] (x) = x
```

The original configuration is now

```
[t-reverse(length-1(_))] (x) =
    if eq(x,0) then 4
       else add(15
            add(t-reverse(length-1(sub(x,1)))
                mul(10,sub(x,1)) ))
```

This difference equation can be solved to

```
[t-reverse(length-1(_))] (x) =
    add(4,add(mul(10,x),mul(5,mul(x,x)) ))
```

Finally the program can be reduced to

```
time (x) =  add(4,add(mul(10,x),mul(5,mul(x,x)) ))
```

This could also be written as

$$\texttt{time}\,(x) = 5x^2 + 10x + 4$$

## 11 Related Works

The idea of specializing functions with functions as arguments can be found as procedure-expressions in Scherlis [6] and in super-compilation and driving in Turchin [11]. Also Wegbreit [15] describes a way to improve function composition by matching sub-goals in unfolded expressions from more general goals. The works by Scherlis and Wegbreit describe transformation rules, but they cannot answer the question about in which order the rules should be applied. Super compilation gives a general answer to this question although the set of basic configurations is difficult to characterize.

Several authors have considered similar transformation systems which may remove intermediate data structures [12], [13]. Wadler's work on listlessness and deforestation may perform many of the same simplifications of programs as reported here. Our work is not, however, restricted to trees or lists and does not require branching to be done using pattern matching.

More recently Sørensen, Glück and Jones [9],[3],[8] have re-examined Turchin's work and defined the notion of positive super compilation. Their aim is more general than ours and they will capture most of the power of Turchin's supercompiler.

Driving and super compilation may be seen as a generalization of partial evaluation. If we used configurations of the form $[f_i(u_1, \ldots, u_{m_i})]$ where $u_k$ are either constants or the special symbol *var* then we would produce specialized version of programs when some arguments were known. In partial evaluation, however, the problem of when to generalize is not trivial and it would have to be implemented in the generalization strategy.

## 12 Conclusion

The present program transformation system is fairly easy to implement. The original version was implemented in Lisp, but we have since rewritten it in ML. The system is especially useful as a tool to simplify automatically generated programs. Such programs are not always just intended to be run by computer. If the intended recipient is a human then we may appreciate receiving it in a simplified version where unnecessary computations are removed.

The transformation system is guaranteed to terminate since there is only a finite number of basic configurations for any given program. There is a risk of code explosion since the number of basic configurations in theory is quite large but we have not been able to construct examples where that would occur. Non-trivial basic configurations are sparsely distributed in real programs.

Some of the reductions we use may remove non-terminating sub-expressions. Expressions of the form `car(cons(`$a$`,`$b$`))` may be reduced to $a$ even though the expression $b$ may contain non-terminating sub-expressions. Due to this the resulting program may terminate with input for which the original program would not terminate. It is not a very likely situation and will not be an issue if the original programs are total.

The driving technique does not impose any special requirements on the programs. If there are no non-trivial basic configurations in the program or if we cannot produce optimized versions for the configurations then no transformation will take place and we will just return the original program.

**Acknowledgement.** To Neil D. Jones for inspiration and encouragement through the years.

## References

1. R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J.ACM.* vol. 24, no 1, pp. 44 - 67, Jan., 1977.
2. J. Cohen and J. Katcoff. Symbolic Solution of Finite-Difference Equations. *Transactions on Mathematical Software.* vol. 3, no. 3, pp. 261 - 271, 1977.
3. N. D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. *Logic, Language, and Computation.* LNCS vol 792, pp. 206-224, Springer-Verlag, 1994
4. M. Rosendahl. Automatic Program Analysis. Master's thesis, Department of Computer Science, University of Copenhagen, 1986
5. M. Rosendahl. Automatic Complexity Analysis. *FPCA'89, London, England*, ACM Press, pp. 144–156, 1989.
6. W. L. Scherlis. Program Improvement by Internal Specialization. *8'th Symposium on Principles of Programming Language.* ACM, pp. 41-49, Jan., 1981.
7. Peter Sestoft. The stucture of a Self-applicable Partial Evaluator *Programs as Data Objects* LNCS, vol 217, pp. 236 - 256, 1986
8. M. H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1990
9. M. H. Sørensen, R. Glück. Introduction to Supercompilation *Partial Evaluation: Practice and Theory,* LNCS, vol. 1706, pp. 246–270, 1999
10. V. F. Turchin, R. M. Nirenburg, D. V. Turchin. Experiments with a Supercompiler. *ACM Symposium on Lisp and Functional Programming.* Pittsburgh PA, August, 1982.
11. V. F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS.* vol. 8, no. 3, pp. 292-325, July, 1986.
12. P. Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection on Compile-time. *ACM Symposium on Lisp and Functional Programming.* Austin Texas, August, 1984.

13. P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science,* vol. 73, pp. 231-248, 1990
14. B. Wegbreit. Mechanical Program Analysis. *C.ACM.* vol. 18, pp. 528 - 539, Sept., 1975.
15. B. Wegbreit. Goal-Directed Program Transformation. *IEEE Transaction on Software Engineering,* vol SE-2, no 2, pp. 69 - 80, June, 1976
16. B. Wegbreit. Verifying Program Performance. *J.ACM.* Vol. 23,No. 4, pp. 691 - 699, October, 1976

# Demonstrating Lambda Calculus Reduction

Peter Sestoft

Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark
and
IT University of Copenhagen, Denmark
sestoft@dina.kvl.dk

**Abstract.** We describe lambda calculus reduction strategies, such as call-by-value, call-by-name, normal order, and applicative order, using big-step operational semantics. We show how to simply and efficiently trace such reductions, and use this in a web-based lambda calculus reducer available at ⟨http://www.dina.kvl.dk/~sestoft/lamreduce/⟩.

## 1 Introduction

The pure untyped lambda calculus is often taught as part of the computer science curriculum. It may be taught in a computability course as a classical computation model. It may be taught in a semantics course as the foundation for denotational semantics. It may be taught in a functional programming course as the archetypical minimal functional programming language. It may be taught in a programming language course for the same reason, or to demonstrate that a very small language can be universal, e.g. can encode arithmetics (as well as data structures, recursive function definitions and so on), using encodings such as these:

$$\begin{aligned} \mathit{two} &\equiv \lambda f.\lambda x.f(fx) \\ \mathit{four} &\equiv \lambda f.\lambda x.f(f(f(fx))) \\ \mathit{add} &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \end{aligned} \tag{1}$$

This paper is motivated by the assumption that to appreciate the operational aspects of pure untyped lambda calculus, students must experiment with it, and that tools encourage experimentation with encodings and reduction strategies by making it less tedious and more fun.

In this paper we describe a simple way to create a tool for demonstrating lambda calculus reduction. Instead of describing a reduction strategy by a procedure for locating the next redex to be contracted, we describe it by a big-step operational semantics. We show how to trace the $\beta$-reductions performed during reduction.

To do this we also precisely define and clarify the relation between programming language concepts such as call-by-name and call-by-value, and lambda calculus concepts such as normal order reduction and applicative order reduction. These have been given a number of different interpretations in the literature.

## 2 Motivation and Related Work

Much has been written about the lambda calculus since Church developed it as a foundation for mathematics [6]. Landin defined the semantics of programming languages in terms of the lambda calculus [11], and gave a call-by-value interpreter for it: the SECD-machine [10]. Strachey used lambda calculus as a meta-language for denotational semantics, and Scott gave models for the pure untyped lambda calculus, making sure that self-application could be assigned a meaning; see Stoy [22]. Self-application $(x\,x)$ of a term $x$ is used when encoding recursion, for instance in Church's $Y$ combinator:

$$Y \equiv \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \tag{2}$$

Plotkin studied the call-by-value lambda calculus corresponding to the functional language ISWIM [12] implemented by Landin's SECD-machine, and also a related call-by-name lambda calculus, and observed that one characteristic of a functional programming language was the absence of reduction under lambda abstractions [19].

Barendregt [4] is the standard reference on the untyped lambda calculus, with emphasis on models and proof theory, not programming languages.

Many textbooks on functional programming or denotational semantics present the pure untyped lambda calculus, show how to encode numbers and algebraic data types, and define evaluators for it. One example is Paulson's ML textbook [16], which gives interpreters for call-by-name as well as call-by-value.

So is there really a need for yet another paper on lambda calculus reduction? We do think so, because it is customary to look at the lambda calculus either from the programming language side or from the calculus or model side, leaving the relations between the sides somewhat unclear.

For example, Plotkin [19] defines call-by-value reduction as well as call-by-name reduction, but the call-by-name rules take free variables into account only to a limited extent. By the rules, $x\,((\lambda z.z)\,v)$ reduces to $x\,v$, but $(x\,y)\,((\lambda z.z)\,v)$ does not reduce to $x\,y\,v$ [19, page 146]. Similarly, the call-by-value strategy described by Felleisen and Hieb using evaluation contexts [8, Section 2] would not reduce $(x\,y)\,((\lambda z.z)\,v)$ to $x\,y\,v$, since there is no evaluation context of the form $(x\,y)\,[\,]$. This is unproblematic because, following Landin, these researchers were interested only in terms with no free variables, and in reduction only outside lambda abstractions.

But it means that the reduction rules are not immediately useful for terms that have free variables, and therefore not useful for experimentation with the terms that result from encoding programming language constructs in the pure lambda calculus.

Conversely, Paulson [16] presents call-by-value and call-by-name interpreters for the pure lambda calculus that do handle free variables. However, they also perform reduction under lambda abstractions (unlike functional programming languages), and the evaluation order is not leftmost outermost: under call-by-name, an application $(e_1\,e_2)$ is reduced by first reducing $e_1$ to head normal form, so redexes inside $e_1$ may be contracted before an enclosing leftmost redex. This makes the relation between Paulson's call-by-name and normal order (leftmost outermost) reduction strategies somewhat unclear.

Therefore we find that it may be useful to contrast the various reduction strategies, present them using big-step operational semantics, present their (naive) implementation in ML, and show how to obtain a trace of the reduction.

## 3 The Pure Untyped Lambda Calculus

We use the pure untyped lambda calculus [4]. A lambda term is a *variable* $x$, a lambda *abstraction* $\lambda x.e$ which binds $x$ in $e$, or an *application* $(e_1\, e_2)$ of a 'function' $e_1$ to an 'argument' $e_2$:

$$e ::= x \mid \lambda x.e \mid e_1\, e_2 \tag{3}$$

Application associates to the left, so $(e_1\, e_2\, e_3)$ means $((e_1\, e_2)\, e_3)$. A lambda term may have free variables, not bound by any enclosing lambda abstraction. Term identity $e_1 \equiv e_2$ is taken modulo renaming of lambda-bound variables. The notation $e[e_x/x]$ denotes substitution of $e_x$ for $x$ in $e$, with renaming of bound variables in $e$ if necessary to avoid capture of free variables in $e_x$.

A *redex* is a subterm of the form $((\lambda x.e)\, e_2)$; the *contraction* of a redex produces $e[e_2/x]$, substituting the argument $e_2$ for every occurrence of the parameter $x$ in $e$. By $e \longrightarrow_\beta e'$ we denote *$\beta$-reduction*, the contraction of some redex in $e$ to obtain $e'$.

A redex is to the *left* of another redex if its lambda abstractor appears further to the left. The *leftmost outermost* redex (if any) is the leftmost redex not contained in any other redex. The *leftmost innermost* redex (if any) is the leftmost redex not containing any other redex.

## 4 Functional Programming Languages

In practical functional programming languages such as Scheme [20], Standard ML [14] or Haskell [18], programs cannot have free variables, and reductions are not performed under lambda abstractions or other variable binders, because this would considerably complicate their efficient implementation [17].

However, an implementation of lambda calculus reduction must perform reductions under lambda abstractions. Otherwise, *add two two* would not reduce to *four* using the encodings (1), which would disappoint students.

Because free variables and reduction under abstraction are absent in functional languages, it is unclear what the programming language concepts call-by-value and call-by-name mean in the lambda calculus. In particular, how should free variables be handled, and to what normal form should call-by-value and call-by-name evaluate? We propose the following answers:

– A free variable is similar to a data constructor (in Standard ML or Haskell), that is, an uninterpreted function symbol. If the free variable $x$ is in function position $(x\, e_2)$, then call-by-value should reduce the argument expression $e_2$, whereas call-by-name should not. This is consistent with constructors being strict in strict languages (e.g. ML) and non-strict in non-strict languages (e.g. Haskell).
– Functional languages perform no reduction under abstractions, and thus reduce terms to weak normal forms only. In particular, call-by-value reduces to weak normal form, and call-by-name reduces to weak head normal form. Section 6 define these normal forms.

## 5 Lazy Functional Programming Languages

Under lazy evaluation, a variable-bound term is evaluated at most once, regardless how often the variable is used [17]. Thus an argument term may not be duplicated before it has been reduced, and may be reduced only if actually used. This evaluation mechanism may be called call-by-need, or call-by-name with sharing of argument evaluation. The equational theory of call-by-need lambda calculus has been studied by Ariola and Felleisen [2] among others. (By contrast, the lazy lambda calculus of Abramsky and Ong [1] is not lazy in the sense discussed here; rather, it is the theory of call-by-name lambda calculus, without reduction under abstractions.)

Lazy functional languages also permit the creation of cyclic terms, or cycles in the heap. For instance, this declaration creates a finite (cyclic) representation of an infinite list of 1's:

```
val ones = 1 :: ones
```

Thus to be true also to the intensional properties of lazy languages (such as time and space consumption), a model should be able to describe such constant-size cyclic structures. Substitution of terms for variables cannot truly model them, only approximate them by unfolding of a recursive term definition, possibly encoded using a recursion combinator such as (2). To properly express sharing of subterm evaluation, and the creation of cyclic terms, one must extend the syntax (3) with mutually recursive bindings:

$$e ::= \; x \mid \lambda x.e \mid e\, e \mid \mathit{letrec}\ \{x_i = e_i\}\ \mathit{in}\ e \tag{4}$$

The sharing of subterm evaluation and the dynamic creation of cyclic terms may be modelled using graph reduction, as suggested by Wadsworth [24] and used in subsequent work [3,17,23], or using an explicit heap [13,21].

Thus a proper modelling of lazy evaluation, with sharing of argument evaluation and cyclic data structures, requires syntactic extensions as well as a more elaborate evaluation model than just term rewriting. We shall not consider lazy evaluation any further in this paper, and shall consider only the syntax in (3) above.

## 6 Normal Forms

We need to distinguish four different normal forms, depending on whether we reduce under abstractions or not (in functional programming languages), and depending on whether we reduce the arguments before substitution (in strict languages) or not (in non-strict languages).

Figure 1 summarizes the four normal forms using four context-free grammars. In each grammar, the symbol $E$ denotes a term in the relevant normal form, $e$ denotes an arbitrary lambda term generated by (3), and $n \geq 0$. Note how the two dichotomies generate the four normal forms just by varying the form of lambda abstraction bodies and application arguments.

| | **Reduce under abstractions** | |
|---|---|---|
| **Reduce args** | **Yes** | **No** |
| **Yes** | Normal form<br>$E ::= \lambda x.E \mid x\, E_1 \ldots E_n$ | Weak normal form<br>$E ::= \lambda x.e \mid x\, E_1 \ldots E_n$ |
| **No** | Head normal form<br>$E ::= \lambda x.E \mid x\, e_1 \ldots e_n$ | Weak head normal form<br>$E ::= \lambda x.e \mid x\, e_1 \ldots e_n$ |

**Fig. 1.** Normal forms. The $e_i$ denote arbitrary lambda terms generated by (3).

## 7 Reduction Strategies and Reduction Functions

We present several reduction strategies using big-step operational semantics, or natural semantics [9], and their implementation in Standard ML. The premises of each semantic rule are assumed to be evaluated from left to right, although this is immaterial to their logical interpretation. We exploit that Standard ML evaluates a function's arguments before calling the function, evaluates the right-hand side of `let`-bindings before binding the variable, and evaluates subterms from left to right [14].

We model lambda terms $x$, $\lambda x.e$ and $(e\, e)$ as ML constructed data, representing variable names by strings:

```
datatype lam = Var of string
             | Lam of string * lam
             | App of lam * lam
```

We also assume an auxiliary function `subst : lam -> lam -> lam` that implements capture-free substitution, so `subst ex (Lam(x, e))` is the ML representation of $e[e_x/x]$, the result of contracting the redex $(\lambda x.e)\, e_x$.

### 7.1 Call-by-Name Reduction to Weak Head Normal Form

Call-by-name reduction $e \xrightarrow{bn} e'$ reduces the leftmost outermost redex not inside a lambda abstraction first. It treats free variables as non-strict data constructors.

For terms without free variables, it coincides with Plotkin's call-by-name reduction [19, Section 5], and is closely related to Engelfriet and Schmidt's outside-in derivation (in context-free tree grammars, or first-order recursion equations) [7, page 334].

$$x \xrightarrow{bn} x$$

$$(\lambda x.e) \xrightarrow{bn} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn} e'}{(e_1\ e_2) \xrightarrow{bn} e'} \tag{5}$$

$$\frac{e_1 \xrightarrow{bn} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn} (e_1'\ e_2)}$$

It is easy to see that all four rules generate terms in weak head normal form. In particular, in the last rule $e_1'$ must have form $y\, e_{11}' \ldots e_{1n}'$ for some $n \geq 0$, so $(e_1'\, e_2)$ is a weak head normal form. Assuming that the rule premises are read and 'executed' from left to right, it is also clear that only leftmost redexes are contracted. No reduction is performed under abstractions.

The following ML function `cbn` computes the weak head normal form of a lambda term, contracting redexes in the order implied by the operational semantics (5) above:

```
fun cbn (Var x)         = Var x
  | cbn (Lam(x, e))     = Lam(x, e)
  | cbn (App(e1, e2))   =
    case cbn e1 of
        Lam (x, e) => cbn (subst e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

The first function clause above handles variables $x$ and implements the first semantics rule. Similarly, the second function clause handles lambda abstractions $(\lambda x.e)$ and implements the second semantics rule. In both cases, the given term is returned unmodified. The third function clause handles applications $(e_1\, e_2)$ and implements the third and fourth semantics rule by discriminating on the result of reducing $e_1$. If the result is a lambda abstraction $(\lambda x.e)$ then the `cbn` function is called to reduce the expression $e[e_2/x]$; but if the result is any other expression $e_1'$, the application $(e_1'\, e_2)$ is returned.

In all cases, this is exactly what the semantics rules in (5) describe. In fact, one can see that $e \xrightarrow{bn} e'$ if and only if `cbn` $e$ terminates and returns $e'$.

## 7.2 Normal Order Reduction to Normal Form

Normal order reduction $e \xrightarrow{no} e'$ reduces the leftmost outermost redex first. In an application $(e_1\, e_2)$ the function term $e_1$ must be reduced using call-by-name (5).

Namely, if $e_1$ reduces to an abstraction $(\lambda x.e)$, then the redex $((\lambda x.e)\, e_2)$ must be reduced before redexes in $e$, if any, because they would not be outermost.

$$
\begin{array}{c}
x \xrightarrow{no} x \\[2ex]
\dfrac{e \xrightarrow{no} e'}{(\lambda x.e) \xrightarrow{no} (\lambda x.e')} \\[2ex]
\dfrac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{no} e'}{(e_1\ e_2) \xrightarrow{no} e'} \\[2ex]
\dfrac{e_1 \xrightarrow{bn} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{no} e_1'' \qquad e_2 \xrightarrow{no} e_2'}{(e_1\ e_2) \xrightarrow{no} (e_1''\ e_2')}
\end{array}
\tag{6}
$$

It is easy to see that these rules generate normal form terms only. In particular, in the last rule $e_1'$ must have form $y\, e_{11}' \, \dots \, e_{1n}'$ for some $n \geq 0$, so $e_1''$ must have form $y\, E_{11}'' \, \dots \, E_{1n}''$ for some normal forms $E_{1i}''$, and therefore $(e_1''\, e_2')$ is a normal form. Any redex contracted is the leftmost one not contained in any other redex; this relies on the use of call-by-name in the application rules. Reductions are performed also under lambda abstractions. Normal order reduction is *normalizing*: if the term $e$ has a normal form, then normal order reduction of $e$ will terminate (with the normal form as result).

The Standard ML function `nor : lam -> lam` below implements the reduction strategy. Note that it uses the function `cbn` defined in Section 7.1:

```
fun nor (Var x)       = Var x
  | nor (Lam (x, e))  = Lam(x, nor e)
  | nor (App(e1, e2)) =
    case cbn e1 of
        Lam(x, e) => nor (subst e2 (Lam(x, e)))
      | e1'       => let val e1'' = nor e1'
                     in App(e1'', nor e2) end
```

Again the first two cases of the function implement the first two reduction rules. The third case implements the third and fourth rules by evaluating $e_1$ using call-by-name `cbn` and then discriminating on whether the result is a lambda abstraction or not, as in the third and fourth rule in (6).

## 7.3 Call-by-Value Reduction to Weak Normal Form

Call-by-value reduction $e \xrightarrow{bv} e'$ reduces the leftmost innermost redex not inside a lambda abstraction first. It treats free variables as strict data constructors. For terms without free variables, it coincides with call-by-value reduction as defined by Plotkin [19, Section 4] and Felleisen and Hieb [8]. It is closely related to

Engelfriet and Schmidt's inside-out derivations (in context-free tree grammars, or first-order recursion equations) [7, page 334]. It differs from call-by-name (Section 7.1) only by reducing the argument $e_2$ of an application $(e_1\, e_2)$ before contracting the redex, and before building an application term:

$$x \xrightarrow{bv} x$$

$$(\lambda x.e) \xrightarrow{bv} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2' \qquad e[e_2'/x] \xrightarrow{bv} e'}{(e_1\ e_2) \xrightarrow{bv} e'} \tag{7}$$

$$\frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2'}{(e_1\ e_2) \xrightarrow{bv} (e_1'\ e_2')}$$

It is easy to see that these rules generate weak normal form terms only. In particular, in the last rule $e_1'$ must have form $y\, E_{11}' \ldots E_{1n}'$ for some $n \geq 0$ and weak normal forms $E_{1i}'$, and therefore $(e_1'\, e_2')$ is a weak normal form too. No reductions are performed under lambda abstractions. This is Paulson's `eval` auxiliary function [16, page 390]. The implementation of the rules by an ML function is straightforward and is omitted.

## 7.4 Applicative Order Reduction to Normal Form

Applicative order reduction $e \xrightarrow{ao} e'$ reduces the leftmost innermost redex first. It differs from call-by-value (Section 7.3) only by reducing also under abstractions:

$$x \xrightarrow{ao} x$$

$$\frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{ao} (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2' \qquad e[e_2'/x] \xrightarrow{ao} e'}{(e_1\ e_2) \xrightarrow{ao} e'} \tag{8}$$

$$\frac{e_1 \xrightarrow{ao} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2'}{(e_1\ e_2) \xrightarrow{ao} (e_1'\ e_2')}$$

It is easy to see that the rules generate only normal form terms. As before, note that in the last rule $e_1'$ must have form $y\, E_{11}' \ldots E_{1n}'$ for some $n \geq 0$ and normal forms $E_{1i}'$. Also, it is clear that when a redex $((\lambda x.e)\, e_2')$ is contracted, it contains no other redex, and it is the leftmost redex with this property.

Applicative order reduction is not normalizing; with $\Omega \equiv (\lambda x.(x\,x))(\lambda x.(x\,x))$ it produces an infinite reduction $((\lambda x.y)\,\Omega) \longrightarrow_\beta ((\lambda x.y)\,\Omega) \longrightarrow_\beta \ldots$ although the term has normal form $y$.

In fact, applicative order reduction fails to normalize applications of functions defined using recursion combinators, even with recursion combinators designed for call-by-value, such as $Y_v$:

$$Y_v \equiv \lambda h.(\lambda x.\lambda a.h\,(x\,x)\,a)\,(\lambda x.\lambda a.h\,(x\,x)\,a) \tag{9}$$

### 7.5 Hybrid Applicative Order Reduction to Normal Form

Hybrid applicative order reduction is a hybrid of call-by-value and applicative order reduction. It reduces to normal form, but reduces under lambda abstractions only in argument positions. Therefore the usual call-by-value versions of the recursion combinator, such as (9) above, may be used with this reduction strategy. Thus the hybrid applicative order strategy normalizes more terms than applicative order reduction, while using fewer reduction steps than normal order reduction. The hybrid applicative order strategy relates to call-by-value in the same way that the normal order strategy relates to call-by-name. It resembles Paulson's call-by-value strategy, which works in two phases: first reduce the term by $\xrightarrow{bv}$ , then normalize the bodies of any remaining lambda abstractions [16, page 391].

$$\begin{gathered}
x \xrightarrow{ha} x \\[1em]
\frac{e \xrightarrow{ha} e'}{(\lambda x.e) \xrightarrow{ha} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{ha} e_2' \qquad e[e_2'/x] \xrightarrow{ha} e'}{(e_1\ e_2) \xrightarrow{ha} e'} \\[1em]
\frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{ha} e_1'' \qquad e_2 \xrightarrow{ha} e_2'}{(e_1\ e_2) \xrightarrow{ha} (e_1''\ e_2')}
\end{gathered} \tag{10}$$

### 7.6 Head Spine Reduction to Head Normal Form

The head spine strategy performs reductions inside lambda abstractions, but only in head position. This is the reduction strategy implemented by Paulson's `headNF` function [16, page 390].

$$
\begin{gathered}
x \xrightarrow{he} x \\[1em]
\frac{e \xrightarrow{he} e'}{(\lambda x.e) \xrightarrow{he} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{he} e'}{(e_1\ e_2) \xrightarrow{he} e'} \\[1em]
\frac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e)}{(e_1\ e_2) \xrightarrow{he} (e_1'\ e_2)}
\end{gathered}
\tag{11}
$$

It is easy to see that the rules generate only head normal form terms. Note that this is not a *head reduction* as defined by Barendregt [4, Definition 8.3.10]: In a (leftmost) head reduction only head redexes are contracted, where a redex $((\lambda x.e_0)\, e_1)$ is a *head redex* if it is preceded to the left only by lambda abstractors of non-redexes, as in $\lambda x_1. \ldots \lambda x_n.(\lambda x.e_0)\, e_1\ \ldots\ e_m$, with $n \geq 0$ and $m \geq 1$.

To define head reduction, one should use $e_1 \xrightarrow{bn} e_1'$ in the above application rules (11) to avoid premature reduction of inner redexes, similar to the use of $\xrightarrow{bn}$ in the definition of $\xrightarrow{no}$.

## 7.7 Hybrid Normal Order Reduction to Normal Form

Hybrid normal order reduction is a hybrid of head spine reduction and normal order reduction. It differs from normal order reduction only by reducing the function $e_1$ in an application to head normal form (by $\xrightarrow{he}$) instead of weak head normal form (by $\xrightarrow{bn}$) before applying it to the argument $e_2$.

The hybrid normal order strategy resembles Paulson's call-by-name strategy, which works in two phases: first reduce the term by $\xrightarrow{he}$ to head normal form, then normalize unevaluated arguments and bodies of any remaining lambda abstractions [16, page 391].

$$
\begin{gathered}
x \xrightarrow{hn} x \\[1em]
\frac{e \xrightarrow{hn} e'}{(\lambda x.e) \xrightarrow{hn} (\lambda x.e')} \\[1em]
\frac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{hn} e'}{(e_1\ e_2) \xrightarrow{hn} e'} \\[1em]
\frac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{hn} e_1'' \qquad e_2 \xrightarrow{hn} e_2'}{(e_1\ e_2) \xrightarrow{hn} (e_1''\ e_2')}
\end{gathered}
\tag{12}
$$

These rules generate normal form terms only. The strategy is normalizing, because if the term $(e_1\, e_2)$ has a normal form, then it has a head normal form, and then so has $e_1$ [4, Proposition 8.3.13].

## 8 Properties of the Reduction Strategies

The relation defined by each reduction strategy is idempotent. For instance, if $e \xrightarrow{bn} e'$ then $e' \xrightarrow{bn} e'$. To see this, observe that $e'$ is in weak head normal form, so it has form $\lambda x.e''$ or $x\, e_1 \ldots e_n$, where $e''$ and $e_1, \ldots, e_n$ are arbitrary lambda terms. In the first case, $e'$ reduces to itself by the second rule of (5). In the second case, an induction on $n$ shows that $e'$ reduces to itself by the first and third rule of (5). Similar arguments can be made for the other reduction strategies.

Figure 2 classifies the seven reduction strategies presented in Sections 7.1 to 7.7 according the normal forms (Figure 1) they produce.

| | **Reduce under abstractions** | |
|---|---|---|
| **Reduce args** | **Yes** | **No** |
| **Yes** | Normal form<br>**ao**, *no*, *ha*, *ho* | Weak normal form<br>**bv** |
| **No** | Head normal form<br>**he** | Weak head normal form<br>**bn** |

**Fig. 2.** Classification of reduction strategies by the normal forms they produce. The 'uniform' reduction strategies are shown in boldface, the 'hybrid' ones in italics.

Inspection of the big-step semantics rules shows that four of the reduction strategies (**ao**, **bn**, **bv**, **he**, shown in bold in Figure 2) are 'uniform': their definition involves only that reduction strategy itself. The remaining three (*no*, *ha*, *hn*) are 'hybrid': each uses one of the 'uniform' strategies for the reduction of the expression $e_1$ in function position in applications $(e_1\, e_2)$. Figure 3 shows how the 'hybrid' and 'uniform' strategies are related.

| **Hybrid** | **Uniform** |
|---|---|
| *no* | **bn** |
| *ha* | **bv** |
| *hn* | **he** |

**Fig. 3.** Derivation of hybrid strategies from uniform ones.

## 9 Tracing: Side-Effecting Substitution, and Contexts

The reducers defined in ML in Section 7 perform the substitutions $e[e_2/x]$ in the same order as prescribed by the operational semantics, thanks to Standard ML semantics: strict evaluation and left-to-right evaluation. But they only return the final reduced lambda term; they do not trace the intermediate steps of the reduction, which is often more interesting from a pedagogical point of view.

ML permits expressions to have side effects, so we can make the substitution function report (e.g. print) the redex just before contracting it. To do this we define a modified substitution function `csubst` which takes as argument another function `c` and applies it to the redex `App(Lam(x, e), ex)` representing $(\lambda x.e)\, e_x$, just before contracting it:

```
fun csubst (c : lam -> unit) ex (Lam(x, e)) =
    (c (App(Lam(x, e), ex));
     subst ex (Lam(x, e)))
```

The function `c : lam -> unit` is evaluated for its side effect only, as shown by the trivial result type `unit`. Evaluating `csubst c ex (Lam(x, e))` has the *effect* of calling `c` on the redex $((\lambda x.e)\, e_x)$, and its *result* is the result of the substitution $e[e_x/x]$, which is the contracted redex.

We could define a function `printlam : lam -> unit` that prints the given lambda term as a side effect. Then replacing the call `subst e2 (Lam(x, e))` in function `cbn` of Section 7.1 by `csubst printlam e2 (Lam(x, e))` will cause the reduction of a term by `cbn` to produce a printed trace of all redexes $((\lambda x.e)\, e_x)$, in the order in which they are contracted.

This still does not give us a usable trace of the evaluation: we do not know where in the current term the printed redex occurs. This is because the function `printlam` is applied only to the redex itself; the term surrounding the redex is implicit. To make the term surrounding the redex explicit, we can use a *context*, a term with a single hole, such as $\lambda x.[\,]$ or $(e_1\, [\,])$ or $([\,]\, e_2)$, where the hole is denoted by $[\,]$. Filling the hole of a context with a lambda term produces a lambda term. The following grammar generates all single-hole contexts:

$$C \ ::= \ [\,] \mid \lambda x.C \mid e\, C \mid C\, e \tag{13}$$

A context can be represented by an ML function of type `lam -> lam`. The four forms of contexts (13) can be created using four ML context-building functions:

```
fun id      e  = e
fun Lamx x  e  = Lam(x, e)
fun App2 e1 e2 = App(e1, e2)
fun App1 e2 e1 = App(e1, e2)
```

For instance, `(App1` $e_2$`)` is the ML function `fn e1 => App(e1, e2)` which represents the context $([\,]\, e_2)$. Filling the hole with the term $e_1$ is done by computing `(App1` $e_2$`)` $e_1$ which evaluates to `App(`$e_1$`,` $e_2$`)`, representing the term $(e_1\, e_2)$.

Function composition (`f o g`) composes contexts. For instance, the composition of contexts $\lambda x.[\,]$ and $([\,]\, e_2)$ is `Lamx` $x$ `o` `App1` $e_2$, which represents the context $\lambda x.([\,]\, e_2)$. Similarly, the composition of the contexts $([\,]\, e_2)$ and $\lambda x.[\,]$ is `App1` $e_2$ `o` `Lamx` $x$, which represents $((\lambda x.[\,])\, e_2)$.

## 10 Reduction in Context

To produce a trace of the reduction, we modify the reduction functions defined in Section 7 to take an extra context argument `c` and to use the extended substitution function `csubst`, passing `c` to `csubst`. Then `csubst` will apply `c` to the redex before contracting it. We take the call-by-name reduction function `cbn` (Section 7.1) as an example; the other reduction functions are handled similarly. The reduction function must build up the context `c` as it descends into the term. It does so by composing the context with the appropriate context builder (in this case, only in the `App` branch):

```
fun cbnc c (Var x)        = Var x
  | cbnc c (Lam(x, e))    = Lam(x, e)
  | cbnc c (App(e1, e2)) =
    case cbnc (c o App1 e2) e1 of
        Lam (x, e) => cbnc c (csubst c e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

By construction, if $c$ `: lam -> lam` and the evaluation of `cbnc` $c$ $e$ involves a call `cbnc` $c'$ $e'$, then $c[e] \longrightarrow_{\beta}^{*} c'[e']$. Also, whenever a call `cbnc` $c'$ $(e_1\, e_2)$ is evaluated, and $e_1 \xrightarrow{bn} (\lambda x.e)$, then function $c'$ is applied to the redex $((\lambda x.e)\, e_2)$ just before it is contracted. Hence a trace of the reduction of term `e` can be obtained just by calling `cbnc` as follows:

```
cbnc printlam e
```

where `printlam : lam -> unit` is a function that prints the lambda term as a side effect. In fact, computing `cbnc printlam (App (App` *add two*`)` *two*`)`, using the encodings from (1), prints the two intermediate terms below. The third term shown is the final result (a weak head normal form):

```
(\m.\n.\f.\x.m f (n f x)) (\f.\x.f (f x)) (\f.\x.f (f x))
(\n.\f.\x.(\f.\x.f (f x)) f (n f x)) (\f.\x.f (f x))
\f.\x.(\f.\x.f (f x)) f ((\f.\x.f (f x)) f x)
```

The trace of a reduction can be defined also by direct instrumentation of the operational semantics (5). Let us define a *trace* to be a finite sequence of lambda terms, denote the empty trace by $\epsilon$, and denote the concatenation of traces $s$ and $t$ by $s \cdot t$. Now we define the relation $e \overset{s}{\xrightarrow{bn}}_{C} e'$ to mean: under call-by-name, the expression $e$ reduces to $e'$, and if $e$ appears in context $C$, then $s$ is the trace of the reduction. The trace $s$ will be empty if no redex was contracted in the reduction. If some redex was contracted, the first term in the trace will be $e$.

The tracing relation corresponding to call-by-name reduction (5) can be defined as shown below:

$$x \overset{bn}{\longrightarrow}{}_C^{\,\epsilon}\; x$$

$$(\lambda x.e) \overset{bn}{\longrightarrow}{}_C^{\,\epsilon}\; (\lambda x.e)$$

$$\frac{e_1 \overset{bn}{\longrightarrow}{}_{C[[\,]\,e_2]}^{\,s}\; (\lambda x.e) \qquad e[e_2/x] \overset{bn}{\longrightarrow}{}_C^{\,t}\; e'}{(e_1\ e_2) \overset{bn}{\longrightarrow}{}_C^{\,s\cdot C[(\lambda x.e)\,e_2]\cdot t}\; e'} \tag{14}$$

$$\frac{e_1 \overset{bn}{\longrightarrow}{}_{C[[\,]\,e_2]}^{\,s}\; e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \overset{bn}{\longrightarrow}{}_C^{\,s}\; (e_1'\ e_2)}$$

Thus reduction of a variable $x$ or a lambda abstraction $(\lambda x.e)$ produces the empty trace $\epsilon$. When $e_1$ reduces to a lambda abstraction, reduction of the application $(e_1\, e_2)$ produces the trace $s \cdot C[(\lambda x.e)\, e_2] \cdot t$, where $s$ traces the reduction of $e_1$ and $t$ traces the reduction of the contracted redex $e[e_2/x]$.

Tracing versions of the other reduction strategies can be defined analogously.

## 11 Single-Stepping Reduction

For experimentation it is useful to be able to perform one beta-reduction at a time, or in other words, to single-step the reduction. Again, this can be achieved using side effects in the implementation language. We simply make the context function `c` count the number of redexes contracted (substitutions performed), and set a step limit $N$ before evaluation is started.

When $N$ redexes have been contracted, `c` aborts the reduction by raising an exception `Enough` $e'$, which carries as its argument the term $e'$ that had been obtained after $N$ reductions. An enclosing exception handler returns $e'$ as the result of the reduction. The next invocation of the reduction function simply sets the step limit $N$ one higher, and so on. Thus the reduction of the original term starts over for every new step, but we create the illusion of reducing the term one step at a time.

The main drawback of this approach is that the total time spent performing $n$ steps of reduction is $O(n^2)$. In practice, this does not matter: noboby wants to single-step very long computations.

## 12 A Web-Based Interface to the Reduction Functions

A web-based interface to the tracing reduction functions can be implemented as an ordinary CGI script. The lambda term to be reduced, the name of the desired reduction strategy, the kind of computation (tracing, single-stepping, etc.) and the step limit are passed as parameters to the script.

Such an implementation has been written in Moscow ML [15] and is available at ⟨http://www.dina.kvl.dk/~sestoft/lamreduce/⟩. The implementation uses the `Mosmlcgi` library to access CGI parameters, and the `Msp` library for efficient structured generation of HTML code.

For tracing, the script uses a function `htmllam : lam -> unit` that prints a lambda term as HTML code, which is then sent to the browser by the web server. Calling `cbnc` (or any other tracing reduction function) with `htmllam` as argument will display a trace of the reduction in the browser.

A trick is used to make the next redex into a hyperlink in the browser. The implementation's representation of lambda terms is extended with labelled subterms, and `csubst` attaches labels $0, 1, \ldots$ to redexes in the order in which they are contracted. When single-stepping a reduction, the last labelled redex inside the term can be formatted as a hyperlink. Clicking on the hyperlink will call the CGI script again to perform one more step of reduction, creating the illusion of single-stepping the reduction as explained above.

## 13 Conclusion

We have described a simple way to implement lambda calculus reduction, describing reduction strategies using big-step operational semantics, implementing reduction by straightforward reduction functions in Standard ML, and instrumenting them to produce a trace of the reduction, using contexts. This approach is easily extended to other reduction strategies describable by big-step operational semantics.

We find that big-step semantics provides a clear presentation of the reduction strategies, highlighting their differences and making it easy to see what normal forms they produce.

The extension to lazy evaluation, whether using graph reduction or an explicit heap, would be complicated mostly by the need to represent the current term graph or heap, and to print it in a comprehensible way.

The functions for reduction in context were useful for creating a web interface also, running the reduction functions as a CGI script written in ML. The web interface provides a simple platform for students' experiments with lambda calculus encodings and reduction strategies.

*Acknowledgements* Thanks for Dave Schmidt for helpful comments that have improved contents as well as presentation.

## References

1. Abramsky, S., Ong, C.-H.L.: Full Abstraction in the Lazy $\lambda$-Calculus. Information and Computation **105**, 2 (1993) 159–268.

2. Ariola, Z.M., Felleisen, M.: The Call-by-Need Lambda Calculus. Journal of Functional Programming **7**, 3 (1997) 265–301.
3. Augustsson, L.: A Compiler for Lazy ML. In: 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas. ACM Press (1984) 218–227.
4. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland (1984).
5. Barendregt, H.P. *et al.*: Needed Reduction and Spine Strategies for the Lambda Calculus. Information and Computation **75** (1987) 191–231.
6. Church, A.: A Note on the Entscheidungsproblem. Journal of Symbolic Logic **1** (1936) 40–41, 101–102.
7. Engelfriet, J., Schmidt, E.M.: IO and OI. Journal of Computer and System Sciences **15** (1977) 328–353 and **16** (1978) 67–99.
8. Felleisen, M., Hieb, R.: The Revised Report on the Syntactic Theories of Sequential Control and State. Theoretical Computer Science **103**, 2 (1992) 235–271.
9. Kahn, G.: Natural Semantics. In: STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany. Lecture Notes in Computer Science, Vol. 247. Springer-Verlag (1987) 22–39.
10. Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal **6**, 4 (1964) 308–320.
11. Landin, P.J.: A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. Communications of the ACM **8**, 2 (1965) 89–101.
12. Landin, P.J.: The Next 700 Programming Languages. Communications of the ACM **9**, 3 (1966) 157–166.
13. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. ACM Press (1993) 144–154.
14. Milner, R., Tofte, M., Harper, R., MacQueen, D.B.: The Definition of Standard ML (Revised). The MIT Press (1997).
15. Moscow ML is available at ⟨http://www.dina.kvl.dk/˜sestoft/mosml.html⟩.
16. Paulson, L.C.: ML for the Working Programmer. Second edition. Cambridge University Press (1996).
17. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall (1987).
18. Peyton Jones, S.L., Hughes, J. (eds.): Haskell 98: A Non-Strict, Purely Functional Language. At ⟨http://www.haskell.org/onlinereport/⟩.
19. Plotkin, G.: Call-by-Name, Call-by-Value and the $\lambda$-Calculus. Theoretical Computer Science **1** (1975) 125–159.
20. Revised$^4$ Report on the Algorithmic Language Scheme, IEEE Std 1178-1990. Institute of Electrical and Electronic Engineers (1991).
21. Sestoft, P.: Deriving a Lazy Abstract Machine. Journal of Functional Programming **7**, 3 (1997) 231–264.
22. Stoy, J.E.: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977).
23. Turner, D.A.: A New Implementation Technique for Applicative Languages. Software – Practice and Experience **9** (1979) 31–49.
24. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. D.Phil. thesis, Oxford University, September 1971.

# From Type Inference to Configuration

Morten Heine Sørensen[1]* and Jens Peter Secher[2]

[1] IT-Practice A/S,
Kronprinsessegade 54, 5.,
DK-1306 Copenhagen K, Denmark,
mhs@it-practice.dk

[2] Department of Computer Science, University of Copenhagen,
Universitetsparken 1,
DK-2100 Copenhagen Ø, Denmark,
jpsecher@diku.dk

**Abstract.** A *product line* is a set of products and features with constraints on which subsets are available. Numerous *configurators* have been made available by product line vendors on the internet, in which procurers can experiment with the different options, e.g. how the selection of one product or feature entails or precludes the selection of another product or feature.

We explore an approach to configuration inspired by type inference technology. The main contributions of the paper are a formalization of the configuration problem that includes aspects related to the interactive dialogue between the user and the system, a result stating that the configuration problem thus formalized has at least exponential complexity, and some techniques for computing approximate solutions more efficiently. While a substantial number of papers precede the present one in formalizing configuration as a constraint satisfaction problem, few address the aspects concerning interactivity between the user and the system.

## 1 Introduction

A *product line* is a set of products and features with constraints on which subsets are available. For example, a company may sell a number of different computers (lap-tops, desk-tops, etc.) with different storage device, memory size, etc., but some combinations may be unavailable. For instance, DVD drive and CD-ROM may be mutually exclusive in lap-top models.

Another product line example is computer *software*. What in the beginning is a single software tool often ends up as a family of tools that can be composed in various ways with some constraints. For instance, the software may exist for various platforms (Unix, Windows NT, etc.), but some packages are only available for some platforms.

A third product line example is cars. A car usually comes in various models, where each model can be configured in different ways. For instance, the station

* The work was carried out while the author was employed by Terma A/S.

wagon does not come with interior trunk release, the three-door model does not come with automatic transmission, etc.

It is natural for the vendor to make a system available to the procurer in which he can experiment with the different options, e.g. how the selection of one product or feature entails or precludes the selection of another product or feature. Indeed, such systems have appeared in online-shops on the internet—see, for instance, the home pages of Dell, Compaq, and Ford Motor Company. In fact, there is an emerging market for tools devoted to building such systems—one example is ILOG's configurator.[1] In general we shall refer to such tools as *configurators*.

A related idea concerns the generation of requirements to a system that is to be developed, where the system in advance is known to respect a model with constraints on how the product can be built. That is, although the product does not exist yet, there is a model that expresses constraints pertaining to the product. In this case the user produces a requirement specification instead of a procurement order.

The PROMIS[2] project [8] was concerned with the development of a prototype of such a tool, in which the procurement officer of a military organisation can experiment with the constraints pertaining to so-called $C^3$I-systems.[3] The idea is that a so-called *reference model* is available, containing a specification of all components that can be present in a $C^3$I-system along with the constraints among these components. The procurement officer first makes a consistent choice of components using the tool. Based on this selection, the tool then generates a requirements specification describing the desired $C^3$I-system.

The motivation for the present paper came from two challenges that were identified, but left open in the PROMIS project:

1. Only constraints of very simple forms were considered, namely *aggregation* of entities in the usual object-oriented sense, *specialization* (also in the object-oriented sense), and *implication* among selection of entities. In particular, mutual exclusion of selection of entities was omitted. In the present paper, a more general approach accommodating *any* set of logical constraints is considered.
2. It was not possible to *deselect* an entity—selection could only be *undone* (back-tracked), but there was no way to deselect an entity selected 100 clicks ago without also undoing all the intermediate selections. In the present paper deselections are treated as analogous to selections making them first-class citizens of the approach.

More broadly, in this paper we explore an approach to configurators inspired by type inference technology. Our fundamental idea, strongly inspired by the AnnoDomini project [12, 13], is to view the product line constraints or reference

[1] We do not mean to suggest that any of the mentioned home pages or tools use the techniques described in the present paper.
[2] PROMIS is short for PRocurement Officer Reference Model Information System.
[3] $C^3$I is short for Command, Control, Communication, and Intelligence.

model as a program whose types indicate the selection or de-selection of components. In other words, the user interacts with the system in terms of type-based specification.

Working within this framework provides a natural approach to several problems. For instance, the propagation of selections according to the constraints of the model becomes a type inference problem, the handling of inconsistent desires of the user amounts to resolution of type errors, and the generation of a procurement order or requirement specification is a type-directed translation. Having said this, we should admit that the main advantage of the approach is not the derivation of any new efficient constraint algorithms directly derived from type inference, but rather some insights concerning what type of constraint propagation might be desirable in some configurators and concerning how a user might interact with such constraint propagation algorithms.

In the remainder of the paper we shall be concerned exclusively with procurement from product lines, and leave the connection to the generation of requirements satisfying a reference model implicit. This is not to say that the two problems are identical. There *are* differences between selecting from a product line to produce a purchase specification on the one hand and generating a requirement specification according to a reference model on the other hand. For instance, a requirement specification may be vague, since it is usually the starting point of a discussion with the vendor. In contrast, a purchase specification for an e-shop normally uniquely identifies the product and associated price. Nevertheless, in the remainder of the paper, such differences will be ignored, and we will focus entirely on selection from a product line.

The remainder of the paper is organized in three parts. The first part (Section 2) contains a small *analysis* of the problem to be solved and introduces some concepts relevant for reasoning about product lines—in particular, what objects product line constraints pertain to, and what functionality these constraints must accommodate. The second part (Section 3) presents a *formalization* of the problem by introducing the syntax and semantics for constraints as well a notion of optimal constraint propagation. The third part (Section 4) then presents several *solutions* to the problem in terms of algorithms for propagating constraints, based on type inference, program transformation, and binary decision diagrams. We also outline a configuration system based on these algorithms. The paper ends with a review of related work (Section 5) and some ideas for future work (Section 6).

## 2 Analysis of Product Line Concepts

This section analyzes some concepts relevant for reasoning about product lines. The two first subsections present the building blocks for the constraints that we shall consider; that is, what do the constraints pertain *to?* What do they constrain? The third subsection sets the limits of our scope, and the last subsection finally discusses how the user can interact with our constraints.

Our inspiration for the analysis comes from the PROMIS project [8] mentioned above as well as various online-shops that offer configuration, see e.g. [30]. There are models significantly more complex than what we shall propose, but our working hypothesis is that there are cases, particularly in online-shops, where this complexity is neither required nor desired.

## 2.1 Product Lines

What we buy from a product line are *entities*. For instance, when we buy a car, the entities are the engine (size), the (number of) doors, the transmission (type), etc. In a computer, they are the different hardware components, e.g. storage device. In software they are the different software components.

We configure the product we are buying by *selecting* and *deselecting* entities. For instance, when we buy a computer, we select or deselect the DVD entity, and we select or deselect the CD-ROM entity. A specification, for each entity, whether it is selected, deselected, or none of these, will be called a *configuration*. A configuration in which every entity is either selected or deselected (and hence not unspecified) is called *total*.

When we select from a product line, we cannot choose whatever we like: our selection is *constrained* by the vendor. We have already mentioned the possible mutual exclusion of DVD and CD-ROM drive in a lap-top. The constraints may state such properties as mutual exclusion of selection of two entities (if one is selected, the other cannot be selected), implication from selection of one entity to another (if one is selected, the other must be selected too), etc. In general, any logical relation ("if-then," "and," "or," etc.) between selection of any number of entities is possible. If a configuration respects all the constraints it is called *consistent*. The final order from a user to an online-shop is always a total, consistent configuration.

Thus, the setup is as follows:

- The user is confronted with entities.
- The user may select or deselect an entity.
- When the user selects or deselects an entity, constraints pertaining to entities are checked or propagated.

## 2.2 Views on Product Lines

One can discern at least two different views of the above product lines and constraints. The *domain* view expresses the fundamental constraints of the product line. For instance, it does not make sense to furnish a station wagon with an interior trunk release.

The *sales* view is how the marketing manager sees the product line. This view can be seen as a refinement of the domain view—the marketing manager cannot change the fundamental rules concerning how a car can be composed. However, the marketing manager will add entities and constraints of the following types among others:

- *Packages:* several related entities may be grouped in a package. For instance, a car may be configured with a sports package that selects a powerful engine, rear spoiler, extra front lights, etc.
- *Models:* to make the possible selections simpler for the user, and to standardize the actual construction of the purchased items, models are employed. For instance, computers usually come in various models at various prices. Cars also usually come in various models (Ford Focus, Ford Mustang) and submodels (Sedan, Sedan 3-door, station wagon).
- *Additional constraints:* from a marketing point of view it could be desirable e.g. to exclude fancy features in low-price models. For instance, leather interior might be precluded from cars that include the family package, despite the fact that it is possible for the vendor to furnish all models with leather interior.

The domain view is materialized simply by building a model with the correct entities and constraints. The sales view is then built on top of that by adding:

- Entities corresponding to packages and models.
- Constraints reflecting the aggregation relationship between a package or a model and its constituent entities, i.e. constraints expressing that selection of, say, a model entails selection of all its constituent entities.
- Constraints reflecting the marketing manager's additional constraints.

Thus, both views are accommodated by our simple set-up.

## 2.3 What the Analysis Does Not Cover

An entity can have a number of *features*. These are not entities but rather properties of the entities. For instance, a CPU has a frequency. The frequency itself is not a an entity (at least not a physical one), but rather a property of the CPU. Similarly, the CPU probably has a price. Each feature has a *value*. For instance, the CPU frequency is some number, as is the CPU price.

Features may also pertain to a group of entities, rather than a single entity. For instance, the acceleration of a car is not merely a property of the engine: a station wagon probably has a smaller accelaration than a sedan model with the same engine. Similarly, the user is most likely interested in the price of the car he has configured, and the total price is not a property of any of the selected entities.

In this paper we will not consider features or constraints pertaining to them. Of course, in an online-shop, there must be an approach to calculating the price(!), but this can be handled in an ad-hoc manner, by letting each entity have a price, and letting the total price be just the sum of all prices for selected entities. In principle, a feature with $n$ different possible values can be encoded as $n$ entities with a constraint expressing that exactly one of the entities must be selected, but this encoding may not be practical if $n$ is very large; however, our working assumption is that there are online-shops whose configuration problem can be implemented without features and values.

The constraints discussed above are those imposed *on* the user *by* the vendor: if an entity is selected, another is precluded; and if a certain selected entity's feature has a certain value, a certain other selected entity's feature must have a certain other value. The converse type is also conceivable: the user might have constraints. For instance, "If I get a station wagon, I want a 2.0 engine" or "I want the cheapest possible station wagon with an acceleration from 0 to 100 km/h better than 20 seconds."

In this paper, we will not deal explicitly with user constraints. As far as we are concerned, these constraints must be satisfied by the user by making appropriate choices among the entities, respecting the constraints developed by the vendor.

There are a number of other model concepts that we also preclude from our analysis. For instance, we do not consider inheritance or other relations between entities—except the relations expressed by the logical constraints pertaining to possible selection and deselection.

## 2.4 Selecting from Product Lines

Whereas the preceding subsections have been concerned with *what* the user can select, the precent one addresses the *how.* How does the user select or deselect entities? An important aspect of this is the *order* in which the user is asked to select various entities. There are several options, e.g.:

– *Sequential selection and deselection of desired components.* First the user selects or deselects the first entity (in some given order), then the next and so on. For instance, first the user selects model, then submodel, etc.
– *Arbitrary selection and deselection among all components.* In this approach the user can freely select or deselect any entity in any order, respecting the constraints along the way, of course.
– *Combinations.* The above can be combined. For instance, first the user selects model, then submodel, then he selects among the available packages, and finally he has a chance to review the whole list of selected entities and change any selection, respecting the constraints.

We believe that the choice among these should not be a property of the constraint engine, but rather of a presentation layer built on top of that, following the usual *Model-View-Controller* pattern [17]. It should be possible to use different selection orders with the same underlying constraint engine. Thus, in the present paper, we will only be concerned with how the underlying constraint engine works; any of the above orders can be used by building a presentation layer above the constraint engine, guiding the user through the different choices in the desired order.

Each selection can also be presented in various ways, e.g. with

– *check boxes*: choosing an option or not.
– *radio buttons*: choosing exactly one of the options from a list.
– *list boxes*: choosing zero or more of the options from a list.

For instance, a mandatory choice between a number of mutually exclusive entities can be presented with a radio button—this way, the mutual exclusion will be enforced in the presentation layer. Moreover it is presented in a form likely to be familiar to the user.

But again, the choice between these different types of presentation should not be a property of the constraint engine. That is, the engine should not *rely* on the presentation layer working in one way or another. The presentation layer should collect input in a structured form (e.g. a radio button) and submit it to the engine in a flat form (selection, deselection, or unspecified status for every entity).

It is important to understand that even a check box can be used for different forms of choice:

1. checked means "select entity," unchecked means "unspecified whether to select entity."
2. checked means "select entity," unchecked means "deselect entity."
3. checked means "deselect entity," unchecked means "unspecified whether to select entity."
4. checked means "deselect entity," unchecked means "select entity."

In other words, the constraint engine can receive three types of user input concerning an entity: *select*, *deselect*, and *unspecified*, and the two options of a check box can be used by the presentation layer to indicate any combination of two of these.

Concerning the amount of work done by the system based on a selection, there are at least the following three types of solutions:

– *checking:* The system verifies that the user's current selection is consistent and reports errors if there are selections or deselections that contradict the set of constraints.
– *propagation:* The system, in addition to verifying the consistency of the selections and deselections, also infers all consequences of these, thereby selecting and deselecting a number of other entities. The system also reports inconsistent selections and deselection (both those of the user and those inferred) as in the checking approach.
– *propagation and default selection:* The system, in addition to verifying the consistency of the selections and deselections and in addition to inferring all consequences of these, makes some default choices concerning selection and deselection of entities in cases where a unique solution cannot be inferred from the user's selection. For instance, if a constraint requires selection of one among a number of entities, the system could arbitrarily select one of these.

The choice among these is a true property of the constraint engine. We believe that the second is preferable over the first one. The user should know the consequences of these selections as soon as possible—it is annoying to find out that some selections have undesirable consequence long time and many considerations after the selection has been made.

The choice between the second and third is, at least partly, a matter of taste, we believe. The advantage of the third is that it can be used in such a way that at any given time, the system has a total, consistent configuration: those choices the user has not made himself, have been made by the system. On the other hand, it may be difficult for the user to distinguish between selections made by the system that were dictated by the user's previous selections on the one hand, and default selections on the other hand. It may be relevant for the user to tell the difference, because he can override the latter type, but not the former (at least not without changing some of his previous selections).

We prefer the second option (some of the effect of the third option can be obtained in the second option by having suggestions for the user in addition to the actual propagation). Although we prefer the second option—propagation—we use in fact a solution which is an intermediate step between checking and propagation, because in our set-up the full propagation solutions appears to be intractable, as will be explained later.

Finally, there is the question of *when* to do propagation. The following two options are possible:

– *online:* whenever the user makes a selection or deselection, the system checks or progates the configuration.
– *batch:* the user makes a number of selections or deselections, and the system then checks or propagates the configuration.

If the batch version is used by only invoking it when the user has specified a total configuration, then an engine implementing propagation will actually only work as one that checks, since the configuration is total. In any event, the choice between these two possibilities will also be left to the presentation layer; the contrainst engine will be able to handle both.

In conclusion, the choice between all the options of this subsection will be left to the presentation layer, which is not specified in this paper, with the one exception that we will build a constraint engine where the option between checking and propagation is made in favor of propagation—to the extent feasible.

## 3 Formalization of Constraints

Having established some idea about what form constraints may have, and how the user should interact with them, we now proceed to present a syntax and semantics of entities and constraints and a definition of the constraint propagation problem. This latter definition is novel and one of the main contributions of the paper.

## 3.1 Syntax of Product Line Constraints

**Definition 1.** The syntax of the universe of discourse is as follows:

$$\begin{array}{lll}
\mathit{Problem} \ni \mathit{problem} & ::= & \mathit{conf} \quad \mathit{constrset} \\
\mathit{Conf} \ni \mathit{conf} & ::= & E_1 : \tau_1 \,, \ldots \,, E_n : \tau_n \\
\mathit{Indic} \ni \tau & ::= & \mathbf{selected} \mid \mathbf{deselected} \mid \mathbf{unspecified} \\
\mathit{Constrset} \ni \mathit{constrset} & ::= & \mathit{constr}_1, \ldots \,, \mathit{constr}_m \\
\mathit{Constr} \ni \mathit{constr} & ::= & E \mid \mathit{constr} \Rightarrow \mathit{constr} \mid \neg \mathit{constr}
\end{array}$$

We assume a set of entity names, ranged over by $E, E_1, E_2$, etc. A *constraint problem* consists of a *configuration* and a *constraint set.* The configuration, in turn, consists of zero or more pairs each comprising an *entity name* and a *select indication*, i.e. an indication of whether the user has selected the entity, deselected the entity, or none of these. The configuration with zero such pairs is denoted by $\varepsilon$. A constraint set consists of zero or more constraints each of which is a propositional formula in which the entities are the atomic propositions. The constraint set with zero constraints is denoted by $\varepsilon$. We require that every entity occurring in *constrset* also occurs in *conf* and vice versa.

A configuration is called *total* if no entity has select indication **unspecified.** Similarly, a configuration is called *empty* if *every* entity has select indication **unspecified**.

As can be seen from the definition, we take $\Rightarrow$ and $\neg$ as primitives. Thus, the three forms of constraints are:

- $E$ must be selected.
- If constraint $\mathit{constr}_1$ is true, then $\mathit{constr}_2$ must be true.
- Constraint *constr* must not be true.

It is well-known (see e.g. [24]) that the remaining connectives can be derived, and we shall use disjunction, conjunction, etc., as well as the atomic formulas **false** and **true**, freely in the rest of the paper. In practice, such derivations should be handled by a mapping layer between the presentation layer and the constraint engine.

Some common binary constraints are:

1. The entity $E_1$ is an aggregation of among others $E_2$.[4]
2. The selection of entity $E_1$ precludes the selection of $E_2$.
3. The selection of entity $E_1$ implies the selection of $E_2$.

More precisely, these are informal linguistic constraints. The actual constraints are:

1. $E_1 \Rightarrow E_2$.
2. $E_1 \Rightarrow \neg E_2$.
3. $E_1 \Rightarrow E_2$.

[4] Aggregation is also called *structural decomposition*, see e.g. [39].

Incidentally, note that the first of these is identical to the last one. To express that $E$ is composed of $E_1$ and $E_2$ one uses the constraints

$$E \Rightarrow E_1.$$
$$E \Rightarrow E_2.$$

These constraints state that one cannot select a whole without selecting the parts. One can also consider adding the constraint

$$E \Leftarrow E_1 \wedge E_2, \tag{1}$$

which states that one has selected the whole, if one has selected all the parts. However, this latter constraint is not necessarily relevant. For instance, $E_1$ could be "rear spoiler," $E_2$ could be "extra front lights," and $E$ could be "sports package." An easy way for the user to select both $E_1$ and $E_2$ would be to select $E$, then the system infers that the two entities must be selected as well. But even if the user has selected $E_1$ and $E_2$ it may be irrelevant for him to know that this is the sports package, so it might not be necessary to have the system infer $E$. Moreover, the user may deselect the sports package, and yet select manually the two constituents. This works fine without (1), but does not work with (1) present. So whether one should include (1) or not, depends on what one wants to express.

To sum up, our "model" (the constraints) only allows elements that express boolean relationships between selection and deselection of entities. Other traditional model elements such as aggregation, specialization (inheritance), etc. must be translated to boolean constraints, and such a translation might be handled by a layer between the presentation layer and the constraint engine.

A typical process involving selections and deselections starts with an empty configuration and makes progress towards a total configuration. The following definition formalizes the notion of "making progress" in terms of "refinements."

**Definition 2.**

1. For select indications $\tau, \tau'$ define the relation $\tau \leq \tau'$ (read $\tau'$ is a *refinement* of $\tau$) as follows:
$$\begin{array}{ll} \tau & \leq \tau. \\ \textbf{unspecified} & \leq \tau. \end{array}$$
2. For configurations *conf*, *conf'*, define the relation *conf* $\leq$ *conf'* (read *conf'* is a *refinement* of *conf*) as follows:
$$\begin{array}{lll} \varepsilon & \leq \varepsilon & \\ \mathit{conf}\ E : \tau \leq \mathit{conf'}\ E : \tau' & & \text{if } \mathit{conf} \leq \mathit{conf'} \text{ and } \tau \leq \tau' \end{array}$$
3. If *conf* $\leq$ *conf'* and *conf* $\neq$ *conf'* then *conf* $<$ *conf'*.
4. If *conf* is the configuration $E_1 : \tau_1, \ldots, E_n : \tau_n$ then $\mathit{conf}[E_i := \tau]$ is the configuration
$$E_1 : \tau_1, \ldots, E_{i-1} : \tau_{i-1}, E_i : \tau, E_{i+1} : \tau_{i+1}, \ldots, E_n : \tau_n.$$

### 3.2 Semantics of Product Line Constraints

We now proceed to provide the semantics of our constraints. Roughly, the semantics of our constraint sets is obtained by identifying **selected** and **deselected** with the truth values "true" and "false," respectively, and using the usual semantics of classical, propositional logic.

**Definition 3.**

1. A *valuation* is a map from entity names to $\{t, f\}$.
2. For a valuation $\nu$ and constraint *constr*, define $[\![constr]\!]_\nu$ by:

$$\begin{array}{ll} [\![E]\!]_\nu & = \nu(E) \\ [\![constr_1 \Rightarrow constr_2]\!]_\nu & = \begin{cases} t & \text{if } [\![constr_1]\!]_\nu = f \text{ or } [\![constr_2]\!]_\nu = t \\ f & \text{otherwise} \end{cases} \\ [\![\neg constr]\!]_\nu & = \begin{cases} t & \text{if } [\![constr]\!]_\nu = f \\ f & \text{otherwise} \end{cases} \end{array}$$

3. For a valuation $\nu$ and constraint set *constrset*, define $[\![constrset]\!]_\nu$ by:

$$\begin{array}{ll} [\![\varepsilon]\!]_\nu & = t \\ [\![constrset\ constr]\!]_\nu & = \begin{cases} t & \text{if } [\![constrset]\!]_\nu = t \text{ and } [\![constr]\!]_\nu = t \\ f & \text{otherwise} \end{cases} \end{array}$$

4. For a valuation $\nu$, an element $b \in \{t, f\}$, and an entity name $E$, define the valuation $\nu[E \mapsto b]$ by:

$$\nu[E \mapsto b](E') = \begin{cases} b & \text{if } E = E' \\ \nu(E') & \text{otherwise} \end{cases}$$

*Remark 1.* When we consider a valuation together with some constraint or constraint set, it is always implicitly assumed that the valuation is defined for all the entities occurring in the constraint or constraint set.

**Definition 4.**

1. A constraint *constr* is *valid* if $[\![constr]\!]_\nu = t$ for all valuations $\nu$.
2. A constraint *constr* is *satisfiable* if there exists a valuation $\nu$ with $[\![constr]\!]_\nu = t$.
3. A constraint *constr* is *uniquely satisfiable* if it is satisfiable and any two valuations $\nu_1$ and $\nu_2$ with $[\![constr]\!]_{\nu_1} = t = [\![constr]\!]_{\nu_2}$ satisfy $\nu_1(E) = \nu_2(E)$ for every entitity $E$ occuring in *constr*.
4. A constraint *constr* is *true* in a valuation $\nu$ iff $[\![constr]\!]_\nu = t$.
5. The above notions are generalized from constraints to constraint sets by replacing all occurrences of *constr* by *constrset*.

*Remark 2.* As usual, *constr* is valid iff $\neg constr$ is unsatisfiable, and *constr* is satisfiable iff $\neg constr$ is not valid.

We often need to consider the above notions relative to a given configuration *conf*. For instance, we want to know whether *constrset* is satisfiable in such a way that the corresponding valuation $\nu$ that makes *constrset* true respects the assignments in *conf*, i.e. $\nu$ assigns true and false to all entitites that are mapped to **selected** and **deselected**, respectively, in *conf*.

**Definition 5.** Let *conf* be a configuration.

1. The valuation $\nu_{conf}$ *determined by* a total configuration *conf* is defined as follows:
$$\begin{aligned} \nu_{\varepsilon} &= \{\} \\ \nu_{conf\ E:\tau} &= \begin{cases} \nu_{conf}[E \mapsto t] & \text{if } \tau = \textbf{selected} \\ \nu_{conf}[E \mapsto f] & \text{if } \tau = \textbf{deselected} \end{cases} \end{aligned}$$
2. A *conf*-valuation is a valuation of the form $\nu_{conf'}$ for some total refinement *conf′* of *conf*.
3. All the notions in Definition 4 are generalized by replacing valuation, valid, satisfiable, uniquely satisfiable, and true by *conf*-valuation, *conf*-valid, *conf*-satisfiable, *conf*-uniquely satisfiable, and *conf*-true, respectively.

*Remark 3.* The valuation $\nu_{conf}$ is only defined for total configurations.

## 3.3 The Constraint Propagation Problem

The preceding two subsections have presented constraints and their semantics. It remains to address the question: for the actual application of constraints to product line shopping, which properties are we interested in—satisfiability, validity, etc? We now proceed to answer this question.

The following three mutually exclusive and together complete cases could be of interest for a constraint problem *conf constrset*:

1. *constrset* is *conf*-valid. This means that no further conflicts are possible, provided the user from now on only makes choices about entities with indication unspecified. A special case is when there are no more entities with select indication unspecified, i.e. when the selection is total. In this case the user has made a total selection which makes all the constraints true.
2. *constrset* is not *conf*-satisfiable. This means that the user has made a selection which violates one or more constraints. Regardless of how the user handles entities which are currently unspecified, it will not be possible to make all constraints true. A special case is when there are no more entities with select indication unspecified, i.e. when the selection is total. In this case the user has made a total selection which does not make all the constraints true.
3. *constrset* is *conf*-satisfiable, but not *conf*-valid. This means that the user still has a chance to arrive at a total selection that makes all the constraints true. In this case we can distinguish two subcases:

(a) *constrset* is *conf*-uniquely satisfiable. In this case, the remaining selection is dictated by what the user has already selected.
(b) *constrset* is not *conf*-uniquely satisfiable. In this case, there are several different possible selections the user can make.

In addition to these properties we are interested in *propagating* selections. The user does not simply want to know "the constraint set now has a single solution;" he also wants to know what the solution is, that is, which selections are forced by other selections. In fact, even when there is not a single unique solution for the constraint set, the user still wants to be informed if the selection or deselction of one or more of the entities participating in the constraint set can be inferred. Moreover, when there are conflicts, he wants to know what the conflicts consist of and, if possible, have suggestions concerning how to resolve the conflicts. In short, the above cases must be ammended by the following:

– *Inference:* propagation of select indications.
– *Errors:* reporting of inconsistent select indications.
– *Corrections:* suggestions for changes in inconsistent select indications.

Thus, our inference algorithms will not compute validity or satisfiability directly, but rather a mixture in which propagation plays a role, and it will report errors and suggest corrections. We now formalize these considerations in terms of an *optimal inference algorithm.*

**Definition 6.**

1. An *inference algorithm* is an algorithm
$$I\colon \mathit{Conf} \times \mathit{Constrset} \to \mathit{Conf} \cup \{\mathbf{fail}\},$$
where $\mathit{conf} \leq I(\mathit{conf}, \mathit{constrset})$ when $I(\mathit{conf}, \mathit{constrset}) \neq \mathbf{fail}$.
2. We say that *conf makes constrset true* if $[\![\mathit{constrset}]\!]_{\nu_{\mathit{conf}}} = t$, for a total *conf*. We also call *conf* a *solution to constrset.*
3. An inference algorithm is *optimal* if the following two conditions are satisfied:
   (a) If *constrset* is *conf*-satisfiable, then
   $$I(\mathit{conf}, \mathit{constrset}) = \mathit{conf}'. \qquad (2)$$
   Moreover,
   i. For all total $\mathit{conf}''$ such that
   $$\mathit{conf} \leq \mathit{conf}''$$
   and $\mathit{conf}''$ makes *constrset* true, it holds that $\mathit{conf}' \leq \mathit{conf}''$.
   ii. There is no $\mathit{conf}'''$ with $\mathit{conf}' < \mathit{conf}'''$ such that for all total $\mathit{conf}''$ with
   $$\mathit{conf} \leq \mathit{conf}''$$
   and $\mathit{conf}''$ makes *constrset* true, it holds that $\mathit{conf}''' \leq \mathit{conf}''$.[5]

[5] Condition ii states that the configuration *conf* cannot be further refined and still satisfy property i.

(b) If *constrset* is not *conf*-satisfiable, then

$$I(\mathit{conf}, \mathit{constrset}) = \textbf{fail}. \tag{3}$$

The definition of optimality is inspired by *completeness* of polymorphic type inference.[6]

## 3.4 Properties of Optimal Inference Algorithms

Have we succeeded with the preceding definition in capturing the desired form of inference algorithm? We aim to show that the answer is *yes*. We will do so with some informal considerations pertaining to the definition combined with a series of propositions stating some of the desired properties.

The following shows that there is at most one optimal inference algorithm. This can be viewed as a sort of completeness of the definition in that it does not make sense to develop additional desired properties of optimal inference algorithms: those developed already uniquely identify the algorithm (if it exists at all). Of course, only the input-output behaviour of the algorithm is uniquely identified. It may *work* in many ways.

**Proposition 1.** *Let $I_1$ and $I_2$ be optimal inference algorithms, and consider the problem conf constrset. Then for all conf and constrset:*

$$I_1(\mathit{conf}, \mathit{constrset}) = I_2(\mathit{conf}, \mathit{constrset}).$$

*Proof.* Let $I_1, I_2$ be optimal inference algorithms, and let *conf constrset* be a problem. If *constrset* is not *conf*-satisfiable then by optimality

$$I_1(\mathit{conf}, \mathit{constrset}) = \textbf{fail} = I_2(\mathit{conf}, \mathit{constrset}).$$

If *constrset* is *conf*-satisfiable then, again by optimality,

$$I_1(\mathit{conf}, \mathit{constrset}) = \mathit{conf}'_1$$

and

$$I_2(\mathit{conf}, \mathit{constrset}) = \mathit{conf}'_2,$$

[6] There are two changes compared to completeness of polymorphic type inference. The first is clause (ii) of part (a), which is added to force as much refinement as possible. Indeed, without this addition, the inference algorithm could simply return the argument *conf* unaltered. The reason is that we do not require that the output of an optimal inference algorithm be a solution (in particular a total configuration), since most general solutions do not exist. In contrast, in type inference there is a requirement that the answer actually be a type for the term. This means that the analogous situation—that the algorithm simply returns a type variable as the most general type—does not occur there.

The second change is the phrase "total" in clause (i) and (ii) of part (a). The reason for this is that we only speak of solutions among total configurations. In contrast, a type for a term may contain type variables.

where

$$\mathit{conf}'_1 \geq \mathit{conf} \leq \mathit{conf}'_2.$$

Since *constrset* is *conf*-satisfiable, there are a number of total configurations $\mathit{conf}''$ with $\mathit{conf} \leq \mathit{conf}''$ such that $\mathit{conf}''$ makes *constrset* true. By optimality, each $\mathit{conf}''$ is in fact a refinement of both $\mathit{conf}'_1$ and $\mathit{conf}'_2$, i.e.

$$\mathit{conf}'_1 \leq \mathit{conf}'' \geq \mathit{conf}'_2.$$

Let $E$ be an entity in *constrset.* We split into two disjoint cases.

1. Every total refinement $\mathit{conf}''$ of *conf* that makes *constrset* true assigns the same select indication (**selected** or **deselected**) to $E$. In this case, $\mathit{conf}'_1$ must assign the same indication to $E$. [Reason: if another indication is assigned to $E$, no $\mathit{conf}''$ is a refinement of $\mathit{conf}'_1$, and if **unspecified** is assigned to $E$, there is a proper refinement of $\mathit{conf}'_1$ that still has all total configurations making *constrset* true as refinements. Both cases contradict optimality.] Similarly, $\mathit{conf}'_2$ must assign the same indication to $E$.
2. There exist two total refinements $\mathit{conf}''_a$ and $\mathit{conf}''_b$ of *conf* that assign **selected** and **deselected** to $E$, respectively, and which both make *constrset* true. In this case, $\mathit{conf}'_1$ must assign **unspecified** to $E$. [Reason: otherwise there would be a total refinement of *conf* making *constrset* true that is not also a refinement of $\mathit{conf}'_1$, contradicting optimality.] Similarly, $\mathit{conf}'_2$ must assing **unspecified** to $E$.

Thus, for every entity, the indication is uniquely determined. □

The equations (2) and (3) state that the algorithm must return **fail** if, and only if, the current configuration is such that there does not exists a refinement of this configuration that makes the constraint set true. That is, we have:

**Proposition 2.** *Let I be an optimal inference algorithm, and consider the problem conf constrset. Then*

$$I(\mathit{conf}, \mathit{constrset}) = \textbf{fail}$$

*if, and only if, constrset is not conf-satisfiable.*

*Proof.* Immediate from the definition. □

The following proposition shows that if an optimal inference algorithm returns a configuration $\mathit{conf}'$ at all (i.e. not **fail**) for a contraint set *constrset*, then *constrset* is *conf*-satisfiable. Moreover, although the returned configuration does not need to be total and make the constraints true, it must be such that all total configurations that make the constraints true are obtained by further refinements, and such refinements do exist.

**Proposition 3.** *Let I be an optimal inference algorithm, and consider the problem conf constrset. Then*

$$I(\mathit{conf}, \mathit{constrset}) = \mathit{conf}' \tag{4}$$

*if, and only if, constrset is conf-satisfiable. Moreover, when (4) holds, constrset is conf′-satisfiable.*

*Proof.* The only part that does not follow immediately from the definition is that (4) implies that *constrset* is *conf′*-satisfiable. Suppose that (4) holds. Since *constrset* is *conf*-satisfiable there is a total refinement of *conf* that makes *constrset* true. By optimality, this refinement must also be a refinement of *conf′*. Hence *constrset* is *conf′*-satisfiable. □

Clause (i) in part (a) of the definition of completeness states that all total refinements of the current configuration that make the constraint set true, must also be refinements of the computed configuration. That is, an optimal algorithm must not refine so much that solutions that were refinements of the configuration we started with, can no longer be obtained—the refinement must preserve all solutions. Another way of putting this is that the algorithm must not make arbitrary choices. Clause (ii) states that there must not be a proper refinement of the computed configuration that also preserves all solutions—in this case the proper refinement should have been the result. Clause (i) and (ii) together then state that the algorithm must make all the non-arbitrary choices, thereby computing a sort of *most specific generalization.*

As a special case, if the configuration has a single total refinement that makes the constraint set true, the algorithm must find this one.

**Proposition 4.** *Let I be an optimal inference algorithm, and consider the problem conf constrset. If constrset is conf-uniquely satisfiable, and $\nu$ is the corresponding conf-valuation, i.e.*

$$[\![constrset]\!]_{\nu} = t = [\![constrset]\!]_{\nu_{conf'}} \ ,$$

*for some total refinement conf′ of conf, then*

$$I(conf, constrset) = conf'.$$

*Proof.* Since *constrset* is *conf*-satisfiable,

$$I(conf, constrset) = conf',$$

for some refinement *conf′* of *conf*. Since *constrset* is *conf*-uniquely satisfiable there is exactly one total refinement *conf″* that makes *constrset* true. We must then have *conf′* = *conf″*; otherwise, there would be a contradiction with the criterion (a), part (ii), in the definition of optimality. □

Conversely, we have the following.

**Proposition 5.** *Let I be an optimal inference algorithm, and consider the problem conf constrset. If*

$$I(conf, constrset) = conf',$$

*where conf′ is total, then constrset is conf-uniquely satisfiable, and $\nu_{conf'}$ is the corresponding valuation.*

*Proof.* If

$$I(conf, constrset) = conf',$$

where *conf′* is total, then every total refinement of *conf* that makes *constrset* true is also a refinement of *conf′*. Since *conf′* is already total it must be the only total refinement of *conf* that makes *constrset* true. Thus, *constrset* is *conf*-uniquely satisfiable. The corresponding evaluation must be $\nu_{conf'}$. □

As another special case, if all total refinements make the constraint set true, the algorithm is not able to make any propagation without user intervention.

**Proposition 6.** *Let I be an optimal inference algorithm, and consider the problem conf constrset. If constrset is conf-valid, then*

$$I(conf, constrset) = conf.$$

*Proof.* If *constrset* is *conf*-valid, it is *conf*-satisfiable, so

$$I(conf, constrset) = conf'$$

for some refinement *conf′* of *conf*. Let $E$ be some entity that *conf* assigns **unspecified**. Then there is a total refinement of *conf* that assigns **selected** to $E$ and there is another total refinement of *conf* that assigns **deselected** to $E$. Hence, *conf* must assign **unspecified** to $E$ as well, by optimality. □

The converse does not hold; that is, it may be that

$$I(conf, constrset) = conf$$

and yet *constrset* is not *conf*-valid. Consider, for instance, the problem:

$$\begin{array}{l} E_1 : \textbf{unspecified} \\ E_2 : \textbf{unspecified} \\ E_1 \Leftrightarrow \neg E_2. \end{array}$$

We have

$$I(conf, constrset) = conf,$$

eventhough the constraint is not valid. The point is that it is non-uniquely satisfiable.

It follows from the preceding propositions that we can recognize satisfiability (getting **fail** or not) and unique satisfiability (getting a total configuration back or not). We therefore have:

**Proposition 7.**

1. *Computing an optimal inference algorithm is as hard as SAT (the general satisfiability problem).*
2. *Computing an optimal inference algorithm is as hard as USAT (the unique satisfiability problem).*

*Proof.* With *conf* equal to the empty configuration (i.e. all entities mapped to **unspecified**), *conf*-satisfiability and *conf*-unique satisfiability degenerate to satisfiability and unique satisfiability, respectively. Hence the result follows from Propositions 3, 4, and 5. □

It follows that it may not be feasible to compute an optimal algorithm.

## 4 Solution of Contraints

In this section we finally provide several algorithms for solving constraint problems. The first subsection describes several conceptual algorithms, and the second subsection outlines the elements of a configuration system based on these algorithms. The last two subsections explore ideas for implementation of the conceptual algorithms based on program transformation techniques and binary decision diagrams, respectively.

### 4.1 Conceptual Algorithms

We will assume that the overall constraint set may be quite large (say several hundred entities and a similar number of constraints). It follows from the last result in the preceding section that it is probably intractable to compute an optimal inference algorithm on the entire constraint set.

In contrast we will assume that each constraint is typically quite small, say involving less than 10 entities. Therefore, it may be tractable to compute an optimal inference algorithm at the level of the individual constraints. The following is the obvious such algorithm; we call it $I_l$ ($l$ for "local").

**Definition 7.** For $b \in \{f, t\}$ define

$$\underline{b} = \begin{cases} \textbf{selected} & \text{if } b = t \\ \textbf{deselected} & \text{if } b = f. \end{cases}$$

**Algorithm 1** *Define*

$$I_l : \mathit{Conf} \times \mathit{Constr} \rightarrow \mathit{Conf} \cup \{\textbf{fail}\}$$

*by:*

```
1. input: configuration $conf$ and constraint $constr$.
2. let $E_1, \dots, E_n$ be the entities in $conf$ mapped to $\textbf{unspecified}$.
3. let $c_1 = \dots = c_n = \textbf{unspecified}$.
4. let $sol = \textbf{false}$.
5. for each $conf$-valuation $\nu$:
      if $[\![constr]\!]_\nu = t$ then
         if $sol = \textbf{true}$ then for $i \in \{1, \dots, n\}$:
            if $c_i \neq \underline{\nu(E_i)}$ then $c_i = \textbf{unspecified}$.
         if $sol = \textbf{false}$ then
            $sol = \textbf{true}$.
            for $i \in \{1, \dots, n\}$: $c_i = \underline{\nu(E_i)}$.
6. $conf' = conf[E_1 \mapsto c_1, \dots, E_n \mapsto c_n]$.
7. if $sol = \textbf{false}$ then return $\textbf{fail}$ else return $conf'$.
```

This algorithm is inefficient in that it traverses all possible (and impossible) valuations for the entities in each constraint. This can be remedied by including in the language some predefined constraints such as:

$$E_1 \wedge \ldots \wedge E_n \Rightarrow E'_1 \wedge \ldots \wedge E'_n,$$

where all the entities are distinct. The propagation along this constraint can be done more efficiently than by traversing all possible valuations and checking whether only one is possible, namely in the following way:

1. if $E_1, \ldots, E_n$ are **selected** and $E'_1, \ldots, E'_n$ are **selected** or **unspecified**, then those that are **unspecified**, must be refined to **selected**.
2. if at least one of $E'_1, \ldots, E'_n$ is **deselected**, and all of $E_1, \ldots, E_n$ are **selected**, except $E_i$ which is **unspecified**, then $E_i$ must be refined to **deselected.**

By having such special treatment for certain constraints, we are pruning the search for solutions in the space of all valuations. In the next two sections, other techniques for improving the efficiency of $I_l$ are introduced.

By viewing a constraint set as the conjunction of all its constraints, and using the above algorithm, we obtain the following result.

**Proposition 8.** *There exists an optimal inference algorithm.*

*Proof.* The algorithm corresponds to the cases in the proof of Proposition 1. □

However, for efficiency reasons we might not wish to use the algorithm (or some other implementation of it) at the constraint set level. Instead we might use the following algorithm $I_g$ ($g$ for "global"), which calls $I_l$ for each individual constraint.

**Algorithm 2** *Define*

$$I_g : Conf \times Constrset \to Conf \times Constrset$$

*by:*

```
1. input: $conf_0$ and $constrset = constr_1, \ldots, constr_m$.
2. let failures= {}.
3. for $i = 1, \ldots, m$
       let $I_l(conf_{i-1}, constr_i) = result$.
       if $result = \mathbf{fail}$ then
           let failures=failures $\cup \{constr_i\}$.
           let $conf_i = conf_{i-1}$.
       else let $conf_i = result$.
4. return $conf_m$ and failures.
```

The algorithm returns two objects. The first is the refined configuration $conf_m$, which should be considered as a suggestion from the system concerning how to refine the selections. The user can either accept these, or make other selections. The second object returned by the algorithm is the set *failures*, which is the list of constraints that have no solution. One can imagine that the algorithm, in addition to this set, produces a list of suggested *corrections*, i.e. suggestions for changing the user's selections in such a way that the constraints become satisfiable. Among all the different possibilities, the system might present the first few with smallest distance to the configuration $conf_0$, for some notion of distance.

The following example, that could not be handled by the techniques developed in the PROMIS project [8], was the original motivation for the research reported in the present paper.

*Example 1.* Consider the constraint problem *conf constrset*:

$$\begin{aligned}&E_1 : \textbf{unspecified}\\&E_2 : \textbf{unspecified}\\&E_3 : \textbf{unspecified}\\&E_1 \Rightarrow E_2\end{aligned}$$

Suppose the user selects $E_1$ and runs $I_g$, i.e. let

$$conf' = conf[E_1 \mapsto \textbf{selected}].$$

Then $I_g(conf', constrset)$ will assign **selected** to $E_2$, so this assignment will be suggested to the user. Suppose that the user selects entity $E_3$, so that he has now selected $E_1$ and $E_3$, i.e. let

$$conf'' = conf'[E_3 \mapsto \textbf{selected}].$$

Then $I_g(conf'', constrset)$ will again assign **selected** to $E_2$, so this is again suggested to the user. Now suppose the user deselects $E_1$, i.e. let

$$conf''' = conf''[E_1 \mapsto \textbf{deselected}].$$

Then $I_g(conf''', constrset)$ will not assign **selected** to $E_2$, since the propagation from $E_1$ to $E_2$ no longer happens, so the suggestion to select entity $E_2$ will go away.

This example illustrates that our set-up can accommodate interactions with the user that are not possible with an undo-feature, because in the above example, undoing instead of deselecting $E_1$ would also roll back the selection of $E_3$, i.e. the whole history of interactions is rolled back a number of steps. This is generally not the desired behaviour.

The algorithm $I_g$ has at least two shortcomings. The first is that the propagation depends on the order in which the constraints are processed, and correct propagation may be lost due to this order. For instance, if the constraint problem

$$\begin{aligned}&E_1 : \textbf{unspecified}\\&E_2 : \textbf{selected}\\&E_3 : \textbf{unspecified}\\&E_1 \Leftrightarrow \neg E_3\\&E_2 \Rightarrow E_1\end{aligned}$$

is processed in the order in which the constraints are listed, then one execution of $I_g$ will refine $E_1$ to selected, but $E_3$ will not be refined, although clearly $E_3$ should be refined to be deselected.

A simple way to remedy this is by iterating $I_g$ until no further refinements are obtained. While such compromises are ugly from a theoretical point of view, they can be necessary and useful in practice. A more efficient remedy is to not iterate $I_g$ on the entire constraint set, but only on constraints containing an entity which was refined in the previous iteration.

The second shortcoming of $I_g$ is that some propagation is left out, regardless of the order or number of times the constraints are processed. For instance, consider the constraint problem *conf* $\mathit{constr}_1, \mathit{constr}_2, \mathit{constr}_3$:

$$\begin{array}{l} E_1 : \textbf{unspecified} \\ E_2 : \textbf{unspecified} \\ E_3 : \textbf{unspecified} \\ E_1 \Leftrightarrow \neg E_2 \\ E_2 \Leftrightarrow \neg E_3 \\ E_3 \Leftrightarrow \neg E_1 \end{array}$$

All three constraints are satisfiable (though not uniquely), and we have that $I_l(\mathit{conf}, \mathit{constr}_i) = \mathit{conf}$ for each $i = 1, 2, 3$. It follows that we also have $I_g(\mathit{conf}, \mathit{constr}_1\ \mathit{constr}_2\ \mathit{constr}_2) = \mathit{conf}$. However, for an optimal algorithm $I$ we have $I(\mathit{conf}, \mathit{constr}_1\ \mathit{constr}_2\ \mathit{constr}_2) = \textbf{fail}$. This cannot be remedied: we are paying a price for the fact that we are using an approximate solution.

The consequence of this example is that the user may start from a situation without any failures, but regardless of how he refines the configuration, he ends up with failures. In other words, the reporting of some inconsistent selections are postponed until the user makes more choices. We do not know how rare or typical such phenonema are in actual models.

One way of minimizing the damage is to collect all such constraints in a set *warnings* and return them as warnings to the user. This does not make the algorithm optimal, but it high-lights the risks of non-optimality.

**Algorithm 3** *Define*

$$I'_g : \mathit{Conf} \times \mathit{Constrset} \rightarrow \mathit{Conf} \times \mathit{Constrset} \times \mathit{Constrset}$$

*by:*

```
1. input: conf and constrset = $\mathit{constr}_1, \ldots, \mathit{constr}_m$.
2. let failures= {} and todo= $\{\mathit{constr}_1, \ldots, \mathit{constr}_m\}$.
3. while there is a constr ∈todo:
       let todo=todo\{constr}.
       let $I_l(\mathit{conf}, \mathit{constr}) = \mathit{result}$.
       if result = fail then failures=failures ∪{constr} else
           let $E_1, \ldots, E_t$ be the entities refined from conf to result.
           let $\mathit{constr}'_1, \ldots, \mathit{constr}'_s$ be all constraints in which one or more of
           $E_1, \ldots, E_t$ occur.
```

```
        let todo=todo$\cup\{constr'_1, \dots, constr'_s\}$.
        let $conf = result$.
4. let $F_1, \dots, F_k$ be the entities mapped to **unspecified** in $conf$.
5. let $constr_{i_1}, \dots, constr_{i_l}$ be all constraints in which one or more of $F_1, \dots, F_k$
   occur.
6. let warnings=$\{constr_{i_1}, \dots, constr_{i_l}\}$.
7. return $conf$, failures, and warnings.
```

As mentioned above, $I'_g$ is not optimal, since some propogation is left out. However, the algorithm is also non-optimal in another sense: it never returns **fail**! In fact, the same applies to $I_g$. Rather than reporting **fail** when an unsatisfiable contraint is encountered, the algorithm reports an error and proceeds with the remaining constraints. In practice, we believe that this is the desired behaviour. In this sense, the algorithm resembles a type inference algorithm; the latter does not abandon the typing of a whole program when a sub-expression cannot be typed.

### 4.2 System Overview

The following should be the main functionality of a configuration system based on the above algorithm; the formulation of the three points are inspired by type-based specification, type-inference, and type-directed translation, respectively.

– *Editing*: There should be an editor in which the user can browse the current configuration, and in which the user can select or deselect an entity, or leave unspecified the select indication of an entity. This should happen through some presentation layer, as previously explained.
– *Inference*: There should be an inference engine that the user can invoke (or it is invoked automatically whenever the user changes the select indication of an entity). It propagates choices using $I'_g$, reporting suggestions (the refined configuration), failures (constraints that could not be satisfied), and possibly corrections and warnings. These should be mapped back through the presentation layer so that they are presented to the user at the same level as he made the original choices on.
  Notice that the inferred select indications are offered to the user as *suggestions*. He might decline to accept them. For instance, he might prefer to change some of the select indications that led to these suggestions. In other words, one can distinguish the following five select states:
  1. unspecified.
  2. selected.
  3. deselected.
  4. selected by the system as part of a propagation.
  5. deselected by the system as part of a propagation.

  In the two latter states the system has made a selection or deselection, which is presented to the user as a suggestion which he can accept or reject. Thus, an acceptance by the user of selection proposed by the system, corresponds to a shift from, say, state 4 to state 2.

However, it is important to understand that the system's reported failures for some constraints and the absence of failure for the other constraints assumes that all suggestions are accepted.
- *Code generation*: There should be a component translating a total configuration which makes all the constraints true into a procurement document (the totality and truth should be checked by the component).

The system can be tuned by only calling $I'_g$ with those constraints that contain one or more of the entities that the user has changed select indication for in the *todo* set. Another obvious improvement is to allow some short-cuts for the user such as a "deselect all unspecified entities" option. This is useful when the user is done selecting and deselecting, and just want to deselect everything else.

## 4.3 A Program Transformation Approach

The authors have recently presented a *program inversion* method called *explicitation* [32]. The method can invert a first-order function-oriented program with respect to a particular output. In general, the result of an inversion is a grammar that approximates the set of inputs that would result in the particular output. We can attempt to obtain an optimal inference algorithm by inverting a program that checks whether a constraint set is true in a given valuation:

```
 data Term = And Term Term | Or Term Term | Not Term |
Imply Term Term | Val Bool eval (And x y) = if eval x then
eval y else false eval (Or x y) = if eval x then true else
eval y eval (Imply x y) = if eval x then eval y else true
eval (Not x) = if eval x then false else true eval (Val x)
= x
```
(5)

This program expects as input a constraint set *constrset* in which the entities have been replaced according to a valuation $\nu$. It returns **true** or **false** depending on the value of $[\![constrset]\!]_\nu$.

Now, if we invert the checker program w.r.t. the output **true** and a constraint set in which the entities have been replaced by program variables, the result will be a description of all $\nu$ such that $[\![constrset]\!]_\nu = \mathbf{t}$.

*Example 2.* For instance, consider the constraint set

$$\begin{aligned} z &\Rightarrow v \wedge (y \vee x \vee x_2) \\ x_2 &\Rightarrow x_1 \wedge \neg w \\ w &\Rightarrow v \vee y \vee x \end{aligned} \tag{6}$$

where $x$, $y$, $z$, $v$, $w$, $x_1$ and $x_2$ are entities. The encoding of the above constraint would be

```
 And (Imply (Val z) (And (Val v) (Or (Val y) (Or (Val x)
(Val x2))))) (And (Imply (Val x2) (And (Val x1) (Not (Val     (7)
w)))) (Imply (Val w) (Or (Val v) (Or (Val y) (Val x)))))
```

where x, y, z, v, w, x1, and x2 are program variables. The result of inverting the checker program (5) w.r.t. constraint (7) will give us a grammar like the following.

$$S ::= [\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}] \,|\, [\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{f}] \,|\, \ldots \,|\, [\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}] \ . \qquad (8)$$

The grammar explicitly describes all valuations making the constraint set true. In all assignments, the entities are implicitly represented by a position in the list.

*Remark 4.* Inversion of the checker program w.r.t. any constraint set in which the entities have been replaced by program variables will always be a grammar that precisely lists the possible of the entities. Since the search space is finite, the same result could be obtained by using a functional-logic programming language directly.

Having obtained the set of valuations making the constraint set true, it is a trivial task to translate it into a set of total configurations by making the entities explicit and replacing **t/f** by **selected/deselected**. Let us call this set of configurations *valids*. The inference algorithm is then quite simple:

**Algorithm 4** Define

$$I_1 : \mathit{Conf} \times \mathit{Conf\,list} \longrightarrow \mathit{Conf} \cup \{\mathbf{fail}\}$$

by:

```
I₁ ($conf$,$valids$) =
    case filter ($conf$ ≤) $valids$ of $c :: cs$ ⟶ foldl msg $c$ $cs$
                                  | []     ⟶ fail
    where msg [ ][ ]= [ ]
          msg ( ($E$:$x$) ::$xs$) ( ($E$:$y$) ::$ys$) =
              ($E$:(if $x = y$ then $x$ else unspecified)):: msg xs ys
```

The functions ‘filter’ and ‘foldl’ are the standard functions on lists:

```
filter : ($\alpha \rightarrow$ Bool) $\rightarrow \alpha$ List $\rightarrow \alpha$ List
filter $f$ [ ]= [ ]
filter $f$ ($x$ :: $xs$) = if $f$ $x$ then $x$ :: (filter $f$ $xs$) else filter $f$ $xs$
foldl : ($\alpha \rightarrow \beta \rightarrow \beta$) $\rightarrow \beta \rightarrow \alpha$ List $\rightarrow \beta$
foldl $f$ $z$ [ ]= $z$
foldl $f$ $z$ ($x$ :: $xs$) = foldl $f$ ($f$ $x$ $z$) $xs$
```

This algorithm is very similar to $I_l$. The main differences are:

- the set of valuations making the constraint set true is computed in advance by inversion.
- among the resulting configurations those that are not refinements of the starting configuration are subsequently filtered away.
- the overall algorithm is expressed in functional rather than imperative style.

The *valids* set is in general exponential in the number of entities in the constraint set, so the space bound on the algorithm is

$$O(|conf|\,|valids|) = O(2^{entities}) \ , \tag{9}$$

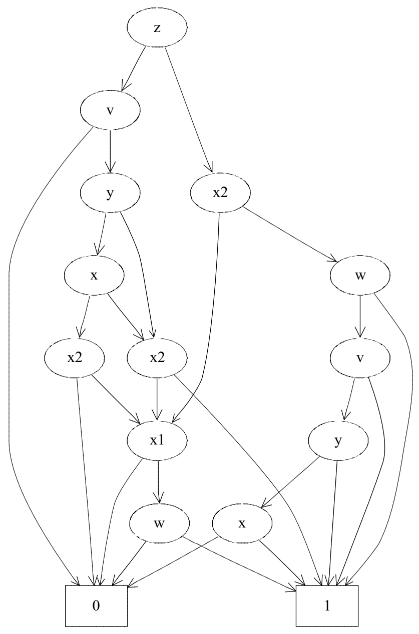
which makes this solution intractable, even though *valids* can be represented compactly as a vector of bit vectors.

Instead of obtaining all valuations making the constraint set true explicitly by inverting the checker program, we could *specialise* it with respect to the constraint set to get a program that can accept or reject total configurations. If the specialisation is done by one of the techniques collectively called *partial evaluation* [20], the result will be the following simple program:[7]

```
cond a b c = if c then cond0 a b else true
cond0 d e = if e then if d then false else true else false
main x y z v w x1 x2 =
    if z then if v
         then if y then cond w x1 x2
              else if x then cond w x1 x2
                   else if x2 then cond0 w x1
                        else false else false
    else if x2 then cond0 w x1
         else if w
              then if v then true
                   else if y then true else x
              else true
```

The interpretative overhead from the checker has been completely eliminated, and the program has become a simple, recursion-free, read-once boolean program. In fact, the program can be represented as the following *free binary-decision diagram* [40]:

[7] The specialised program has been produced by a variant [35, 33] of so-called *supercompilation* [36].



Every interior node is labelled with a variable and has two "legs," a solid and a dashed. Selecting a solid leg means assigning the value **t** to the variable, whereas selecting a dashed legs means assigning the **f** to the variable. Each path in a the diagram thus represents an assignment of truth values to the variables. If a path leads to the leaf **0**, it means that that particular assignment result in the value **f**; conversely **1** means **t**. If a variable is not mentioned on a particular path, it means that the value can be either **f** or **t**.

The merit of representing the constraint set this way, is that several total configurations can share common subconfigurations. This is the key observation that prompts us to cast the problem of optimal inference in a binary-decision-diagram setting in the next subsection.

### 4.4 Inference by BDDs

We will now encode the constraint set as a *reduced ordered binary-decision diagram* [7] (from now on simply called BDDs). The advantages of using BDDs are that there exist numerous efficient implementations, and that the inference problem can be solved by using only basic BDD operations, as we will explain below.

It is well known how to construct a BDD from a propositional formula, although the chosen variable (i.e. entity) order is crucial for the size of the BDD—in extreme cases a difference between linear and exponential size. But also optimal variable ordering is a hard problem: The best known algorithm runs in $O(n3^n)$, and even improving the variable ordering is NPTIME-complete, see [6, 16]; so heuristics are used in practice. In the following, we will simply assume that a near-optimal variable order has been obtained by some standard method (see, e.g.[21, 29, 11]). Given a constraint set $C$, we will denote the resulting BDD by $B_C$.

*Remark 5.* Calculation of a good variable ordering can be done once and for all when the constraint set $C$ is fixed.

The operations we will need are the 'restrict' operation, that specialises a BDD to a variable assignment, and the 'vars' operation that returns the list of variables in a BDD. Both run in time $O(|B_C|)$. We refer to the unique unsatisfiable BDD by '**0**', and similarly we refer to the unique valid BDD by **1**. Comparing other BDDs to **0** is a constant time operation. An optimal inference algorithm can now be cast as follows. An explanation follows the algorithm.

**Algorithm 5** Define

$$I_2 : \mathit{Conf} \times \mathit{BDD} \to \mathit{Conf} \cup \{\mathbf{fail}\}$$

by:

```
$\mathsf{I}_2$ $(conf, bdd)$ =
    case restrict $bdd$ $\nu_{conf}$
    of 0 $\longrightarrow$ fail
    | $bdd'$ $\longrightarrow$ foldl speculate $conf$ (vars $bdd'$)
        where speculate $E$ $conf$ =
                    if restrict $bdd'$ $\{E \mapsto \mathbf{t}\} = \mathbf{0}$
                    then $conf[E{:=}\mathbf{deselected}]$
                    else if restrict $bdd'$ $\{E \mapsto \mathbf{f}\} = \mathbf{0}$
                        then $conf[E{:=}\mathbf{selected}]$
                        else $conf$
```

where $\nu_{conf}$ is the translation of *conf* into a partial valuation, omitting the entities that are **unspecified**.

In the above algorithm, we first specialise the given BDD to the given configuration, possibly aborting if the resulting $bdd'$ is unsatisfiable. We then deduce impossible variable assignments by speculatively assuming that each variable $E$ is **t** or **f**: If, say, $E = \mathbf{t}$ results in an unsatisfiable BDD, then $E$ *must* be given the indication **deselected** to fulfil the constraints; we say that (the indication of) variable $E$ is *forced.* The forced variable indications are returned together with the original configuration.

**Proposition 9.** *Algorithm 5 is an optimal inference algorithm.*

*Proof.* By definition [7], there is only one unsatisfiable BDD, namely **0**. Therefore, if $bdd' =$ (restrict $bdd$ $\nu_{conf}$) is unsatisfiable, then $bdd' = \mathbf{0}$, and thus the algorithm **fail**s, as required. Otherwise, $bdd'$ is satisfiable and $\forall E \in$ (vars $bdd'$) : $conf(E) =$ **unspecified**. If there exists $E$ such that $\nu(E) = b$ for all valuations $\nu$ which make $bdd'$ true, then restrict $bdd'$ $\{E \mapsto \neg b\} = \mathbf{0}$, and thus 'speculate

$E$ *conf*' will infer the corresponding indication for $E$. Since 'speculate' is called for all variables in $bdd'$, every forced indication will be changed (from **unspecified**) in the original configuration. Conversely, if for any $E$ both restrict $bdd'$ $\{E \mapsto \mathbf{t}\} \neq \mathbf{0}$ and restrict $bdd'$ $\{E \mapsto \mathbf{f}\} \neq \mathbf{0}$, then there exist valuations $\nu$ and $\nu'$ making $bdd'$ true, where $\nu(E) = \mathbf{f}$ and $\nu'(E) = \mathbf{t}$. Thus, if an indication is *not* forced, it is not changed. Hence, the most specific partial configuration is returned. □

The running time of the algorithm is

$$O(|bdd| + |\text{var}\, bdd|\, |bdd|) \geq O(|bdd| \log |bdd|) \ , \qquad (10)$$

so there is certainly need for improvements, considering that the BDDs can be exponential in size of the constraint set. In practice, we expect constraint sets to be well-behaved, meaning that a good variable ordering can be found to reduce the size of the corresponding BDDs.

**Improvements** The deduction carried out by the function 'speculate' can be done more efficiently by a single breadth-first traversal of $bdd'$.[8] The key observation is that, if, say,

$$\text{restrict } bdd' \ \{E \mapsto \mathbf{t}\} = \mathbf{0} \ ,$$

then it holds for $bdd'$ that

1. every branch that leads to **1** will contain a node labelled $E$, and
2. every node labelled $E$ must have the **t**-leg connected to **0**.

It is relatively easy to see how the forced variable indications can be obtained by traversing the BDD with the help of priority queue.

**Algorithm 6** Let $Q$ be a priority queue that can contain nodes of the BDD such that the order of the nodes are dictated by the order of the variables.[9] The front of $Q$ will thus always be the nodes that are closest to the top of the BDD. Initially, $Q$ contains the single root node.

1. If $Q$ is empty, terminate.
2. Let the set $N$ be all the nodes at the front of $Q$ with the same variable label. Remove $N$ from $Q$.
3. If $Q$ is empty, condition 1 is true. If furthermore all nodes in $N$ have the same leg in **0**, condition 2 is true, and thus we have found a forced indication for a particular variable; output this indication.
4. Let $M$ be all the children of nodes in $N$. Remove all **0**-nodes from $M$.

[8] The function 'speculate' was suggested by Ken Friis Larsen (IT-C, Denmark) as a conceptual simplification of the more efficient function that follows.

[9] Recall that the variables in a BDD are totally ordered; the largest variable is the one labelling the root.

5. If $M$ contains **1**-nodes, the algorithm terminates, since condition 1 cannot be fulfilled by any of the remaining variables.
6. Add $M$ to $Q$, and repeat step 1.

In algorithm 6, each node is inserted at most once into $Q$, and each such insertion takes time $O(\log|Q_{\max}|)$ where $Q_{\max}$ is the largest $Q$. The running time of algorithm 6 is thus $O(|bdd|\log|Q_{\max}|)$. If we replace the traversal of all the variables in algorithm 5 with algorithm 6, the total running time of this improved algorithm is

$$O(|bdd| + |bdd|\log|Q_{\max}|) \leq O(|bdd|\log|bdd|) \ , \tag{11}$$

a clear improvement of the previous upper bound (10).

**Partitioning the Constraint Set** The upper bound $O(|B_C|\log|B_C|)$ is still not tractable, since $|B_C| \leq 2^{|C|}$. We might therefore look for properties of the constraint set that can make the inference more tractable. In particular, we could aim at building a set of much smaller BDDs instead of building one large BDD. Consider a constraint set $C$. What we are looking for is a partition of $C$ such that

$$\begin{gathered} C = C_1 \wedge C_2 \wedge \cdots \wedge C_n \\ \mathrm{I}_2(\mathit{conf}, B_C) \succeq \mathrm{I}_4(\mathit{conf}, [B_{C_1}, B_{C_2}, \ldots, B_{C_n}]) \succeq \mathit{conf} \end{gathered} \tag{12}$$

where

$$\mathit{conf} \overset{\text{def}}{\succeq} \mathit{conf}' \quad \text{iff} \quad \mathit{conf} \geq \mathit{conf}' \vee \mathit{conf} = \textbf{fail}$$

and

**Algorithm 7** Define

$$I_4 : \mathit{Conf} \times \mathit{BDD\ list} \rightarrow \mathit{Conf} \cup \{\textbf{fail}\}$$

by:

```
I_4 (conf, bdds) = fix (runthrough bdds) conf
where fix f d = let d' = f d in if d = d' then d else fix f d'
      runthrough bdds conf = foldl propagate conf bdds
      where propagate fail bdd = fail
            propagate conf bdd = I_2 (conf,bdd)
```

Algorithm 7 repeatedly applies algorithm 5 to each part $C_i$, accumulating new configurations until a fixpoint is reached.

The use of $\succeq$ in (12) allows us to select a partitioning of the constraint set which leads to non-optimal inference.

*Remark 6. Any* partitioning of the constraint set will fulfil (12), so choosing a particular partitioning only affects the precision of the inference.

Clearly, if the constraint set can be divided into *independent* parts, in the sense that each pair of parts have no variables in common, (12) will hold even if the first occurrence of $\succeq$ is replaced by $=$, and thus optimality is regained. Partitioning into independent parts has apparently been used successfully in the hardware-varification community. At the present time, we have not been able to find better partioning schemes that ensure optimality.

**Incremental BDDs** Algorithm 5, and thus algorithm 7, can easily be modified to avoid repeated re-specialisation of the initial constraint set $B_C$ by maintaining a state between user interactions. If the algorithm remembers the last configuration $\mathit{conf}_{i-1}$ as well as $\mathit{bdd}_{i-1}$ resulting from specialising $\mathit{bdd}'$ w.r.t. the inferred configuration (cf. algorithm 5) between each interaction with the user, then $\mathit{bdd}_{i-1}$ can be used instead of $B_C$ as long as each received configuration is a refinement of the previous one. Indeed, only the difference between configurations needs to be transmitted between the user and the inference algorithm. The constraint set thus decreases in size with each refinement.

Of course, when the algorithm receives a configuration that is *not* a refinement of the previous one, it is necessary to restart and use $B_C$ again.

Another merit of using the BDDs incrementally, is that default select indications could be represented as a total, satisfiable configuration $D$ : When the user has finished his selections, the defaults $D$ are used to further specialise the constraint set. Since no user-(de)selected entity is mentioned in the specialised constraint set, the default indication for such an entity is effectively ignored; only entities deferred by the user will be affected by the default indications.

**Explanations** Providing the user with an explanation of why a particular selection is unsatisfiable is also easier when the algorithm maintains a state (cf. previous section). Consider that the configuration $\mathit{conf}_{i-1}$ was accepted, but $\mathit{conf}_i$ is not accepted. In this case, the variables that constitute the difference between the configuration

$$\text{diffvars}\ \mathit{conf}\ \mathit{conf}' \overset{\text{def}}{=} \{E \mid \mathit{conf}(E) \neq \mathit{conf}'(E)\} \tag{13}$$

can be used to provide an explanation of why a configuration could not be accepted: The initial constraint set specialised to the most specific generalisation of the two configurations

$$\mathit{bdd}_{\text{msg}} = \text{restrict}\ B_C\ (\text{msg}\ \mathit{conf}_{i-1}\ \mathit{conf}'_i) \tag{14}$$

can be "existentially qualified" w.r.t. to the changed variables:

$$\mathit{bdd}_{\text{explanation}} = \text{exist}\ \mathit{bdd}_{\text{msg}}\ (\text{diffvars}\ \mathit{conf}_{i-1}\ \mathit{conf}_i))\ . \tag{15}$$

*Example 3.* Consider the constraint set (6) and assume the user has first made the selections

$$x : \textbf{deselected}\ y : \textbf{selected}\ (\text{the rest } \textbf{unspecified})$$

which succeeds. Assume now that the user changes his mind w.r.t. $y$ and makes additional selections

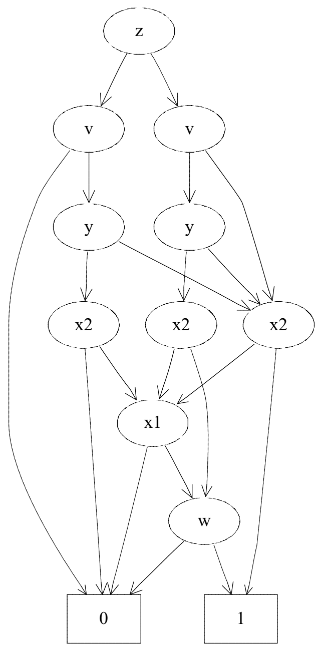
$x$ : **deselected** $y$ : **deselected** $z$ : **selected** $x_2$ : **deselected**
(the rest **unspecified**)

which leads to failure. The msg of the two configurations is simply

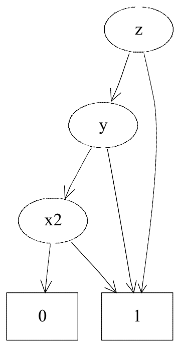
$x$ : **deselected** (the rest **unspecified**)

and the set of different variables is $\{y, z, x_2\}$. Hence,

$bdd_{\text{msg}} =$



$bdd_{\text{explanation}} =$



$bdd_{\text{explanation}}$ could then easily be converted to

$$z \Rightarrow (\neg y \Rightarrow x2) \ ,$$

which can be post-processed to the more understandable

$$z \Rightarrow y \vee x2 \ .$$

## 5 Related Work

The problems addressed in this work arose in the PROMIS project [8], and the corresponding solutions were developed based more or less on the work in the

AnnoDomini project [12, 13]. The paper [27] reports on a pre-study of a project with similar aims as the PROMIS project.

There is a significant literature on configuration, in particular a body of papers that view the configuration problem as a *constraint satisfaction problem* (CSP)—see, e.g. [31, 39]. The idea is that the configuration problem is formalized as the CSP $(X, D, C)$ where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = D_1 \cup \ldots \cup D_n$ is a domain made up of domains $D_i$ from which the values of variable $x_i$ must be drawn, and where $C$ is a set of constraints limiting the possible values of $x_1, \ldots, x_n$. The setting of the present paper can be seen as the special case $D_1 = \ldots = D_n = \{\mathbf{selected}, \mathbf{deselected}\}$.

However, this formalization does not address the aspects related to the dialogue between the user and the system. We are not simply interested in a solution to $(X, D, C)$. We are also interested in knowing if there is more than one solution, and if some values for variables are the same in all solutions given the user's current selections, and so on. Such issues are addressed in [1], which has similar aims as the present paper. Whereas we consider the user's current selection as a configuration, the former paper formalizes it as a set of unary constraints $H$. The problem then is to find a subset $E$ of $H$ such that $C$ and $H$ together have a solution. The user assigns weights to the constraints in $H$ indicating how desirable it is for him that a given constraint in $H$ is satisfied, and this is used to find an $E$ with maximal "customer satisfaction."

To this end the CSP is compiled into a finite automaton using the technique described in [38]. This compilation is done off-line, i.e. before questions are asked concerning the constraints, though the automaton must be updated when the set $H$ changes. In principle, the varibles of $X$ are ordered in some way, say, $x_1, \ldots, x_n$, and a search tree is constructed where the nodes at level $i$ correspond to the variable $x_i$, and where the arcs out of each node correspond to the possible values for the variable of the node, given the values for $x_1, \ldots, x_{i-1}$ determined by the path from the root to the node. This search tree can then be turned into a DFA or NFA, and questions concerning the original constraint set can be rephrased as questions pertaining to the automaton. These questions (at least some of them) can be answered in linear time in terms of the size of the automaton. However, this size can be exponential in terms of the constraint set.

In the present setting, the search tree amounts to a binary tree encoding all the possible valuations of a constraint, and the more compact NFA or DFA is in our setting a binary decision diagram.

A main difference between the work described in [1] and the present line of work is that Amilhastre and Fargier consider the situation that the user has made an inconsistent selection, and they use the weights assigned by the user to the constraints $H$ to find a relaxation of the user constraints that is consistent. In contrast, in the present line of work the user is presented with the conflicts, and he must then resolve the conflicts in some way himself. Both techniques seem to have advantages and disadvantages. This means that the questions asked by Amilhastre and Fargier concerning a given constraint set are somewhat different than the questions asked in this paper, though some of the questions are the same or closely related.

Another difference is that we consider boolean domains, whereas Amilhastre and Fargier consider more general domains. Again, both techniques seem to have advantages and disadvantages.

Gelle and Weigel [18] represent constraints by relations containing all tuples that satisfy the constraints, and argue that this representation is easier to maintain than a representation using explicity rules. In our setting this corresponds to representing constraints by truth tables.

In constraint terminology, *global consistency* means that for every variable in the CSP, there exists some value for it that participates in a solution to the problem. In our setting, this translates into plain consistency. In constraint terminology one also considers the notion of *arc-consistency* (see [2] for a good survey) meaning, roughly, that for every constraint $C$ involving variables $x$ and $y$, and every value $d_x$ currently considered valid for $x$, there exists a value $d_y$ considered valid for $y$, such that the constraint $C$ is satisfied. Arc consistency algorithms remove elements from the domains so as to ensure that all constraints are arc-consistent. Indeed, Algorithm 3 can be seen as a variation of the arc-consistency algorithm AC-3 [23, 22], where we consider $n$-ary constraints rather than binary ones, but where the domains are boolean rather than arbitrary finite domains.

There are other techniques for testing satisfiability, notably the *Davis-Putnam procedure* [10, 9]. A comparison between BDD-based techniques and the Davis-Putnam procedure appears in [37].

The company ConfigIt Software (www.configit-software.com) has developed a so-called *virtual table* technique [26] that seems to address the inference problems we have stated here. Their technique is not well described, however, since they have a pending patent on it.

Our inference algorithm on BDDs is akin to the Dilemma Rule used in Stålmarck's proof procedure for propositional logic [34].

## 6 Future Work

A natural next step of this work is to develop an implementation of the techniques described in the paper using efficient techniques, e.g. based on compilation to binary decision diagrams, to conduct experiments with a realistic case study, and to compare the performance to related techniques.

It would also be interesting to consider subclasses of the general set of contraints for which more efficient optimal algorithms can be devised, e.g. the class of horn formula—see [3]. Another idea is to refine Algorithm 3 inspired by other arc-consistency algorithms [25, 28, 19, 4, 5].

*Acknowledgments.* This paper grew out of work in the PROMIS project, which is joint work between the following companies: Terma A/S (Denmark), MSI and Thomson CSF (France), Intracom and HAI (Greece), Datamat and Marconi

(Italy), and MRC (Turkey). The authors are indebted to Steffen Mogensen, Terma A/S, for raising some of the issues discussed in the present paper and for discussions about them.

The solutions in this paper were more or less based on the work in the AnnoDomini project, which is joint work with Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, and Mads Tofte.

## References

[1] J. Amilhastre and H. Fargier. Handling interactivity in a constraint-based approach of configuration. In *Configuration Workshop at the 14th European Conference on Artificial Intelligence*, pages 7–12. Electronic proceedings available at url `www.cs.hut.fi/ pdmg/ECAI2000WS/Proceedings.pdf`, 2000.

[2] R. Barták. Theory and practice of constraint propagation. In *Proceedings of the 3rd workshop on Constraint Programming for Decision and Control*, 2001.

[3] K. A. Berman, J. Franco, and J.S. Schlipf. Unique satisfiability of horn sets can be solved in nearly linear time. In *Discrete Applied Mathematics*, volume 60, pages 77–91, 1995.

[4] C. Bessiere. Arc-consistency and arc-consistency again. *Aartificial Intelligence*, 65:179–190, 1994.

[5] C. Bessiere, E.C. Freuder, and J.-R. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Aartificial Intelligence*, 107:125–148, 1999.

[6] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] J.F. Challine. *PROMIS Concluding Report - Summary of the Study.* Thomson-CSF Communications, 2000.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[11] R. Drechsler, N. Göckel, and B. Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 730–739, Berlin, 1996. Springer.

[12] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H.B. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM press, 1999.

[13] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H.B. Sørensen, and M. Tofte. AnnoDomini in practice: A type-theoretic approach to the year 2000 problem. In *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science, pages 6–13. Springer-Verlag, 1999.

[14] B. Faltings and E. Freuder, editors. *Configuration. Papers from the 1996 Fall Symposium.* Number FS-96-03 in AAAI Fall Symposium Series. The AAAI Press, 1996.

[15] B. Faltings, E.C. Freuder, G. Friedrich, and A. Felfernig, editors. *Configuration. Papers from the AAAI workshop.* Number WS-99-05 in AAAI Workshop technical report series. The AAAI Press. Electronic proceedings available at url `wwwold.ifi.uni-klu.ac.at/ alf/aaai99/index.html`, 1999.
[16] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.
[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.
[18] E. Gelle and R. Weigel. Interactive configuration using constraint satisfaction techniques. In Faltings and Freuder [14], pages 37–44.
[19] P. Van Hentenryck, Y. Deville, , and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
[21] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, January 1993.
[22] A. K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
[23] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
[24] E. Mendelson. *Introduction to Mathematical Logic.* Wadswoth & Brooks/Cole Advanced Books and Software, third edition, 1987.
[25] R. Mohr and T.C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.
[26] J. Møller, H. R. Andersen, and H. Hulgaard. Product configuration over the internet. In *Proceedings 6th International INFORMS Conference on Information Systems and Technology*, Miami Beach, Florida, November 2001.
[27] K. Osvärn and O. Hansson. Prestudy of configuration of a naval combat management system. In Faltings and Freuder [14], pages 138–139.
[28] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.
[29] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE /ACM International Conference on CAD*, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
[30] J. Rahmer, A. Boehm, H.-J. Mueller, and St. Uellner. A discussion of internet configuration systems. In Faltings et al. [15], pages 138–140.
[31] D. Sabin and E. Freuder. Configuration as composite constraint satisfaction. In Faltings and Freuder [14], pages 28–36.
[32] J. P. Secher and M. H. Sørensen. From checking to inference via driving and dag grammars. In Peter Thiemann, editor, *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 41–51. ACM Press, 2002.
[33] J.P. Secher. Driving in the jungle. In Olivier Danvy and Andrzej Filinski, editors, *Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217. BRICS, Springer-Verlag, May 2001.

[34] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design: An International Journal*, 16(1):23–58, January 2000.

[35] H. Sørensen, M and R. Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.

[36] V.F. Turchin. Semantic definitions in Refal and the automatic production of compilers. In N.D. Jones, editor, *Workshop on Semantics-Directed Compiler Generation, Århus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 441–474. Springer-Verlag, 1980.

[37] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J. P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49. Springer-Verlag, 1994.

[38] N. R. Vempaty. Solving constraint satisfaction problems using finite state automota. In W. R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 453–458. American Association for Artificial Intelligence press/MIT press, 1992.

[39] M. Veron, H. Fargier, and M. Aldanondo. From CSP to configuration problems. In Faltings et al. [15], pages 101–106.

[40] I. Wegener. *The Complexity of Boolean Functions*. Wiley Teubner Series in Computer Science. John Wiley and Sons, New York, 1987.

# Author Index