Lecture Notes in Computer Science 1490
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Catuscia Palamidessi Hugh Glaser
Karl Meinke (Eds.)

Principles of Declarative Programming

10th International Symposium, PLILP'98
Held Jointly with the
6th International Conference, ALP'98
Pisa, Italy, September 16-18, 1998
Proceedings

Springer

Series Editors

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

Volume Editors

Catuscia Palamidessi
The Pennsylvania State University
Department of Computer Science and Engineering
University Park, PA 16802-6106, USA
E-mail: catuscia@cse.psu.edu

Hugh Glaser
University of Southampton
Department of Electronics and Computer Science
Highfield, Southampton, SO17 1BJ, UK
E-mail: hg@ecs.soton.ac.uk

Karl Meinke
Royal Institute of Technology/NADA
Osquars backe 2, S-10044 Stockholm, Sweden
E-mail: karlm@nada.kth.se

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Principles of declarative programming : proceedings / 10th International Symposium PLILP '98, held jointly with the 6th International Conference ALP '98, Pisa, Italy, September 16 - 18, 1998 / Catuscia Palamidessi ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1490)
ISBN 3-540-65012-1

CR Subject Classification (1991): D.1, D.3, F.3, F.4, I.2.1, I.1.3, K.3.2

ISSN 0302-9743
ISBN 3-540-65012-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10638855 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This volume contains the proceedings of the Joint International Symposium PLILP/ALP '98, which has united the Tenth PLILP (Programming Languages, Implementations, Logics and Programs) and the Seventh ALP (Algebraic and Logic Programming) conferences. This symposium was held in Pisa, Italy, during September 16–18, in cooperation with the Fifth Static Analysis Symposium (SAS '98), chaired by Giorgio Levi, and in conjunction with a number of satellite events.

The previous PLILP meetings took place in Orléans, France (1988), Linköping, Sweden (1990), Passau, Germany (1991), Leuven, Belgium (1992), Tallinn, Estonia (1993), Madrid, Spain (1994), Utrecht, The Netherlands (1995), Aachen, Germany(1996), and Southampton, UK (1997). All proceedings have been published by Springer-Verlag as Lecture Notes in Computer Science, volumes 348, 456, 528, 631, 714, 844, 982, 1140, and 1292, respectively.

The previous ALP meetings took place in Gaussig, Germany (1988), Nancy, France (1990), Volterra, Italy (1992), Madrid, Spain (1994), Aachen, Germany (1996), and Southampton, UK (1997). All proceedings have been published by Springer Verlag as Lecture Notes in Computer Science, volumes 343, 463, 632, 850, 1139, and 1298, respectively.

The PLILP symposia traditionally aim at stimulating research in declarative programming languages, and seek to disseminate insights in the relation between the logics of those languages, implementation techniques, and the use of these languages in constructing real programs. The ALP conferences traditionally promote the exchange of ideas and experiences among researchers from the declarative programming communities. In addition to the standard topics, the 98 Joint Symposium encouraged contributions also from other programming areas, in particular from the concurrent and object-oriented fields.

The program committee met electronically during the second and third week of May, 1998, and selected 26 papers out of 68 submissions (38%). Besides the selected contributions, the scientific program included three invited talks, by Charles Consel (University of Rennes/Irisa, France), Amir Pnueli (Weizmann Institute of Science, Rehovot, Israel), and Scott A. Smolka (SUNY Stony Brook, USA). The last two presentations were shared with SAS '98. In addition, the program included three tutorials, by Andrea Asperti (University of Bologna, Italy), John Hannan (The Pennsylvania State University, USA), and Andrew Pitts (University of Cambridge, UK). This volume contains all the selected papers, and the contributions (either in form of a short abstract or a full paper) of the invited speakers and tutorialists, except for the contribution of Amir Pnueli, which is included in the proceedings of SAS '98.

We would like to thank all those who submitted papers, for their interest in PLILP/ALP '98, the Program Committee members and their referees, for their careful work in the reviewing and the selection process, and the invited speakers and tutorialists, for their contribution to the success of the symposium.

Finally, we would like to express our gratitude to the members of the local committee, for the effort they have invested in organizing this event.

July 1998 Catuscia Palamidessi, Hugh Glaser and Karl Meinke

Program Committee

Organizing Committee

Maurizio Gabbrielli (Chair)

Gianluca Amato, Roberto Bagnara, Roberto Giacobazzi, Roberta Gori, Ernesto Lastres, Giorgio Levi, Rene Moreno, Francesca Scozzari, Fausto Spoto, and Paolo Volpe.

List of Referees

Maria Alpuente, Christian Attiogbe, Lennart Augustsson, Roberto Bagnara, Rudolf Berghammer, Gilles Bernot, Richard Bird, Urban Boquist, Antony Bowers, Francisco Bueno, Carlos Canal, Maura Cerioli, Iliano Cervesato, Manuel M. T. Chakravarty, Anna Ciampolini, Lars R. Clausen, Graham Collins, Marco Comini, Charles Consel, Thomas Conway, Agostino Cortesi, Vitor Santos Costa, Olivier Danvy, Giorgio Delzanno, Enrico Denti, Stephan Diehl, Agostino Dovier, Tyson Dowd, Kent Dybvig, Kerstin Eder, Christian Ferdinand, Gerard Ferrand, Gilberto Filé, Peter Flach, Gianluca Franco, John Gallagher, Maria del Mar Gallardo, Benedict R. Gaster, P. M. H. M. A. Gorissen, Frederic Goualard, Laurent Granvilliers, Andy Gravell, Steve Gregory, Gopal Gupta, John Hannan, Michael Hanus, Therese Hardin, Reinhold Heckmann, Fergus Henderson, Miki Hermann, Kohei Honda, Jan Hoogerbrugge, Herman ter Horst, Graham Hutton, Tetsuo Ida, Bart Jacobs, Tomi Janhunen, Barry Jay, Bharat Jayaraman, David Jeffery, Johan Jeuring, Thomas Johnsson, Paul H. J. Kelly, Richard Kelsey, Siau Cheng Khoo, Helene Kirchner, Herbert Kuchen, Francois Laburthe, Laura Lafave, Arnaud Lallouet, Evelina Lamma, Cosimo Laneve, Giorgio Levi, Michael Levin, Frank van der Linden, Jorge Lobo, Rita Loogen, Salvador Lucas, Simon Marlow, Luc Maranget, Maurizio Martelli, Erik Meijer, Aart Middeldorp, Michela Milano, Dale Miller, Juan Miguel Molina, Rafael Morales, Pierre-Etienne Moreau, Jon Mountjoy, Manfred Muench, Lee Naish, Zsolt Nemeth, Ilkka Niemelä, Ulrich Nitsche, Michael J. O'Donnell, Atsushi Ohori, Chris Okasaki, Javier Oliver, Andrea Omicini, Luke Ong, Wouter van Oortmerssen, Vincent van Oostrom, Bruno Pagano, David Pearce, Susanna Pelagatti, Javier Piris, Andreas Podelski, Norbert Podhorszki, Maria Jose Ramirez, Francesco Ranzato, Jakob Rehof, Gilles Richard, Fabrizio Riguzzi, Christophe Ringeissen, Kristoffer H. Rose, Salvatore Ruggieri, David E. Rydeheard, Amr Sabry, Mooly Sagiv, Masahiko Sakai, David Sands, Davide Sangiorgi, Frederic Saubion, Peter Schachte, Tony Scurr, Helmut Seidl, Peter Sestoft, Sylvain Soliman, Harald Søndergaard, Michael Sperber, Fausto Spoto, Cesare Stefanelli, Peter Stuckey, Taro Suzuki, Val Tannen, Simon Taylor, David N. Turner, German Vidal, Paolo Volpe, Fer-Jan de Vries, Philip Wadler, David Wakeling, Roland Yap, Hirofumi Yokouchi, Enea Zaffanella, and Hans Zantema.

Joint events

Chair: Roberto Bagnara

The SAS/PLILP/ALP '98 conference was held in conjunction with the following satellite events:

The Workshop on Principles of Abstract Machines.
Organized by Stephan Diehl and Peter Sestoft.

The 1st International Workshop on Component-based software development in Computational Logic.
Organized by Antonio Brogi and Patricia Hill.

The Second International Workshop on Verification, Model Checking and Abstract Interpretation.
Organized by Annalisa Bossi, Agostino Cortesi, and Francesca Levi.

The ERCIM Working Group on Programming Languages.
Coordinated by Neil Jones.

Sponsors of SAS/PLILP/ALP '98

Università di Pisa
Compulog Network
CNR-Gruppo Nazionale di Informatica Matematica
Consiglio Nazionale delle Ricerche
Comune di Pisa
Unione Industriali di Pisa

Table of Contents

Object Oriented Programming

Term Rewriting

Semantics: Tutorial

Functional Programming

Metaprogramming: Tutorial

Logic Programming and Model Checking*

Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar,
C. R. Ramakrishnan, I. V. Ramakrishnan, Abhik Roychoudhury,
Scott A. Smolka, and David S. Warren

Department of Computer Science, SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA
http://www.cs.sunysb.edu/~lmc

Abstract. We report on the current status of the LMC project, which seeks to deploy the latest developments in logic-programming technology to advance the state of the art of system specification and verification. In particular, the XMC model checker for value-passing CCS and the modal mu-calculus is discussed, as well as the XSB tabled logic programming system, on which XMC is based. Additionally, several ongoing efforts aimed at extending the LMC approach beyond traditional finite-state model checking are considered, including compositional model checking, the use of explicit induction techniques to model check parameterized systems, and the model checking of real-time systems. Finally, after a brief conclusion, future research directions are identified.

1 Introduction

In the summer of 1997, C.R. Ramakrishnan, I.V. Ramakrishnan, Smolka, and Warren were awarded a four-year NSF Experimental Software Systems (ESS) grant[1] to investigate the idea of combining the latest developments in concurrency research and in logic programming to advance the state-of-the art of system specification and verification. This was the first year of the ESS program at NSF, and its goal is to support experimental investigations by research teams dedicated to making fundamental progress in software and software engineering. The ESS program director is Dr. William W. Agresti.

The current primary focus of our ESS-sponsored project is *model checking* [CE81,QS82,CES86], the problem of determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [CW96b].

* Research supported in part by NSF grants CDA–9303181, CCR–9404921, CCR–9505562, CDA–9504275, CCR–9705998, CCR–9711386 and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

[1] There are two additional co-Principal Investigators on the grant who are no longer at Stony Brook: Y.S. Ramakrishna of Sun Microsystems and Terrance Swift located at the University of Maryland, College Park.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 1–20, 1998.

We call our approach to model checking *logic-programming-based model checking*, or LMC for short, and it is centered on two large software systems developed independently at SUNY Stony Brook: the Concurrency Factory [CLSS96] and XSB [XSB98]. The Concurrency Factory is a specification and verification environment supporting integrated graphical/textual specification and simulation, and model checking in the modal mu-calculus [Koz83] temporal logic. XSB is a logic programming system that extends Prolog-style SLD resolution with *tabled resolution*. The principal merits of this extension are that XSB terminates on programs having finite models, avoids redundant subcomputations, and computes the well-founded model of normal logic programs.

Verification systems equipped with model checkers abound. For example, `http://www.csr.ncl.ac.uk:80/projects/FME/InfRes/tools` lists over 50 specification and verification toolkits, most of which support some form of model checking. Although these tools use different system-specification languages and property-specification logics, the semantics of these logics and languages are typically specified via structural recursion as (least, greatest, alternating) fixed points of certain types of functionals.

It is therefore interesting to note that the semantics of negation-free logic programs are given in terms of minimal models, and Logic Programming (LP) systems attempt to compute these models. The minimal model of a set of Horn clauses is equivalent to the least fixed point of the clauses viewed as equations over sets of atoms. Hence, model checking problems involving least fixed points can be naturally and concisely cast in terms of logic programs. Problems involving greatest fixed-point computations can be easily translated into least fixed-point computations via the use of logical negation.

However, Prolog-style resolution is incomplete, failing to find minimal models even for datalog (function-free) programs. Moreover, the implementation of negation in Prolog differs from the semantics of logical negation in the model theory. Consequently, traditional Prolog systems do not offer the needed support to directly implement model checkers. As alluded to above, evaluation strategies such as tabling [TS86,CW96a] overcome these limitations (see Section 2.2). Hence, tabled logic programming systems appear to offer a suitable platform for implementing model checkers. The pertinent question is whether one can construct a model checker using this approach that is efficient enough to be deemed practical.

The evidence we have accumulated during the first year of our LMC project indicates that the answer to this question is most definitely "yes." In particular, we have developed XMC [RRR+97], a model checker for Milner's CCS [Mil89] and the alternation-free fragment [EL86] of the modal mu-calculus. The full *value-passing* version of CCS is supported, and a generalized prefix operator is used that allows arbitrary Prolog terms to appear as computational units in XMC system specifications. Full support for value-passing is essential in a specification language intended to deal with real-life systems such as telecommunications and security protocols.

XMC is written in approximately 200 lines of XSB tabled-logic-programming Prolog code, and is primarily intended for the model checking of finite-state systems, although it is capable of handling certain kinds of infinite-state systems, such as those exhibiting "data independence" [Wol86]. With regard to the efficiency issue, XMC is highly competitive with state-of-the-art model checkers hand-coded in C/C++, such as SPIN [HP96] and the Concurrency Factory [CLSS96]. This performance can be attributed in part to various aspects of the underlying XSB implementation, including its extensive support of tabling and the use of trie data structures to encode tables. In [LRS98] we describe how XMC can be extended to the full modal mu-calculus.

Buoyed by the success of XMC, we are currently investigating ways in which the LMC approach can be extended beyond traditional finite-state model checking. In particular, the following efforts are underway.

- An LMC-style specification of a model checker is given at the level of semantic equations, and is therefore not limited to any specific system-specification language or logic. For example, we have built a *compositional* model checker, simply by encoding the inference rules of the proof system as Horn clauses (Section 4.1).
- Traditionally, model checking has been viewed as an *algorithmic* technique, although there is a flurry of recent activity on combining model checking with *deductive* methods. Observe that (optimized) XSB meta-interpreters can be used to execute arbitrary deductive systems. Hence, the LMC approach offers a unique opportunity to fully and flexibly integrate algorithmic and deductive model checking, arguably the most interesting problem being currently researched by the verification community. To validate this claim, we have been experimenting with ways of augmenting XMC with the power of induction, with an eye toward the verification of *parameterized systems* (Section 4.2).
- By using constraints (as in Constraint LP [JL87]) to finitely represent infinite sets and tabled resolution to efficiently compute fixed points over these sets, we are looking at how tabled constraint LP can be used to verify *real-time systems* (Section 4.3).

The rest of the paper is structured follows. Section 2 shows how model checking can be essentially viewed as a problem of fixed-point computation, and how fixed points are computed in XSB using tabled resolution. Section 3's focus is our XMC model checker, and Section 4 describes ongoing work on extending the XMC technology beyond traditional finite-state model checking. After a brief conclusion, Section 5 identifies several promising directions for future research.

2 Preliminaries

In this section we describe the essential computational aspects of model checking and tabled logic programming. This side-by-side presentation exposes the primary rationale for the LMC project.

2.1 Model Checking

As remarked in the Introduction, model checking is essentially a problem of fixed-point computation. To substantiate this view, consider an example. Suppose we have a transition system T (transition systems are often used to model systems in the model-checking framework) and we wish to check if the start state of T satisfies the CTL branching-time temporal logic formula EFp. This will prove to be true just in case there is a run of the system in which a state satisfying the atomic proposition p is encountered.

Let S_0 be the set of states of T that satisfy EFp. If a state s satisfies p, written $s \vdash p$, then clearly it satisfies EFp. Further if s and t are states such that $t \vdash EFp$ and s has a transition to t, then $s \vdash EFp$ as well. In other words, if S is a set of states each of that satisfies EFp and $\mathbf{EFp} : 2^T \rightarrow 2^T$ is the function given by

$$\mathbf{EFp}(S) = \{s \mid s \vdash p\} \cup \{s \mid s \rightarrow t \wedge t \in S\}$$

then $\mathbf{EFp}(S) \subseteq S_0$. As a matter of fact, S_0 is the least fixed point of **EFp**. Thus, one way to compute the set of states that satisfy EFp is to evaluate the least fixed point of the function **EFp**. Assuming that T is finite-state, by the Knaster-Tarski theorem, it suffices to start with the empty set of states and repeatedly apply the function **EFp** till it converges.

There is a straightforward Horn-clause encoding of the defining equation of **EFp**. Tabled LP systems can evaluate such programs efficiently, and hence yield an efficient algorithm for model checking EFp. This observation forms the basis for the XMC model checker (Section 3).

Other temporal logic properties of interest may involve the computation of greatest fixed points. For example, consider the CTL formula AGp asserting that p holds at all states and along all runs. The set of states satisfying this formula (which turns out to be the negation of the formula $EF\neg p$) is the greatest fixed point of the function **AGp** given by:

$$\mathbf{AGp}(S) = \{s \mid s \vdash p\} \cap \{s \mid s \rightarrow t \Rightarrow t \in S\}$$

Once again, using Knaster-Tarski, the set of states satisfying the property AGp may be computed by starting with the set of all states and repeatedly applying the function **AGp** till it converges.

More complicated properties involve the nesting of least and greatest fixed-point properties and their computation becomes a more complex nesting of iterations. The modal mu-calculus, the logic of choice for XMC, uses explicit least and greatest fixed-point operators and consequently subsumes virtually all other temporal logics in expressive power.

2.2 The XSB Tabled Logic Programming System

The fundamental theorem of logic programming [Llo84] is that, given a set of Horn clauses (i.e., a "pure" Prolog program), the set of facts derivable by SLD

resolution (the computational mechanism of Prolog) is the same as the set of facts logically implied by the Horn clauses, which is the same as the set of facts in the least fixed point of the Horn clauses considered as set equations. From this result it might seem obvious that logic programming is well suited to solving general fixed-point problems and could be directly applied to model checking

This, however, is not the case. The foundational theorem on fixed points is weak in that it ensures only the existence of successful computations (SLD derivations); but there may be infinitely many computations that do not lead to success. This means that while a standard Prolog engine may be able to show that a particular fact is in the least fixed point of its rules, it can never show that a fact is not in the fixed point when there are infinitely many unsuccessful computation paths, which is the case for the fixed points needed for model checking. And even if the fact is in the fixed point, Prolog's search strategy may not find it. So even though the semantics of logic programming is a useful semantics for model checking, its computation mechanism of SLD is too weak to be practical.

The XSB system implements SLG resolution [CW96a], which to a first approximation can be understood as a tabled version of SLD resolution. This means that XSB can avoid rederiving a fact that it has previously tried to derive. (In the procedural interpretation of Horn clauses, this means that XSB will never make two calls to the same procedure passing the same arguments.) It is easy to see that for a system that has only finitely many possibly derivable facts, as for example in a finite-state model checking problem, XSB will always terminate.

To see how this works in practice, consider the following logic program:

```
reach(X,Y) :- trans(X,Y).
reach(X,Y) :- trans(X,Int), reach(Int,Y).
```

which defines reachability in a transition system. We assume that the predicate `trans(X,Y)` defines a transition relation, meaning that the system can make a direct transition from state X to state Y. Given a definition of the `trans` relation, these rules define `reach` to be true of a pair (X,Y) if the system can make a (nonempty) sequence of transitions starting in state X and ending in state Y. `trans` could be defined by any set of facts (and/or rules), but for our motivating example, we'll assume it is defined simply as:

```
trans(a,b).
trans(b,c).
trans(c,b).
```

Given a query of `reach(a,X)`, which asks for all states reachable starting from state `a`, Prolog (using SLD resolution) will search the tree indicated in Figure 1. The atoms to the left of the `:-` symbols in the tree nodes capture the answers; the list of atoms to the right are waiting to be resolved away. Each path from the root to a leaf is a possible SLD derivation, or in the procedural interpretation of Prolog programs are computation paths through the nondeterministic Prolog program. Notice that the correct answers are obtained, in the three leaves. However, the

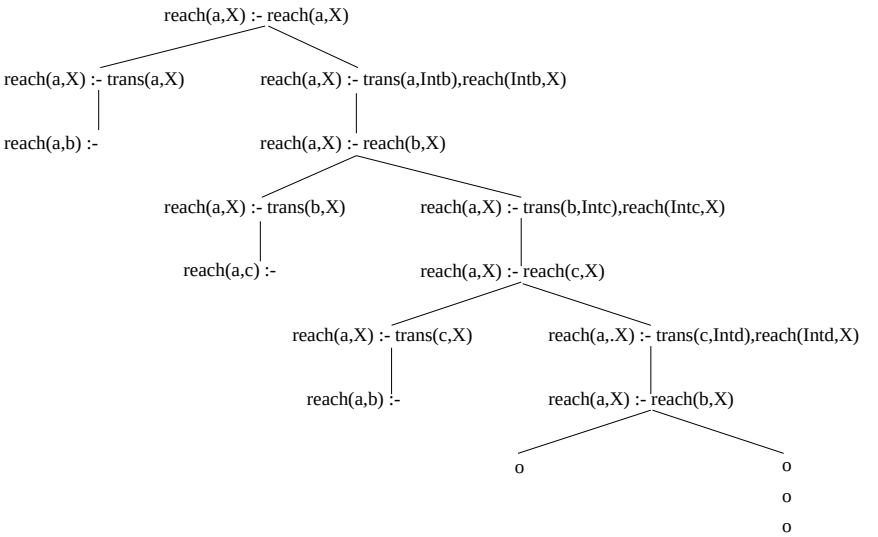


Fig. 1. Infinite SLD tree for reach(a,X)

point of more interest is that this is an infinite tree, branching infinitely to the lower right. Notice that the lower right node is identical to an ancestor four nodes above it. So the pattern will repeat infinitely, and the computation will never come back to say it has found all answers. A query such as `reach(c,a)` would go into an infinite loop, never returning to say that a is not reachable from c.

Now let us look at the same example executed using SLG resolution in XSB. The program is the same, but we add a directive `:- table reach/2.` to indicate that all subgoals of `reach` should be tabled. In this case during execution, an invocation of a `reach` subgoal creates a new subtree with that subgoal at its root if there is no such tree already. If there is such a tree, then the answers from that tree are used, and no new (duplicate) tree is created. Figure 2 shows the initial partial computation of the same query to the point where the subgoal `reach(b,X)` is about to be invoked, at the lower right node of that tree. Since

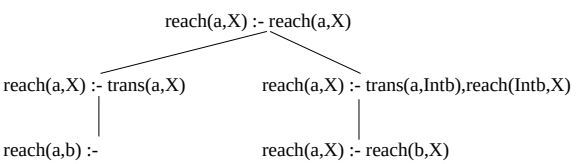


Fig. 2. Initial SLG subtree for reach(a,X)

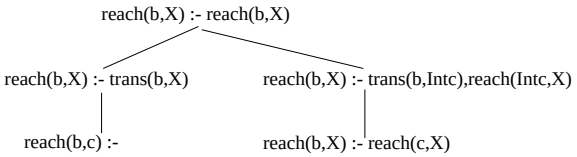


Fig. 3. Partial SLG subtree for reach(b,X)

there is no subtree for this subquery, a new one is created and computation continues with that one yielding another subtree, as shown in Figure 3. Now here

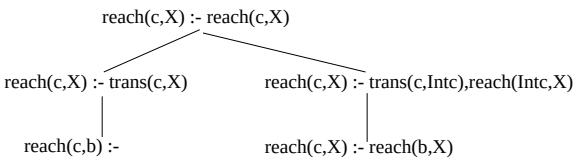


Fig. 4. Partial SLG subtree for reach(c,X)

again a new subgoal, `reach(c,X)`, is invoked, leading to a new subtree, which is shown in Figure 4. Here again we have encountered a subgoal invocation, this time of `reach(b,X)`, and a tree for this subgoal already exists; it is in Figure 3. So no more trees are created (at least at this time.) Now we can use answers in the subtrees to answer the queries in the trees that generated them. For example we can use the answer `reach(c,b)` in Figure 4 to answer the query of `reach(c,X)` generated in the lower rightmost node of Figure 3. This results in another answer in Figure 3, `reach(b,b)`. Now the two answers in the tree for `reach(b,X)` can be returned to the call that is the lower rightmost node of Figure 4, as well as to the lower rightmost node of Figure 2.

After all these answers have been returned, no new subgoals are generated, and the computation terminates, having reached a fixed point. The final state of the tree of Figure 2 is shown in Figure 5. The final forms of the other subtrees are similar.

This very simple example shows how tabling in XSB terminates on computations that would be infinite in Prolog. All recursive definitions over finite sets will terminate in a similar way. Finite-state model checkers are essentially more complicated versions of this simple transitive closure example.

3 Model Checking of Finite-State Systems

In this section we present XMC, our XSB-based model checker for CCS and the modal mu-calculus. We first focus on the alternation-free fragment of the

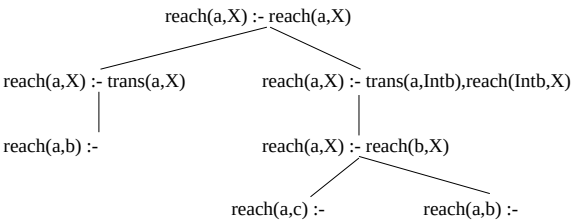


Fig. 5. The final SLG subtree for reach(a,X)

modal mu-calculus, to illustrate the strikingly direct encoding of its semantics as a tabled logic program. The full modal mu-calculus is treated next using a sophisticated semantics for negation developed within the logic-programming community. Finally, we show how the structural operational semantics of CCS, with full value-passing support, can also be naturally captured as a tabled logic program.

3.1 Model Checking the Alternation-Free Modal Mu-Calculus

The modal mu-calculus [Koz83] is an expressive temporal logic whose semantics is usually described over sets of states of labeled transition systems. We encode the logic in an equational form, the syntax of which is given by the following grammar:

$$F \longrightarrow Z \mid \texttt{tt} \mid \texttt{ff} \mid F \vee F \mid F \wedge F \mid \texttt{diam(}A\texttt{,}\ F\texttt{)} \mid \texttt{box(}A\texttt{,}\ F\texttt{)}$$
$$D \longrightarrow Z\ \texttt{+=}\ F \quad \text{(least fixed point)}$$
$$\mid Z\ \texttt{-=}\ F \quad \text{(greatest fixed point)}$$

In the above, Z is a set of formula variables (encoded as Prolog atoms) and A is a set of actions; `tt` and `ff` are propositional constants; $\vee$ and $\wedge$ are standard logical connectives; and `diam(`A`,` F`)` (possibly after action A formula F holds) and `box(`A`,` F`)` (necessarily after action A formula F holds) are dual modal operators. For example, a basic property, the absence of deadlock, is expressed in this logic by a formula variable `deadlock_free` defined as:

```
deadlock_free -= box(-, deadlock_free) /\ diam(-, tt)
```

where the '`-`' in `box` and `diam` formulas stand for any action. The formula states, essentially, that from every reachable state (`box(-,deadlock_free)`) a transition is possible (`diam(-,tt)`).

We assume that the labeled transition system corresponding to the process specification is given in terms of a set of facts `trans(`*Src*`,` *Act*`,` *Dest*`)`, where *Src*, *Act*, and *Dest* are the source state, label and target state, respectively, of each transition. The semantics of the modal mu-calculus is specified declaratively in XSB by providing a set of rules for each of the operators of the logic, as follows:

```
models(State_S, tt).

models(State_S, (F1 \/ F2))  :- models(State_S, F1).
models(State_S, (F1 \/ F2))  :- models(State_S, F2).

models(State_S, (F1 /\ F2))  :- models(State_S, F1), models(State_S, F2).

models(State_S, diam(A, F))  :- trans(State_S, A, State_T),
                                models(State_T, F).

models(State_S, box(A, F))   :- findall(T, trans(State_S, A, T),
                                States_L),map_models(States_L, F).
```

Consider the rule for `diam`. It declares that a state `State_S` (of a process) satisfies a formula of the form `diam(A, F)` if `State_S` has an `A` transition to some state `State_T` and `State_T` satisfies `F`.

The semantics of logic programs are based on minimal models, and accordingly XSB directly computes least fixed points. Hence, the semantics of the modal mu-calculus's least fixed-point operator can be directly encoded as:

```
models(State_S, Z)           :- Z += F, models(State_S, F).
```

To compute greatest fixed points in XSB, we exploit its capability to handle normal logic programs: programs with rules whose right-hand side literals may be negated using XSB's `tnot`, which performs negation by failure in a tabled environment. In particular, we make use of the duality $\nu X.F(X) = \neg\mu X.\neg F(\neg X)$, and encode the semantics of greatest fixed-point operator as:

```
models(State_S, Z)           :- Z -= F, negate(F, NF),
                                tnot(models(State_S, NF)).
```

The auxiliary predicate `negate(F, NF)` is defined such that NF is a positive formula equivalent to (¬`F`).

For alternation-free formulas, the encoding yields dynamically stratified programs (i.e., a program whose evaluation does not involve traversing loops with negation), and has a two-valued minimal model. In [SSW96] it was shown that the evaluation method underlying XSB correctly computes this class of programs. Tabling ensures that each explored system state is visited only once in the evaluation of a modal mu-calculus formula. Consequently, the XSB program will terminate under XSB's tabling method when there are a finite number of states in the transition system.

3.2 Model Checking the Full Modal Mu-Calculus

Intuitively, the *alternation depth* of a modal mu-calculus formula [EL86] `f` is the level of nontrivial nesting of fixed points in `f` with adjacent fixed points being of different type. When this level is 1, `f` is said to be "alternation-free". When this level is greater than 1, `f` is said to "contain alternation." The full modal mu-calculus refers to the class of formulas of all possible alternation depths.

In contrast to the alternation-free fragment of the modal mu-calculus, when a formula contains alternation, the resultant XSB program is not dynamically stratified, and hence the well-founded model may contain literals with unknown values [vRS91]. For such formulas, we need to evaluate one of the stable models of the resultant program [GL88], and the choice of the stable model itself depends on the structure of alternation in the formula. Such a model can be computed by extending the well-founded model. When the well-founded model has unknown values, XSB constructs a *residual program* which captures the dependencies between the predicates with unknown values.

We compute the values of these remaining literals in the preferred stable model by invoking the stable model generator *smodels* [NS96] on the residual program. The algorithm used in *smodels* recursively assigns truth values to literals until all literals have been assigned values, or an assignment is inconsistent with the program rules. When an inconsistency is detected, it backtracks and tries alternate truth assignments for previously encountered literals. By appropriately choosing the order in which literals are assigned values, and the default values, we obtain an algorithm that correctly computes alternating fixed points.

Initial experiments indicate that XMC computes alternating fixed points very efficiently using the above strategy, even outperforming existing model checkers crafted to carry out the same kind of computation. Details appear in [LRS98].

3.3 On-the-Fly Construction of Labeled Transition Systems

The above encoding assumes that processes are given as labeled transition systems. For processes specified using a process algebra such as CCS [Mil89], we can construct the labeled transition system *on the fly*, using CCS's structural operational semantics. In effect, we can treat `trans` as a computed (IDB) relation instead of as a static (EDB) relation, without changing the definition of `models`. Below, we sketch how the `trans` relation can be obtained for processes specified in XL (a syntactically sugared version of value-passing CCS), the process specification language of XMC.

The syntax of processes in our value-passing language is described by the following grammar:

$$
\begin{aligned}
E \longrightarrow\ & PN \mid \texttt{in(}A\texttt{)} \mid \texttt{out(}A\texttt{)} \mid \texttt{code(}C\texttt{)} \mid \texttt{if(}C\texttt{, }E\texttt{, }E\texttt{)} \\
& E\ \texttt{o}\ E \mid\ E\ \texttt{'||'}\ E \mid\ E\ \texttt{\#}\ E \mid\ E \setminus L \mid\ E\ \texttt{@}\ F \\
\mathit{Def} \longrightarrow\ & (\texttt{PN ::= E})^*
\end{aligned}
$$

In the above, E is a process expression; PN is (parameterized) process name, represented as a Prolog term; C is a computation, (e.g., `X is Y+1`); Process `in(a(`t`))` inputs a value over port `a` and unifies it with term t; `out(a(`t`))` outputs term t over port `a`. Process `if(`C`,` E_1`,` E_2`)` behaves like E_1 if computation C succeeds and otherwise like E_2. Operator `o` is generalized prefixing. The remaining operators are like their CCS counterparts (modulo occasional changes in syntax to avoid clashes with Prolog lexicon). For example, `#` is nondeterministic choice; `'||'` is parallel composition; `@` is relabeling, where F is a list of

substitutions; and '\' is restriction, where L is a list of port names. Recursion is provided by a set of *defining equations*, *Def*, of the form PN `::=` E.

The formal semantics of our language is given using structural operational semantics, closely paralleling that of CCS [Mil89]. Due to space limitations, we present here the axioms and inference rules for only a few key constructs. In order to emphasize the highly declarative nature of our encoding, these are presented exactly as they are encoded in the Prolog syntax of XSB.

```
trans(in(A), in(A), nil).
trans(out(A), out(A), nil).
trans(code(X), _, code) :- call(X).

trans(P1 o P2, A, Q) :- trans(P1, A, Q1),
                        (Q1 == code -> trans(P2, A, Q);
                            (Q1 == nil -> Q = P2 ; Q = Q1 o P2))).

trans(if(X,P1,P2),A,Q) :- call(X) -> trans(P1,A,Q) ; trans(P2,A,Q).

trans( P '||' Q,  A,  P1 '||' Q ) :- trans(P, A, P1).
trans( P '||' Q,  A,  P  '||' Q1) :- trans(Q, A, Q1).
trans( P '||' Q, tau, P1 '||' Q1) :- trans(P, A, P1),
                                     trans(Q, B, Q1), comp(A, B).

comp(in(A), out(A)).
comp(out(A), in(A)).

trans(P, A, Q) :-  P ::= R, trans(R, A, Q).
```

In the above, `A -> B ; C` is Prolog syntax for `if A then B else C`. The `trans` predicate is of the form `trans(P, A, Q)` meaning that process `P` performs an `A` transition to become process `Q`. The axiom for input says that `in(A)` can execute an `in(A)` transition and then terminate; similarly for the output axiom. The axiom for internal computation forces the evaluation of `X` and then terminates (without exercising any transition). The rule for generalized prefix states that `P1 o P2` behaves like `P1` until `P1` terminates; at that point it behaves as `P2`. The conditional process `if(X, P1, P2)` behaves like `P1` if evaluation of `X` succeeds, and like `P2` otherwise. Finally, the rules for parallel composition state that `P '||' Q` can perform an autonomous `A` transition if either `P` or `Q` can (the first two rules), and `P '||' Q` can perform a synchronizing `tau` transition if `P` and `Q` can perform "complementary" actions (the last rule); i.e., actions of the form `in(A)` and `out(A)`. The final rule above captures recursion: a process `P` behaves as the process expression `R` used in its definition.

To illustrate the syntax and semantics of XL, our value-passing language, consider the following specification of a channel `chan` (with input port `get` and output port `give`) implemented as a bounded buffer of size `N`.

```
chan(N, Buf) ::= code(length(Buf, Len)) o
                 if( (Len == 0)
                   , receive_only(N, Buf)
                   , if( (Len == N)
                       , send_only(N, Buf)
                       , receive_only(N, Buf) # send_only(N, Buf)
                 )).

receive_only(N, Buf) ::= in(get(Msg)) o chan(N, [Msg|Buf]).
send_only(N,Buf) ::= code(rm_last(Buf,Msg,RBuf)) o
                          out(give(Msg)) o chan(N,RBuf).
```

In the above definition `rm_last(Buf,Msg,RBuf)` is a computation, defined in Prolog, that removes the last message `Msg` from `Buf`, resulting in a new (smaller) buffer `RBuf`.

3.4 Implementation and Performance

The implementation of the XMC system consists of the definition of two predicates `models/2` and `trans/3`; in addition, it contains a compiler to translate input XL representation to one with smaller terms that is more appropriate for efficient runtime processing. Overall the system consists of less than 200 lines of well-documented tabled Prolog code.

Preliminary experiments show that the ease of implementation does not penalize the performance of the model checker. In fact, XMC has been shown (see [RRR+97]) to consistently outperform the Concurrency Factory's model checker [CLSS96] and virtually match the performance of SPIN [HP96] on a well-known set of benchmarks.

We recently obtained results from XMC on the i-protocol, a sophisticated sliding-window protocol used for file transfers over serial lines, such as telephone lines. The i-protocol is part of the protocol stack of the GNU UUCP package available from the Free Software Foundation, and consists of about 300 lines of C code.

Table 1 contains the execution-time and memory-usage requirements for XMC, SPIN, COSPAN [HHK96], and SMV [CMCHG96] applied to the i-protocol to detect a non-trivial livelock error that can occur under certain message-loss conditions. This livelock error was first detected using the Concurrency Factory.

Run-time statistics are given for window sizes $W = 1$ and $W = 2$, with the livelock error present (~fixed) and not present (fixed). All results were obtained on an SGI IP25 Challenge machine with 16 MIPS R10000 processors and 3GB of main memory. Each individual execution of a verification tool, however, was carried out on a single processor with 1.8GB of available main memory.

As can be observed from Table 1, XMC performs exceptionally well on this demanding benchmark. This can be attributed to the power of the underlying Prolog data structuring facility (the i-protocol makes use of non-trivial data structures such as arrays and records), and the fact that data structures in XSB are evaluated lazily.

Version	Tool	Completed?	Memory (MB)	Time (min:sec)
W=1 ~fixed	COSPAN	Yes	4.9	0:41
	SMV	Yes	4.0	41:52
	SPIN	Yes	749	0:10
	XMC	Yes	18.4	0:03
W=1 fixed	COSPAN	Yes	116	24:21
	SMV	Yes	5.3	74:43
	SPIN	Yes	820	1:02
	XMC	Yes	128	0:46
W=2 ~fixed	COSPAN	Yes	13	1:45
	SMV	No	28	>35 hrs
	SPIN	Yes	751	0:12
	XMC	Yes	68	0:11
W=2 fixed	COSPAN	Yes	906	619:49
	SMV	No	—	—
	SPIN	Yes	1789	6:23
	XMC	Yes	688	3:48

Table 1. i-protocol model-checking results.

4 Beyond Finite-State Model Checking

In Section 3 we provided evidence that finite-state model checking can be efficiently realized using tabled logic programming. We argue here that tabled LP is also a powerful and versatile vehicle for verifying infinite-state systems. In particular, three applications of tabled logic programming to infinite-state model checking are considered. In Section 4.1, we show how an XMC-style model checker can be extended with compositional techniques. Compositional reasoning can be used to establish properties of infinite-state systems that depend only on some finite subpart. Section 4.2 treats parameterized systems. Finally, in Section 4.3, the application of tabled LP to real-time systems is discussed.

4.1 Compositional Model Checking

Consider the XL process `A o nil`. Clearly it satisfies the property `diam(A,tt)`. We can use this fact to establish that `(A o nil) # T` also satisfies `diam(A,tt)`, without consideration of `T`. This observation forms the basis for *compositional model checking*, which seeks to solve the model checking problem for complex systems in a modular fashion. Essentially, this is accomplished by examining the syntactic structure of processes rather than the transition relation. (Recall, that the XMC model checker, presented in Section 3, does exactly the latter in its computation of the predicate `trans/3`.) Besides yielding potentially significant

efficiency gains, the compositional approach is intriguing also when one considers that the T in our example may well have been infinite-state or even undefined.

Andersen, Stirling and Winskel [ASW94] present an elegant compositional proof system for the verification of processes defined in a synchronous process calculus similar to Milner's SCCS [Mil83] with respect to properties expressed in the modal mu-calculus. A useful feature of their system is the algorithmic nature of the rules. The only nondeterminism in the choice of the next rule to apply arises from the disjunction operator in the logic and the choice of action in the process. Both of these sources of nondeterminism are unavoidable. In this respect, it differs from many systems reported in literature which require a clever choice of intermediate assertions to guide the choice of rules.

Andersen et al. in [ASW94] also present an encoding of CCS into their synchronous process calculus and consequently it is possible to use their proof system to verify CCS processes. This encoding, however, has two disadvantages. First, the size of the alphabet increases exponentially with the number of parallel components, and, secondly, the translation of the CCS parallel composition operator is achieved via a complex nesting of synchronous parallel, renaming, and restriction operators.

To mitigate the problems with their proof system in the context of CCS, we have adapted it to work directly for CCS processes under the restriction that relabeling operators use only injective functions. Our system retains the algorithmic nature of their system, yet incorporates the CCS parallel composition operator and avoids the costly alphabet blowup.

This adaptation is achieved by providing rules at three levels as opposed to two in [ASW94]. The first level deals with processes that are not in the scope of a parallel composition operator, the second for processes in the scope of a parallel composition operator, and the third for processes appearing in the scope of relabeling and parallel composition operators.

A Level-1 Rule:

```
models(P1 # P2, box(A,F)) :- models(P1, box(A,F)),
                             models(P2, box(A,F)).
```

A Level-2 Rule:

```
models((P1 # P2) '||'  P3, box(A,F)) :-
                       models(P1 '||' P3, box(A, F)),
                       models(P2 '||' P3, box(A, F)).
```

A Level-3 Rule:

```
models((B o P1) @ R '||' P2, box(A,F)) :-
                maps(R,B,C), models(C o (P1 @ R) '||' P2, box(A,F)).
```

Our system is sound for arbitrary processes and complete for finite-state processes. It has been implemented using XSB in the same declarative fashion as our XMC model checker. The compositional system is expected to improve on XMC's space efficiency by avoiding the calculation of intermediate states and

by reusing subproofs, though worst-case behavior is unchanged. Performance evaluation is ongoing.

Our compositional system can indeed provide proofs for properties of partially defined processes as illustrated by the following example from [ASW94]. Let `p ::= (tau o p) # T` and `q ::= (tau o q) # T` where `T` is an unspecified process. The formula `x += box(tau, x)` expresses the *impossibility* of divergence. The following is a proof that `p '||' q` may possibly diverge.

```
models(p '||' q, x)
   ?- models(p '||' q, box(tau, x))
      ?- models((tau o p) # T) '||' q, box(tau, x))
         ?- models((tau o p) '||' q, box(tau, x))
            ?- models(p '||' q, x)
               ?- fail.
```

4.2 Model Checking Parameterized Systems using Induction

We have thus far described how inference rules for a variety of verification systems can be encoded and evaluated using XSB. These inference systems specify procedures that are primarily intended for model checking finite-state systems. We now sketch how more powerful deductive schemes offer (albeit incomplete) ways to verify properties of *parameterized systems*. A parameterized system represents an infinite family of finite-state systems; examples include an n-bit shift register and a token ring containing an arbitrary number of processes.

An infinite family of shift registers can be specified in XL as follows:

```
reg(0) :== bit
reg(s(N)) :== (bit @ [get/temp] || reg(N) @ [give/temp]) \ {temp}
bit :== in(get)  o out(give)  o bit
```

In the above specification, natural numbers are represented using successor notation (`0`, `s(0)`, `s(s(0))`, ...) and `reg(`n`)` defines a shift register with $n + 1$ bits.

Now consider what happens when the query `models(reg(M),` ϕ`)`, for some nontrivial property ϕ and `M` unbound (thereby representing an arbitrary instance of the family), is submitted to an XMC-style model checker. Tabled evaluation of this query will attempt to enumerate all values of `M` such that `reg(M)` models the formula ϕ. Assuming there are an infinite number of values of `M` for which this is the case, the query will not terminate. Hence, instead of attempting to enumerate the set of parameters for which the query is true, we need a mechanism to derive answers that compactly *describe* this set.

For this purpose, we exploit XSB's capability to derive *conditional* answers: answers whose truth has not yet been (or cannot be) independently established. This mechanism is used already in XSB for handling programs with non-stratified negation under well-founded semantics. For instance, for the program fragment `p :- q. q :- tnot r. r :- tnot q.` XSB generates three answers: one for `p` that

is conditional on the truth of `q`, and one each for `q` and `r`, both conditional on the falsity of the other. Now, if `r` can be proven false independently, conditional answers for `q` and `p` can be *simplified*: both can be marked as unconditionally true.

Our approach to model checking parameterized systems is to implement a scheme to uncover the inductive structure of a verification proof based on the above mechanism for marking and simplifying conditional answers. Consider, again, the query `models(reg(M),` ϕ`)`. When resolving this query, we will encounter two cases, corresponding to the definition of the `reg` family: (i) `M = 0`, and (ii) `M = s(N)`, corresponding to the base case and the recursive case, respectively. For the base case, the model checking problem is finite-state and the query `models(reg(0),` ϕ`)` can be completely evaluated by tabled resolution.

For the recursive case, we will *eventually* encounter a subgoal of the form `models(reg(N),` ϕ'`)`, where ϕ' is some subformula of ϕ. For simplicity of exposition, consider the case in which $\phi = \phi'$. Under tabled resolution, since this subgoal is a variant of one seen before, we will begin resolving answers to `models(reg(N),` ϕ`)` from the table of `models(reg(M),` ϕ`)`, and eventually add new answers to `models(reg(M),` ϕ`)`. This naive strategy leads to an infinite computation. However, instead of generating (enumerating) answers for `models(reg(N),` ϕ`)` for adding answers to `models(reg(M),` ϕ`)`, we can generate one conditional answer, of the form:

```
models(reg(M), φ) :- M = s(N), models(reg(N), φ).
```

which captures the dependency between the two sets of answers. In effect, we have evaluated away the finite parts of the proof, "skipping over" the infinite parts while retaining their structure. For instance, in the above example, we skipped over `models(reg(N),` ϕ`)` (*i.e.*, did not attempt to establish its truth independently), and retained the structure of the infinite part by marking the truth of `models(reg(s(M)),` ϕ`)` as conditional on the truth of `models(reg(N),` ϕ`)`. Using this mechanism, we are left with a *residual program*, a set of conditional answers that reflects the structure of the inductive proof. The residual program, in fact, computes exactly the set of instances of the family for which the property holds, and hence compactly represents a potentially infinite set.

Resolution as sketched above, by replacing instances of heads (left-hand sides) of rules by the corresponding bodies (right-hand sides) *unfolds* the recursive structure of the specification. In order to make the structure of induction explicit, it is often necessary to perform *folding* steps, where instances of rule bodies are replaced by the corresponding heads. In [RRRS98] we describe how tabled resolution's ability to compute conditional answers, and folding mechanisms can be combined to reveal the structure of induction.

It should be noted that although we have a representation for the (infinite) set of instances for which the property holds, the proof is not yet complete; we still need to show that the set of instances we have computed covers the entire family. What we have done is to simply evaluate away the finite parts, leaving behind the core induction problem, which can then be possibly solved using theorem provers. In many cases, however, the core induction problem is simple

enough that the proof can be completed by heuristic methods that attempt to find a counter example, i.e., an instance of the family that is not generated by the residual program. For example, we have successfully used this counter-example technique to verify certain liveness properties of token rings and chains (shift registers), and soundness properties of carry look-ahead adders.

4.3 Model Checking Real-Time Systems

Another kind of infinite-state system we are interested in is *real-time systems*. Most embedded systems such as avionics boxes and medical devices are required to satisfy certain timing constraints, and real-time system specifications allow a system designer to explicitly capture these constraints.

Real-time systems are often specified as *timed automata* [AD94]. A timed automaton consists of a set of *locations* (analogous to states in a finite automaton), and a set of edges between locations denoting transitions. Locations as well as transitions may be decorated with constraints on *clocks*. An example of a timed automaton appears in Figure 6. A *state* of a timed automaton comprises a location and an assignment of values to clock variables. Clearly, since clocks range over infinite domains, timed automata are infinite-state automata.

Real-time extensions to temporal logics, such as timed extensions of the modal mu-calculus [ACD93,HNSY94,SS95], are used to specify the properties of interest.

Traditional model-checking algorithms do not directly work in the case of real-time systems since the underlying state-space is infinite. The key then is to consider only finitely many *regions* of the state space. In [AD94] it is shown that when the constraints on clocks are restricted to those of the form $X < Y + c$ where X and Y are clock variables and c is a constant, the state space of a timed automaton can be partitioned into finitely many *stable* regions—sets of indistinguishable elements of the state space.

For example, in the automaton of Figure 6, states $\langle L_0, t = 3\rangle$ and $\langle L_0, t = 4\rangle$ are indistinguishable. States $\langle L_0, t = 4\rangle$ and $\langle L_0, t = 6\rangle$, however, can be distinguished, since only from the latter can we make a transition to $\langle L_1, t = 6\rangle$, where an a-transition is enabled.

In [SS95], we presented a *local* algorithm for model checking real-time systems, where the finite discretization of the state space is done on demand, and only to the extent needed to prove or disprove the formula at hand. We can encode the essence of this algorithm with three predicates: `models/2`, which expresses when a region satisfies a timed temporal formula, `refinesto/2`, which relates a region with its partitions (obtained during finite discretization), and `edge/3`, which captures the transitions between regions.

Refinement adds a new inference rule to models:

$$\frac{R \texttt{ refinesto } \{R_i\}, \forall i \quad R_i \vdash F}{R \vdash F}$$

which is captured by the following Horn clause:

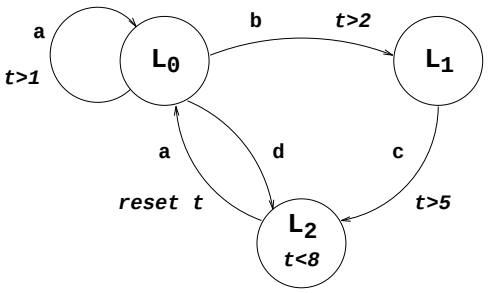


Fig. 6. Timed Automaton

```
models(R, F) :- refinesto(R, Refinements), map_models(Refinements, F).
```

Refinement creates new regions, and hence introduces new edges. The presence or absence of such edges may force further refinement. Therefore, `refinesto` and `edge` are mutually recursive predicates. Regions themselves are represented as a set of linear constraints, and operations on regions (such as splitting, which is needed when two points in a region can be distinguished) manipulate these constraints. Thus the resultant program is a *tabled* constraint logic program. Such programs can be evaluated in XSB using a meta interpreter, without modifying XSB's SLG-resolution engine. For better performance, however, we plan to directly augment the engine with a constraint-handling facility.

5 Conclusions

We have surveyed the current state of the LMC project, which seeks to use the latest developments in logic-programming technology to advance the state of the art of system specification and verification. In particular, the XMC model checker was discussed as well as several directions in which we are extending the LMC approach beyond traditional finite-state model checking.

Additional efficiency and ease-of-use issues are worthy of future investigation. First, since model checkers are specified at the level of semantic equations, equations of *abstract* semantics [CC77] can be encoded with equal ease. These can be used to incorporate process and formula abstractions, which have been used successfully to ameliorate state explosion in model checking [Dam96], into an LMC-style model checker. Secondly, the programmability of an LP system allows for direct encoding of traditional model-checking optimizations, such as partial order reduction [HPP96]. Finally, the high level at which model checking is specified correspondingly elevates the level at which erroneous system specifications can be diagnosed and debugged.

References

ACD93. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *Information and Computation*, 104(1):2–34, 1993. 17

AD94. R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, 126(2), 1994. 17

AH96. R. Alur and T. A. Henzinger, editors. *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer-Verlag. 19

ASW94. H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal mu-calculus. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, July 1994. 14, 15

CC77. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977. 18

CE81. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981. 1

CES86. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986. 1

CLSS96. R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In Alur and Henzinger [AH96], pages 398–401. 2, 3, 12

CMCHG96. E. M. Clarke, K. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic model checking. In Alur and Henzinger [AH96], pages 419–422. 12

CW96a. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996. 2, 5

CW96b. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996. 1

Dam96. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996. 18

EL86. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, 1986. 2, 9

GL88. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming*, pages 1070–1080, 1988. 10

HHK96. R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [AH96], pages 423–427. 12

HNSY94. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 1994. 17

HP96. G. J. Holzmann and D. Peled. The state of SPIN. In Alur and Henzinger [AH96], pages 385–389. 3, 12

HPP96. G. Holzmann, D. Peled, and V. Pratt, editors. *Partial-Order Methods in Verification (POMIV '96)*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, New Brunswick, New Jersey, July 1996. American Mathematical Society. 18

JL87. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987. 3

Koz83. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983. 2, 8

Llo84. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984. 4

LRS98. X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science*. Springer-Verlag, 1998. 3, 10

Mil83. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983. 14

Mil89. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 2, 10, 11

NS96. I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996. 10

QS82. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag. 1

RRR$^+$97. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag. 2, 12

RRRS98. A. Roychoudhury, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Tabulation-based induction proofs with applications to automated verification. In *Workshop on Tabulation in Parsing and Deduction*, 1998. 16

SS95. O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification*. American Mathematical Society, 1995. 17

SSW96. K. Sagonas, T. Swift, and D. S. Warren. An abstract machine to compute fixed-order dynamically stratified programs. In *International Conference on Automated Deduction (CADE)*, 1996. 9

TS86. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. MIT Press, 1986. 2

vRS91. A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 1991. 10

Wol86. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming*, pages 184–192, St. Petersburgh, January 1986. 3

XSB98. XSB. The XSB logic programming system v1.8, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog`. 2

CAT: The Copying Approach to Tabling

Bart Demoen and Konstantinos Sagonas

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
{bmd,kostis}@cs.kuleuven.ac.be

Abstract. The SLG-WAM implements tabling by freezing the WAM stacks: this technique has a reasonably small execution overhead, but is not easy to implement on top of an existing Prolog system. We propose a new technique for the implementation of tabling: the Copying Approach to Tabling. CAT does not interfere with normal Prolog execution and can be introduced in an existing Prolog system orthogonally. We have implemented CAT starting from XSB by taking out SLG-WAM and adding CAT. We describe the additions needed for adopting CAT in a WAM implementation. We show a case in which CAT performs arbitrarily worse than SLG-WAM, but on the other hand we present empirical evidence that CAT is competitive and often faster than SLG-WAM. We discuss issues related to memory management and the impact of the scheduling.

1 Introduction

Tabling in logic programming has been proven useful in a wide range of application areas such as parsing, deductive databases, program analysis based on abstract interpretation, and recently verification through model checking. The most practical implementation of tabling is found in XSB [11]: it also seems the only general Prolog system with tabling. The *Table Space* of XSB is organized using *tries* and table access mechanisms are optimised even further through *substitution factoring* [8]. Also, XSB currently implements two different scheduling strategies [5]. In this paper, we will be concerned mainly with the third aspect of tabling, which is the control and which is orthogonal to the other two. Control, i.e. the need to *suspend* and *resume* computations, is a main issue in a tabling implementation, because some subgoals, called *generators*, generate answers that go into the tables, while other subgoals, called *consumers*, consume answers from the tables; as soon as a generator depends on a consumer, the generator and the consumer must be able to work in a coroutining fashion, something that is not readily possible in a WAM implementation of Prolog. The execution of the query `?- p`$_g$`(X).` against the following small program exemplifies this situation.

```
:- table p/1.
p(1) :- p_c(Y).
p(2).
```

The subscripts g and c denote the occurrence of a subgoal that is a generator or consumer for this particular query. The answer `p(1)` cannot be generated

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 21–36, 1998.

before p_c has consumed the other answer, `p(2)`, from the answer table that p_g fills. At the moment that p_c consumes the answer `p(2)`, it must be in an execution state which is the same as when it was selected first. But, in a WAM implementation, backtracking has removed part of that state — because without backtracking, the answer `p(2)` could not have been generated — so the state of p_c must be preserved. The SLG-WAM [10], the abstract machine of XSB, preserves consumer states by *freezing* them, i.e. by not allowing backtracking to reclaim space on the stacks as is done in WAM. In implementation terms, this means that the SLG-WAM has a extra set of *freeze registers*, one freeze register for each of heap, trail, local and choice point stack.[1] Moreover, the trail needs to record also values because they have to be reinstalled together with the consumer.

This management of control through freezing slows down execution which is not related to tabling: this overhead is actually smaller than people usually assume (order of 10% for an emulated implementation [10]), but we will show in this paper that it can be completely avoided through the adoption of CAT. Moreover, freezing is complicated and not easy to put into an existing Prolog system: this fact might be the main reason why logic programming systems do not yet generally offer tabling. Also here, CAT saves the day because by using CAT, tabling can be added to a WAM implementation in an orthogonal way.

Instead of freezing the consumer state, one could imagine that the whole state of the abstract machine (i.e. all the stacks) is saved in a separate memory area, and then execution just fails over the consumer. When we need to reinstall the consumer, we can just revert to the saved copy and feed the consumer with its answers. This is not a good solution for two reasons: copying the whole WAM state is unnecessary (as we will show later) and it also leads to unnecessary recomputation. The execution of the query `?-` p_g`(X).` against the following program shows this.

```
:- table p/1.
p(1) :- b.     b :- pc(Y).
p(2).          b :- ... .
```

The state of the abstract machine at the moment p_c is called, contains the choice point for `b`. Still, the second alternative of `b` will have been exhausted by the time p_g generates its first answer (in this case `p(1)`). When we reinstall the consumer p_c, we do not want to reinstall the alternative for `b` as the ... represent an arbitrary amount of computation. Thus, a more selective copying of the WAM state can and should be done. CAT does exactly this: it copies selectively (and incrementally) execution states of consumers and reinstalls these copies when needed.

CAT does not require any changes to the WAM, only additions in the form of a few new instructions which can be alternatively seen as new built-in predicates. The choice points for tabled subgoals in CAT differ slightly from usual WAM

[1] Throughout this paper, we assume a WAM model with environment and choice point stacks separated (as XSB or SICStus Prolog implement) rather than combined as in the original WAM. We also assume that stacks grow downwards, i.e. higher in the stack means older, lower means younger.

choice points but this is not visible for non-tabled execution. Moreover, unlike SLG-WAM, no freeze-registers are needed for CAT, nor a more complicated trail. In short, CAT allows introduction of tabling into a WAM like implementation without any performance overhead. Therefore, we believe that CAT is an attractive approach to incorporate tabling in any logic programming system.

In the next section, a brief introduction to tabling and the SLG-WAM is given and some terminology is set. We then explain CAT step by step in situations of increasing complexity in Section 3. All along the additions to WAM are introduced and memory management is discussed. Section 4 discusses a worst case behaviour of CAT and a possible remedy. Section 5 discusses in more detail the CAT implementation and the relation between SLG-WAM and CAT. Section 6 compares the performance of CAT and SLG-WAM in the context of XSB. We end with an overview of related and future work.

2 Tabling and SLG-WAM: Concepts and Terminology

Due to space limitations, we only present concepts and terminology of tabled evaluation and of the SLG-WAM which are necessary to make the paper reasonably self-contained. We assume the usual terminology of logic programming and refer the reader to [2] for issues related to SLG resolution and to [10] for a detailed description of the SLG-WAM. Also due to space limitations we restrict ourselves to mainly definite programs and we keep the presentation informal.

2.1 Basic Overview of Tabling

A *tabled program* is a program augmented (automatically or by the programmer) with tabling declarations of the form: `:- table` $p_1/n_1, \ldots, p_k/n_k$`.` where p_i is a predicate symbol and n_i is an integer. These declarations ensure that all queries to the predicate p_i of arity n_i will be executed using tabled evaluation (e.g. SLG resolution). Other predicates are implicitly assumed as *non-tabled* in which case SLD resolution is used for queries to these predicates. Slightly abusing terminology, we will speak of tabled subgoals as well as tabled predicates. Following SLG resolution we will consider two tabled subgoals to be the same if they are *variants* of each other; i.e. identical up to variable renaming; however note that this is an orthogonal issue to the design of SLG-WAM or CAT. Tabled subgoals which are encountered in the evaluation of a query against a program are persistently stored in a global data structure called a *subgoal table*. When a tabled subgoal, s, is called, a check must be made to see whether s exists in the subgoal table or not. This is the purpose of the SLG NEW SUBGOAL operation. If s is new, it is termed a *generator*, it is entered in the table and will use PROGRAM CLAUSE RESOLUTION to derive answers. Through the NEW ANSWER operation, the set of derived answers of s will also be recorded in a global data structure called the *answer table* of s. Note that there is a one-to-one correspondence between generators and answer tables. If, on the other hand, (a variant of) s already exists in the table, it will resolve against answers from

its answer table. In this case, we call the subgoal a *consumer* of s. Answers are fed to the consumer one at a time through the ANSWER RETURN operation.

A basic concept in tabled evaluations is *completion* of (generator) subgoals and their associated answer tables. Informally, a subgoal s (and its answer table) is called *complete* if all its answers have been derived. On a slightly more operational level, through the SLG COMPLETION operation a subgoal can be determined as complete if all program and answer clause resolution has been performed for clauses of this subgoal. As there might exist dependencies between subgoals, it is often the case that subgoals cannot be determined complete on an individual basis, but their (mutual) dependencies also have to be taken into account. This suggests that the subgoal dependency graph DG has to be examined and *sets* of mutually dependent subgoals can be completed when they are involved in a *strongly connected component* Λ of DG that is *independent*: i.e. none of Λ's subgoals depends on a subgoal outside Λ. When all subgoals are completed, the evaluation has reached a fixpoint and stops.

2.2 Implementation of Tabling in the SLG-WAM

As one of the examples in the introduction shows, tabling cannot be implemented using the pure depth-first search of the WAM: this is mainly due to the fact that the generation and consumption of answers are asynchronous events. This means that an abstract machine for tabling has to maintain or reconstruct execution environments of consumers until these have consumed all answers that are generated for the subgoals; i.e. consumers have to be retained until fixpoint or completion of the associated generators. Likewise, newly derived answers must be queued to resolve against subgoals which do not necessarily correspond to the current execution environment. SLG-WAM offers a particular way to implement these features.

Suspending and Resuming Consumers in the SLG-WAM The SLG-WAM implements tabling by *suspending* consumers when these have exhausted all answers currently in the table and *resuming* them when new answers have been derived for them. Suspension is performed in SLG-WAM by creating a *consumer choice point* to represent the suspended environment, freezing all stacks by setting the freeze registers to point to the current top, and then failing to a previous choice point without reclaiming any stack space; in particular, the freeze registers are not reset. Space is not reclaimed above these freeze registers until completion of the appropriate generator. Resuming, besides restoring the WAM registers to the values saved in the consumer choice point, uses the addresses and the values saved in a *forward trail* [13,12] to restore variable bindings along the path to the suspended consumer; see [10] for exactly how this is done. An unconsumed answer is then returned to the restored consumer and execution continues by taking the forward continuation of the restored computation.

The purpose of freezing is that execution states of consumers are retained until the consumption of all their answers. So suspension interacts with completion: if upon consuming the last currently available answer of a subgoal, the

subgoal cannot be determined complete, i.e. that it has all its answers, the consumer needs to be suspended so that its execution environment is available if new answers are generated for it. If, on the other hand, a consumer of a complete subgoal is encountered, a *completed table optimization* can be performed: suspension through freezing is not necessary and the consumer can backtrack through the answers in the table as if they were program clauses stored as facts.

Scheduling and Incremental Completion in the SLG-WAM A *scheduling strategy* determines when answers are returned to consumers: SLG allows for many different scheduling strategies each of which can have different performance characteristics; see [5]. XSB currently implements two different scheduling strategies called *batched* and *local evaluation*. Both of them are based on partitioning the subgoals encountered during the course of an evaluation into *scheduling components*. As this partitioning also interacts with and is influenced by completion we examine scheduling and completion together.

In definite programs, completion (determining fixpoint) can be postponed till the end of the evaluation. However, for the SLG-WAM to reclaim space and thus be effective on large programs a more fine grained, incremental completion is needed. To efficiently perform incremental completion, the SLG-WAM (and CAT) contains an area of memory new to the WAM, the *Completion Stack*. The completion stack can be seen as a restriction of the choice point stack to just the choice points for generators and is used to efficiently keep track of dependencies between subgoals and of scheduling components. Specifically, the completion stack maintains, for each subgoal s, a representation of the older (highest) generator subgoal s_L upon which s or any subgoal below s *may* depend. This subgoal is called the *leader* of its scheduling component. When s and all subgoals younger than s have exhausted all program clause resolution, s can be checked for completion. If s is the leader of its component, A, and thus does not depend on subgoals higher in the stack than itself, if all consumers of subgoals in A have consumed all answers, then s and all other subgoals in A can complete and the (possibly frozen) space that corresponds to subgoals of the component can be reclaimed. Otherwise, if the leader s_L is higher in the completion stack than s, then s may depend upon subgoals that appear higher than itself on the completion stack, and execution backtracks to the previous alternative without reclaiming any space. This is the main idea behind the implementation of incremental completion in the SLG-WAM (see [10] for more details).

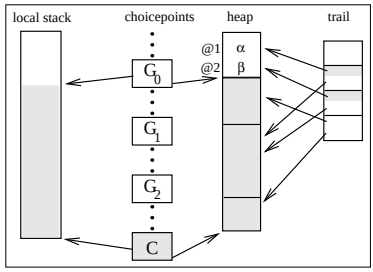
In both batched and local evaluation, scheduling of ANSWER RETURN operations is based on (and limited to the subgoals of) the component that is on the top of the completion stack. This explains why these are called scheduling components. More specifically, the leader of the topmost component is responsible for checking whether all answers have been returned to all consumers of subgoals that it leads, and schedule ANSWER RETURN operations if unresolved answers exist for some consumer. This check, called fixpoint_check in [10], is needed independently of whether each generator schedules its consumers or not after performing all program clause resolution and checking whether it can complete. Scheduling of consumers on failing back to the leader is always possible

as some scheduling strategies cannot determine fixpoint in a purely stack-based manner; see [10,5] for why this is so. To appreciate the design of CAT, it is thus important to keep in mind that consumers of subgoals of a scheduling component may need to be resumed when execution has failed back to the leader.

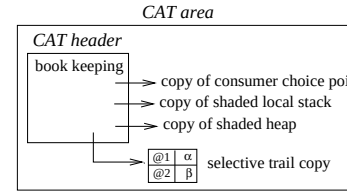
3 A Step by Step Introduction to CAT

3.1 A First Approximation: No Incremental Completion

Let us first make the approximation that incremental completion is not performed: there is only one scheduling component (for all subgoals), and the *single leader* does *all* the scheduling (for all consumers of subgoals that it leads) on failing back to it. In this setting consider how execution goes: At the moment a consumer is found, its consumer choice point is installed on the choice point stack. Fig. 1(a) shows the stacks: generator choice points G_0 (which is the leader) up to G_2, followed by the consumer choice point C. The vertical dots in between these choice points and above G_0 denote possible Prolog choice points, not related to execution of tabled predicates. The heap is shown segmented according to the tabling choice points and so is the trail. The same segmentation is not shown in the local stack, as it is a spaghetti stack. From the trail, some pointers point to cells older than the segment between G_0 and G_1: these cells have addresses @1 and @2 in the picture and the values in these cells are α and β.



(a) Stacks after creation of a consumer choice point: shaded parts are copied.



(b) The CAT area with selective trail.

Fig. 1. Conservative saving of a consumer state (without incremental completion).

Fig. 1(b) shows the information as saved by CAT: there is a (dynamically allocated) frame of fixed size which we name the *CAT header*. Apart from some bookkeeping fields, it contains a pointer to areas which contain copies of the shaded parts of the stacks: for heap and local stack, these shaded areas of Fig. 1(a) are copied as is; they are the part of heap and local stack created between the creation of G_0 and the consumer C. From the choice point stack, CAT only needs to save the consumer choice point: the justification is that at the moment C is scheduled to consume its answers, all the Prolog choice points

as well as the non-leader generator choice points will have exhausted their alternatives, and will have become redundant. This also means that when a consumer choice point is reinstalled, this can happen immediately below the leader G_0.

CAT copies the trail selectively as well: since CAT copies all of the heap between G_0 and C, there is no need to save the trail entries that point into this region and similarly for the trail entries that point to the saved part of the local stack. On the other hand, just saving the trail entries pointing to the older region of the heap (and local stack), is not enough to reinstall the consumer, because backtracking (up to G_0) will have undone the binding of say cell @1 to α. It means that CAT must also save the value α, or in general the current value of heap (and local stack) cells older than G_0 and pointed to by trail entries younger than G_0. Fig. 1(b) shows in more detail a saved trail. We call the CAT header together with the saved stacks, a *CAT area*.

After CAT has copied the consumer state, it removes the consumer choice point from the stack and activates a general failure in the WAM. Forward execution might then create other consumers (and CAT areas). Execution will eventually fail back to the leader G_0.

In this setting, after exhausting all alternatives of G_0, the leader must also make sure that the consumers consume their answers. The scheduling and restoration of consumers happens as follows: For each consumer, C, the saved portion of the heap and the local stack is copied back to its original place. Also, the saved values on the trail are reinstalled and the saved consumer choice point is copied just below G_0. This reinstalled consumer choice point can now start consuming answers from the tables as in SLG-WAM. After all currently available answers have been consumed, CAT must also update the *LastAnswer* field of the consumer choice point in the corresponding CAT area. Note that after reinstalling the consumer, the trail and choice point stack are in general smaller than at the moment of saving the consumer state. See Fig. 2(a).

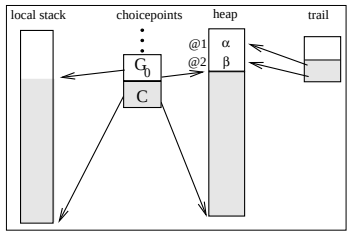
In the following sections we will refine CAT, but here already, we have laid out the basics for understanding CAT: the state of a consumer is saved by copying it; this copy consists of the parts of the heap and local stack between the consumer and a generator, a more selective trail copy and only the consumer choice point.

3.2 Adding Incremental Completion Based on Fixed Leaders

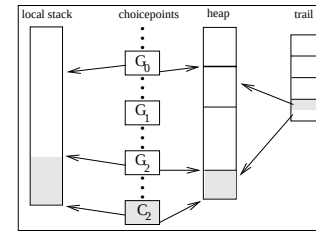
In the previous section, we assumed that completion was non-incremental and all subgoals belonged to one scheduling component led by a single leader. As mentioned in Section 2.2, for definite programs, this is a valid scheme for CAT as scheduling and completion can always be postponed till the end of the evaluation. However, as in the SLG-WAM, it is more efficient to perform incremental completion and have smaller scheduling components because then (1) CAT would copy and reinstall a smaller part of the stacks, and (2) subgoals can complete and free the CAT areas of their consumers earlier. Indeed, look at the execution of `?- p`$_g$`(X).` against the following program:

```
:- table p/1, q/1.
p(1) :- q_g(Y).     q(3) :- q_c(Z).
p(2).               q(4).
```

The answers of `q/1` do not depend on the answers of `p/1`, so a generator of `q/1` can be a leader and form a scheduling component in itself. It means that at the moment q_c`(Z)` is called, its leader is not p_g`(X)`, but the subgoal q_g`(Y)`, and the consumer q_c can always be scheduled on failing back to the generator q_g. So, in order to reinstall the state of q_c, it is enough to copy stacks between q_c and the leader q_g. Fig. 2(b) shows a variant of Fig. 1(a): the consumer is C_2 and its leader is G_2; the shaded areas to be copied are smaller than before. Note, however, that this schema is not practical because it assumes that the fixed leaders of scheduling components are known beforehand.



(a) The stacks just after the consumer has been restored.



(b) The leader is closer to the consumer: the copy is smaller.

Fig. 2. Reinstallation and incremental saving of consumer states.

3.3 A Coup: The Leader Changes

The principle of the previous two sections was: save consumer state up to the leader generator that might or will schedule the consumer. Even in cases where the fixed leaders are not known in advance, this works well until there is a change of leader: in practice a change of leader happens often. The query `?-` p_g`(X).` executed against the following program shows such a coup:

```
:- table p/1, q/1.
p(X) :- qg(Y), fail.  q(3) :- qc(X).
p(1).                 q(4) :- pc(X).
```

When the consumer q_c is saved, its leader (as maintained by dependencies kept in the completion stack) is the generator q_g. Indeed, at that point, it is not yet known that the answers of `q` will depend on answers of `p`. Later, at the moment the consumer p_c is created, a coup takes place: the generator q_g is no longer the leader of a scheduling component and p_g has become the leader of q_c. It means that in the future q_c might need to be restored by p_g, so the saved state of q_c should at restoration time contain also the part of the stacks between p_g and q_g. We could eagerly — at the moment of the coup — save this missing part and add it to the CAT area of q_c. The alternative is to wait until execution is about to backtrack over generator q_g: then, CAT saves the increment needed for q_c (i.e. the part of the stacks between p_g and q_g) and links it up to the CAT area of q_c. The advantage of waiting until this moment to save the increment, is

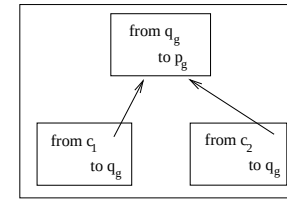
that there might be other consumers in the same need for an extension of their CAT area, and thus the part of the stacks between $\mathtt{p}_g$ and $\mathtt{q}_g$ can potentially be shared between all these consumers, instead of copying the same part for every one of them. This will become more clear below.

3.4 Incremental Copying of Consumer States

The previous section showed that in the absence of precise information about leaders and scheduling components, there is possibly a need for extending the saved state whenever the leader changes. Now consider the following program, which differs from the one before only by an extra (last) clause for `q/1`:

```
:- table p/1, q/1.   q(3) :- q_{c1}(X).
p(X) :- q_g(Y), fail. q(4) :- p_c(X).
p(1).                q(5) :- q_{c2}(X).
```

The two consumers for `q/1` have been given an index for ease of reference. At the moment consumer $\mathtt{q}_{c_1}$ is saved, copying happens up to the leader which is then $\mathtt{q}_g$; when consumer $\mathtt{q}_{c_2}$ is saved, the coup has happened already, meaning that for $\mathtt{q}_{c_2}$, we copy up to $\mathtt{p}_g$, the new leader. Later when backtracking happens over the generator $\mathtt{q}_g$, we copy the part between $\mathtt{q}_g$ and $\mathtt{p}_g$ so that this can be linked to consumer $\mathtt{q}_{c_1}$. Note though that $\mathtt{q}_{c_2}$ already contains that part of the stacks ! So, we have copied twice the same information from the stacks (the part between $\mathtt{p}_g$ and $\mathtt{q}_g$) and it is very difficult to avoid this in the schema which copies a consumer state up to its current leader. Since we have already the mechanism to link parts of saved states, we can use it in a more systematic way as follows: instead of saving a consumer state up to its leader, CAT always saves up to the closest generator G. When execution fails back to this generator, all consumers younger than G have copied all information needed for their restoration: they can be scheduled to consume their answers. If G is a leader of a scheduling component, on reaching fixpoint, completion can occur and the space for the CAT areas can be freed. Otherwise, since backtracking over this generator will occur, a new increment up to the previous generator is linked to all the consumers that need it. Applied to the above example, it means that the double copying of the old schema, does not happen anymore. This sharing of CAT areas between consumers is also shown schematically in the following rough picture. This incremental saving of consumer states, is the one finally implemented in CAT: it performs less copying by improving sharing of consumer states. CAT also allows for more flexible scheduling strategies since now even non-leaders can schedule consumers; moreover in the context of CAT, it is natural that a generator can schedule all consumers with a saved state that reaches up to this generator, not just its own.



4 The Main Problem of CAT and a Remedy

Consider the program:

```
:- table p/1.
p(X) :- produce(LargeTerm), p_c(Z), X = 2.
p(1).
```

The query `?- p_g(X).` produces the answers `p(1)` and `p(2)` and the consumer p_c is saved and restored once. `produce/1` is a computation which produces a large term as output argument. In the setting above, the consumer state contains this large term, so it is copied on saving and copied on restoring the consumer. So the CAT area for a consumer can be arbitrarily large, meaning that both saving and restoring a consumer can take arbitrarily long. The SLG-WAM does not suffer from this problem: freezing the stacks is a constant time operation and the cost of restoring a consumer in SLG-WAM is related to the trail. Since it is easy to construct examples in which consumers have to be restored arbitrarily often, it follows that CAT can be made to perform arbitrarily worse than SLG-WAM. However, this bad behaviour of CAT was not observed in any of the programs we have so far used tabling for. [2]

As it happens in the above example, this large term is not used in the continuation of the consumer, which means that copying it on saving the consumer was unnecessary. In general one can say that none of the heap entries which are not reachable for the continuation of a consumer, need to be saved. So, it seems worthwhile to perform a garbage collection that is restricted to the part of the heap to be copied, just before saving a consumer state. This is explored in more detail in [4] which also deals with a similar compaction of the local stack.

5 Implementation of CAT and Relation to SLG-WAM

We have implemented CAT within XSB; XSB itself implements SLG-WAM which freezes stacks as a means to save a consumer state. This means that XSB has a more complicated trail, trail test, setting of top-of-stack, etc. As a CAT prefers warmth, we have first *heated up* XSB, by removing all the freeze related code and replacing it with the plain WAM equivalent. In this warm version of XSB, we have then implemented CAT, reusing from XSB everything related to the implementation of incremental completion and the access and storing mechanisms for tabling of [8]. Besides adding the incremental CAT copying and restoration described in Section 3.4 and slightly modifying the scheduling (as will be explained below), only instructions related to tabling (tabletry, answer_return, ...) which do not belong to the WAM, were changed. So, now there exist two versions of XSB: one that implements SLG-WAM and one that implements CAT. These will form the basis of a comparison between SLG-WAM and CAT. Since to our knowledge, there exists no other implementation of SLG-WAM (nor of CAT) we will henceforth refer to the two mechanisms as if they were implementations.

[2] The bad figures for read_o in Table 1 and 2 have another reason: see Section 6.

Current Similarities and Differences with the SLG-WAM In our first version of CAT we have retained the incremental completion algorithm of the SLG-WAM. As mentioned, in definite programs, completion can always be postponed and it is mainly used for space reclamation. As the SLG-WAM freezes the stacks at the top, non stack-based selective completion (and space reclamation) of subgoals is not possible without fragmenting the stacks and thus requires stack compaction. As this is probably costly, the SLG-WAM always completes the component in which the youngest generator belongs: therefore, the SLG-WAM uses a completion algorithm based on *approximate* subgoal dependencies. This algorithm is known to trap subgoals in scheduling components (see [10]) and may arbitrarily postpone their completion despite the fact that they are independent of other subgoals. CAT, as a true feline, is much more flexible in completion because reclamation of the CAT areas is not based on compaction. We plan to add an exact completion algorithm to our CAT, but for fairness, in this paper we compare CAT and SLG-WAM under the same completion algorithm.

Unlike the SLG-WAM, CAT reclaims the trail and choice point stack even before completion of a component.[3] In particular, it reclaims generator choice points for non-leaders when these have exhausted program clause resolution. This means that the substitution factor variables (an optimization of [8]) cannot be stored in the choice point stack as in the SLG-WAM, but in a place that survives backtracking until completion: we have opted for the heap. As another small technical point, since non-leader generators are reclaimed by CAT, local scheduling is implemented by creating a CAT area for generators that are not leaders. This is in accordance with the definition of local evaluation that specifies that these generators behave as consumers (cf. [5]).

Finally, as noted, CAT allows for more flexible scheduling algorithms, in particular more fine-grained ones. However, the current version only performs scheduling on failing to the leader in a manner similar to the fixpoint_check of the SLG-WAM. A difference with the SLG-WAM is that scheduling decisions are taken more often by CAT since CAT can reinstall only one consumer at a time, while SLG-WAM can schedule several consumers in one go.

Extending CAT to normal logic programs The XSB implementation of the SLG-WAM evaluates programs according to the well-founded semantics. There is really nothing that prevents CAT to also work for this class of programs as the handling of negation is an orthogonal issue. The same low-overhead (around 1%) mechanisms that the SLG-WAM uses to maintain exact subgoal dependencies [10], to detect and break cycles through negation can be combined with CAT as well. Space limitations prohibit a full discussion but we simply note that the handling of negative (tabled) literals in CAT is analogous to the handling of consumer subgoals: the execution state of negative literals is also preserved in a CAT area by copying. A small difference is that a *negation suspension frame* [10] rather than a consumer choice point is copied and that this frame is reinstalled *once* and *at the place of the generator choice point* (upon its completion) rather than immediately below it.

[3] Actually, it reclaims all stacks, but parts of heap and local stack stay in CAT areas.

6 Performance Evaluation

As expected, CAT performs better than SLG-WAM in Prolog code; around 10% according to our measurements (see also [10]). Also, CAT and SLG-WAM have indistinguishable performance in artificial tabling benchmark programs from the database community like transitive closures over chains, cycles and trees of various lengths and same generation over cylinders (cf. [11,10,5]). So we compare CAT and the SLG-WAM on more realistic sets of programs from an application area where tabling has been proven worth having in a general purpose logic programming system: abstract interpretation. All measurements were conducted on an Ultra Sparc 2 (168 MHz) under Solaris 2.5.1. Times are reported in seconds, space in KBytes. Space numbers measure the maximum use of the stacks (for SLG-WAM) and the total of max. stack + max. CAT area (for CAT).

6.1 A Benchmark Set Dominated by Tabled Execution

The first benchmark set is taken from [3]: the programs perform type analysis by program abstraction and execution of the abstracted program under tabled evaluation. Tabling is used both for termination, efficient storage of the analysis results and to avoid redundant subcomputations in the domain-dependent abstract operations (see [3]). With the exception of a few utility predicates like `append/3`, all other predicates are tabled in this benchmark set and the size of the table space (not shown) is quite large: this set of benchmarks programs is heavily dominated by tabling operations.

Tables 1 and 2 show time and space performance of the analysis under the two scheduling strategies of XSB. On this set of programs, SLG-WAM performs more or less the same time-wise with batched (B) and local (L) strategy with a very noticeable advantage for the local strategy in space consumption as its scheduling components are tighter. CAT under local scheduling performs slightly better than SLG-WAM in time and slightly worse in space. CAT under batched scheduling is slower than the SLG-WAM for the last three benchmarks by 15–60%: these are also the benchmarks for which CAT uses more than 10 times more space than SLG-WAM. This behaviour is mostly due to the approximate completion algorithm that is used. In this benchmark set, a high percentage of the tabled subgoals, and more specifically the abstract operations, is semi-det:

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	0.23	0.45	0.13	0.17	0.15	0.44	0.12	0.58
CAT(B)	0.22	0.41	0.13	0.15	0.14	0.50	0.15	0.92
SLG-WAM(L)	0.23	0.43	0.13	0.16	0.16	0.42	0.12	0.61
CAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.55

Table 1. Time performance of CAT vs. SLG-WAM under batched & local scheduling.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	9.7	11.4	8.8	20.6	40	317	119	512
CAT(B)	13.6	19.4	11.7	45.3	84	3836	1531	5225
SLG-WAM(L)	6.7	7.6	5.8	17.2	13.3	19	15.8	93
CAT(L)	7.9	10.7	7.1	29.5	12.5	17	23.5	246

Table 2. Space performance of CAT vs. SLG-WAM under batched & local scheduling.

i.e. produces at most one answer. Because completion is based on an approximation of subgoal dependencies, these semi-det subgoals often get trapped in an approximate scheduling component and cannot be completed on their own. As it is not known whether new answers will be derived for these subgoals, their consumers have to suspend (and create a CAT copy) rather than use the completed table optimisation (see Section 2.2). Some extra measurements for the read_o program shed more light: under batched scheduling 3112 consumers are saved (max. CAT area of 5208 KBytes) and only 192 times a consumer is restored (totaling 371 KBytes) to consume new answers; in contrast, the corresponding numbers for local scheduling are 264 consumers saved and 224 restorations (240 and 371 Kbytes respectively — the bigger restoration is due to sharing of the saved space by incremental copying, but not by restoration). With the current approximate completion algorithm, under batched scheduling, CAT performs worse than SLG-WAM. We strongly believe that for this benchmark set CAT will be more competitive to SLG-WAM (under batched scheduling) if completion is based on exact subgoal dependencies (cf. also Section 5).

6.2 A more Realistic Mix of Tabled and Prolog Execution

The second set of benchmarks is taken from [6]. These programs perform abstract interpretation. The abstract operations are implemented in Prolog and occupy a high percentage of the total execution time (around 75–80%). Regardless of the issue SLG-WAM vs. CAT, only the local scheduling makes sense in this program set because the analyses are based on an abstract least upper bound operation, resulting in a very poor performance for the batched strategy (see also [5]). Tables 3 and 4 compare SLG-WAM and CAT in time and space. Overall, CAT performs time-wise slightly better (5–20%) in this set. It performs considerably better (25–140%) than SLG-WAM in space with only two exceptions.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	1.48	0.67	1.11	2.56	9.64	32.6	1.17	3.07	10.02	7.61	9.01
CAT	1.24	0.62	0.97	2.50	9.56	32.2	0.83	2.75	9.96	6.38	8.54

Table 3. Time performance of CAT vs. SLG-WAM.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	998	516	530	472	5186	9517	279	1131	2050	1456	1784
CAT	552	223	206	486	8302	7847	227	821	1409	1168	1373

Table 4. Space performance (in KBytes) of CAT vs. SLG-WAM.

7 Related Work

There is an analogy between the SRI-model [12] for implementing OR-parallelism and MUSE [1] on one hand, and the SLG-WAM for implementing tabling and CAT on the other: like the SLG-WAM, the SRI-model has a complicated management of the stacks and switching from one worker to another — the analogue of suspending one consumer to resume another one — uses a trail structure that is more complicated than the WAM trail, because bindings have to be undone as well as reinstalled. Like MUSE, CAT avoids complicated stacks by copying the portion of the stacks that is particular to a consumer or in the case of MUSE a worker. We believe this analogy is so strong, that although not conscious at the time, we must have been influenced by our knowledge of MUSE when getting the idea for CAT. [1] notes that the overhead of copying is small compared to all other work to be performed and our experience with CAT is similar.

In [9] a design for combining tabling and or-parallelism was presented. The implementation of tabling was based on the SLG-WAM, the only available WAM-based model of implementing tabling at that time, while or-parallelism was supported through environment copying as in the MUSE model. In view of the above similarity between CAT and MUSE, we believe that CAT also offers a more natural way of combining tabling and or-parallelism as the same basic machinery can be used for satisfying the implementation requirements of both forms of suspend/resume mechanisms.

Another related technique based on copying is described in [7]. It provides a set of library functions for introducing backtracking in C programs: the parts of the C-stack that must become active again after failure are copied incrementally. This is achieved by changing at run time the return address in activation records. In contrast, in CAT the zones to be copied are delimited by invocations of tabled predicates, so that by compiling these with special instructions, such a change is performed effectively at compile time.

8 Conclusion and Future Work

Since any logic programming system can benefit from having tabling, it has been our long term goal to make tabling more accessible and more attractive to add to existing systems. It always seemed to us that the main obstacle to this were some aspects of the SLG-WAM: in particular the machinery that the SLG-WAM imposes on the WAM to implement the suspend/resume mechanism needed for tabling. This machinery is quite complicated and unnecessarily slows down the underlying basic system. The other issues in tabling implementations — the

tabling data structures and the scheduler — were always orthogonal and do not affect the underlying implementation. We have shown that CAT is a true alternative to SLG-WAM for implementing the control that tabling requires. CAT is simpler to add and it has no influence on the efficiency of the underlying implementation. In addition, CAT helps in reasoning about reachability of objects in the execution tree and it seems that it allows more easily for more flexible scheduling strategies. Empirical tests show that most of the benchmark programs are not slowed down by the CAT technique compared to the implementation of SLG-WAM in XSB and that under the local evaluation strategy, the memory consumption is most often better than under SLG-WAM. This might come as a surprise, but the simplicity of CAT is the key to its performance.

However, to fully achieve our goal, at least the following are needed: a clear definition of the interface between the tabling components and a general purpose LP system; a complete integration of CAT in the memory management of a LP system (a forthcoming paper deals with this [4]); and a better understanding of the interaction of CAT with scheduling strategies.

Acknowledgements

The second author is supported by a K.U. Leuven junior scientist fellowship.

References

1. K. A. M. Ali and R. Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, Apr. 1990. 34
2. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, Jan. 1996. 23
3. M. Codish, B. Demoen, and K. Sagonas. Semantics-Based Program Analysis for Logic-Based Languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, Aug./Sept. 1998. To appear. 32
4. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. Technical Report CW 261, K.U. Leuven, Apr. 1998. Submitted for publication. 30, 35
5. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. *Journal of Functional and Logic Programming*, 1998(3), Apr. 1998. 21, 25, 26, 31, 32, 33
6. G. Janssens and K. Sagonas. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *Proceedings of TAPD-98: Tabulation in Parsing and Deduction*, pages 118–126, Paris, France, Apr. 1998. 33
7. P.-E. Moreau. A choice-point library for backtrack programming. In K. Sagonas, editor, *Proceedings of the JICSLP-98 Workshop on Implementation Technologies for Programming Languages based on Logic*, Manchester, U.K., June 1998. 34
8. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 687–711, Tokyo, Japan, June 1995. The MIT Press. Extended version to appear in the JLP. 21, 30, 31

9. R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parellelism to Tabled Evaluations. In *Proceedings of the First International Workshop on Tabling in Logic Programming*, pages 33–45, Leuven, Belgium, June 1997. 34
10. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM TOPLAS*, 20, 1998. To appear. 22, 23, 24, 25, 26, 31, 32
11. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM. 21, 32
12. D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog — Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, San Francisco, California, Sept. 1987. IEEE CS Press. 24, 34
13. D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 198–202, Atlantic City, New Jersey, Feb. 1984. IEEE Computer Science Press. 24

SICStus MT — A Multithreaded Execution Environment for SICStus Prolog

Jesper Eskilson and Mats Carlsson

Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
Phone: +46-8-7521500, Fax: +46-8-7517230
{jojo,matsc}@sics.se

Abstract. The development of intelligent software agents and other complex applications which continuously interact with their environments has been one of the reasons why explicit concurrency has become a necessity in a modern Prolog system today. Such applications need to perform several tasks which may be very different with respect to how they are implemented in Prolog. Performing these tasks simultaneously is very tedious without language support.
This paper describes the design, implementation and evaluation of a prototype multithreaded execution environment for SICStus Prolog. The threads are dynamically managed using a small and compact set of Prolog primitives implemented in a portable way, requiring almost no support from the underlying operating system.

Keywords: logic programming, implementation, multithreading, abstract machines

1 Introduction

Prolog has historically been centered around the concept of asking queries to a database and receiving answers and other output on a terminal or other output device. This has proven to be an excellent way of providing many software solutions.

But computer applications have become more and more complex, involving many independent and different subproblems. For example, a WWW-server normally contains a part which continuously listens for connections on sockets and a word-processor with on-the-fly spell-check has a part which continuously scans the spelling dictionary for the words which are typed in. This is usually done by employing multiple *threads of execution*, or just *threads*. Doing it without threads forces the programmer to manually switch between executing the different subproblems, making the program inefficient and difficult to write and maintain.

Now, threads is not a new concept in Prolog or in other languages. Most modern languages support threads in some way. Most notable are C [22,18],

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 36–53, 1998.

Erlang [1], Java [14], and Oz 2.0 [30,31]. Many logic programming systems implement threads (although they are mostly referred to as "processes"), such as CS-Prolog [15], KL1 [8], IC Prolog II [10], Multi-Prolog [4], but these are all relatively small and experimental systems. There are no widely-used commercial Prolog systems supporting threads (to the best of our knowledge). EMRM [32] includes a multi-client WWW-server based on Aurora [23], an or-parallel extension to SICStus Prolog. Amzi! Prolog + Logic Server supports multiple threads, but not in the same Prolog engine; they have to be coded using C or Java threads.

Summing up, multithreading is a very useful language extension for developing and maintaining large software systems, such as WWW-servers and intelligent software agents. This paper presents the design, implementation and evaluation of such an extension to SICStus Prolog [7]. Different options are discussed for the following design issues: native threads, scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. A preliminary performance analysis is given.

The rest of the paper is organized as follows: We state the requirements of our prototype in Section 2. We then describe the important design issues (Section 3, 4, 5) and their impact in the implementation. Section 6 describes the programming interface and Section 7 describes the solution to the problem of blocking system calls. Sections 9 and 10 evaluate the performance of our implementation and include references to related work. We end with some conclusions.

2 Requirements

The purpose of this work has primarily been to prove that support for multiple threads is realistic in an industrial-strength Prolog system, executing *full* Prolog. There are no restrictions, for example, on backtracking as in committed-choice languages. Also, threads performs separate and independent computations, not to be confused with *parallel* Prolog, where threads (or processes as they are sometimes referred to) cooperate to perform the same computation in parallel.

To prove that multiple threads are realistic under these requirements, we must show that introducing support for multiple threads does not cause any significant inconvenience for the user such as reduced execution speed or excessive memory consumption. In other words, the prototype must provide the benefits of multiple threads without removing any (or as little as possible) of the benefits of *not* supporting threads.

3 Native Threads

The first design issue which needs to be settled is whether to use native threads. Native threads have two major benefits. The first, and perhaps most important, is that they make it possible to utilize multiple CPUs (and other machine- and operating-system specific features). The second is related to blocking system calls and will be discussed in Section 7.

We have chosen not to use native threads at all in this first version of SICStus MT, for a couple of reasons. First, native threads are not available on all platforms. If this prototype relies on native threads we restrict our results to those platforms with native thread support. Second, native threads are not crucial in order to prove the feasibility of threads in a industrial-strength Prolog. Third, native threads can be implemented later on if desirable; avoiding native threads in the prototype does not rule them out completely. See also Section 11.

4 Necessary Modifications

The two basic characteristics of any language implementation are its storage and execution models. In order to support multithreading in SICStus, changes are needed to both of these.

4.1 Storage Model

The storage model need to be modified so that some data-areas are kept private to each thread. There are four data-areas in a WAM-based [2] emulator:

The Static Area contains a variety of objects, such as interpreted and compiled clauses, atoms, indexing tables, and so on.
The Local Stack contains procedure frames and choice point records.
The Global Stack contains Prolog terms. This is usually the largest area.
The Trail Stack contains conditional variable bindings, i.e. variables that should be reset to unbound upon backtracking.

In addition to this, we have the set of abstract machine registers organized as a data structure *WS* for *W*orker *S*tructure, which contains program counters, stack boundaries, choicepoint-information, etc.

The bulk of the static area is kept global in order for threads to be able to share code. The local and trail stacks must be kept private, since they are directly related to how the program is executed. The same goes for the abstract machine registers, the WS. The WS is combined with thread-related information (such as status-flags, thread ID, message-port, etc.) to form a data-structure representing a thread.

In a WAM-based emulator, all three stacks shrink on backtracking. Thus, if threads perform independent computations, the global stack too must be kept private. This unfortunately incurs an overhead on communication between threads (see Section 6.1), as messages need to be copied between stacks. It is conceivable to have a design that avoids copying, with a shared global stack that does not shrink except on garbage collection, but that would represent a major departure from the WAM and is out of scope of this work.

4.2 Execution Model

Like the storage model, the execution model needs to be modified in order to support multiple threads. Since native threads are not utilized, the execution model must support time-sharing of the emulator between the different threads (see also Section 7.2).

The main problem to solve is to determine where in the emulator loop threads should be switched in and out. The place where this is done is called *the synchronization point*. A natural candidate for the synchronization point is the *event-handler*, which handles certain kinds of asynchronous events, such as stack overflows, goals woken by variable bindings, and goals invoked by interrupts.

The benefit of using the event-handler as the location of the synchronization point is that the execution state is fully well-defined, so that a context switch between two threads is trivially implemented by exchanging references to the WS.

5 Scheduling

Scheduling [21] can be compared to motion picture soundtracks: if it is done well, it is not noticed—it just contributes to the overall impression of the performance.

The main requirements of the scheduler, apart from having a low scheduling overhead, is that it should be *preemptive*. Without preemption, it is impossible to write applications which use threads to perform independent work simultaneously (like serving two clients at the same time).

The algorithm used in SICStus MT is a variant of Round-Robin scheduling [29,33], called Priority Round-Robin (PRR). PRR scheduling is conducted like Round-Robin scheduling with the difference that Round-Robin is only practiced among threads with equal priority. Among threads with different priority, the priority determines which thread is scheduled for execution.

This algorithm is far from perfect. The main disadvantage is that it is not *fair* (a *fair* scheduling mechanism guarantees that threads will not starve, regardless of the number of threads waiting to run). If a thread decides to increase its priority above everybody else and then initiate a lenghty calculation, all other threads will starve for sure.

On the other hand, by not having priorities, threads with important tasks will have to wait for threads with very low priority tasks. For example, a spell-checker in a word-processor would naturally run with very low priority in order to avoid stealing resources from the more important task of handling user input and displaying it on the screen. Without priorities, the spell-checker would take up as much computing resources as the actual word-processor.

Summing up, PRR does not guarantee fairness, but it is robust, easy to implement, and provides good performance in most cases.

5.1 Choice of Time Quantum

The size of the time quantum is crucial to (P)RR scheduling. The time-quantum is defined as the maximum period of time a thread is allowed to execute before it

is interrupted by the scheduler. The size of this period has a large impact on the efficiency of the scheduling. If the period is too large, response times will become too long. If the time-quantum is allowed to be infinite, we lose the preemptive property of PRR. If the period is too small, the scheduling overhead will become too large.

We obtained a time quantum by using using a timer interrupt to generate an asynchronous signal, and then switch out the thread when it reaches the event-handler (as described in Section 4.2). If timer signals occurs every, say, 50 ms, the length of the time-quantum would then be 50 ms plus the time taken to reach the synchronization point. This will guarantee a finite time-quantum.

This solution has the drawback of relying on signals to ensure preemption. ANSI C does support timer signals [18], but only with low resolution (seconds). For non-ANSI C implementations which lack signals entirely, PRR cannot be implemented with guaranteed preemption. See Section 8 for a further discussion of the portability aspects of using timer signals.

It is also possible to base the time quantum on the number of executed WAM instructions. This has quite a large overhead (almost 20% in some cases), but it has a major benefit: it is possible to repeat the same execution sequence (for debugging purposes, for example). However, it would also require substantial modification to the native code kernel, which would be tedious to implement and would also degrade the performance of native code execution.

Another possibility is to base the time quantum on the number of executed procedures. This is more efficient than counting WAM-instructions and still supports repeatable execution sequences. This method is used by Oz and Erlang [30,31,1].

6 Programming Interface

The following predicates are introduced in SICStus MT. The usage of each predicate is indicated by prefixing each argument X by `+`, denoting that X should be instantiated to a non-variable in any call; `-`, denoting that X should be unbound; or `?`, denoting no restrictions on X.

`spawn(+Goal, -ThreadID)` Launches a new, independent computation in a separate thread. Thus, the declarative semantics of this predicate is "true". The new thread will execute the thread `Goal`. `ThreadID` will be bound to the identifier of the new thread. Also, a message-port is created for *send/receive* operations. See Section 6.1.

`send(+ThreadID, +Term)` Sends `Term` to the thread indicated by `ThreadID`. This predicate always succeeds (or throws a domain error exception). `Term` will be inserted last in the receiving thread's input queue. Unbound variables are consistently copied, i.e. `f(X,X)` becomes `f(Y,Y)` on the other side.

`receive(?Term)` Extracts the first element in the thread's input queue that is unifiable with `Term` and unifies it with `Term`. If no such term exists, the thread is suspended.

```
echo :-
    receive(Term),
    write(Term),
    nl,
    echo.

run :-
    spawn(echo, EchoThread),
    send(EchoThread, term1),
    send(EchoThread, term2),
    ...
    send(EchoThread, termn),
    ...
```

Fig. 1. Example of using send/receive. The example spawns a simple echo thread which lies in the background and echos everything sent to it

`self(-ThreadID)` `ThreadID` is the thread identifier of the running thread.
`kill(+ThreadID)` Causes `ThreadID` to terminate. Always succeeds, even when the thread does not exist or has died, i.e. the semantics is that after the call to `kill/1`, the specified thread is dead, regardless of its state before the call.
`wait(+Ms)` Suspends the currently running thread and then waits at least `Ms` milliseconds before resuming. The actual time elapsed before the thread is resumed is guaranteed to be *at least* `Ms` but could exceed this limit depending on two factors; the frequency of the timer-interrupts and the number of threads waiting to run. See Section 8.

See Figure 1 and 2 for examples of how to use these predicates. Figure 1 is a trivial example while Figure 2 is a little more complex example on how to code the predicate `join/2` using SICStus MT.

6.1 Communication and Synchronization

The model of communication and synchronization is *message-based* as opposed to *blackboard-based* [5,34], also known as *tuple space based* [16]. This means that the communication is based on sending explicit messages as opposed to using a shared store of some kind. A message can be any kind of Prolog term. Unbound variables are allowed in messages, but they will be renamed.

Furthermore, the message-port is *anonymous*, i.e. it is an integral part of the thread and addressed using the thread's identifier. Messages are addressed by direct naming, but only of the destination thread; the source thread is not specified, allowing a server, for example, to receive messages from unnamed clients.

```
spawn_joinable(Goal, JoinableThreadID) :-
    self(Self),
    spawn(joinable_wrapper(Goal, Self), JoinableThreadID).

joinable_wrapper(Goal,Parent) :-
    self(Self),
    ( on_exception(Exception,Goal,
                   joinable_handler(Exception,Parent,Self)) ->
        send(Parent,Self -> success(Goal))
    ;   send(Parent,Self -> fail)
    ).

joinable_handler(Exception,Parent,Self) :-
    send(Parent, Self -> exception(Exception)).

join(ThreadID,Result) :-
    receive(ThreadID -> Result).

run(Goal) :-
    spawn_joinable(Goal,ThreadID),
    ...
    join(ThreadID,Result),
    format('~w has completed. Result = ~w.~n', [Goal,Result]).
```

Fig. 2. Example of how to implement `join/2` using SICStus MT. The semantics of `join/2` is to suspend until the specified thread has completed. This implementation catches exceptions which are thrown in the thread and also sends back goal-term with eventual variable bindings.

Direct naming is not as flexible as indirect communication using named *channels* [9]. More specifically, channels allow many-to-many communications which direct naming does not. The main reason for using direct naming in the prototype was to keep the implementation simple, and this might well be reconsidered in a released version of SICStus MT.

Oz and Gypsy [30,17], are examples of language implementations using channels. PLITS and ERLANG [13,1] is an example that uses direct naming and allows the receiver to leave out the sender address.

The communication mechanism is *asynchronous*. This means that the sender does not need to wait until the receiver is ready to receive the message, i.e. the *send*-primitive is *non-blocking*. This means also that the mechanism is buffered, i.e. the communication media (the input queue) has a memory of its own where it can store messages until they are ready to be picked up by the receiving thread. The message port is basically a FIFO-structure, which means that it is

completely ordered. However, as we will discuss later on, the programmer can specify the order in which terms are received.

Due to our design choices, messages must be copied when sending them between threads. The absence of a shared heap causes at least one copying to be done. Full Prolog (with unrestricted backtracking) prevents us from copying directly to/from another thread's heap, so we must copy the message twice, via the static area.

If the receiving thread is suspended on a call to `receive/1` it is *lazily* switched in for execution, i.e. just moved to the ready-list. If the receiving thread is suspended for some other reason, nothing happens. The alternative would be *eager* thread switching, i.e. preempting the receiving thread, disregarding any priorities. See Section 9 for a discussion on the performance of these two approaches. When the receiving thread is eventually resumed, it must unpack the message on its own heap.

Message Non-determinism Recall that `receive/1` extracts the first *matching* message. The makes it possible to specify which messages to accept for a particular call to `receive/1`. This is also referred to as *message non-determinism* [5] and is also used in Erlang.

In the Game-of-Life benchmark, described in Section 9, there is a construction which relies on this particular feature. Since the cells work asynchronously (without a global "conductor" telling them when to do a state transition), each cell needs to make sure that the incoming messages are grouped by generation. This means that if a cell in generation x receives a message from a cell which is in generation $x + 1$, it must be able to defer that message until itself is in generation $x + 1$. In our implementation, this is solved by letting each cell loop through all its neighbors and for each one, wait for a message from that particular neighbor. In this way, we are guaranteed that the messages are processed in the correct generation. This would be very tedious to code without language support, since we would need to keep a separate list of terms which were received "too early", a list which needs to be maintained, sent around to all predicates calling `receive/1`, and searched for each such call.

However, there are performance issues worth discussing here. Each time the receiving thread tries to execute a `receive(Term)` goal (either the first time around, or upon a thread switch in case it was suspended), it has to traverse the input queue, unpack each message, delete it from the queue if it unifies with `Term`, and otherwise keep searching. Unpacked messages that do not unify are reclaimed by Prolog's backtracking and so must be unpacked again the next time around. Even though the time to unpack a message is linear in its size, a given message may get re-examined many times. Also, message non-determinism will result in a list of "currently unmatched" messages, i.e. messages that have arrived but are not unifiable with the argument to `receive/1`. This means that messages can be delayed in the input queue for a potentially indefinite period of time. See Section 9 for some performance figures.

7 Blocking System Calls

The problem of blocking system calls in user-level (as opposed to kernel-level) thread implementations is a well-known and well-investigated problem [33,12]. The core of the problem is that the operating system kernel (by definition) is unaware of the existence of user-level threads. Therefore, when a blocking system call is performed, the kernel suspends the entire process for the duration of the system call, instead of scheduling another thread for execution, which is the desirable behavior.

7.1 Possible Solutions

There are not that many ways of solving the problem. We must in some way prevent a given blocking system call from blocking the entire process and find a way of scheduling another thread instead. We have explored two approaches to the problem, the *cautious approach* and the *cavalier approach*.

The cautious approach uses a relatively complex mechanism in order to examine system resources in order to determine, without making the call, whether or not the system call would block the process. If the call could not be performed without blocking the process, the thread is suspended and another thread is switched in. Otherwise, the thread continues with the read and returns normally. The main problem of this approach is its complexity. Each system call which might block must be preceded by a piece of code (called *jacket*) in order to determine whether or not the system call would block or not. This turned out to be quite non-trivial—the documentation on when system calls block is often inadequate and examining system resources not a very portable procedure. Another drawback is the additional overhead of the jacket code.

The cavalier approach relies on the possibility of performing system calls without blocking the process at all, commonly known as *asynchronous I/O*. Instead of trying to determine beforehand whether or not a system call is about to block, we simply perform the system call asynchronously. A check is made after the call to determine if the call was completed and if not, the thread is suspended and then resumed when the asynchronous system call can be retried (as a result of a `SIGIO` signal indicating I/O completion).

The cavalier approach wins the game on the fact that it is simpler to implement and more robust. We leave it to the individual system call to determine whether or not it is about to block. This relieves us from having to write specialized code for each system call which is not only tedious but also error prone. The drawback is that is relies on the availability of asynchronous I/O (see Section 8).

We have not made any measurements of the performance penalty of these mechanisms (i.e. what the overhead is when threads are not used at all), but we do not believe that it is significant.

7.2 Emulator Support

The solutions discussed above both need a mechanism for communicating with the emulator. More precisely, they need to be able to inform the emulator when

a thread should be suspended as a result of a blocking system call. The idea is to introduce an extra return code for predicate-calls in addition to `TRUE/FALSE` (which represent success and failure, respectively). The new return code is called `SUSPEND` and is returned when the thread executing the predicate should be suspended.

Since the process of suspending a thread as a result of making a blocking system call varies significantly depending on the nature of the system call, the actual work of suspending a thread (setting bits, moving threads between lists, etc.) is done by the code performing the system call. The only action required by the emulator is to immediately jump to the synchronization point in order to perform a thread switch.

Native Code SICStus Prolog compiles Prolog code to native code on some platforms [3,19]. Thus instead of interpreting predicates or executing byte-compiled code, the Prolog code is compiled to native code and inserted directly into memory and executed as if it were a regular C function. The purpose of this is, of course, execution speed, and speedups of 3-4 times are typical.

The multithreaded execution model maintains full compatibility with native code execution, since the thread scheduling mechanism is built upon an already existing mechanism—the event handler—for which the native kernel already has support. The scheduler raises an asynchronous event which will cause the native code kernel to automatically escape back to the emulator, jump to the event-handler and thereby reschedule. When the thread later on is scheduled again, the native code execution will continue as normal. See also [27].

Suspending The Emulator The emulator will sometimes be in the situation where there are no more threads to schedule. For example, this happens immediately at startup in the development system when the top-level thread waits for input from the user. This should cause the Prolog process to suspend itself, just as if it would have if it had performed a normal, blocking system call.

This is implemented by a small piece of code in the scheduler. When the scheduler has suspended the top-level thread and realizes that the ready-list is empty, it suspends the process by calling `pause()`. Suspending and resuming processes is—as one would expect—platform-dependent. We will only describe the procedure used under Solaris; we expect the procedure to be fairly similar on most modern operating systems. See also Section 8.

The process is then resumed (i.e. `pause()` returns) when a signal is received. The signal is triggered by one of two reasons. Either the asynchronous I/O mechanism sends a signal to indicate that the I/O call was completed, or the timer mechanism sends a signal to indicate that we have one or more threads suspended on a call to `wait/1` and that we need to examine the queue to schedule those threads for which the time-quantum has expired. These actions are taken in the signal-handler routines, so that when `pause()` returns, we examine the ready-list and if everything went right we should have a thread waiting to execute.

However, for different reasons we might have received a false alarm, in which case we will simply be suspended again.

8 Portability Aspects

This implementation has been done on Solaris, and while most parts of the solution are directly portable to most operating systems supported by SICStus, there are some issues worth special attention.

Asynchronous I/O relies on the use of signal `SIGPOLL` to indicate I/O completion. This signal is not ANSI-C, but is included in POSIX and thus available on most UNIX dialects. The most notable exception is the Win32 platforms. However, on those platforms there are other (Win32-specific) ways of performing asynchronous I/O which mimic the use of `SIGIO`/`SIGPOLL`.

The same applies to the predicate `wait/1` (Section 6) and the mechanism for suspending the emulator (Section 7.2). They both use POSIX extensions which have counterparts in the Win32 interface.

Implementing SICStus MT in a bare ANSI-C environment is certainly possible, but it would mean some restrictions in functionality: Time-quantums cannot be specified with higher resolution than seconds, the `wait/1` predicate will have reduced accuracy, and blocking system calls will suspend the entire process.

9 Performance Evaluation

We briefly evaluate the space complexity and execution speed of our implementation. We have used two benchmark programs: *Game-of-Life* [20], a simulation of a simple biological environment, and a matrix multiplication program. The purpose of the former benchmark is to measure message-passing overheads, whereas the latter benchmark gives an idea of the intrinsic scheduling overheads.

9.1 Memory Consumption

Naturally, the objective during the implementation process has been to keep the threads as lightweight as possible. It is quickly realized that the space occupied by the WS is negligible compared to the stackconsumption.

Related to this is the fact that the address space in SICStus is only $2^{28} =$ 256 Mb (on a 32-bit architecture) since the 4 upper bits are used for tagging data-cells. This not only means that the data-areas cannot exceed 256 Mb, it also means that they have to be located at optimal places for this to be possible. In other words, if the address space becomes fragmented (by stack-shifting, for example), the limit of 256 Mb will drop even further. The impact of this limit will become clear in the next section.

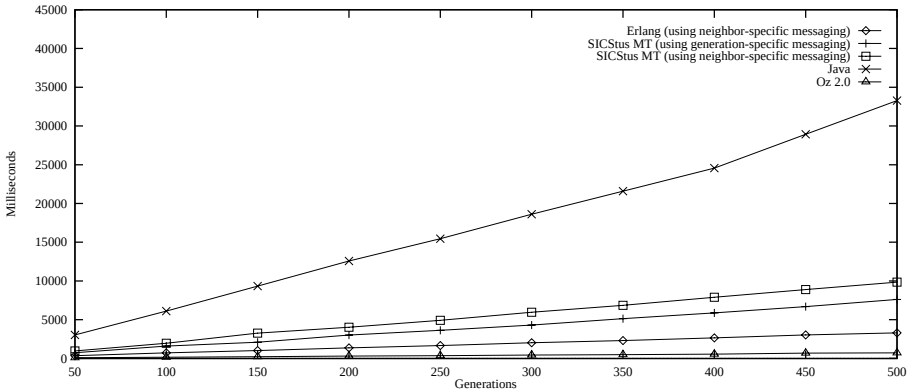


Fig. 3. Execution times for the Game-of-Life benchmark. The board is 10×10 cells.

9.2 Execution Speed

The figures in the following section have been obtained by running SICStus MT executing emulated code on a Sun UltraSPARC 248 MHz (unless otherwise noted).

Game-of-Life This is a variant of Conway's Game Of Life [20], a simulation of a simple biological environment. Basically, it consists of a matrix of cells, each of which may or may not contain an organism. There are rules for how the live cells reproduce for each generation and, depending on the initial population, many interesting and fascinating patterns occur. We have implemented the game in a way which makes it an excellent benchmark for measuring the efficiency of employing several (hundreds or even thousands) threads. By spawning a separate thread for each cell in the matrix and using the inter-thread communication mechanism to propagate information, we can get good information about the scheduling and communications overhead. See Figure 3.

The Oz 2.0 benchmark is not as relevant as the other three since it does not use the message passing mechanism to propagate information. Since Oz 2.0 has a shared heap (see Section 4.1), the threads can simply read the state of their neighbors directly from the heap. This eliminates two overheads: suspension overhead while sending and receiving messages and message copying overhead.

The Java-implementation (Sun's JDK 1.1.4) displays surprisingly bad performance, despite the fact that no message copying is done. We also found that there is also only a very small difference between JIT-compiled Java and byte-interpreted Java, which leads us to believe that it is the Java scheduler which is inefficient.

Comparing lazy switching and eager switching (see Section 6.1) using the Game-of-Life benchmark showed that lazy switching is more efficient. The difference was approximately 700 ms or 7%, measured on neighbor-specific messaging (the square markers in Figure 3). The difference is caused by the fact that lazy switching decreases the number of messages that are unpacked in vain (i.e. messages that do not match the first argument of the call to `receive/1`).

There is a considerable overhead in the message passing mechanism. The overhead can be divided into two parts. The first part is caused by having to copy messages between heaps. This is not a very big overhead, especially when the messages are small. The dominating overhead is caused by the message nondeterminism (see Section 6.1). This mechanism allows the receiving thread to specificy which messages should be received. The problem is that the sending thread has no way of finding out if a message is "appropriate", i.e. if it matches the first argument to `receive/1`. Instead, it must always resume the receiving thread so it can make the decision itself. This is where *generation-specific* and *neighbor-specific* messaging come in. The former means that messages are only identified by their generation and the latter means that messages are identified by the sending cell (or thread). Neighbor-specific messaging causes more messages to arrive "out-of-order". This induces a higher overhead in the parsing of the message queue which can clearly be seen in the figure.

This overhead can be reduced by being more selective about when to resume threads blocked on `receive/1`, so we minimize the number of threads which are resumed in vain. Lazy switching is one way of doing this. Another way is to allow the *sending* thread to inspect the first argument to `receive/1` (in the receiving thread, that is) and avoid resuming threads when the message does not match. This should enable us to eliminate all unnecessary scheduling of threads blocked on `receive/1`.

Another improvement is to utilize indexing [26] to search the input queue. The queue is currently searched linearly, but by using indexing, the search would become considerably more efficient. The actual improvement, however, depends heavily on the application. If the average length of the input queues is large (which is the case if there are many messages arriving out of order), this improvement will have larger impact than if the queues are mostly empty.

Yet another, more orthogonal way of reducing this overhead is to introduce named channels (see Section 6.1). Named channels can be used to reduce the amount of out-of-order messaging, since messages of a certain kind can be sent to a dedicated channel.

Matrix Arithmetic Since our implementation of the Game-of-Life benchmark relies on the existence of threads, it is difficult to use that benchmark to get a grip on the overhead of SICStus MT compared with single-threaded SICStus.

To get some figures on this, we used a very simple matrix multiplication benchmark which can operate in two different modes, *threaded* and *meta-called*. The first spawns a thread for each cell in order to compute its value and the second simply performs a meta-call (using `call/1`), thereby not spawning any

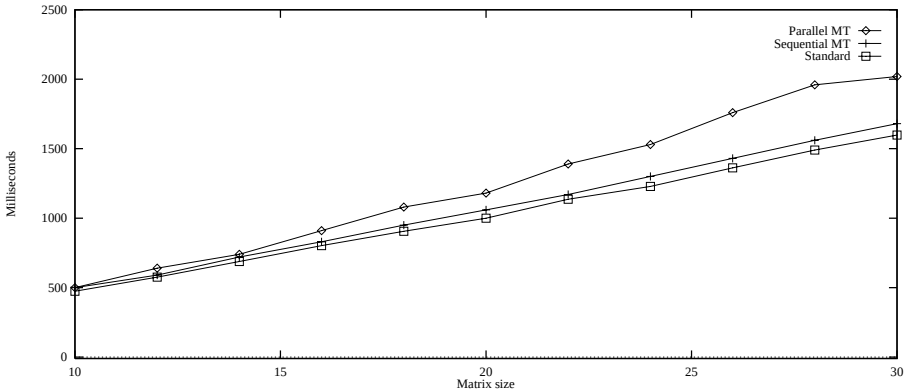


Fig. 4. Execution times for multiplying a matrix using different version of SICStus.

threads. These two modes correspond to the *Parallel MT* and *Sequential MT* lines in the graph in Figure 4, which were obtained using SICStus MT. The third line is obtained by running the matrix benchmark in sequential mode but using plain SICStus (version 3#5).

While the difference between lazy and eager switching was quite small in the Game-of-Life benchmark, it made a considerable difference here. The main thread receives a messages from each thread when it has completed, and eager switching caused the main thread to start executing for each such message while lazy switching only let the main thread execute once all multiplier threads had completed. The difference was large enough (roughly a factor 2 compared to Sequential MT) to exclude eager switching from an interesting comparison.

The data in Figure 4 tell us that the overhead of supporting threads is very small (the benchmark had to be executed several times in order to obtain a reliable difference). This means that it is reasonable to include support for multiple threads in a released version of SICStus.

The benchmark also shows that the overhead of spawning a thread as opposed to doing a meta-call is reasonably small. Also, the overhead of the meta-call itself (not shown in the graph) was insignificant; we did not observe a significant difference between the meta-called version and the standard sequential version.

Performance Conclusion Added up, it is reasonable to include support for multiple threads in a released version of SICStus, but the message passing overheads needs to be reduced in order to be competitive with implementations such as Erlang and Oz 2.0.

10 Related Work

A great deal of work on parallel execution of Prolog has been done during the past 15 years; see e.g. [11,23]. However, this work has mostly been concerned with exploiting the parallelism which is implicit in logic programs.

Several concurrent logic programming system based on explicitly creating processes (or threads) have evolved during the last 15 years. A survey of the general issues can be found in [5].

CS-Prolog Professional [24] supports multiple processes and uses the message-passing paradigm for communication using explicitly created channels as media. Asynchronous message passing is not supported; instead a rendezvous (or hand-shake) model is used. However, forcing threads to synchronize whenever communication takes place tends to encourage deadlocks and is generally restrictive to the programmer.

KL1 [8] is a concurrent logic programming language based on *GHC* (Guarded Horn Clauses). The concurrency is similar to that of SICStus MT; threads are managed explicitly and communication is done using blocking variable bindings.

Multi-prolog [4] and *BlackLog* [28] use the blackboard paradigm for communicating between processes. Blackboard-based systems are inherently less scalable in applications where there is heavy communication between processes since all messages need to pass through one single point. On the other hand, blackboards tend to give the programmer more expressiveness since messages do not need to be sent to an explicit destination.

11 Future Work

11.1 Implementing Native Threads

The perhaps most interesting area for future work is the incorporation of native threads. As mentioned in section 3, native threads are not used at all in this prototype implementation. However, the benefits of using native threads (multiple CPU utilization, blocking systems calls, etc.) are important enough to justify native threads in a released version of SICStus. Incorporating native threads raises a couple of design issues itself.

First, should native threads replace the emulated threads of the current design? The answer to this question is “no”. Not all platforms have multiple CPUs and there is a certain amount overhead with using native threads; both in the handling of the native threads themselves but also with the additional cost of needing to synchronize accesses to global data in the emulator. So, in some cases it might well be justified (from a performance point of view) to avoid native threads to some extent.

Second, how should the underlying native threads implementation be chosen to minimize portability problems? Are Pthreads (POSIX threads) [25] suitable for our purposes? This question is probably the most difficult one to give a good answer to. There are many different native threads implementations with different properties regarding portability, user-space vs. kernel-space, scheduling, etc.

Answering the first question with "no", raises another question: How do we map Prolog threads onto native threads? Should it be automatic or user-controlled? This question is still open for discussion. Since native threads do use resources in the operating system, using too many will degrade overall system performance. On the other hand, a Prolog application (such as Game-of-life) might want to create a very large number of threads. In such a situation, it is necessary to strike a balance between the number of native threads used and how the Prolog threads should map onto these. This dynamic scenario makes it unlikely that a static mapping—i.e. a Prolog thread is attached to a specific native thread, for example when calling `spawn/2`, and is thereafter fixed to that native thread—is suitable. A dynamic mapping where Prolog threads could be dynamically scheduled on the existing native threads could more easily adopt to different user-needs and different platforms.

11.2 Debugger

Debugging a multithreaded application is not an easy task. However, we are not really talking about debugging Prolog code (for which SICStus has very good support), but rather about debugging the concurrency introduced by multithreading. Extending the existing debugger with the basic support for debugging multithreaded code is not that difficult; the problem is the support for more advanced features such as deadlock detection, debugging race conditions, etc.

12 Conclusion

We have presented the design and implementation of SICStus MT, a multithreaded extension to SICStus Prolog. The following design issues were discussed in detail: whether to use native threads, the scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. We gave a preliminary performance analysis, indicating minimal scheduling overheads, but significant message passing overheads in the implemented prototype. Work is under way to reduce these overheads.

References

1. Joe Armstrong, Robert Virding, Claes Wiström, and Mike Williams. *Concurrent Programming In Erlang*. Prentice Hall, second edition, 1996. 37, 40, 42
2. Hassan Aït-Kaci. *Warren's Abstract Machine—A Tutorial Reconstruction*. MIT Press, 1991. 38
3. Kent Boortz. SICStus maskinkodskompilering. SICS Technical Report T91:13, Swedish Institute of Computer Science, August 1991. 45
4. K. De Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In D. S. Warren, editor, *Proceedings of the ICLP'93 conference*, pages 299–313, Budapest, Hungary, June 1993. The MIT Press. 37, 50

5. Koen De Bosschere. Process-based parallel logic programming: A survey of the basic issues. In Bosschere et al. [6]. 41, 43, 50
6. Koen De Bosschere, Jean-Marie Jacquet, and Antonio Brogi, editors. *ICLP94 Post-Conference Workshop on Process-Based Parallel Logic Programming*, June 1994. 52, 53
7. Mats Carlsson, Johan Widén, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. SICStus Prolog User's Manual. SICS Technical Report T91:15, Swedish Institute of Computer Science, June 1995. Release 3 #0. 37
8. Takashi Chikayama, Tetsuro Fujise, and Hiroshi Yashiro. A portable and reasonably efficient implementation of KL1. In Warren [35], page 833. 37, 50
9. Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms*. Addison-Wesley, March 1997. 42
10. Damian Chu. I.C. Prolog II: a Multi-threaded Prolog System. In Evan Tick and Giancarlo Succi, editors, *ICLP-Workshops on Implementations of Logic Programming Systems*, pages 17–34. Kluwer Academic Publishers, 1993. 37
11. J. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983. 50
12. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems—Concepts And Design*. Addison-Wesley, second edition, 1994. 44
13. J. A. Feldman. High Level Programming for Distributed Computing. *Communications of the ACM*, 22(6):353–368, 1979. 42
14. David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, second edition, 1997. 37
15. Iván Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In Warren [35], pages 3–17. 37
16. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1989. 41
17. D. I. Good, R. M. Cohen, and J. Keeton-Williams. Principles of Proving Concurrent Programs in Gypsy. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 42–52, 1979. 42
18. Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual*. Prentice Hall, third edition, 1991. 36, 40
19. R. C. Haygood. Native code compilation in SICStus Prolog. In *Proceedings of the Eleventh International Conference of Logic Programming*. The MIT Press, 1994. 45
20. John Horton. Computer Recreations. *Scientific American*, March 1984. 46, 47
21. B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5):347–360, May 1968. 39
22. Bill Lewis and Daniel J. Berg. *Threads Primer—A Guide To Multithreaded Programming*. Prentice Hall, 1996. 36
23. Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990. 37, 50
24. ML Consulting and Computing Ltd, Applied Logic Laboratory, Budapest, Hungary. *CS-Prolog Professional User's Manual, Version 1.1*, 1997. 50
25. Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996. 50

26. R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-Driven Indexing of Prolog Clauses. In *Proceedings of the Principles of Programming Languages*, 1990. 48
27. John H. Reppy. Asynchronous Signals in Standard ML. Technical Report 90-1144, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1990. 45
28. D. G. Schwartz. *Cooperating Heterogenous Systems: A Blackboard-based Meta Approach*. PhD thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1993. 50
29. Abraham Silberschatz and Peter B. Galvin. *Operating Systems Concepts*. Addison-Wesley, fourth edition, 1994. 39
30. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995. 37, 40, 42
31. Gert Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section. 37, 40
32. Péter Szeredi, Katalin Molnár, and Rob Scott. Serving multiple HTML clients from a Prolog application. In Paul Tarau, Andrew Davison, Koen de Bosschere, and Manuel Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, in conjunction with JICSLP'96, Bonn, Germany*, pages 81–90. COMPULOG-NET, September 1996. Available from: `http://clement.info.umoncton.ca/~lpnet/lp-internet/archive.html`. 37
33. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992. 39, 44
34. Hamish Taylor. Design of a resolution multiprocessor for the parallel virtual machine. In Bosschere et al. [6]. 41
35. David S. Warren, editor. *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993. The MIT Press. 52

A Framework for Bottom Up Specialisation of Logic Programs

Wim Vanhoof*, Danny De Schreye**, and Bern Martens***

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
Tel: ++32 16 32 7638, Fax: ++32 16 32 7996
{wimvh,dannyd,bern}@cs.kuleuven.ac.be

Abstract. In this paper, we develop a solid theoretical foundation for a bottom up program transformation, capable of specialising a logic program with respect to a set of unit clauses. Extending a well-known operator, originally introduced for composing logic programs, we define a bottom up partial deduction operator and prove correctness of the transformation with respect to the S-semantics. We also show how, within this framework, a concrete control strategy can be designed.
The transformation can be used as a stand-alone specialisation technique, useful when a program needs to be specialised w.r.t. its internal structure (e.g. a library of predicates w.r.t. an abstract data type) instead of a goal. On the other hand, the bottom up transformation can be combined with a more traditional top down partial deduction strategy. We conjecture that such a combined approach will finally enable good automatic specialisation of meta-programs.

1 Introduction

Partial deduction is an important transformation technique for logic programs, capable of removing substantial inefficiencies from programs [16,9,5]. As an online specialisation technique, it is based on an evaluation mechanism for logic programs. The input to a typical partial deducer is a program and a partially instantiated query. The instantiated part represents the information with respect to which one would like to specialise; the uninstantiated part represents the information not yet known. Therefore, all classical partial deduction techniques use *top down* evaluation (or SLD-resolution) to evaluate the program parts that depend on the known input and generate a new program that computes its result using only the remainder of the input. Since the new program has less computations to perform, in general, it will be more efficient.

In recent work [21], we argued the need for a complementary partial deduction technique, capable of "specialising" a program w.r.t. a set of (unit) clauses

* Supported by a specialisation grant of the Flemish Institute for the Promotion of Scientific-Technological Research in Industry (IWT), Belgium.
** Senior Research Associate of the Belgian National Fund for Scientific Research.
*** Partially supported by Esprit project 25503, ARGo.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 54–72, 1998.

instead of a goal. It seems natural to define such a specialisation scheme in terms of bottom up evaluation. During evaluation, the information from the unit clauses is then propagated upwards, and new facts and clauses are derived. In [21], we developed a specific, and very concrete control scheme for such a bottom up transformation and provided some examples showing that a combination of bottom up transformation and classical (goal-directed) top down partial deduction achieves at least equally good results as a top down scheme alone, while requiring a much less complicated control mechanism for either part.

In this work, we establish a solid theoretical foundation for such bottom up transformation schemes, rendering a general framework in which concrete control strategies can be proven correct.

In Section 2, we first recapitulate and illustrate briefly the main motivation for bottom up specialisation. Subsequently, Section 3 – the main one – develops the framework. Next, we address concrete control in Section 4. Section 5, finally, discusses achievements and plans for future work.

Throughout the paper, we restrict attention to definite programs (and goals).

2 Motivating Bottom Up Specialisation

In this section, we recapitulate briefly our motivation for bottom up specialisation. More information can be found in [19,21].

In a logic program, some information flows naturally in a bottom up fashion, starting from a set of unit clauses. Take as example a program in which the concrete representation of some data structure is hidden by an abstract data type (ADT) and all operations manipulating the structure are defined using functionality provided by the ADT implementation. Abstracting such information through several layers makes sense from a software engineering point of view: concrete representations can be altered without much effort. On the other hand, every layer of abstraction decreases efficiency. Therefore, it makes sense to propagate the concrete information up into the program, to the places where it is really used. In the ADT-example: propagating the concrete representation into the program can eliminate all calls to the ADT, removing a layer of overhead.

This sort of information propagation can in principle be obtained by a top down specialisation scheme. Achieving it in a *general* and *completely automatic* way, however, is far from trivial since control information, needed by the specialiser to decide whether or not to continue the specialisation, might also flow bottom up. In particular, (Vanilla-like) *meta-programs typically present difficulties of this kind* [20]. Consider the following example, taken from [21]: The predicate $make\_list(T, I, R)$ can be used to create a list of a fixed length (type T), with each element initialised with I. The result is returned in R.

Example 1.

$fill\_list(L, T, I, L) \leftarrow type(T, L).$
$fill\_list(L, T, I, R) \leftarrow fill\_list([I|L], T, I, R).$
$make\_list(T, I, R) \leftarrow fill\_list([], T, I, R).$

$type(list1, [X])$
$type(list3, [X_1, X_2, X_3]).$

Example 1 represents a class of recursive predicates that build up some structure between calls before a base clause of the predicate (ending the recursion) is reached, depending on the structure built. All top down specialisers are based on unfolding techniques, which are known to have problems with these: Often, it cannot be derived from the SLD-tree built so far whether or not unfolding such a recursive call will eventually terminate.

If this recursive predicate is handled in a bottom up fashion, structure is shrinking between recursive calls, resulting in the facts

$make\_list(list1, I, [I])$.
$make\_list(list3, I, [I, I, I])$.

Apart from control, there are other reasons why a bottom up approach might be preferred. Sometimes goal-directed specialisation simply is not needed, because all information to be propagated into the program is already there, and need not be provided from elsewhere. Consider a program library M, in which n routines are defined, all using an ADT. A top down specialiser needs a single goal to start specialisation from, which might not be available when specialising the library. Or it/they will likely contain no information (all arguments free) since the latter flows bottom up (further complicating top down control).

So, in a number of cases, proceeding bottom up is a more natural solution. bottom up transformation and specialisation has been considered occasionally before (see e.g. [11,4]). However, to the best of our knowledge, our ongoing effort is the first attempt to achieve these in a completely general and automatic way.

3 Defining the Framework

First, we recall the concept of a non-ground T_P operator, which normally acts on atoms. However, for reasons that will become clear below, we consider a non-ground T_P operator acting on *sets of clauses*[1] as in compositional semantics (see e.g. [2,1]). To keep the presentation simple, atoms will be considered as unit clauses and vice versa, as the context requires. In what follows, $\mathcal{HC}$ denotes the set of all Horn clauses over a fixed first order language $\mathcal{L}$, underlying a given definite program P. Let $\equiv$ be the variant relation on $\mathcal{HC}$. For any set S, 2^S denotes its powerset. Unless specified otherwise, uppercase characters A,B,... denote atoms, $\tilde{A}$,$\tilde{B}$ conjunctions of atoms, r, s, t, u terms and $\tilde{r}, \tilde{s}, \tilde{t}, \tilde{u}$ appropriate numbers of terms used as predicate arguments. As usual, $A \preceq B$ denotes that A is more general than B. We mean by $\theta = mgu((B_1, \ldots, B_m), (C_1, \ldots, C_m))$ that $(B_1, \ldots, B_m)$ and $(C_1, \ldots, C_m)$ are unifiable and θ is an mgu of the m-tuples, so, $B_i\theta = C_i\theta$ for $i = 1 \ldots m$. For a program P, $Pred(P)$ denotes the set of predicate names occurring in P. For an atom A, $Var(A)$ denotes the set of variables of A.

First, we define a basic operator, adapted from [2].

[1] Hence the notation: T_P^C.

Definition 1. *The* compositional, non-ground T_P-operator, *T_P^C, is a function $T_P^C : 2^{\mathcal{HC}/\equiv} \rightarrow 2^{\mathcal{HC}/\equiv}$, defined for any $S \subseteq \mathcal{HC}/\equiv$ as: $T_P^C(S) = S \cup \{H\theta \leftarrow (\tilde{L_1}, \ldots, \tilde{L_n})\theta$ where $H \leftarrow B_1, \ldots, B_n \in P (n \geq 0)$, $A_1 \leftarrow \tilde{L_1}, \ldots, A_n \leftarrow \tilde{L_n}$ fresh variants of clauses in S and $\theta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n))\}$.*

From [1], we know that T_P^C is continuous and its least fixpoint $T_P^C \uparrow \omega$ exists. By definition, $T_P^C \uparrow \omega$ is a set of unit clauses denoting the *least S-Herbrand model* [8,1] of P, $M_s(P)$. That is, $M_s(P) = Heads(T_P^C \uparrow \omega)$ where *Heads* is a function that projects a set of (unit) clauses onto their heads. In the remainder of this paper, we simply consider $M_s(P) = T_P^C \uparrow \omega$.

Example 2. Consider the following program:

$P : p(X, c) \leftarrow \qquad r(a).$
$\quad p(a, a) \leftarrow \qquad r(f(X)) \leftarrow r(X).$
$\quad q(X, Y) \leftarrow p(X, Y)$
Then $M_s(P) = \{p(X, c), p(a, a), r(a), q(X, c), q(a, a), r(f(a)), r(f(f(a))), \ldots\}$

Our aim is to define a partial deduction operator that transforms a program P into a program P' in which information is propagated in a bottom up fashion. T_P^C can serve as a basis for such an operator, but we need to ensure that consecutive applications of our operator always terminate within finite time. To achieve this, unlike [2,1], we define and incorporate an abstraction operator on $\mathcal{HC}$.

First, we introduce the concept of a predicate renaming.

Definition 2. *A* predicate renaming *of an atom $p(t_1, \ldots, t_n)$ is $p'(t_1, \ldots, t_n)$ where either $p' = p$, or p' denotes a new, unique predicate name.*

Thus, a predicate renaming of an atom $p(\tilde{t})$ is the atom itself or $p'(\tilde{t})$, where p' is a newly introduced predicate symbol (not used before in the same program transformation).

Definition 3. *An* abstraction function, *Abst, is an idempotent function $2^{\mathcal{HC}/\equiv} \rightarrow 2^{\mathcal{HC}/\equiv}$, such that for any $S \subseteq 2^{\mathcal{HC}/\equiv}$, if $S' = Abst(S)$, then for every clause $p(\tilde{t}) \leftarrow \tilde{B} \in S$, either $p(\tilde{t}) \leftarrow \tilde{B} \in S'$ or there exists a clause $p(\tilde{s}) \leftarrow p'(\tilde{s}) \in S'$ such that $p(\tilde{s}) \preceq p(\tilde{t})$ and $p'(\tilde{s})$ is a predicate renaming of $p(\tilde{s})$.*

As we show below, the abstraction $p(\tilde{s}) \leftarrow p'(\tilde{s})$ can generate, in combination with an appropriate renaming of the original clause, $p'(\tilde{t}) \leftarrow \tilde{B}$, the same atoms of $M_s(P)$ as the original clause $p(\tilde{t}) \leftarrow \tilde{B}$ (since $\theta = mgu(p'(\tilde{t}), p'(\tilde{s}))$ and $p(\tilde{s})\theta = p(\tilde{t})$). Since possibly more clauses can be mapped on the same generalisation, abstraction enables us to replace the computation of $M_s(P)$ (which is possibly infinite) with the computation of a *finite* set of *clauses*, defining the *same* least S-Herbrand model.

Therefore, we combine T_P^C with an abstraction function to obtain a bottom up partial deduction operator.

Definition 4. *The* abstracting T_P^C-operator, *A_P^C, associated to a program P and an abstraction function Abst is defined as $A_P^C = Abst \circ T_P^C$.*

Example 3. Reconsider the program of Example 2. We define $Abst(\{p(a,a) \leftarrow\}) = \{p(a,Y) \leftarrow p(a,Y)\}$ and $Abst(\{r(a)\} \cup \{r(f(X)) \leftarrow r(X)\}) = \{r(X) \leftarrow r(X)\}$ while all other clauses remain unchanged by *Abst*. Then

$$I_0 = \{\}$$

$$I_1 = A_P^C(I_0) = \left\{ \begin{array}{l} p(a,Y) \leftarrow p(a,Y) \\ p(X,c) \leftarrow \\ r(X) \leftarrow r(X) \end{array} \right\} \quad I_2 = A_P^C(I_1) = \left\{ \begin{array}{l} p(a,Y) \leftarrow p(a,Y) \\ p(X,c) \leftarrow \\ r(X) \leftarrow r(X) \\ q(a,Y) \leftarrow p(a,Y) \\ q(X,c) \leftarrow \end{array} \right\}$$

The reader can easily verify that I_2 is the least fixpoint of A_P^C.

In general, A_P^C is not a monotonic operator, since *Abst* may replace clauses by their generalisations. However, the following proposition holds:

Proposition 1. *For any* $S \subseteq \mathcal{HC}/_{\equiv}$*, such that* $Abst(S) = S$*, we have* $S \subseteq A_P^C(S)$.

Proposition 1 follows straightforwardly from the definition of A_P^C and monotonicity of T_P^C. Thus, if we start off from an initial set S_0 which satisfies $Abst(S_0) = S_0$, monotonicity of a sequence of A_P^C-applications is ensured since *Abst* is required to be idempotent. Therefore, we can define ordinal powers of A_P^C as follows:

Definition 5. $A_P^C \uparrow 0 = \{\}$, $A_P^C \uparrow n = A_P^C(A_P^C \uparrow (n-1))$ *for* $n \in \mathbb{N}$ *and* $A_P^C \uparrow \omega = \bigcup_{n \in \mathbb{N}} A_P^C \uparrow n$.

Henceforth, we will always use A_P^C as in Definition 5. As long as no abstraction occurs, subsequent A_P^C applications, starting from an initial set $\{\}$ derive sets of unit clauses. Abstraction introduces clauses in a set $A_P^C \uparrow i$ that are no longer unit clauses, but clauses capable of generating (among others) the same atoms as the abstracted ones, *provided the originals are present*. To that extent, we define two complementary functions on sets generated by A_P^C: $\boldsymbol{Res}$, containing the clauses $p'(\tilde{t}) \leftarrow \tilde{B}$ where $p(\tilde{t}) \leftarrow \tilde{B}$ was removed because of an abstraction $p(\tilde{s}) \leftarrow p'(\tilde{s})$ and $\boldsymbol{Gen}$, comprising the abstractions themselves.

Definition 6. $\boldsymbol{Res}, \boldsymbol{Gen} : 2^{\mathcal{HC}/_{\equiv}} \rightarrow 2^{\mathcal{HC}/_{\equiv}}$ *are defined as:*

- $\boldsymbol{Res}(A_P^C \uparrow 0) = \boldsymbol{Gen}(A_P^C \uparrow 0) = \{\}$.
- $\boldsymbol{Res}(A_P^C \uparrow n) = \boldsymbol{Res}(A_P^C \uparrow (n-1)) \cup \{p'(\tilde{t}) \leftarrow \tilde{B} | p(\tilde{t}) \leftarrow \tilde{B} \in T_P^C(A_P^C \uparrow (n-1))$ *and* $p(\tilde{t}) \leftarrow \tilde{B} \notin A_P^C \uparrow n$ *and* $Abst(\{p(\tilde{t}) \leftarrow \tilde{B}\}) = \{p(\tilde{s}) \leftarrow p'(\tilde{s})\}\}$ *for any* $n > 0$.
- $\boldsymbol{Gen}(A_P^C \uparrow n) = \boldsymbol{Gen}(A_P^C \uparrow (n-1)) \cup Abst(T_P^C(A_P^C \uparrow (n-1))) \setminus T_P^C(A_P^C \uparrow (n-1))$ *for any* $n > 0$.

Obviously, all clauses in any $\boldsymbol{Gen}$ set are of the form $p(\tilde{s}) \leftarrow p'(\tilde{s})$. Note also that $\boldsymbol{Gen}(A_P^C \uparrow n) \subseteq A_P^C \uparrow n$ while $\boldsymbol{Res}(A_P^C \uparrow n) \cap A_P^C \uparrow n = \{\}$.

Example 4. In Example 3, $M_s(I_2) = \{p(X,c), q(X,c), p(a,c), q(a,c)\} \neq M_s(P)$. We will return to the presence of $p(a,c)$ and $q(a,c)$ in Example 5 below. Here we first note that a.o. $p(a,a)$ and $q(a,a)$ have disappeared. This is due to the abstraction of $p(a,a) \leftarrow$ into $p(a,Y) \leftarrow p(a,Y)$. There is however a clause $q(a,Y) \leftarrow p(a,Y)$ in $A_P^C \uparrow 2$ such that $\theta = mgu(p(a,a), p(a,Y))$ exists and $q(a,Y)\theta = q(a,a)$. Hence, while $p(a,a)$ is abstracted away and also $q(a,a)$ therefore not directly derived by A_P^C, the situation is rectified by storing $p(a,a)$ through $\boldsymbol{\mathcal{Res}}$ and adding it to the final program, as in Definition 7 below.

In what follows, we will assume *Abst* to be defined such that A_P^C is finitary[2]; in other words, $A_P^C \uparrow \omega = A_P^C \uparrow n_0$, for some $n_0 \in \mathbb{N}$. Note that if A_P^C is indeed finitary and reaches its least fixpoint in $n_0 \in \mathbb{N}$, $\boldsymbol{\mathcal{Res}}(A_P^C \uparrow n_0)$ and $\boldsymbol{\mathcal{Gen}}(A_P^C \uparrow n_0)$ are necessarily finite sets, since all sets $A_P^C \uparrow n$, $n \leq n_0$ are finite.

We are now in a position to define the notion of a bottom up partial deduction.

Definition 7. *Given a program P and an abstraction function Abst, giving rise to a finitary abstracting T_P^C-operator, A_P^C, with least fixpoint $A_P^C \uparrow n_0$, the* bottom up partial deduction *of P (using Abst) is the program*
$A_P^C \uparrow n_0 \cup \boldsymbol{\mathcal{Res}}(A_P^C \uparrow n_0)$.

Before reconsidering our example, we return to the use of predicate-renamings in the abstraction function. Notice that by Definition 2, a valid predicate renaming of an atom A is the atom itself. As a consequence, clauses in $\boldsymbol{\mathcal{Gen}}$ (and hence in the residual program, P') may be of the form $p(\tilde{s}) \leftarrow p(\tilde{s})$. Such clauses are "useless" and possibly introduce non-termination in top down evaluation. However, since our transformation is based on the T_P^C-operator, it seems natural to formulate soundness and completeness of bottom up partial deduction in terms of least S-Herbrand models, and the presence of such clauses in P' does not influence $M_s(P')$. Moreover, *after completion of the transformation*, they can be easily excluded from the residual program. However, as Example 5 below shows, renamings are sometimes necessary to guarantee soundness of the transformation. First, we recapitulate the definition of soundness and completeness with respect to the least S-Herbrand model. Obviously, also in the transformed program, we are really only interested in the original predicates. Hence, in the remainder of this paper and in the following definition of transformation correctness, $M_s(P')_{|Pred(P)} = \{p(\tilde{t}) \in M_s(P') \mid p \in Pred(P)\}$.

Definition 8. *Let P' be a bottom up partial deduction of a program P. P' is* sound *w.r.t. P if $M_s(P')_{|Pred(P)} \subseteq M_s(P)$. P' is* complete *w.r.t. P if $M_s(P) \subseteq M_s(P')$.*

Without the use of predicate renamings, a bottom up partial deduction is not sound in general.

[2] Clearly, this is a matter of control: It involves deciding when to abstract and how to do it. See Section 4.

Example 5. Reconsider the program P from Example 2 and *Abst* from Example 3. Since $A_P^C(I_2) = I_2$, the fixpoint is reached in $A_P^C \uparrow 2$ and the bottom up partial deduction of P is $P' = I_2 \cup \boldsymbol{Res}(I_2)$ and $\boldsymbol{Res}(I_2) = \{p(a,a) \leftarrow\} \cup \{r(a) \leftarrow\} \cup \{r(f(X)) \leftarrow r(X)\}$. It is easy to see that, while P and P' have equal least Herbrand models, $M_s(P') \not\subseteq M_s(P)$:

$$M_s(P) = \{p(X,c), p(a,a), r(a), q(X,c), q(a,a), r(f(a)), r(f(f(a))), \ldots\}$$

$$M_s(P') = \{p(X,c), p(a,a), r(a), q(X,c), q(a,a), \boldsymbol{p(a,c)}, \boldsymbol{q(a,c)}, r(f(a)), \ldots\}.$$

In this example, due to the abstraction of $p(a,a) \leftarrow$ into $p(a,Y) \leftarrow p(a,Y)$, we introduce a clause in P' that is capable of further instantiating elements of $M_s(P')$: $p(X,c) \in M_s(P')$ and $p(X,c)$ unifies with $p(a,Y)$ without being an instance of it, and hence $p(a,c) \in M_s(P')$ while it is not an element of $M_s(P)$. Soundness can be obtained by imposing an extra condition on a bottom up partial deduction, ensuring that if any element $p'(\tilde{t})$ of $M_s(P')$ unifies with the body atom of a clause $p(\tilde{s}) \leftarrow p'(\tilde{s})$ introduced by *Abst*, $p'(\tilde{t})$ must be an instance of $p'(\tilde{s})$.

Definition 9. *Let S be a set of clauses, such that $\boldsymbol{Gen}(S)$ and $\boldsymbol{Res}(S)$ are defined. S is* inside-closed *if for all $A \leftarrow \tilde{B} \in S \cup \boldsymbol{Res}(S)$ holds: if $H \leftarrow H' \in \boldsymbol{Gen}(S)$ such that $\theta = mgu(A, H')$ exists, then $H'\theta = A$.*

As shown in Example 5, when this condition is not imposed on $A_P^C \uparrow n_0$, *Abst* might create clauses by which atoms in $M_s(P')$ are derived that are more instantiated than any atom in $M_s(P)$. Note that a concrete control strategy (as in Section 4, e.g.) need not actually rename as long as the set $A_P^C \uparrow i$ stays inside-closed. Hence the possibility to re-use the same predicate in Definition 2.

Example 6. Reconsider the program P from Example 2 but *Abst* now defined as $Abst(\{p(a,a) \leftarrow\}) = \{p(a,Y) \leftarrow p'(a,Y)\}$ and $Abst(\{r(a) \leftarrow\} \cup \{r(f(X)) \leftarrow r(X)\}) = \{r(X) \leftarrow r(X)\}$ (*Abst* introduces the new predicate p'). The new transformed program is $P' = A_P^C \uparrow 2 \cup \boldsymbol{Res}(A_P^C \uparrow 2)$ and $\boldsymbol{Res}(A_P^C \uparrow 2) = \{p'(a,a) \leftarrow\} \cup \{r(a) \leftarrow\} \cup \{r(f(X)) \leftarrow r(X)\}$ where P' contains now $p(a,Y) \leftarrow p'(a,Y)$. Hence,

$$M_s(P')_{|Pred(P)} = M_s(P) = \{p(X,c), p(a,a), r(a), q(X,c), q(a,a), r(f(a)), \ldots\}.$$

Now, we can prove the correctness of the transformation.

Theorem 1. *Let P' be a bottom up partial deduction of a program P, $P' = A_P^C \uparrow n_0 \cup \boldsymbol{Res}(A_P^C \uparrow n_0)$. Then, if $A_P^C \uparrow n_0$ is inside-closed, $M_s(P')_{|Pred(P)} = M_s(P)$, thus P' is sound and complete w.r.t. P.*

In order to prove this theorem, we include a useful lemma from [6].

Lemma 1. *(Proposition II.1 from [6])*
Let $\mathcal{A} = \{A_i\}_{1 \le i \le n}$ be a set of positive literals, $\mathcal{B} = \{B_i \leftarrow B_{i,1}, \ldots, B_{i,n_i}\}_{1 \le i \le n}$ and $\mathcal{C} = \{C_{i,j} \leftarrow C_{i,j,1}, \ldots, C_{i,j,p_{i,j}}\}_{1 \le i \le n, 1 \le j \le n_i}$ be sets of definite clauses. Let

us further assume that $Var(\mathcal{A})$, $Var(\mathcal{B})$ *and* $Var(\mathcal{C})$ *are disjoint sets. Then, there exist substitutions* α *and* β *such that:*

$$(I)\begin{cases}\alpha = mgu((A_i)_{1\leq i\leq n},(B_i)_{1\leq i\leq n}),\\ \beta = mgu((B_{i,j}\alpha)_{1\leq i\leq n,1\leq j\leq n_i},(C_{i,j})_{1\leq i\leq n,1\leq j\leq n_i})\end{cases}$$

if and only if there exists substitutions γ *and* δ *such that:*

$$(II)\begin{cases}\gamma = mgu((B_{i,j})_{1\leq i\leq n,1\leq j\leq n_i},(C_{i,j})_{1\leq i\leq n,1\leq j\leq n_i}),\\ \delta = mgu((A_i)_{1\leq i\leq n},(B_i\gamma)_{1\leq i\leq n})\end{cases}$$

Moreover, if (I) *or* (II) *holds then* $\alpha\beta = \gamma\delta$ *(modulo variable renaming).*

The following corollary states that if $mgu((A_i)_{1\leq i\leq n},(B_i)_{1\leq i\leq n})$ exists, so does $mgu((H)_{1\leq i\leq n},(B_i)_{1\leq i\leq n})$ if $H_i \preceq A_i$ $(1 \leq i \leq n)$ and $Var((A_i)_{1\leq i\leq n})$, $Var((B_i)_{1\leq i\leq n}))$ and $Var((H)_{1\leq i\leq n})$ are disjoint sets.

Corollary 1. *Let* $(A_1,\ldots,A_n)$, $(B_1,\ldots,B_n)$ *and* $(H_1,\ldots,H_n)$ *be sets of atoms with* $Var(A_1,\ldots,A_n)$, $Var(B_1,\ldots,B_n)$ *and* $Var(H_1,\ldots,H_n)$ *disjoint sets such that* $(A_1,\ldots,A_n) = (H_1,\ldots,H_n)\gamma$. *Then, if* $mgu((A_1,\ldots,A_n),(B_1,\ldots,B_n))$ *exists, so does* $mgu((H_1,\ldots,H_n),(B_1,\ldots,B_n))$.

Proof. Apply the only-if of Lemma 1 with $\mathcal{A} = \{A_1,\ldots,A_n\}$, $\mathcal{C} = \{B_1,\ldots,B_n\}$ (seen as unit clauses) and $\mathcal{B}$ the set of clauses $\{H_1 \leftarrow H_1,\ldots,H_n \leftarrow H_n\}$.

Below, in order to avoid uninteresting technical details, we assume renaming apart and reason modulo variable renaming wherever appropriate.

Proof of Theorem 1

Proof. ***Soundness.*** Since every step in the computation of A_P^C represents a Horn clause program, we introduce the following definition of intermediate programs: $P_0 = \{\}$, $P_i = A_P^C \uparrow i \cup \boldsymbol{Res}(A_P^C \uparrow i)$. We assume that $P' = A_P^C \uparrow n \cup \boldsymbol{Res}(A_P^C \uparrow n)$ and prove by induction that for all $i \leq n$: $M_s(P_i)_{|Pred(P)} \subseteq M_s(P)$. Obviously, $M_s(P_0) \subseteq M_s(P)$. Suppose now that $M_s(P_k)_{|Pred(P)} \subseteq M_s(P)$ (Induction Hypothesis). The difference between P_{k+1} and P_k consists of:

1. clauses added by T_P^C: Consider $H_1 \leftarrow\tilde{L_1},\ldots,H_n \leftarrow\tilde{L_n}$ in $A_P^C \uparrow k$ and a clause $H \leftarrow B_1,\ldots,B_n$ in P such that $\alpha = mgu((B_1,\ldots,B_n),(H_1,\ldots,H_n))$ exists and $H\alpha \leftarrow (\tilde{L_1},\ldots,\tilde{L_n})\alpha \in A_P^C \uparrow(k+1)$. We have to prove that for each conjunction of atoms, $\tilde{D}$ in $M_s(P')$ holds that if $\beta = mgu(\tilde{D},(\tilde{L_1},\ldots,\tilde{L_n})\alpha)$, $H\alpha\beta \in M_s(P)$. By Lemma 1, the existence of α and β implies the existence of $\gamma = mgu((\tilde{L_1},\ldots,\tilde{L_n}),\tilde{D})$ and $\delta = mgu((H_1,\ldots,H_n)\gamma,(B_1,\ldots,B_n))$. By induction hypothesis, $H_1\gamma,\ldots,H_n\gamma \in M_s(P)$, and since $H \leftarrow B_1,\ldots,B_n$ is a clause of P, $H\delta = H\gamma\delta = H\alpha\beta \in M_s(P)$.
2. clauses $p(\tilde{t}) \leftarrow \tilde{B}$ removed by *Abst* and recovered through $\boldsymbol{Res}$ as $p'(\tilde{t}) \leftarrow \tilde{B}$ with $p'(\tilde{t})$ a predicate renaming of $p(\tilde{t})$. Obviously, for each conjunction of atoms, $\tilde{D}$ in $M_s(P')$ such that $\beta = mgu(\tilde{D},\tilde{B})$, $p'(\tilde{s})\beta \in M_s(P')$ and $p(\tilde{s})\beta \in M_s(P)$ (since the induction hypothesis holds for the original clause).
3. clauses $p(\tilde{s}) \leftarrow p'(\tilde{s})$ introduced by *Abst* (with $p'(\tilde{s})$ a predicate renaming of $p(\tilde{s})$): By (2), we know that if $p'(\tilde{t}) \in M_s(P')$, $p(\tilde{t}) \in M_s(P)$. Since abstraction must guarantee inside-closedness, if $\theta = mgu(p'(\tilde{s}),p'(\tilde{t}))$, $p'(\tilde{s})\theta = p'(\tilde{t})$ and thus $p(\tilde{s})\theta = p(\tilde{t}) \in M_s(P)$.

Completeness. We prove by induction that for every k, $T_P^C \uparrow k \subseteq M_s(P')$. Since $M_s(P) = T_P^C \uparrow \omega$, the result follows. $T_P^C \uparrow 0 = \{\} \subseteq M_s(P')$. Suppose $T_P^C \uparrow k \subseteq M_s(P')$. By monotonicity of T_P^C we need to consider only clauses added in the $(k+1)$-th application of T_P^C: $T_P^C \uparrow (k+1) \setminus T_P^C \uparrow k = \{H\delta \leftarrow true | H \leftarrow B_1, \ldots, B_n \in P, A_1 \leftarrow true, \ldots, A_n \leftarrow true \in T_P^C \uparrow k$, and $\delta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n))\}$. We need to prove that $H\delta \in M_s(P')$. Since $A_1, \ldots, A_n \in T_P^C \uparrow k$, $A_1, \ldots, A_n \in M_s(P')$ (by induction hypothesis). This implies the existence of clauses $H_1 \leftarrow \tilde{L_1}, \ldots, H_n \leftarrow \tilde{L_n}$ in P' and conjunctions of atoms $\tilde{D_1}, \ldots, \tilde{D_n} \in M_s(P')$ such that

$$\gamma = mgu((\tilde{D_1}, \ldots, \tilde{D_n}), (\tilde{L_1}, \ldots, \tilde{L_n})) \tag{1}$$

and $(H_1, \ldots, H_n)\gamma = (A_1, \ldots, A_n)$. Since $H_1 \leftarrow \tilde{L_1}, \ldots, H_n \leftarrow \tilde{L_n}$ are clauses of P', there exists an m such that some of these clauses are in $A_P^C \uparrow m$, and the others in $\boldsymbol{Res}(A_P^C \uparrow m)$. Let us assume that (possibly after renumbering) $H_1 \leftarrow \tilde{L_1}, \ldots, H_k \leftarrow \tilde{L_k} \in A_P^C \uparrow m$ and $H_{k+1} \leftarrow \tilde{L_{k+1}}, \ldots, H_n \leftarrow \tilde{L_n} \in \boldsymbol{Res}(A_P^C \uparrow m)$. Since the latter clauses have been abstracted in the process of computing $A_P^C \uparrow m$, there exist clauses $H'_{k+1} \leftarrow H'_{k+1}, \ldots, H'_n \leftarrow H'_n$ in $A_P^C \uparrow m$ such that

$$(H'_{k+1}, \ldots, H'_n)\eta = H_{k+1}, \ldots, H_n.$$

Since

$$\delta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n))$$

and $(A_1, \ldots, A_n) = (H_1, \ldots, H_n)\gamma$, $mgu((B_1, \ldots, B_n), (H_1, \ldots, H_n))$ exists (Corollary 1). Next, again applying Corollary 1, since

$$(H_1, \ldots, H_n) = (H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n)\eta,$$
$$\theta = mgu((B_1, \ldots, B_n), (H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n)) \tag{2}$$

exists. Recall that

$$H \leftarrow B_1, \ldots, B_n \in P \text{ and } \begin{cases} H_1 \leftarrow \tilde{L_1} \\ \ldots \\ H_k \leftarrow \tilde{L_k} \\ H'_{k+1} \leftarrow H'_{k+1} \\ \ldots \\ H'_n \leftarrow H'_n \end{cases} \in A_P^C \uparrow m \tag{3}$$

Together (2) and (3) imply that $H\theta \leftarrow (\tilde{L_1}, \ldots, \tilde{L_k}, H'_{k+1}, \ldots, H'_n)\theta \in T_P^C(A_P^C \uparrow m)$ and hence is a clause of P'. Now, from $(H'_{k+1}, \ldots, H'_n)\eta = (H_{k+1}, \ldots, H_n)$ it follows that there is an $\eta' = mgu((H'_{k+1}, \ldots, H'_n), (H_{k+1}\gamma, \ldots, H_n\gamma))$ where γ is defined as in (1). Moreover, (1) implies the existence of

$$\gamma' = mgu((\tilde{L_1}, \ldots, \tilde{L_k}), (\tilde{D_1}, \ldots, \tilde{D_k})).$$

Since $Var(H'_{k+1}), \ldots, Var(H'_n), Var(H_{k+1}\gamma), \ldots, Var(H_n\gamma), Var(\tilde{L_1}), \ldots, Var(\tilde{L_k}), Var(\tilde{D_1}), \ldots, Var(\tilde{D_k})$ are disjoint sets, η' and γ' can be composed into $\eta'\gamma'$:

$$\eta'\gamma' = mgu((\tilde{L_1}, \ldots, \tilde{L_k}, H'_{k+1}, \ldots, H'_n), (\tilde{D_1}, \ldots, \tilde{D_k}, H_{k+1}\gamma, \ldots, H_n\gamma)).$$

Since

$$\begin{aligned} &(H_1, \ldots, H_k, H'_{k+1}, \ldots, H'n)\eta'\gamma' \\ &= (H_1\gamma', \ldots, H_k\gamma', H_{k+1}\gamma\gamma', \ldots, H_n\gamma\gamma') \\ &= (H_1\gamma, \ldots, H_k\gamma, H_{k+1}\gamma, \ldots, H_n\gamma) \\ &= A_1, \ldots, A_n \end{aligned}$$

and $\delta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n))$,

$$\delta = mgu((B_1, \ldots, B_n), (H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n)\eta'\gamma')$$

Using the if-part of Lemma 1 with $\mathcal{A} = \{B_1, \ldots, B_n\}$, $\mathcal{B}$ the $A_P^C \uparrow m$-clauses of (3) and as $\mathcal{C}$ the set $\{\tilde{D}_1, \ldots, \tilde{D}_k, H_{k+1}\gamma, \ldots, H_n\gamma\}$ (seen as a set of unit clauses), the existence of $\eta'\gamma'$ and δ implies the existence of θ (as in (2)) and

$$\sigma = mgu((\tilde{L}_1, \ldots, \tilde{L}_k, H'_{k+1}, \ldots, H'_n)\theta, (\tilde{D}_1, \ldots, \tilde{D}_k, H_{k+1}\gamma, \ldots, H_n\gamma)).$$

$\tilde{D}_1, \ldots, \tilde{D}_k \in M_s(P')$ and $H_{k+1}\gamma, \ldots, H_n\gamma = (A_{k+1}, \ldots, A_n) \in M_s(P')$ and $H\theta \leftarrow (\tilde{L}_1, \ldots, \tilde{L}_k, H'_{k+1}, \ldots, H'_n)\theta$ is a clause of P'. It follows that $H\theta\sigma \in M_s(P')$, $H\theta\sigma = H\eta'\gamma'\delta$ through Lemma 1 and $H\eta'\gamma'\delta = H\delta$ since $H\eta'\gamma' = H$. Therefore, we have shown that $H\delta \in M_s(P')$. □

In [8,1,2] the equivalence between the operational semantics and the S-semantics is established. This important result guarantees that if $M_s(P) = M_s(P')$, the operational behavior of P and P' is essentially the same: For any goal G, $P \cup G$ has an SLD-refutation with c.a.s. θ if and only if $P' \cup G$ has an SLD-refutation with c.a.s. θ. Since all classical, top down, partial deduction techniques are formulated with respect to this operational semantics, this equivalence result together with Theorem 1 enables an integration of the two transformations.

Note that while our transformation preserves finite failures, it may convert infinite failure into finite failure, as the following example shows:

Example 7.

P : $q(a) \leftarrow$
$q(X) \leftarrow q(X)$

P' : $q(a) \leftarrow$

The query $q(b)$ fails infinitely in the original program P, whereas it fails finitely in the transformed program P'.

4 Concrete Control

In order to show how our framework can be used in practice, we now cast a concrete bottom up control strategy within it. The basic strategy was already operationally described in [21], but by presenting it within our conceptual framework, we are now able to derive an algorithm which is operationally both sound and complete (which was not completely the case in [21]).

Example 8. Consider the following set of clauses, $\mathcal{C}$, defining two well-known predicates *append* and *reverse* working with an abstract data type for list representation.

$$\begin{aligned} append(L_1, L_2, L_2) \leftarrow\ & list\_nil(L_1) \\ append(L_1, L_2, LR) \leftarrow\ & list\_notnil(L_1), list\_head(L_1, H), list\_tail(L_1, T), \\ & append(T, L_2, R), list\_cons(H, R, LR) \end{aligned}$$

$$\begin{aligned} reverse(L, A, A) \leftarrow\ & list\_nil(L_1) \\ reverse(L, A, R) \leftarrow\ & list\_notnil(L_1), list\_head(L_1, H), list\_tail(L_1, T), \\ & list\_cons(H, A, NA), reverse(T, NA, R) \end{aligned}$$

The ADT itself is represented by the following set of clauses, $\mathcal{A}$, using the classical concrete list representation.

$list\_nil([]) \leftarrow$ $\quad$ $list\_head([X|X_s], X) \leftarrow$
$list\_notnil([X|X_s]) \leftarrow$ $\quad$ $list\_tail([X|X_s], X_s) \leftarrow$
$\quad$ $list\_cons(X, X_s, [X|X_s]) \leftarrow$

The program $P = \mathcal{C} \cup \mathcal{A}$ can be seen as a library that we want to specialise w.r.t. the ADT represented by $\mathcal{A}$.

In top down partial deduction, ordering partially deduced atoms in tree-structures rather than sets is a commonly used control tool [17,18,15]. Likewise, rather than just using T_P^C to derive *sets* of clauses, we define a new operator, D_P, which orders the derived clauses into a DAG. By doing so, we make the relation between derived clauses explicit, facilitating later on the formulation of a sufficiently precise abstraction function. We denote with $\mathcal{D}_{\mathcal{HC}}$ the set of all such DAGs, and with *Clauses* a function $\mathcal{D}_{\mathcal{HC}} \rightarrow 2^{\mathcal{HC}}$, mapping a DAG to the set of clauses present in it. Before defining D_P, we introduce an intermediate function, *Add*:

Definition 10. *Add* : $\mathcal{D}_{\mathcal{HC}} \rightarrow \mathcal{D}_{\mathcal{HC}}$ *is a function such that for* $D \in \mathcal{D}_{\mathcal{HC}}$, $\{C_1, \ldots, C_n\} \subseteq Clauses(D)$ *and* $C \in \mathcal{HC}$, $Add(D, \{C_1, \ldots, C_n\}, C)$ *denotes a* DAG $D' \in \mathcal{D}_{\mathcal{HC}}$ *that is the result of adding* C *to* D, *with an edge from every* $C_i \rightarrow C$ *for* $(1 \leq i \leq n)$.

Definition 11. *For a definite program* P, *the* D_P*-operator is a function,* $D_P : \mathcal{D}_{\mathcal{HC}} \rightarrow \mathcal{D}_{\mathcal{HC}}$ *defined for any* DAG $D \in \mathcal{D}_{\mathcal{HC}}$ *as* $D_P(D) = D'$, *where* D' *is the result of repeatedly applying* $Add(D, \{C_1, \ldots, C_n\}, C)$ *for every* $C \in T_P^C(Clauses(D)) \setminus Clauses(D)$ *where* C *was derived from* $C_1, \ldots, C_n$ *by* T_P^C.

D_P is continuous and monotonic, and we define its ordinal powers as usual.

Example 9. With P defined as in Example 8, $D_P \uparrow 1$, $D_P \uparrow 2$ and $D_P \uparrow 3$ are depicted in Figures 1, 2 and 3 (see Appendix A) respectively.

The relation between D_P and T_P^C is obvious: For any DAG $D \in \mathcal{D}_{\mathcal{HC}}$:

$$Clauses(D_P(D)) = T_P^C(Clauses(D)) \text{ holds.}$$

This implicitly defines $\boldsymbol{\mathcal{G}en}(D)$ and $\boldsymbol{\mathcal{R}es}(D)$ for any D constructed through AD_P, defined below. Henceforth, we will often leave implicit the distinction between the DAG and the set of clauses it contains.

We want our abstraction function to abstract clauses that would otherwise lead to an infinite DAG. To that extent, following the terminology of [15] (and references therein), we first define a *well-quasi* relation and give a concrete instance of such a relation that can be used as a criterion to decide when to perform abstraction.

Definition 12. *We call a relation* $\mathcal{W} \subseteq \mathcal{HC}/_{\equiv} \times \mathcal{HC}/_{\equiv}$ well-quasi *on* $\mathcal{HC}$ *if and only if any infinite series of clauses* $C_1, \ldots, C_k, C_{k+1}, \ldots \in \mathcal{HC}/_{\equiv}$ *contains two clauses* C_i *and* C_j, $i \leq j$, *such that* $(C_i, C_j) \in \mathcal{W}$.

A well-known example of a well-quasi relation is the *homeomorphic embedding relation*. Basically, it is a well-quasi order on a finite alphabet (e.g. [7,18]), but the refined "strict" version below is not transitive. It remains well-quasi, though [15]. The following definition is taken from [14,15,10].

Definition 13. Let X, Y range over variables, f over functors, and p over predicates. As usual, $e_1 \prec e_2$ denotes that e_2 is a strict instance of e_1. Define $\trianglelefteq$ on terms and atoms:

$$
\begin{array}{ll}
X \trianglelefteq Y & \\
s \trianglelefteq f(t_1, \ldots, t_n) & \Leftarrow s \trianglelefteq t_i \text{ for some } i \\
f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n) & \Leftarrow s_i \trianglelefteq t_i \text{ for all } i \\
p(s_1, \ldots, s_n) \trianglelefteq p(t_1, \ldots, t_n) & \Leftarrow s_i \trianglelefteq t_i \text{ for all } i \text{ and } p(t_1, \ldots, t_n) \not\prec p(s_1, \ldots, s_n)
\end{array}
$$

The basic intuition behind the homeomorphic embedding relation is that $A \trianglelefteq B$ if A can be obtained from B (modulo variable renaming) by striking-out some functors in B. As such, it provides a starting point for detecting growing structure and hence possibly non-terminating processes.

Example 10. In the DAG of Example 9: $reverse([], R, R) \trianglelefteq reverse([X], A, [X|A])$ and $append([], L, L) \trianglelefteq append([X], L, [X|L])$.

Based on $\trianglelefteq$, we can define a well-quasi relation on $\mathcal{HC}$:

Definition 14. $\mathcal{W}_{\trianglelefteq} \subseteq \mathcal{HC}/_{\equiv} \times \mathcal{HC}/_{\equiv} = \{(H_1 \leftarrow \tilde{B}_1, H_2 \leftarrow \tilde{B}_2) \mid H_1 \trianglelefteq H_2\}$

Given a well-quasi relation $\mathcal{W}$ on $\mathcal{HC}$, we are in a position to define an abstraction function $Abst_{\mathcal{W}} : \mathcal{D}_{\mathcal{HC}} \rightarrow \mathcal{D}_{\mathcal{HC}}$ through the algorithm given below. It uses operations $remove(D, C) : \mathcal{D}_{\mathcal{HC}} \times \mathcal{HC} \rightarrow \mathcal{D}_{\mathcal{HC}}$ which removes a clause C from a DAG D and $replace(D, C, C') : \mathcal{D}_{\mathcal{HC}} \times \mathcal{HC} \times \mathcal{HC} \rightarrow \mathcal{D}_{\mathcal{HC}}$ replacing a clause C in a DAG D by C'. Given atoms A and B, $msg(A, B)$ denotes their *most specific generalisation* (see a.o. [18,15]).

Algorithm 1
Input: a DAG $D \in \mathcal{D}_{\mathcal{HC}}$, a well-quasi relation $\mathcal{W}$ on $\mathcal{HC}$.
Output: a DAG $Abst_{\mathcal{W}}(D) \in \mathcal{D}_{\mathcal{HC}}$.
$D_0 := D; i := 0;$
repeat
  if there exists a path $C_1 \Longrightarrow C_2$ in D_i such that $(C_1, C_2) \in \mathcal{W}$
    where C_1 denotes $p_1(\tilde{t}) \leftarrow \tilde{B}_1$ and C_2 denotes $p_2(\tilde{s}) \leftarrow \tilde{B}_2$.
  then if there exists $p_2(\tilde{r}) \leftarrow p_2'(\tilde{r}) \in \boldsymbol{Gen}(D_i)$ such that $p_2(\tilde{r}) \preceq p_2(\tilde{s})$
    then $D_{i+1} := remove(D_i, C_2)$;
    else $D_{i+1} := replace(D_i, C_2, p_2(\tilde{u}) \leftarrow p_2''(\tilde{u}))$;
      where $p_2(\tilde{u}) = msg(p_1(\tilde{t}), p_2(\tilde{s}))$
      and $p_2''(\tilde{u})$ is a new predicate renaming of $p_2(\tilde{u})$
  $i := i + 1;$
until $D_i = D_{i-1}$;
$Abst_{\mathcal{W}}(D) := D_i;$

Definition 15. *We call a well-quasi relation $\mathcal{W}$ on $\mathcal{HC}$* suitable (for abstraction) *if $(A, B) \in \mathcal{W}$ implies $msg(A, B) \neq B$.*

Note that strict homeomorphic embedding as defined in Definitions 13 and 14 is suitable [15]. Now we can state the following correctness results.

Theorem 2. *Given a suitable well-quasi relation $\mathcal{W}$ on $\mathcal{HC}$, Algorithm 1 terminates for any input* DAG *$D \in \mathcal{D}_{\mathcal{HC}}$.*

Proof. In the algorithm, $D_i \neq D_{i-1}$ as long as there exists a path $C_1 \Longrightarrow C_2$ in D_i such that $(C_1, C_2) \in \mathcal{W}$. In each loop, either C_2 is removed from the DAG, resulting in precisely one path less to consider in the next loop, or a generalisation is made. Thus, the only thing left to prove is that there cannot occur an infinite sequence of consecutive generalisations. This follows because suitability of $\mathcal{W}$ implies that generalising is strict and therefore well-founded. □

It can be easily verified that, given a suitable $\mathcal{W}$, $Abst_{\mathcal{W}}$ as defined in the above algorithm complies with Definition 3, and hence is an abstraction function, $Abst_{\mathcal{W}} : \mathcal{D}_{\mathcal{HC}} \rightarrow \mathcal{D}_{\mathcal{HC}}$. Again, we combine $Abst_{\mathcal{W}}$ with D_P to obtain a concretisation of A_P^C: $AD_P = Abst_{\mathcal{W}} \circ D_P$ and $AD_P : \mathcal{D}_{\mathcal{HC}} \rightarrow \mathcal{D}_{\mathcal{HC}}$. Since $Abst_{\mathcal{W}}(\{\}) = \{\}$, Proposition 1 ensures monotonicity of a sequence of AD_P applications and we can define its ordinal powers in the usual way.

Theorem 3. *AD_P is finitary.*

Proof. Suppose AD_P is not finitary: There does not exist $n \in \mathbb{N}$ such that $AD_P \uparrow n$ is a fixpoint. Since AD_P is monotonic, it will then construct an infinite D. Now, for any $D \in \mathcal{D}_{\mathcal{HC}}$, $Clauses(D)$ and $T_P^C(Clauses(D))$ are both finite sets, thus in each iteration of AD_P, only a finite number of new clauses are added to D. The only way then in which D can become infinite is through an infinite path in D. But this would mean there exists an infinite series of clauses $C_1, \ldots, C_k, \ldots$ without two clauses C_i, C_j $(i \leq j)$ such that $(C_i, C_j) \in \mathcal{W}$, contradicting well-quasiness of $\mathcal{W}$. □

Corollary 2. *Given a program P and an abstraction function $Abst_{\mathcal{W}}$, the program $AD_P \uparrow n_0 \cup \boldsymbol{\mathcal{R}es}(Clauses(AD_P \uparrow n_0))$, where $AD_P \uparrow n_0$ is the fixpoint of AD_P, is a bottom up partial deduction of P (using $Abst_{\mathcal{W}}$).*

Example 11. The bottom up partial deduction of P (using $Abst_{\mathcal{W}_{\trianglelefteq}}$) is the following program (where only the predicates of interest, *append* and *reverse*, are given)

$append([], L, L) \leftarrow$
$append([X], L, [X|L]) \leftarrow$
$append([X|X_s], Y, [X|Z]) \leftarrow append(X_s, Y, Z)$
$append(X, Y, Z) \leftarrow append(X, Y, Z)$

$reverse([], R, R) \leftarrow$
$reverse([X], A, [X|A]) \leftarrow$
$reverse([X|X_s], A, R) \leftarrow reverse(X_s, [X|A], R)$
$reverse(X, Y, Z) \leftarrow reverse(X, Y, Z)$

For both predicates, input lists of length 0 or 1 are captured by a unit clause. This is due to the fact that when *Abst* detects that

$$(reverse([], R, R) \trianglelefteq reverse([X], A, [X|A]))$$

the latter is generalised (so called *child generalisation* in top down partial deduction terminology, see Figures 3 and 4). Generalising the former instead – and removing its descendant graph – (*parent generalisation*) usually results in less generated code (in the example: only one unit clause, for the empty input list, would be generated). However, child generalisation is necessary if we want A^C_P (and AD_P) to remain monotonic. On the other hand, when parent generalisation is used, a sequence of A^C_P applications is no longer monotonic, but A^C_P can still be finitary due to a "sufficiently large" monotonic subsequence. How to incorporate this exactly into the framework is a topic of further research. The concrete control strategy of [21] uses parent generalisation.

5 Discussion & Future Work

This paper provides a formal framework for bottom up specialisation. We enrich the compositional T^C_P operator [2,1] with abstraction and thus obtain a generic bottom up partial deduction operator A^C_P. We prove soundness and completeness of the transformation provided a particular condition is imposed on A^C_P. This inside-closedness condition plays a key role in deciding whether actually to rename an atom or use the atom itself in the abstraction. Always renaming during abstraction creates a set $A^C_P \uparrow n_0$ which trivially is inside-closed, and hence the residual program derived from it is sound and complete. The clauses from $\boldsymbol{Gen}(A^C_P \uparrow n_0)$ serve as "translation" clauses between the renamings and the original predicates. On the other hand, never renaming during abstraction creates a set $A^C_P \uparrow n_0$ leading to a residual program P' that *possibly* is not sound with respect to P. Deriving P' from P by the control strategy of [21] provides an example: if $A \in M_s(P')$, it is possible that $A \notin M_s(P)$; however A is "correct" in the sense that there does exist an $A' \in M_s(P)$ such that $A' \preceq A$ (Theorem 3.16 in [21]). In general, clauses from $\boldsymbol{Gen}(A^C_P \uparrow n_0)$ that bear the same predicate in head and body, can be filtered from the residual program since they do not serve as "translation" clauses. Herefore, less renaming results in smaller transformed programs. Examples 10 and 11 required none.

The aim of bottom up partial deduction should be to maximize the propagation of information in the program, while preserving the least S-Herbrand model. Without abstraction, the process reduces to bottom up evaluation and $P' = M_s(P)$. Enriching T_P^C with abstraction introduces several new complications. While in general A_P^C is not continuous, it can be defined finitary. This is mandatory if we plan to use it for program transformation, since the transformation as well as the resulting program must be finite. Also, while the defined A_P^C operator is monotonic, practical control (as the one in [21]) may use parent-generalisation (cutting away parts of a DAG built so far), and destroy overall A_P^C monotonicity.

As noted in [2,1], $T_P^C \uparrow \omega$ is in essence a set of resultants which can be seen as the result of a top down partial deduction of P w.r.t. a set of atomic goals of the form $p(X_1, \ldots, X_n)$. As a consequence of abstraction, this will not be the case with our transformation. Reconsider the program of Example 2 to which the clause $q(Y) \leftarrow p(X, Y)$ is added. *Abst* is defined as in Example 3, resulting in a set I_2 which includes the clause $q(Y) \leftarrow p(a, Y)$. Although this clause was produced by T_P^C starting from an abstraction, it can not be derived as a resultant from a top down partial deduction of $q(Y)$, since its body contains the constant a as the result of an earlier bottom up propagation.

Recently, an integration between partial deduction and abstract interpretation (both top down) has been established [13]. A similar integration between our framework and bottom up abstract interpretation (see e.g. [3]) might be feasible and is a topic of further research. Also the exact relation between bottom up and top down partial deduction needs to be scrutinised, as well as an integration between the two techniques. Examples in [21] indicate that a combination of the described bottom up transformation with a top down component using a trivial control strategy – deterministic unfolding – is capable of specialising the Vanilla meta interpreter satisfactorily. Equally good results *can* be obtained by a top down control strategy alone, but often at the cost of not completely general and/or automatic techniques (e.g. [12]), or a non-trivial and complex control mechanism. In [20], a sophisticated approach is described based on extending a local unfolding rule by global information. Although the resulting technique is completely general and automatic, in spite of its complexity, it turns out to be not sufficiently effective when more involved meta interpreters are at hand. The same results can be obtained by two separate but rather straightforward control strategies, and the combined approach is conceptually cleaner than the sophisticated top down one. We plan to investigate how the two control strategies can be tuned - separately and combined - to obtain good specialisation of other, more involved meta interpreters.

Acknowledgments

We thank Karel De Vlaminck for his continuing support and interest in this work. We also thank anonymous referees for interesting and constructive comments.

References

1. 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. 56, 57, 63, 67, 68
2. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994. 56, 57, 63, 67, 68
3. Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 451–464, Charleston, South Carolina, January 10–13, 1993. ACM Press. 68
4. Y. Cosmadopoulos, M. Sergot, and R. W. Southwick. Data-driven transformation of meta-interpreters: A sketch. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 301–308. Springer Verlag, 1991. 56
5. D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 615–616, Portland, Oregon, December 1995. MIT Press. 54
6. F. Denis and J.P. Delahaye. Unfolding, procedural and fixpoint semantics of logic programs. In *8th Annual Symposium on Theoretical Aspects of Computer Science*, volume 480 of *LNCS*, pages 511–522. Springer, 1991. 60
7. 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. 65
8. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science*, 69:289–318, 1989. 57, 63
9. J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press. 54
10. R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S.D. Swierstra, editors, *Proceedings PLILP'96*, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag, LNCS 1140. 65
11. H. J. Komorowski. Synthesis of programs in the partial deduction framework. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 377–403. AAAI Press, 1991. 56
12. A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8(1):61–70, 1990. 68
13. M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Manchester, United Kingdom, June 1998. MIT-Press. 68
14. 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, *Proceedings Dagstuhl Seminar on Partial Evaluation*, pages 263–283, Schloss Dagstuhl, Germany, 1996. Springer-Verlag, LNCS 1110. 65
15. 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), 1998. 64, 65, 66

16. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991. 54
17. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Kanagawa, Japan, June 1995. MIT Press. 64
18. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 465–479, Portland, Oregon, December 1995. MIT Press. 64, 65
19. W. Vanhoof. Bottom up information propagation for partial deduction. In M. Leuschel, editor, *Proceedings of the International Workshop on Specialization of Declarative Programs and its Applications*, pages 73 – 82, Port Jefferson, Long Island N.Y. (USA), October 1997. Proceedings available as report CW255, Dept. of Computer Science, K.U.Leuven, Belgium. 55
20. W. Vanhoof and B. Martens. To parse or not to parse. In N. E. Fuchs, editor, *Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation LOPSTR '97*, volume 1463 of *Lecture Notes in Computer Science*, pages 314 – 333, Leuven, Belgium, 1997. Springer-Verlag. 55, 68
21. W. Vanhoof, B. Martens, D. De Schreye, and K. De Vlaminck. Specialising the other way around. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Manchester, United Kingdom, June 1998. MIT-Press. 54, 55, 63, 67, 68

A Figures

In this appendix, the DAGs built in the examples throughout Section 4 are depicted. For simplicity, variables in the DAGs are not renamed.

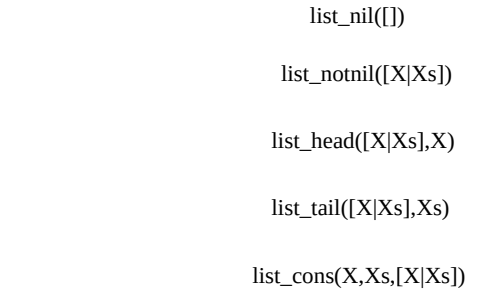


Fig. 1. $D_P \uparrow 1 = AD_P \uparrow 1$

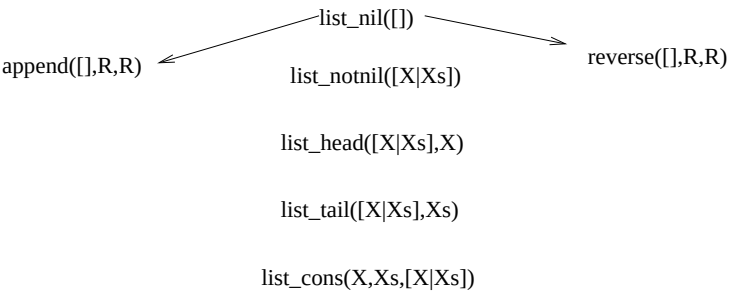


Fig. 2. $D_P \uparrow 2 = AD_P \uparrow 2$

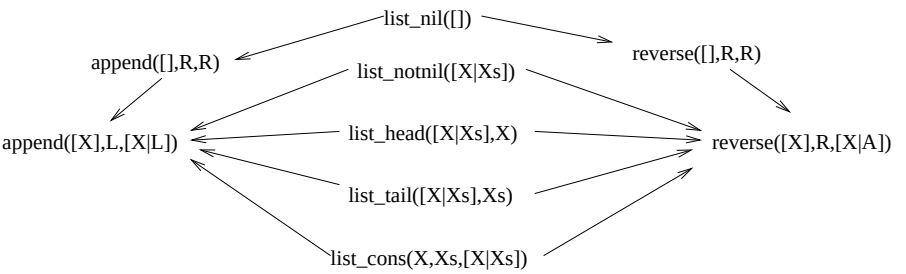


Fig. 3. $D_P \uparrow 3$

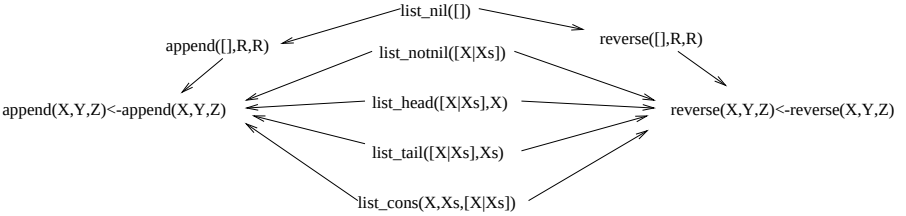


Fig. 4. $AD_P \uparrow 3$

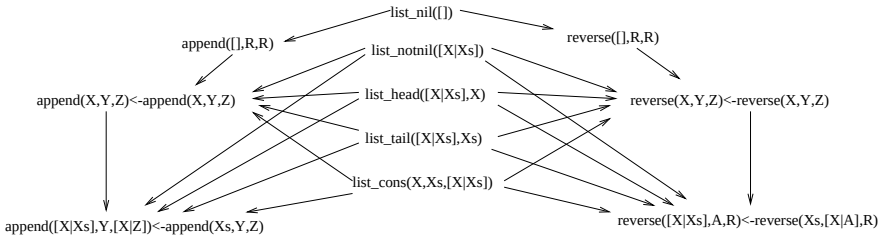


Fig. 5. $D_P \uparrow 4$

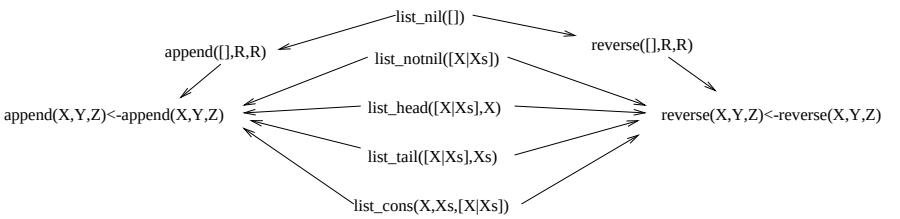


Fig. 6. $AD_P \uparrow 4 = AD_P \uparrow 3$

Termination of Logic Programs with `block` Declarations Running in Several Modes

Jan–Georg Smaus[1], Pat Hill[2], and Andy King[1]

[1] University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom
{j.g.smaus,a.m.king}@ukc.ac.uk
[2] University of Leeds, Leeds, LS2 9JT, United Kingdom
hill@scs.leeds.ac.uk

Abstract. We show how termination of logic programs with delay declarations can be proven. Three features are distinctive of this work: (a) we assume that predicates can be used in several modes; (b) we show that `block` declarations, which are a very simple delay construct, are sufficient; (c) we take the selection rule into account, assuming it to be as in most Prolog implementations. Our method is based on identifying the so-called *robust* predicates, for which the textual position of an atom using this predicate is irrelevant. The method can be used to verify existing programs, and to assist in writing new programs. As a byproduct, we also show how programs can be proven to be free from occur-check and floundering.

1 Introduction

Delay declarations are provided in several logic programming languages to allow for more user-defined control [7,8,18] as opposed to the standard left-to-right selection rule. An atom in a query is selected for resolution only if its arguments are instantiated to a specified degree.

In this paper we present a method of ensuring termination of programs with delay declarations. As far as possible, we translate the problem to showing termination for a corresponding program with ordinary left-to-right execution. We assume that for the corresponding program, termination has been shown using some existing technique [1].

Three distinctive features of this work make its contribution: (a) it is assumed that procedures may run in more than one mode; (b) we concentrate on `block` declarations, which are a particularly simple and efficient delay construct; (c) the selection rule is taken into account.

(a) Apart from the test-and-generate paradigm (coroutining) [15], allowing procedures to run in more than one mode is probably the most important application of delay declarations. Although other authors have not explicitly assumed multiple modes, their theory and examples only become fully relevant under that assumption. Whether this is a better approach than generating multiple versions of each predicate [18] is an ongoing discussion [6].

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 73–88, 1998.

(b) The `block` declarations declare that certain arguments of an atom must be *non-variable* before that atom can be selected. Insufficiently instantiated atoms are delayed. As demonstrated in SICStus [8], `block` declarations can be efficiently implemented; the test whether the arguments are non-variable has negligible impact on performance. Termination clearly depends on the instantiation of the arguments of the query. For example, the `append` predicate has infinitely many answers when called with uninstantiated arguments and therefore does not terminate, but it terminates when either the first *or* the third argument is a list of bounded length. Although it cannot be tested in a single step *which* of these arguments is a list of bounded length, `block` declarations are still sufficient.

(c) The property of termination may critically depend on the selection rule, that is the rule which determines, for a derivation, the order in which atoms are selected. We assume that derivations are *left-based*, which are derivations where (allowing for some exceptions, concerning the execution order of two literals woken up simultaneously) the left-most non-delayed atom is selected. This is intended to model derivations in the common implementations of Prolog with `block` declarations. Other authors have avoided the issue by abstracting from a particular selection rule [2,10]; considering left-based selection rules on a heuristic basis [15]; or making the very restrictive assumption of *local selection rules* [11].

Circular modes (when a predicate uses its own output as input) and *speculative output bindings* (bindings made before it is known that a solution exists) are known sources of loops [15]. We develop this explanation further by identifying predicates which have the undesirable property of looping when they are called with *insufficient* (that is, non-variable but still insufficiently instantiated) input. For instance, the query `permute(A,[1|B])` loops, although the query `permute(A,[1,2])` terminates. The idea of our method for proving termination is that, for such predicates, calls with insufficient input should never arise. This can be ensured by appropriate ordering of atoms in the clause bodies. This actually works in several modes, provided not too many predicates have this undesirable property.

This paper is organised as follows. The next section defines some essential concepts and notations. Sect. 3 introduces some concepts needed later, which are also useful for proving programs free from occur-check and floundering. Sect. 4 is about termination. Sect. 5 investigates related work. Sect. 6 concludes with a summary and a look at ongoing and future work.

2 Essential Concepts and Notations

We base the notation on [2,9]. For the examples we use SICStus notation [8]. The set of variables in a syntactic object o is denoted by $vars(o)$. A syntactic object is **linear** if every variable occurs in it at most once. A **flat** term is a variable or a term $f(x_1, \ldots, x_n)$, where $n \geq 0$ and the x_i are distinct variables. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$.

For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. A **mode of a program** is a set of modes, one mode for each of its predicates. A program can have several modes, so whenever we refer to the input and output positions, this is always with respect to the particular mode which is clear from the context. To simplify the notation, an atom written as $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ and $\mathbf{t}$ are the vectors of terms filling the input and output positions, respectively.

A **type** is a set of terms closed under substitution. A type is called **variable type** if it contains variables and **non-variable type** otherwise. In the examples, we use the following types: *any* is the type containing all terms, *list* is the type of all (nil-terminated) lists, *int* the type of integers, and *il* is the type of all integer lists. We write $t : T$ for "t is in type T". It is assumed that each argument position of each predicate p/n has a type associated with it. These types are indicated by writing the atom $p(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n$ are types. The type of a program is a set of such atoms, one for each predicate. A term t is **typeable wrt.** T if there is a substitution θ such that $t\theta : T$. A term t occurring in an atom in some position is **typeable** if it is typeable wrt. the type of that position.

A `block` declaration [8] for a predicate p/n is a set of atoms of the form $p(b_1, \ldots, b_n)$, where $b_i \in \{?, -\}$ for $i \in \{1, \ldots, n\}$. A **program** consists of a set of clauses and a set of `block` declarations, one for each predicate defined by the clauses. If P is a program, an atom $p(t_1, \ldots, t_n)$ is **selectable in** P if for each atom $p(b_1, \ldots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \ldots, n\}$ such that t_i is non-variable and $b_i = -$.

A **query** is a finite sequence of atoms. A **derivation step** for a program P is a pair $\langle Q, \theta\rangle; \langle R, \theta\sigma\rangle$, where $Q = Q_1, a, Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; a an atom; $h \leftarrow B$ (a variant of) a clause in P and σ the most general unifier of $a\theta$ and h. We call $a\theta$ the **selected atom** and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$.

A **derivation** ξ **for** a program P is a sequence $\langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$, where $\theta_0 = \emptyset$ and each successive pair $\langle Q_i, \theta_i\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$ in ξ is a derivation step. Alternatively, we also say that ξ is a **derivation of** $P \cup \{Q_0\}$. We also denote ξ by $Q_0; Q_1\theta_1; \ldots$. A derivation is an **LD-derivation** if the selected atom is always the leftmost atom in a query. A **delay-respecting derivation** for a program P is a derivation where the selected atom is always selectable in P. We say that it **flounders** if it ends with a non-empty query where no atom is selectable.

If $Q, a, R; (Q, B, R)\theta$ is a step in a derivation, then each atom in $B\theta$ is a **direct descendant** of a, and $b\theta$ is a **direct descendant** of b for all $b \in Q, R$. We say b is a **descendant of** a if (b, a) is in the reflexive, transitive closure of the relation *is a direct descendant*. The descendants of a *set* of atoms are defined in the obvious way. If, for a derivation $\ldots Q; \ldots; Q'; Q'' \ldots$, the selected atom in $Q'; Q''$ is a descendant of an atom a in Q, then $Q'; Q''$ is called an **a-step**.

Consider a delay-respecting derivation $Q_0; \ldots; Q_i; Q_{i+1}$, where $Q_i = R_1, a, R_2$, and R_1 contains no selectable atom, and a is *not* the selected

atom in $Q_i; Q_{i+1}$. Then a is **delayed** in $Q_i; Q_{i+1}$. An atom is **waiting** if it is the descendant of a delayed atom. A delay-respecting derivation $Q_0; Q_1 \ldots$ is **left-based** if in each Q_i, a non-waiting atom is selected only if there is no selectable atom to the left of it in Q_i.

3 Permutations and Modes

In [2], the concepts of *nicely moded* and *well typed* are introduced, assuming that each predicate has a single mode. They are used to show that the occur-check can safely be omitted and that derivations do not flounder. The idea is that in a query, every piece of data is produced (i.e. output) before it is consumed (i.e. input), and every piece of data is produced only once. Here "before" refers to the textual position in a query.

We generalise these concepts and results by considering a permutation of the atoms in each clause body in a program (and in each query), such that an LD-derivation for the reordered program is automatically delay-respecting, and thus, `block` declarations are effectively unnecessary. These permutations are used to compare a program with the (theoretically) reordered program; it is not intended that the program is *actually* changed. Since the permutations are different in each mode, this would commit the program to a particular mode.

3.1 Permutation Nicely Moded Programs

In a *nicely moded* query, a variable occurring in an input position does not occur later in an output position, and each variable in an output position occurs only once. We generalise this to *permutation nicely moded*.

Definition 3.1 (permutation nicely moded). Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \ldots, n\}$. Q is **π-nicely moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$

$$vars(\mathbf{s}_i) \cap \bigcup_{\pi(j) \geq \pi(i)} vars(\mathbf{t}_j) = \emptyset.$$

The query[1] $\pi(Q)$ is a **nicely moded query corresponding to** Q.

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-nicely moded** if Q is π-nicely moded and $\mathbf{t}_0, \ldots, \mathbf{t}_n$ is a linear vector of terms. The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow \pi(Q)$ is a **nicely moded clause corresponding to** C.

A query (clause) is **permutation nicely moded** if it is π-nicely moded for some π. A program P is **permutation nicely moded** if all of its clauses are. A **nicely moded program corresponding to** P is a program obtained from P by replacing every clause C in P with a nicely moded clause corresponding to C.

[1] Given a sequence $o_1, \ldots, o_n$, we write $\pi(o_1, \ldots, o_n)$ for $o_{\pi^{-1}(1)}, \ldots, o_{\pi^{-1}(n)}$, i.e. the sequence obtained by applying π to $o_1, \ldots, o_n$.

Note that in the clause head, the letter t is used for input and s is used for output, whereas in the body atoms it is vice versa.

Example 3.1.

```
:- block permute(-,-).
permute([], []).
permute([U | X], Y) :-
  permute(X, Z),
  delete(U, Y, Z).

:- block delete(?,-,-).
delete(X,[X|Z],Z).
delete(X,[U|Y],[U|Z]) :- delete(X,Y,Z).
```

In mode $\{\texttt{permute}(I,O), \texttt{delete}(I,O,I)\}$, this program is nicely moded. In mode $\{\texttt{permute}(O,I), \texttt{delete}(O,I,O)\}$, it is permutation nicely moded, since the second clause for `permute` is $\langle 2,1\rangle$-nicely moded, and the other clauses are nicely moded.

Note that the problem of *finding* a mode for a program so that it is nicely moded is considered in [4]. We are not concerned with this here.

We show that there is a persistence property for permutation nicely-modedness similar to that for nicely-modedness in [2].

Lemma 3.1. Every resolvent of a permutation nicely moded query Q and a permutation nicely moded clause C, where $vars(Q) \cap vars(C) = \emptyset$, is permutation nicely moded.

Proof. Let $Q = a_1, \ldots, a_n$ be a π-nicely moded query and $h \leftarrow b_1, \ldots, b_m$ be a ρ-nicely moded clause, and suppose for some $k \in \{1, \ldots, n\}$, h and a_k are unifiable with unifier θ. By Def. 3.1, $a_{\pi^{-1}(1)}, \ldots, a_{\pi^{-1}(n)}$ and $h \leftarrow b_{\rho^{-1}(1)}, \ldots, b_{\rho^{-1}(m)}$ are nicely moded. Thus by [2, Lemma 11][2]

$$a_{\pi^{-1}(1)}, \ldots, a_{\pi^{-1}(\pi(k)-1)}, b_{\rho^{-1}(1)}, \ldots, b_{\rho^{-1}(m)}, a_{\pi^{-1}(\pi(k)+1)}, \ldots, a_{\pi^{-1}(n)}\ \theta$$

is nicely moded. This implies that $a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n\ \ \theta$ is ϱ-nicely moded, where $\varrho(i)$ is defined as:

$\pi(i)$	if $i < k,\ \pi(i) < \pi(k)$
$\pi(i) + m - 1$	if $i < k,\ \pi(i) > \pi(k)$
$\pi(k) - 1 + \rho(i - k + 1)$	if $k \leq i \leq k + m - 1$
$\pi(i - m + 1)$	if $k + m \leq i \leq n + m - 1,\ \pi(i - m + 1) < \pi(k)$
$\pi(i - m + 1) + m - 1$	if $k + m \leq i \leq n + m - 1,\ \pi(i - m + 1) > \pi(k)$

□

Fig. 1 illustrates ϱ when $Q = a_1, a_2, a_3, a_4$, $\pi = \langle 4,3,1,2\rangle$, $C = h \leftarrow b_1, b_2$, $\rho = \langle 2,1\rangle$, and $k = 2$. Thus $\varrho = \langle 5,4,3,1,2\rangle$. The following corollary generalises this from a single derivation step to derivations.

[2] Unlike [2], we included the condition that $\mathbf{t}_0$ is linear in Def. 3.1.

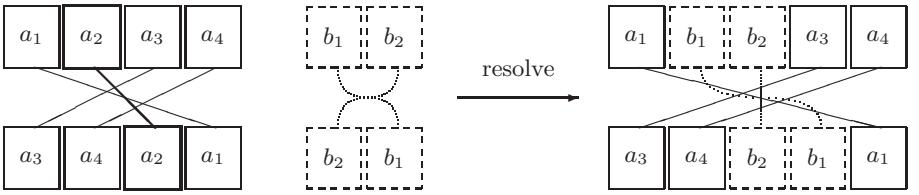


Fig. 1. The permutation ϱ for the resolvent

Corollary 3.2. Let P be a permutation nicely moded program, $Q = a_1, \ldots, a_n$ be a π-nicely moded query and $i, j \in \{1, \ldots, n\}$ such that $\pi(i) < \pi(j)$. Let $Q; \ldots; R$ be a derivation for P and suppose $R = b_1, \ldots, b_m$ is ρ-nicely moded. If for some $k, l \in \{1, \ldots, m\}$, b_k is a descendant of a_i and b_l is a descendant of a_j, then $\rho(k) < \rho(l)$. (*Proof [17]*)

As an aside, we now use permutation nicely-modedness to show when the occur-check can safely be omitted.

Definition 3.2. A derivation is called **occur-check free** [2,3] if no execution of the Martelli-Montanari unification algorithm [13] performed during this derivation ends with a system of term equations including an equation $x = t$, where x is not t, but x occurs in t.

If P and Q are nicely moded, then all derivations of $P \cup \{Q\}$ are occur-check free [2, Thm. 13]. The following theorem is a trivial consequence of this and Lemma 3.1.

Theorem 3.3 (occur check). Let P and Q be permutation nicely moded. Then all derivations of $P \cup \{Q\}$ are occur-check free.

3.2 Permutation Well Typed Programs

To show that derivations do not flounder, [2] defines *well-typedness*, which is a generalisation of a simpler concept called *well-modedness*. The idea is that given a query H, a, F, if H is resolved away, then a becomes sufficiently instantiated to be selected. As with the modes, we assume that the types are given. In the examples, they will be the obvious ones.

Definition 3.3 (permutation well typed). Let $n \geq 0$ and π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \ldots, n\}$. Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$[3] is the type of p_i for all $i \in \{1, \ldots, n\}$. Then Q is **π-well typed** if for all $i \in \{1, \ldots, n\}$ and every substitution σ

$$\models \Big(\bigwedge_{\pi(j)<\pi(i)} \mathbf{t}_j\sigma : \mathbf{T}_j \Big) \Rightarrow \mathbf{s}_i\sigma : \mathbf{S}_i. \qquad (*)$$

[3] $\mathbf{S}_i$, $\mathbf{T}_i$ are the vectors of types of the input and output arguments, respectively.

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$, where $p(\mathbf{T}_0, \mathbf{S}_{n+1})$ is the type of p, is **π-well typed** if $(*)$ holds for all $i \in \{1, \ldots, n+1\}$ and every substitution σ.

A **permutation well typed** query (clause, program) and a **well typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Def. 3.1.

Example 3.2. Consider Ex. 3.1 and assume the type $\{\texttt{permute}(list, list), \texttt{delete}(any, list, list)\}$. The program is well typed for mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, I)\}$, and permutation well typed for mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$, with the same permutations as in Ex. 3.1. The same holds assuming type $\{\texttt{permute}(il, il), \texttt{delete}(int, il, il)\}$.

We now give a statement analogous to Lemma 3.1. The proof is like that of Lemma 3.1, using Lemma 23 instead of 11 in [2].

Lemma 3.4. Every resolvent of a permutation well typed query Q and a permutation well typed clause C, where $vars(Q) \cap vars(C) = \emptyset$, is permutation well typed.

Theorem 3.5. Let P be a permutation well typed program and Q be a permutation well typed query. Assume that an atom is selectable if it is non-variable in all input positions of non-variable type. Then no delay-respecting derivation of $P \cup \{Q\}$ flounders. *(Proof [17])*

For the program in Ex. 3.2, the above lemma shows that no permutation well typed query can flounder.

4 Termination

So far we have introduced two useful concepts of "modedness" and "typedness". In this section, we will build on these to show termination.

We are interested in termination in the sense that *all* derivations of a query are finite. Therefore the clause order in a program is irrelevant. Furthermore, we are concerned with how delay declarations can affect the termination of a program. Thus it is assumed that termination for the corresponding nicely moded and well typed programs has been shown by some existing method for LD-derivations [1]. We first give some examples to illustrate the issues.

Example 4.1. The `permute` predicate (Ex. 3.1) loops for the query `permute(V,[1])` because `delete` produces a *speculative output binding* [15]: The output variable `Y` is bound before it is known that this binding will never have to be undone. Assuming left-based derivations, termination in both modes can be ensured by replacing the second clause with

```
permute([U | X1], Y) :-
  delete(U, Y, Z),
  permute(X1, Z).
```

This heuristic is called *putting recursive calls last* [14]. The example suggests that one cannot give reasonable termination guarantees without making such strong assumptions about the selection rule.

However, the heuristic of putting recursive calls last cannot explain the appropriate atom order in the following example.

Example 4.2. This program for the n-queens problem shows an application of `block` declarations other than enabling multiple modes: implementing the test-and-generate paradigm. Here `permute` is defined as in Ex. 4.1.

```
nqueens(N,Sol) :-
  sequence(N,Seq),
  safe(Sol),
  permute(Sol,Seq).

:- block sequence(-,?).
sequence(0,[]).
sequence(N,[N|Seq]):-
  0 < N,
  N1 is N-1,
  sequence(N1,Seq).

:- block safe(-).
safe([]).
safe([N|Ns]) :-
  safe_aux(Ns,1,N),
  safe(Ns).
```

```
:- block safe_aux(-,?,?), safe_aux(?,-,?),
                          safe_aux(?,?,-).
safe_aux([],_,_).
safe_aux([M|Ms],Dist,N) :-
  no_diag(N,M,Dist),
  Dist2 is Dist+1,
  safe_aux(Ms,Dist2,N).

:- block no_diag(-,?,?), no_diag(?,-,?).
no_diag(N,M,Dist) :-
  Dist =\= N-M,
  Dist =\= M-N.
```

With the mode $\{\texttt{nqueens}(I,O), \texttt{safe}(I), \texttt{sequence}(I,O), \texttt{permute}(O,I), \texttt{is}(O,I), <(I,I)\}$ and the type $\{\texttt{nqueens}(int,il), \texttt{sequence}(int,il), \texttt{safe}(il), \texttt{permute}(il,il)\}$, the first clause is $\langle 1,3,2\rangle$-nicely moded and $\langle 1,3,2\rangle$-well typed. Moreover, the query `nqueens(4,Sol)` terminates.

However, if in the first clause, the atom order is changed by moving `sequence(N,Seq)` to the end, then `nqueens(4,Sol)` loops. This is because resolving `sequence(4,Seq)` with the second clause for `sequence` makes a (not speculative!) binding which triggers the call `permute(Sol,[4|T])`. This call results in a loop. Note that `[4|T]`, although non-variable, is insufficiently instantiated for `permute(Sol,[4|T])` to be correctly typed in its input position: `permute` is called with *insufficient input*.

To ensure termination, atoms in a clause body that loop when called with insufficient input should be placed so that all atoms which produce the input for these atoms occur textually earlier.

In the following three subsections, we first define *permutation robustly typed*, which is an elementary property a program must have for our method to be applicable. We then identify the *robust* predicates, which terminate for every delay-respecting selection rule. Finally, we show how predicates which are not robust must be placed in clause bodies to ensure termination.

4.1 Preventing Instantiation of Own Input

A prerequisite of our formalism is that no call arising in a derivation can ever instantiate its own input arguments.

Example 4.3. Consider the following version[4] of `delete`(O, I, O).

```
:- block delete(?,-,-).
delete(X,[U|[H|T]],[U|Z]) :-
  delete(X,[H|T],Z).
delete(A,[A|B],B).
```

Consider the query `delete(A,L,R), delete(B,[1,2],L)`. The second atom produces `L`, which is used by the first atom as input. The query loops, since the second atom partially binds `L`, which wakes up the first atom, which then instantiates `L` further (i.e. the call instantiates its own input), resulting in a recursive call to `delete`, and so forth.

To prevent a call from instantiating its input, the `block` declarations must enforce that an atom is *only* selected if all input positions of non-variable type are non-variable. As the previous example shows, this is not enough. It also has to be ensured that each input argument in the clause head is flat (which the clause head `delete(X,[U|[H|T]],[U|Z])` violates). The next example shows that even that is not enough.

Example 4.4. Consider the following program in mode `p`(I, O).

```
:- block p(-,?).
p(g(Y),Y).
```

A call to `p(g(X),3)` instantiates `X` to `3`, and thus instantiates its own input.

The easiest solution seems to be to require that the output positions in a query are always filled by variables. In mode `p`(I, O), the query `p(g(X),3)` should not arise, since its output is already instantiated. This is considered in [2] (*simply-modedness*). However, it is often too restrictive.

Example 4.5. The following is an excerpt from a version of `quicksort`.

```
:- block qs(-,-).
qs([],[]).
qs([X|Xs],Ys) :-
  append(As2,[X|Bs2],Ys),
  partition(Xs,X,As,Bs),
  qs(As,As2),
  qs(Bs,Bs2).
```

For the mode $\{$`qs`(O, I), `append`(O, O, I), `partition`$(O, I, I, I)\}$, the non-variable term `[X|Bs2]` occurs in an output position.

[4] It is part of the *most specific program* [12] corresponding to Ex. 3.1, proposed in [15] to prevent looping for `permute`(O, I). However, it does not work. The query `permute(A,[1])` indeed terminates, but `permute(A,[1,2])` still loops.

In the sequel, we assume that a label *free* or *bound* is associated with each output position of each predicate. Non-variable terms in output positions in a query are allowed only in bound positions. The bound positions must be of non-variable type. As with assigning the mode and the type to a predicate, we do not propose a method of deciding which positions should be free or bound. In the examples however, an output position of a predicate p is bound if and only if there is a clause body with an atom using p, which has a non-variable term in that position.

For notational convenience, we use the notion of free and bound positions also for *input* positions. An input position is free if and only if it is of variable type. We denote the projection of a vector of arguments $\mathbf{r}$ onto its free positions as $\mathbf{r}^{\mathsf{f}}$, and the projection onto its bound positions as $\mathbf{r}^{\mathsf{b}}$.

Definition 4.1 (permutation robustly typed). Let π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \ldots, n\}$. A query $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **π-robustly typed** if it is π-nicely moded and π-well typed, $\mathbf{t}_1^{\mathsf{f}}, \ldots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_1^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat typeable terms.

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-robustly typed** if it is π-nicely moded and π-well typed, and

1. $\mathbf{t}_0^{\mathsf{f}}, \ldots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_0^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat typeable terms.
2. if a position in $\mathbf{s}_{n+1}^{\mathsf{b}}$ of type τ is filled with a variable x, then x also fills a position of type τ in $\mathbf{t}_0^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$.

A **permutation robustly typed** query (clause, program) and a **robustly typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Def. 3.1.

Example 4.6. The `permute`-program of Ex. 4.1, for any of the types in Ex. 3.2, assuming all output positions are free, is robustly typed in mode $\texttt{permute}(O, I)$ and permutation robustly typed in mode $\texttt{permute}(I, O)$.

Consider Ex. 4.5 with the usual definition for the missing clauses, with type $\{\texttt{qs}(il, il), \texttt{append}(il, il, il), \texttt{partition}(il, int, il, il)\}$. This program is permutation robustly typed in mode $\texttt{qs}(O, I)$, assuming the second position of `append` is the only bound output position. It is also permutation robustly typed in mode $\texttt{qs}(I, O)$, assuming that all output positions are free.

Definition 4.2 (input selectability). Let P be a permutation robustly typed program. P has **input selectability** if for every permutation robustly typed query Q, an atom in Q is selectable in P if and only if it is non-variable in all input positions of non-variable type.

Example 4.7. Consider $\texttt{append}(O, O, I)$ where the second position is the only bound output position (Exs. 4.5, 4.6), and the `block` declaration is

```
:- block append(-,?,-), append(?,-,-).
```

This program has input selectability. $Q =$ `append(A,[B|Bs],[1])` is a permutation robustly typed query, and its only atom is selectable. The atom `append([],[],C)` is also selectable, although its input position is variable. This does not contradict Def. 4.2, since this atom cannot be an atom in a permutation robustly typed query with respect to mode `append`(O, O, I).

Looking at Def. 4.1, one is tempted to think that it is best to associate the label *bound* with *all* output positions, because that would make the definition less restrictive. However, we require a program to have input selectability in each of its modes. Since input selectability is defined with respect to atoms in permutation robustly typed queries, and permutation robustly typed queries are defined with respect to given free and bound positions, it turns out that the choice of free and bound positions constrains the possible set of modes. For reasons of space, we cannot explain this in detail. Anyway, we have not encountered a case where a "natural" mode of a program was ruled out.

The following lemma shows a persistence property of permutation robustly typedness, and shows furthermore that a derivation step cannot instantiate the input arguments of the selected atom.

Lemma 4.1. Let P be a permutation robustly typed program with input selectability, $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a permutation robustly typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a clause in P such that $vars(Q) \cap vars(C) = \emptyset$. Suppose $\langle Q, \emptyset\rangle; \langle R, \sigma\rangle$ is a derivation step with clause C and selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$.

Then $R\sigma$ is permutation robustly typed, and $dom(\sigma) \cap vars(\mathbf{s}_k) = \emptyset$ and $vars(\mathbf{s}_k) \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\sigma) = \emptyset$. *(Proof [17])*

4.2 Robust Predicates

In this subsection, derivations are not required to be left-based. Therefore we do not need to consider arbitrary permutations and we can, without loss of generality, assume that the programs and queries are robustly typed (rather than *permutation* robustly typed). This simplifies the notation. In Subsect. 4.3, we go back to allowing for arbitrary permutations.

Definition 4.3 (robust). A predicate p in a robustly typed program P is **robust** if, for each robustly typed query $p(\mathbf{s}, \mathbf{t})$, any delay-respecting derivation of $P \cup \{p(\mathbf{s}, \mathbf{t})\}$ is finite. An atom is **robust** if its predicate is.

This means that for queries consisting of robust atoms, termination does not depend on left-based derivations. Thus the *position* of a robust atom in a clause body or query does not affect termination. The following lemma says that a robust atom cannot proceed indefinitely unless it is repeatedly "fed" by some other atom.

Lemma 4.2. Let P be a robustly typed program with input selectability and F, a, H a robustly typed query where a is a robust atom. A delay-respecting derivation of $P \cup \{F, a, H\}$ can have infinitely many a-steps only if it has infinitely many b-steps, for some $b \in F$. *(Proof [17])*

The following lemma is a simple consequence and states that the robust atoms in a query on their own cannot produce an infinite derivation.

Lemma 4.3. Let P be a robustly typed program with input selectability and Q a robustly typed query. A delay-respecting derivation of $P \cup \{Q\}$ can be infinite only if there are infinitely many steps where a non-robust atom is resolved. *(Proof [17])*

For LD-derivations, termination proofs usually rely on some norm to measure the size of a term or atom [1,5]. For a query F, a, H, the query F is resolved away before a is resolved, and thus a is sufficiently instantiated to be *bounded* with respect to the norm. In contrast, for arbitrary derivations, the decrease in argument size must be independent of the order in which atoms are selected. We assume a simple norm where a term is smaller than another term if it is a proper subterm. This method could be enhanced by considering other norms.

Example 4.8. Consider Ex. 4.2, where all arguments are input, and the type is $\{\texttt{safe}(il),\ \texttt{safe\_aux}(il, int, int),\ \texttt{no\_diag}(int, int, int)\}$. All delay-respecting derivations of a permutation robustly typed query $\texttt{safe\_aux}(l, n, m)$ terminate, because in the first argument of `safe_aux`, there is a strict decrease with respect to the "subterm" norm.

The following definition is adapted from [1].

Definition 4.4 (depends on). Let p, q be predicates in a program P. We say that p **refers to** q if there is a clause in P with p in its head and q in its body, and p **depends on** q (written $p \sqsupseteq q$) if (p, q) is in the reflexive, transitive closure of *refers to*. We write $p \sqsupset q$ if $p \sqsupseteq q$ and $q \not\sqsupseteq p$, and $p \approx q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$.

To show robustness, one has to find argument positions, one for each predicate, such that there is a decrease in argument size in that position.

Definition 4.5 (decreasing clause). Assume that for each predicate p in a program P, there is a designated position called **decreasing position**. Let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a clause in P. Suppose that for each $\mu \in \{1, \ldots, m\}$ where $q_\mu \approx q$, $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)$ has a variable in its decreasing position which is a proper subterm of the term in the decreasing position of $q(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then C is **decreasing**.

To show that a predicate p is robust, we assume that all predicates q with $p \sqsupset q$ have already been shown to be robust.

Lemma 4.4. Let P be a robustly typed program with input selectability and p a predicate in P. Suppose that for each predicate q, where $p \sqsupseteq q$, either:

1. $p \sqsupset q$ and q is robust.
2. $p \approx q$ and each clause defining q is decreasing.

Then p is robust. *(Proof [17])*

Of course, a predicate in a permutation robustly typed program is not always robust, and so the technique given by the above lemma cannot always be applied. Often there is no decreasing position for a predicate.

Example 4.9. We demonstrate for Ex. 4.8 how Lemma 4.4 is used. Given that the built-in `=\=` terminates, it follows that `no_diag` is robust. We show that the second clause for `safe_aux` meets assumption 2 of Lemma 4.4. With the first position of `safe_aux` as decreasing position, the recursive call to `safe_aux` has `Ms` in the decreasing position, which is a proper subterm of `[M|Ms]`. Similar arguments can be applied for the other clauses, showing that `safe` and `safe_aux` are robust.

4.3 Well Fed Programs

So far we have shown for some predicates that all delay-respecting derivations of queries with these predicates terminate. As $\texttt{permute}(O, I)$ shows, this does not work for all predicates. In a program which uses such predicates, the selection rule must be taken into account. We assume left-based derivations. Consequently we now also give up the assumption, made to simplify the notation, that the clauses and query are robustly typed, rather than just *permutation* robustly typed. All statements from the previous subsection generalise in the obvious way.

A query is called *well fed* if each atom has been shown to be robust or occurs in such a position that all atoms which "feed" the atom occur earlier. Of course, since robustness is undecidable, we must assume a predicate to be non-robust if it has not been shown to be robust.

Definition 4.6 (well fed). Let P be a permutation robustly typed program. For a π-robustly typed query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, an atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well fed** if it is robust, or $\pi(j) < \pi(i)$ implies $j < i$ for all j. A π-robustly typed query (clause) is **well fed** if all of its (body) atoms are. P is **well fed** if all of its clauses are well fed and it has input selectability.

Example 4.10. The programs mentioned in Ex. 4.6 are well fed in the given modes. The program in Ex. 4.2 is well fed in the given mode. It is not well fed in mode $\{\texttt{nqueens}(O, I), \texttt{permute}(I, O), \texttt{sequence}(O, I), <(I, I), \texttt{is}(O, I)\}$, because it is not permutation nicely moded in this mode: in the second clause for `sequence`, `N1` occurs twice in an output position.

Lemma 4.5. Every resolvent of a well fed query Q and a well fed clause C, where $vars(Q) \cap vars(C) = \emptyset$, is well fed.

Proof. By obvious analogy, Corollary 3.2 also holds if *nicely moded* is replaced with *robustly typed*. The result then follows from Lemma 4.1. □

The following theorem reduces the problem of showing termination of left-based derivations for well fed programs to showing termination of LD-derivations for the corresponding robustly typed program.

Theorem 4.6. Let P and Q be a well fed program and query, and P' and Q' a robustly typed program and query corresponding to P and Q. If every LD-derivation of $P' \cup \{Q'\}$ is finite, then every left-based derivation of $P \cup \{Q\}$ is finite. *(Proof [17])*

Given that for the programs of Ex. 4.10, the corresponding robustly typed programs terminate for robustly typed queries, it follows from the above theorem that the former programs terminate for well fed queries.

5 Related Work

In using "modedness" and "typedness", we follow **Apt** and **Luitjes** [2], and also adopt their notation. Our results on occur-check freedom and non-floundering are straightforward variations of their results. For termination, they propose a method limited to deterministic programs.

Naish [15] gives excellent intuitive explanations why programs loop, which directed our own search for further ideas and their formalisation. To ensure termination, he proposes some heuristics, without any formal proof.

Predicates are assumed to have a single mode. Naish suggests that alternative modes should be achieved by multiple versions of a predicate.[5] However, under that assumption, why have delay declarations in the first place? For instance, in the mentioned example `permute`, if we only consider `permute`(O, I), then Ex. 4.1 does not loop for the plain reason that no atom ever delays, and thus the program behaves as if there were no delay declarations. In this case, the interpretation that one should "put recursive calls last" is misleading. If we only consider `permute`(I, O), then the version of Ex. 4.1 is much less efficient than Ex. 3.1. In short, the whole discussion on delay declarations makes little sense when only one mode is assumed.

Lüttringhaus-Kappel [10] proposes a method of generating control automatically, and has applied it successfully to many programs. However, rather than pursuing a formalisation of some intuitive understanding of why programs loop, and imposing appropriate restrictions on programs, he attempts a high degree of generality. This has certain disadvantages.

The method only finds *acceptable* delay declarations, ensuring that the most general selectable atoms have finite SLD-trees. What is required however are *safe* delay declarations, ensuring that *instances* of most general selectable atoms have finite SLD-trees. A *safe* program is a program for which every acceptable delay declaration is safe. No hint is given as to how it is shown that a program is safe. This is a missing link.

The delay declarations for some programs such as `quicksort` require an argument to be a nil-terminated list before an atom can be selected. As Lüttringhaus-Kappel points out himself, "in NU-Prolog [*or SICStus*] it is not possible to express such conditions". We have shown here that, with a knowledge of modes and types, `block` declarations are sufficient.

[5] This is also the approach taken in Mercury [18], where these versions are generated by the compiler.

Floundering cannot be ruled out systematically, but only be avoided on a heuristic basis. Thus in principle, the method sometimes enforces termination by floundering. This lies in the nature of the weak assumptions made, and thus is sometimes unavoidable, but there is no way of knowing whether for a particular program, it was unavoidable or not.

Marchiori and **Teusink** [11] base termination on norms and the *covering* relation between subqueries of a query. This is loosely related to well-typedness. However, their results are not comparable to ours because they assume a *local selection rule*, that is a rule which always selects an atom which was introduced in the most recent step. We are not aware of an existing language that uses a local selection rule. The authors state that programs that do not use *speculative bindings* deserve further investigation, and that they expect any method for proving termination with *full* coroutining either to be very complex, or very restrictive in its applications.

6 Discussion and Future Work

We have presented a method of proving termination for programs with `block` declarations. This was both a refinement and a formalisation of the heuristics presented in [15].

We required programs to be *permutation robustly typed*, a property which ensured that no call instantiates its own input. In the next step, we identified when a predicate is *robust*, which means that every delay-respecting derivation for a query using the predicate terminates. Robust atoms could be placed in clause bodies arbitrarily. Non-robust atoms had to be placed such that their input is sufficient when they are called.

The main purpose of this work is software development, and it is envisaged that an implementation should take the form of a program development tool. The programmer would provide mode and type information for the predicates in the program. The tool would then generate the `block` declarations and try to reorder the atoms in clause bodies so that the program is well fed with respect to these modes and types. Finding the free and bound positions, as well as the decreasing position used to prove robustness, should be done by the tool. As already indicated, these choices are very constrained anyway, which suggests that this should be feasible.

In [16] we discuss how to prevent errors related to built-ins, in particular arithmetic built-ins. Another interesting issue is how achieving multiple modes using `block` declarations affects the efficiency of programs.

Acknowledgements

We thank the anonymous referees for their helpful suggestions and comments. Jan-Georg Smaus was supported by EPSRC Grant No. GR/K79635.

References

1. K. R. Apt. *From Logic Programming to Prolog*, chapter 6. Prentice Hall, 1997. 73, 79, 84
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *AMAST'95*, LNCS, Berlin, 1995. Springer Verlag. Invited Lecture. 74, 76, 77, 78, 79, 81, 86
3. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994. 78
4. R. Chadha and D. A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, University of North Carolina, 1991. 77
5. Stefaan Decorte and Danny De Schreye. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of ILPS*, pages 420–436. MIT Press, 1993. 84
6. P. M. Hill, editor. *ALP Newsletter*, `http://www-lp.doc.ic.ac.uk/alp/`, February 1998. Pages 17,18. 73
7. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994. 73
8. Intelligent Systems Laboratory, SICS, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1997. `http://www.sics.se/isl/sicstus/sicstus_toc.html`. 73, 74, 75
9. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. 74
10. S. Lüttringhaus-Kappel. Control generation for logic programs. In D. Warren, editor, *Proceedings of ICLP*, pages 478–495. MIT Press, 1993. 74, 86
11. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. Lloyd, editor, *Proceedings of ILPS*, pages 447–461. MIT Press, 1995. 74, 87
12. K. Marriott, L. Naish, and J. L. Lassez. Most specific logic programs. *Annals of mathematics and artificial intelligence*, 1(2), 1990. Also in proceedings of the Fifth JICSLP. 81
13. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982. 78
14. L. Naish. Automatic control of logic programs. *Journal of Logic Programming*, 2(3):167–183, 1985. 80
15. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992. 73, 74, 79, 81, 86, 87
16. J.-G. Smaus, P. M. Hill, and A. King. Preventing instantiation errors and loops for logic programs with several modes using `block` declarations. In Pierre Flener, editor, *Pre-proceedings of LOPSTR*, number UMCS-98-6-1. University of Manchester, 1998. Extended abstract. 87
17. J.-G. Smaus, P. M. Hill, and A. King. Verification of logic programs with `block` declarations running in several modes. Technical Report 7-98, University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, July 1998. 78, 79, 83, 84, 86
18. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996. 73, 86

The Boolean Logic of Set Sharing Analysis

Michael Codish[1] and Harald Søndergaard[2]

[1] Department of Mathematics and Computer Science
Ben-Gurion University of the Negev, Beer-Sheva, Israel
mcodish@cs.bgu.ac.il
[2] Department of Computer Science, The University of Melbourne
Parkville 3052, Australia
harald@cs.mu.oz.au

Abstract. We show that Jacobs and Langen's domain for set-sharing analysis is isomorphic to the domain of positive Boolean functions, introduced by Marriott and Søndergaard for groundness dependency analysis. Viewing a set-sharing description as a minterm representation of a Boolean function leads to re-casting sharing analysis as an instantiation dependency analysis. The key idea is to view the sets of variables in a sharing domain element as the models of a Boolean function. In this way, sharing sets are precisely dual negated positive Boolean functions. This new view improves our understanding of sharing analysis considerably and opens up new avenues for the efficient implementation of this kind of analysis, for example using ROBDDs. To this end we express Jacobs and Langen's abstract operations for set sharing in logical form.

1 Introduction

Of the abstract domains used in abstract interpretation of logic programs, the two that have received the most attention are Pos and Sharing. The former, originally introduced by Marriott and Søndergaard [15] and more formally presented in [6,16,8] consists of the class of positive Boolean functions. The Pos domain is most commonly applied to the analysis of groundness dependencies for logic programs. The Sharing domain, introduced by Jacobs and Langen [11,13,12], consists of sets of sets of variables and is applied to the analysis of possible sharing amongst sets of program variables. The two abstract domains seem very different in nature. While Pos is considered a clean, intelligible abstract domain, many theoreticians have grappled with the subtleties of Sharing, and implementors have found that analysis based on Sharing is cumbersome and inefficient.

We show that Sharing *is* Pos, in the sense that the two domains are isomorphic, even though the interpretation of the Boolean functions (and hence some of the associated abstract operations) differ from one to the other. This insight helps to de-mystify Sharing, and it has ramifications for set sharing analysis, which may become more amenable to efficient implementation, witness the number of fast Pos-based groundness analyzers that have been built.

The extra complexity of Sharing and its operations is often attributed to the fact that set-sharing is a polyadic property ("it takes more than one to share")

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 89–101, 1998.

while in contrast, Pos is concerned with dependencies related to groundness, a monadic property. The intuition behind our result is to view set-sharing analysis as "dual negative"—a term that will become clear shortly—to groundness analysis. We effectively recast the variable sharing (or variable independence) problem as one of instantiation dependency. We claim that it is not only a practical, but also a conceptual advantage, to view the polyadic "sharing" property in terms of dependencies related to the monadic "instantiatedness" property.

An element of Sharing is a set of variable sets. It conveys sophisticated information about possible sharing amongst program variables. For example, the substitution $\theta = \{w \mapsto c, x \mapsto f(y,z)\}$ is abstracted by the set $S = \{\emptyset, \{x,y\}, \{x,z\}\}$ of variable sets. The set $\{x,y\}$ indicates the (possible) presence of a common variable in the terms that x and y are mapped to by θ. Similarly, for the set $\{x,z\}$. Note that the absence of w in S indicates that w must be ground.

The key idea applied in this paper is to view the sets of variables in an element of the Sharing domain as models of a Boolean function. Because the elements of the Sharing domain always contain the empty set, we are concerned with the class of Boolean functions which are satisfied by assigning the value *false* to all of the variables. We call this class of functions $\overline{\mathsf{coPos}}$, as these are precisely the set of dual negated Pos functions (or equivalently, the negated dual Pos functions). The order isomorphism between Sharing and $\overline{\mathsf{coPos}}$ and hence also between Sharing and Pos is immediate and yet intriguing. Considering the well-known subset Def $\subseteq$ Pos and its dual negated counterpart $\overline{\mathsf{coDef}} \subseteq \overline{\mathsf{coPos}}$, we show that groundness analyses using Def and Sharing coincide, thus extending a result by Cortesi et al. [7].

Before we continue, let us reflect briefly on the relation between the two representations. It emerges that Sharing gives a "truth-table like" representation of the dual negated positive Boolean functions, whereas Pos usually is implemented using much more compact, symbolic representations of Boolean functions, such as ROBDDs [4]. The hope is that such representations may enable simpler and more efficient abstract implementation of the essential abstract operations for Sharing.

2 Preliminaries

We assume familiarity with the standard definitions and notation for logic programs [14] and abstract interpretation [9,10]. We denote the set of all (logic) variables by *Vars* but usually restrict attention to a fixed finite set *VI* of variables of interest. Object language variables are typically chosen from the set $\{s,t,u,v,w,x,y,z\}$, while we use c as an object language constant. For a syntactic object O, $vars(O)$ denotes the set of variables in O. A substitution θ is a mapping from variables to terms which has a finite support. Namely, the set, $dom(\theta) = \{v \in Vars \mid v\theta \neq v\}$ (often referred to as the *domain* of θ) is finite. The identity substitution is denoted ε. We use Sub to denote the set of (idempotent) substitutions.

2.1 Set-Sharing

For a fixed finite set VI of variables of interest, the abstract domain Sharing is defined as

$$\mathsf{Sharing} = \left(\left\{\ S \subseteq \mathcal{P}(VI) \,\middle|\, \emptyset \in S\ \right\},\ \subseteq\right).$$

We say that a set s of variables *occurs* in a substitution θ *through* the variable v, if s is (exactly) the set of variables (of interest) which are mapped to terms containing v. If s occurs in θ through some variable v then we say that s is a set of variables which *share* under θ. Jacobs and Langen [12] provide the following definition:

$$\begin{aligned} &occs : Sub \times VI \to \mathcal{P}(VI) \\ &occs(\theta, v) = \left\{\ x \in VI \,\middle|\, v \in vars(x\theta)\ \right\}. \end{aligned}$$

The abstraction function $\alpha_{sh} : 2^{Sub} \to \mathsf{Sharing}$ for set sharing analysis is defined by

$$\alpha_{sh}(\Theta) = \left\{\ occs(\theta, v) \,\middle|\, \theta \in \Theta,\ v \in Vars\ \right\}.$$

Observe that a variable $x \in VI$ is absent from each set in $\alpha_{sh}(\Theta)$ if and only if every $\theta \in \Theta$ maps x to a ground term. Also note that the presence of $\emptyset$ in every element of Sharing is natural: the empty set can be seen as the contribution from the variables in $Vars \setminus VI$. Namely, because both VI and the domain of a substitution θ are always finite, there will always be a variable $v \notin VI$ for which $\theta v = v$ and hence also $occs(\theta, v) = \emptyset$.

Example 1. Let $VI = \{w, x, y, z\}$, recall that ε denotes the identity substitution and let $\theta_1 = \{x \mapsto [s, t], y \mapsto [t, u, v], z \mapsto [s, t, u, v]\}$ and $\theta_2 = \{w \mapsto c, x \mapsto f(y, z)\}$. Then

$$\begin{aligned} \alpha_{sh}(\{\varepsilon\}) &= \{\emptyset, \{w\}, \{x\}, \{y\}, \{z\}\} \\ \alpha_{sh}(\{\theta_1\}) &= \{\emptyset, \{w\}, \{x, z\}, \{y, z\}, \{x, y, z\}\} \\ \alpha_{sh}(\{\theta_2\}) &= \{\emptyset, \{x, y\}, \{x, z\}\} \\ \alpha_{sh}(\{\theta_1, \theta_2\}) &= \{\emptyset, \{w\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\} \quad \blacksquare \end{aligned}$$

2.2 Boolean Functions

Let $B = \{true, false\}$. A Boolean function (on VI) is a function $f : B^n \to B$. An *interpretation* $\mu : VI \to B$ is an assignment of truth values to the variables in VI. An interpretation μ is a *model* for f, denoted $\mu \models f$, if $f(\mu(x_1), \ldots, \mu(x_n)) = true$. We will often write an interpretation as the set of variables which are assigned the value *true*. The set of models of f is thus viewed as a set of sets of variables defined by:

$$models_{VI}(f) = \left\{\ \{x \mid \mu(x) = true\} \,\middle|\, \mu \models f\ \right\}.$$

Let f be a Boolean function on VI. We say that f is *positive* if $VI \in models_{VI}(f)$, that is, $f(true, \ldots, true) = true$. The abstract domain Pos consists of the positive Boolean functions ordered by logical consequence. We say that a positive Boolean function is *definite*, if its set of models is closed under intersection. We denote the class of definite Boolean functions by Def. A definite function can be written as a conjunction of clauses of the form $a_0 \leftarrow a_1, \ldots, a_n$, and this form is functionally complete for Def, that is, every element in Def can be written this way [1]. The acceptable clauses can also be characterised as the disjunctions $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_n$, that contain exactly one non-negated variable. Figure 1 (a) and (b) illustrates Pos and Def for $VI = \{x, y\}$.

The abstraction function $\alpha_{gr} : 2^{Sub} \rightarrow$ Pos for groundness analysis is defined by

$$\alpha_{gr}(\Theta) = \bigvee_{\theta \in \Theta} \bigwedge_{v \in dom(\theta)} (v \leftrightarrow \bigwedge vars(v\theta)).$$

Example 2. Let $VI = \{w, x, y, z\}$, $\theta_1 = \{x \mapsto c, y \mapsto c, z \mapsto c\}$ and $\theta_2 = \{w \mapsto c, x \mapsto f(y, z)\}$. Then

$$\begin{aligned} \alpha_{gr}(\{\varepsilon\}) &= true \\ \alpha_{gr}(\{\theta_1\}) &= x \wedge y \wedge z \\ \alpha_{gr}(\{\theta_2\}) &= w \wedge (x \leftrightarrow (y \wedge z)) \\ \alpha_{gr}(\{\theta_1, \theta_2\}) &= (x \vee w) \wedge (x \leftrightarrow (y \wedge z)) \end{aligned}$$ ■

We say that f is *negative* if and only if $f(true, \ldots, true) = false$, that is, if and only if f is not positive. Let Pos and $\overline{\text{Pos}}$ denote the classes of positive and negative Boolean functions respectively.

The dual of a Boolean function is the function that results when the roles of *false* and *true* are interchanged. Given a propositional formula, a formula for the dual function is easily obtained: interchange, wherever they occur, the constants *false* and *true*, the connectives $\vee$ and $\wedge$, the connectives $\leftrightarrow$ and $\not\leftrightarrow$ (exclusive or), $\rightarrow$ and $\not\leftarrow$, $\leftarrow$ and $\not\rightarrow$. Negation is self-dual, so $\neg$ is left unchanged, as are variables. It follows that existential quantification ($\exists$) and universal quantification ($\forall$) over propositional variables are dual. Table 1 summarizes the truth tables for the binary propositional connectives and their duals.

For any Boolean function φ we denote by $coneg(\varphi)$ the dual of the negation of φ (or, equivalently, the negation of its dual). For a set of Boolean functions Ψ

x	y	$x \wedge y$	$x \vee y$	$x \leftrightarrow y$	$x \not\leftrightarrow y$	$x \rightarrow y$	$x \not\leftarrow y$	$x \leftarrow y$	$x \not\rightarrow y$
false	*false*	*false*	*false*	*true*	*false*	*true*	*false*	*true*	*false*
false	*true*	*false*	*true*	*false*	*true*	*true*	*true*	*false*	*false*
true	*false*	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*
true	*true*	*true*	*true*	*true*	*false*	*true*	*false*	*true*	*false*

Table 1. Truth tables for binary propositional connectives and their duals

we let $coneg(\Psi)$ denote the set of dual negated functions from Ψ. We let $\overline{\mathsf{coPos}} = coneg(\mathsf{Pos})$ denote the dual negative functions.

It is easy to see that $\overline{\mathsf{coPos}}$ shares many properties with Pos. For example, $\overline{\mathsf{coPos}}$ is closed under conjunction, disjunction and existential quantification, but not under negation.

We are interested in one additional class of Boolean functions, the dual negated Def functions. By duality, these are the $\overline{\mathsf{coPos}}$ functions whose models are closed under union. It is not difficult to see that a function in $\overline{\mathsf{coDef}} = coneg(\mathsf{Def})$ can be written as a conjunction of clauses of the form $a_0 \rightarrow a_1, \ldots, a_n$. In other words, a clause in $\overline{\mathsf{coDef}}$ is a disjunction with exactly one negated variable. Figure 1 (c) and (d) illustrates $\overline{\mathsf{coPos}}$ and $\overline{\mathsf{coDef}}$ for $VI = \{x, y\}$.

Clearly, *coneg* is an order isomorphism between Pos and $\overline{\mathsf{coPos}}$ and between Def and $\overline{\mathsf{coDef}}$. Observe that *coneg* is its own inverse.

3 The Logic of Sharing

Jacobs and Langen [11] prove that Sharing enjoys a Galois insertion into the domain of concrete substitutions. It is easy to see that Sharing and $\overline{\mathsf{coPos}}$ are order isomorphic. The isomorphism is given by the function $ss : \overline{\mathsf{coPos}} \rightarrow \mathsf{Sharing}$ defined by

$$ss(\varphi) = \{s \subseteq VI \mid s \models \varphi\}.$$

Conversely, we can view a set of variable sets as a propositional formula in minterm form [3]. For example, for $VI = \{w, x, y, z\}$, the set $\{\emptyset, \{x, y, z\}\}$ is read as

$$(\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee (\neg w \wedge x \wedge y \wedge z) \quad = \quad \neg w \wedge (x \leftrightarrow y) \wedge (y \leftrightarrow z).$$

This function can further be mapped into Pos by the function *coneg*, yielding $w \wedge (x \leftrightarrow y) \wedge (y \leftrightarrow z)$. The domain orderings, set (model) containment and logical consequence, clearly correspond.

Note that it is natural, and common, to also allow $\emptyset$ as an additional, least, element of Sharing, and to allow *false* as an element of $\overline{\mathsf{coPos}}$ and of Pos. In either case the element corrsponds to the empty set of substitutions. The isomorphism is easily extended so that these elements correspond.

If we consider just Def and $\overline{\mathsf{coDef}}$, *coneg* is in fact homomorphic with respect to all the interesting (abstract) operations, including $\wedge$, $\vee$, and $\exists$. We state this as Theorem 1 below. It was already known that Sharing properly contains Def [7,5][1].

Casting Sharing in terms of Boolean functions makes this immediate, and we easily extend that result to the following corollary of Theorem 1: A sharing analysis which uses $\overline{\mathsf{coDef}}$ rather than full Sharing is exactly equivalent to a groundness analysis that uses Def.

[1] Cortesi et al. [7] showed that the groundness component of Sharing was equivalent to the domain Cov which they defined syntactically, and which in hindsight is exactly Def. Cortesi et al. [5] gave a precise mapping from Sharing to Def, by which the exact groundness information that is present in a Sharing element is made clear.

3.1 The Relation between Sharing and Def

To understand the relation between Sharing and $\overline{\mathsf{coDef}}$ (and hence also Def), consider those elements of Sharing that are closed under set union.

Definition 1. *A Boolean function φ is* up-closed *if $models(\varphi)$ is closed under union. We denote by $\varphi\uparrow$ the smallest logical consequence of φ which is up-closed. Similarly, φ is* down-closed *if $models(\varphi)$ is closed under intersection. We denote by $\varphi\downarrow$ the smallest logical consequence of φ which is down-closed.* ■

In other words, $models(\varphi\uparrow)$ is the smallest set that contains $models(\varphi)$ and is closed under union. It is clear that $\varphi\uparrow$ is well-defined, as the function *true* is both up-closed and a logical consequence of φ, so $\varphi\uparrow$ can be characterised as the conjunction of all up-closed logical consequences of φ. Similarly, $\varphi\downarrow$ is well-defined.

The abstract domain Def consists of the down-closed positive Boolean functions. It follows that Def is not closed under disjunction. Armstrong et al. [1] discuss the abstract operations for Def and show how the clausal form of the join $(\varphi \vee \psi)\downarrow$ can be calculated. We give a similar result for $\overline{\mathsf{coDef}}$ in Section 3.2.

Example 3. Let $VI = \{w, x, y, z\}$. The formula $\neg w \wedge (x \leftrightarrow (y \vee z))$ is closed under union of models:

$$models(\neg w \wedge (x \leftrightarrow (y \vee z))) = \big\{\, \emptyset, \{x, y\}, \{x, z\}, \{x, y, z\} \,\big\}\,. \quad \blacksquare$$

It is not hard to see that the operation $\uparrow$ is an upper closure operator on $\overline{\mathsf{coPos}}$ and that it thus induces a Galois insertion between $\overline{\mathsf{coPos}}$ and $\overline{\mathsf{coDef}}$.

Definition 2. *We say that $\sigma \in \overline{\mathsf{coPos}}$ and $\gamma \in Pos$* correspond *if $\sigma = coneg(\gamma)$, or equivalently $\gamma = coneg(\sigma)$, and write this $\sigma \sim \gamma$.* ■

The Pos formula that best describes the unification $x = t$ (where x is a variable and t a term) is

$$\gamma = x \leftrightarrow \bigwedge vars(t).$$

The sharing formula which best describes the same unification is

$$\sigma = x \leftrightarrow \bigvee vars(t),$$

and it is easily checked that $\sigma \sim \gamma$. We can read this as saying that x can be instantiated if some of the variables in t can, and vice versa.

The description of a computation's "initial state", that is, the identity substitution ε, can be considered an abstract operation *init*. This operation may depend on VI. The Pos formula which best describes the identity substitution is then $init_{gr}(VI) = true$, while the corresponding Sharing description consists of the empty set together with all of the singleton sets over VI. So, for $VI = \{x_1, \ldots, x_n\}$, we have the Sharing description

$$\big\{\, \emptyset, \{x_1\}, \ldots, \{x_n\} \,\big\}\,.$$

The corresponding $\overline{\mathsf{coPos}}$ formula is

$$init_{sh}(VI) = \bigwedge_{x \in VI} (x \rightarrow \bigwedge_{y \in VI \setminus \{x\}} \neg y)$$

which specifies that if one of the variables is true in a model then all of the others must be false. In terms of instantiatedness dependencies this means that further instantiation of any *one* variable cannot effect the other variables.

Note that $init_{sh}(VI)$ is not up-closed. It is easily checked that $init_{sh}(VI)\uparrow = init_{gr}(VI)$. One of the main reasons why a sharing analysis using $\overline{\mathsf{coDef}}$ would lose precision compared to one based on $\overline{\mathsf{coPos}}$ is exactly the inability to capture the initial state $init(VI)$ accurately.

Theorem 1. *Let $\sigma, \sigma' \in \overline{\mathsf{coDef}}$ and $\gamma, \gamma' \in \mathsf{Def}$ be such that $\sigma \sim \gamma$ and $\sigma' \sim \gamma'$. Then:* **(1)** $(\sigma \wedge \sigma') \sim (\gamma \wedge \gamma')$, **(2)** $(\sigma \vee \sigma') \sim (\gamma \vee \gamma')$, **(3)** $(\exists v : \sigma) \sim (\exists v : \gamma)$, *and* **(4)** $init_{sh}(VI)\uparrow \sim init_{gr}(VI)\downarrow$.

Proof. **(1)** Negating $(\gamma \wedge \gamma')$ yields $(\neg\gamma \vee \neg\gamma')$. Dualising yields $(coneg(\gamma) \wedge coneg(\gamma'))$, that is, $(\sigma \wedge \sigma')$. **(2)** Similar. **(3)** Negating $\exists v : \gamma$ yields $\forall v : \neg\gamma$. Dualising this we get $\exists v : coneg(\gamma)$, that is, $\exists v : \sigma$. **(4)** Both sides are *true*. ■

It follows that a sharing analysis using $\overline{\mathsf{coDef}}$ is exactly identical to a groundness analysis using Def. The abstract operations are the ones covered in Theorem 1, except that the join in $\overline{\mathsf{coDef}}$ is given by $\sigma \sqcup \sigma' = (\sigma \vee \sigma')\uparrow$ while the join in Def is given by $\gamma \sqcup \gamma' = (\gamma \vee \gamma')\downarrow$.

It is, however, clear that also these joins preserve correspondence, since $\sigma\uparrow \sim \gamma\downarrow$ if $\sigma \sim \gamma$. This can be seen by considering the models of σ and γ. If the set of models for σ is M then the set of models for $\gamma = coneg(\sigma)$ is $M' = \{\overline{\mu} \mid \mu \in M\}$, where $\overline{\mu}$ denotes the complement of μ. Hence closing M under union and M' under intersection preserves the $\sim$ relationship.

3.2 Finding the Best Up-Closed Approximation

Definition 3. *Let φ be a propositional formula. We denote by $m(\varphi)$ the formula obtained by replacing each negated variable in the minterm representation of φ by true.* ■

Note that m is an upper closure operator and that $m(\varphi)$ is the smallest monotonic logical consequence of φ.

Example 4. $m((x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z)) = (x \wedge y) \vee (x \wedge z)$. ■

Theorem 2. *Let φ be a $\overline{\mathsf{coPos}}$ function for a set VI of variables of interest. Then*

$$\varphi\uparrow = \bigwedge_{x \in VI} (x \rightarrow m(\varphi \wedge x)) \,. \quad ■$$

Before we prove this, let us give some examples.

Example 5. Let $VI = \{w, x, y, z\}$ and $\varphi = \{\emptyset, \{x\}, \{w, x\}, \{x, y\}, \{x, y, z\}\}$ be a $\overline{\mathsf{coPos}}$ function given by its set of models. Then

$$\begin{aligned}\varphi{\uparrow} = {} & (w \rightarrow (w \wedge x)) \wedge \\ & (x \rightarrow x \vee (x \wedge w) \vee (x \wedge y) \vee (x \wedge y \wedge z)) \wedge \\ & (y \rightarrow (x \wedge y) \vee (x \wedge y \wedge z)) \wedge \\ & (z \rightarrow (x \wedge y \wedge z))\end{aligned}$$

We can write $\varphi{\uparrow}$ more simply as $(w \rightarrow x) \wedge (y \rightarrow x) \wedge (z \rightarrow y)$. That is, if w or y are instantiated, so is x, and if z is instantiated, so is y (and hence x). Note that

$$models(\varphi{\uparrow}) = \big\{\, \emptyset, \{x\}, \{w, x\}, \{x, y\}, \{x, y, z\}, \{w, x, y\}, \{w, x, y, z\} \,\big\}\,. \quad \blacksquare$$

Example 6. Let $VI = \{w, x, y, z\}$ and $\varphi = \{\emptyset, \{x, y\}, \{y, z\}\}$ be a $\overline{\mathsf{coPos}}$ function given by its sets of models. Then

$$\begin{aligned}\varphi{\uparrow} = {} & (w \rightarrow \mathit{false}) \quad \wedge \quad (x \rightarrow (x \wedge y)) \quad \wedge \\ & (y \rightarrow ((x \wedge y) \vee (y \wedge z))) \quad \wedge \quad (z \rightarrow (y \wedge z)).\end{aligned}$$

We can write $\varphi{\uparrow}$ more simply as $\neg w \wedge (y \leftrightarrow (x \vee z))$. Note that

$$models(\varphi{\uparrow}) = \big\{\, \emptyset, \{x, y\}, \{y, z\}, \{x, y, z\} \,\big\}\,. \quad \blacksquare$$

Example 7. Let $VI = \{w, x, y, z\}$ and $\varphi = \{\emptyset, \{w\}, \{x\}, \{y, z\}\}$ be a $\overline{\mathsf{coPos}}$ function given by its sets of models. Then

$$\varphi{\uparrow} = (w \rightarrow w) \wedge (x \rightarrow x) \wedge (y \rightarrow (y \wedge z)) \wedge (z \rightarrow (y \wedge z)).$$

We can write $\varphi{\uparrow}$ more simply as $y \leftrightarrow z$. Note that

$$models(\varphi{\uparrow}) = \big\{\, \emptyset, \{w\}, \{x\}, \{y, z\}, \{w, x\}, \{w, y, z\}, \{x, y, z\}, \{w, x, y, z\} \,\big\}\,. \quad \blacksquare$$

Proof. (Of Theorem 2.) Let φ be a $\overline{\mathsf{coPos}}$ function and let

$$\psi = \bigwedge_{x \in VI} (x \rightarrow m(\varphi \wedge x))\,.$$

We first show that if M is a finite union of models of φ then M is a model of Ψ. Assume, without loss of generality, that M is the union of two models μ_1 and μ_2 of φ. Note that for $x \in \mu_1$, $m(\varphi \wedge x) = (\bigwedge \mu_1) \vee \sigma$ (for some σ) and likewise for $y \in \mu_2$, $m(\varphi \wedge y) = (\bigwedge \mu_2) \vee \sigma'$ (for some σ'). So for each $x \in \mu_1$, Ψ contains a conjunct of the form $x \rightarrow (\bigwedge \mu_1) \vee \sigma$ and for each $y \in \mu_2$ a conjunct of the form $y \rightarrow (\bigwedge \mu_2) \vee \sigma'$. The interpretation $\mu_1 \cup \mu_2$ is a model of Ψ because it satisfies each of the above implications; and because all of the other implications in Ψ are satisfied vacuously, as their left-hand sides are false.

Now we prove that Ψ is minimal. Namely, every model of Ψ is the union of some models of φ. Assume, for contradiction, that μ is a model of Ψ which is not

the union of some models of φ. Then there must be a variable $x \in \mu$ for which no model of φ which includes x is contained in μ. Let $\mu_1, \ldots, \mu_k$ be the models of φ which include x. By construction, Ψ contains a conjunct of the form

$$x \rightarrow \bigvee_{1 \leq i \leq k} \bigwedge \mu_i.$$

Namely, if x is in any model of Ψ then so are all of the other elements of one of the models $\mu_1, \ldots, \mu_k$. This contradicts the assumption that $x \in \mu$. ■

4 Full Sharing and Its Abstract Operations

We now turn to the abstract operations on Sharing. Our aim is to give these in a logical form, by considering sharing information to be expressed in $\overline{\text{coPos}}$. We need to define the abstract operations *init*, *projection* (or *variable elimination*), *join* and *aunify* (for *abstract unification*).

We have already handled *init* in Section 3.1, and it is straightforward to show that projection corresponds to existential quantification and that the join corresponds to disjunction. Abstract unification is (as expected) the only tricky operation.

Jacobs and Langen define abstract unification using the following auxiliary functions.

- The *up-closure* of $S \in \mathsf{Sharing}$, denoted S^* is the smallest superset of S such that $A \in S^* \wedge B \in S^* \rightarrow A \cup B \in S^*$.
- The set of components of $S \in \mathsf{Sharing}$ which are *relevant* to a set of variables V is defined as $rel(S, V) = \{\ A \in S \mid A \cap V \neq \emptyset\ \}$.
- The cross union of the sets of sets of variables $S_1, S_2 \subseteq \mathcal{P}(VI)$ is defined as $S_1 \uplus S_2 = \{\ A \cup B \mid A \in S_1,\ B \in S_2\ \}$.

Let $S \in \mathsf{Sharing}$ be given, together with a program constraint $x = t$ (where x is a variable and t a term), and let $A = rel(S, \{x\})$ and $B = rel(S, vars(t))$. Then the abstract unification operation is defined by

$$aunify(S, x, t) = (S \setminus (A \cup B)) \cup (A \uplus B)^*.$$

Abstract unification for a pair of atoms a and b in a sharing element S is then defined inductively on the primitive constraints in $mgu(a, b)$.

For the logical formulation of *aunify*, let $\varphi \in \overline{\text{coPos}}$ be the function corresponding to S, namely $models(\varphi) = S$, and let d be the disjunction of variables in $vars(t)$, that is, $d = \bigvee vars(t)$. Further, let $d' = x \vee d$. The result of *aunify* is then

$$(x \leftrightarrow d) \wedge ((\varphi \wedge \neg d') \vee (\varphi \wedge d')\uparrow).$$

The intuition is this: The $x \leftrightarrow d$ is the contribution from the equation $x = t$. There is also an impact on the current abstract substitution φ, and to determine this impact, we proceed by cases: **(1)** If x is (or becomes) ground, then all of

the variables in t also become ground, and so the effect on φ is described by $\varphi \wedge \neg d'$; **(2)** Alternatively, there are possibly non-ground variables in t. Since sharing is possible amongst any set of variables that are not ground, the total effect is not $\varphi \wedge d'$, but its up-closure, $(\varphi \wedge d')\uparrow$.

Conjecture 4.1 With the notation introduced above, if $S = models(\varphi)$ then

$$aunify(S, x, t) = models((x \leftrightarrow d) \wedge ((\varphi \wedge \neg d') \vee (\varphi \wedge d')\uparrow)). \quad \blacksquare$$

Example 8. Let $VI = \{x, y, z\}$ and consider the analysis of $x = f(y, z)$. The initial approximation $init_{sh}(VI)$ (in minterm form) is

$$\varphi = (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z).$$

In the notation from above, $\varphi \wedge d' = (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z)$, and the up-closure of this is $x \vee y \vee z$. Taking the disjunction with $(\varphi \wedge \neg d')$ yields *true*. Hence the result is simply $x \leftrightarrow (y \vee z)$. $\blacksquare$

Example 9. Again let $VI = \{x, y, z\}$ but now consider the analysis of $x = y$, $y = z$. First consider $x = y$. In the notation from above, $d' = x \vee y$, and so $\varphi \wedge d' = (x \not\leftrightarrow y) \wedge \neg z$. The up-closure of this is $(x \vee y) \wedge \neg z$.

We also have $\varphi \wedge \neg d' = \neg x \wedge \neg y$, and taking the disjunction with the above, we get $(\neg x \wedge \neg y) \vee \neg z$. Conjoining this with $x \leftrightarrow y$ we arrive at

$$\varphi' = (\neg x \wedge \neg y) \vee (x \wedge y \wedge \neg z)$$

as the $\overline{\mathsf{coPos}}$ description that applies at the point between the two constraints. Notice that the result differs significantly from what a groundness analysis yields here, namely $x \leftrightarrow y$.

From this point let d' denote instead $y \vee z$. Now $\varphi' \wedge d' = (\neg x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z)$, the up-closure of which is $(\neg x \wedge \neg y \wedge z) \vee (x \wedge y)$. Taking the disjunction with $\neg d'$ yields $x \leftrightarrow y$.

This leads to the result after both constraints: $(x \leftrightarrow y) \wedge (y \leftrightarrow z)$. $\blacksquare$

Notice that the processing of a unification such as $x = f(y, z)$ yields a result which, as far as x, y, and z are concerned, is up-closed, in this case $x \leftrightarrow (y \vee z)$, with models $\{\emptyset, \{x, y\}, \{x, z\}, \{x, y, z\}\}$, rather than $\{\emptyset, \{x, y\}, \{x, z\}\}$. This up-closed result is easily seen to be the correct result when we consider that some of the three variables could well have been bound to non-linear terms before the unification.

Also notice that if we were to perform the sharing analysis in the domain $\overline{\mathsf{coDef}}$ instead of full $\overline{\mathsf{coPos}}$, then projection and join would remain unchanged, while abstract unification becomes exactly the same as that for a groundness analysis using Def: Since $\varphi \in \overline{\mathsf{coDef}}$ is up-closed, so is $\varphi \wedge d'$, and hence

$$aunify(\varphi, x, t) \quad = \quad (x \leftrightarrow d) \wedge ((\varphi \wedge \neg d') \vee (\varphi \wedge d')) \quad = \quad (x \leftrightarrow d) \wedge \varphi.$$

5 Conclusion

A considerable degree of imprecision is present in a sharing analysis that uses Sharing because the linearity of variables is not tracked. For this reason the domains Asub [17] and Esharing [13] incorporate linearity information with sharing information, but Sharing is, as far as the handling of unification goes, no better than a groundness analysis using the domain Def, sometimes used for groundness analysis. In fact, if we use Def's natural counterpart $\overline{\text{coDef}}$ for sharing analysis then sharing and groundness analysis *coincide*. This result extends the observation by Cortesi et al. [7] that Sharing properly subsumes Def.

We have found that Sharing and Pos are surprisingly similar. In fact, Sharing is nothing but a "truth-table like" representation of the dual negated positive Boolean functions, $\overline{\text{coPos}}$. Viewing Sharing as a class of Boolean functions has advantages, not only because it helps our understanding of sharing analysis but also because it points towards efficient implementation of set sharing analysis, using for example ROBDDs. To this end we have recast Jacobs and Langen's abstract operations for Sharing in logical form and shown the equivalence of the formulations. We believe that this will lead to greatly improved set sharing analysis for logic programs.

It remains to be seen whether the logic-based approach suggested here can lead to faster implementations of sharing analysis. Bagnara, Hill and Zaffanella [2] have shown that if the ultimate goal of a sharing analysis is to determine *pair-sharing*, that is, whether a given pair of variables are guaranteed to be independent, then a set-sharing analysis can be simplified considerably. It remains to be investigated whether this result can help also in a logic-based implementation.

Acknowledgements

The ideas in this paper were developed when Michael Codish visited Melbourne in early 1998, partly supported by The University of Melbourne. We would like to thank the anonymous referees for helpful suggestions and Roberto Bagnara, Andy King, Grigory Mashevitzky, Lee Naish, Peter Schachte, Peter Stuckey and Enea Zaffanella for stimulating discussions.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998. 92, 94
2. R. Bagnara, P. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In P. Van Hentenryck, editor, *Static Analysis: Proc. Fourth Int. Symp.*, volume 1302 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 1997. 99
3. F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publ., 1990. 93

4. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. 90
5. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. In A. Mycroft, editor, *Static Analysis: Proc. Second Int. Symp.*, volume 983 of *Lecture Notes in Computer Science*, pages 100–117. Springer, 1995. 93
6. A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 322–327, Amsterdam, The Netherlands, July 15–18 1991. IEEE Computer Society Press. 89
7. A. Cortesi, G. Filé, and W. Winsborough. Comparison of abstract interpretations. In W. Kuich, editor, *Automata, Languages and Programming: Proc. Nineteenth Int. Coll.*, volume 623 of *Lecture Notes in Computer Science*, pages 521–532. Springer-Verlag, 1992. 90, 93, 99
8. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, 1996. 89
9. 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. 90
10. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. 90
11. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proc. North American Conf. 1989*, pages 154–165. MIT Press, 1989. 89, 93
12. D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992. 89, 91
13. A. Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs*. PhD thesis, University of Southern California, California, 1991. 89, 99
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987. 90
15. K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. Tutorial Notes for 1989 North American Conf. Logic Programming. Assoc. for Logic Programming, 1989. 89
16. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993. 89
17. 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. 99

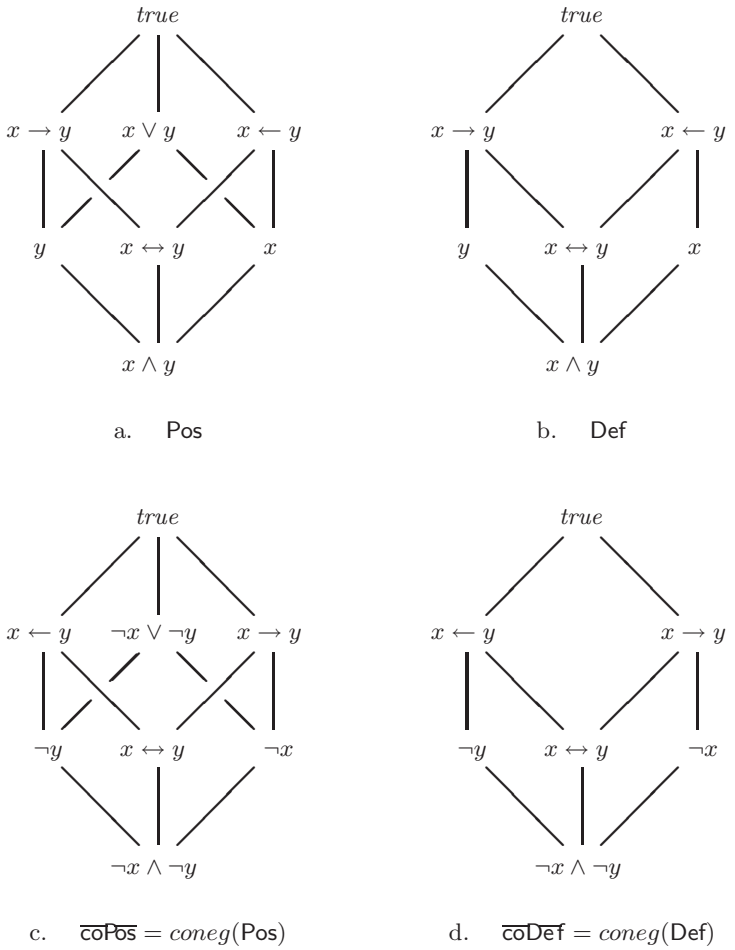


Fig. 1. Classes of Boolean functions for $VI = \{x, y\}$. (a) Pos: the positive functions. (b) Def: the positive functions with models closed under intersection. (c) $\overline{\mathsf{coPos}}$: the dual negated positive functions. The models of these functions give corresponding elements of the Sharing domain. For example $\neg y \equiv \{\emptyset, \{x\}\}$ and $x \leftarrow y \equiv \{\emptyset, \{x\}, \{x, y\}\}$. (d) $\overline{\mathsf{coDef}}$: the dual negated definite functions. The models of these functions are closed under union.

Derivation of Proof Methods by Abstract Interpretation

Giorgio Levi and Paolo Volpe

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
Tel: +39-50-887248, Fax: +39-50-887226
{levi,volpep}@di.unipi.it

Abstract. We study the application of abstract interpretation to the design of inductive methods for verifying properties of logic programs. We give a unified view of inductive assertion-based proof methods for logic programs, by systematically deriving them in a uniform way using Abstract Interpretation. The resulting verification framework allows us to reconstruct several existing verification methods and to understand the relation among them in terms of abstractions. Moreover, we can tackle the problem of establishing the completeness of the proof methods.

Keywords Inductive proof methods, abstract interpretation, abstract domains.

1 Introduction

Abstract interpretation theory [17,18] is successfully used to reason about the relation among different semantics and to statically analyze programs. More recently it has been used in the framework of debugging, diagnosis and validation [6,13,12,8] and to derive proof methods [16]. In this paper we use abstract interpretation as a tool for systematically deriving sufficient partial correctness conditions for logic programs.
In fact, the problem of verification of logic programs can be very naturally approached by abstract interpretation, since it is a typical semantics-based task. The specifications used in verification can simply be viewed as a suitable intended abstract semantics. The existing notions of correctness and related verification methods [23,5,21,3,1] can then be explained in terms of different abstractions.
Here we can use the first important feature of abstract interpretation, namely the ability to compare different semantics by reasoning in terms of abstraction. In the case of logic programs verification, this makes easier to compare the different techniques and to show the essential differences. Two examples of controversial issues which can be better understood by adopting the abstract interpretation approach are

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 102–117, 1998.

- the problem of the relation between declarative and operational properties (and the complexity of the related verification methods), recently raised by Drabent [22];

- the intrinsic ambiguity of specifications given in terms of pre- and post-conditions, for which we can have two different notions of correctness (see I/O correctness and call correctness of Sections (4) and (5)).

The second feature of abstract interpretation is its ability to systematically derive the (optimal) abstract semantics from the concrete one and from the abstraction. The resulting abstract semantics is a correct approximation of the concrete semantics by construction and no additional "correctness" theorems need to be proved. The above nice features are inherited by verification techniques, if the problem is approached by abstract interpretation. Namely, we can define a verification framework, parametric with respect to the (abstract) property we want to model. Given a specific property (abstraction), the corresponding verification conditions are systematically derived from the framework and guaranteed to be indeed sufficient partial correctness conditions. In addition the verification method is guaranteed to be complete, if the abstraction is precise (complete according to abstract interpretation theory).
Properties we deal with in this paper are abstractions of SLD-trees. The verification framework is based on a hierarchy of semantics [14,11,10], whose collecting semantics [14] is defined in a concrete domain of SLD-derivations. A generic (abstract) semantics in the hierarchy, corresponding to the asbtraction α, is the least fixpoint of an operator T_P^α, systematically derived from the corresponding operator of the collecting semantics. In Section (2), we show the general results about sufficient partial correctness conditions and completeness of the verification methods. In the next Sections, we show some instances of the verification framework, with the aim of reconstructing existing verification methods. We first consider the case where specifications are given in an extensional way and are simply intended abstract semantics. We start with more abstract properties (and corresponding weaker verification methods), moving from the success behaviour (Section (3)), to the Input-Output behaviour (Section (4)) and to the call behaviour (Section (5)). In particular we reconstruct the Drabent and Maluszynski method [23], the Bossi and Cocco method [5], and various verification methods reviewed in [3]. Finally in Section (6), we consider the case of intensional specifications given in a formal specification language. Intensional specifications lead to a further layer of abstraction, whose properties (decidability, precision) strongly depend on the specification language. In the case of success behaviour we reconstruct the results by Clark [9] and Deransart [21].

Throughout the paper, we assume familiarity with the standard notions of lattice theory [4], abstract interpretation [17,18], logic programming [2] and verification methods [1,15].

2 Description of the Framework

We assume as a concrete semantics the denotational semantics of [14]. The domain is given by the set of *pure collections*, $\mathbb{D} = (Pred \rightarrow \widetilde{\wp}(\mathcal{SLD}), \leq)$, where a pure collection is a function from the set $Pred = \{p(\mathbf{x}) \mid p$ predicate symbol and $\mathbf{x}$ a tuple of distinct variables$\}$ of *pure atoms* to $\widetilde{\wp}(\mathcal{SLD})$, that is the set of prefix-closed sets of finite *SLD*-derivations. The order $\leq$ is the pointwise extension of the subset order of $\widetilde{\wp}(\mathcal{SLD})$.
We use the following functions (for a formal definition we refer to [14]). Let $\mathcal{SLD}$ be the set of *SLD*-derivations. Given $\delta \in \mathcal{SLD}$, $first(\delta)$ and $last(\delta)$ are respectively the first goal and the last goal of δ; $clauses(\delta)$ is the set of clauses used in δ; $\partial_\theta(\delta)$ is the derivation obtained by instantiating the first goal with the substitution θ and applying the same clauses as in δ as far as possible; $res(\delta)$ is the computed answer substitution of the successful derivation δ.
Given a program P, we consider the associated semantic function $T_P : \mathbb{D} \rightarrow \mathbb{D}$ (also defined in [14], where it is called $\mathcal{P}[\![P]\!]$). T_P is continuous on $\mathbb{D}$ and the semantics $[\![P]\!]$ of P is defined as $lfp(T_P) = T_P^\omega = \lambda p(\mathbf{x}).\{\delta \in \mathcal{SLD} \mid first(\delta) = p(\mathbf{x}), clauses(\delta) \subseteq P\}$.

The main idea of this paper is to view (extensional or intensional) specifications as an abstract domain $(\mathcal{A}, \sqsubseteq)$. Under reasonable hypotheses, this relationship can be formalized through a Galois connection between $(\mathbb{D}, \leq)$ and $(\mathcal{A}, \sqsubseteq)$, with $\alpha : \mathbb{D} \rightarrow \mathcal{A}$ and $\gamma : \mathcal{A} \rightarrow \mathbb{D}$ respectively the abstraction and the concretization functions. Then each element D of $\mathbb{D}$ is *correct* with respect to specifications $\mathcal{S}$ such that $\alpha(D) \sqsubseteq \mathcal{S}$. In this paper, (abstract) domains of specifications will always have the form $(Pred \rightarrow \mathbb{S}, \sqsubseteq)$. Each one of its element Ψ can be thought of as a set $\{\Psi_p\}_{p \in Pred}$, where each Ψ_p is a specification for what we observe in the set of derivations started by predicate p. In fact the generic correctness condition can be expressed as

$$\forall \delta \in [\![P]\!](p(x)) : \ \delta \text{ verifies } \Psi_p,$$

which can be rewritten, for a suitable *observable* α [11], as

$$\forall \delta \in [\![P]\!](p(x)) : \ \alpha(\delta) \sqsubseteq \Psi_p. \tag{1}$$

This point of view allows us to study the verification problem using methods and results of abstract interpretation. In fact the problem of checking whether a program P verifies a specification $\mathcal{S}$ in $\mathcal{A}$, can be rephrased in abstract interpretation terms as

$$[\![P]\!] \leq \gamma(\mathcal{S}) \quad \text{or, equivalently, } \alpha([\![P]\!]) \sqsubseteq \mathcal{S}. \tag{2}$$

Indeed, in our case, condition (1) is equivalent to $\alpha([\![P]\!](p(x))) \sqsubseteq \Psi_p$, for each p, that is to $\alpha([\![P]\!]) \sqsubseteq \Psi$. Since $[\![P]\!]$ is defined as the least fixpoint of the operator T_P, a sufficient condition for (2) to hold is

$$T_P(\gamma(\mathcal{S})) \leq \gamma(\mathcal{S}) \quad \text{or, equivalently, } T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}, \tag{3}$$

where $T_P^\alpha = \alpha \circ T_P \circ \gamma$, is the *best abstraction* of T_P in $\mathcal{A}$. In fact $T_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$ implies $lfp(T_P^\alpha) \sqsubseteq \mathcal{S}$ and, since $\alpha(lfp(T_P)) \sqsubseteq lfp(T_P^\alpha)$ (because α is correct), the condition $\alpha([\![P]\!]) \sqsubseteq \mathcal{S}$ can be derived.

Condition (3) can often be unfolded and transformed into a verification condition for P and $\mathcal{S}$, which may be the base for an inductive proof method. Obviously how to do it depends on the abstract domain and the abstraction. In this paper we show how the verification conditions of several well known methods can be derived.

In general, given an inductive proof method, if a program is correct with respect to a specification $\mathcal{S}$, the verification condition might not hold for $\mathcal{S}$. However if the method is *complete*, then when the program P is correct with respect to specification $\mathcal{S}$, there exists a property $\mathcal{R}$, stronger than $\mathcal{S}$, which verifies the verification condition. We have proved that, for verification conditions which have the form of condition (3) for a suitable α, the derived method is complete if and only if the abstraction is *precise with respect to* T_P, that is if $\alpha(lfp(T_P)) = lfp(T_P^\alpha)$.

In fact, it is known from simple lattice theoretic facts (Park's fixpoint induction [28]) that for a monotonic operator F on a complete lattice

$$lfp(F) \leq \varphi \text{ if and only if } \exists \psi \leq \varphi \; F(\psi) \leq \psi. \tag{4}$$

We can easily derive the following lemma.

Lemma 1. *Let (C, A, α, γ) be a Galois connection between the complete lattices C and A. Let $F : C \to C$ be a monotonic operator on C and $F^\alpha = \alpha \circ F \circ \gamma : A \to A$ be its best abstraction on A. Then, for each $\varphi \in A$,*

$$\alpha(lfp(F)) \sqsubseteq \varphi \textit{ implies } \exists \psi \sqsubseteq \varphi \; F^\alpha(\psi) \sqsubseteq \psi.$$

if and only if (C, A, α, γ) is precise with respect to F.

Then, if the proof method is derived from condition (3) and the abstraction is precise, if a program P is correct with respect to the property $\mathcal{S}$ (that is $\alpha([\![P]\!]) \sqsubseteq \mathcal{S}$) then there exists a property $\mathcal{R}$ stronger than $\mathcal{S}$ (that is $\mathcal{R} \sqsubseteq \mathcal{S}$), which verifies the verification condition of the method (that is $T_P^\alpha(\mathcal{R}) \sqsubseteq \mathcal{R}$). Notice that precision of abstract interpretation can be quite difficult to prove. A sufficient condition for precision, generally easier to check, is *full precision*, that is $\alpha \circ T_P = T_P^\alpha \circ \alpha$. In this paper we show the completeness of some methods by showing the full precision of the underlying abstraction.
[25] contains some methods which allow us to systematically enrich a domain of properties so as to obtain an abstraction which is fully precise with respect to a given function. These methods can be viewed as the base for systematic development of complete proof methods.

3 Success Behaviour of Programs

Let us start by focusing on the program behaviour with respect to the success of non-ground atoms. Let us suppose a set φ_p of atoms (the *extensional specification*

of a property) to be associated with each predicate p. The program P is *success-correct* with respect to *success-properties* $\{\varphi_p\}_{p\in Pred}$ iff

$$\forall p(t) \in Atoms\, p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ implies } p(t)\theta \in \varphi_p,$$

where $\mathbf{G} \overset{\theta}{\rightsquigarrow} \Box$ means that $\mathbf{G}$ succeeds with computed answer θ. Now a set $\{\varphi_p\}_{p\in Pred}$ is just a function $\varphi : Pred \to \wp(Atoms)$. We can introduce then the abstract domain

$$\mathcal{C} = (Pred \to \wp(Atoms), \leq),$$

i.e., the set of functions from predicates to sets of atoms ordered by the pointwise extension of subset order. The idea is that if $p(t) \in \varphi(p(x))$, $\varphi \in \mathcal{C}$, then $p(t)$ is a successful instance of $p(x)$. Elements of $\mathcal{C}$ then, can be viewed as specifications of the success behaviour of programs. The abstraction from the basic domain $\mathbb{D}$ is given by the function

$$\alpha(D) = \lambda p(\mathbf{x}).\{p(\mathbf{x})\theta \mid \delta \in D(p(\mathbf{x})), \partial_\theta(\delta) \text{ successful}\}. \tag{5}$$

α is indeed additive, hence there exists an adjoint function $\gamma : \mathcal{C} \to \mathbb{D}$, giving a Galois connection between $\mathbb{D}$ and $\mathcal{C}$. It can be easily checked that success-correctness can be rephrased as the condition

$$\alpha([\![P]\!]) \leq \varphi.$$

The best abstraction $T_P^{\mathcal{C}} = \alpha \circ T_P \circ \gamma$ in $\mathcal{C}$ of the concrete function T_P can explicitly be defined as

$$T_P^{\mathcal{C}}(I) = \lambda p(\mathbf{x}).\{p(\mathbf{t})\theta \mid p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n) \in P, \\ \forall i \in \{1, \ldots, n\}\, p_i(\mathbf{t}_i)\theta \in I(p_i(\mathbf{x}))\}.$$

The following lemma shows that the abstract interpretation is *fully precise*.

Lemma 2. *Let $\alpha : \mathbb{D} \to \mathcal{C}$ be defined as in (5). Then there exists a function $\gamma : \mathcal{C} \to \mathbb{D}$ such that $(\mathbb{D}, \mathcal{C}, \alpha, \gamma)$ is a Galois connection and is* fully precise *with respect to T_P.*

This means that there is no loss of precision for what concerns the success behaviour, when using $T_P^{\mathcal{C}}$ instead of T_P. As a consequence of lemma (2), the least fixpoint of $T_P^{\mathcal{C}}$ exists and $lfp(T_P^{\mathcal{C}}) = \alpha([\![P]\!])$ [19]. Indeed the semantics specified by the pair $(\mathcal{C}, T_P^{\mathcal{C}})$ is just a reformulation of the well known *C-semantics* [9,24].

Now recalling Section (2), a sufficient condition for success-correctness is

$$T_P^{\mathcal{C}}(\varphi) \leq \varphi, \tag{6}$$

from which a verification condition can then be constructively derived by unfolding $T_P^{\mathcal{C}}$ with its definition.

Theorem 1. *Let P be a logic program and $\{\varphi_p\}_{p\in Pred}$ be a success property. A sufficient condition for P to be success-correct with respect to $\{\varphi_p\}_{p\in Pred}$ is that for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n) \in P$ it is true that*

$$\forall\theta \bigwedge_{i=1}^{n} p_i(\mathbf{t}_i)\theta \in \varphi_{p_i} \textbf{ \textit{implies}} \ p(\mathbf{t})\theta \in \varphi_p. \tag{7}$$

Example 1. Let P be the program

```
app([],Y,Y).
app([X:Xs],Ys,[X:Zs])←app(Xs,Ys,Zs).
```

Let $\varphi_{app} = \{app(t_1, t_2, t_3) \mid t_3$ ground implies t_1 and t_2 ground$\}$. It can easily be checked that for each θ, $app([], Y, Y)\theta \in \varphi_{app}$ and, if $app(Xs, Ys, Zs)\theta \in \varphi_{app}$, then $app([X : Xs], Ys, [X : Zs])\theta \in \varphi_{app}$. By theorem (1), if $app(t_1, t_2, t_3) \overset{\theta}{\rightsquigarrow} \Box$ and $t_3\theta$ is ground, then $t_1\theta$ and $t_2\theta$ are ground.

Note that the verification condition is expressed extensionally. Anyway we can see this case as laying the ground on which methods based on specifications expressed by a formal language can be introduced, as shown in Section (6).

Clearly condition (7) is only sufficient for *success-correctness,* since it is equivalent to property (6), which is just a sufficient condition for $\alpha([\![P]\!]) \leq \varphi_p$. Anyway by lemmas (1) and (2) it follows immediately that the method is complete, that is if P is success-correct with respect to $\{\varphi_p\}_{p\in Pred}$, then there exists a success property $\{\psi_p\}_{p\in Pred}$ with $\forall p\ \psi_p \subseteq \varphi_p$, which verifies condition (7) for each clause of P. Obviously given a valid property φ, there are no general methods to compute a stronger property ψ which verifies (7). Anyway we think that approximated methods to that aim can be better tackled in a framework like ours based on abstract interpretation.

4 Input/Output Behaviour

The properties that we can prove by the above method are properties of the success set of a program. We are now interested in the input/output behaviour, i.e., in the relation between the arguments of a predicate at call time and their instantiation in case of success. This can be specified by a set $\eta_q \subseteq Atoms \times Subst$ associated to each predicate q. The set of properties $\{\eta_q\}_{q\in Pred}$ can be viewed then as a specification of I/O patterns for logic programs. A program is *I/O-correct* with respect to *I/O properties* $\{\eta_q\}_{q\in Pred}$ iff

$$\forall p(t) \in Atoms\ p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ then } (p(t), \theta) \in \eta_p.$$

The collecting domain we can take to capture these observations is $\mathcal{S} = (Pred \rightarrow \wp(Atoms \times Subst), \leq)$. Namely for each predicate p we have a set of pairs $(p(t), \theta)$ with the intended meaning that the predicate p computes a substitution θ for

its arguments if called with argument t. The order is obtained again by pointwise extension of the subset order on $\wp(Atoms \times Subst)$. There exists a Galois connection from $\mathbb{D}$ to $\mathcal{S}$. The abstraction is defined as

$$\alpha(D) = \lambda p(\mathbf{x}).\{(p(\mathbf{t}),\sigma) \mid p(\mathbf{t}) \in Atoms, \delta \in D(p(\mathbf{x})) \text{ successful}, \\ \theta = res(\delta), \exists\sigma = mgu(p(\mathbf{x})\theta, p(\mathbf{t}))\} \tag{8}$$

Again it can be checked that program P is I/O-correct with respect to $\{\eta_q\}_{q\in Pred}$ iff $\alpha([\![P]\!]) \leq \eta$. The best abstraction of T_P in $\mathcal{S}$ is explicitly defined as

$$T_P^{\mathcal{S}}(I) = \lambda p(\mathbf{x}).\{(p(\mathbf{s}),\rho\circ\theta) \mid \exists(p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n)) \in P, p(\mathbf{s}) \in Atoms, \\ \rho = mgu(p(\mathbf{t}),p(\mathbf{s})), \forall i \leq n\ (p_i(\mathbf{t}_i)\rho,\theta_i) \in I(p_i(\mathbf{x}_i)), \\ \theta = mgu((p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n))\rho, \\ (p_1(\mathbf{t}_1)\rho\theta_1,\ldots,p_n(\mathbf{t}_n)\rho\theta_n))\}.$$

The semantics $(\mathcal{S}, T_P^{\mathcal{S}})$ (a variant of the *S-semantics* [24]) is guaranteed not to lose any information about the Input/Output behaviour by the following lemma.

Lemma 3. *Let $\alpha : \mathbb{D} \to \mathcal{S}$ be defined as in (8). Then there exists a function $\gamma : \mathcal{S} \to \mathbb{D}$ such that $(\mathbb{D}, \mathcal{S}, \alpha, \gamma)$ is a Galois connection and is* fully precise *with respect to T_P.*

As before the sufficient condition $T_P^{\mathcal{S}}(\eta) \leq \eta$ can be unfolded and a verification condition for I/O-correctness be obtained.

Theorem 2. *Let P be a logic program and $\{\eta_p\}_{p\in Pred}$ be an I/O property. P is I/O-correct with respect to $\{\eta_p\}_{p\in Pred}$ if for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n) \in P$ and for each $p(\mathbf{s}) \in Atoms$ it is true that*

$$\rho = mgu(p(\mathbf{t}),p(\mathbf{s})) \ \textbf{\textit{and}} \ \bigwedge_{i=1}^{n}(p_i(\mathbf{t}_i)\rho,\theta_i) \in \eta_{p_i} \ \textbf{\textit{implies}} \ (p(\mathbf{s}),\rho\circ\theta) \in \eta_p, \tag{9}$$

where $\theta = mgu((p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n))\rho, (p_1(\mathbf{t}_1)\rho\theta_1,\ldots,p_n(\mathbf{t}_n)\rho\theta_n))$.

By lemma (3) the abstraction is precise. Then, by lemma (1), the proof method is complete.
It is worth noting that for particular classes of specifications, verification of I/O-correctness boils down to verification for success-correctness. For example consider *closed specifications*, such that for each q, $(a\theta,\varepsilon) \in \eta_q$ implies $(a,\theta) \in \eta_q$.

Lemma 4. *Given a set of closed specification $\{\eta_q\}_{q\in Pred}$ then there exists a success property $\{\zeta_q\}_{q\in Pred}$, with each $\zeta_q = \{a \mid (a,\varepsilon) \in \eta_q\}$, such that P is I/O-correct with respect to $\{\eta_q\}_{q\in Pred}$ if and only if P is success-correct with respect to $\{\zeta_q\}_{q\in Pred}$*

This result is essentially a generalization of results in [7, Th. 4.4] and in [22, Prop. 3.2], where it has been proved for I/O specifications like $\varphi_p \twoheadrightarrow \psi_p = \{(a,\theta) \mid a \in \varphi_p \Rightarrow a\theta \in \psi_p\}$, where φ_p and ψ_p are subsets of $Atoms$ and each φ_p is substitution closed.

5 I/O and Call Correctness

Some verification conditions proposed in the literature (see, for example, [23,5,3,7]) take into account a property which is stronger than I/O correctness, i.e. *call correctness*. In fact it requires in addition to I/O correctness that each predicate is called accordingly to a given specification. Having to consider call patterns, which depend on the selection rule, we assume a leftmost selection rule for *SLD* derivations.
The call and I/O behaviour can be specified by a set $\xi_q \subseteq Atoms \times (Atoms \cup Subst)$ associated to each predicate q. The idea is that, if the pair $(p(t), \theta) \in \xi_p$, then if p is called with argument t, it computes the substitution θ. If $(p(t), q(s)) \in \xi_p$ then a call $q(s)$ is generated in the derivation for $p(t)$ with a leftmost selection rule. A program P is *call-correct* with respect to *call-properties* $\{\xi_q\}_{q \in Pred}$ of *Call* iff for each predicate p

$$p(t) \overset{\theta}{\rightsquigarrow} \square \text{ implies } (p(t), \theta) \in \xi_p$$

and

$$p(t) \rightsquigarrow_L^* \langle q(s), \mathbf{G} \rangle \text{ implies } (p(t), q(s)) \in \xi_p,$$

where $\mathbf{G}_1 \rightsquigarrow_L^* \mathbf{G}_2$ means that by applyng SLD resolution with a leftmost selection rule the goal G_1 rewrites to G_2.
A suitable domain is given by $Call = (Pred \to \wp(Atoms \times (Atoms \cup Subst)), \leq)$. The abstraction is

$$\begin{aligned} \alpha(D) = \lambda p(\mathbf{x}).\ \{(p(\mathbf{t}), q(\mathbf{s})) \mid \delta \in D(p(\mathbf{x})), \delta' = \partial_\theta(\delta), first(\delta') = p(\mathbf{t}), \\ last(\delta') = \langle q(\mathbf{s}), \mathbf{G} \rangle\} \\ \bigcup \{(p(\mathbf{t}), \sigma) \mid \delta \in D(p(\mathbf{x})), \delta' = \partial_\theta(\delta) \text{ successful}, \\ first(\delta') = p(\mathbf{t}), res(\delta') = \sigma\}. \end{aligned} \tag{10}$$

It collects calls and computed substitutions. The corresponding best abstract operator T_P^{Call} is defined as follows

$$\begin{aligned} T_P^{Call}(I) = \lambda p(\mathbf{x}).\ Id_p \cup \{(p(\mathbf{s}), q(\mathbf{r})) \mid \exists p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n), \exists k \leq n, \\ \exists \theta_0, \ldots, \theta_{k-1}, \theta_0 = mgu(p(\mathbf{t}), p(\mathbf{s})), \\ \forall i < k\ (p_i(\mathbf{t}_i)\theta_0 \cdots \theta_{i-1}, \theta_i) \in I(p_i(\mathbf{x}_i)), \\ (p_k(\mathbf{t}_k)\theta_0 \cdots \theta_{k-1}, q(\mathbf{r})) \in I(p_k(\mathbf{x}_k))\} \\ \bigcup \{(p(\mathbf{s}), \theta_0 \cdots \theta_n) \mid \exists p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n), \\ \theta_0, \ldots, \theta_n \text{such that} \theta_0 = mgu(p(\mathbf{t}), p(\mathbf{s})), \\ \forall i \leq n\ (p_i(\mathbf{t}_i)\theta_0 \cdots \theta_{i-1}, \theta_i) \in I(p_i(\mathbf{x}_i))\}, \end{aligned}$$

where $Id_p = \{(p(\mathbf{s}), p(\mathbf{s})) \mid p(\mathbf{s}) \in Atoms\}$. The following lemma shows that no information about calls and successes is lost in the computation of abstract iterates.

Lemma 5. *Let $\alpha : \mathbb{D} \to Call$ be defined as in (10). Then there exists a function $\gamma : Call \to \mathbb{D}$ such that $(\mathbb{D}, Call, \alpha, \gamma)$ is a Galois connection and is* fully precise *with respect to T_P.*

The usual sufficient verification condition is $T_P^{Call}(\xi) \leq \xi$, and because of lemma (5), the method is complete.

Theorem 3. *Let P be a logic program and $\{\xi_p\}_{p\in Pred}$ be a call-property. P is call-correct with respect to $\{\xi_p\}_{p\in Pred}$ if for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$ it is true that*

$$\forall p(\mathbf{s}) \in Atoms,\ \theta_0 = mgu(p(\mathbf{s}), p(\mathbf{t})),\ \forall \theta_1, \ldots, \theta_n$$
$$\forall k \leq n\ \textbf{\textit{if}}\ \forall i < k\ (p_i(\mathbf{t}_i)\theta_0 \cdots \theta_{i-1}, \theta_i) \in \xi_{p_i}\ \textbf{\textit{and}}\ (p_k(\mathbf{t}_k)\theta_0 \cdots \theta_{k-1}, q(\mathbf{r})) \in \xi_{p_k}$$
$$\textbf{\textit{then}}\ (p(\mathbf{s}), q(\mathbf{r})) \in \xi_p$$

and

$$\textbf{\textit{if}}\ \forall i \leq n\ (p_i(\mathbf{t}_i)\theta_0 \cdots \theta_{i-1}, \theta_i) \in \xi_{p_i}\ \textbf{\textit{then}}\ (p(\mathbf{s}), \theta_0 \cdots \theta_n) \in \xi_p$$

Indeed the verification condition is quite complex. Anyway, if we consider more restricted classes of assertions, we can derive well known verification methods.

5.1 The Method of Drabent and Maluszynski

The method of Drabent and Maluszynski [23] can be derived by considering as specifications $\mathcal{DM}$ *pre-post properties* $\{pre^i \rightarrowtriangle post^i\}_{i\in I}$, where I is a set of indices and, for each $i \in I$, $pre^i \rightarrowtriangle post^i$ is a *basic pre-post specification*, that is a set $\{pre^i_p \rightarrowtriangle post^i_p\}_{p\in Pred}$, with $pre_p \subseteq Atoms$ and each $post_p \subseteq Atoms \times Atoms$. A program P is $\mathcal{DM}$*-correct* with respect to $\{pre^i \rightarrowtriangle post^i\}_{i\in I}$ iff for each $i \in I$ and each predicate p

$$\forall p(t) \in pre^i_p\ p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ implies } (p(t), p(t)\theta) \in post^i_p$$

and

$$\forall p(t) \in pre^i_p\ p(t) \rightsquigarrow^*_L \langle q(s), \mathbf{G}\rangle \text{ implies } q(s) \in pre^i_q.$$

The domain of such specifications can be defined as

$$\mathcal{DM} = (\wp(Pred \rightarrow \wp(Atoms)\times\wp(Atoms\times Atoms)), \supseteq)$$

and each element $pre \rightarrowtriangle post = \{pre^i \rightarrowtriangle post^i\}_{i\in I}$ can be concretized into an element of *Call*

$$\gamma(pre \rightarrowtriangle post) = \lambda p(x).\{(p(t), \theta) \mid \forall i \in I\ p(t) \in pre^i_p \Rightarrow (p(t), p(t)\theta) \in post^i_p\}$$
$$\bigcup\{(p(t), q(s)) \mid \forall i \in I\ p(t) \in pre^i_p \Rightarrow q(s) \in pre^i_q\}.$$

It can be checked that γ has a left adjoint α, hence there is a Galois connection between *Call* and $\mathcal{DM}$. By composing with the Galois connections between $\mathbb{D}$ and *Call*, we soon obtain a Galois connection between $\mathbb{D}$ and $\mathcal{DM}$.

As usual, the verification condition can be extracted from

$$T_P^{\mathcal{DM}}(pre \rightarrowtriangle post) \leq pre \rightarrowtriangle post, \tag{11}$$

where $T_P^{\mathcal{DM}}$ is the best abstraction of T_P, which is equal to the best abstraction of T_P^{Call} with respect to the Galois connection between *Call* and $\mathcal{DM}$, by abstract interpretation theory. It can be checked that in case the property to verify is a single basic pre-post specification, that is a $\mathcal{DM}$ property $\{pre^i \rightarrowtriangle post^i\}_{i\in I}$ with I a singleton, then the verification condition is exactly (an extensional version of) the verification condition of the method of Drabent and Maluszynski.

Theorem 4. *Let P be a logic program and $\{pre_p \rightarrowtriangle post_p\}_{p\in Pred}$ be a basic pre-post property. Then P verifies the verification condition of the method of Drabent and Maluszynski with respect to $\{pre_p \rightarrowtriangle post_p\}_{p\in Pred}$ if and only if condition (11) is verified.*

It is not difficult to show that the abstraction is precise (it suffices to prove the precision of the abstraction from *Call* and $\mathcal{DM}$ with respect to T_P^{Call}). Hence, the method derived from condition (11) is complete.
Notice that this does not represent a proof of completeness of the method of Drabent and Maluszynski, since in this case, the stronger property to be found must be again a pre-post property $\{pre^i \rightarrowtriangle post^i\}_{i\in I}$ with I a singleton and this is a stronger requirement than precision of abstraction.

5.2 The Method of Bossi and Cocco, Types and Modes

The method of Bossi and Cocco is obtained by considering as basic pre-post specifications a pair $pre_p \rightarrowtriangle post_p$ for each predicate p, where pre_p and $post_p$ are substitution closed subsets of atoms. A program P is $\mathcal{BC}$*-correct* with respect to $\mathcal{BC}-$*properties* $\{pre^i \rightarrowtriangle post^i\}_{i\in I}$ iff for each $i \in I$ and each predicate p

$$\forall p(t) \in pre_p^i \; p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ implies } p(t)\theta \in post_p^i$$

and

$$\forall p(t) \in pre_p^i \; p(t) \rightsquigarrow_L^* \langle q(s), \mathbf{G}\rangle \text{ implies } q(s) \in pre_q^i.$$

We can consider as a domain

$$\mathcal{BC} = (\wp(Pred \rightarrow \wp_\uparrow(Atoms) \times \wp_\uparrow(Atoms)), \supseteq),$$

where $\wp_\uparrow(Atoms)$ is the set of substitution closed sets of *Atoms*. Following [3], an element $pre \rightarrowtriangle post$ of $\mathcal{BC}$ can be viewed as an element of $\mathcal{DM}$ through the function $\gamma(pre \rightarrowtriangle post) = \{\lambda p(x).pre_p^i \rightarrowtriangle (Atoms \times post_p^i) \mid i \in I, (pre^i \rightarrowtriangle post^i) \in (pre \rightarrowtriangle post)\}$. It gives raise to a Galois connection between $\mathcal{DM}$ and $\mathcal{BC}$ Then, by composing abstractions and concretizations, a Galois connection $(\mathbb{D}, \mathcal{BC}, \alpha, \gamma)$ is obtained. As in the previous case a sufficient condition for correctness is

$$T_P^{\mathcal{BC}}(pre \rightarrowtriangle post) \leq pre \rightarrowtriangle post, \tag{12}$$

from which a verification condition can be derived. The condition obtained in the case of singleton $\mathcal{BC}-$properties, is exactly (an extensional version of) the verification condition of the method of Bossi and Cocco [5].

Theorem 5. *Let P be a logic program and $\{pre_p \rightarrowtail post_p\}_{p \in Pred}$ be a basic $\mathcal{BC}-$property. Then P verifies the verification condition of the method of Bossi and Cocco if and only if condition (12) is verified.*

Also in this case completeness of the method can be shown by proving precision of the abstraction from $\mathcal{DM}$ and $\mathcal{BC}$. Also in this case, the same proviso applies about the difference between the completeness of the method derived from (12) and the method of Bossi and Cocco.
For what concerns types and modes, we can view the set of type assignments and mode assignments to predicates as further abstractions with respect to $\mathcal{BC}$ and can use the same techniques used to go from $\mathcal{DM}$ to $\mathcal{BC}$. The result is that we completely reconstruct the hierarchy of [3], by reducing the relationships between proof methods to Galois connections between the corresponding domains.

6 Intensional Specifications

Properties considered so far were extensional sets of atoms. In this section we want to consider intensional properties expressed in a formal specification language. Our idea is that this corresponds to a further step of abstraction.
We will consider the case of success-correctness only. Similar constructions can be given for the other notions of correctness.

Let us consider a first order language $\mathcal{L} = \langle \Sigma, \Pi, V \rangle$. Let $\mathcal{F}$ be a set of formulas (also called *assertions*) of $\mathcal{L}$, expressing properties of interest of arguments of predicates. We assume the signature of $\mathcal{L}$ to include functions, constants and variables of the programs we want to verify. Let a tuple of variables $x_1^p, \ldots, x_n^p$ be associated to each $p \in \Pi$ with arity n.

We need to define what it means for an atom $p(t)$ *to satisfy a property* Φ of $\mathcal{F}$. We consider two cases.
In the first one we fix a term-interpretation $\mathcal{I} = \langle Terms(\Sigma, V), \Sigma_{\mathcal{I}}, \Pi_{\mathcal{I}} \rangle$, that is the set of non-ground terms seen as an $\mathcal{L}$ structure. An atom $p(t_1, \ldots, t_n)$ is said to *$\mathcal{I}$-satisfies* the formula $\Phi[x_1^p, \ldots, x_n^p]$ of $\mathcal{F}$ iff for each σ

$$\mathcal{I} \models_{\sigma[x_1^p, \ldots, x_n^p \setminus t_1, \ldots, t_n]} \Phi[x_1^p, \ldots, x_n^p].$$

In the second case we consider derivability from a theory Γ of formulas of $\mathcal{F}$. That is the atom $p(t_1, \ldots, t_n)$ *Γ-satisfies* the formula $\Phi[x_1^p, \ldots, x_n^p]$ of $\mathcal{F}$ iff $\Gamma \vdash \Phi[x_1^p, \ldots, x_n^p \setminus t_1, \ldots, t_n]$.

Example 2. Let the formulas of $\mathcal{F}$ be the set of first-order classical logic formulas built from atomic formulas $gr(t)$, with the signature Σ of the program in example (2).
Let $\mathcal{I} = (Terms(\Sigma, V), \Sigma_{\mathcal{I}}, \{gr_{\mathcal{I}}\})$ be the interpretation given by the set of all terms of $\mathcal{L}$, with $gr_{\mathcal{I}}$ the set of ground terms. Then $app([a], [\,], d)$ *$\mathcal{I}$-satisfies* the formula $gr(x_3^{app}) \Rightarrow gr(x_1^{app}) \wedge gr(x_2^{app})$. In fact it is true that $d \in gr_{\mathcal{I}}$ implies $[a] \in gr_{\mathcal{I}}$ and $[\,] \in gr_{\mathcal{I}}$.
If we consider the theory $\Gamma = \{gr(f(t_1, \ldots, t_n)) \Leftrightarrow gr(t_1) \wedge \cdots \wedge gr(t_n)\}_{f \in \Sigma}$,

we have that $app([a],[\,],d)$ *Γ-satisfies* the formula $gr(x_1^{app}) \wedge (gr(x_2^{app}) \Leftrightarrow gr(x_3^{app}))$, since $gr([a]) \wedge (gr([\,]) \Leftrightarrow gr(d))$ can be derived straightforwardly from the axioms of Γ.

Let us define what it means for a program P to verify a set of assertions. Given the set of assertions $\{\Theta_p\}_{p \in Pred}$, $\Theta_p \in \mathcal{F}$, P is *$\mathcal{I}-$success-correct* (resp., *Γ-success-correct*) with respect to *assertions* $\{\Theta_p\}_{p \in Pred}$ iff $\forall p(t) \in Atomsp(t) \overset{\theta}{\rightsquigarrow} \Box$ implies $p(t)\theta$ $\mathcal{I}$-satisfies (resp., Γ-satisfies) Θ_p.

Two natural pre-orders can be defined on $\mathcal{F}$, i.e.

$$\Theta \preceq_{\mathcal{I}} \Phi \text{ iff } \mathcal{I} \models \Theta \Rightarrow \Phi$$

and

$$\Theta \preceq_{\Gamma} \Phi \text{ iff } \Gamma, \Theta \vdash \Phi.$$

In both cases we can take the quotients of $\mathcal{F}$ with respect to the induced equivalences and define the partial orders $\mathcal{A}_{\mathcal{I}} = (Pred \rightarrow \mathcal{F}/_{\equiv_{\mathcal{I}}}, \preceq_{\mathcal{I}})$ and $\mathcal{A}_{\Gamma} = (Pred \rightarrow \mathcal{F}/_{\equiv_{\Gamma}}, \preceq_{\Gamma})$, whose elements will be represented as sets $\{\Theta_p\}_{p \in Pred}$, where each Θ_p is a formula of $\mathcal{F}$ with free variables corresponding to arguments of p. The order is given by the pointwise extension of the order between formulas of $\mathcal{F}$ (modulo $\equiv_{\mathcal{I}}$ or $\equiv_{\Gamma}$, resp.).

Consider the following function from $\mathcal{A}_{\mathcal{I}}$ to $\mathcal{C}$:

$$\gamma_{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}).\{p(\mathbf{t}) \in Atoms \mid p(\mathbf{t}) \ \mathcal{I}\text{-}satisfies \ \Theta_p\},$$

If $(\mathcal{F}/_{\equiv_{\mathcal{I}}}, \preceq_{\mathcal{I}})$ is a complete lattice and the meets of $\mathcal{F}/_{\equiv_{\mathcal{I}}}$ behave like classical **and** (that is $p(t)$ $\mathcal{I}$-satisfies $\Theta \sqcap \Phi$ iff $p(t)$ $\mathcal{I}$-satisfies Θ and Φ), it can easily be checked that $\mathcal{A}_{\mathcal{I}}$ is a complete lattice and the function $\gamma_{\mathcal{I}}$ is meet-additive. Hence $\gamma_{\mathcal{I}}$ determines a Galois connection with $\mathcal{C}$. In the case of the domain $\mathcal{A}_{\Gamma}$ the corresponding function is

$$\gamma_{\Gamma}(\Theta) = \lambda p(\mathbf{x}).\{p(\mathbf{t}) \in Atoms \mid p(\mathbf{t}) \ \Gamma\text{-}satisfies \ \Theta_p\},$$

which determines a Galois connection with $\mathcal{C}$, if $(\mathcal{F}/_{\equiv_{\Gamma}}, \preceq_{\Gamma})$ is a complete lattice and the meet of $\mathcal{F}/_{\equiv_{\Gamma}}$ behaves like a classical **and**. We assume from now on to work in classical logic and $\mathcal{F}$ to be closed by **and**. By what has been said, $\gamma_{\mathcal{I}}$ and γ_{Γ} establish then Galois connections with $\mathcal{C}$.
We can define the best abstractions of $T_P^{\mathcal{C}}$ on $\mathcal{A}_{\mathcal{I}}$ and $\mathcal{A}_{\Gamma}$. For example

$$T_P^{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}). \bigvee_{p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n) \in P} \bigwedge \{\Phi \mid \mathcal{I} \models \bigwedge_{i=1}^{n} \Theta_{p_i}[\mathbf{x}_i \backslash \mathbf{t}_i] \Rightarrow \Phi_p[\mathbf{x} \backslash \mathbf{t}]\}.$$

The definition of T_P^{Γ} is similar.
It can be checked that P is *$\mathcal{I}-$success-correct* (resp., *Γ-success-correct*) with respect to *assertions* $\{\Theta_p\}_{p \in Pred}$ iff $\alpha_{\mathcal{I}}(lfp(T_P^{\mathcal{C}})) \leq \Theta$ (resp., $\alpha_{\Gamma}(lfp(T_P^{\mathcal{C}})) \leq \Theta$), where $\alpha_{\mathcal{I}}$ and α_{Γ} are the adjoint functions to $\gamma_{\mathcal{I}}$ and γ_{Γ}. Again $T_P^{\mathcal{I}}(\Theta) \leq \Theta$ and $T_P^{\Gamma}(\Theta) \leq \Theta$ are sufficient conditions for $\mathcal{I}-$success-correctness and Γ-success-correctness, respectively. By exploiting their definition we obtain the following verification conditions.

Theorem 6. *Let P be a logic program and $\{\Phi_p\}_{p\in Pred}$ be assertions of $\mathcal{F}$. A sufficient condition for P to be $\mathcal{I}$-success-correct with respect to $\{\Phi_p\}_{p\in Pred}$ is that for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$ it is true that*

$$\mathcal{I} \models \bigwedge_{i=1}^{n} \Phi_{p_i}[\mathbf{x}_i \backslash \mathbf{t}_i] \Rightarrow \Phi_p[\mathbf{x} \backslash \mathbf{t}] \tag{13}$$

A similar theorem holds for Γ-success-correctness. These results are essentially the verification methods proposed by Clark in [9] and by Deransart, who in [21] decomposes condition (13) to get a more efficient proof method (*annotation method*).

Example 3. Let us consider the program of example (1), the assertions $\mathcal{F}$, the interpretation $\mathcal{I}$ and the theory Γ of (2).
Let us consider the assertion $\Phi_{app} = gr(x_3^{app}) \Rightarrow gr(x_1^{app}) \wedge gr(x_2^{app})$. For the clause $app([], Y, Y)$, it can easily be checked that $\mathcal{I} \models gr(Y) \Rightarrow gr([]) \wedge gr(Y)$ and $\Gamma \vdash gr(Y) \Rightarrow gr([]) \wedge gr(Y)$. For the clause $app([X : Xs], Ys, [X : Zs]) \leftarrow app(Xs, Ys, Zs)$ it can be showed that $\mathcal{I} \models (gr(Zs) \Rightarrow gr(Xs) \wedge gr(Ys)) \Rightarrow (gr([X : Zs]) \Rightarrow gr([X : Xs]) \wedge gr(Ys))$ and $\Gamma \vdash (gr(Zs) \Rightarrow gr(Xs) \wedge gr(Ys)) \Rightarrow (gr([X : Zs]) \Rightarrow gr([X : Xs]) \wedge gr(Ys))$.
By the above theorems we conclude that the program of example (1) is $\mathcal{I}$-success-correct and Γ-success-correct with respect to the assertion $gr(x_3^{app}) \Rightarrow gr(x_1^{app}) \wedge gr(x_2^{app})$.

If the relations $\vdash$ or $\models$ are decidable, we have an effective test to check the conditions. As an example, we could consider the language of properties by Marchiori [26,27], which allows us to express groundness, freeness and sharing of terms. [26] contains also a decidable axiomatization for a fragment of the language.

Concerning the completeness of the axiomatic method, the same result of extensional properties holds, namely the completeness of the method is equivalent to the precision of the abstraction. In this framework this means that the strongest set of assertions $\{\varphi_p\}_{p\in Pred}$, for which P is $\mathcal{I}$-success-correct, verifies condition (13). Similarly for Γ-success-correctness. Obviously the precision strongly depends on the choice of the language and of the set of properties $\mathcal{F}$.

7 Conclusion

Abstract interpretation allows us to make explicit the relation among semantics at different levels of abstraction. As already advocated by the Cousots [20,16], this can be very helpful to understand, organize and synthesize proof methods for program verification. We have shown this in practice, by providing a framework for the verification of logic programs, based on abstract interpretation. In this framework we have defined various proof methods, each obtained in a uniform way by unfolding pre-fixpoint relations on domains obtained by abstracting a concrete semantics of derivations. Some of these methods were already known.

However, we have derived new more general methods and provided a unified view of a class of proof methods, which makes clear the mutual relationship using tools and structures of abstract interpretation.
In particular, we have shown that the relation between operational and declarative properties is just a matter of abstraction. I/O correctness boils down to success correctness for properties which are closed under instantiation. As we might expect, stronger properties require more complex proof methods. Analogously, the same specification given in terms of pre and post conditions can be interpreted as a more abstract (weaker) property, in the case of I/O correctness, or as a stronger property in the case of call correctness.

It is worth noting again, that, by using abstract interpretation theory, the definition of a new verification method simply requires the formalization of the property we are interested in as an abstraction of SLD-derivations (and the abstraction can be based on a formal assertions language). Once we have the abstraction, we systematically derive the specific sufficient correctness conditions. All the general theorems about correctness and completeness hold by construction.

Our results can be applied to abstract diagnosis [12]. The verification condition on the abstract operator is essentially the same as the notion of partial correctness used there. Our conditions might then be used in a similar way to find potential errors. The new relevant features, to be added to abstract diagnosis, are intensional specifications and specifications given as pre- and post-conditions. Using the "call behaviour" model for pre- and post-conditions would allow us to get information about wrong procedure calls.

There are some interesting open problems related to intensional specifications and their relation to traditional abstract domains used in program analysis. The latter have indeed been developed to model specific properties (such as modes, types, groundness, etc.). An abstract element can indeed be viewed as an intensional specification of the set of concrete atoms which is its concretization. Traditional program analysis domains usually lead to abstract interpretations which are not complete. The proof methods based on these domains are then sound but in general not complete. However, most of the proving boils down to computation on the abstract domain. On the other hand, the same program properties can logically be modeled by intensional specifications such as those considered in Section (6) (see, for example, [27]). These domains might be compared to the traditional ones from the viewpoint of precision. Some of them might even turn out to be complete. Hence the logical domains derived for verification might be useful for program analysis.

The hierarchy of proof methods discussed in this paper is shown in Figure (1). It is worth noting that every proof method in the hierarchy can be further abstracted by using intensional specifications or properties specified by elements of program analysis abstract domains.

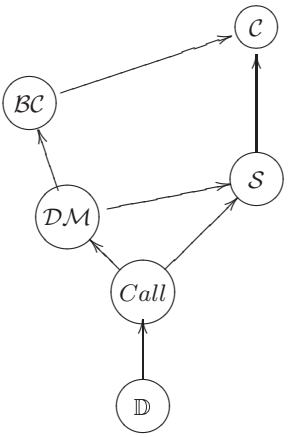


Fig. 1. The hierarchy of semantics and verification methods.

References

1. K. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997. 102, 103
2. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990. 103
3. K. R. Apt and E. Marchiori. Reasoning about Prolog Programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 3, 1994. 102, 103, 109, 111, 112
4. G. Birkhoff. Lattice Theory. In *AMS Colloquium Publication, third ed.*, 1967. 103
5. A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89*, pages 96–110, 1989. 102, 103, 109, 111
6. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation '93*, pages 46–55, 1993. 102
7. J. Boye and J. Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, 1997. 108, 109
8. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press. 102
9. K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979. 103, 106, 114
10. M. Comini. *An abstract interpretation framework for Semantics and Diagnosis of logic programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1998. 103
11. M. Comini, G. Levi, and M. C. Meo. Compositionality of *SLD*-derivations and their abstractions. In J. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming*, pages 561–575. The MIT Press, 1995. 103, 104
12. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. Submitted for publication, 1996. 102, 115

13. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996. 102
14. M. Comini and M. C. Meo. Compositionality Properties of *SLD*-derivations. *Theoretical Computer Science*, 1997. To appear. Available at http://www.di.unipi.it/∼comini/papers.html. 103, 104
15. P. Cousot. Methods and Logics for Proving Programs. In J. V. Leeuwen, editor, *Formal Methods and Semantics*, volume B of Handbook of Theoretical Computer Science, pages 843–993. Elsevier Science Publishers B.V. (North-Holland), 1990. 103
16. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition system by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL:http://www.elsevier.nl/locate/entcs/volume6.html. 102, 114
17. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977. 102, 103
18. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979. 102, 103
19. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992. 106
20. P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992. 114
21. P. Deransart. Proof Methods of Declarative Properties of Definite Programs. *Theoretical Computer Science*, 118(2):99–166, 1993. 102, 103, 114
22. W. Drabent. It is Declarative. In *ILPS'97. Workshop on Verification, Model Checking and Abstract Interpretation*, 1997. 103, 108
23. W. Drabent and J. Maluszynski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988. 102, 103, 109, 110
24. 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. 106, 108
25. R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proc. of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 231–245. Springer-Verlag, Berlin, 1997. 105
26. E. Marchiori. A Logic for Variable Aliasing in Logic Programs. In G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP'94)*, number 850 in LNCS, pages 287–304. Springer Verlag, 1994. 114
27. E. Marchiori. Design of Abstract Domains using First-order Logic. In M. Hanus and M. Rodriguez-Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96)*, number 1139 in LNCS, pages 209–223. Springer Verlag, 1996. 114, 115
28. D. Park. Fixpoint Induction and Proofs of Program Properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, number 5, pages 59–78. Edinburgh Univ. Press, 1969. 105

Detecting Unsolvable Queries for Definite Logic Programs

Maurice Bruynooghe[1], Henk Vandecasteele[1], D. Andre de Waal[2], and Marc Denecker[1]

[1] Departement Computerwetenschappen, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{maurice,henkv,marcd}@cs.kuleuven.ac.be
[2] Centre for Business Mathematics and Informatics
Potchefstroom University for Christian Higher Education, South Africa
BWIDADW@puknet.puk.ac.za

Abstract. In logic programming, almost no work has been done so far on proving that certain queries cannot succeed. Work in this direction could be useful for queries which seem to be non-terminating. Such queries are not exceptional, e.g. in planning problems. The paper develops some methods, based on abduction, goal-directedness, tabulation, and constraint techniques, for proving failure of queries for definite logic programs. It also reports some experiments with various tools.

1 Introduction

Given some logic program, executing a query may have many different outcomes. It may terminate or run forever (in practice until some resource is exhausted). In both cases, the query may or may not lead to answers. There is a large body of literature on termination analysis (see [7] for a survey). However, termination conditions are not decidable and automated methods are based on analysing the size of syntactical structures. So there is a substantial class of programs for which automated termination analysis fails. For some of these, loop checking methods [2,3,15] monitoring the execution, bring some relief, by pruning some infinitely failing branches. Methods have to choose between pruning too much, causing incompleteness in the search for solutions, and preserving completeness but still allowing some infinite computations. Also an execution mechanism augmented with tabling such as XSB [14] or an approach such as [4] reduces the number of non-terminating queries.

So far, there are almost no works on program analysis which attempt to recognise (infinitely) failing queries. Problems with planners, which generate more and more complex objects until one is found with a particular property have been the incentive to start our research. It would be very useful to be able to stop their infinite search for a solution to an unsolvable problem. Also certain program properties can be proven by proving failure of a particular query. As a trivial example, consider a program which knows about even and odd numbers.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 118–133, 1998.

One can prove the program does not define a number which is both even and odd by proving that a query asking for such a number fails.

Conceptually, there is a simple way to show that a query must fail: find a model of the program in which the query is false. In [8] some of the authors of the current paper made a first exploration of the issues involved in finding such a model for definite programs. The current paper develops two methods for automating the search for a pre-interpretation underlying such a model. A first approach combines abduction and tabulation to direct the search for a pre-interpretation. A second approach considers the abducibles as constraints and uses techniques from finite domain constraint solving [20] to further prune the search. Also the suitability of alternative methods for solving this problem is analysed. The use of a general purpose model generation tool [16,17] is evaluated. We have also explored whether tools for type inference [11,5] can show that such queries have an empty success set and whether conjunctive partial deduction [13] can specialise such queries into a trivially failing program.

Section 2 recalls the basics about pre-interpretations and introduces a trivial example. Section 3, explains how a pre-interpretation can be described by a number of facts, how a program can be abstracted as a DATALOG program, and how the least model based on that pre-interpretation can be queried by evaluating the abstracted query on the DATALOG program. In section 4, a procedure combining abduction with tabulation and a variant handling the abducibles as constraints are developed. Other approaches we are aware of which can directly or indirectly prove failure are discussed in Section 5. In section 6, the different approaches are compared. Finally, in section 7, we draw some conclusions. We assume some familiarity with the basics of tabulation, e.g. [19,14,21].

2 Preliminaries

A pre-interpretation J of a program P consists of a domain $D = \langle d_1, \ldots, d_m \rangle$ and, for every functor f/n a mapping f_J from D^n to D. An interpretation I based on a pre-interpretation J consists of a mapping p_I from D^n to $\{true, false\}$ for every predicate p/n in P. An interpretation is often identified by the set of atoms $p(d_1, \ldots, d_n)$ for which $p_I(d_1, \ldots, d_n)$ is mapped to $true$. An interpretation M is a model of a program P iff all clauses of P are true under the interpretation M. A definite program always has a model (map $p_I(d_1, \ldots, d_n)$ to $true$ for all predicates and all domain elements). The intersection of two models is also a model and there is always a unique least model. As a consequence, if an existentially quantified conjunction $\exists_{\overline{X}} L_1 \wedge \ldots \wedge L_n$ is false in a model based on a pre-interpretation J then it is also false in the least model based on that pre-interpretation. So, given a pre-interpretation, it suffices for our purposes to evaluate the conjunction in the least model.

Example 1. Even/odd

`even(0).` `even(s(X))` ← `odd(X).` `odd(s(X))` ← `even(X).`

$D = \{\mathcal{E}, \mathcal{O}\}$

$0_J = \mathcal{E} \quad s_J(\mathcal{E}) = \mathcal{O} \quad s_J(\mathcal{O}) = \mathcal{E}$

The least model is $\{even(\mathcal{E}), odd(\mathcal{O})\}$. The query $\leftarrow$ `even(X), odd(X)` fails because $\exists_X even(X), odd(X)$ is false in this model. Executing the program with SLD or with a tabulating procedure (e.g. XSB [14]) results in infinite failure.

3 Proof Procedures

Pre-interpretations based on finite domains have been used in the analysis of logic programs. The approach was pioneered by Codish and Demoen [6] for groundness analysis and subsequently by others, e.g. by [10] for type analysis. They used *abstract compilation* and represented the functions f_J/n of the pre-interpretation by $n+1$-ary relations over the domain and replaced all terms in the program by their pre-interpretation. This gives a so called abstract program which is a DATALOG program. Its finite model expresses declarative properties.

Example 2. In Example 1, we can define the pre-interpretation by:
$0_J(\mathcal{E})$. $s_J(\mathcal{E}, \mathcal{O})$. $s_J(\mathcal{O}, \mathcal{E})$.
To abstract the program, non-variable terms are replaced by fresh variables which are defined by the appropriate relations (a term $f(t_1, \ldots, t_n)$ is replaced by a fresh variable X and the atom $f_J(t_1, \ldots, t_n, X)$ is added to the body; this construction is repeated until all terms have disappeared). Variables are left as they are, the effect of the abstraction is that they now range over the domain of the pre-interpretation. This gives the following program:

```
even(X) ← 0_J(X)
even(Y) ← s_J(X,Y), odd(X).
odd(Y) ← s_J(X,Y), even(X).
```

The clauses together with the facts of the pre-interpretation are a DATALOG program. The least model is $\{0_J(\mathcal{E}), s_J(\mathcal{E}, \mathcal{O}), s_J(\mathcal{O}, \mathcal{E}), even(\mathcal{E}), odd(\mathcal{O})\}$. The formula $\exists_X even(X), odd(X)$ is false in this model. While the query $\leftarrow$ `even(X), odd(X)` is nonterminating under SLD, it fails finitely under well known proof procedures such as bottom-up evaluation after magic-set transformation or top-down methods enriched with tabulation such as OLDT [19] and XSB [14].

4 The Search for the Right Pre-interpretation

To prove that a query for a program P, which seems to run forever without returning a solution, fails, one has to select a domain and a pre-interpretation and has to show finite failure when executing the abstracted query with the abstracted program. A straightforward way consists of selecting a domain, and trying all pre-interpretations until one is found for which the query fails. If none exists, one can try again with a larger domain. However, for programs with a substantial number of function symbols and constants, this quickly results in a very large search space. Indeed, with a n-element domain, an m-ary functor has $n^{(n^m)}$ possible pre-interpretations.

4.1 An Abductive Approach

To prune the search one can employ an abductive procedure: execute the abstract program with an initially unknown pre-interpretation, and abduce the different components of the pre-interpretation when they are needed during the execution. As soon as the query succeeds, backtracking can be initiated. To do so, one declares the predicates $f_J/n+1$ as abducibles, add constraints that the pre-interpretation of each functor f/n is a total function, i.e. that one fact $f_J(d_1,\ldots,d_n,d)$ must be abduced for every combination $\langle d_1,\ldots,d_n\rangle$ of domain elements and employ a general purpose abductive procedure such as SLDNFA [9]. Some initial experiments showed the feasibility[1], but also the need for a dedicated procedure which allows to experiment with different control strategies. We designed and implemented an abductive procedure for definite programs which makes use of tabulation and which has the integrity constraints on the pre-interpretation hardwired in the code. By doing some experiments we gained a better understanding of the issues which are important to control the search. For example we observed that it performed better when tabling only the most general call for each predicate. Below we describe the procedure as (inference) rules which map sets of clauses (a "state") to sets of clauses. The control —in which order the different rules are applied— is left undetermined. The system attempts to abduce a pre-interpretation such that the query fails under a top-down execution with tabulation of the abstract program, i.e. is false in the corresponding model of the abstract program.

We need some notational conventions. The state of the computation is represented as a set of clauses. The symbol `Cl` is used to represent a clause and the symbol `Cls` to represent a set of clauses. We use `Cl :: Cls` to represent the set of clauses $\{$`Cl`$\} \cup$ `Cls` . `A` and `B` are used to represent atoms, `As` and `Bs` to represent sequences of atoms. A clause is represented as `H` $\leftarrow$ `As` in which the head `H` is an atom (or `false`). Given a query $\leftarrow$ `As`, the initial state of the derivation is represented as `(false` $\leftarrow$ `As)`. `p/n` refers to a predicate of the original program, calls to such predicates are tabled. $\texttt{abduce}_f\texttt{(t}_1\texttt{,...,t}_n\texttt{,X)}$ is the notation we use for a call to an abducible predicate $\texttt{f}_J\texttt{(t}_1\texttt{,...,t}_n\texttt{,X)}$ of the pre-interpretation. These calls are not tabled. In a state of the derivation (a set of clauses), the calls $\texttt{p(}\overline{X}\texttt{)}$ which are tabled are represented implicitly through the occurrences of literals $\texttt{Lookup(p(}\bar{t}\texttt{))}$ in the bodies of the clauses. The answers to tabled calls are represented by clauses with an empty body.

We assume a fixed number of domain elements. Rule 1 handles a new call to a program predicate. The call is wrapped inside `Lookup` to indicate that it is waiting for answers. Nothing else needs to be done when the predicate was called before. Otherwise, the clauses defining the predicate are added. Eventually, they will lead to facts which are answers to the most general call of the predicate. Rule 2 describes the *lookup* step: a (wrapped) call is unified with an answer and the resolvent is added (for simplicity of presentation, we assume the state remains the same when —up to renaming— the same clause is derived a second

[1] Because the procedure lacks tabulation, some extra transformation of the clauses of recursive predicates was required.

time). Rule 3 resolves an abducible with an already abduced fact and adds the resolvent. Rule 4 abduces a new fact. Several domain elements are available, so a *choice point* is created. Rule 5 detects that the query has an answer, i.e. that the chosen pre-interpretation does not satisfy, and triggers backtracking.

1. $\mathtt{H} \leftarrow \mathtt{p}(\overline{t}),\mathtt{As} :: \mathtt{Cls}$
   Let $\mathtt{p}(\overline{X_i}) \leftarrow \mathtt{Bs}_i$ (i: 1,...,m) be the clauses in the definition of p. If this is the first call to p (Cls does not contain a clause with an atom $\mathtt{Lookup}(\mathtt{p}(\overline{r}))$) then the new state is:
   $\mathtt{p}(\overline{X_1})\leftarrow\mathtt{Bs}_1 :: \ldots :: \mathtt{p}(\overline{X_m})\leftarrow\mathtt{Bs}_m :: \mathtt{H} \leftarrow \mathtt{Lookup}(\mathtt{p}(\overline{t})),\mathtt{As} :: \mathtt{Cls}$
   Else the new state is:
   $\mathtt{H} \leftarrow \mathtt{Lookup}(\mathtt{p}(\overline{t})),\mathtt{As} :: \mathtt{Cls}$
2. $\mathtt{H} \leftarrow \mathtt{Lookup}(\mathtt{p}(\overline{t})),\mathtt{As} :: \mathtt{p}(\overline{s}) \leftarrow :: \mathtt{Cls}$
   where $\mathtt{p}(\overline{s}) \leftarrow$ is a fact which unifies with $\mathtt{p}(\overline{t})$.
   The new state is:
   $(\mathtt{H} \leftarrow \mathtt{As})\mathtt{mgu}(\overline{t},\overline{s}) :: \mathtt{H} \leftarrow \mathtt{Lookup}(\mathtt{p}(\overline{t})), \mathtt{As} :: \mathtt{p}(\overline{s}) \leftarrow :: \mathtt{Cls}$
3. $\mathtt{H} \leftarrow \mathtt{abduce}_f(\overline{t}),\mathtt{As} :: \mathtt{abduce}_f(\overline{s}) \leftarrow :: \mathtt{Cls}$
   where $\mathtt{abduce}_f(\overline{s}) \leftarrow$ is a fact unifying with $\mathtt{abduce}_f(\overline{t})$.
   The new state is (use of abduced fact):
   $(\mathtt{H} \leftarrow \mathtt{As})\mathtt{mgu}(\overline{t},\overline{s}) :: \mathtt{H} \leftarrow \mathtt{abduce}_f(\overline{t}),\mathtt{As} :: \mathtt{abduce}_f(\overline{s}) \leftarrow :: \mathtt{Cls}$
4. $\mathtt{H} \leftarrow \mathtt{abduce}_f(\mathtt{t}_1,\ldots,\mathtt{t}_m,\mathtt{t}),\mathtt{As} :: \mathtt{Cls}$
   Let $\mathtt{d}_1,\ldots,\mathtt{d}_m$ be domain elements which unify respectively with $\mathtt{t}_1,\ldots,\mathtt{t}_m$ and such that Cls has not yet a fact $\mathtt{abduce}_f(\mathtt{d}_1,\ldots,\mathtt{d}_m,\mathtt{d}) \leftarrow$ for some domain element d. A **choice point** is created. A domain element d is selected and the new state is (abduction of a new fact):
   $\mathtt{abduce}_f(\mathtt{d}_1,\ldots,\mathtt{d}_m,\mathtt{d}) \leftarrow :: \mathtt{H} \leftarrow \mathtt{abduce}_f(\mathtt{t}_1,\ldots,\mathtt{t}_m,\mathtt{t}),\mathtt{As} :: \mathtt{Cls}$
5. $(\mathtt{false} \leftarrow) :: \mathtt{Cls}$
   Backtrack to the state corresponding to the most recent **choice point** with an untried domain element d.

We have a proof that the query fails when the system reaches a stable final state (no new clauses can be inferred —up to renaming— and $(\mathtt{false} \leftarrow)$ is not part of the state). The search for a proof fails when the rewriting fails (rule 4 has exhausted all choices for a tuple $\mathtt{d}_1,\ldots,\mathtt{d}_m$ of domain elements). In the latter case, one could increase the size of the domain and start over. The application of the rules is nondeterministic. An obvious control strategy delays the introduction of choice points as long as possible[2]. We have experimented with various search rules (selection of clause) and computation rules (selection of literal). In section 6 we give more details on the system which behaved best. Experiments revealed that none of the strategies is superior and that overall performance on a particular problem is rather dependent on the order in which the pre-interpretations of the different functors are abduced. This motivated the search for a better approach.

[2] Symmetries: with n domain elements, each pre-interpretation can be mapped into an equivalent one (for what concerns the truth of the query in the least model) by permuting the domain elements. Our implementations avoid most symmetries.

4.2 A Constraint Approach

From now on, we use a slightly different notation for the abstract program. The abstraction of an atom `p(f(a))` is denoted as `abduce(f(a),X), p(X)` (We use `abduce(f(a),X)` as an abbreviation of $\texttt{abduce}_a\texttt{(Y), abduce}_f\texttt{(Y,X)}$.).

The problem with the abductive approach can be illustrated with the following example: Assume that the pre-interpretation of a functor `f/1` has already been abduced as `abduce(f(d1),d1)` and `abduce(f(d2),d2)` and that a clause `false` ← `abduce(f(g(h(a))),X), abduce(g(h(a)),X)` is derived. Whatever is the pre-interpretation for `a`, `h`, and `g`, `false` ← will be derived. The abductive systems will not revise the pre-interpretation for `f` before having *thrashed* over most of the pre-interpretations of `a`, `h`, and `g`.

A constraint based approach can to a large extent avoid such problems. We consider the abducibles as constraints and use a special purpose constraint solver which checks the existence of a pre-interpretation which satisfies all constraints. In the above example, if the pre-interpretation of `f/1` is constrained to the shown one and the clause `false` ← `abduce(f(g(h(a))),X), abduce(g(h(a)),X)` is derived, then the solver detects the inconsistency and triggers backtracking.

This approach makes it necessary to reformulate our abductive system. The major difference is wrt. the tabulation. The answers to a tabled predicate are no longer ground facts but constrained facts (of the form $\texttt{p}(\overline{X})$ ← `abduce(...), ..., abduce(...)`). A problem is that one can have an infinite number of syntactically different answers. However, with a finite domain and a fixed pre-interpretation, the set of answers (its model) is finite. So it must be possible to add constraints which enforce the finiteness. Before presenting the formal system, we illustrate the main ideas with the `even/odd` example.

Example 3. Even/odd
The program is as follows:

```
even(X) ← abduce(0,X).
even(Y) ← abduce(s(X),Y), odd(X).
odd(Y) ← abduce(s(X),Y), even(X).
```

We represent the state of the derivation by three components, the set of clauses, the set of answers and the constraint store, holding the set of constraints (as before the component with the fixed abstract program is left out). ϵ stands for the empty set. *Lookup* is abbreviated as *L*, and *false, abduce, even* and *odd* respectively as *f, ab, e* and *o*. Finally, *sb* is the abbreviation of *subsumed*. In the initial state ((0) in Table 1) the only clause is the query. The leftmost atom of a program predicate is selected and the two clauses defining `even/1` are activated (1). The second clause is a constrained fact. In principle, we have to create a choice point. The first alternative adds the constraint `subsumed(even(X)` ← `abduce(0,X), {})` to the constraint store in an attempt to have the new fact subsumed by the existing ones. The constraint is false for every pre-interpretation, so the second alternative is taken: The fact is added to the set of answers and the constraint `not(subsumed(even(X)` ← `abduce(0,X), {}))` is added to the store. The constraint is redundant wrt. the (empty) store, so

	clauses	answers	constraint store
0	$f \leftarrow e(X), o(X)$	ϵ	ϵ
1	$f \leftarrow L(e(X)), o(X)$ $e(X) \leftarrow ab(0, X)$ $e(Y) \leftarrow ab(s(X), Y), o(X)$	ϵ	ϵ
2	$f \leftarrow L(e(X)), o(X)$ $e(Y) \leftarrow ab(s(X), Y), o(X)$	$e(X) \leftarrow ab(0, X)$	ϵ
4	$f \leftarrow L(e(X)), o(X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$	ϵ
6	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), o(X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$ $o(Y) \leftarrow ab(s(0), Y)$	$e(X) \leftarrow ab(0, X)$	ϵ
8	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	ϵ
10	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $f \leftarrow ab(0, X), ab(s(0), X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $e(Y) \leftarrow ab(s(s(0)), Y)$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	ϵ
11	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $e(Y) \leftarrow ab(s(s(0))), Y),$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y),$	$f \leftarrow ab(0, X), ab(s(0), X)$
12	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	$f \leftarrow ab(0, X), ab(s(0), X)$ $sb(e(Y)) \leftarrow ab(s(s(0)), Y),$ $\{e(X) \leftarrow ab(0, X)\})$

Table 1. Constraint based execution of even/odd

the store remains empty (2). The call `odd(X)` is selected in the second clause, the clause defining `odd/1` is added, and its atom `even(X)` is selected (4). The stored answer is used to resolve with the first and third clause, this results in two new clauses [3] (6). In the first, the atom `odd(X)` is selected, the last is an answer for `odd/1`. We have a choice point but again the first alternative creates an inconsistent store and the second alternative a redundant constraint, so the net effect is that the fact is added to the answer set (8). This answer is used to resolve with the second and third clause, resulting in two new clauses (10). The first one

[3] Remark that `abduce(s(X), Y), abduce(0, X)` is abbreviated by `abduce(s(0), Y)`.

is a constraint which is consistent with the current store and added to it (11), It means that `0` and `s(0)` should be different under the pre-interpretation. The second is an answer for `even/1`. This time we have a real choice point. The first alternative enforces the constraint that the new answer is subsumed by the existing answers, so the answer is not stored and the constraint `subsumed(even(Y))` ← `abduce(s(s(0)),Y)`, {`even(X)` ← `abduce(0,X)`}, which is consistent with the current store is added to it. It actually means that 0 and $s(s(0))$ must be equal under the pre-interpretation. A stable state is reached so the query is false in the models of the pre-interpretations satisfying the constraints of the store. The pre-interpretation `abduce(0,d1)`, `abduce(s(d1),d2)` and `abduce(s(d2),d1)` is a solution (the subsumption test reduces to `subsumed(even(d1)`, {`even(d1)`}) and yields true). Observe that a `not(subsumed(...))` constraint is added to the constraint store each time that a new answer is added to the answer set. The conjunction of these constraints enforces finiteness of the answer set and termination of the algorithm (The number of distinct atoms in the model of a m-ary predicate is limited to m^n, so at most m^n times an answer can be added that is not subsumed by the previous ones.). An alternative approach which also ensures termination and completeness of the search discards the `not(subsumed(...))` constraints and uses a weaker but easier to verify constraint which restrict the number of answers for a predicate p/m to m^n.

Below, we use $\diamond$ to separate the three components of the state. The symbols `As` and `Bs` stand for any sequence of atoms, while `Abds` stands for a sequence consisting solely of abduce atoms. `Store` stands for a conjunction (set) of constraints, `Answers` for a set of answers (constrained facts) and $\texttt{Answers}_p$ for the subset of answers about predicate `p` . The initial state is given by `false` ← `As` $\diamond$ ϵ $\diamond$ ϵ where `As` is the query. Assume n domain elements. Remember that arguments of program predicates of the abstracted program are always variables.

1. `H` ← `Abds, p(`$\overline{X}$`), As :: Cls` $\diamond$ `Answers` $\diamond$ `Store`
   Let `p(`$\overline{X_i}$`)` ← $\texttt{Bs}_i$ $(i : 1, \ldots, m)$ be the clauses in the definition of `p`. If this is the first call to `p` (`Cls` does not contain a clause with an atom `Lookup(p(`$\overline{Y}$`))` then the new state is :
   `p(`$\overline{X_1}$`)` ← $\texttt{Bs}_1$ `::...::` `p(`$\overline{X_m}$`)` ← $\texttt{Bs}_m$ `:: H` ← `Abds,Lookup(p(`$\overline{X}$`)), As :: Cls` $\diamond$ `Answers` $\diamond$ `Store`
   Else the new state is:
   `H` ← `Abds, Lookup(p(`$\overline{X}$`)), As :: Cls` $\diamond$ `Answers` $\diamond$ `Store`
2. `H` ← $\texttt{Abds}_1$`, Lookup(p(`$\overline{X}$`)), As :: Cls` $\diamond$
   `p(`$\overline{Y}$`)` ← $\texttt{Abds}_2$ `:: Answers` $\diamond$ `Store`
   The new state is:
   `(H` ← $\texttt{Abds}_1$`,` $\texttt{Abds}_2$`, As)mgu(`$\overline{X}$`,`$\overline{Y}$`) :: H` ← $\texttt{Abds}_1$`, Lookup(p(`$\overline{X}$`)), As :: Cls` $\diamond$ `p(`$\overline{Y}$`)` ← $\texttt{Abds}_2$ `:: Answers` $\diamond$ `Store`
3. `false` ← `Abds :: Cls` $\diamond$ `Answers` $\diamond$ `Store`
   The new state is:
   `Cls` $\diamond$ `Answers` $\diamond$ `false` ← `Abds :: Store`

4. `Cls` $\diamond$ `Answers` $\diamond$ `Store` where `Store` is inconsistent.
   Backtrack to the state corresponding to the most recent **choice point** with an untried alternative.
5. `p(`$\overline{X}$`)` $\leftarrow$ `Abds :: Cls` $\diamond$ `Answers` $\diamond$ `Store`
   A **choice point** is created. Under the first alternative the new system is:
   `Cls` $\diamond$ `Answers` $\diamond$ `subsumed(p(`$\overline{X}$`)` $\leftarrow$ `Abds, Answers`$_p$`) :: Store`
   Under the second alternative the new system is:
   `Cls` $\diamond$ `p(`$\overline{X}$`)` $\leftarrow$ `Abds :: Answers` $\diamond$
   `not(subsumed(p(`$\overline{X}$`)` $\leftarrow$ `Abds, Answers`$_p$`)) :: Store`

Rules 1 and 2 are as before. Rule 3 adds a new constraint to the constraint store and rule 4 checks its consistency. Rule 5 processes a new answer and creates a choice point. The first alternative adds a subsumption constraint and drops the new fact. The second alternative adds a not-subsumed constraint to the store and the answer to the answer set. The not-subsumed constraint is redundant when the subsumed constraint creates an inconsistent store (often the case when a first answer is added). The store is inconsistent when it has more than m^n answers for p/m. So an alternative is to drop the not-subsumed constraints and to check the weaker constraint on the size of `Answers`$_{p/m}$. Both cases guarantee termination and completeness of the search for a given domain size.

The efficiency of the consistency checks is crucial for the overall performance. Space lacks to give a full description of the encoding as a finite domain problem. We sketch the main ideas using the even-odd example. Let $\mathcal{D}$ be the domain of the pre-interpretation. We use two kinds of finite domain variables: variables D_t ranging over $\mathcal{D}$ and representing the pre-interpretation of the term t and boolean variables $B_{t_1=t_2}$ indicating whether or not the terms t_1 and t_2 have the same pre-interpretation. Such boolean variables are linked to the domain variables through definitions $B_{t_1=t_2} \leftrightarrow D_{t_1} = D_{t_2}$ which ensure propagation of new information. Which domain variables are created is determined by the terms which occur in the constraints. Consider the constraint `false` $\leftarrow$ `abduce(0,X), abduce(s(0),X)`. To handle it we introduce domain variables D_0 and $D_{s(0)}$. We can translate the constraint in `false` $\leftarrow$ D_0 `= X,` $D_{s(0)}$ `= X` or, after elimination of `X`: `false` $\leftarrow$ $B_{0=s(0)}$ or $B_{0=s(0)} = 0$. To express the connection between 0 and $s(0)$, we add for all $d \in \mathcal{D}$ the constraint $B_{0=d} \leq B_{s(0)=s(d)}$[4]. Note that this implies the creation of domain variables $D_{s(d)}$. Now consider the constraint `subsumed(even(Y))` $\leftarrow$ `abduce(s(s(0)),Y),{even(X)` $\leftarrow$ `abduce(0,X),}`. It contains a new term $s(s(0))$, so a domain variable $D_{s(s(0))}$ is created and it is linked with $D_{s(0)}$ by adding for all $d \in \mathcal{D}$ the constraint $B_{s(0)=d} \leq B_{s(s(0))=s(d)}$. The subsumption constraint is expressed as $B_{s(s(0))=0} = 1$ (Its negation as $B_{s(s(0))=0} = 0$). This translation ensures that all choices which are made are immediately propagated.

[4] Or equivalently $B_{0=d} \rightarrow B_{s(0)=s(d)}$.

5 Alternative Approaches

Model generation. The logic program and the clause `false` ← `query` can be considered as a logical theory. A model of this theory is a proof that the query fails. There exist general purpose tools for generating models of logical theories. FINDER [16,17], written in C, is such a tool, it takes as input a set of clauses in a many-sorted first order language, together with specifications of finite cardinalities of the domains for the sorts, and generates interpretations on the given domains which satisfy all the clauses [16]. The systems performs an exhaustive search for interpretations of the given language, using the declared clauses as constraints to direct backtracking. A major difference between FINDER and our approach is that there is no explicit control on the order of evaluation. FINDER generates an interpretation and tests it against all clauses; if all clauses are true with respect to the generated interpretation, we have a model that is accepted and printed. If one or more of the clauses are false, the interpretation is adjusted so as to generate an improved candidate interpretation. The process continues until the search space is exhausted or a model has been found.

Regular approximations. Within the context of program analysis, the most obvious approach to prove failure is to add a clause `shouldfail(`$\overline{X}$`)` ← `query(`$\overline{X}$`)` and to use one or another kind of type inference to show that the success-set of `shouldfail(`$\overline{X}$`)` is empty. A typical representative of such systems is described in [11] which computes a regular approximation of the program. Roughly speaking, for each argument of each predicate, the value it can take in the success-set[5] are approximated by a type (a canonical unary logic program). Failure of the query is proven if the types of `shouldfail(`$\overline{X}$`)` are empty. Also set based analysis [12] can be used to approximate the success-set.

Program specialisation. One could also employ program transformation, and more specifically program specialisation techniques to prove failure of the query. If for the given query, the program can be specialised in the empty program, then the query trivially fails. A technique which has almost the same power as transformations based on the fold/unfold approach is conjunctive partial deduction [13]. By specialising conjunctions of atoms instead of single atoms, it can achieve substantially better results than other specialisers. For example it can specialise the even/odd program into the empty program for our example query.

6 Experiments

For the abductive systems we experimented extensively with 5 different control regimes. We report here on the most successful one i.e. the system with the best worst cases. In this system, as in all others, rule 5, which triggers backtracking, has top priority. Rules 2 and 3, which use respectively tabled answers and abduced facts to perform a resolution step, have equal priority, as well as rule 1

[5] The set of ground atoms which are logical consequences of the program.

for the case that the clause has a call to an already tabled predicate. If none of these rules apply and there is an unprocessed clause with only calls to not yet tabled program predicate, then rule 1 is applied on that clause. Otherwise rule 4 is applied on a selected clause and the pre-interpretation is extended.

In the constraint system, rule 4 which checks for the consistency of the Store has top priority each time a constraint is added to the store as it can trigger backtracking. Rule 5, which can create a choice point, has lowest priority. A first implementation used the weaker constraints on the number of answers for each predicate instead of the not-subsumed constraints. Using the latter resulted in significant better pruning on the hard problems (and marginal slow-down on easier ones). The finite domain solver is incremental.

Table 2 details the benchmarks[6]. Besides the name, the table gives the domain size, the number of functors, the number of abduced facts in a complete pre-interpretation[7], the number of predicates defined in the program, the total number of program clauses and the number of program clauses with a head predicate of arity 1, of arity 2 and of arity 3[8].

`odd_even` is a trivial example about even and odd numbers. `wicked_oe` is an extension which adds a call to each clause and 4 functors which are irrelevant for success or failure. It allows to see the effect of this on the search space. `appendlast, reverselast, nreverselast` and `schedule` are small but hard examples from the domain of program specialisation. The queries express program properties (which have to be recognised by a specialiser to derive the empty program). `multiseto` and `multisetl` are programs to check the equivalence of two multisets, the first uses a binary operator "o" to build sets, the second uses a list representation. The others are typical examples from a large set of planning problems reasoning on multisets of resources. The first two use the "o" representation for the multiset, the next two the "l" representation. `blockpair2o` and `blockpair2l` omit the for success or failure irrelevant argument for collecting the plan (and have 6 functors less). `blocksol` is there to show what happens when the query does not fail.[9]

The abductive systems are implemented in Prolog. The queries have been executed with Prolog by BIM on a SUN sparc Ultra-2. The constraint system is also written in Prolog, uses the SICSTUS finite domain solver and was running under SICSTUS Prolog [18] on the same machine. FINDER is implemented in C and was running on a IBM RS6000. Regular approximations (RA) were computed with a system due to John Gallagher, conjunctive partial deduction (CPD) with a system due to Michael Leuschel. Witold Charatonik was so kind to run our examples on a tool (some info can be found in [5]) for set based analysis (SBA) he developed together with Harald Ganzinger, Christoph Meyer, Andreas Podelski and Christoph Weidenbach.

[6] The code is available at http://www.cs.kuleuven.ac.be/~henkv/pre

[7] n facts and m domain elements yields $m^n/m!$ different pre-interpretations.

[8] With arity k, a predicate can have $2^{(m^k)}$ different interpretations.

[9] One should in parallel run the query and interrupt the search for failure when a solution is found.

name	domain	functor	abduced	predicates	clauses	arity 1	arity 2	arity 3
odd_even	2	2	3	2	3	3	0	0
wicked_oe	2	6	10	3	4	1	3	0
appendlast	3	4	12	2	4	0	2	2
reverselast	3	4	12	2	4	0	2	2
nreverselast	5	4	28	3	6	0	4	2
schedule	3	4	12	6	12	9	3	0
multiseto	2	4	7	1	7	0	7	0
multisetl	2	4	7	2	4	0	2	2
blockpair2o	2	9	19	3	15	0	15	0
blockpair3o	2	15	36	3	15	0	7	8
blockpair2l	2	9	19	5	14	0	8	6
blockpair3l	2	15	36	5	14	0	0	14
blocksol	2	9	19	5	14	0	0	14

Table 2. Properties of benchmark programs

Table 3 gives the results: For the abductive system the time and the number of backtracks (wrt. the choice made in the pre-interpretation); for the constraint system the time, the number of backtracks (wrt. the choices regarding tabulation) and the number of consistency checks (each check verifies the existence of a pre-interpretation satisfying all constraints and has internal backtracking); for FINDER the time and the number of backtracks. For CPD, RA and SBA we only indicate whether failure of the query follows from the result (It makes little sense to give times as these systems do not perform an exhaustive search and no sense to try `blocksol`.). If followed by H, time is in hours, otherwise in seconds.

	AB		CS			FINDER		CPD	RA	SBA
name	time	#bckt	time	#bckt	#cstch	time	#bckt			
odd_even	0.00	4	0.00	0	2	0.02	1	yes	yes	yes
wicked_oe	0.08	64	0.00	0	2	0.02	64	yes	yes	yes
appendlast	6.63	949	0.45	1	6	0.83	19023	yes	no	yes
reverselast	9.37	1267	3.70	2	9	1590	94583354	no	no	yes
nreverselast	>19H	-	>19H	-	-	>15H	$> 100.10^6$	yes	no	no
schedule	0.11	33	0.31	1	10	0.07	508	no	no	yes
multiseto	0.05	12	0.04	0	6	0.47	2849	no	yes	yes
multisetl	0.01	3	0.06	1	8	77.10	2583088	yes	no	yes
blockpair2o	3.36	103	0.38	0	12	112	1359532	no	no	no
blockpair3o	639.05	97246	0.42	0	12	>1.65H	$> 10.10^6$	no	no	no
blockpair2l	0.9	34	2.36	2	17	>1.41H	$> 10.10^6$	no	no	no
blockpair3l	2.46	162	2.49	2	17	>3.80H	$> 10.10^6$	no	no	no
blocksol	293.33	12832	4.48H	299	609	>1.01H	$> 10.10^6$	-	-	-

Table 3. Results.

6.1 Discussion

The abductive system is rather sensitive to the presence of functors which are irrelevant to the existence of a solution (but which occur in terms met during top-down execution) as a comparison of `odd_even` and `wicked_oe`, of `blockpair2o` and `blockpair3o` and of `blockpair2l` and `blockpair3l` shows. When larger domain sizes are needed, the size of the search space increases dramatically, so it is hardly a surprise that it fails on `nreverselast`. It did surprisingly well in showing that there is no solution based on a 2-element domain for `blocksol`.

The constraint system is doing consistently well on all planning problems which fail. Its performance is unaffected by the presence of irrelevant functors. It has serious problems with `blocksol` where it backtracks a lot and has to do a big number of consistency checks before having exhausted the search space (not a real problem, one should run the query and find a solution). More serious is the problem it has with `nreverselast` where one needs to distinguish 5 different kinds of lists. The resulting search-space is of the order $5^{28} = 4 * 10^{20}$. The constraints generated by this example, while correct, does not seem to prune the search-space a lot. Adding redundant but stronger constraints will be needed to avoid the current trashing behaviour.

FINDER, which has no equivalent of a top-down strategy and of tabling, is doing poor on these problems. It behaves worse than our first abductive procedure. FINDER's input being sorted, it is possible to associate different sorts with the different functors and different domain sizes with different sorts. Very recently, experimenting with this feature and using the intuitively right domain sizes for the sort "list" (3 for `reverselast` and 5 for `nreverselast`) and 2 for all other types (also for a functor which maps two lists to a pair[10]), we succeeded in finding a solution in respectively 0.2s with 2738 backtracks (`reverselast`) and 18H with 111.10^6 backtracks (`nreverselast`). This suggests that separate types and separate domains for different functors could also restrict the search space in our systems. We plan to explore this in the future.

Conjunctive partial deduction can handle some of the problems which are difficult for us (and a planned extension [11] likely even more), but cannot handle any of the planning problems. Computing regular approximations is fast, but it can show failure of the most simple problems only. The set based analyser is more precise and fails only on the planning problems and `nreverselast`.

7 Conclusion

For definite logic programs, we have addressed the problem of proving that certain queries cannot succeed with an answer. A problem which is particularly

[10] It is not obvious from the problem formulation that this latter can result in a solution. But it drastically reduces the search space: with m domain elements, the natural thing is that the sort of pair has m^2 domain elements.

[11] A planned extension of CPD should be able to handle also `reverselast` and `multiseto`.

relevant when the query does not fail finitely. We have developed two new approaches which aim at searching a model of the program in which the query is false. We have performed some experiments using (rather small) example programs and queries which do not terminate[12]. We also did a comparison with other approaches which could be used to tackle this problem: a general purpose model generation tool which does not allow the user to control the search, the use of type inference, and the use of program specialisation. In the case of type inference, the approach is in fact also to compute a model. However, the chosen model is the one which best reflects the type structure of the program. If the query happens to be false in this model, then failure is shown. Also in the case of program specialisation, showing failure is a byproduct of the approach: for some queries, the program happens to be specialised into the empty program.

Abduction is a very powerful and general problem solving technique. It was pretty easy to formulate the problem of searching a pre-interpretation such that the query is false in the least model based on it as an abductive problem and to use a general purpose abductive procedure[9]. But we quickly realised that we had almost no control over the search for a solution. Our first approach was to built a special purpose abductive procedure for definite programs which employs tabulation and which hard wired the constraints that pre-interpretation of functors are total functions. The idea behind the proof procedure is to use a top-down evaluation strategy —abducing a part of the pre-interpretation only when needed in evaluating the query— and to prevent looping by the use of tabulation. Experiments confirmed our intuition that it was important to delay the abduction of new components in the pre-interpretation as long as possible (to propagate all consequences of what was already abduced to check whether it was part of a feasible solution). The system was performing in general much better than FINDER which is at least an order of magnitude faster in raw speed than ours. This suggests that our basic strategy —using a goal directed proof procedure and using tabulation— is a good one. However, the results were not really convincing. We tried to delay the choice of components in the pre-interpretation even more: we considered them as constraints. This allowed to delay the decisions up to the point were answers had to be tabled: at such a point one need to know whether the answer is new or not. Still we do not fix the pre-interpretation at such a point but formulate constraints on the pre-interpretation, using a finite domain solver to check the existence of a pre-interpretation satisfying all constraints. With this approach, the overall speed crucially depends on the speed by which consistency can be checked. We were coding this consistency check as a finite domain problem and obtained quite impressive results. Still the system has some weaknesses. It starts to slow down when it needs a lot of backtracking over its decisions with respect to new answers being subsumed by the existing ones or not. The number of possible backtracks quickly goes up with the arity of the predicates, as the example `blocksol`, where the query does not fail, illustrates. It also increases quickly with the size of the domain needed to show failure e.g.

[12] These programs also loop when using tabulation or when executing bottom-up after a magic set transformation.

`nreverselast`[13]. Further work is possible to reduce this number of backtracks: tabling one predicate in each strongly connected component of the program is enough to prevent looping of the procedure. This can reduce the number of choice points. Still it is desirable to find ways to extract more knowledge from the problem to further prune the search.

In general, our approach is doing much better than the general purpose model generation tool FINDER and type inference based on Regular Approximations. The comparison with conjunctive partial deduction is less straightforward. Both approaches seem to be good at a different classes of problems. In fact it could be interesting to apply program specialisation as a pre-processing step: the specialisation may reduce the number of predicates and the number of functors, making the problem easier to solve by our tool (and in extreme cases making it a trivial problem by returning the empty program).

In a broader context, this paper makes contributions to the following topics:

- A (first) study of methods to prove (infinite) failure of definite logic programs.
- The development of a proof procedure which combines tabulation with abduction and of a constraint based procedure which treats the abducibles as constraints and uses a constraint solver to check the existence of a solution for the abducibles. Also the latter procedure uses tabulation.
- A better understanding of the power and limitations of abduction. While very expressive, our findings suggests that abductive procedures need to be augmented with "background" knowledge to direct the search for abductive solutions. Simply specifying the properties of an abductive solution as an integrity constraint cannot provide sufficient guidance to the search for a solution. It is interesting to observe that background knowledge is also often the key to success in Inductive Logic Programming which makes use of inductive procedures which are in more than one respect "twins" of abductive procedures [1].
- The further development of model based program analysis. [10] showed that model based program analysis pioneered by [6] is also an excellent method for type inference. In [10] it was shown that there exist pre-interpretations which encode various other declarative properties of programs. Our work takes this work one step further by developing methods for automatically constructing a pre-interpretation which expresses a particular program property (expressed as a query which should fail).

A possible extension of this work is for programs with negation. Transformations are known which preserve a 3-valued completion semantics and transform a program with negation into a definite program, having for each predicate p/n two definitions, one for p/n and one for $p*/n$. It seems feasible to apply our method on this transformed program.

[13] Some problems require an infinite domain, e.g. `less(N,s(N)). less(N,s(M))<-less(N,M).` and the query `?- less(N,M),less(M,N).`.

References

1. H. Ade and M. Denecker. Abductive inductive logic programming. In *Proc. IJCAI'95*, pages 201–1209. Morgan Kaufman, 1995. 132
2. K.R. Apt, R.N. Bol, and J.W. Klop. On the safe termination of Prolog programs. In *Proc. ICLP'89*, pages 353–368, Lisbon, june 1989. MIT Press. 118
3. R.N. Bol, K.R. Apt, and J.W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, august 1991. 118
4. M.P. Bonacina and J. Hsiang. On rewrite programs: semantics and relationship with Prolog. *J. Logic Programming*, 14(1&2):155–180, october 1992. 118
5. W. Charatonik and A. Podelski. Directional type inference for logic programs. In *Proc. SAS'98*, LNCS, Pisa, Italy, 1998. To appear. 119, 128
6. M. Codish and B. Demoen. Analysing logic programs using "prop"-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, December 1995. 120, 132
7. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *J. Logic Programming*, 19 & 20:199–260, may/july 1994. 118
8. D. A. de Waal, M. Denecker, M. Bruynooghe, and M. Thielscher. The generation of pre-interpretations for detecting unsolvable planning problems. In *Proc. IJCAI Workshop on Model based Automated reasoning*, pages 103–112, 1997. 119
9. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *J. Logic Programming*, 34(2):201–226, Februari 1998. 121, 131
10. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pages 351–365, Portland, Oregon, December 1995. MIT Press. 120, 132, 132
11. J. P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs,. In *Proc. ICLP94*, pages 599–613. MIT Press, 1994. 119, 127
12. N. Heintze. *Set based program analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. 127
13. Michael Leuschel, Danny De Schreye, and André de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In *Proc. JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. 119, 127
14. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. SIGMOD 1994 Conf. ACM*. Acm Press, 1994. 118, 119, 120, 120
15. Y. D. Shen. An extended variant of atoms loop check for positive logic programs. *New Generation Computing*, 15(2):187–203, 1997. 118
16. J. Slaney. Finder: Finite domain enumerator system description. Technical Report TR-ARP-2-94, Centre for Information Science Research, Australian National University, Australia, 1994. Also in Proc. CADE-12. 119, 127, 127
17. J. Slaney. Finder - finite domain enumerator - version 3.0 - notes and guide. Technical report, Centre for Information Science Research, Australian National University, Australia, 1997. 119, 127
18. Swedish Institute of Computer Science. *SICSTUS Prolog User's Manual*, november 1997. 128
19. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings ICLP'86*, LNCS, pages 84–98, London, 1986. Springer-Verlag. 119, 120
20. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989. 119
21. David S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111, March 1992. 119

Staging Static Analyses Using Abstraction-Based Program Specialization

John Hatcliff*, Matthew Dwyer**, and Shawn Laubach

Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA
{hatcliff,dwyer,laubach}@cis.ksu.edu

Abstract. Conventional partial evaluators specialize programs with respect to concrete values, but programs can also be specialized with respect to abstractions of concrete values. We present a novel method for staging static analyses using abstraction-based program specialization (ABPS). Building on earlier work by Consel and Khoo and Jones, we give an ABPS system that serves as a formal foundation for a suite of analysis and verification tools that we are developing for Ada programs. Our tool set makes use of existing verification packages. Currently many programs must be hand-transformed before they can be submitted to these packages. We have determined that these hand-transformations can be carried out automatically using ABPS. Thus, preprocessing programs using ABPS can significantly extend the applicability of existing tools without modifying the tools themselves.

1 Introduction

Conventional program specialization systems [14] specialize programs with respect to concrete values. For example, given a program P that takes a collection of data items c and some instructions i to perform on each data item, if P will often be called with a particular collection $\{d_1, d_2, d_3\}$, one might use a specialization system to customize P with respect to $\{d_1, d_2, d_3\}$.

There are many situations where certain properties of expected input data are known even when the actual concrete values are not known. For example, one might know that P is usually called with collections of size 3 even though concrete items d_1, d_2, d_3 are not known. Collection size can be viewed as a particular abstraction of collections. Optimizations based on abstractions (such as unrolling a loop that walks over the collection) can be performed by propagating abstract values — in essence, by combining specialization with an abstract interpretation instead of a concrete interpretation. Unfortunately, the most sophisticated specialization systems [1,6] specialize programs only with respect to concrete values and not abstractions.

* Supported in part by NSF under grant CCR-9701418 and NASA under award 21209.
** Supported in part by NSF and DARPA under grants CCR-9633388, CCR-9703094, CCR-9708184 and NASA under award 21209.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 134–152, 1998.

Many researchers (*e.g.*, [7,13,16]), have recognized the utility of abstraction-based specialization (ABPS). In particular, Consel and Khoo [7] and Jones [13] have developed formal frameworks to support the idea for functional and imperative languages, respectively. Despite these significant efforts, the technology has not been incorporated into full-fledged implementations. It seems that there have not been enough interesting applications to justify the cost of extending systems to include ABPS.

1.1 Applying Abstraction-Based Program Specialization

We are building a suite of tools for analysis and verification of software systems written in Ada. Several of our tools use existing analysis tools [9,12,17] that vary in sophistication. These existing tools often require programs to be hand-transformed before analysis. The purpose of many of these translations is to reduce the state space to be explored by the analysis tool. For example, when verifying properties of a protocol control logic, one might hand transform the non-control data that transmits through a software communication system to a single dummy value, thereby simplifying the work of the following analysis.

We have determined [8] that many of these transformations can be automated using abstraction-based program specialization. In essence, ABPS can be used to stage analyses by carrying out part of an analysis at specialization time, while an existing tool carries out the remainder of the analysis. This strategy allows us to significantly extend the applicability of existing analysis/verification tools without having to modify the tools themselves.

The goal of this paper is to give a formal justification for staging analyses (as required by our tools) using abstraction-based program specialization. This formal presentation provides the foundation for a full-scale ABPS system currently being implemented for a large subset of Ada.

1.2 Related Work

Since we rely heavily on ideas of Consel and Khoo [7] and Jones [13], we briefly summarize their work below, and describe how we modify and extend it to satisfy our requirements.

Consel and Khoo [7] give a framework for abstraction-based partial evaluation of first-order functional programs. The system is parameterized on algebras that define the meaning of constants and primitive operations in the language. Both online and offline as well as concrete and abstraction-based partial evaluation can be obtained through a suitable choice of algebras. A denotational semantics framework is used to establish correctness, and the safety relationship between a concrete and abstract algebra is expressed using a family of logical relations. The framework was implemented for first-order ML [5,15].

Jones [13] provides an elegant language-independent framework for describing partial evaluation and supercompilation. An abstract property is represented by the set of values abstracted by the particular property. For example, the property "x is even" is represented by "$x \in \{0, 2, 4, \ldots\}$". The meaning of programs

is given using a transition relation on states (l, σ) where l is the label of a program point and σ is the current store. Correctness of specialization is expressed by showing a correspondence between the transition relations of the source and residual programs. Although Jones's framework is language-independent, he gives examples using a simple flowchart language and outlines an algorithm for supercompilation of flowchart programs.

Other elements in our work are inspired by Ashley and Consel's mechanisms for controlling polyvariance and generalization in flow analyses [3,2], Glück and Klimov's presentation of supercompilation [10], Schmidt's presentation of abstract interpretation [18], and Sørensen and Glück's work on generalization for tree-structured data [19].

1.3 Contributions and Organization

To obtain a framework for staging analyses, we recast in an imperative language setting Consel and Khoo's idea of parameterization using algebras, and we adopt Jones's presentation of semantics using transition relations.

Below we list the novel aspects of our ABPS system. These stem from our applications to analysis and verification, and our desire to handle large imperative programs.

Preservation of abstract semantics: Program specialization has always sought to preserve a program's concrete (*i.e.*, execution) semantics. In our target applications, residual programs will be analyzed instead of executed. *Thus specialization need not preserve execution semantics, but only abstract semantics.* This allows more specialization to occur since fewer operations need to be residualized. For example, if one already knows the abstract value resulting from an operation, the abstract value itself can be residualized. It is not necessary to residualize the operation to compute a concrete value at run-time. "Run-time" becomes "analysis-time", and at analysis time one is only interested in abstractions. Our system is parameterized on operations that allows the user to determine if concrete or abstract semantics is to be preserved.

Control of polyvariance: In both the Consel and Khoo and Jones presentations, specialization is *maximally polyvariant* — specialized code is generated for each unique specialization-point/context pair or *state*. While this gives maximum specialization, it can lead to residual programs that are quite large (and sometimes specialization doesn't terminate). Since we will be working on large programs, it is important to give the user some control over the degree of polyvariance. For example, one might want only monovariant specialization for some program points, or only polyvariance on a particularly relevant subset of program variables. We introduce a mechanism for this.

Control of generalization: To avoid infinite specialization (and too much polyvariance), one must have a mechanism for merging multiple computational states $s_1, s_2, \ldots$ to obtain a single state s that covers each s_i. We parameterize the system on a widening operator θ that defines the merging of states. Generating residual code in conjunction with systematic generalization is more complicated than in convential specializers. For example, one may generate residual code for

Syntax Domains

$p \in \text{Programs}[\Sigma]$ $e \in \text{Expressions}[\Sigma]$ $j \in \text{Jumps}[\Sigma]$
$b \in \text{Blocks}[\Sigma]$ $a \in \text{Assignments}[\Sigma]$ $o \in \text{Operations}[\Sigma]$
$l \in \text{Block-Labels}[\Sigma]$ $al \in \text{Assignment-Lists}[\Sigma]$ $t \in \text{Tests}[\Sigma]$
$x \in \text{Variables}[\Sigma]$

Grammar

$p ::= (l)\ b^+$ $a ::= x := e;$
$b ::= l : al\ j$ $e ::= x \mid o(e^*)$
$al ::= a\ al \mid \cdot$ $j ::= \textbf{goto}\ l; \mid \textbf{return}; \mid \textbf{if}\ t(x^*)\ \textbf{then}\ l_1\ \textbf{else}\ l_2;$

Fig. 1. Syntax of the Flowchart Language FCL

a particular state s only to discard the code later if s is merged with s' [19]. We incorporate a strategy for dealing with this in the context of imperative languages.

The rest of the paper is organized as follows. Section 2 presents the simple flowchart language FCL that we use for illustrating the principles of ABPS. Section 3 gives our presentation of ABPS. Section 4 shows how ABPS can be used to stage a simple static analysis. Section 5 concludes and gives plans for future research.

An implementation of the system presented here, as well as an extended version of this paper with examples of application to Ada, and more technical details, discussion, and proofs can be found off of the first author's web page (`http://www.cis.ksu.edu/~hatcliff`).

2 The Flowchart Language FCL

We present the principles of our ABPS using the simple flowchart language FCL of Gomard and Jones [11,13,14]. As they note, FCL is small enough to allow a clean semantic presentation, but rich enough conceptually to illustrate a multitude of issues associated with program specialization.

Figure 1 presents the syntax of FCL. An FCL program $(l)\ b^+$ consists of a list of *basic blocks* b^+ and l is the *label* of the initial block to be executed. Each basic block consists of a label followed a (possibly empty) list of *assignments*. Each block concludes with a *jump* that transfers control from that block.

FCL contains three kinds of jumps: an unconditional jump **goto** to label l, a conditional jump **if**, and a special jump **return** that terminates program execution. Note that we only allow variables to occur as arguments to tests. This simplifies the treatment of supercompilation-like features. This restriction can be lifted by introducing a preprocessing phase that introduces temporary variable names for test arguments. We will consider the output of the program to be the collective value of the program's variables.

The syntax of FCL is parameterized by a signature Σ that specifies a set of operator symbols Operations[Σ] and a set of of test symbols Tests[Σ]. Each operator $o \in$ Operations[Σ] has an associated arity $\mathsf{arity}(o)$ (similarly for tests). Constants (literals) are realized as 0-ary operators. We write Constants[Σ] to denote the set of constants in Operations[Σ].

Example 1. Σ_{num} is a signature for simple computation on natural numbers.

$$\begin{aligned} \text{Operations}[\Sigma] &= \{+, *, -, \mathtt{0}, \mathtt{1}, \ldots\} \\ \text{Tests}[\Sigma] &= \{\mathtt{even?}, <, \mathtt{equal?}\} \end{aligned}$$

where numerals have arity 0, `even?` has arity 1, and all others have arity 2.

The meaning of a Σ-program is parameterized by a Σ-algebra A that provides an interpretation for the signature Σ. A Σ-algebra A consists of a carrier set Values[A] (*e.g.*, an upper semi-lattice) with partial order $\sqsubseteq_A$, sets Operations[A] and Tests[A] that contain functions implementing the operations and tests of Σ, and a map $[\![\cdot]\!]^{\Sigma}_{A}$ that maps each operation and test symbol in Σ to the corresponding implementation in Operations[A] and Tests[A]. For simplicity, we sometimes omit the map $[\![\cdot]\!]^{\Sigma}_{A}$ and write o_A for the operation in A associated with symbol o (simimarly for tests). We will define the semantics of programs so that one can obtain a *concrete interpretation* of a program by plugging in an Σ-algebra A_{con} and an *abstract interpretation* by plugging in a suitable Σ-algebra A_{abs}.

Figure 2 formalizes the semantics of a Σ-program with respect to a Σ-algebra A in terms of *traces*. A big-step semantics is used to define the evaluation of expressions, assignments, and jumps. A small-step semantics is used to define the transitions in traces. Intuitively, a trace shows the transitions

$$(l_0, \sigma_0) \;\rightarrow\; (l_1, \sigma_1) \;\rightarrow\; (l_2, \sigma_2) \;\rightarrow\; \ldots$$

a program can make between *computational states* $(l_i, \sigma_i) \in$ States[A] where $l_i \in$ Labels[Σ] is the label of current basic block and $\sigma_i \in$ Stores[A] is the current store (memory). A store $\sigma \in$ Stores[A] is partial function from Variables[Σ] to Values[A]. A store σ is p-compatible when it is defined on all variables occuring in program p and undefined otherwise. We write $dom(\sigma)$ for the set of defined variables in the domain of σ.

A concrete interpretation typically will generate linear (non-branching) traces (although, our definition of tests is general enough to allow non-deterministic choice). An abstract interpretation typically will generate branching traces (a computation tree) since some tests will not be decidable due to incomplete information.

To represent terminal states, we add a special label halt to Block-Labels[Σ] to obtain the set Labels[Σ]. All finite branches of a trace will end in a state (halt, σ) for some store σ.

A program is represented using a partial function Γ called a *block map* that maps a label $l \in$ Block-Labels[Σ] to a block $b \in$ Blocks[Σ]. Intuitively, Γ will be defined for exactly the labels that name blocks in the program being interpreted.

Semantic Domains

$$v \in \text{Values}[A] \qquad o_A \in \text{Operations}[A] \qquad t_A \in \text{Tests}[A]$$

$$\begin{array}{lll} \sigma & \in \text{Stores}[A] & = \text{Variables}[\Sigma] \rightharpoonup \text{Values}[A] \\ l & \in \text{Labels}[\Sigma] & = \text{Block-Labels}[\Sigma] \cup \{\mathsf{halt}\} \\ \Gamma & \in \text{Block-Maps}[\text{FCL}] & = \text{Block-Labels}[\Sigma] \rightharpoonup \text{Blocks}[\Sigma] \end{array}$$

Expressions

$$\frac{}{\sigma \Vdash_{expr} x \Rightarrow \sigma(x)} \qquad \frac{\sigma \Vdash_{expr} e_i \Rightarrow v_i \qquad o_A(v_1 \ldots v_n) = v}{\sigma \Vdash_{expr} o(e_1 \ldots e_n) \Rightarrow v}$$

Assignments

$$\frac{\sigma \Vdash_{expr} e \Rightarrow v}{\sigma \Vdash_{assign} x := e; \Rightarrow \sigma[x \mapsto v]} \qquad \frac{}{\sigma \Vdash_{assigns} \cdot \Rightarrow \sigma}$$

$$\frac{\sigma \Vdash_{assign} a \Rightarrow \sigma' \qquad \sigma' \Vdash_{assigns} al \Rightarrow \sigma''}{\sigma \Vdash_{assigns} a\ al \Rightarrow \sigma''}$$

Jumps

$$\frac{}{\sigma \Vdash_{jump} \textbf{goto } l; \Rightarrow \{(l, \sigma)\}} \qquad \frac{}{\sigma \Vdash_{jump} \textbf{return}; \Rightarrow \{(\mathsf{halt}, \sigma)\}}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}}{\sigma \Vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}} \quad \text{where } n \in \{1, 2\}$$

Transitions

$$\frac{\Gamma(l) = l : \; al\ j \qquad \sigma \Vdash_{assigns} al \Rightarrow \sigma' \qquad \sigma' \Vdash_{jump} j \Rightarrow \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}}{\Sigma, A \Vdash_{\Gamma} (l, \sigma) \rightarrow (l'_i, \sigma'_i) \;\; \forall i \in \{1, \ldots, n\}}$$

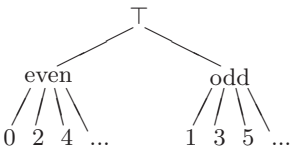
Fig. 2. Trace semantics of Σ-programs with respect to Σ-algebra A

The rule for deriving transitions depends on rules for each of the syntactic categories of FCL (expressions, assignments, jumps). The only noteworthy aspect of these rules is the parameterization on operations and tests. For each $o \in \text{Operations}[\Sigma]$, $\text{Operations}[A]$ contains a monotonic function $o_A \in \text{Values}[A]^{\mathsf{arity}(o)} \rightarrow \text{Values}[A]$. For each $t \in \text{Tests}[\Sigma]$, $\text{Tests}[A]$ contains a function

$$t_A \in \text{Values}[A]^{\mathsf{arity}(t)} \times \text{Block-Labels}[\Sigma] \times \text{Block-Labels}[\Sigma] \times \text{Stores}[A] \rightarrow \wp(\text{States}[A]).$$

In a concrete interpretation, a test t_A will usually be defined so that $t_A(x^*, l_1, l_2, \sigma)$ yields $\{(l_1, \sigma)\}$ if the test succeeds and $\{(l_2, \sigma)\}$ otherwise. In an abstract interpretation, there may not be sufficient information to determine the appropriate branch, and one would have $t_A(x^*, l_1, l_2, \sigma)$ yields $\{(l_1, \sigma_1), (l_2, \sigma_2)\}$. Allowing σ_1 and σ_2 to be different from σ lets us describe situations where in-

Carrier



Operations (excerpts)

$$add(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} n_1 + n_2 & \text{if } v_1 = n_1 \text{ and } v_2 = n_2 \\ even & \text{if } \neg\exists n_1, n_2 \text{ such that } v_1 = n_1 \text{ and } v_2 = n_2 \text{ and} \\ & \quad v_1 \sqcup v_2 \in \{even, odd\} \\ odd & \text{if } \neg\exists n_1, n_2 \text{ such that } v_1 = n_1 \text{ and } v_2 = n_2 \text{ and} \\ & \quad v_1 \sqcup even = even \text{ and } v_2 \sqcup odd = odd, \text{ or} \\ & \quad v_1 \sqcup odd = odd \text{ and } v_2 \sqcup even = even \\ \top & \text{otherwise} \end{cases}$$

Tests (excerpts)

$$even?(x, l_1, l_2, \sigma) \stackrel{\text{def}}{=} \begin{cases} \{(l_1, \sigma)\} & \text{if } \sigma(x) \sqcup even = even \\ \{(l_2, \sigma)\} & \text{if } \sigma(x) \sqcup odd = odd \\ \{(l_1, \sigma[x \mapsto even]), & \\ \ (l_2, \sigma[x \mapsto odd])\} & \text{otherwise} \end{cases}$$

$$equal?(x_1, x_2, l_1, l_2, \sigma) \stackrel{\text{def}}{=} \begin{cases} \{(l_1, \sigma)\} & \text{if } \exists n \,.\, \sigma(x_1) = \sigma(x_2) = n \\ \{(l_2, \sigma)\} & \text{if } \neg\exists v \,.\, \sigma(x_1) \sqcap \sigma(x_2) = v \\ \{(l_1, \sigma[x_1 \mapsto v, x_2 \mapsto v]), & \\ \ (l_2, \sigma)\} & \text{if } \exists v \,.\, \sigma(x_1) \sqcap \sigma(x_2) = v \\ & \text{and } \neg\exists n \,.\, (\sigma(x_1) = \sigma(x_2) = n) \end{cases}$$

Fig. 3. Abstract algebra A_{eo} for even/odd abstract interpretation

formation about variables x^* increases as conditionals are crossed as in *e.g.*, supercompilation.

We now consider some example algebras. Let A_{num} be the expected concrete algebra associated with Σ_{num} (Values$[A_{\text{num}}] = \{0, 1, 2, \ldots\}$). Figure 3 presents an Σ_{num}-algebra A_{eo} designed for an odd/even abstract interpretation of Σ_{num}-programs. For techniques to relate concrete and abstract algebras, we refer the reader to the extended version of this paper.

3 Abstraction-Based Specialization

Our abstraction-based specialization framework combines the trace generation system from the previous section with *code generation*. The idea is to carry out the trace while simultaneously generating code that is specialized with respect to the information accumulated in the trace.

Semantic Domains

$$v \in \text{Values}[A] \qquad o_A \in \text{Operations}[A] \qquad t_A \in \text{Tests}[A]$$

$$\begin{array}{lll} w \in \text{Spec-Values}[A] & = \text{Values}[A] \times \text{Expressions}[\Sigma] \\ \sigma \in \text{Stores}[A] & = \text{Variables}[\Sigma] \rightharpoonup \text{Values}[A] \\ l \in \text{Labels}[\Sigma] & = \text{Block-Labels}[\Sigma] \cup \{\mathsf{halt}\} \\ \Gamma \in \text{Block-Maps[FCL]} & = \text{Block-Labels}[\Sigma] \rightharpoonup \text{Blocks}[\Sigma] \end{array}$$

Expressions

$$\frac{}{\sigma \vdash_{expr} x \Rightarrow \langle\, \sigma(x)\,,\, x \,\rangle}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow \langle\, v_i\,,\, e_i' \,\rangle \qquad o_A(v_1 \ldots v_n) = v \qquad v \in R}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle\, v\,,\, \mathit{lift}(v) \,\rangle}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow \langle\, v_i\,,\, e_i' \,\rangle \qquad o_A(v_1 \ldots v_n) = v \qquad v \notin R}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle\, v\,,\, o(e_1' \ldots e_n') \,\rangle}$$

Assignments

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle\, v\,,\, e' \,\rangle}{\sigma \vdash_{assign} x := e; \Rightarrow \langle\, \sigma[x \mapsto v]\,,\, [x := e';] \,\rangle} \qquad \frac{}{\sigma \vdash_{assigns} \cdot \Rightarrow \langle\, \sigma\,,\, [] \,\rangle}$$

$$\frac{\sigma \vdash_{assign} a \Rightarrow \langle\, \sigma'\,,\, al' \,\rangle \qquad \sigma' \vdash_{assigns} al \Rightarrow \langle\, \sigma''\,,\, al'' \,\rangle}{\sigma \vdash_{assigns} a\ al \Rightarrow \langle\, \sigma''\,,\, al' +\!\!+ al'' \,\rangle}$$

Fig. 4. Abstraction-based specialization (part 1)

Figures 4 and 5 present the abstraction-based program specializer. The specializer is parameterized on a *specialization structure*

$$\Xi \;=\; (\Sigma, \Sigma_{res}, A, \pi, \theta, R, \mathit{lift}).$$

- Σ is the signature of the program being specialized.
- Σ_{res} is the signature of the residual program. If abstract tokens (*e.g.*, *even*, *odd*) are being residualized, Σ_{res} will differ from Σ (*e.g.*, Σ_{res} will contain constants `even`, `odd`).
- Programs will be specialized with respect to Σ-algebra A.
- π controls the degree of polyvariance by specifying which states are to be merged.
- θ is a widening operator used to merge stores.
- R is the set of values from Values$[A]$ that can be residualized.
- *lift* generates code for values $v \in R$, *i.e.*, it maps a residualizable value v to a constant in Σ_{res}.

We explain each of these components and constraints they must satisfy in the subsections below.

Jumps

$$\frac{}{\sigma \vdash_{jump} \textbf{goto } l; \Rightarrow \langle\, \{(l,\sigma)\}\,,\, \textbf{goto } \pi(l,\sigma);\,\rangle}$$

$$\frac{}{\sigma \vdash_{jump} \textbf{return}; \Rightarrow \langle\, \{(\mathsf{halt},\sigma)\}\,,\, \textbf{return};\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_1, \sigma')\}}{\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \langle\, \{(l_1,\sigma')\}\,,\, \textbf{goto } \pi(l_1,\sigma');\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_2, \sigma')\}}{\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \langle\, \{(l_2,\sigma')\}\,,\, \textbf{goto } \pi(l_2,\sigma');\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_1, \sigma_1'), (l_2, \sigma_2')\}}{\begin{array}{l}\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \\ \qquad \Rightarrow \langle\, \{(l_1,\sigma_1'), (l_2,\sigma_2')\}\,,\, \textbf{if } t(x^*) \textbf{ then } \pi(l_1,\sigma_1') \textbf{ else } \pi(l_2,\sigma_2');\,\rangle\end{array}}$$

Blocks

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\, al_1\,\rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{(l_{2_i}, \sigma_{2_i}) \;\mid\; i \in \{1,\ldots,n\}\}\,,\, j_2\,\rangle\end{array}}{\sigma \vdash_{block} l\colon al\; j \Rightarrow \langle\, \{(l_{2_i}, \sigma_{2_i}) \;\mid\; i \in \{1,\ldots,n\}\}\,,\, (l,\sigma)\colon al_1\; j_2\,\rangle}$$

Specialization steps

$$\frac{\mathcal{C}_0(\iota) \vdash_{block} \Gamma(\pi^{-1}(\iota)) \Rightarrow \langle\, \{(l_i', \sigma_i') \;\mid\; i \in \{1,\ldots,n\}\}\,,\, b'\,\rangle}{\vdash_\Gamma \langle \mathcal{S}\,,\, \mathcal{C}_0\,,\, \Gamma_{\mathrm{R}}\rangle \longmapsto \langle \mathcal{S}_n\,,\, \mathcal{C}_n\,,\, \Gamma_{\mathrm{R}}[\iota \mapsto b']\rangle} \qquad \text{if } \iota^\circ = \mathit{first}(\mathcal{S}_0)$$

where

$$\iota_i = \pi(l_i', \sigma_i') \qquad \text{for } i \in \{1,\ldots,n\}$$

$$\mathcal{C}_i = \left\{\begin{array}{ll}\mathcal{C}_{i-1}[\iota_i \mapsto \theta(\sigma_i', \mathcal{C}_{i-1}(\iota_i))] & \text{if } \mathcal{C}_{i-1}(\iota_i) \text{ is defined} \\ \mathcal{C}_{i-1}[\iota_i \mapsto \sigma_i'] & \text{if } \mathcal{C}_{i-1}(\iota_i) \text{ is undefined}\end{array}\right\} \qquad \text{for } i \in \{1,\ldots,n\}$$

$$\mathcal{S}_0 = \mathit{remove\text{-}arcs}(\mathit{mark}(\mathcal{S}, \iota), \iota)$$

$$\mathcal{S}_i = \left\{\begin{array}{ll}\mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota, {\iota_i}^\bullet) & \text{if } \iota_i = \mathsf{halt} \\ \mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota, {\iota_i}^\circ) & \text{if } \iota_i \text{ not in } \mathcal{S}_{i-1} \text{ and } \iota_i \neq \mathsf{halt} \\ \mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota_i, {\iota_i}^m) & \text{if } {\iota_i}^{m'} \text{ in } \mathcal{S}_{i-1} \text{ and } \iota_i \neq \mathsf{halt} \\ & \text{where } m = \circ \text{ if } \mathcal{C}_0(\iota_i) \sqsubset \mathcal{C}_n(\iota_i) \\ & \text{and } m = m' \text{ if } \mathcal{C}_0(\iota_i) = \mathcal{C}_n(\iota_i)\end{array}\right\} \qquad \text{for } i \in \{1,\ldots,n\}$$

Fig. 5. Abstraction-based specialization (part 2)

The semantic domains for the specializer have a structure similar to those of the trace generator. The key difference is that the specializer manipulates *specialization values* Spec-Values[A] instead of only Values[A]. A specialization value is pair $\langle v\,,\, e\rangle$ where $v \in$ Values[A] and e is a piece of residual code whose value is abstracted by v.

Specialization will transform a Σ-program into a Σ_{res}-program. The constants from Σ and Σ_{res} may be different. For example, if we are using the algebra A_{eo} to stage an even/odd analysis and the tokens *odd* and *even* are residualized, then $\{odd, even\} \subseteq$ Constants[Σ_{res}]. However, we require that the non-constant operations and tests be the same in Σ and Σ_{res}.

Expression specialization uses the injectitve code generation function $lift \in R \to$ Constants[Σ_{res}]. The definition of the set of residualizable values R and the definition of *lift* controls whether specialization will preserve the concrete semantics or only the abstract semantics of a program. For example, if we want to preserve the concrete semantics using the even/odd abstraction, we only allow lifting of "concrete" values:

$$\begin{aligned} R_c &= \{0, 1, 2, \ldots\} \\ lift_c(n) &= \texttt{n} \qquad\qquad \forall n \in R_c \end{aligned}$$

Using this definition and the store $\sigma \stackrel{\text{def}}{=} [x_1 \mapsto \langle\, even\,,\, x_1\,\rangle, x_2 \mapsto \langle\, even\,,\, x_2\,\rangle]$, we have $\sigma \vdash_{expr} +(x_1, x_2) \Rightarrow \langle\, even\,,\, +(x_1, x_2)\,\rangle$. Here, the value *even* cannot be residualized, so we must residualize an expression to compute a value at run-time (using the second rule for operations in Figure 5).

If we only need to preserve abstract semantics, then we can define R_a and $lift_a$ as follows:

$$\begin{aligned} R_a &= \{even, odd, 0, 1, 2, \ldots\} \\ lift_a(n) &= \texttt{n} \qquad\qquad \forall n \in \{0, 1, 2, \ldots\} \\ lift_a(even) &= \texttt{even} \\ lift_a(odd) &= \texttt{odd} \end{aligned}$$

Using this definition, we have $\sigma \vdash_{expr} +(x_1, x_2) \Rightarrow \langle\, even\,,\, \texttt{even}\,\rangle$. Here, the residual program will have a signature Σ_{res} which include constants `even` and `odd`. Abstract tokens like these are represented in Ada using enumerated types.

When preserving the concrete semantics using R_c, the `+` operation must be residualized because the concrete value of the operation cannot be determined at specialization time. When preserving the abstract semantics using R_a, the operation need not be residualized because the abstract value of the operation can be determined at specialization time. In general, specializing to preserve abstract semantics yields smaller programs since the information to computed by the residual program need not be as precise.

At the end of this section, we define a notion of *strong correctness* (where no precision is lost during specialization) and *weak correctness* (where precision may be lost during specialization). Essentially, *strong correctness conditions* will hold when R is discretely ordered and downwards closed. If R is not discretely

ordered, then precision may be lost since a less precise value may be residualized at specialization time, but a more precise value may flow into the program point at run-time. Note that R_c satisfies the strong correctness conditions, but R_a does not. To see the implications of this, consider executing the residual fragments above that resulted from specializing with respect to σ using R_c and R_a. We will use the store $\sigma' \stackrel{\text{def}}{=} [x_1 \mapsto 2, x_2 \mapsto 4]$ for executing the residual fragments (note $\sigma' \sqsubseteq \sigma$). Executing the residual fragment obtained from R_c gives $\sigma' \Vdash_{expr} +(x_1, x_2) \Rightarrow 6$ but executing the residual fragment obtained from R_a gives $\sigma' \Vdash_{expr} \texttt{even} \Rightarrow \mathit{even}$. Thus, precision has been lost in the second case. Requiring R to be discretely ordered and downwards-closed means that no value more precise than the value of residual token can flow into the relevant program point at run-time.

The previous section showed that an abstract interpretation produces a series of states (l_i, σ_i). Maximally polyvariant specialization would produce a specialized block for each of these states. Often one does not want maximal polyvariance since this would lead to a prohibitively large number of specialized blocks (or worse yet, the number of specialized blocks would be infinite).

To help avoid these problems, specialization is parameterized by a projection operator $\pi \in \text{States}[A] \to \text{Indices}$ where Indices is some (as yet) unspecified set of tokens depending on the definition of π. Intuitively, each specialized block will be labeled by an index ι, and $\pi(l, \sigma)$ yields the label of the residual block associated with (l, σ). Thus, the rules for specializing jumps use π to determine the label of the jump destinations in the residual program. As a technicality, we assume that Indices always includes an index halt and that for all stores σ, $\pi(\mathsf{halt}, \sigma) = \mathsf{halt}$.

The degree of polyvariance is controlled by mapping one or more states to the same index. For example, the following two definitions would yield a maximally polyvariant analysis and a monovariant analysis, respectively.

$$\pi(l, \sigma) \stackrel{\text{def}}{=} (l, \sigma) \quad \forall l \in \text{Block-Labels}[\Sigma], \forall \sigma \in \text{Stores}[A] \quad \text{and States}[A] \subseteq \text{Indices}$$
$$\pi(l, \sigma) \stackrel{\text{def}}{=} l \quad \forall l \in \text{Block-Labels}[\Sigma], \forall \sigma \in \text{Stores}[A] \quad \text{and Labels}[\Sigma] \subseteq \text{Indices}$$

If at least two states (l, σ) and (l, σ') map to the same index, then the associated residual block must be general enough to handle both σ and σ'. Since we intend that π merge different states of the same block l, we require π to be *label distinguishing*, *i.e.*, $\pi(l_1, \sigma_1) = \pi(l_2, \sigma_2)$ implies $l_1 = l_2$. We write $\pi^{-1}(\iota)$ to denote the unique label that π associates with ι.

There are many interesting variations between the two extremes above (maximally polyvariant and monovariant) that can be realized by π. For instance, π can easily be defined to give monovariance at some blocks, but polyvariance at others. Additionally, one can have polyvariance on some particular variables, but monovariance on others. As an example, it is useful to have monovariance on the dead variables (detected by a live variable analysis). In our full system, the user choses between several default settings, but can also attach annotations that override the defaults at each block.

To represent the structure of the residual program, the specializer incrementally constructs a control-flow graph (program skeleton) $\mathcal{S}$ with nodes $m \in$ Indices. Each node is annotated with a mark $m \in \{\circ, \bullet\}$. An unmarked node ι° indicates that the associated block is pending for specialization. A marked node $\iota^\bullet$ indicates that the associated specialized block is currently up-to-date with respect to the information determined to be flowing into the block. Intuitively, new nodes added to the graph are unmarked, and a marked node may be unmarked at a later stage due to generalization (widening). Consider a π that maps both $s = (l, \sigma)$ and $s' = (l, \sigma')$ to the same index ι. If state s is encountered first during specialization, a specialized block will be generated for it at that time. Later, if s' is encountered, node ι will need to be unmarked to instruct the specializer to reprocess the residual block associated with node ι in $\mathcal{S}$. For example, the specializer will need to create a new specialized version of block l specialized to a generalized store $\sigma \sqcup \sigma'$. Here, $\sigma \sqcup \sigma'$ represents the component-wise application of $\sqcup$.

We use the following operations to manipulate control-flow graphs.

- *mark*$(\mathcal{S}, \iota)$: returns a graph identical to $\mathcal{S}$ except that the ι is now marked (similarly for *unmark*$(\mathcal{S}, \iota)$).
- *make-arc*$(\mathcal{S}, \iota_1, {\iota_2}^m)$: returns a graph identical to $\mathcal{S}$ except that an arc from ι_1 to ι_2 is added. If the arc is already present, the set of arcs is unchanged. If node ι_2 is not already in the graph, then it is added with mark m. If node ι_2 is already in the graph, then its mark is changed to m.
- *remove-arcs*$(\mathcal{S}, \iota)$: returns a graph identical to $\mathcal{S}$ except all arcs leading out of ι are removed (the set of nodes is unchanged). If node ι is not in the graph, then $\mathcal{S}$ is returned unchanged.

A cache $\mathcal{C} \in$ Indices $\rightharpoonup$ Stores$[A]$ collects the information that can flow into each block ι in the residual program. A block-map Γ_{R} maps a label (index ι) to the associated specialized block.

According to the definition in Figures 4 and 5, the specializer repeatedly transforms a control-flow graph $\mathcal{S}$, a cache $\mathcal{C}$, and a block map Γ_{R} for the residual program. A specialization step begins by using a topological sort on $\mathcal{S}$ to find an unmarked node with no unmarked predecessors (*first*$(\mathcal{S})$ returns such a node, if it exists). Choosing a node in this manner ensures that the specializer is not wasting computation, and prevents non-terminating specialization (provided π and widening operator θ are chosen appropriately).

After a node ι° is chosen, a specialized version b' of the source block $\pi^{-1}(\iota)$ is created using the store $\mathcal{C}(\iota)$ currently held in the cache for node ι. Information is propagated by processing the set of descendent states as follows.

After obtaining an index ι_i for each state, the cache entry for ι_i is updated by merging the new store σ'_i with the previously cached store for node ι (if it exists). The merging is parameterized on a widening operator θ that must satisfy the following conditions.

1. $\sigma_1 \sqcup \sigma_2 \quad \sqsubseteq \ \theta(\sigma_1, \sigma_2)$
2. $\pi(l, \sigma) = \iota = \pi(l, \sigma')$ implies $\pi(l, \theta(\sigma, \sigma')) = \iota$
3. θ is monotonic

The second condition can be removed by giving a slightly more complicated method for generating labels for residual blocks.

The control-flow graph is updated by marking the node ι just processed. All out-going arcs of ι are removed since the current in-coming store may be less precise than previous stores that have flowed into the node (and thus the out-going arcs may change). For each successor state (l'_i, σ'_i), the corresponding index ι_i is added as a child to ι in the graph. If $\iota_i = \mathsf{halt}$, then it represents a terminal node that requires no further processing (so it is marked). If ι_i is not already in the graph, it is unmarked to signal that it has not been processed yet. If ι_i is already in the graph and if generalization caused the previously cached value to be out of date (*i.e.*, the previously cached value is strictly less than the current value), the node ι_i is unmarked to signal that the node needs to be reprocessed. If the cached value has not changed, then the node's mark is not changed.

Let $p = (l_{init})\ b^+$ be a Σ-program, Γ be the blockmap associated with p, A a Σ-algebra, and $\sigma_{init} \in \mathrm{Stores}[A]$ a p-compatible store. An *initial specialization configuration* for specializing p with respect to σ_{init} is the configuration $\langle \mathcal{S}\,, \mathcal{C}\,, \Gamma_{\mathrm{R}} \rangle$ where $\mathcal{S}$ contains only the index $\iota_{init}{}^{\circ}$ where $\pi(l_{init}, \sigma_{init}) = \iota_{init}, \mathcal{C}(\iota_{init}) = \sigma_{init}$ and is undefined for all other arguments, and Γ_{R} is undefined on all argument values. If $\vdash_{\Gamma} \langle \mathcal{S}\,, \mathcal{C}\,, \Gamma_{\mathrm{R}} \rangle \longmapsto^* \langle \mathcal{S}'\,, \mathcal{C}'\,, \Gamma_{\mathrm{R}}{}' \rangle$ where $\langle \mathcal{S}'\,, \mathcal{C}'\,, \Gamma_{\mathrm{R}}{}' \rangle$ is a terminal state (*i.e.*, no unmarked node is reachable *via first* in $\mathcal{S}'$), the residual program has initial label ι_{init} and blocks $\{\Gamma_{\mathrm{R}}{}'(\iota) \mid \forall \iota \in \mathcal{S}'$ reachable from $\iota_{init}\}$.

It is important to note that specialization with respect to algebra A is not guaranteed to terminate unless Values$[A]$ is finite. When Values$[A]$ is infinite but of finite height, defining π for maximal polyvariance can still give infinite specialization. In this case, defining π for monovariance and taking $\theta = \sqcup$ does ensure termination. For Values$[A]$ of infinite height, $\theta = \sqcup$ with π set for monovariance does not ensure termination. Instead, one must have a more sophisticated definition of θ to ensure termination.

In contrast to the Søresen and Glück generalization strategy [19] which ensures termination, we must rely on the user to give appropriate definitions of π and θ if guaranteed termination is desired. Søresen and Glück are able to ensure termination because they deal specifically with tree-structured data (*e.g.*, cons-lists) and can take advantage of technical results concerning homeomorphic embeddings.

Specialization using the specialization structure $\Xi = (\Sigma, \Sigma_{res}, A, \pi, \theta, R, \mathit{lift})$ transforms a Σ-program p_1 into a Σ_{res}-program p_2. The meaning of a p_1 is given by Σ-algebra A. The meaning of a p_2 is given by Σ_{res}-algebra A_{res} that is induced from the definition of Σ,Σ_{res}, R, and *lift*. In summary, A_{res} will be be the same as A except that the meaning of the new residual tokens in Σ_{res} is given using the inverse of *lift*.

Correctness of ABPS follows from the following theorems which state how the trace semantics of the residual program under the algebra A_{res} compares with the trace semantics of the source program under A. The properties in Theorem 1 are Jones's *conditions for a driven program* [13] recast in our setting.

Theorem 1 (strong correctness). *Let $\Xi = (\Sigma, \Sigma_{res}, A, \pi, \theta, R, lift)$ be a specialization structure, and A_{res} be the residual algebra induced from Ξ. Assume that specialization of some a Σ-program p with block map Γ produces skeleton $\mathcal{S}$, cache $\mathcal{C}$, and residual program Γ_{R}. If the strong correctness constraints are satisfied then the three following properties hold.*

- **Invariance:** *Let σ_1 be a store such that $\sigma_1 \sqsubseteq \mathcal{C}(\iota_1)$. If*

$$\Sigma_{res}, A_{res} \Vdash_{\Gamma_{\mathrm{R}}} (\iota_1, \sigma_1) \rightarrow (\iota_2, \sigma_2) \text{ then } \sigma_2 \sqsubseteq \mathcal{C}(\iota_2).$$

- **Soundness:** *If $\Sigma_{res}, A_{res} \Vdash_{\Gamma_{\mathrm{R}}} (\iota_1, \sigma_1) \rightarrow (\iota_2, \sigma_2)$ and $\sigma_1 \sqsubseteq \mathcal{C}(\iota)$, then*

$$\Sigma, A \Vdash_{\Gamma} (\pi^{-1}(\iota_1), \sigma_1) \rightarrow (\pi^{-1}(\iota_2), \sigma_2') \text{ and } \sigma_2' = \sigma_2.$$

- **Completeness:** *If $\iota_1 \in \mathcal{S}$ and $\Sigma, A \Vdash_{\Gamma} (\pi^{-1}(\iota_1), \sigma_1) \rightarrow (l_2, \sigma_2')$ and $\sigma_1 \sqsubseteq \mathcal{C}(\iota_1)$, then there exists an ι_2 such that $\Sigma_{res}, A_{res} \Vdash_{\Gamma_{\mathrm{R}}} (\iota_1, \sigma_1) \rightarrow (\iota_2, \sigma_2)$. and $\pi^{-1}(\iota_2) = l_2$ and $\sigma_2' = \sigma_2$.*

Invariance states that safety between residual program stores and computed cache values is preserved (thus, each residual block is sufficiently general for any store that can flow into it). **Soundness** states that for every transition in the residual program there is a corresponding transition in the source program. **Completeness** states that for every transition in the source program, there is a corresponding transition in the residual program.

If the strong correctness conditions do not hold, then a weak correctness holds: the **Soundness** property is dropped and the last condition in **Completeness** is weakened from $\sigma_2' = \sigma_2$ to $\sigma_2' \sqsubseteq \sigma_2$. The difference from strong correctness is that, since precision is lost, there may be branches in a residual trace that do not occur in a source trace and thus the residual program is "unsound" with respect to the source program. However, the weakened version of **Completeness** guarantees that the residual program is a valid approximation of the source program.

The key distinction between strong correctness and weak correctness is that in strong correctness, specialization does not introduce any loss of precision. This holds whether one is considering concrete interpretation or abstract interpretation. For instance, performing an abstract interpretation on the source program will give exactly the same information as performing the same abstract interpretation on the residual program since one has a mirroring of traces in the source and residual programs.

The following section gives example traces which give further intuition about the strong/weak correctness properties.

Source program

```
(b1)
b1: if equal?(x,y) then b2 else b3;
b2: y := 10;
    z := *(z,3);
    goto b4;
b3: x := +(x,2);
    y := +(*(5,x),y);
    goto b4;
b4: if <(y,x) then b1 else b5;
b5: if even?(x) then b6 else b1;
b6: return;
```

Concrete residualization program

```
(b1,[0,T])
(b1,[0,T]): if equal?(x,y)
               then b2
               else (b3,[T]);
b2:         y := 10;
            z := *(z,3);
            goto b4;
(b3,[T]):   x := +(x,2);
            y := +(*(5,x),y);
            goto b4;
b4:         if <(y,x)
               then (b1,[even,T])
               else b5;
(b1,[even,T]): if equal?(x,y)
                  then b2
                  else (b3,[T]);
b5: goto b6;
b6: return;
```

Abstract resdiualization program

```
(b1,[0,T])
(b1,[0,T]): if equal?(x,y)
               then b2
               else (b3,[T]);
b2:         y := 10;
            z := even;
            goto b4;
(b3,[T]):   x := even;
            y := +(even,y);
            goto b4;
b4:         if <(y,x)
               then (b1,[even,T])
               else b5;
(b1,[even,T]): if equal?(x,y)
                  then b2
                  else (b3,[T]);
b5: goto b6;
b6: return;
```

Fig. 6. Example specialization using ABPS

4 Staging Static Analyses Using ABPS

We illustrate ABPS by specializing the FCL program at the top of Figure 6 using the odd/even abstraction A_{eo} defined earlier. Two specializatons are performed: the first preserves concrete semantics by using lift_c of Section 3, and the second only preserves abstract semantics by using lift_a of Section 3. We have chosen the program of Figure 6 to illustrate the projection operator and generalization — only a small degree of specialization occurs here.

The first specialization uses the specialization structure $\Xi = (\Sigma_{num}, \Sigma_{num}, A_{\text{eo}}, \pi, \theta, R_c, \mathit{lift}_c)$. Since lift_c does not residual any abstractions, the signature of the source and residual programs are both Σ_{num}. We choose the projection operator π so that it illustrates several concepts at

once. Specialization is specified to be monovariant at some blocks and polvariant (to various degrees) at other blocks. The program has three variables x,y, and z so the store will have the shape $\sigma = [x \mapsto v_x, y \mapsto v_y, z \mapsto v_z]$ (abbreviated $[v_x, v_y, v_z]$).

$$
\begin{array}{ll}
\pi(b1, [v_x, v_y, v_z]) = (b1, [v_x, v_y]) & \textit{polyvariant on } x \textit{ and } y \\
\pi(b2, [v_x, v_y, v_z]) = b2 & \textit{monovariant} \\
\pi(b3, [v_x, v_y, v_z]) = (b3, [v_y]) & \textit{polyvariant on } y \\
\pi(b4, [v_x, v_y, v_z]) = b4 & \textit{monovariant} \\
\pi(b5, [v_x, v_y, v_z]) = b5 & \textit{monovariant} \\
\pi(b6, [v_x, v_y, v_z]) = b6 & \textit{monovariant}
\end{array}
$$

Thus, the abstract set of indices contains block labels, and pairs of block labels and stores. For widening, we simply define $\theta \equiv \sqcup$.

Specializing the source program to the store $\sigma_{init} = [x \mapsto 0, y \mapsto \top, z \mapsto \mathit{even}]$ results in the following control-flow graph (left) and cache (right).

$$
\begin{array}{ll}
\bullet\ (b1, [0, \top]) \ \rightarrow\ b2, (b3, [\top]) & (b1, [0, \top]) = [0, \top, \mathit{even}] \\
\bullet\ b2 \ \rightarrow\ b4 & b2 = [\mathit{even}, \mathit{even}, \mathit{even}] \\
\bullet\ (b3, [\top]) \ \rightarrow\ b4 & (b3, [\top]) = [\mathit{even}, \top, \mathit{even}] \\
\bullet\ b4 \ \rightarrow\ (b1, [\mathit{even}, \top]), b5 & b4 = [\mathit{even}, \top, \mathit{even}] \\
\bullet\ (b1, [\mathit{even}, \top]) \ \rightarrow\ b2, (b3, [\top]) & (b1, [\mathit{even}, \top]) = [\mathit{even}, \top, \mathit{even}] \\
\bullet\ b5 \ \rightarrow b6 & b5 = [\mathit{even}, \top, \mathit{even}] \\
\bullet\ b6 \ \rightarrow \mathsf{halt} & b6 = [\mathit{even}, \top, \mathit{even}] \\
\bullet\ \mathsf{halt} & \mathsf{halt} = [\mathit{even}, \top, \mathit{even}]
\end{array}
$$

These resulting structures are the same regardless of the choice of *lift* (either lift_c or lift_a). In other words, the definition of *lift* affects only the code generation, and not the information propagation.

The bottom left of Figure 6 gives the code generation resulting from the steps above using lift_c. A block with index ι is specialized with respect to $\mathcal{C}(\iota)$. In this simple example, the only specialization that takes place is the resolution of the conditional in $b5$. One might expect that the assignment `y := 10;` could be specialized away and the value `10` propagated in the store. However, doing so may cause loss of precision (concrete semantics will not be preserved) if the value of y is generalized at a later block. This is the case in our example since y will be generalized to $\top$ at $b4$. A more clever (and more complicated) treatment of assignments allows their residualization to be delayed until it is detected that a merge is occuring. If no merge occurs then the assignment will not be residualized. In addition, we do not compress **goto** transitions. This optimization can also be performed in a straightforward way, *e.g.*, in post-processing.

The bottom right of Figure 6 gives the code generation from the steps above using the specialization structure $\Xi = (\Sigma_{num}, \Sigma_{num-eo}, A_{\mathrm{eo}}, \pi, \theta, R_a, \mathit{lift}_a)$ based on lift_a. Σ_{num-eo} is identical to Σ_{num} except that it also contains constants `even` and `odd`. Here more specialization can occur because we are allowed to residualize abstractions. Even though the program with residual abstractions cannot be *executed* with the original semantics preserved, it can be *analyzed* on

a store σ where $\sigma \sqsubseteq \sigma_{init}$ using the even/odd abstraction given the residual algebra induced by A_{eo}.

In our experience, using ABPS to preserve abstract semantics often gives dramatic specialization. Many interesting verification problems require knowledge of only a small subset of a program's variable values. The irrelevant variables can be abstracted to the single point domain $\{\top\}$. Here are some examples of abstractions that we have encountered.

- When verifying properties of a queue, one may be interested only in *empty* and *non-empty* states. For this case, the data type of queues is abstracted to $\{\mathsf{empty}, \mathsf{non\text{-}empty}, \top\}$. In the residual program, queue operations are replaced by operations on an enumerated type containing tokens empty and non-empty. For occurences of top, we take advantage of *bounded static variation* of the enumerated type and introduce non-deterministic choice over empty and non-empty by inserting a conditional test.
- When manipulating a collection c, sometimes one is only concerned with whether an object d is present or absent from the collection. This is abstracted to a three-point domain similar to the case above.
- Some verification problems depend on knowing if a certain variable's value lies with in particular particular subrange of integers. If, for example, four different subranges are of interest, the one can abstract the integer type using the domain $\{\mathsf{subrange1}, \mathsf{subrange1}, \mathsf{subrange3}, \mathsf{subrange4}\}$ and residualize these similarly to the examples above.

5 Conclusion

We have given a method for staging static analyses using abstraction-based program specialization. This method significantly extends the applicability of existing analysis and verification tools (*e.g.*, SPIN [12] and SMV [4]) without modifying the tools themselves. The key insight is that the required staging can be performed as a source-to-source transformation that need only preserve abstract semantics instead of concrete semantics. This relaxing of the correctness criteria allows the specializer to be more aggressive. In constructing our ABPS system, we have merged ideas from previous work and added new mechanisms for controlling polyvariance and for generalization.

Finite-state verification systems are often applied to verify properties of reactive and concurrent systems (*e.g.*, graphical user interfaces, railway interlocking systems, and industrial control systems). We are extending the framework here to handle concurrency primitives and to handle more complicated forms of abstractions related to relational abstract interpretations.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1994. DIKU Report 94-19. 134

2. J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, pages 184–194, 1996. 136
3. J. Michael Ashley and Charles Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, September 1994. 136
4. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. 150
5. Chris Colby and Peter Lee. An implementation of parameterized partial evaluation. *Bigre Journal*, 74:82–89, 1991. 135
6. Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Proceedings of the 1996 International Seminar on Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Dagstuhl Castle, Germany, February 1996. 134
7. Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993. 135
8. Matthew Dwyer, John Hatcliff, and Muhammad Nanda. Using partial evaluation to enable verification of concurrent software. *ACM Computing Surveys*, 1998. (in press). 135
9. M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering. 135
10. Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123, Padova, Italy, September 1993. 136
11. 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. 137
12. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997. 135, 150
13. Neil D. Jones. The essence of program transformation by partial evaluation and driving. In Masahiko Sato Neil D. Jones, Masami Hagiya, editor, *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu*, pages 206–224. Springer-Verlag, April 1994. 135, 137, 147
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. 134, 137
15. Siau Cheng Khoo. *Parameterized partial evaluation: theory and practice*. PhD thesis, Yale University, June 1992. 135
16. Michael Leuschel. Program specialization and abstract interpretation reconciled. Technical Report CW 259, Departement Computerwetenschappen, K.U.Leaven, Belgium, May 1998. 135
17. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 135

18. David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation*. (in press). 136
19. Morten H. Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In John Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995. 136, 137, 146

An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs

Giorgio Levi and Fausto Spoto

Dipartimento di Informatica - Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
Phone: +39-50-887246, Fax: +39-50-887226
{levi,spoto}@di.unipi.it

Abstract. We apply the methodology of domain refinement to systematically derive domains for type analysis. Domains are built by iterative application of the Heyting completion operator to a given set of basic types. We give a condition on the type system which assures that two steps of iteration are sufficient to reach the fixpoint. Moreover, we provide a general representation for type domains through transfinite formulas. Finally, we show a subset of finite formulas which can be used as a computationally feasible implementation of the domains and we define the corresponding abstract operators.

Keywords: Abstract interpretation, abstract domain, static analysis, type analysis, logic programming.

1 Introduction

Type analysis for untyped logic programs is useful both to the programmer (for debugging and verification) and to the compiler (for code optimization). This is the motivation of many different proposals for type analysis. It is hard to compare the various techniques in terms of precision, efficiency and generality, because they use different methods and are often based on different assumptions.

There exist type inference techniques similar to those developed for (higher order) functional languages (see, for example, [15,16]) and techniques inspired by program verification methods [1]. Finally, there are plenty of type analysis techniques based on abstract interpretation [8]. The most relevant feature of abstract interpretation is that the analysis can systematically be derived from the (concrete) semantics and is guaranteed to be correct. The starting point is always the definition of an abstract domain modeling a given type system. In principle, the theory would allow us to systematically derive from the concrete semantics the optimal abstract operations and the corresponding abstract semantics, which is by construction a correct approximation. However, in practical type analysis systems, ad-hoc non-optimal abstract operations and abstract semantics computation algorithms are often considered. The theory is then used to prove the correctness of the construction.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 152–169, 1998.

The basic step in every abstract interpretation approach to type analysis is the choice of the abstract domain, which defines how we assign types to terms. A ground type language does not allow one to handle type dependencies [4]. This is the case of [14,18]. Some type dependencies among different arguments of a procedure can be expressed using type variables in the type language. This is a standard solution, used for instance in [3,13,19,6]. The same solution is used in the framework of regular approximations of the success set in [11]. However, the use of type variables does not allow one to express all type dependencies between argument positions. [5] is the only example of an analysis which *explicitly* expresses type dependencies. However, the underlying type language does not contain variables. Hence type dependencies arising from polymorphism can not be handled without an infinite set of dependencies. Moreover, [5] does not consider dependencies between different types, since the analysis is performed separately for every type. Finally, there is no approach which is able to express negative information on types. One of the contributions of this paper is to give a general technique for explicitly expressing general type dependencies and negative information on types.

We achieve this result by pushing forward the *systematic derivation* view of abstract interpretation by applying it to the design of the abstract domain. The systematic derivation of abstract domains can be based on the theory of domain refinement operators [9]. One example of such an operator is the Heyting completion [12], which was recently used to systematically reconstruct various domains for groundness analysis [17] and to show that the domain $\mathcal{POS}$ [2,7] is indeed optimal.

In this paper, we apply the same methodology to the derivation of new type domains. We start by defining a basic domain of elementary polymorphic types (section 4). We then define a hierarchy of refined domains by iteratively applying the Heyting completion operator (section 5). We prove (in section 6) that, for a large class of elementary types, the refinement procedure derives the optimal domain after two steps of refinement only, as was the case for the groundness domain. The optimal domain can then be viewed as a version of $\mathcal{POS}$ for types, and is similar to the domains of type dependencies defined in [5].

Once we have defined the abstract domain, we are left with the problems of defining a domain representation, suitable for being implemented in an abstract analyzer, and of giving a precise algorithmic definition of the abstract operators. We tackle the above problems in two steps. In the first step we represent type domains by formulas in a fragment of transfinite logic (section 7) and we define correct approximations for the abstract operators (section 8). The results of the first step can effectively be used for type analysis only if the set of types is finite. In the more interesting case of elementary type domains containing infinitely many types, transfinite formulas are not finitely representable. Hence the representation by means of transfinite formulas is introduced essentially to establish some theoretical results to be used in the next step (section 9), where we use a representation in terms of finite formulas with type variables. The resulting domain turns out to be formed by logic programs. Using logic programs to

represent abstract domains for logic programs is not new (see, for example, [10]). However, in our experiment, we succeed in providing a formal justification of the construction.

2 Preliminaries

1 Terms and Substitutions

Given a set of variables V and a set of function symbols Σ with associated arity, we define terms(Σ, V) as the set of terms built from V and Σ in the usual way. In the following sections, we will consider different signatures Σ for the set of terms built from the functor symbols of a logic program and for the set of terms built from a type signature. To distinguish these sets, the former will be denoted as U_V, while the latter as terms(Σ, V), where Σ will be a type signature and V is a set of type variables. We will often abridge $V \cup \{x\}$ as $V \cup x$ and $V - \{x\}$ as $V - x$.

We define Θ_{V,U_W} as the set of idempotent substitutions θ such that $\mathrm{dom}(\theta) \subseteq V$ and $\theta(x) \in U_W$ for every $x \in V$. If $\theta \in \Theta_{V,U_W}$ and $V' \subseteq V$, $\theta|_{V'}(x) = \theta(x)$ if $x \in V'$ and $\theta|_{V'}(x) = x$ if $x \notin V'$. A substitution $\theta \in \Theta_{V,U_W}$ is called grounding for a set of variables $G \subseteq V$ if and only if $\theta(x)$ is ground for every $x \in G$. For every set of variables V, a partial pre-ordering is defined on substitutions $\theta, \theta' \in \Theta_{V,U_W}$ as $\theta' \leq_V \theta$ if and only if there exists a substitution $\sigma \in \Theta_{W,U_W}$ such that $\theta' = \theta \circ \sigma$. When the set V is clear from the context, we will often remove the subscript V. A set $S \subseteq \Theta_{V,U_W}$ of substitutions is downward closed if and only if $\theta \in S$ and $\theta' \leq \theta$ entail $\theta' \in S$. We define $\downarrow S = \{\theta | \theta \leq \theta' \text{ and } \theta' \in S\}$.

Given $t_1, t_2 \in U_V$, we define $t_1 \leq_V t_2$ if and only if $t_1 = t_2\theta$ for a suitable $\theta \in \Theta_{V,U_V}$. As usual, the subscript will be always removed. $S \subseteq U_V$ is called downward closed if and only if $t \in S$ and $t' \leq t$ entail $t' \in S$.

If S is a set endowed with a relation $\leq$, $\mathcal{P}(S)$ is the set of all the subsets of S, while $\mathcal{P}{\downarrow}(S)$ is the set of all the downward closed (with respect to $\leq$) sets of S.

We will often define types as solutions of recursive equations over sets of terms. In such definitions, we will use classical λ-notation as well as a least fixpoint operator μ.

2 Abstract Interpretation

Abstract interpretation [8] is a theory developed to reason about the abstraction relation between two different semantics. The theory requires these semantics to be defined on domains which are posets. $(\mathcal{C}, \preceq)$ (the concrete domain) is the domain of the concrete semantics, while $(\mathcal{A}, \leq)$ (the abstract domain) is the domain of the abstract semantics. The partial order relations reflect an approximation relation. The two domains are related by a pair of functions α (*abstraction*) and γ (*concretization*), which form a Galois connection.

Let $f : \mathcal{C}^n \to \mathcal{C}$ be an operator and assume that $\tilde{f} : \mathcal{A}^n \to \mathcal{A}$ is its abstract counterpart. Then $\tilde{f}$ is *(locally) correct* with respect to f if and only if for all

$x_1, \ldots, x_n \in \mathcal{A}$ we have $\alpha(f(\gamma(x_1), \ldots, \gamma(x_n))) \preceq \tilde{f}(x_1, \ldots, x_n)$. According to the theory, for each operator f, there exists an optimal (most precise) locally correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$, where α is extended to sets $S \in \mathcal{C}$ defining $\alpha(S) = \wedge_{s \in S} \alpha(s)$.

We briefly recall the equivalence between the Galois insertion and the closure operator approach to the design of abstract domains. Let $\langle L, \leq, \wedge, \vee, \top, \bot \rangle$ be a complete lattice. An upper closure operator on L is an operator $\rho : L \mapsto L$ monotonic, idempotent and extensive. Each closure operator ρ is uniquely determined by the set of its fixpoints, which is its image $\rho(L)$. A set $X \subseteq L$ is the set of fixpoints of a closure operator if and only if X is a Moore family, i.e., $\top \in X$ and X is completely $\wedge$–closed. For any $X \subseteq L$, we denote by $\curlywedge(X)$ the Moore–closure of X, i.e., the least subset of L containing X which is a Moore family of L.

The complete lattice of all abstract interpretations (identified up to isomorphism) of a domain $\mathcal{C}$ is isomorphic to the complete lattice of upper closure operators on $\mathcal{C}$.

A systematic approach to the development of abstract domains is based on the use of domain refinement operators. Intuitively, given an abstract domain $\mathcal{A}$, a domain refinement operator R yields an abstract domain $R(\mathcal{A})$ which is more precise than $\mathcal{A}$. Classical domain refinement operations are reduced product and disjunctive completion [9].

Heyting completion was proposed in [12] as a powerful domain refinement operation. It allows us to include in a domain the information related to the propagation of the abstract property. Let L be a complete lattice and $a, b \in L$. The relative pseudo–complement (or intuitionistic implication) of a relative to b, if it exists, is the unique element $a \rightarrow b \in L$ such that for any $x \in L$: $a \wedge_L x \leq b$ if and only if $x \leq_L a \rightarrow b$. Relative pseudo–complements, when they exist, are uniquely given by $a \rightarrow b = \bigvee_L \{c | a \wedge_L c \leq_L b\}$. A complete lattice A is a complete Heyting algebra if and only if it is relatively pseudo–complemented, i.e., $a \rightarrow b$ exists for every $a, b \in A$. An example of complete Heyting algebra which will be used throughout this paper is $\langle \mathcal{P}\downarrow(\Theta_{V,U_V}), \subseteq, \cap, \cup, \Theta_{V,U_V}, \emptyset \rangle$. Given $a, b \in \mathcal{P}\downarrow(\Theta_{V,U_V})$, the intuitionistic implication $a \rightarrow b = \bigcup \{c \in \mathcal{P}\downarrow(\Theta_{V,U_V}) | a \cap c \subseteq b\}$ is also given by $a \rightarrow b = \{\theta \in \Theta_{V,U_V} |$ for all $\delta \leq \theta$ if $\delta \in a$ then $\delta \in b\}$. Note that this corresponds exactly to the definition of the concretization of implication in the case of the $\mathcal{POS}$ [2,7] domain for groundness analysis. The concretization of $x \Rightarrow y$ is the set of substitutions such that every instance which grounds x must also ground y. This is not by chance, as shown in [17]. Roughly speaking, an arrow $x \rightarrow y$ represents the set of substitutions such that the property of interest propagates from the variable x to the variable y for every possible instance.

Given the domains $\mathcal{D}^1_V \subseteq \mathcal{D}_V$ and $\mathcal{D}^2_V \subseteq \mathcal{D}_V$, we define $\mathcal{D}^1_V \xrightarrow{\curlywedge} \mathcal{D}^2_V = \curlywedge\{d_1 \rightarrow d_2 | d_1 \in \mathcal{D}^1_V$ and $d_2 \in \mathcal{D}^2_V\}$. This domain is called the Heyting completion of $\mathcal{D}^1_V$ with respect to $\mathcal{D}^2_V$ and contains all possible dependencies between an element of $\mathcal{D}^1_V$ and an element of $\mathcal{D}^2_V$.

3 The Concrete Domain

Since types are downward closed properties of substitutions, our concrete domain consists of downward closed sets of substitutions. Given a substitution θ, its downward closure represents the set of substitutions which are *compatible* with θ, i.e., which might be obtained by refining θ by further computation steps. For instance, if the computed substitution at a program point is $\{y \mapsto \mathtt{f}(x)\}$, then, as computation proceeds, the new substitution might be $\{y \mapsto \mathtt{f}(\mathtt{g}(w)), x \mapsto \mathtt{g}(w)\}$. With this interpretation in mind, a downward closed set of substitutions contains exactly all the substitutions which are compatible with the rest of the computation. For instance, if S_1 is the (downward closed) set of substitutions which are compatible with a procedure call p_1 and S_2 is the (downward closed) set of substitutions which are compatible with a procedure call p_2, then $S_1 \cap S_2$ is the (downward closed) set of substitutions which are compatible with the calls p_1, p_2 and p_2, p_1.

We endow downward closed sets of substitutions with two operations:

unification: if $S_1, S_2 \subseteq \Theta_{V,U_V}$, then $\mathrm{unify}_V(S_1, S_2) = S_1 \cap S_2$;
cylindrification: if $S \subseteq \Theta_{V,U_V}$, its cylindrification with respect to $x \in V$ is

$$\mathrm{cyl}_V(S, x) = \left\{ \theta' \in \Theta_{V,U_V} \;\middle|\; \begin{array}{l} \text{there exist } \theta, \sigma \in \Theta_{V\cup\{n\},U_{V\cup\{n\}}} \text{ s.t.} \\ \theta|_V = \theta', \theta \leq_{V\cup\{n\}} \sigma \text{ and } \sigma \in S[n/x] \end{array} \right\}, \quad (1)$$

where n is a new variable ($n \notin V$), $S[n/x] = \{\sigma[n/x] | \sigma \in S\}$ and $\sigma[n/x](n) = \sigma(x)$, $\sigma[n/x](x) = x$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$.

While the definition of unification is the classical one for the case of downward closed sets of substitutions (see for instance [17]), and is justified by the above considerations, it turns out that an *explicit* definition of concrete cylindrification on downward closed sets of substitutions was never given.

Definition (1) should be read as follows. In order to compute the cylindrification of a set S of substitutions, we consider x as a new variable n. Then we instantiate all the substitutions in S and we select those instantiations θ such that $\theta|_V$ does not contain n. We try now to get some insight on the meaning of this definition. Consider a procedure defined as $\mathtt{p(y)} : -\mathtt{y} = \mathtt{f(x)}$.. The set of substitutions which are consistent with the body of the procedure is $S = {\downarrow}\{\theta\}$, where $\theta = \{y \mapsto \mathtt{f}(x)\}$. Note that $\theta' = \{y \mapsto \mathtt{f}(\mathtt{g}(x))\} \notin S$, as long as we consider idempotent substitutions (or, equivalently, logic programming with occur–check). Consider now the set of substitutions which are consistent with the procedure call $\mathtt{p(y)}$. We have to consider x in the body of the procedure as existentially quantified. Hence x in the body of the procedure is *not* the same x that we have outside the procedure $\mathtt{p}$. This means that θ' is now consistent with the procedure call $\mathtt{p}$. For instance, if we make a procedure call $\mathtt{p(y)}$, with a partial computed substitution $\{y \mapsto \mathtt{f}(\mathtt{g}(x))\}$, we obtain success, even when the occur–check is performed. Definition (1) should now be clear. We consider x as a new variable n, then we instantiate the new variable in every possible way, including with terms which contain x.

The domain of downward closed sets of substitutions will be considered as our concrete domain. We will show in the following sections how some elements of this domain can be selected in order to get a hierarchy of type domains.

4 Basic Domains for Types

In this section we build a domain for type analysis which is able to model elementary monomorphic as well as polymorphic types. We assume a given set of functor symbols Σ (the type signature) and a finite set of variables $V = \{x, y, z, \dots\}$ (variables of interest).

Given Σ, we define a related interpretation $I_{\Sigma,V}$ (denoted in the following simply by I). The domain of I is $\mathcal{P}(U_V)$ (U_V could be the set of terms over V induced by the signature of the program). Functors are interpreted in a "user–defined" way. For instance the user can define:

$$\begin{aligned} I(\texttt{top}) &= U_V \\ I(\texttt{int}) &= \{\texttt{0}, \texttt{1}, \texttt{2}, \dots\} \\ I(\texttt{list}) &= \lambda\alpha.\mu\beta.\{[]\} \cup \{[h|t] | h \in \alpha \text{ and } t \in \beta\} \\ I(\texttt{tree}) &= \lambda\alpha.\mu t.\{\texttt{void}\} \cup \{\texttt{tree}(x,l,r) | x \in \alpha, l \in t \text{ and } r \in t\} \ . \end{aligned} \tag{2}$$

Note that in equations (2) `list` stands for the polymorphic list constructor, through the use of the λ-abstraction. Monomorphic lists can be defined as $I(\texttt{list}') = \mu l.\{[]\} \cup \{[h|t] | h \in U_V \text{ and } t \in l\}$. `tree` is the polymorphic tree constructor.

I allows us to evaluate a type $t \in \text{terms}(\Sigma, \emptyset)$ into a set of terms:

$$\begin{aligned} [\![c]\!]I &= I(c) \quad \text{if } c \text{ has arity } 0 \\ [\![f(t_1, \dots, t_n)]\!]I &= [\![f]\!]I([\![t_1]\!]I, \dots, [\![t_n]\!]I) \quad \text{if } f \text{ has arity } n. \end{aligned}$$

According to the above definitions, `tree(int)` contains the terms `void`, `tree(2, void,void)` and `tree(3,tree(1,void,void),void)`. Moreover, $I(\texttt{list(top)}) = I(\texttt{list}')$.

A type system is a triple $\langle \Sigma, V, I \rangle$ [1]. Given a variable x and a type $t \in \text{terms}(\Sigma, \emptyset)$, the set $x \in_{V,I} t = \{\theta \in \Theta_{V,U_V} | \theta(x) \in [\![t]\!]I\}$ is a *basic type property*. It represents the set of substitutions which bind x to a term which belongs to the type t. The domain of basic types on variables V, $\mathcal{TYPE}^0_{\Sigma,V,I}$ (in the following simply $\mathcal{TYPE}^0$), is defined as follows:

$$\mathcal{TYPE}^0 = \curlywedge \{x \in_{V,I} t \,|x \in V, t \in \text{terms}(\Sigma, \emptyset)\} \ .$$

As already mentioned, the Moore family operator selects the least set of downward closed sets of substitutions which contains top and all basic type properties

[1] Strictly speaking, we should include U_V in a type system. This information will be left unspecified, in order to simplify the notation.

and is closed with respect to intersection. The selected set is ordered with respect to the original ordering relation on downward closed sets of substitutions (set inclusion). Note that if I yields downward closed sets for constants and downward closed sets transformers for symbols with arity greater than zero, then $\mathcal{TYPE}^0$ is a set of downward closed sets of substitutions. In the following, we will assume this hypothesis to be satisfied. An abstraction map from the set of downward closed sets of substitutions into $\mathcal{TYPE}^0$ can be defined in a standard way.

$\mathcal{TYPE}^0$ is able to model only conjunction of simple type properties, like for instance "x is an integer and y is a list of integers". It is not able to model type dependencies (directionality). This means that the set $\downarrow\{\theta\}$, where $\theta = \{x \mapsto [h|t]\}$ is abstracted into Θ_{V,U_V}, the set of all the substitutions, in the type system defined by equations (2). $\mathcal{TYPE}^0$ is not able to model the directionality of θ. Actually, if h is of type `int` and t is of type `list(int)`, then x is of type `list(int)`, and vice versa. In the following section we will introduce directionality, by systematically refining $\mathcal{TYPE}^0$ by the Heyting completion operator.

5 A Hierarchy of Domains for Directional Types

We want now to build a (possibly infinite) hierarchy of type domains as follows:

$$\begin{array}{l} \mathcal{TYPE}^0 = \curlywedge\{x \in_{V,I} t \,|x \in V, t \in \text{terms}(\Sigma, \emptyset)\} \ , \\ \mathcal{TYPE}^i = \mathcal{TYPE}^{i-1} \xrightarrow{\curlywedge} \mathcal{TYPE}^{i-1} \text{ for } i \geq 1 \ . \end{array} \tag{3}$$

Note that we do not know whether this refinement chain is finite or not. In the following, we will show that under proper conditions on the type system this chain is finite. Namely, it converges at $\mathcal{TYPE}^2$.

Consider now a generic element of $\mathcal{TYPE}^1$. It has the form $(b_1^1 \to b_2^1) \cap \ldots \cap (b_1^n \to b_2^n) \cap \ldots$ where $b_1^i, b_2^i \in \mathcal{TYPE}^0$ for $i \geq 1$. The intersection can be finite as well as infinite. Moreover, every b_j^i has the form $b_j^i = (x_1^{i,j} \in_{V,I} t_1^{i,j}) \cap \ldots \cap (x_{m_{i,j}}^{i,j} \in_{V,I} t_{m_{i,j}}^{i,j}) \cap \ldots$ where, again, the intersection can be finite as well as infinite. Hence $\mathcal{TYPE}^1$ is able to model directional types.

Assume the type system to contain two disjoint types t_1 and t_2. Namely, we require $[\![t_1]\!]I \cap [\![t_2]\!]I = \emptyset$. In such a case, $\mathcal{TYPE}^1$ would contain the element $(x \in_{V,I} t_1 \cap x \in_{V,I} t_2) \leftarrow y \in_{V,I} t$, where t is a type and x, y are two variables. The meaning of this element is that y is not and will not be eventually bound to a term of type t. This is a form of intuitionistic negation. Note that the variable x can be substituted with whatever other variable. Its name is irrelevant. Hence we could simply write $\bot \leftarrow y \in_{V,I} t$, where $\bot$ means failure. Moreover, this argument can be applied even if there are not disjoint types. We just need to add a distinguished type `bot` to the type system, whose interpretation is $I(\texttt{top}) = \emptyset$. In such a case, $\bot$ would be an abridged form for $x \in_{V,I}$ `bot`. The introduction of negative information seems to be a distinguishing feature of our approach. This information is essentially useless in the case of groundness analysis. For instance, if we add the distinguished type `bot` to the basic domain for groundness containing the unique property `g` such that $I(\texttt{g}) = \{t \in U_V | \text{ vars}(t) = \emptyset\}$, then

an element of the form $\bot \leftarrow x \in_{V,I} \mathtt{g}$ is $\bot$ itself. This is because every term can always be made ground.

Negative information is extremely powerful for generic type systems. In our concluding example of analysis (section 10) we will show a case where it plays a key role.

It is worth noting that polymorphism is treated in a "ground fashion". For instance, consider the type signature and the interpretation given by equations (2). In $\mathcal{TYPE}^1$ we are able to say that if x is of type T then y is of type $\mathtt{list}(T)$. However, the element representing this information is an infinite intersection of the form $(x \in_{V,I} \mathtt{int} \rightarrow y \in_{V,I} \mathtt{list(int)}) \cap x \in_{V,I} \mathtt{list(int)} \rightarrow y \in_{V,I} \mathtt{list(list(int))}) \cap (x \in_{V,I} \mathtt{list(list(int))} \rightarrow y \in_{V,I} \mathtt{list(list(list(int)))}) \cap \ldots$ This observation means that the way elements are built in $\mathcal{TYPE}^i$ can not be used directly as a guide for devising a computationally effective representation for $\mathcal{TYPE}^i$. We have to represent a possibly infinite intersection in a finite way. This can be accomplished through the use of type variables in the representation, as it will be shown in section 9.

6 Well Formed Type Systems

In this section we investigate a class of type systems which enjoy the property that the refinement chain (3) is finite. Namely, for these type systems the refinement chain converges at the second refinement step to a domain which contains, by construction, all possible type dependencies.

Definition 1. *A type system* $\langle \Sigma, V, I \rangle$ *is called* well formed *if and only if, for every* $\theta \in \Theta_{V,U_V}$*, there exists a grounding substitution* σ *for* V*, such that for every type* $t \in \mathrm{terms}(\Sigma, \emptyset)$*,* $\theta(x) \in [\![t]\!]I$ *if and only if* $(\theta(x))\sigma \in [\![t]\!]I$*.*

Roughly speaking, well formed type systems are such that terms which do not belong to types can be instantiated in such a way that they will definitively not belong to those types. It turns out that all sensible type systems are well formed. For instance, the type system INT_LIST_TOP with $\Sigma = \{\mathtt{int}, \mathtt{list}, \mathtt{top}\}$, $I[\mathtt{int}] = \mu i.i = \{\mathtt{0}\} \cup \{\mathtt{s}(j) | j \in i\}$, $I[\mathtt{list}] = \lambda x.\mu l.l = \{[]\} \cup \{[h|t] | h \in x, t \in l\}$ and $I[\mathtt{top}] = U_V$, is well formed.

Let b_i, c_j, d_k be basic type properties for $i \in I$, $j \in J$ and $k \in K$, where I, J, K are index sets. Let $B = \bigcap_{i \in I} b_i$, $C = \bigcup_{j \in J} c_j$ and $D = \bigcup_{k \in K} d_k$. Assume the following condition holds:

H1: $(B \rightarrow C) \rightarrow D = (B \cup D) \cap (C \rightarrow D)$.

Intuitively, hypothesis H1 means that deep arrows can be factorized into simpler arrows. The following results can be proved by extending similar proofs done in [17] for the groundness domains:

1. If $\langle \Sigma, V, I\rangle$ satisfies condition H1, then $\mathcal{TYPE}^2 = \mathcal{TYPE}^i$ for every $i \geq 2$;
2. If $\langle \Sigma, V, I\rangle$ satisfies condition H1, then $\mathcal{TYPE}^2$ can be obtained as implication between conjunctions of basic type properties and disjunctions of basic type properties. Formally, we have:

$$\mathcal{TYPE}^2 = \bigwedge \{a \to o \,|\, a \in \mathcal{TYPE}^0 \text{ and } o \in \mathcal{OR}\} \text{ ,where}$$
$$\mathcal{OR} = \left\{ \bigcup_i \{x_i \in_{V,I} t_i\} \,\middle|\, \begin{array}{l} x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \\ \text{and the union is not empty} \end{array} \right\} .$$

$\mathcal{OR}$ is able to model disjunction of basic type properties, while $\mathcal{TYPE}^2$ is able to model propagation of type information from conjunctions of basic type properties to disjunctions of basic type properties.

The importance of well formed type systems is that they satisfy condition H1:

Proposition 1. *Condition H1 holds for well formed type systems.*

7 A Hierarchy of Intermediate Representations

In this section we consider a hierarchy of representations for the above defined type domains. They are intermediate because they are not adequate for a direct implementation in an analysis tool. However, they will be used to obtain some theoretical results, and to devise an effective representation for our type domains.

We start by defining a fragment of transfinite logic which will be called $\mathcal{LOG}_{\Sigma,V}$ (in the following simply $\mathcal{LOG}$). It is the least set such that $false, true \in \mathcal{LOG}$, $(x \in t) \in \mathcal{LOG}$, for $x \in V$ and $t \in \text{terms}(\Sigma, \emptyset)$, if S is a (possibly infinite) subset of $\mathcal{LOG}$ then $\wedge(S) \in \mathcal{LOG}$ and $\vee(S) \in \mathcal{LOG}$, if $s_1, s_2 \in \mathcal{LOG}$ then $s_1 \Rightarrow s_2 \in \mathcal{LOG}$ and if $s \in \mathcal{LOG}$ then $\neg s \in \mathcal{LOG}$.

Given a substitution $\theta \in \Theta_{V,U_V}$ and an interpretation I, we define the interpretation of a formula of $\mathcal{LOG}$ as follows:

$$\begin{array}{rl} [\![false]\!]^I_\theta = 0 \text{ and } [\![true]\!]^I_\theta = 1 & \\ [\![x \in t]\!]^I_\theta = 1 & \text{iff } \theta(x) \in [\![t]\!]I \text{ , for } x \in V \text{ and } t \in \text{terms}(\Sigma, \emptyset), \\ [\![\wedge(S)]\!]^I_\theta = 1 & \text{iff } [\![s]\!]^I_\theta = 1 \text{ for every } s \in S, \\ [\![\vee(S)]\!]^I_\theta = 1 & \text{iff there exists } s \in S \text{ such that } [\![s]\!]^I_\theta = 1 \text{ ,} \\ [\![s_1 \Rightarrow s_2]\!]^I_\theta = 1 & \text{iff if } [\![s_1]\!]^I_\theta = 1 \text{ then } [\![s_2]\!]^I_\theta = 1, \\ [\![\neg s]\!]^I_\theta = 1 & \text{iff } [\![s]\!]^I_\theta = 0 \text{ .} \end{array}$$

An interpretation I induces an equivalence relation $\equiv_I$ on formulas. Namely, $\phi_1 \equiv_I \phi_2$ if and only if for every $\theta \in \Theta_{V,U_V}$, $[\![\phi_1]\!]^I_\theta = 1$ entails $[\![\phi_2]\!]^I_\theta = 1$ and vice versa. This equivalence will be called *logical* equivalence in the following.

Given $\phi \in \mathcal{LOG}$ and an interpretation I, we define the map $\gamma_I : \mathcal{LOG} \mapsto \mathcal{P}{\downarrow}(\Theta_{V,U_V})$ as $\gamma_I(\phi) = \{\theta \in \Theta_{V,U_V} | \text{for all } \sigma \leq \theta, [\![\phi]\!]^I_\sigma = 1\}$. γ_I induces an equivalence relation on formulas defined as $\phi_1 \equiv_{\gamma_I} \phi_2$ if and only if

$\gamma_I(\phi_1) = \gamma_I(\phi_2)$. This equivalence will be called γ-equivalence in the following. Note that if ϕ_1 and ϕ_2 are logically equivalent then they are γ-equivalent, by definition of γ. However, the converse, in general, does not hold.

We define now a hierarchy of representations as follows:

$$\mathcal{LOG}^0 = \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \bigcup_i \{x_i \in t_i\} \ , \\ x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \end{array} \right\} /_{\equiv_I} \ ,$$

$$\mathcal{LOG}^1 = \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \bigcup_j \{s_j^1 \Rightarrow s_j^2\} \ , \\ s_j^1, s_j^2 \in \mathcal{LOG}^0 \end{array} \right\} /_{\equiv_I} \ ,$$

$$\mathcal{LOG}^2 = \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \cup_j \{a_j \Rightarrow o_j\} \ , \\ [a_j]_{\equiv_I} \in \mathcal{LOG}^0, \\ [o_j]_{\equiv_I} \in \mathcal{LOG}_{\mathcal{OR}} \end{array} \right\} /_{\equiv_I} \text{, where}$$

$$\mathcal{LOG}_{\mathcal{OR}} = \left\{ \vee(S) \;\middle|\; \begin{array}{l} S = \bigcup_i \{x_i \in t_i\} \text{ is non empty}, \\ x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \end{array} \right\} /_{\equiv_I} \ .$$

The following inclusions can easily be proved: $\mathcal{LOG}^0 \subseteq \mathcal{LOG}^1 \subseteq \mathcal{LOG}^2$.

We extend the concretization map on equivalence classes as $\gamma_I([\phi]_{\equiv_I}) = \gamma_I(\phi)$. This definition is well given because logical equivalence entails γ-equivalence. Note that γ_I is monotonic. In the following, a formula will stand for its (logical) equivalence class.

It can be shown that every element of $\mathcal{TYPE}^0$ is the image through γ_I of an element of $\mathcal{LOG}^0$, that every element of $\mathcal{TYPE}^1$ is the image through γ_I of an element of $\mathcal{LOG}^1$ and that every element of $\mathcal{TYPE}^2$ is the image through γ_I of an element of $\mathcal{LOG}^2$. Note that this last result holds only for the particular form of the elements of $\mathcal{LOG}^2$. Hence in these three cases γ_I is onto. We have already shown that it is monotonic. However, we know that it is not always one to one. If we make the assumption:

H2: γ_I is one to one, i.e., logical equivalence is γ-equivalence;

then we conclude that the following isomorphisms hold: $\mathcal{TYPE}^0 \approx \mathcal{LOG}^0$, $\mathcal{TYPE}^1 \approx \mathcal{LOG}^1$ and $\mathcal{TYPE}^2 \approx \mathcal{LOG}^2$. The following result shows that well formed type systems satisfy condition H2. Hence the representations of this section are isomorphic to the type domains of section 5 for well formed type systems.

Proposition 2. *Condition H2 holds for well formed type systems.*

8 Abstract Operators on Transfinite Logic

In this section we give an explicit definition of correct abstract operators on the domains of transfinite logic formulas.

Let us first note that if S_1 is represented by ϕ_1 and S_2 is represented by ϕ_2, then $S_1 \cap S_2$ is represented by $\phi_1 \wedge \phi_2$. Moreover, this is the best possible approximation.

We consider now the approximation of the concrete cylindrification operator. Let $\mathcal{T} = \curlywedge\{[\![t]\!]I | t \in \text{terms}(\Sigma, \emptyset)\}$ be the least Moore family (with respect to set intersection) which contains all the types. The substitution of a type $t \in \mathcal{T}$ for a variable x in a formula ϕ is defined as follows:

$$
\begin{array}{rr}
false[t/x] = false & (y \in t')[t/x] = (y \in t') \\
true[t/x] = true & \text{if } x \neq y \\
(x \in t')[t/x] = \begin{cases} true & \\ \text{if } t \subseteq [\![t']\!]I & \\ false & \\ \text{otherwise} & \end{cases} & \begin{array}{r} \wedge(S)[t/x] = \wedge(\{s[t/x] | s \in S\}) \\ \vee(S)[t/x] = \vee(\{s[t/x] | s \in S\}) \\ (s_1 \Rightarrow s_2)[t/x] = s_1[t/x] \Rightarrow s_2[t/x] \\ (\neg s)[t/x] = \neg(s[t/x]) \ . \end{array}
\end{array}
$$

We define the abstract cylindrification operator on the logical domain as $\exists_x \phi = \vee(\{\phi[t/x] | t \in \mathcal{T}'\})$, where $\mathcal{T}' \subseteq \mathcal{T}$ is the set of types which are the most specific type of some term. Note that this subset is not empty because $\mathcal{T}$ is a Moore family. Hence every term has a most specific type. This definition is similar to the Schröder elimination used in the case of groundness analysis [2,7]. Note that x does not occur in $\exists_x \phi$. For generic type systems, $\exists_x$ is not a correct cylindrification operator. However, it turns out that $\exists_x$ is a correct abstract operator with respect to concrete cylindrification in the case of well formed type systems.

9 Logic Programs as Finite Representations of Type Domains

Transfinite formulas can be used as a computationally effective representation domain if the set of types is finite. In such a case the set of transfinite formulas is isomorphic to a set of finite formulas and the abstract operators on the domain can correctly be approximated by effective algorithms. For instance, the operator $\exists_x$ becomes an algorithm for computing cylindrification. Even the equivalence test between two formulas becomes effective, though very expensive, being a classical $\mathcal{NP}$–complete problem.

A more interesting case is when we deal with an infinite set of basic types, i.e., when $\text{terms}(\Sigma, \emptyset)$ is infinite. In such a case, transfinite formulas are not finitely representable. However, the full power of transfinite formulas is seldom useful for our purposes. For instance, to express the relationship between the variables in the binding $x = [h|t]$ we must write an infinite conjunction: $\bigwedge_{t \in \text{terms}(\Sigma,\emptyset)} (x \in \texttt{list}(t) \iff h \in t \wedge t \in \texttt{list}(t))$. However, this could be expressed more compactly using type variables as $x \in \texttt{list}(T) \iff h \in T \wedge t \in \texttt{list}(T)$.

In this section, we introduce a domain of finite formulas with type variables and the corresponding abstract operators.

Let $V' = \{X, Y, Z, \dots\}$ be an infinite set of type variables. Type variables are denoted by uppercase letters to distinguish them from the variables of interest $V = \{x, y, z, \dots\}$. $\mathcal{REP}_{\Sigma,V,V',I}$ (in the following, simply $\mathcal{REP}$) is

the least set of first order formulas containing $x_i \in t_i$ for $x_i \in V$ and $t_i \in \text{terms}(\Sigma, V')$ and closed with respect to $\wedge$ and $\Rightarrow$, modulo the equivalence relation $\equiv_{\mathcal{REP}}$ defined as follows: $\phi_1 \equiv_{\mathcal{REP}} \phi_2$ if and only if $\gamma_{\mathcal{REP}}(\phi_1) \equiv_I \gamma_{\mathcal{REP}}(\phi_2)$, where $\gamma_{\mathcal{REP}}(\phi(X_1, \ldots, X_2)) = \bigwedge_{t_1,\ldots,t_n \in \text{terms}(\Sigma,\emptyset)} \phi[t_1/X_1, \ldots, t_n/X_n]$ (where $\phi(X_1, \ldots, X_n)$ means that the type variables contained in ϕ are *exactly* $X_1, \ldots, X_n$). Intuitively, the last formula is the transfinite formula represented by the finite formula ϕ. Note that there exist transfinite formulas which can not be represented this way. Moreover, $\mathcal{REP}$ is not closed with respect to infinite $\wedge$. This means that we are not guaranteed to have optimal operators. Finally, in general $\mathcal{REP}$ is not finite and not even noetherian. For example, the instance of $\mathcal{REP}$ induced by the type system INT_LIST_TOP is not noetherian, because there exists an infinite chain $X \in \texttt{top}$, $X \in \texttt{list(top)}$, $X \in \texttt{list(list(top))}$, and so on.

In order to make $\mathcal{REP}$ finite, we consider the approximate domain $\mathcal{REP}^k$, $k \in \mathbb{N}$, $k > 0$, whose formulas contain constraints of the form $x_i \in t_i$, such that $t_i \in \text{terms}(\Sigma, V')$ and the depth of t_i is less than or equal to k. We define $\gamma_{\mathcal{REP}^k} = \gamma_{\mathcal{REP}}$ and $\equiv_{\mathcal{REP}^k} = \equiv_{\mathcal{REP}}$ restricted to formulas in $\mathcal{REP}^k$.

Considering only constraints with bounded term depth does not boil down to the case of a finite set of types. In fact, type variables are free to assume any value, with arbitrary depth. As the concluding example will show, this restriction on the constraints does not introduce a big loss in precision, thanks to the use of type variables.

We give now algorithmic definitions for the two abstract operators and for the abstraction map.

Abstract unification. It can be shown that $\wedge$ is correct with respect to intersection of downward closed sets of substitutions. Given $\phi_1, \phi_2 \in \mathcal{REP}^k$, we have $\gamma_I(\gamma_{\mathcal{REP}^k}(\phi_1 \wedge \phi_2)) = \gamma_I(\gamma_{\mathcal{REP}^k}(\phi_1)) \cap \gamma_I(\gamma_{\mathcal{REP}^k}(\phi_2))$.

As a consequence, we have that $(x \in \texttt{list}(T) \Leftarrow y \in T) \wedge (z \in T \Leftarrow w \in T) \equiv_{\mathcal{REP}^k} (x \in \texttt{list}(T) \Leftarrow y \in T) \wedge (z \in D \Leftarrow w \in D)$. Actually, two occurrences of the same type variable in two different implications can be replaced by different type variables.

This suggests an interesting interpretation for formulas in $\mathcal{REP}^k$. Since $(A_1 \wedge \ldots \wedge A_n) \Leftarrow B$ can be equivalently rewritten as $(A_1 \Leftarrow B) \wedge \ldots \wedge (A_n \Leftarrow B)$, the elements of $\mathcal{REP}^k$ can be viewed as definite Horn clauses. We only need to interpret a constraint of the form $x \in t$ as an atom $x(t)$, where the variables of interest become predicate symbols. For instance, the abstract constraint

$$(x \in \texttt{list}(T) \Leftarrow (y \in T \wedge z \in \texttt{list}(T))) \wedge (y \in T \Leftarrow x \in \texttt{list}(T)) \wedge z \in \texttt{list}(T)$$

can be seen as the logic program shown in figure 1(a). The above remark is important when we look for a correct approximation of the cylindrification operator. We already have a non effective definition of the approximation: given ϕ and a variable x, we compute $\gamma_{\mathcal{REP}^k}(\phi)$, then we apply Schröder elimination. This definition is not adequate because Schröder elimination destroys the clause

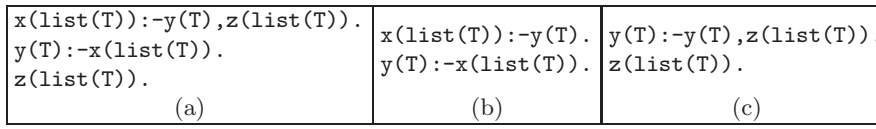


Fig. 1. A constraint and two of its cylindrifications.

structure of the elements of $\mathcal{REP}^k$, which is extremely important for representing in a compact way a possibly infinite quantification on types. If the elements of $\mathcal{REP}^k$ are represented as logic programs, cylindrification of an element P with respect to a variable x means computing a program P' which expresses the same dependencies among the predicate symbols (variables of interest) different from x in the same way as P does, but does not contain x anymore. The simplest technique for removing a predicate from a program is unfolding. Given two clauses $H_1 \Leftarrow B_1 \wedge \ldots \wedge B_n$ and $H_2 \Leftarrow C_1 \wedge \ldots \wedge C_m$, if $\theta = \text{mgu}(B_i, H_2)$ exists, one of their unfoldings is $(H_1 \Leftarrow B_1 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge \ldots \wedge C_m \wedge B_{i+1} \wedge \ldots \wedge B_n)\theta$. It can be shown that the unfoldings of two clauses are logical consequences of them. Note, however, that the unfolding of two clauses whose terms have depth less than or equal to k can contain a term with depth greater than k.

Abstract cylindrification. We define the operator $\exists_x^{\mathcal{REP}^k}$ through the unfolding operation. Any element $P \in \mathcal{REP}^k$ is viewed as a set of clauses. In order to compute $\exists_x^{\mathcal{REP}^k} P$, we perform the following three steps:

1. we add to P all the possible unfoldings of any clause containing x in the body with any clause containing x in the head. Let P' be the resulting program;
2. we remove from P' all the clauses containing x thus obtaining P'';
3. we remove from P'' all the clauses which contain terms with depth greater than k, thus obtaining $\exists_x^{\mathcal{REP}^k} P$.

For instance, the abstract cylindrification of the program of figure 1(a) with respect to the variable `z` and for $k = 2$ is the program shown in figure 1(b), while the abstract cylindrification of the same program with respect to the variable `x` and for $k = 2$ is the program shown in figure 1(c), which is $\equiv_I$-equivalent to the program `z(list(T)).`.

Note that the algorithm for computing the abstract cylindrification introduces a loss in precision in the last two steps. In order to improve the precision, we can repeat the first step to decrease the number of clauses which contain x in the body. However, the loss in precision of the third step can not be avoided. It can be shown that $\exists_x^{\mathcal{REP}^k}$ is indeed correct with respect to concrete cylindrification for well-formed type systems.

The algorithm for cylindrification uses concrete unification between type terms. This is not related to the unification operator of the domain. It is simply a consequence of the use of logic programs as abstract domains. However, since

types are partially ordered with respect to a subtyping relation (for instance: `int` $\subseteq$ `top`), the unification procedure used in the unfolding step might be too coarse. For instance, if we have a clause whose head is $x \in$ `list(int)` and we try to unfold it in the body of a clause containing $x \in$ `list(top)`, the unification procedure fails. Actually, unfolding should be allowed because if x is a list of integers then it is even a list of generic terms. Similarly, if we have a clause whose body contains $x \in T$, we can remove $x \in T$ from the body and instantiate the resulting clause with the substitution $\{T \mapsto \texttt{top}\}$. This is correct because every term is always in `top`. This means that we could improve the precision of the cylindrification operator using *in its algorithmic definition* a unification procedure which embeds subtyping information.

Abstraction map. We define now a correct approximation of the abstraction of $\downarrow\{\theta\}$. We only need to find a formula $\phi \in \mathcal{RECP}^k$, such that $\downarrow\{\theta\} \subseteq \gamma_I\gamma_{\mathcal{RECP}^k}(\phi)$. Let $\alpha(\theta)$ be one such a formula. Assume we have a procedure `type`, which, for every term $u \in U_V$ with $\mathrm{vars}(u) = \{x_1, \ldots, x_n\}$, behaves as $u \overset{\texttt{type}}{\rightsquigarrow} \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle\}$, where t^i and $t^i_{x_j}$ belong to $\mathrm{terms}(\Sigma, V')$, i.e., they are types possibly containing type variables. We require that, for every $\sigma \in \Theta_{V,U_V}$, $u\sigma \in [\![t^i\sigma']\!]I$ for a suitable $\sigma' \in \Theta_{V',\mathrm{terms}(\Sigma,\emptyset)}$ grounding for V', if and only if, for all $j = 1, \ldots, n$, $x_j\sigma \in [\![t^i_{x_j}\sigma']\!]I$.
Roughly speaking, `type`(u) computes a finite set of possible types for u, and, for each possible type, it computes some necessary and sufficient conditions on the variables of u in order for u to belong to the type. Note that a straightforward definition of the `type` procedure can be automatically derived from the definition of types and that this definition is compositional with respect to addition of new types to the type system.
Given the procedure `type` and a substitution θ, such that $x \in \mathrm{dom}(\theta)$, we define $\alpha_x(\theta) = \bigwedge_{i=1}^m \big(x \in t^i \iff \big(x_1 \in t^i_{x_1} \wedge \ldots \wedge x_n \in t^i_{x_n}\big)\big)$, where $\mathrm{vars}(\theta(x)) = \{x_1, \ldots, x_n\}$ and $\texttt{type}(\theta(x)) = \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle\}$. Finally, we define $\alpha(\theta) = \bigwedge_{x \in \mathrm{dom}(\theta)} \alpha_x(\theta)$. It can be proved that, for every substitution $\theta \in \Theta_{V,U_V}$, $\downarrow\{\theta\} \subseteq \gamma_I\gamma_{\mathcal{RECP}^k}(\alpha(\theta))$.

For instance, if we consider the top type, integers, and polymorphic lists, we can implement `type` as a Prolog procedure `type(Term,Type)` which enumerates all possible types `Term` can take. Moreover, the variables of `Term` are bound to types to represent necessary and sufficient conditions for `Term` to belong to `Type`. For instance, `type([H|T],Type)` yields a computed answer substitution $\{\texttt{Type} \mapsto \texttt{list(S)}, \texttt{H} \mapsto \texttt{S}, \texttt{T} \mapsto \texttt{list(S)}\}$, meaning that `[H|T]` has type `list(S)` if and only if `H` has type `S` and `T` has type `list(S)`. An example of such a procedure is shown in figure 2.

The above algorithmic definition of the abstraction map can be improved by extracting from a substitution even the negative information that it contains. We just need to modify `type`. We can assume that `type`(t) contains even pairs of the form $\langle t_i, \bot\rangle$, meaning that the term t can never belong to the type t_i. For instance, $[H|T]$ can never be an integer, while $\mathtt{s}(X)$ can. As a consequence, we define $\alpha_x(\theta) = \bigwedge_{i=1}^m \big(x \in t^i \iff \big(x_1 \in t^i_{x_1} \wedge \ldots \wedge x_n \in t^i_{x_n}\big)\big) \wedge \bigwedge_{i=1}^k \big(\bot \Leftarrow (t')^i\big)$,

meta-clause `type(X,S):-var(X),!,X=S.`
the whole universe U_V `type(X,top).`
integers: $\mu i.i = \{0\} \cup \{s(l) \mid l \in i\}$ $\texttt{type}(\texttt{X}, \texttt{int}) : -\texttt{X} = 0.$ $\texttt{type}(\texttt{X}, \texttt{int}) : -\texttt{X} = \texttt{s}(\texttt{I}), \texttt{type}(\texttt{I}, \texttt{int}).$
polymorphic lists: $\lambda s.\mu l.l = \{[]\} \cup \{[h\|t] \mid h \in s \text{ and } t \in l\}$ $\texttt{type}(\texttt{X}, \texttt{list}(\texttt{S})) : -\texttt{X} = [].$ $\texttt{type}(\texttt{X}, \texttt{list}(\texttt{S})) : -\texttt{X} = [\texttt{H}\|\texttt{T}], \texttt{type}(\texttt{H}, \texttt{S}), \texttt{type}(\texttt{T}, \texttt{list}(\texttt{S})).$

Fig. 2. An example of the procedure `type`.

where $\text{vars}(\theta(x)) = \{x_1, \ldots, x_n\}$ and $\texttt{type}(\theta(x)) = \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle, \langle (t')^1, \bot\rangle, \ldots, \langle (t')^k, \bot\rangle\}$.

10 An Example

We implemented in Prolog an abstract analyzer which uses the $\mathcal{REP}^k$ abstract domain and which is parametric with respect to a given set of types. In this section, we show how it behaves on the program shown in figure 3, which computes the derivative of an expression involving the variable `x`. The types used in the

```
int(0).
int(s(I)):-int(I).

der(x,s(0)).
der(X,0):-int(X).
der(X*Y,(DX*Y)+(X*DY)):-der(X,DX),der(Y,DY).
der(X+Y,DX+DY):-der(X,DX),der(Y,DY).
der(-(X),-(DX)):-der(X,DX).
der(X-Y,DX-DY):-der(X,DX),der(Y,DY).
der(X^K,DK*K*(X^(K-s(0)))):-der(K,DK).
der(exp(X),DX*exp(X)):-der(X,DX).
der(sin(X),DX*cos(X)):-der(X,DX).
der(cos(X),-(DX*sin(X))):-der(X,DX).
```

Fig. 3. The program of our example.

analysis are the top type, denoted by `top`, integers, denoted by `int`, generic expressions on `x`, denoted by `expr`, and algebraic expressions on `x`, i.e., expressions on `x` which do not involve exponentiation or trigonometric functions, denoted by `algebraic`. We compute the abstract fixpoint of the above program through

our analyzer. Then we evaluate the query (mode) `der(algebraic,top)` in the abstract fixpoint. We get the set of constraints shown in figure 4, where `var0` and `var1` stand for the first and for the second argument of the predicate `der`, respectively. Every constraint is a logic program. If the predicate `bot` is derivable from

```
constraint 1:
bot:-var0(int).
var0(algebraic).
var0(expr).
var1(algebraic).
var1(expr).
var1(int).
var1(top).

constraint 2:
var0(algebraic).
var0(expr).
var0(int).
var1(algebraic).
var1(expr).
var1(int).
var1(top).

constraint 3:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(expr).
 var1(top).

constraint 4:
bot:-var0(int).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(algebraic).
var1(expr).
var1(top).

constraint 5:
bot:-var0(int).
bot:-var1(int).
var0(algebraic).
var0(expr).
var0(expr):-var1(expr).
var1(algebraic):-var0(algebraic).
var1(expr):-var0(expr).
var1(top).

constraint 6:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr):-var1(expr).
var1(expr):-var0(expr).
var1(top).

constraint 7:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr):-var1(expr).
var1(expr):-var0(expr).
var1(top).

constraint 8:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(expr).
var1(top).
```

Fig. 4. The set of constraints computed for our query.

the logic program, then the constraint can be dropped since it is not satisfiable. In the case at hand, constraints 3, 6, 7 and 8 are dropped. From the remaining four constraints, we derive the fact `var1(expr)`. This means that the second argument is bound to an expression. More interestingly, the same constraints allow us to derive the fact `var1(algebraic)`, i.e., the second argument is bound to an algebraic expression. Roughly speaking, our analyzer concludes that the derivative of an algebraic expression is an algebraic expression too. Note that this result was possible only through the use of negative information. Namely, the dropped constraints do not allow to derive the fact `var1(algebraic)`. Hence only if we remove them we can obtain the desired result.

11 Conclusions

We presented a polymorphic type analysis scheme based on abstract interpretation. The construction of the abstract domains is made through a formal methodology, namely domain refinement starting from a simple domain of elementary

types. We have introduced an isomorphic representation of the elements of the domain by means of transfinite formulas. We have given sufficient conditions on the type systems which assure that the resulting type domains and type representations with transfinite formulas enjoy some desirable properties, namely factorization of deep type implications, identity between logical equivalence on the representation and concretization equality and correctness of the Schröder elimination procedure on the representation. Finally, we have shown how a finite domain of finite formulas (represented by definite Horn clauses) with type variables can be selected in order to make the analysis effective.

We are left with several important open problems. Some of the problems we are currently investigating are:

- the optimality of Schröder elimination;
- the automatic derivation and the use of subtyping information in the algorithm for abstract cylindrification;
- the relation, in terms of precision, between the domain of finite formulas with type variables and the domain of transfinite formulas;
- the definition of a better approximation of the abstraction map into formulas in $\mathcal{REP}^k$;
- the definition of a generic implementation based on a *type specification language* (as we have shown, a type analyzer can be constructed from a type specification in an automatic way);
- the comparison of our technique to other existing techniques for type analysis based on abstract interpretation. The domain refinement methodology we use should be useful to compare existing domains, as already shown in the case of groundness analysis.

References

1. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994. 152
2. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, 1994. 153, 155, 162
3. R. Barbuti and R. Giacobazzi. A Bottom-up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19(3):281–313, 1992. 153
4. J. Boye. *Directional Types in Logic Programming*. PhD thesis, Linköping University (Sweden), 1996. 153
5. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994. 153, 153, 153

6. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI–unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996. Extended version to appear in Theoretical Computer Science. 153
7. A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996. 153, 155, 162
8. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992. 152, 154
9. G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view on abstract domain design. *ACM Computing Surveys*, 28(2):333–336, 1996. 153, 155
10. T. Frühwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press. 154
11. J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximation of Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press. 153
12. R. Giacobazzi and F. Scozzari. Intuitionistic implication in abstract interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189. Springer-Verlag, 1997. 153, 155
13. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992. 153
14. T. Kanomori and K. Horiuchi. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Logic Programming 87- Tokyo*, volume 315 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, 1988. 153
15. M. Kifer and J. Wu. A first–order theory of types and polymorphism in logic programming. In *IEEE Symposium on logic in Computer Science*, 1991. 152
16. C. Pyo and U. S. Reddy. Inference of Polymorphic Types for Logic Programming. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 1115–1132. The MIT Press, 1989. 152
17. F. Scozzari. Logical optimality of groundness analysis. In P. Van Hentenryck, editor, *Proceedings of International Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997. 153, 155, 156, 159
18. J. Xu and D. S. Warren. A Type Inference System for Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 604–619. The MIT Press, 1988. 153
19. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10:125–135, 1991. 153

Architecturing Software Using a Methodology for Language Development

Charles Consel and Renaud Marlet

IRISA/INRIA - University of Rennes 1
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
{consel,marlet}@irisa.fr
http://www.irisa.fr/compose

1 Introduction

Domain-specific languages (DSLs) can be viewed from both a programming language and a software architecture perspective. The goal of this paper is to relate the two viewpoints. In particular, we demonstrate that DSLs can be constructed using an existing formal methodology for developing general purpose languages (GPLs) while expressing software architecture concerns.

1.1 A Programming Language Perspective

A DSL can be viewed as a programming (or specification) language dedicated to a particular domain or problem. It provides appropriate built-in abstractions and notations; it is usually small, more declarative than imperative, and less expressive than a GPL.

Consider for example the Unix command `make`. This tool is a utility to maintain programs: it determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them. The language of makefiles is small (at least in the early versions of `make`) and mainly declarative, although it also contains some imperative constructs. Its expressive power is limited to updating task dependencies; actual recompilation actions are delegated to a shell. It hides implementation details like file last-modification time and provides domain abstractions such as file suffixes and implicit compilation rules. As a result, the user may concisely express precise update dependencies.

This example illustrates several important DSL features, which make DSLs more attractive than GPLs for a variety of applications.

Easier programming. Because of appropriate abstractions, notations and declarative formulations, a DSL program is more concise and readable than its GPL counterpart. Hence, development time is shortened and maintenance is improved. As programming focuses on *what* to compute as opposed to *how* to compute, the user does not have to be a skilled programmer. For example, in the case of recompilation, writing a program to explicitly test all file modification times in order to incrementally rebuild a system would clearly be lengthy, tedious and error-prone.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 170–194, 1998.

Systematic re-use. Most GPL programming environments include the ability to group common operations into libraries. Though some are standard libraries, re-use is left to the programmer. On the other hand, a DSL offers guidelines and built-in functionalities which enforce re-use. Additionally, a DSL captures domain expertise, either implicitly by hiding common program patterns in the DSL implementation, or explicitly by exposing appropriate parameterization to the DSL programmer. Thus, any user necessarily re-uses library components and domain expertise.

Easier verification. Another important advantage of DSLs is that they enable more properties about programs to be checked. In contrast to a GPL, the semantics of a DSL can be restricted to make decidable some properties that are critical to a domain. For example, `make` reports any cycle in dependencies and thus totally prevents non-termination (assuming the individual actions do not loop).

Although all DSL features listed above address important software engineering concerns, they do not say much about the way applications based on DSLs should be structured. In fact, DSLs strongly suggest particular software architectures.

1.2 A Software Architecture Perspective

Software architectures express how systems should be built from various components and how those components should interact. From a software architecture perspective, a DSL can be seen as a parameterization mechanism as well as an interface model.

Parameterization mechanism. A program or a library can be more or less generic depending on the scope of the problems it addresses [3]. For example, a scientific library can be highly generic considering the vast variety of problems it can be used for. Pushing the idea of genericity further leads to complex parameters that can be seen as DSLs. For example, the format string argument of function `printf` can be seen as both a complex parameter and a very simple DSL. Considering a DSL program as a complex argument to a highly parameterized component may sound contrived but it actually is the final step of a chain of increasingly expressive power in parameterization. This situation is illustrated by Unix commands `grep`, `sort`, `find`, `sed`, `make`, `awk`, etc., and the progression from simple command-line parameters to program files. At the end of the spectrum, the data parameter ends up being a program to be processed, yielding increased parameterization power.

Interface to a library. As a library becomes larger or more generic, its usability decreases due to the multiplication of entry points, parameters and options offered. As a result, the library might be ignored by programmers

because it is too complex to use. In this situation, a DSL can offer a domain-specific interface to the library so that the programmer does not have to directly manipulate numerous highly-parameterized building blocks; the complexity is hidden. Another common situation is when some patterns of library calls occur frequently. In this case, a DSL interface can provide direct access to those commonly used combinations. For example, Unix shells are interfaces to standard Unix libraries. This idea is shared by scripting languages, that glue together a set of powerful components written in traditional programming languages. For example, Tcl/Tk provides a Tcl interface on top of the Tk graphic toolkit.

Recognizing a DSL as both a parameterization mechanism and an interface has an impact on structuring and reasoning about the software. In fact, the range of software adaptability is defined by the DSL. Such software is thus naturally separable into two parts: the decoding of the parameterization expressed by DSL programs and a library of components.

One may wonder when complex parameters and library interfaces are used. In the first case, complex parameters are introduced when, instead of offering separate but related tools, a single, versatile program is provided. In the second case, libraries are in essence created to enable re-use of data types and basic operations among related programs. Thus, the common motivation of those architectures is to build a set of related programs. This observation leads us to another, more conceptual aspect of DSLs: a program family.

Program family. A DSL program designates a member of a program family. A *program family* is a set of programs that share enough characteristics that it is worthwhile to study them as a whole [26]. A program family can also be seen as providing a solution to a *problem family*, *i.e.*, a set of related problems. Drivers, for a given type of device, form a natural example of a program family: in addition to having the same API (for a given operating system), they all share similar operations, although they vary according to the hardware.

1.3 When to Develop a DSL?

Conversely, we believe that whenever a problem family must be solved, *i.e.*, whenever a program family must be developed, basing the software architecture on a DSL makes configuration (*i.e.*, DSL programming) simpler. More generally, the following issues should be raised even when developing new software: does the program to be developed address an isolated problem? Could it be a member of a future program family?

The fact is that existing DSLs do implement program families. Examples are numerous; DSLs have been used in various domains such as graphics [12,19], financial products [1], telephone switching systems [13,21], protocols [4,35], operating systems [29], device drivers [37], routers in networks [35] and robot languages [2]. This profusion also shows the recent attention that DSLs have received from both the research and industrial communities.

1.4 How to Develop a DSL?

These applications have clearly illustrated the advantages of DSLs over GPLs, recording benefits such as productivity, reliability and flexibility [20]. However, they also raise a key issue which, if not addressed, could obstruct the use of DSLs [5]: how does one design and implement a DSL? Resolving this issue is critical to make the approach profitable since there is no point in reducing the complexity of program development by shifting all the complexity into the construction and maintenance of a DSL programming environment.

Another related question is: who will develop DSLs? Even in the programming language community, only a few people have actually designed a language. *A fortiori*, we cannot expect software engineers to have the full expertise to build up new languages. Thus, it is crucial that a methodology and tools are provided to make the DSL approach widely accessible.

1.5 Our Methodology for Developing DSLs

We propose a methodology for designing and implementing DSLs. This methodology is based on an existing formal framework for defining GPLs and integrates software architecture concerns.

This formal framework is based on *denotational semantics*, which has been extensively used to formally define GPLs [31]. It identifies key concepts in language design and semantics. Furthermore, techniques have been developed to derive implementations from definitions in denotational semantics [14]. These techniques typically produce compilers that are less efficient than ad hoc GPL compilers. However, in the context of DSLs, efficiency often relies on the underlying building blocks. As will be shown in this paper, structuring a DSL definition allows these building blocks to be isolated and implemented efficiently.

Our methodology is based on a framework outlined in an earlier paper by Thibault and Consel [33]. It can be summarized as follows. For the sake of clarity, the phases of our methodology are presented sequentially. In practice, the whole process needs to be *iterated*. Notice that the working example used to illustrate each phase throughout the paper is the final result of this iteration.

Language analysis. Assuming a problem family has been identified, the first step is to analyze the commonalities and the variations in the corresponding program family. This analysis is fueled by domain knowledge. The result of this analysis includes a description of objects and operations that are needed to express solutions to the family of problems, as well as language requirements (*e.g.*, analyzability) and elements of design (*e.g.*, notations).

Interface definitions. The next phase is to refine the design elements of the DSL. To do so, the syntax of the DSL is defined and its informal semantics is developed. The informal semantics relates the syntactic constructs to the objects and operations (*i.e.*, the building blocks) identified previously. Additionally, the domain of objects and the type of operations are formalized, thus forming the signature of semantic algebras.

Staged semantics. The semantics of a GPL is typically split between the compile-time and the run-time actions. These two parts are also referred to as the *static* and the *dynamic semantics* of a language. We propose to perform the same separation in the semantics of a DSL. With respect to software architecture concerns, this separation makes stages of configuration explicit.

Formal definition. Once the static and dynamic components of the language have been determined, the DSL is formally defined. Valuation functions define the semantics of the syntactic constructs. They specify how the operations of the semantic algebras (*i.e.,* the building blocks) are combined.

Abstract machine. Then, the dynamic semantic algebras are grouped to form a dedicated abstract machine which models the dynamic semantics of the DSL. From the denotational semantics, the DSL is given an interpretation in terms of this abstract machine. The state of the semantics is globalized and mapped into abstract machine entities (*e.g.*, registers) dedicated to the program family.

Implementation. The abstract machine is then given an implementation (typically, a library), or possibly many, to account for different operational contexts. The valuation function can be implemented as an interpreter based on an abstract machine implementation, or as a compiler to abstract machine instructions.

Partial evaluation. While interpreting is more flexible, compiling is more efficient. To get the best of both worlds, we use a program transformation technique, namely, partial evaluation, to automatically transform a DSL program into a compiled program, given only an interpreter.

Each of the above methodology steps is further detailed in a separate section of this paper.

1.6 A Working Example

To illustrate our approach, an example of DSL is used throughout the paper. We introduce a simple electronic mail processing application as a working example. Conceptually this application enables users to specify automatic treatments of incoming messages depending on their nature and contents: dispatching messages to people or folders, filtering spam, offering a shell escape (*e.g.*, to feed an electronic agenda), replying to messages when absent, etc.

This example is inspired by a Unix program called `slocal` which offers users a way of processing inbound mail. With `slocal`, user-defined treatments are expressed in the form of rules. Each rule consists of a string to be searched in a message field (*e.g.*, `Subject`, `From`) and an action to be performed if the string is found. Each rule stands on a single line and the whole specification is a flat series of rule lines, as opposed to a structured program.

This simple application illustrates the situation where a family of problems has to be handled: addressing different needs for the treatment of messages.

One could imagine a combination of various GPL programs being written to address each kind of treatment. This would form a family of programs which would most likely rely on a dedicated library. This library would consist of basic operations such as parsing a message, accessing and modifying message header fields, archiving and sending messages, etc.

We present a DSL solution to this problem family. More precisely, we show how to design and implement MAILSH, a simple DSL aimed at specifying the automatic treatment of incoming e-mails. Some details are left out of the following discussion. Our goal is to illustrate our methodology, not to propose an alternative to `slocal`.

2 Language Analysis

In the first phase of our approach, we analyze the problem family. During this analysis, the commonalities (shared features and assumptions that hold for all family members) and variabilities (variations in behavior and assumptions that differ among family members) must be identified. The analysis takes into account domain knowledge such as technical literature, existing programs, and current and future requirements. It can be conducted using methodologies used for *commonality analysis*, such as FAST [40,11], and *domain analysis* [24,25,28]. The main results of this analysis phase are: language requirements, a description of the common objects and operations, and design elements of the DSL. We examine each of these items in turn and illustrate them with our working example.

2.1 Language Requirements

Analyzing the family of problems leads to requirements for the language. Those requirements mainly consist of the functionalities that must be expressible in the DSL. Requirements also include language constraints (*e.g.*, domain issues such as safety and security) and implementation constraints (*e.g.*, resource bounds).

This phase does not differ much from a problem analysis that occurs when initiating any software development. The difference is that requirements are expressed in terms of language issues rather than general features of the application.

Working example. Concerning our message processing application, it should be possible, at the language level, to copy, move, delete, forward, pipe to a shell command, and reply to a message. Those actions should be triggered according to conditions depending on the inbound message. Those conditions should be string patterns matched against fields of the message.

Moreover, we have determined four language constraints. First, the user-defined treatments determined by a MAILSH program should not loop. Second, treatments should be guaranteed not to lose inbound messages. Third, inbound messages should not be duplicated in the same folder when archived. Fourth, automatically forwarding messages should not cause endless loops.

2.2 Objects and Operations

Identifying the common objects and operations essentially corresponds to defining the basic building blocks needed to express solutions for the family of problems. From a software architecture point of view, this process can be viewed as designing a library since it captures the common program patterns in the family and abstracts over the differences. The building blocks are grouped with respect to the objects they manipulate.

Working example. The program family analysis of our e-mail processing application results in the following fundamental objects and operations.

Messages. An electronic message consists of header fields and a body. We need operations to manipulate these message fields and to create new messages.

Folders. Folders contain a list of messages. Assuming we limit ourselves to dispatching messages, the only operation needed is to add a message to a folder.

Hierarchies of folders. A user typically has many folders to which (s)he directs messages, *e.g.*, according to topic or source. To cope with an increasing number of folders, e-mail systems offer the ability to create a folder hierarchy. To treat this feature in our system we need to associate an actual filename to a folder path in the folder hierarchy.

Files of Folders. Because of the layer introduced by the hierarchy of folders, the actual folders need to be captured by a separate object. Operations to read and write a folder from/to the file system need to be introduced.

Streams. Messages need to be sent, received or piped into a shell command. To model this, we need streams of inbound and outbound messages, as well as a command stream.

Miscellaneous. There are other, less fundamental objects and operations that we do not further detail here. This includes the ability to know the user's name (to send messages) and the current date (to timestamp the messages). There are also operations on booleans and strings, in particular a pattern matcher used in the message filtering condition.

2.3 Elements of Design

The last part of the language analysis phase consists of determining elements of the language design. These elements include the language paradigm (*e.g.*, declarative or imperative) as well as the language level: from low-level for expressivity, to high-level for usability. In addition, a terminology and notations are developed both from the domain and the set of problems to be addressed. These notations must correspond to the way domain experts express a solution, *i.e.*, a member of the problem family.

Working example. To apply this phase to our example, we have to introduce assumptions about the users of this message processing system. We assume such users to be typical Unix shell programmers. As a result, we decide the DSL

should be imperative like shell languages. Moreover, selection criteria should include regular expressions to achieve pattern matching in messages, as provided in the shell languages.

3 Interface Definitions

Given the information collected previously, we are now ready to develop a preliminary specification of the DSL. This preliminary specification consists of defining interfaces: the signature of semantic algebras and the DSL syntax. The semantics is kept informal; it will be made explicit in a later phase (see Section 5). Still, it allows taking into account some language requirements and to prepare the structuring of the actual language definition.

3.1 Semantic Algebras

The common objects and operations collected in the previous phase are now grouped with respect to the objects they manipulate to produce abstract data types. In the denotational framework, this form of abstract data types can be formalized as semantic algebras. A semantic algebra formally defines a domain (*i.e.*, a structured value space) and the operators on that domain [31]. At this stage, we only provide signatures; we postpone details until a complete view of basic building blocks is determined.

Working example. Let us illustrate the notion of semantic algebra with our message processing application. To do so, we present in Figure 1 the signature of semantic algebras which follow the common objects and operations determined earlier in Section 2.1.

Messages. Function *msg-to-string* converts a message into a string. This function is used when piping a message into a shell command and when forwarding a message. In the latter case, the body of the new message (a string) contains the forwarded message. *FieldName* is defined as *String*.

Folders. We consider a folder as an ordered list of messages; function *add-msg* adds a message at the end of this list. There are other obvious common operations on folders; we do not mention them here as they are not needed for our example.

Hierarchies of folders. Function *get-filename* maps a folder path into a filename. Note that the folder hierarchy may define aliases: two paths may be mapped into the same filename.

Files of Folders. There are several common implementations of a folder, depending on the user's mailing system. We let actual implementations of abstract operators *read-folder* and *write-folder* deal with that.

Streams. Function *next-msg* reads the next incoming message. Note that an implementation of it must not return until a new message has arrived, thus suspending the application. Function *send-msg* ships a message to the system stream. Function *pipe-msg* passes a string to the standard input of a command. We also define *CmdString* as *String*.

Messages
Domain: *Message*
Operations:
new-msg : *Message*
get-field : *FieldName* → *Message* → *String*
set-field : *FieldName* → *String* → *Message* → *Message*
get-body : *Message* → *String*
set-body : *String* → *Message* → *Message*
msg-to-string : *Message* → *String*

Folders
Domain: *Folder*
Operations:
add-msg : *Message* → *Folder* → *Folder*

Hierarchy of Folders
Domain: *FolderHierarchy*
Operations:
get-filename : *FolderPath* → *FolderHierarchy* → *FileName*

Files of Folders
Domain: *FolderFiles*
Operations:
read-folder : *FileName* → *FolderFiles* → *Folder*
write-folder : *FileName* → *Folder* → *FolderFiles* → *FolderFiles*

Streams
Domains: *InStream*, *OutStream*, *CmdStream*
Operations:
next-msg : *InStream* → (*Message* × *InStream*)
send-msg : *Message* → *OutStream* → *OutStream*
pipe-msg : *Message* → *CmdString* → *CmdStream* → *CmdStream*

Fig. 1. Signature of the main semantic algebras for MAILSH

$B \in \mathit{BoolExpr}$
$C \in \mathit{Command}$
$F \in \mathit{FolderPath}$
$S \in \mathit{String}$

$B ::= \texttt{match}\ S_{\mathsf{field}}\ S_{\mathsf{pat}}$
$\quad \mid \texttt{not}\ B \mid B_1\ \texttt{and}\ B_2 \mid B_1\ \texttt{or}\ B_2$

$C ::= C_1\ \texttt{;}\ C_2$
$\quad \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$
$\quad \mid \texttt{skip}$
$\quad \mid \texttt{delete}$
$\quad \mid \texttt{copy}\ F$
$\quad \mid \texttt{forward}\ S_{\mathsf{to}}$
$\quad \mid \texttt{reply}\ S_{\mathsf{body}}$
$\quad \mid \texttt{pipe}\ S_{\mathsf{cmd}}$

Fig. 2. Abstract syntax of MAILSH

Miscellaneous. We do not detail here other miscellaneous semantic algebras. We will later only explicitly use $match : String \to StringPattern \to Bool$ as the pattern matching operator.

3.2 The DSL Syntax

The syntax of the DSL is defined based on information collected earlier, namely, the language requirements (functionalities as well as constraints) and the design elements (language paradigm, language level, terminology and notations). It may also explicitly refer to some of the objects and operations identified as fundamental; the others remain hidden in the underlying semantics. Intuitively, a syntactic construct in the DSL corresponds to a pattern of operations.

One of the key issues in designing the abstract syntax (*i.e.*, the interface of the language) is to restrict the programmability so that required properties are provable. At the same time, raising the level of the DSL may hinder future needs for expressiveness. For example, a common practice to ensure the termination of DSL programs is to provide the programmer only with restricted loop constructs, if any, so that the property can be syntactically checked. Issues regarding the design of a concrete syntax are beyond this work.

As the DSL syntax is developed, its semantics is informally defined. This preliminary definition allows the semantic algebras to be further refined.

Working example. The requirements were that MAILSH should express conditional treatment of incoming messages, be imperative and close to Unix shells. Figure 2 presents the BNF definition of an abstract syntax which fulfills those requirements. Folders are an example of a domain-specific object explicitely referred to by the syntax via folder paths. In contrast, folder filenames remain hidden. For the sake of simplicity, we have intentionally reduced this DSL to a kernel language, rich enough to allow us to illustrate the various aspects of our approach. Obviously, to make it usable, more constructs and actions on messages should be added. For example, the following abbreviations could be provided:

- `move` F $\equiv$ `copy` F `; delete`
- `if` B `then` C $\equiv$ `if` B `then` C `else skip`

A concrete syntax close to Unix shells can easily be developed.

We shall not comment here on the semantics of the various constructs of the language since most of them are self-explanatory. An example program is given in Figure 3. (Indentation emphasizes the nesting of constructs.) Note that the `reply` construct can be used to setup a `vacation`-like tool, *i.e.*, a message-sensitive answering machine.

It must be noted that the language only specifies the treatment of a single message; there is an implicit loop over the inbound messages. This kind of treatment encapsulation is typical to DSLs. Common examples are text processing DSLs, like `sed`, that assume an implicit loop over each line of the text input.

Before providing a formal definition for the language and tackling its implementation, a staging phase is required to separate the language semantic entities.

```
if match "Subject" "DSL" then
  forward "jake";
    copy Research.Lang.DSL; delete
else if match "From" "hotmail.com" then
       reply "Leave me alone!"; delete
     else if match "Subject" "seminar" then
            pipe "agenda --stdin"; delete
          else
            skip
```

Fig. 3. Abstract syntax of a MAILSH program

4 Staged Semantics

From a programming language viewpoint, the semantics of a GPL is traditionally split into two parts: the static and the dynamic semantics. In practice, the static semantics of a language corresponds to the actions performed by a compiler; these actions are thus the ones which depend on the program being compiled. The dynamic semantics represents the computations which may depend on the input data of the program. Necessarily, these computations must be postponed until run time.

Because a compiler processes the static semantics of a language with respect to a given program, it is in effect a syntax-to-dynamic-semantics mapping [10]. Concretely, the dynamic semantics is a compiled program; it consists of a combination of instructions for a machine (either abstract or concrete).

From a software architecture viewpoint, the static semantics corresponds to computations that determine the member of a program family. The dynamic semantics corresponds to computations that produce the answer to the corresponding problem, *i.e.*, program execution.

From an implementation viewpoint, processing the static semantics of an application can be seen as configuring generic software with respect to a given context. More concretely, configuring amounts to processing the static (*i.e.*, available) information in order to select the appropriate components, and combining them to produce a customized software. Then, processing the dynamic semantics consists of executing the customized software.

Determining staging addresses an important concern in software architecture: reasoning about the genericity of software to predict and control its customization. This is a key step towards reconciling flexibility, as promoted by many approaches to software architecture, and performance. Indeed, inefficiency is a well-known limitation of many of these approaches [23].

We propose to address the staging of a DSL semantics, or equivalently the configuration of its software architecture counterparts, using a language approach. At this point of our methodology, the staging process is limited to semantic algebras, instead of being applied to the complete DSL definition. Later,

when providing an actual definition (the valuation functions), more staging issues will also have to be considered. For the moment, from initial staging constraints coming from the problem family, we introduce staging in the semantic algebras and in the treatment of language constraints.

4.1 Initial Staging Constraints

To achieve the staging of a DSL semantics, the initial step is to determine the *semantic arguments* of the valuation functions which can be assumed to be static (*i.e.* known) given the problem family. These static arguments can be seen as configuration arguments. Just like GPLs, the static semantics of DSLs assumes that the program is available (*i.e.*, static).

Working example. Considering our message processing application, we assume that the static arguments are the DSL program representing the user-defined treatments, the folder hierarchy of the user, and the user's name.

4.2 Staging the Semantic Algebras

Given this initial staging, the semantic algebras need to be analyzed to determine which ones correspond to configuration (*i.e.*, static) computations and which ones define *actual* (*i.e.*, dynamic) computations. For example, in the context of a GPL, a semantic algebra which maintains type information on program variables is typically a static algebra when the language is strongly typed.

For a given DSL, the staging process should answer the following question on each semantic algebra: should this value domain, and its corresponding operations, be static or dynamic?

Working example. Given that the folder hierarchy is a static initial argument to MAILSH, it should remain unchanged throughout the semantics since our DSL does not provide a way to modify this hierarchy. Therefore, the semantic algebra for the hierarchy of folders should be static as well; operations on such values should be processed completely statically.

The other semantic algebras of our DSL are intrinsically dynamic since they rely on values assumed to be known only at run time, *i.e.*, inbound messages.

4.3 Staging the Language Constraints

Staging not only involves the language semantics but also the language constraints, which in turn has an impact on the semantics. Some constraints may be guaranteed statically, before the DSL program is run. Others may rely on run-time information and have to be checked when the DSL program is executed.

Working example. The first language constraint (see Section 2.1) is that the treatment of a message should not loop. This constraint is syntactically enforced given that there is no iteration construct.

The second constraint states that user-defined treatments should be guaranteed not to lose inbound messages. An inbound message is lost when it is neither copied, forwarded, piped, replied-to nor explicitly deleted, *i.e.*, when it is skipped. For example, the program in Figure 3 can lose a message if none of the conditions applies. This constraint can be statically checked by analyzing the possible execution paths of a program with respect to the pattern matching conditions. (Proof omitted.)

The third language constraint can also be statically checked. (Proof omitted.) It states that inbound messages should not be duplicated in the same folder when archived.

The fourth language constraint says that automatically forwarding messages should not be able to cause endless loops. Because of unknown aliases and mailing lists, it is not possible to make sure that, if a message is forwarded, it will not eventually be forwarded back to the sender. This condition can only be checked dynamically by introducing a specific mechanism.

An additional constraint has not been expressed yet because it depends more on the structure of the language than on the domain: it should not be possible to operate on a message after it has been deleted. This amounts to checking paths where there exist message treatments after a `delete` invocation. This property can be statically checked. (Proof omitted.) We make the decision to reject any program not satisfying this property. As we will see, not only does it prevent us from specifying error handling in the dynamic semantics, but it also simplifies the implementation of `delete`, turning it into a mere `skip`.

5 Formal Definition

We now have all the necessary elements to formally define the semantics of a DSL. Fundamentally, the denotational definition represents a guide for the language implementer and a key source of documentation. By postponing implementation issues to a later phase, the DSL developer can better stage decisions. For example, the data layout of objects can be postponed until hardware features are known.

As is customary, the denotational semantics is composed of three parts: the abstract syntax (see Section 3.2), the semantic algebras (see Section 3.1), and the valuation functions. In contrast to the informal semantics given previously, the semantic algebras are now completely specified, including the definition of their operations.

5.1 Semantic Arguments

Valuation functions are inductively defined on the abstract syntax. Besides the program text, a valuation function includes other semantic arguments which define the semantic context. The semantic arguments are drawn from the semantic algebras introduced earlier.

Working example. The semantic arguments in the case of MAILSH are the folder hierarchy, the message being treated, the folder files, the streams and other miscellaneous entities like the current date and the user's name.

5.2 Staging the Semantic Arguments

Beyond the semantic algebras, the valuation functions must further separate the DSL semantics into its static and dynamic parts. To do so, we keep separate the static and dynamic semantic arguments of the valuation functions. This separation is guided by the binding time (static / dynamic) of the semantic algebras determined previously.

Working example. As for MAILSH, the static semantic arguments are the folder hierarchy and the user's name; these are grouped into a product domain named *StaticState*. The dynamic arguments are the message being treated, the folder files, the streams and the current date.

We use the following notations: the tuple projection on the domain X (*e.g.*, *Message*) of $\sigma \in \mathit{DynamicState}$ is denoted σ_{x} (*e.g.*, $\sigma_{\mathsf{message}}$). Updating the X element of the tuple σ with a value y is denoted $[\mathsf{x} \mapsto y]\,\sigma$.

5.3 Control Staging and Dynamic Combinators

The computations described by the valuation functions also need to be staged. The basic operations used by a valuation function have a binding time that has been determined in the previous phase; only the control operations remain to be staged. To do so, the separation between static and dynamic control operations must be made explicit. We thus introduce combinators for the dynamic control operations; these combinators are later turned into control instructions in the abstract machine. Static control operations need not be associated with an explicit combinator.

Working example. The conditional statement `if` B `then` C_1 `else` C_2 is dynamic because it depends on the message to be treated via the `match` construct. We thus introduce a choice function as an explicit *cond* combinator.

5.4 Valuation Functions

The valuation functions may finally be defined. Complete definition of the semantic algebras should be provided at this stage as well.

Working example. Remember that the processing of messages is always active and should be modeled by an infinite loop aimed at polling the stream of inbound messages. This loop must further rely on function *next-msg*, which suspends the message processing application if no inbound message is available. When some messages are received, the dynamic semantic arguments are set up and the valuation function $\mathcal{C}$ is applied to the program, *i.e.*, a possibly structured command.

$$\mathcal{C} : \mathit{Command} \to \mathit{StaticState} \to \mathit{DynamicState} \to \mathit{DynamicState} \quad \text{where}$$
$$\mathit{StaticState} = \mathit{FolderHierarchy} \times \mathit{UserName}$$
$$\mathit{DynamicState} = \mathit{FolderFiles} \times \mathit{OutStream} \times \mathit{CmdStream} \times \mathit{Date} \times \mathit{Message}$$

$$\mathcal{C}\ [\![C_1\ ;\ C_2]\!]\ \rho = (\mathcal{C}\ [\![C_2]\!]\ \rho) \circ (\mathcal{C}\ [\![C_1]\!]\ \rho)$$

$$\mathcal{C}\ [\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ \rho = \mathit{cond}\ (\mathcal{B}\ [\![B]\!])\ (\mathcal{C}\ [\![C_1]\!]\ \rho)\ (\mathcal{C}\ [\![C_2]\!]\ \rho)$$

$$\mathcal{C}\ [\![\texttt{skip}]\!]\ \rho\ \sigma = \sigma$$

$$\begin{aligned}
&\mathcal{C}\ [\![\texttt{copy}\ F]\!]\ \rho\ \sigma = \\
&\quad \text{let}\ \nu = \mathit{get\text{-}filename}\ (\mathcal{F}\ [\![F]\!])\ \rho_{\text{folder-hierarchy}} \\
&\qquad \varphi = \mathit{add\text{-}msg}\ (\mathit{set\text{-}field}\ \text{“}\texttt{Delivery-Date}\text{”}\ \sigma_{\text{date}}\ \sigma_{\text{message}}) \\
&\qquad\qquad (\mathit{read\text{-}folder}\ \nu\ \sigma_{\text{folder-files}}) \\
&\quad \text{in}\ \ [\text{folder-files} \mapsto \mathit{write\text{-}folder}\ \nu\ \varphi\ \sigma_{\text{folder-files}}]\ \sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}\ [\![\texttt{forward}\ S]\!]\ \rho\ \sigma = \\
&\quad [\text{out-stream} \mapsto \mathit{send\text{-}msg} \\
&\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{Resent-by}\text{”}\ (\mathit{concat}\ \rho_{\text{user-name}}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Resent-by}\text{”}\ \sigma_{\text{message}})) \\
&\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ (\mathit{concat}\ \text{“}\texttt{Fwd: }\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ \sigma_{\text{message}})) \\
&\qquad\qquad (\mathit{set\text{-}body}\ (\mathit{msg\text{-}to\text{-}string}\ \sigma_{\text{message}}) \\
&\qquad\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{From}\text{”}\ \rho_{\text{user-name}} \\
&\qquad\qquad\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{To}\text{”}\ (\mathcal{S}\ [\![S]\!]) \\
&\qquad\qquad\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{Date}\text{”}\ \sigma_{\text{date}}\ (\mathit{new\text{-}msg})))))))\ \sigma_{\text{out-stream}}]\ \sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}\ [\![\texttt{reply}\ S_1]\!]\ \rho\ \sigma = \\
&\quad [\text{out-stream} \mapsto \mathit{send\text{-}msg} \\
&\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ (\mathit{concat}\ \text{“}\texttt{Re: }\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ \sigma_{\text{message}})) \\
&\qquad\quad (\mathit{set\text{-}body}\ (\mathcal{S}\ [\![S_1]\!]) \\
&\qquad\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{From}\text{”}\ \rho_{\text{user-name}} \\
&\qquad\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{To}\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{From}\text{”}\ \sigma_{\text{message}}) \\
&\qquad\qquad\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{Date}\text{”}\ \sigma_{\text{date}}\ (\mathit{new\text{-}msg})))))\ \sigma_{\text{out-stream}}]\ \sigma
\end{aligned}$$

$$\mathcal{C}\ [\![\texttt{pipe}\ S]\!]\ \rho\ \sigma = [\text{cmd-stream} \mapsto \mathit{pipe\text{-}msg}\ \sigma_{\text{message}}\ (\mathcal{S}\ [\![S]\!])\ \sigma_{\text{cmd-stream}}]\ \sigma$$

$$\mathcal{B} : \mathit{BoolExpr} \to \mathit{DynamicState} \to \mathit{DynamicState}$$

$$\mathcal{B}\ [\![\texttt{match}\ S_1\ S_2]\!]\ \sigma = \mathit{match}\ (\mathit{get\text{-}field}\ (\mathcal{S}\ [\![S_1]\!])\ \sigma_{\text{message}})\ (\mathcal{S}\ [\![S_2]\!])$$

Fig. 4. Valuation functions for MAILSH

Figure 4 shows the definition of valuation function $\mathcal{C}$. The `delete` construct does not appear in the definition of $\mathcal{C}$ because it is replaced by a `skip` after analysis (see Section 4). The definition of the other valuation functions and the semantic algebras of our DSL are omitted since they are rather simple and do not raise issues with respect to our approach. Setting the "`Resent-by`" field when forwarding allows the encapsulating loop to discard incoming messages that have already been forwarded by the user, thus dynamically verifying the fourth constraint expressed in the language requirements.

Common dynamic patterns of operations in the right-hand side of the semantic equations can be encapsulated into new operators. For example, composing a message for the forward or reply operations shares dynamic operations that could have been grouped into a single higher-level operator.

6 Abstract Machine

Although the valuation functions make a clear separation between the static and dynamic semantics of the DSL, we still have to further encapsulate the dynamic semantics to define a dedicated abstract machine. This is a key step to derive a realistic implementation from the DSL definition. The abstract machine roughly corresponds to the library in a conventional software architecture. However, it is not yet the implementation (see Section 7).

Another benefit of the approach is that it provides a formal model of computation that can be reasoned about using well-established techniques for abstract machines [27]. In fact, the abstract machine offers a model of computation that underlies all programs in the family [5]. The abstract machine model also provides the right level of decomposition to increase reuse of the abstract machine [39]. In particular, since an abstract machine can express a wide range of applications within the domain, and a DSL only a restricted subset of these, several DSLs could share the same abstract machine. For example, it is useful to have multiple DSLs for different users; a DSL could thus manage a whole database while a subset of this DSL might only be able to express queries.

6.1 Single-Threadedness and Globalization

The key issue in expressing a semantics in terms of an efficient abstract machine is the globalization of the dynamic semantic arguments. To enable semantic arguments to be made implicit in the actual implementation, they cannot be manipulated in an arbitrary way by the denotational definition. Schmidt and others have developed specific criteria which allow semantic arguments to be globalized when deriving an implementation from a denotational definition [31]. If these criteria are fulfilled by the denotational definition for a given semantic argument, then the denotational definition is said to be *single-threaded* in this semantic argument. A precise definition of these criteria is beyond the scope of this paper.

If a semantics definition is single-threaded in a dynamic state argument, this argument can be globalized. For example, in case of an imperative GPL, the store is a typical semantic argument which gets globalized in an implementation of its dynamic semantics. Indeed, the store corresponds to the processor memory and thus does not need to be passed explicitly since it is globally available. In the case of a DSL, there may be various semantic arguments which need to be globalized in an actual implementation. This is one of the aspects which reflects the dedicated nature of the abstract machine of DSL.

Note that, when the dynamic state is globalized, abstract machine instructions which perform a state transition are *linearized*.

Working example. The semantics example given in Figure 4 is already single-threaded. Thus, the dynamic semantics arguments can be globalized.

6.2 Abstract Machine Entities

Our goal is to develop an abstract machine dedicated to the dynamic computations of the DSL, based on the semantic algebras. To facilitate this process, the dynamic parts of the semantic context need to be grouped in a unique semantic argument which prefigures the basic entities of the abstract machine (*e.g.*, registers).

Because all of the dynamic context is passed as a unique argument to the valuation functions, the operations in the semantic algebras no longer need to be passed to the dynamic semantic arguments separately; they can be transformed so as to get these values from a unique argument.

Working example. The valuation functions shown in Figure 4 already group the dynamic semantic arguments passed to the valuation functions into a unique semantic argument, namely *DynamicState*. These arguments naturally correspond to registers of the abstract machine. We group them in the domain *AbsMachState*.

In addition, consider the *new-msg* operator. It returns a fresh, new message whose fields are later assigned dynamically. However, from an operational viewpoint, only two messages may exist at any time: the current inbound message and a message being composed (two new messages cannot be composed at the same time). To make globalization more explicit, we dedicate an extra register of the abstract machine for the message being currently composed. The operator *new-msg* : *Message* thus becomes the abstract machine instruction new-msg : $AbsMachState \rightarrow AbsMachState$ which operates indirectly on this new register.

In making operators like *set-field* and *get-field* implicitly access the dynamic registers, an ambiguity has appeared because we now have two registers for messages. To make the message register explicit, we denote get-field_i the instruction that accesses the inbound message and get-field_c the instruction that accesses the message being composed. In an actual implementation, this may be modeled as an argument to the instructions (*e.g.*, a pointer to the actual message).

The resulting semantic definition based on the abstract machine is given in Figure 5.

7 Implementation

The implementation of a DSL can be derived from the implementation of its valuation function and an implementation of the corresponding abstract machine. Like GPLs, DSLs can either be implemented by an interpreter or a compiler. The abstract mahcine provides a portable layer.

7.1 Interpretation

The interpretation is usually the easiest implementation approach because it processes a program in the presence of its data, and thus directly produces an

$\mathcal{C} : \mathit{Command} \to \mathit{StaticState} \to \mathit{AbsMachState} \to \mathit{AbsMachState}$ where
$\mathit{StaticState} = \mathit{FolderHierarchy} \times \mathit{UserName}$
$\mathit{AbsMachState} = \mathit{FolderFiles} \times \mathit{OutStream} \times \mathit{CmdStream} \times \mathit{Date} \times \mathit{Message}_i \times \mathit{Message}_c$

$\mathcal{C}\ [\![C_1\ \texttt{;}\ C_2]\!]\ \rho = (\mathcal{C}\ [\![C_2]\!]\ \rho)\ \circ\ (\mathcal{C}\ [\![C_1]\!]\ \rho)$

$\mathcal{C}\ [\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ \rho = \mathsf{cond}\ (\mathcal{B}\ [\![B]\!])\ (\mathcal{C}\ [\![C_1]\!]\ \rho)\ (\mathcal{C}\ [\![C_2]\!]\ \rho)$

$\mathcal{C}\ [\![\texttt{skip}]\!]\ \rho = \mathsf{no\text{-}op}$

$\mathcal{C}\ [\![\texttt{copy}\ F]\!]\ \rho\ \sigma =$
  let $\nu = \mathsf{get\text{-}filename}\ (\mathcal{F}\ [\![F]\!])\ \rho_{\mathsf{folder\text{-}hierarchy}}$
  in $((\mathsf{write\text{-}folder}\ \nu) \circ$
    $(\mathsf{add\text{-}msg}) \circ$
    $(\mathsf{set\text{-}field}_i\ \text{“}\texttt{Delivery-Date}\text{”}\ \sigma_{\mathsf{date}}) \circ$
    $(\mathsf{read\text{-}folder}\ \nu))\ \sigma$

$\mathcal{C}\ [\![\texttt{forward}\ S]\!]\ \rho\ \sigma =$
  $((\mathsf{send\text{-}msg}) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{Resent-by}\text{”}\ (\mathsf{concat}\ \rho_{\mathsf{user\text{-}name}}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Resent-by}\text{”}\ \sigma))) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{Subject}\text{”}\ (\mathsf{concat}\ \text{“}\texttt{Fwd: }\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Subject}\text{”}\ \sigma))) \circ$
  $(\mathsf{set\text{-}body}_c\ (\mathsf{msg\text{-}to\text{-}string}_i\ \sigma)) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}}) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{To}\text{”}\ (\mathcal{S}\ [\![S]\!])) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}) \circ$
  $(\mathsf{new\text{-}msg}_c))\ \sigma$

$\mathcal{C}\ [\![\texttt{reply}\ S_1]\!]\ \rho\ \sigma =$
  $((\mathsf{send\text{-}msg}) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{Subject}\text{”}\ (\mathsf{concat}\ \text{“}\texttt{Re: }\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Subject}\text{”}\ \sigma))) \circ$
  $(\mathsf{set\text{-}body}_c\ (\mathcal{S}\ [\![S_1]\!])) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}}) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{To}\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{From}\text{”}\ \sigma)) \circ$
  $(\mathsf{set\text{-}field}_c\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}) \circ$
  $(\mathsf{new\text{-}msg}_c))\ \sigma$

$\mathcal{C}\ [\![\texttt{pipe}\ S]\!]\ \rho = \mathsf{pipe\text{-}msg}\ (\mathcal{S}\ [\![S]\!])$

$\mathcal{B} : \mathit{BoolExpr} \to \mathit{AbsMachState} \to \mathit{AbsMachState}$

$\mathcal{B}\ [\![\texttt{match}\ S_1\ S_2]\!]\ \sigma = \mathsf{match}\ (\mathsf{get\text{-}field}_i\ (\mathcal{S}\ [\![S_1]\!])\ \sigma)\ (\mathcal{S}\ [\![S_2]\!])$

Fig. 5. Abstract-machine-based semantic definition of MAILSH

answer. In contrast, a compiler produces a program which, when executed, produces a result. Thus, the compiler approach introduces an indirection which makes it more difficult to develop. Another advantage of the interpretation approach is that interpreters can often be derived from the denotational definition by directly translating the specification into a functional program.

The interpretation approach is also well known for its flexibility. For example, there are existing techniques to extend interpreters (*e.g.*, based on monads [22,32,38]) without corrupting the semantics. More generally, interpretation allows languages to be prototyped rapidly and thus language design can be very reactive. This feature is particularly important in the context of a DSL; given that needs in the domain may evolve over time, so should the DSL. The obvious limitation of the interpretation approach is inefficiency. Depending on the language, interpretation has been commonly cited to be one order of magnitude slower than compiled code [30].

7.2 Compilation

From a software engineering viewpoint, a DSL compiler can be seen as an application generator in that it processes a specification to generate an application. Traditional compilation could be applied to a DSL; that is, native code could be directly produced from a DSL program. Developing a compiler which generates efficient code should not require more effort than for a GPL, considering the restricted nature of a DSL.

Another compilation strategy consists of producing abstract machine instructions from a DSL program. In doing so, the staged semantics is exploited to allow more flexibility in the implementation of the abstract machine.

7.3 Abstract Machine Implementation

As for efficiency, the abstract machine layer should cause a negligible overhead given that each instruction often captures substantial dynamic computations. Therefore, if there exist efficient compilers for the implementation language of the abstract machine, little (if any) overhead should be incurred compared to natively-compiled programs.

Depending on the implementation language which *glues* the abstract machine instructions, the valuation functions may however need linearization. If the implementation language has expressions, the abstract machine code may stay structured: the reason is that the implementation language compiler would linearize them anyways. In our example, these are the expressions involving msg-to-string, concat, get-field, etc. On the contrary, linearizing these instructions further would have been useless, at best, and an obstacle for optimizing compilation, at worst. Indeed, higher-level machine instructions may expose more optimization opportunities than instructions where early operational choices have been made. If the implementation language is flat (*e.g.*, assembly, JVM), then linearization is necessary. However, linearization does not go beyond state transition boundaries.

7.4 The Abstract Machine as an API

Although only a single implementation of the valuation functions is typically needed, there might be several implementations of the same abstract machine, to account for different operational contexts.

Working example. To illustrate the flexibility offered by a staged implementation, let us examine the MAILSH folders. There are several common implementations of a folder, either as a single file being the (formatted) concatenation of messages (*e.g.*, Netscape or GNU Emacs) or as a directory containing one file per message (*e.g.*, `exmh`). We abstracted over these implementation choices by introducing the domains *FileName* and $FolderFiles$.

8 Partial Evaluation

In the previous section, two separate implementation approaches were presented, namely, interpretation and compilation. In this section we propose a third approach which allows compilation to be achieved from an interpreter. This approach relies on partial evaluation [6,16,17]. It consists of developing an interpreter based on the staged semantics of the DSL. Then, a partial evaluator is applied to the interpreter and a given DSL program to process the static semantics. That is, it performs the static computations of the interpreter and produces code for the dynamic computations, as a compiler would do.

Partial evaluation has traditionally been used to specialize an interpreter by removing the interpretation layers [10,18]. More generally, it has been shown to successfully optimize the implementation of various software architectures [7,23]. In the context of DSLs, partial evaluation has been used to successfully optimize DSL interpreters, as demonstrated by GAL, a language to specify device drivers for PC graphics card. Thibault *et al.* have reported that the GAL interpreter can be specialized with respect to a driver specification (known at compile time) to yield an implementation as efficient as an equivalent, hand-written device driver [34].

In addition, partial evaluation has been successfully used to specialize interpreters at *run time*, *i.e.*, with respect to a DSL program not known until run time. This work has been done in the context of PLAN-P, a DSL for active networks [36]. When the PLAN-P interpreter is specialized at run time with respect to a PLAN-P program, the resulting code incurs no overhead in overall system performance in comparison with hand-written C code. Furthermore, in comparison with Java, another mobile code approach, the specialized program is twice as fast as an equivalent Java program compiled with an optimizing off-line byte-code compiler. In effect, run-time specializing interpreters achieve the same functionality as a *Just-In-Time* compiler for the price of an interpreter. Moreover, unlike specialized GPL interpreters, which compete with optimizing compilers producing fine-grained, low-level operations, specialized DSL interpreters can yield domain-specific, coarse-grained operations where the need for efficiency often resides.

```
cond (match (get-field_i "Subject") "DSL")(
  new-msg;
  set-field_c "Date" (date);
  set-field_c "To" "jake";
  set-field_c "From" "bob";
  set-body_c  (msg-to-string_i);
  set-field_c "Subject" (concat "Fwd: " (get-field_i "Subject"));
  set-field_c "Resent-by" (concat "bob" (get-field_i "Resent-by"));
  send-msg;
  read-folder "/home/bob/Mail/Research/Lang/DSL";
  set-field_c "Delivery-Date" (date);
  write-folder "/home/bob/Mail/Research/Lang/DSL"
)(
  cond (match (get-field_i "From") "hotmail.com")(
    new-msg;
    set-field_c "Date" (date);
    set-field_c "To" (get-field_i "From");
    set-field_c "From" "bob";
    set-body_c  "Leave me alone!";
    set-field_c "Subject" (concat "Re: " (get-field_i "Subject"));
    send-msg;
  )(
    cond (match (get-field_i "Subject") "seminar")(
      pipe-msg "agenda --stdin"
    )(
      no-op)))
```

Fig. 6. Implementation of a MAILSH program example

Notice that the specialization of both DSL interpreters (GAL and PLAN-P) were done using Tempo [8,9], a partial evaluator for the C language developed by the Compose group (`http://www.irisa.fr/compose/tempo`).

Working example. Recall the example of a MAILSH program presented in Figure 3. We have taken its denotation and performed all the reductions made possible by the availability of both the program and folder hierarchy. Unlike GAL and PLAN-P, this was done by hand; MAILSH has not been implemented. The resulting term is presented in Figure 6. To illustrate the globalization phase, the dynamic state is eliminated from the reduced denotation; the composition of the state-transforming instructions is noted with a semicolon. As can be noticed, the result is quite close to an imperative program. This representation could be transformed into a very efficient C program for example.

9 Conclusion

DSLs have been successfully used to address software engineering concerns in specific application domains. Yet, methodologies for language development have been focusing on GPLs, designed to be universal. In this paper, we have proposed an approach aimed at bridging the gap between these two perspectives. This approach is a complete software development process starting from the identification of the need for a DSL to its efficient implementation. It uses the denotational framework to formalize the basic components of a DSL. The semantics definition is structured so as to stage design decisions and to smoothly integrate implementation concerns. When implemented as an interpreter, partial evaluation is proposed as an optimization technique to remove the performance overhead. Our methodology builds on two successful developments of DSLs: GAL, a language to specify device drivers for graphics cards, and PLAN-P, a language to program routers.

Beyond a methodology to develop DSLs, we are now studying an approach to allowing one to assemble a DSL from parameterized building blocks. This work stems from the fact that, although specific to a domain, a DSL often includes common functionalities which could correspond to generic components. Providing these components in a DSL development environment could facilitate the work for non-experts in programming languages to develop their own DSL. A related topic involves the definition of properties about these components such that they could be safely composed when defining a new DSL.

Another avenue of research consists of exploring structuring techniques for the DSL definition to enable the derivation of DSL program analyzers. A departure point for this study would include factorized semantics as proposed by Jones and Nielson [15].

Finally, the methodology needs to be further validated by more applications. We plan on investigating other families of problems to develop new DSLs. To do so, we are actively studying networking where various DSL candidates have been identified (*e.g.,* Web caching).

Acknowledgments

A substantial amount of the research reported in this paper builds on work done by the authors with Scott Thibault on DSLs. Another source of inspiration comes from work done by the first author with Olivier Danvy in the early nineties.

We thank Julia Lawall and Scott Thibault for thoughtful comments on earlier versions of this paper, as well as the Compose group for stimulating discussions.

References

1. B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995. 172

2. E. Bjarnason. Applab: a laboratory for application languages. In L. Bendix, K. Nrmark, and K sterby, editors, *Nordic Workshop on Programming Environment Research, Aalborg*. Technical Report R-96-2019, Aalborg University, May 1996. 172
3. Grady Booch. *Software Components with Ada*. Benjamin Cummings, 1987. 171
4. Satish Chandra and James Larus. Experience with a language for writing coherence protocols. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997. 172
5. J. Graig Cleaveland. Building application generators. *IEEE Software*, July 1988. 173, 185
6. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press. 189
7. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Partial evaluation for software engineering. *ACM Computing Surveys, Symposium on Partial Evaluation*, 1998. To appear. 189
8. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 1998. To appear. 190
9. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996. 190
10. C. Consel and Danvy O. Static and dynamic semantics processing. In *Conference Record of the Eighteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, Orlando, FL, USA, January 1991. ACM Press. 180, 189
11. D. Cuka and D. Weiss. Engineering domains: Executable commands as an example. In *Proc. Fifth Internationbal Conference on Software Reuse*, June 1998. 175
12. Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997. 172
13. N.K. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, pages 188–197, January 1997. 172
14. N. D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980. 173
15. N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program ana lysis. Technical report, University of Copenhagen and Aarhus University, Copenhagen, Denmark, 1990. 191
16. N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, sep 1996. 189
17. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993. 189
18. N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989. 189
19. Samuel Kamin and David Hyatt. A special-purpose language for picture-drawing. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997. 172

20. R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553, 1996. 173
21. David Ladd and Christopher Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994. 172
22. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343. ACM, January 1995. 188
23. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society. 180, 189
24. R. McCain. Reusable software component construction: A product-oriented paradigm. In *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, California, October 1985. 175
25. James Neighbors. *Software Construction Using Components*. PhD thesis, University of California, Irvine, 1980. 175
26. D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, mar 1976. 172
27. G. D. Plotkin. *A Structural Approach To Operational Semantics*. University of Aarhus, Aarhus, Denmark, 1981. 185
28. Rubn Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2), April 1990. 175
29. C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign. 172
30. T. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J. Baer, B. Bershad, and H. Levy. The structure and performance of interpreters. In *Proceedings of 7th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, October 1996. 188
31. D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986. 173, 177, 185
32. Guy L. Steele. Building interpreters by composing monads. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*. ACM Press, 1994. 188
33. S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131–135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3). 173
34. S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages*, pages 11–26, Santa Barbara, CA, October 1997. Usenix. 189
35. Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, October 1998. 172

36. Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, October 1998. 189
37. Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997. 172
38. P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 1–14, Albuquerque, New Mexico, USA, January 1992. ACM Press. 188
39. Bruce W. Weide and William F. Ogden. Recasting algorithms to encourge reuse. *IEEE Software*, 11(5), September 1994. 185
40. D.M. Weiss. Family-oriented abstraction specification and translation: the fast process. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland*, pages 14–22. IEEE Press, Piscataway, NJ, 1996. 175

Explicit Substitutions for Objects and Functions

Delia Kesner[1] and Pablo E. Martínez López[2]

[1] CNRS and Laboratoire de Recherche en Informatique
Bât 490, Université de Paris-Sud, 91405 Orsay Cedex, France
Delia.Kesner@lri.fr
[2] LIFIA, Departamento de Informática, Universidad Nacional de La Plata
C.C.11 Correo Central, 1900, La Plata, Prov. de Buenos Aires, Argentina
fidel@sol.info.unlp.edu.ar

Abstract. This paper proposes an implementation of objects and functions via a calculus with explicit substitutions which is confluent and preserves strong normalization. The source calculus corresponds to the combination of the ς-calculus of Abadi and Cardelli [AC96] and the λ-calculus, and the target calculus corresponds to an extension of the former calculus with explicit substitutions. The interesting feature of our calculus is that substitutions are separated – and treated accordingly – in two different kinds: those used to encode ordinary substitutions and those encoding invoke substitutions. When working with explicit substitutions, this differentiation is essential to encode λ-calculus into ς-calculus in a conservative way, following the style proposed in [AC96].

1 Introduction

Object-oriented languages are not yet well understood from a theoretical point of view since there is no widespread agreement on a collection of basic constructs and on their properties. A better understanding of the foundations of these languages is necessary, and in particular, formal calculi to model object-oriented programming become essential to this purpose, as well as various formal calculi have been used as foundations for procedural languages.

In [AC96] Abadi and Cardelli propose object calculi which are as simple and fruitful as λ-calculi and which clarify the general principles of object-oriented languages. They take objects as primitive and concentrate on the intrinsic rules that objects should obey, developing in this way a theory of objects. The theory suggests how to interpret existing constructions, and how to create and assess new ones. It also provides tools for reasoning about languages (existing or new), in particular for soundness proofs.

One of the calculi proposed in [AC96] is the ς-calculus, which is an untyped calculus supporting *method update*, a common property in object oriented languages. Update is the operation that modifies the behavior of an object by replacing one of its methods – the other methods being inherited. As a consequence, the semantics of objects given by the ς-calculus is natural and suggestive of common implementation techniques.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 195–212, 1998.

In λ-calculus as well as in ς-calculus, the evaluation process is modeled by a notion of reduction which performs the replacement of formal parameters by actual arguments. This operation, called *substitution*, is defined as an *atomic* operation described by operators that are external to the language, even if any real implementation of a language has to explicitly describe a method to compute and to apply them – method which is in general very expensive. Thus, there is a conceptual gap between the theory of the calculus and its implementation in programming languages. Calculi with explicit substitutions attempt to bridge this gap by incorporating substitutions explicitly into the language, so that substitutions are handled with symbols and reduction rules belonging to the syntax of the calculus. This allows more refined control over substitutions, more flexible implementations, and a theory tool to study abstract machines.

In these last years there was a growing interest in λ-calculi with explicit substitutions and various calculi have been proposed in the literature. Among them we find the pioneer, λ_σ, which is inspired by de Bruijn's notation [dB72,dB78] and has the main features which characterize a calculus of explicit substitutions: β-reduction is simulated in two stages, first by the application of one rule, which activates the calculus of substitutions, then by propagation of the substitution until variables are reached. Other calculi are $\lambda_{\sigma_\Uparrow}$ [HL89], λ_v [Les94], λ_s [KR95], λ_d [Kes96], λ_χ [Muñ96] and $\lambda\mathtt{x}$ [Ros92,BR95], this last one being the calculus inspiring the presentation of this work in named-variable style. However, explicit substitutions have been always studied in the framework of λ-calculi, the only exceptions being the formalisms of Combinatory Reduction Systems (CRSs) with explicit substitutions given in [BR96] and [Pag97], which are, respectively, generalizations of the calculi $\lambda\mathtt{x}$ and $\lambda_{\sigma_\Uparrow}$ to higher order rewriting. We should also mention the $\lambda\Delta\mathtt{exp}$-calculus studied in [BKR97], which is a particular case of CRS with explicit substitutions à la $\lambda\mathtt{x}$. All these works raise the question of the generality of explicit substitutions, that is, if a particular theory of explicit substitutions is sufficient to capture all the real implementations of programming languages based on higher order systems. As we will see along this paper, neither $\lambda\mathtt{x}$ nor $\lambda_{\sigma_\Uparrow}$ have all the features needed to implement objects and functions via explicit substitutions, so the calculus studied here is not a particular case of the formalism in [BR96] nor in [Pag97].

The aim of this paper is to provide an implementation language for object oriented programming with functions by merging two successful formalisms: explicit substitutions and ς-calculus. We provide an untyped ς-calculus with explicit substitutions, called ς_{ES}, which is confluent and preserves strong normalization. Our calculus can be used to implement an object-oriented language based on the ς-calculus, as well as a functional language modeled by the λ-calculus. This is a relevant feature that gives a theoretical model to implement a programming paradigm combining functional and object oriented styles. The interesting and original feature of the ς_{ES}-calculus is that substitutions are separated – and treated accordingly – in two different kinds: those used to encode ordinary substitutions and those encoding invoke substitutions. These two forms of explicit substitutions are induced from two subtly different uses of meta-substitution in

the ς-calculus: one having the form $\{x \leftarrow b\}$ where the variable x is not free in the term b, and the other having the form $\{x \leftarrow x.l\}$, where x is precisely free in the term $x.l$. The fact of separating explicit substitutions of ς_{ES} in two different kinds is useful not only to differentiate their properties but also to allow some kind of interaction or commutation between them which will not be allowed on two substitutions of the same kind. However, the resulting system is tuned to have the preservation of ς-strong normalization property since interaction of substitution does not have the same effect than the composition operator in λ_σ (which destroys preservation of β-strong normalization [Mel95]).

More precisely, ordinary substitution is used to perform evaluation of methods, while invoke substitution is used to replace a variable by a self method invocation, an operation which is used to implement functions. The differentiation between ordinary and invoke substitutions is essential to encode functions by objects in a conservative way using the same ideas in [AC96], but with explicit substitutions in our case: if only ordinary substitution is used to encode λ-terms into ς_{ES}-terms, β- reduction cannot be simulated anymore.

When working with explicit substitutions variables can also be encoded by natural numbers (using for example de Bruijn's indices) in order to avoid α-conversion, and this is done in most of the well-known calculi with explicit substitutions appearing in the literature. We think however that the named variable presentation makes some essential properties of explicit substitutions more apparent by abstracting out the details of renaming and updating. This is why we choose here to present our ideas via the ς-calculus with explicit substitutions and names. However, one of the reasons to introduce invoke substitutions comes from the de Bruijn's version of the ς-calculus with functions, which clearly shows the difference between the renaming needed by an ordinary or an invoke operation. We will come back to this point in section 3.

The paper is organized as follows. Section 2 recalls the syntax of the ς-calculus together with its reduction rules and properties. We also recall the translation of λ-calculus into ς-calculus and discuss some possible variations of this encoding. Section 3 is devoted to the ς-calculus with explicit substitutions and section 4 shows how to encode λ-calculi with explicit substitutions into our calculus ς_{ES}-calculus. In section 5 we show confluence and preservation of strong normalization of ς_{ES}. Finally, section 6 is devoted to conclusions and future works.

2 The ς-Calculus

We recall in this section the ς-calculus [AC96], which is the kernel language used to model implementations of object oriented languages. The calculus is built out of two distinct infinite sets of variables and labels, by means of *object formation*, *method invocation* and *method update*. An *object* is a set of components of the form $l_i = m_i$, for methods m_i and distinct labels l_i, where the order of the components does not matter. The letter ς is used as a binder for the self parameter of a method in the same way as the symbol λ is used in λ-calculus as a binder

for the parameter of a function. However, methods are inseparable from objects and cannot be recovered as functions. The set of terms of the ς-calculus, denoted by $\mathcal{T}_\varsigma$, is defined by the grammar:

$$\begin{array}{l} a \ ::= x \mid a.l \mid a \lhd \langle l, m \rangle \mid [l_i = m_i{}^{i \in 1 \ldots n}] \\ m ::= \varsigma(x).a \end{array}$$

We say that x is a *variable*, l is a *label*, $a.l$ is a *method invocation*, $a \lhd \langle l, m \rangle$ is a *method update* or *update*, $[l_i = m_i{}^{i \in 1 \ldots n}]$ is an *object*, $\varsigma(x).a$ is a *method*, x is a *bound* variable in $\varsigma(x).a$ and a is the *body* of the method $\varsigma(x).a$. Our notation for method update differs from that in [AC96] in order to avoid possible confusions with method invocation.

The sets of *free* and *bound variables* of a term a, denoted respectively by $FV(a)$ and $BV(a)$, are defined as in [AC96]. Terms are identified modulo α-conversion exactly as in the λ-calculus, by allowing the renaming of bound variables. Indeed, we assume the Barendregt's convention [Bar84] (but on $\mathcal{T}_\varsigma$-terms), which allows to work only with terms having no variables being bound and free at the same time. Substitution is at the core of the primitive semantics and is considered as a meta-operation as it is completely external to the language.

Definition 1 (Substitutions). *Let a and b be two terms, m be a method and $n \geq 1$. Then, the substitution of x by b in the term a, denoted $a\{x \leftarrow b\}$, is defined by induction as follows:*

$$\begin{array}{ll} (\varsigma(y).a)\{x \leftarrow b\} & =_{def} \varsigma(y).a\{x \leftarrow b\} \text{ if } x \not\equiv y \text{ and } y \notin FV(b) \\ [l_i = m_i{}^{i \in 1 \ldots n}]\{x \leftarrow b\} & =_{def} [l_i = m_i\{x \leftarrow b\}{}^{i \in 1 \ldots n}] \\ (a.l)\{x \leftarrow b\} & =_{def} a\{x \leftarrow b\}.l \\ (a \lhd \langle l, m \rangle)\{x \leftarrow b\} & =_{def} a\{x \leftarrow b\} \lhd \langle l, m\{x \leftarrow b\} \rangle \\ y\{x \leftarrow b\} & =_{def} \begin{cases} b & \text{if } x \equiv y \\ y & \text{if } x \not\equiv y \end{cases} \end{array}$$

We may make reference to the substitution lemma which says that $a\{x \leftarrow b\}\{y \leftarrow c\}$ is equal to $a\{y \leftarrow c\}\{x \leftarrow b\{y \leftarrow c\}\}$ if x is not free in c.

The primitive semantics of the ς-calculus can be expressed as a sequence of reduction steps, where reduction is defined by means of the following reduction rules.

$$\begin{array}{lcl} [l_i = \varsigma(x_i).b_i{}^{i \in 1 \ldots n}].l_j & \longrightarrow & b_j\{x_j \leftarrow [l_i = \varsigma(x_i).b_i{}^{i \in 1 \ldots n}]\} \\ [l_i = m_i^{i \in 1 \ldots n}] \lhd \langle l_j, m \rangle & \longrightarrow & [l_j = m,\ l_i = m_i^{i \in 1 \ldots n, i \neq j}] \end{array}$$

The intended semantics of a method invocation $a.l$ is given by the first rule: the idea is to execute the body of the method of a named l with the object a bound to the self parameter, and to return the result of the execution. The semantics of a method update $a \lhd \langle l, m \rangle$ is functional: the idea is to produce a copy of a where the method named l is replaced with the method m. The reduction relation ς is confluent [AC96], but strong normalization fails as the following example shows.

Example 1. Let $b = [l = \varsigma(x).x.l]$. Then, $b.l \longrightarrow x.l\{x \leftarrow b\} = b.l \longrightarrow \ldots$

Abadi and Cardelli show that the ς-calculus is computationally complete: every λ-term can be encoded into a ς-term and every β-step can be simulated by a sequence of ς-steps. This property is essential in order to incorporate functions to the ς-calculus: one simply adds λ-syntax to the ς-calculus – allowing in this way to reason not only about objects but also about functions –, but evaluation is just performed by a simple machinery, namely the ς-calculus. The set of terms of the λ-calculus mixed with the ς-calculus, denoted $\mathcal{T}_{\lambda\varsigma}$, is given by the following grammar:

$$\begin{array}{l} a \ ::= x \mid a.l \mid a \lhd \langle l, m\rangle \mid [l_i = {m_i}^{i\in 1\ldots n}] \mid \lambda x.a \mid (a\ a) \\ m ::= \varsigma(x).a \end{array}$$

Objects and functions are mixed in this syntax in such a way that $\lambda\varsigma$-terms can be reduced, according to their form, by the β or the ς-rules. Many well-motivated examples of objects using functions are given in [AC96]. The combination of β and ς yields a confluent reduction relation over $\mathcal{T}_{\lambda\varsigma}$ (this can be shown for example using the fact that the resulting system is an orthogonal CRS [KvOvR93]). Evaluation of this new combination can be simply implemented as a λ-term can be translated into a ς-term in such a way that β-reduction can be simulated by the rewriting rules of the ς-calculus.

Definition 2 (Translating $\lambda\varsigma$-terms into ς-terms). *The translation $\ll \_ \gg$ from $\lambda\varsigma$-terms to ς-terms is defined in the following way:*

$$\begin{array}{ll} \ll x \gg & =_{def} x \\ \ll a.l \gg & =_{def} \ll a \gg .l \\ \ll a \lhd \langle l, m\rangle \gg & =_{def} \ll a \gg \lhd\langle l, \ll m \gg\rangle \\ \ll \varsigma(x).a \gg & =_{def} \varsigma(x).\ \ll a \gg \\ \ll [l_i = {m_i}^{i\in 1\ldots n}] \gg & =_{def} [l_i = \ll m_i \gg^{\ i\in 1\ldots n}] \\ \ll \lambda x.a \gg & =_{def} [\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(x).\ \ll a \gg \{x \leftarrow x.\mathsf{arg}\}] \\ \ll (a\ b) \gg & =_{def} \ll a \gg \bullet \ll b \gg \\ & \qquad \textit{where } p \bullet q =_{def} (p \lhd \langle \mathsf{arg}, \varsigma(y).q\rangle).\mathsf{val} \textit{ with } y \notin FV(q) \end{array}$$

Summarizing, this function translates only λ-terms, while, for all the other constructions in the language, it is defined as a homomorphism. Indeed, the application first stores the value of the argument into the arg field, and then invokes a method that calculates the final value, accessing the argument via its self argument. Variables that are λ-bound are replaced by themselves accessing the arg field while free and ς-bound variables are left untouched. This translation preserves β-reduction.

Proposition 1 (ς simulates β). *Let A be a term in $\mathcal{T}_{\lambda\varsigma}$ such that $A \longrightarrow_\beta A'$. Then $\ll A \gg \longrightarrow^*_\varsigma \ll A' \gg$.*

Proof. The proof is by induction on the structure of $\lambda\varsigma$-terms. See [AC96] for details.

Fixpoint operators of the λ-calculus can also be encoded into $\lambda\varsigma$. Indeed, let consider a constant fix with its associated rewriting rule $(\mathsf{fix}\ A) \longrightarrow_{fix} (A\ (\mathsf{fix}\ A))$. Let consider the following extension of the translation $\ll\ \_\ \gg$ in order to cover λ-terms with fix-constants:

$$\ll \mathsf{fix} \gg =_{def} [\mathsf{arg} = \varsigma(x).x.\mathsf{arg},\ \mathsf{val} = \varsigma(x).((x.\mathsf{arg}) \lhd \langle \mathsf{arg}, \varsigma(y).x.\mathsf{val}\rangle).\mathsf{val}]$$

As expected, the following property holds ([AC96]).

Proposition 2 (ς simulates fix). *If $A \longrightarrow_{fix} A'$ then $\ll A \gg =_{\varsigma} \ll A' \gg$.*

One of the goals of the calculus proposed in section 3 is to keep these properties (propositions 1 and 2) in the presence of explicit substitutions. The notion of reduction in ς_{ES} is much more finer than that of standard ς-reduction, so that our calculus must be able to capture the subtle operations used to evaluate application of substitutions. We refer the reader to section 4 for the details concerning the translation of terms with explicit substitutions.

Didier Rémy proposed in [Rém97a] a slightly different encoding of λ-terms into ς-terms which may look simpler than the one of [AC96]: *all* λ-variables are replaced by themselves accessing the arg field (that is $\ll x \gg =_{def} x.\mathsf{arg}$ for λ-variables and $\ll x \gg =_{def} x$ for ς-variables), and thus, the encoding of $\lambda x.a$ does not need to replace the free variable x in a by $x.\mathsf{arg}$ (that is $\ll \lambda x.a \gg =_{def} [\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(x).\ \ll a \gg]$). As a consequence, when λ and ς syntax are mixed (as it is the case in [AC96] and here), the encoding of Rémy needs to perform an *a priori* differentiation of variables [Rém97b] in order to modify only λ-variables, leaving ς-variables untouched. Thus, a two-pass analysis will be necessary for Rémy's encoding (one to distinguish variables, the other one to encode them), while a single one-step translation via invoke substitutions is sufficient to implement the encoding proposed in [AC96]. Also, the translation we propose for object-oriented programs with functions *and* explicit substitutions (based on the translation proposed in [AC96]) is well behaved with respect to explicit substitutions operators, but this is no more true if Rémy's encoding is considered. We leave this (technical) discussion for section 4.

3 The ς_{ES}-Calculus

This section presents the ς-calculus with explicit substitutions using variable names as in [BR95,Ros92] instead of de Bruijn's indices as in [ACCL91]. This makes the calculus simpler and allows us to explain the main ideas of the interaction between ordinary and invoke substitution in a more intuitive way.

The set of terms of the ς_{ES}-calculus, denoted $\mathcal{T}_{\varsigma_{ES}}$ is defined by the following grammar:

$$\begin{array}{lll} a & ::= & x \mid a.l \mid a \lhd \langle l, m\rangle \mid [l_i = m_i{}^{i\in 1\ldots n}] \mid a[s] \\ m & ::= & \varsigma(x).a \mid m[s] \\ s & ::= & x \leftarrow a \mid x@l \end{array}$$

A term a is called a *closure*[1] if it has the form $b[s]$, and a *pure* term if it does not contain closure subterms. The sets of *free* and *bound variables* of a term a, denoted respectively by $FV(a)$ and $BV(a)$, are defined as usual by adding the cases $FV(a[x \leftarrow b]) = (FV(a) - \{x\}) \cup FV(b)$, $FV(a[x@l]) = FV(a)$, $BV(a[x \leftarrow b]) = BV(a) \cup BV(b) \cup \{x\}$ and $BV(a[x@l]) = BV(a)$. Terms are identified modulo α-conversion by allowing the renaming of bound variables. Indeed, we have for example $\varsigma(x).x =_\alpha \varsigma(y).y$, $x[x \leftarrow y][y@l] =_\alpha z[z \leftarrow y][y@l]$ and $y[y@l][y \leftarrow w] =_\alpha z[z@l][z \leftarrow w]$. As a consequence, the variable x is never bound in $a[x@l]$ and it is only free if it is free in a.

The meaning of $a[s]$ is the term a affected by the substitution s and we say that $[x \leftarrow b]$ is an *ordinary* substitution and $[x@l]$ is an *invoke* substitution. Thus, $a[x \leftarrow b]$ denotes the term $a\{x \leftarrow b\}$ (the substitution of the term b for all the free occurrences of x in a) and $a[x@l]$ denotes the term $a\{x \leftarrow x.l\}$ (the substitution of the term $x.l$ for all the free occurrences of x in a). Even if these two forms of substitutions may look very similar with respect to their meaning, they do not verify the same constraints with respect to their "types" and "free variables"; so we have to make this difference explicitly as we are concerned with *explicit* substitutions. Let us explain this difference in more detail.

Regarding constraints with respect to "types", in an ordinary substitution $[x \leftarrow b]$ (which denotes $\{x \leftarrow b\}$) the term b is intuitively of the same type than x, while in $[x@l]$ (which denotes $\{x \leftarrow x.l\}$) the invokation of a method l of x is in general, of different type than the object x.

Regarding constraints with respect to "free variables", ordinary substitutions in ς_{ES} must be identified with those substitutions in λ-calculus that are created by the contraction of a β-redex $((\lambda x.a)\ b)$, yielding the term $a\{x \leftarrow b\}$. This operation assumes implicitly the operation of α-conversion, so that the term b does not contain free occurrences of the variable x. As in λ-calculus with explicit substitutions the β-rule is modelized by the B-rule $((\lambda x.a)\ b) \longrightarrow_B a[x \leftarrow b]$, it is natural to impose that b does not contain free occurrences of the variable x in an ordinary substitution $[x \leftarrow b]$. However, since x has a free occurrence in $x.l$, an explicit substitution which is intended to modelize the substitution $\{x \leftarrow x.l\}$ cannot be considered as an ordinary one.

This difference is still more evident when using de Bruijn's indices since the standard renaming performed by an ordinary substitution is not the same needed by an invoke substitution. Indeed, ordinary substitution (exactly as the substitution used to perform β-reduction in λ-calculus) should decrement *all* the indices of the variables, while invoke substitution adds information to a bound variable, leaving the rest untouched. Let us see that on the following example

[1] This is the standard terminology used in the explicit substitution community to denote terms affected by explicit substitutions. Do not make confusion with the terminology of *closed terms* used to denoted terms without free variables.

by considering the terms[2]:

$$o = \varsigma(x).(\varsigma(y).x[y@l_1])[x@l_2] \qquad o_{dB} = \varsigma(\varsigma(\underline{2}[1@l_1])[1@l_2])$$

$$o' = \varsigma(x).(\varsigma(y).(x.l_2)) \qquad o'_{dB} = \varsigma(\varsigma(\underline{2}.l_2))$$

The term o_{dB} (resp. o'_{dB}) is the de Bruijn's representation of o (resp. o'). Now, since $[x@l]$ is an operation which is intended to replace x by $x.l$, the term o intuitively reduces to o'. However, if $[\underline{n}@l]$ is treated exactly as an ordinary explicit substitution $[cons(\underline{n}.l)]$ (see for example [Kes96] for a general treatment of calculi with de Bruijn's indices) the term o_{dB} reduces to the term $\varsigma(\varsigma(\underline{1}))$ which is not the de Bruijn's representation of o'.

This distinction between ordinary and invoke substitution is irrelevant when working with substitutions at a meta-level, but becomes essential when implementing them explicitly, either via names or de Bruijn's indices. We have then to separate the substitutions not only to differentiate the properties we have just explained but also in order to allow some kind of interaction or commutation between ordinary and invoke substitutions, which will not be allowed on two substitutions of the same kind. The semantics of the ς_{ES}-calculus is then given by the following set of rewriting rules.

$[l_i = \varsigma(x_i).b_i^{\,i\in 1\ldots n}].l_j$	$\longrightarrow_{\text{MI}}$	$b_j[x_j \leftarrow [l_i = \varsigma(x_i).b_i^{\,i\in 1\ldots n}]]$	
$[l_i = m_i^{\,i\in 1\ldots n}] \lhd \langle l_j, m\rangle$	$\longrightarrow_{\text{MU}}$	$[l_j = m,\ l_i = m_i^{\,i\in 1\ldots n,\ i\neq j}]$	
$(\varsigma(y).c)[s]$	$\longrightarrow_{\text{SM}}$	$\varsigma(y).c[s]$	if $x \not\equiv y$
$[l_i = m_i^{\,i\in 1\ldots n}][s]$	$\longrightarrow_{\text{SO}}$	$[l_i = m_i[s]^{\ i\in 1\ldots n}]$	
$(a.l)[s]$	$\longrightarrow_{\text{SI}}$	$a[s].l$	
$(a \lhd \langle l, m\rangle)[s]$	$\longrightarrow_{\text{SU}}$	$a[s] \lhd \langle l, m[s]\rangle$	
$x[x \leftarrow b]$	$\longrightarrow_{\text{OSV}}$	b	
$x[x@l]$	$\longrightarrow_{\text{ISV}}$	$x.l$	
$a[x \leftarrow b]$	$\longrightarrow_{\text{DO}}$	a	if $x \notin FV(a)$
$a[x@l]$	$\longrightarrow_{\text{DI}}$	a	if $x \notin FV(a)$
$a[x@l][x \leftarrow [l = \varsigma(y).b, \ldots]]$	$\longrightarrow_{\text{CO}}$	$a[x \leftarrow b]$	if $y \notin FV(b)$
$a[y@l][x \leftarrow b]$	$\longrightarrow_{\text{SW}}$	$a[x \leftarrow b][y@l]$	if $x \not\equiv y$ and $y \notin FV(b)$

The MI-rule activates a **M**ethod **I**nvocation, while the MU-rule implements **M**ethod **U**pdate and SM, SO, SI, SU propagate a **S**ubstitution through a **M**ethod, an **O**bject, an **I**nvocation and an **U**pdate respectively. Application of **O**rdinary (resp. **I**nvoke) **S**ubstitution to **V**ariables is performed by OSV (resp. ISV) and DO and DI are used to **D**iscard **O**rdinary and **I**nvoke substitutions which have not meaning. The CO-rule **CO**mposes ordinary and invoke substitutions and SW **SW**itches them when it is possible. Both rules are necessary to simulate the λx-calculus into ς_{ES} (see section 4). The rules CO and SW are necessary to simulate the λx-calculus via ς_{ES} (see section 4). They justify the existence of invoke

[2] Underlining is used to distinguish de Bruijn indexes from natural numbers.

substitutions and they capture the interaction between invoke and ordinary substitution.

Notice that a simpler rule like $a[x@l][x \leftarrow b] \longrightarrow_{\mathrm{CO}'} a[x \leftarrow b.l]$, which is also able to simulate $\lambda\mathtt{x}$, fails to preserve strong normalization of the rest of the system ς_{ES}: indeed, if b is the object $[l = \varsigma(x).x.l]$ and a is for example the variable y, then $a[x@l][x \leftarrow b]$ is strongly normalizing but $a[x \leftarrow b.l]$ is not as there is an infinite reduction sequence starting at $b.l$ (see example 1).

The set of rules $\{\text{SM}, \text{SO}, \text{SI}, \text{SU}, \text{OSV}, \text{DO}, \text{ISV}, \text{DI}, \text{CO}, \text{SW}\}$ is called the ES rewriting system, and ES together with the MI and MU rules constitutes the ς_{ES} rewriting system. Since ES is not locally confluent, we have to use another calculus to define a function (given by a locally confluent and terminating system) to eliminate explicit substitutions from ς_{ES}-terms in the confluence proof of ς_{ES}. This system is called BES and contains the rules $\{\text{SM}, \text{SO}, \text{SI}, \text{SU}, \text{OSV}, \text{ISV}, \text{DO}, \text{DI}\}$. We will also mention the *minimal* calculus $\text{MES} = \{\text{SM}, \text{SO}, \text{SI}, \text{SU}, \text{OSV}, \text{DO}\}$, and the calculus $\varsigma_{MES} = \text{MES} \cup \{\text{MI}\} \cup \{\text{MU}\}$ which implements object-oriented programs which do not use functions.

The ς-terms are naturally injected into the ς_{ES}-terms, and ς-reduction is simulated by the ς_{ES}-reduction (theorem 1).

4 Encoding λx-Terms into ς_{ES}-Terms

The goal of this section is to provide a satisfactory simulation of the λ-calculus with explicit substitutions, called $\lambda\mathtt{x}$, into our calculus ς_{ES}. We take the same approach presented in section 2 by first giving a syntax for a mixed calculus called $\lambda\varsigma_{ES}$, and a translation from $\lambda\varsigma_{ES}$-terms into ς_{ES}-terms, then showing that the translation preserves reduction. This is an essential property of the ς_{ES}-calculus, which guarantees that it can be used in a conservative way to implement a language with objects and functions.

We first recall the main definition concerning $\lambda\mathtt{x}$-calculus [Ros92,BR95]. The set of $\lambda\mathtt{x}$-terms is defined by $M ::= x \mid (M\ M) \mid \lambda x.M \mid M[x \leftarrow M]$ and the rewriting rules are the following:

$$
\begin{array}{lll}
(B) & (\lambda x.M)N & \longrightarrow M[x \leftarrow N] \\
(Var1) & x[x \leftarrow N] & \longrightarrow N \\
(Var2) & x[y \leftarrow N] & \longrightarrow x \text{ if } x \neq y \\
(Lambda) & (\lambda x.M)[y \leftarrow N] & \longrightarrow \lambda x.M[y \leftarrow N] \text{ if } x \neq y \text{ and } x \notin FV(N) \\
(App) & (M_1\ M_2)[x \leftarrow N] & \longrightarrow (M_1[x \leftarrow N]\ M_2[x \leftarrow N])
\end{array}
$$

The $\lambda\varsigma_{ES}$-calculus is similar to the $\lambda\varsigma$-calculus presented in section 2, but it also contains explicit substitutions. The rules of $\lambda\varsigma_{ES}$ are the ones for ς_{ES} plus those for $\lambda\mathtt{x}$-calculus and the set of terms of $\varsigma_{ES} + \lambda\mathtt{x}$, denoted $\mathcal{T}_{\lambda\varsigma_{ES}}$, is defined by the grammar:

$$
\begin{array}{l}
a ::= x \mid a.l \mid a \lhd \langle l, m \rangle \mid [l_i = m_i{}^{i \in 1 \ldots n}] \mid a[s] \mid \lambda x.a \mid (a\ a) \\
m ::= \varsigma(x).a \mid m[s] \\
s ::= x \leftarrow a \mid x@l
\end{array}
$$

The encoding of $\lambda\mathtt{x}$-terms into ς_{ES}-terms is similar to the one in section 2, but here the invoke substitutions is annotated (and not evaluated) using the new operator @ of explicit invoke substitution.

Definition 3 (Translating $\lambda\varsigma_{ES}$-terms into ς_{ES}-terms). *The translation* $\prec \_ \succ$ *from* $\lambda\varsigma_{ES}$*-terms to* ς_{ES} *is defined in the following way:*

$$
\begin{array}{ll}
\prec x \succ & =_{def} x \\
\prec a.l \succ & =_{def} \prec a \succ .l \\
\prec a \lhd \langle l, m\rangle \succ & =_{def} \prec a \succ \lhd \langle l, \prec m \succ\rangle \\
\prec [l_i = m_i{}^{i\in 1\ldots n}] \succ & =_{def} [l_i = \prec m_i \succ {}^{i\in 1\ldots n}] \\
\prec a[s] \succ & =_{def} \prec a \succ [\prec s \succ] \\
\prec \varsigma(x).a \succ & =_{def} \varsigma(x).\prec a \succ \\
\prec m[s] \succ & =_{def} \prec m \succ [\prec s \succ] \\
\prec x \leftarrow a \succ & =_{def} x \leftarrow \prec a \succ \\
\prec x@l \succ & =_{def} x@l \\
\prec \lambda x.a \succ & =_{def} [\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(x).\prec a \succ [x@\mathsf{arg}]] \\
\prec (a\ b) \succ & =_{def} \prec a \succ \bullet \prec b \succ \\
 & \qquad \textit{where } p \bullet q =_{def} (p \lhd \langle \mathsf{arg}, \varsigma(y).q\rangle).\mathsf{val} \textit{ and } y \notin FV(q)
\end{array}
$$

The translation changes all the lambda expressions into objects, keeping the structure of the rest of the constructions, namely, the objects, variables, methods and explicit substitutions. Even if it would be possible to translate λ-bound variables using a meta-level substitution (like in [AC96]), we prefer to use an explicit invoke substitution to be coherent with our treatment of substitutions and to avoid the use of a complicate formalism dealing with explicit and implicit substitution at the same time. Thus, this translation follows the lines of the rest of our work, and invoke substitutions becomes an operation having the same status as any ordinary substitution.

As an example, given the term $(\lambda x.x)w$, we have that

$$\prec (\lambda x.x)w \succ \equiv [\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(x).x[x@\mathsf{arg}]] \bullet \prec w \succ$$

Now, let o be the object $[\mathsf{arg} = \varsigma(y).\prec w \succ,\ \mathsf{val} = \varsigma(x).x[x@\mathsf{arg}]]$. Then, the expresion $\prec (\lambda x.x)w \succ$ ς_{ES}-reduces to $x[x@\mathsf{arg}][x \leftarrow o]$, which CO-reduces to $x[x \leftarrow \prec w \succ] \equiv \prec x[x \leftarrow w] \succ$. Remark that if one uses the ordinary substitution $[x \leftarrow x.\mathsf{arg}]$ in the translation of $\lambda x.x$ instead of the invoke substitution $[x@\mathsf{arg}]$, then the expression $\prec (\lambda x.x)w \succ$ would ς_{ES}-reduce to $x[x \leftarrow x.l][x \leftarrow o]$, which ς_{ES}-reduces to $\prec w \succ \not\equiv \prec x[x \leftarrow w] \succ$. Thus, simulation of $\lambda\mathtt{x}$ would not be possible via ς_{ES}.

The real interaction between invoke and ordinary substitutions can be seen in the proof of the following lemma where the SW and CO rules are fundamental to simulate $\lambda\mathtt{x}$ via ς_{ES}.

Proposition 3 (ς_{ES} simulates $\lambda\mathtt{x}$). *Let* A *be a term in* $\lambda\varsigma_{ES}$ *such that* $A \longrightarrow_{\lambda\mathtt{x}} A'$. *Then* $\prec A \succ \longrightarrow^*_{\varsigma_{ES}} \prec A' \succ$.

Proof. The proof is done by induction on the structure of the λx-term, using the following sentences:

1. $\prec x[x \leftarrow b] \succ \longrightarrow^*_{\varsigma_{ES}} \prec b \succ$
2. $\prec y[x \leftarrow b] \succ \longrightarrow^*_{\varsigma_{ES}} \prec y \succ$, for $x \not\equiv y$
3. $\prec (c\ d)[x \leftarrow b] \succ \longrightarrow^*_{\varsigma_{ES}} \prec (c[x \leftarrow b]\ d[x \leftarrow b]) \succ$
4. $\prec (\lambda y.c)[x \leftarrow b] \succ \longrightarrow^*_{\varsigma_{ES}} \prec \lambda y.c[x \leftarrow b] \succ$
5. $\prec ((\lambda x.c)\ b) \succ \longrightarrow^*_{\varsigma_{ES}} \prec c[x \leftarrow b] \succ$

Cases 1, 2, and 3 are straightforward. For case 4, we have

$$
\begin{array}{ll}
\prec (\lambda y.c)[x \leftarrow b] \succ & =_{def} \\
[\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(y).\ \prec c \succ [y@\mathsf{arg}]][x \leftarrow \prec b \succ] & \longrightarrow^*_{\mathrm{SO,SM,SI,DO}} \\
[\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(y).\ \prec c \succ [y@\mathsf{arg}][x \leftarrow \prec b \succ]] & \longrightarrow_{\mathrm{SW}} \\
[\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(y).\ \prec c \succ [x \leftarrow \prec b \succ][y@\mathsf{arg}]] & =_{def} \\
\prec \lambda y.c[x \leftarrow b] \succ &
\end{array}
$$

For case 5, we have

$$
\begin{array}{ll}
\prec ((\lambda x.c)\ b) \succ & =_{def} \\
[\mathsf{arg} = \varsigma(z).z.\mathsf{arg},\ \mathsf{val} = \varsigma(x).\ \prec c \succ [x@\mathsf{arg}]] \bullet \prec b \succ & \longrightarrow^* \\
\prec c \succ [x@\mathsf{arg}][x \leftarrow [\mathsf{arg} = \varsigma(y)\ \prec b \succ,\ \mathsf{val} = \ldots]] & \longrightarrow_{\mathrm{CO}} \\
\prec c \succ [x \leftarrow \prec b \succ] & =_{def} \\
\prec c[x \leftarrow b] \succ &
\end{array}
$$

It is worth noticing that Rémy's encoding does not satisfy the properties of proposition 3: the term $x[x \leftarrow b]$ λx-reduces to b but $\prec x[x \leftarrow b] \succ = \prec x \succ [x \leftarrow \prec b \succ] = x.\mathsf{arg}[x \leftarrow \prec b \succ]$ does not ς_{ES}-reduce to $\prec b \succ$. This reason justifies our choice of the encoding proposed in [AC96] instead of that in [Rém97a].

Fixpoint operators can also be encoded into ς_{ES} by extending $\prec \_ \succ$ with $\prec \mathsf{fix} \succ = \ll \mathsf{fix} \gg$, where $\ll \_ \gg$ is the translation in definition 2. Since $\ll \mathsf{fix} \gg$ does not involve neither explicit substitutions nor even λ- terms, then the following property becomes straightforward using the theorem 1 in section 5:

Proposition 4 (ς_{ES} simulates fix). *If* $A \longrightarrow_{fix} A'$ *then* $\prec A \succ =_{\varsigma_{ES}} \prec A' \succ$.

As a consequence of the previous results, $\lambda\varsigma_{ES}$-reduction sequences can be translated in ς_{ES}-reduction sequences, even in the presence of recursion, and thus ς_{ES}-calculus can be used to implement both objects and functions.

5 Properties of ς_{ES}

In this section we study the main properties of the ς_{ES}-calculus, namely, confluence and preservation of ς-strong normalization. Confluence means that whenever $a \longrightarrow^*_{\varsigma_{ES}} b$ and $a \longrightarrow^*_{\varsigma_{ES}} c$ then there exists a term d such that $b \longrightarrow^*_{\varsigma_{ES}} d$ and $c \longrightarrow^*_{\varsigma_{ES}} d$. This property is essential to ensure uniqueness of the results of computations. In order to guarantee correctness of any implementation, preservation of strong normalization is also essential when implementing a (non-

terminating) calculus with another one. Indeed, in contrast to what was expected, the calculus with explicit substitutions λ_σ does not preserve β-strong normalization of λ-calculus [Mel95]. This result is very surprising because it is natural to expect the meta-theory of the calculus with implicit and explicit substitutions to coincide. But λ_σ has a powerful form of composition of substitutions which destroys strong normalization even for simply typed terms. On the other hand all the λ-calculi with explicit substitutions in the literature which preserve β strong normalization do not allow neither an interaction nor a form of *full* [3] composition between substitutions. As a consequence, since the calculus ς_{ES} proposed in this paper allows some interaction between some forms of substitutions, then the proof of preservation of ς-strong normalization given in this section turns out to be essential. It is also worth noting that the ς_{ES}-calculus *is not* a particular case of the Combinatory Reduction Systems with explicit substitutions proposed in [BR96] and [Pag97]. Indeed, both works specify calculi having only *ordinary* substitutions.

5.1 Confluence of ς_{ES}

In this section we prove that ς_{ES} is confluent. For that purpose, we use the interpretation method [Har87] (theorem 3) – which is the standard technique used to prove confluence of calculi with explicit substitutions. We use the calculus BES to interpret ς_{ES}-terms into ς-terms, that is, to eliminate closures from ς_{ES}-terms.

Lemma 1 (SN of sw). *The* sw*-rule is strongly normalizing.*

Proof. The proof uses theorem 6 in appendix A which is based on the weak normalization property of sw. The sw-rule can be shown to be weakly normalizing using the same technique in [KR97] to show weak normalization of the σ-/-transition rule of the $\lambda\omega_e$-calculus. Now, we proceed as follows. Let us define $f : \mathcal{T}_{\varsigma_{ES}} \mapsto \mathbb{N}$ as

$$\begin{array}{ll} f(x) & = 1 \\ f(\varsigma(x).a) & = f(a) \\ f(a.l) & = f(a) \\ f(a[x@l]) & = 2.f(a) \end{array} \qquad \begin{array}{ll} f(a \lhd \langle l, m\rangle) & = f(a) + f(m) \\ f([l_i = m_i{}^{i\in 1\ldots n}]) & = \Sigma_{i=1}^{n} f(m_i) \\ f(a[x \leftarrow b]) & = f(a) + f(b) \end{array}$$

We can easily check that $f(a[x@l][y \leftarrow b]) = 2.f(a)+f(b) < 2.(f(a)+f(b)) = f(a[y \leftarrow b][x@l])$, so that one can prove by induction on the structure of a that whenever $a \longrightarrow_{\text{SW}} b$, then $f(a) < f(b)$. Now, since sw is (trivially) locally confluent, strong normalization of sw follows from weak normalization of sw by application of theorem 6 in appendix A.

[3] The only exception is the λ_d-calculus [Kes96] which provides a weak form of composition for substitutions. However, the λ_d is completely different in spirit to the ς_{ES}-calculus as it only allows interactions between substitutions that perform the renaming of de Bruijn's indices and ordinary substitutions.

Lemma 2 (SN of ES∪{MU}). *The ES∪{MU}-calculus is strongly normalizing.*

Proof. Take the following measure $h : \mathcal{T}_{\varsigma_{ES}} \longrightarrow \mathbb{N}_2$, where $\mathbb{N}_2$ is the set of integers greater or equal to 2.

$$\begin{array}{ll} h(x) & = 2 \\ h(\varsigma(x).a) & = h(a) + 1 \\ h(a.l) & = h(a) + 1 \end{array} \qquad \begin{array}{ll} h(a \lhd \langle l, m \rangle) & = h(a) + h(m) + 1 \\ h([l_i = m_i{}^{i \in 1 \ldots n}]) & = \Sigma_{i=1}^{n} h(m_i) + n + 1 \\ h(a[x \leftarrow b]) & = h(a).(h(x) + h(b)) \\ h(a[x@l]) & = h(a).(h(x) + 1) \end{array}$$

and show that $s \longrightarrow_{\mathrm{ES-SW}} t$ implies $h(s) > h(t)$ and $s \longrightarrow_{\mathrm{SW}} t$ implies $h(s) = h(t)$. Then termination of ES follows from lemma 1 by application of theorem 5 in the appendix A.

Notice that the system ES is not locally confluent, so that we use BES to interpret ς_{ES}-terms into ς-terms, that is, to eliminate explicit substitutions from ς_{ES}-terms. Since the BES-calculus is locally confluent (this can be proved by cases where the only critical pairs arise from DO or DI and SM, SO, SI, SU) and strongly normalizing (because it is a subcalculus of ES which is strongly normalizing) we can then obtain the following:

Corollary 1 (Confluence of BES). *The BES calculus is confluent and BES-normal forms are unique. From now on, we denote by BES(a) the BES-normal form of a.*

Now, we show some technical properties that we need to prove confluence of ς_{ES}.

Proposition 5. *For every term p in $\mathcal{T}_{\varsigma_{ES}}$ and every pure term d in $\mathcal{T}_{\varsigma}$*

1. *BES(p) is a pure term,*
2. *If $d[x \leftarrow p]$ is a term in $\mathcal{T}_{\varsigma_{ES}}$, then BES($d[x \leftarrow p]$) is a pure term in $\mathcal{T}_{\varsigma}$.*

Proof. It can be done by induction on the lexicographic order induced by the pair $\langle p, d \rangle$.

As a consequence, we get:

Corollary 2. *BES-normal forms of $\mathcal{T}_{\varsigma_{ES}}$-terms are pure terms.*

Lemma 3 (Behavior of subst.). *Let $a, b \in \mathcal{T}_{\varsigma_{ES}}$. Then BES($a[x \leftarrow b]$) = BES(a){$x \leftarrow$ BES(b)} and BES($a[x@l]$) = BES(a){$x \leftarrow x.l$}.*

Proof. The property can be shown by first proving the statement whenever a is a pure term (which can be done by induction on the structure of terms, using definition 1). For non-pure terms a, one has that:

- BES($a[x \leftarrow b]$) = BES(BES(a)$[x \leftarrow b]$) by uniqueness of BES-normal forms and BES(a) is a pure term by corollary 2. Thus, the previous case for pure terms applies and gives BES(BES(a)){$x \leftarrow$ BES(b)} = BES(a){$x \leftarrow$ BES(b)} again by uniqueness of normal forms.

– $\mathrm{BES}(a[x@l]) = \mathrm{BES}(\mathrm{BES}(a)[x@l])$ by uniqueness of BES-normal forms and $\mathrm{BES}(a)$ is a pure term by corollary 2. Thus, the previous case for pure terms applies and gives $\mathrm{BES}(\mathrm{BES}(a))\{x \leftarrow x.l\} = \mathrm{BES}(a)\{x \leftarrow x.l\}$ again by uniqueness of normal forms. ∎

Theorem 1 (Simulation of ς by ς_{ES}). *If $a \longrightarrow^*_{\varsigma} b$, then $a \longrightarrow^*_{\varsigma_{ES}} b$.*

Proof. The proof proceeds by induction on the length of the reduction sequence $a \longrightarrow^*_{\varsigma} b$, using, for the base case, induction on the term a and lemma 3.

Theorem 2 (Simulation of ς_{ES} by ς). *If $a \longrightarrow^* \varsigma_{ES} b$, then $BES(a) \longrightarrow^* \varsigma BES(b)$.*

Proof. The proof proceeds by induction on the length of the reduction sequence $a \longrightarrow^*_{\varsigma_{ES}} b$, using, for the base case, induction on the term a. Indeed, we have that $s \longrightarrow_{\mathrm{MI,MU,CO}} t$ implies $\mathrm{BES}(s) \longrightarrow^*_{\varsigma} \mathrm{BES}(t)$ and $s \longrightarrow_{\mathrm{SM,SO,SI,SU,OSV,ISV,SW}} t$ implies $\mathrm{BES}(s) = \mathrm{BES}(t)$.

Theorem 3 (Confluence of ς_{ES}). *The ς_{ES}-calculus is confluent.*

Proof. The proof uses the interpretation method [Har87]. Let $a, b, c \in \mathcal{T}_{\varsigma_{ES}}$ such that $a \longrightarrow^*_{\varsigma_{ES}} b$ and $a \longrightarrow^*_{\varsigma_{ES}} c$. By theorem 2 we have that $\mathrm{BES}(a) \longrightarrow^*_{\varsigma} \mathrm{BES}(b)$ and that $\mathrm{BES}(a) \longrightarrow^*_{\varsigma} \mathrm{BES}(c)$. By confluence of ς we know that there is a pure term d such that $\mathrm{BES}(b) \longrightarrow^*_{\varsigma} d$ and $\mathrm{BES}(c) \longrightarrow^*_{\varsigma} d$ and by theorem 1 we have that $\mathrm{BES}(b) \longrightarrow^*_{\varsigma_{ES}} d$ and that $\mathrm{BES}(c) \longrightarrow^*_{\varsigma_{ES}} d$ so the diagram can be closed by $b \longrightarrow^*_{\mathrm{BES}} \mathrm{BES}(b) \longrightarrow^*_{\varsigma_{ES}} d$ and $c \longrightarrow^*_{\mathrm{BES}} \mathrm{BES}(c) \longrightarrow^*_{\varsigma_{ES}} d$.

This same technique can be used to show that the ς_{MES}-calculus is confluent on the set of terms $\mathcal{T}_{\varsigma_{ES}}$ which do not contain $[\_@\_]$ substitutions. For that, one has just to remark that MES is also locally confluent (and of course terminating since is a subsystem of BES), so that MES can be used to interpret the terms as in theorem 3.

5.2 Preservation of ς-Strong Normalization

In this section we show that every ς-term which is ς-strongly normalizing is also ς_{ES}-strongly normalizing. This property is essential to keep the normalization properties of ς when implementing it by ς_{ES}.

We first define SN_{ς} as the set of all the ς-strongly normalizing pure terms of $\mathcal{T}_{\varsigma_{ES}}$ and $\mathcal{F}$ as $\{a \in \mathcal{T}_{\varsigma_{ES}} \mid \text{ for all subterm } b \text{ of } a, \mathrm{BES}(b) \in SN_{\varsigma}\}$.

Lemma 4. *If $a \in \mathcal{F}$ and $a \longrightarrow_{\varsigma_{ES}} a'$, then $a' \in \mathcal{F}$.*

Proof. By induction on the structure of a using theorem 2. The more delicate case is when $a = e[y@l][x \leftarrow b] \longrightarrow_{\mathrm{SW}} e[x \leftarrow b][y@l] = a'$. Let us suppose that a is in $\mathcal{F}$ but $a' \notin \mathcal{F}$. As e and b are subterms of a, then by hypothesis $\mathrm{BES}(e)$ and $\mathrm{BES}(b)$ are ς-strongly normalizable; and also $\mathrm{BES}(a') = \mathrm{BES}(a)$, so that $\mathrm{BES}(a')$ is in SN_{ς} by hypothesis. As a consequence, the only possible case is $\mathrm{BES}(e\{x \leftarrow b\})$ not in SN_{ς}. Now, $\mathrm{BES}(e[x \leftarrow b])$ is exactly

$\mathrm{BES}(e)\{x \leftarrow \mathrm{BES}(b)\}$ and $\mathrm{BES}(e[y@l][x \leftarrow b])$ is exactly $\mathrm{BES}(e)\{x \leftarrow \mathrm{BES}(b)\}\{y \leftarrow y.l\}$ by lemma 3 and the substitution lemma, so that if $\mathrm{BES}(e)\{x \leftarrow \mathrm{BES}(b)\}$ is not in SN_ς, neither is $\mathrm{BES}(e)\{x \leftarrow \mathrm{BES}(b)\}\{y \leftarrow y.l\} = \mathrm{BES}(a)$ which leads to a contradiction with the hypothesis. We can then conclude that $a' \in \mathcal{F}$.

We are now going to interpret ς_{ES}-terms in $\mathcal{F}$ into another set of terms denoted by Θ which is generated by the following grammar:

$$t ::= \star \mid t._n l \mid \lhd(t,l,u) \mid [l_i = u_i{}^{i\in 1\ldots n}] \mid @(t,l) \mid t\langle t\rangle_n$$
$$u ::= \varsigma(t) \mid u\langle t\rangle_n \mid @(u,l)$$

For every pure term a, we denote by $\nu(a)$ the maximal length of a ς-reduction sequence starting at a. The translation $\mathcal{T} : \mathcal{F} \longrightarrow \Theta$ is then defined as follows:

$$\begin{array}{lll}
\mathcal{T}(x) & = \star & \\
\mathcal{T}(\varsigma(x).a) & = \varsigma(\mathcal{T}(a)) & \\
\mathcal{T}(a.l) & = \mathcal{T}(a)._m l, & \text{for } m = \nu(\mathrm{BES}(a.l)) \\
\mathcal{T}(a \lhd \langle l, m\rangle) & = \lhd(\mathcal{T}(a), l, \mathcal{T}(m)) & \\
\mathcal{T}(a[x \leftarrow b]) & = \mathcal{T}(a)\langle\mathcal{T}(b)\rangle_m & \text{for } m = \nu(\mathrm{BES}(a[x \leftarrow b])) \\
\mathcal{T}(a[x@l]) & = @(\mathcal{T}(a), l) & \\
\mathcal{T}([l_i = m_i{}^{i\in 1\ldots n}]) & = [l_i = \mathcal{T}(m_i)\ {}^{i\in 1\ldots n}] &
\end{array}$$

We define $\mathcal{A}$ as the set of symbols $\{\star, @, \lhd, \varsigma, =, [\ ]\} \cup \{\langle\rangle_n \mid n \in \mathbb{N}\} \cup \{._n \mid n \in \mathbb{N}\}$, and we define a well-defined precedence $\gg$ on $\mathcal{A}$ given by $._{m+1} \gg \langle\rangle_m \gg ._m \gg \lhd \gg \varsigma, =, \star, [\ ]$ and $\langle\rangle_m \gg @ \gg ._h, \varsigma, [\ ], \lhd$. We denote by $>_\Theta$ the rpo order on Θ wrt the precedence $\gg$ and the status function defined as $\tau(\alpha) = mul$, for every $\alpha \in \mathcal{A}$ (for details see definition 4 in appendix A). This order $>_\Theta$ is well-founded [Der82,KL80].

Lemma 5. *Let $a \in \mathcal{F}$. Then $a \longrightarrow_{\varsigma_{ES}} a'$ implies $\mathcal{T}(a) >_\Theta \mathcal{T}(a')$.*

Proof. The proof is by induction on the structure of a. For more details, we refer the interested reader to the full version available by ftp.

Theorem 4. *The ς_{ES}-calculus preserves ς-strong normalization.*

Proof. Suppose that ς_{ES} does not preserve ς-strong normalization. Then there is a pure term $a \in \mathcal{T}_\varsigma$ such that $a \in SN_\varsigma$ and a is not ς_{ES}-strongly normalizing. By lemma 2 $\mathrm{ES} \cup \{\mathrm{MU}\}$ is terminating so that the infinite ς_{ES} reduction starting at a must be of the form

$$a \longrightarrow^*_{\mathrm{ES}\cup\{\mathrm{MU}\}} b'_0 \longrightarrow_{\mathrm{MI}} b_0 \longrightarrow^*_{\mathrm{ES}\cup\{\mathrm{MU}\}} b'_1 \longrightarrow_{\mathrm{MI}} b_1 \ldots \longrightarrow^*_{\mathrm{ES}\cup\{\mathrm{MU}\}} b'_i \longrightarrow_{\mathrm{MI}} b_i \ldots$$

Since a is a pure term, then a is in $\mathcal{F}$ and also by lemma 4 all the b_i and $b'_i, (i \geq 0)$ are in $\mathcal{F}$. As a consequence we can define the translation $\mathcal{T}()$ on all the terms a, b_i, b'_i and we obtain by lemma 5 an infinite sequence

$$\mathcal{T}(a) \geq_\Theta \mathcal{T}(b'_0) >_\Theta \mathcal{T}(b_0) \geq_\Theta \mathcal{T}(b'_1) >_\Theta \mathcal{T}(b_1) \geq_\Theta \mathcal{T}(b'_2) >_\Theta \mathcal{T}(b_2) \dots$$

and hence an infinite sequence

$$\mathcal{T}(a) >_\Theta \mathcal{T}(b_0) >_\Theta \mathcal{T}(b_1) >_\Theta \mathcal{T}(b_2) \dots$$

which yields to a contradiction since $>_\Theta$ is a well-founded order.

As a corollary we obtain that the ς_{MES}-calculus also preserves ς-strong normalization, and can then be used to implement object-oriented programs without functions.

6 Conclusion and Future Work

We have proposed an untyped calculus with explicit substitutions that allows to implement both objects and functions. This language is able to simulate any computation of the λ-calculus (with or without explicit substitutions) as well as any computation of the ς-calculus (with or without explicit substitutions). The interesting feature of our calculus with respect to other calculi with explicit substitutions proposed in the literature, is that substitutions are divided in two different kinds – and treated accordingly – in order to differentiate ordinary from invoke substitution. This treatment is essential to encode the λ-calculus into the ς-calculus using the same ideas in [AC96], but with explicit substitutions. Also, since the interaction between substitutions keeps preservation of ς-strong normalization, the translation from functions to objects is conservative and thus the calculus ς_{ES} proposed in this paper turns out to be a potential real language to implement objects and functions. It is worth noting that ς_{ES} is not a particular case of the combinatory reduction systems with explicit substitutions proposed in [BR96] and [Pag97] as they do not allow different sorts of substitutions having some kind of interaction between them. We have also shown that the minimal calculus ς_{MES} can be used to implement objects without functions: it is confluent and it preserves ς-strong normalization.

This work also suggests that a more general approach than those proposed in [BR96] and [Pag97] is needed to capture all the real implementations of programming languages based on higher order systems. In particular, we would like to pursue this line of research by extending CRSs with a general scheme like in [Kes96].

The specification of objects and functions implemented via explicit substitutions and de Bruijn's indices is the subject of current investigation.

Acknowledgments

This work was initiated when Delia Kesner visited the LIFIA at the Universidad de La Plata in August 1997 and it was continued when Pablo E. Martínez López visited LRI at the Université de Paris-Sud in December 1997. Both visits were supported by CNRS (France) and CONICET (Argentina).

References

AC96. Martín Abadi and Luca Cardelli. *A Theroy of Objects*. Springer Verlag, 1996. 195, 197, 198, 199, 200, 204, 205, 210

ACCL91. Martin Abadi, Luca Cardelli, Pierre Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991. 200

Bar84. Henk Barendregt. The Lambda Calculus; Its syntax and Semantics (revised edition). North Holland. 1984. 198

BKR97. Gilles Barthe, Fairouz Kamareddine, and Alejandro Ríos. Explicit substitutions for control operators. TR 96-26, University of Glasgow, 1997. 196

BR95. Roel Bloo and Kristoffer Rose. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands (CSN)*, pages 62–72, 1995. 196, 200, 203

BR96. Roel Bloo and Kristoffer Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In RTA'96, LNCS 1103, 1996. 196, 206, 210

dB72. N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Mat.*, 5(35):381–392, 1972. 196

dB78. N. de Bruijn. Lambda-calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Mat.*, 40:384–356, 1978. 196

Der82. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. 209

Har87. Thérèse Hardin. *Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs*. Thèse de doctorat, Université de Paris VII, 1987. 206, 208

HL89. Thérèse Hardin and Jean-Jacques Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989. 196

Kes96. Delia Kesner. Confluence properties of extensional and non-extensional λ-calculi with explicit substitutions. In RTA'96, LNCS 1103, 1996. 196, 202, 206, 210

KL80. S. Kamin and J. J. Lévy. Attempts for generalizing the recursive path orderings. University of Illinois, 1980. 209

KR95. Fairouz Kamareddine and Alejandro Ríos. A λ-calculus à la de bruijn with explicit substitutions. In PLILP'95, LNCS 982, 1995. 196

KR97. Fairouz Kamareddine and Alejandro Ríos. Bridging the $\lambda\sigma$- and λ_s-styles of explicit substitutions. TR 97-10, University of Glasgow, 1997. 206

KvOvR93. Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993. 199

Les94. Pierre Lescanne. From λ_σ to λ_υ, a journey through calculi of explicit substitutions. In POPL'94, ACM, 1994. 196

Mel95. Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In TLCA'95, LNCS 902, 1995. 197, 206

Muñ96. César Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus. In LICS'96, IEEE, 1996. 196
Pag97. Bruno Pagano. *X.R.S: Explicit reduction systems a first-order calculus for higher order calculi.* Thèse de doctorat, Université de Paris VII, 1997. 196, 206, 210
Rém97a. Didier Rémy. From classes to objects via subtyping, 1997. Draft. 200, 205
Rém97b. Didier Rémy. Personal communication, 1997. 200
Ros92. Kristoffer Rose. Explicit cyclic substitutions. In CTRS'92, LNCS 656, 1992. 196, 200, 203

A Some Known Properties

Theorem 5. *Let $R = \langle \mathcal{O}, R_1 \cup R_2 \rangle$ be an abstract reduction and $>$ be a well-founded order on S such that*

- *R_2 is strongly normalizing;*
- *there exists a function $h : \mathcal{O} \longrightarrow S$ such that*
  - *$a \longrightarrow_{R_1} b$ implies $h(a) > h(b)$,*
  - *$a \longrightarrow_{R_2} b$ implies $h(a) = h(b)$.*

Then $R_1 \cup R_2$ is strongly normalizing.

Theorem 6. *Let $R = \langle \mathcal{O}, \mathcal{R} \rangle$ be an abstract reduction such that*

- *$\mathcal{R}$ is weakly normalizing*
- *$\mathcal{R}$ is locally confluent*
- *there exists a function $f : \mathcal{O} \mapsto \mathbb{N}$ such that $a \mathcal{R} b$ implies $f(a) < f(b)$.*

Then $\mathcal{R}$ is strongly normalizing.

Definition 4 (RPO order). *Let $\gg$ be a precedence on a set of function symbols $\mathcal{A}$ and τ a status function such that $\tau(\alpha) = lex$ or $\tau(\alpha) = mul$ for every $\alpha \in \mathcal{A}$. Given two terms s and t on $\mathcal{A} \cup \mathcal{X}$ where $\mathcal{X}$ is a set of variables, we say that $s >_{rpo} t$ if and only if $s = f(s_1, \ldots, s_n)$ and either*

1. *$t = g(t_1, \ldots, t_m)$, $s >_{rpo} t_i$ for all $1 \leq i \leq m$ and*
   - *$f \gg g$, or*
   - *$f = g$ and $(s_1, \ldots, s_n) >_{rpo,\tau} (t_1, \ldots, t_m)$*
2. *there exists $1 \leq i \leq m$ such that $s_i >_{rpo} t$ or $s_i = t$.*

where τ is the extension of $>_{rpo}$ associated to the status $\tau(f)$.

The Complexity of Late-Binding in Dynamic Object-Oriented Languages*

Enrico Pontelli, Desh Ranjan, and Gopal Gupta

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science, New Mexico State University
Las Cruces, NM 88003 USA
{epontell,dranjan,gupta}@cs.nmsu.edu

Abstract. We study the algorithmic complexity of supporting *late binding* in *dynamic* object-oriented programming (*OOP*) languages—i.e., languages that allow creation of new class definitions at runtime. We propose an abstraction of the late-binding problem for dynamic OOP languages in terms of operations on dynamic trees, and develop a lower bound of $\Omega(\lg n)$ per operation (where n is the number of nodes in the tree). This result shows that efficient solutions (i.e., solutions that incur a constant time overhead per operation) of this problem are, in general, not possible. We also propose new data-structures and algorithms for solving the late-binding problem for dynamic OOP languages very efficiently, with a worst-case time complexity of $O(\sqrt[3]{n})$ per operation. This result is an improvement over most existing approaches.

1 Introduction

We study the the algorithmic complexity of implementation mechanisms for supporting inheritance in *dynamic* object-oriented languages. The aim of this study is to model these implementation mechanisms in terms of operations on dynamic data structures, with the final goal of determining their complexity. Our complexity study is used to derive novel data structures for implementing inheritance mechanisms efficiently. In this work we limit ourselves to studying the complexity of implementing late-binding inheritance in dynamic OOP systems, with particular focus on prototype-based languages. The inheritance considered is single inheritance, or multiple inheritance with linearization. Generalization of this work to more complex domains (tree and graph-based multiple inheritance) is still under investigation.

Inheritance refers to the property of OOP languages whereby new classes are defined by specializing existing classes. When a class D is defined as a subclass of another class B (sub-typing), it inherits all the attributes (both data and operations) from class B. *Late binding* means that procedure-call name-resolution, i.e., mapping of a procedure call to a procedure definition, is done at run-time,

* This work has been partially supported by NSF grants HRD 93-53271, INT 95-15256, and CCR 96-25358, and by NATO Grant CRG 921318.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 213–229, 1998.

rather than compile-time. This is because due to sub-typing and inheritance, it is not immediately clear which definition should be used corresponding to a procedure call, as procedure definitions may be multiply defined in a class hierarchy. Late-binding is an essential feature of OOP and is present in most languages, e.g., Java, CLOS, Smalltalk, C++ (virtual functions), etc.

Dynamic OOP languages are languages that allow dynamic runtime creation of class definitions. Thus the complete class hierarchies are not known at compile-time. CLOS [17] and Smalltalk are two such languages. Additionally, prototype based languages [2,4,29] are also examples of such languages. Name-resolution in presence of late-binding is easily solved for static OOP languages (such as Java and C++) since the complete class hierarchy is known at compile-time; for dynamic languages, due to absence of this information, it is not as easy. To study the complexity of the problems that arise in implementing name-resolution in late-binding dynamic OOP languages, we first abstract it and formalize it in terms of operations on *dynamic trees* (i.e., trees that grow and shrink with time). This formalization permits us to abstractly study the implementation problems.

The abstraction of the computing machine that we choose is the *Pointer Machine* [18,19,27]. The reason we choose a pointer machine is because it provides operations needed for expressing computations over dynamic data structures; it also allows us to perform a finer level of analysis compared to other models (e.g., RAM). Besides, use of pointer machines enables us to use many existing results [22,12,14,21] which are useful in our study and have been developed on pointer machines. Analysis of the name-resolution problem in late-binding dynamic language on pointer machines gives us a deep insight into the sources of its implementation complexity.

Based on our rigorous study, we develop a lower bound time complexity of $\Omega(\lg n)$ per operation (where n is the number of nodes in the tree), for the name-resolution problem in late-binding dynamic OOP languages. This lower bound implies that the implementation of name-resolution (creation, maintenance, member look-up, etc. of a class hierarchy) in late-binding dynamic OOP languages will at least incur cost proportional to $\lg n$, where n is the number of classes in a hierarchy. In particular, this means that *all the operations needed to support dynamic look-up in a class hierarchy cannot be made constant-time.*

We also propose efficient new data-structures and algorithms for solving this problem which have a worst-case time complexity of $O(\sqrt[3]{n})$ per operation. This result is an improvement over existing approaches that typically have a complexity of $\Omega(n)$ per operation. The data structures are very practical and can be easily employed in actual implementations.

The abstraction of the name-resolution problem for dynamic OOP languages in terms of dynamic trees, the derivation of a lower bound on complexity of name-resolution, and the development of an efficient, new implementation technique for it, are the main contributions of this paper. To the best of our knowledge, this is the first rigorous study of implementation of name-resolution in late-binding dynamic object-oriented languages. Our study is inspired by our observation that implementation mechanisms needed for supporting inheritance

in dynamic OOP languages are similar to implementation mechanisms needed in various other programming paradigms (e.g., advanced forms of execution of logic programming [21], and certain Game-trees and Artificial Intelligence applications [24]). The connection between the problems present in different paradigms is not obvious until we abstract them. These observed similarities allow us to reuse techniques and results developed in one area to solve problems in other areas. All the results presented in this paper, to the best of our knowledge, are novel, and complement similar studies focused on more specialized languages or techniques [23,25,33,1,9]. The data structures adopted to develop our upper-bounds are generalizations and extensions of the tabling and caching mechanisms proposed by various researchers [9,8,15].

1.1 Object Oriented Programming

In object-oriented programming solution of a problem is expressed in terms of creation and manipulation of *objects*. Objects encapsulates a data-structure (data-fields) and its behavior (member functions). We will use the general term *attributes* to refer to the components of an object, namely, its data-fields and member functions. Objects are organized into *classes* (collections of objects with common properties) and classes of objects are organized into hierarchies (subclasses and superclasses). Powerful mechanisms, like inheritance, allows definition of classes/objects as refinements of other existing classes/objects. Hierarchies of classes can assume various formats, depending on the flexibility allowed by the language; e.g., they can be represented as a single tree (single rooted, single inheritance), as multiple trees (multi-rooted, single inheritance), or as arbitrary complex Direct Acyclic Graphs (DAGS) (multiple inheritance). Objects consist of instance variables (which define the state of the object) and methods (which define the behavior of the object)[1]. The structure of an object is defined in terms of the class it belongs to; furthermore the definition of each class is a result of a combination of the components of all its super-classes (via inheritance). Thus the presence of an attribute β in an object of class C depends on the presence of β in C or in any of its super-classes. Attributes can get redefined at different stages of the hierarchy, e.g., β in class C_1 can be redefined in C_2.

1.2 Search Problems in OOP

From the previous discussion it follows that determining the structure of an object (i.e., its attributes) requires a *search*. A static system where the hierarchy is statically defined allows this search to be completely performed at compile-time. Whenever a class definition is processed, the hierarchy can be studied to determine exactly all the variables and all the methods that characterize each instance of such class.

In more complex object oriented system, such as those that allow dynamic creation of classes and/or late binding, part of this search has to be performed at run-time. This search can be more complex, as the look-up table, or its equivalent, may have to be constructed at runtime since the class hierarchy may be

[1] For simplicity we ignore the various distinctions, such as class variables.

extended at runtime. In the next subsections we illustrate situations where more complex search operations are required to determine attributes of a given object.

Self-reference and late binding are considered two essential components of the inheritance mechanism in an object-oriented system; in fact, various researchers [6,7,30] define inheritance as *"...an incremental modification mechanism in the presence of a late-bound self-reference"* [30]. The presence of polymorphism in object-oriented programming allows the assignments of objects belonging to a class C to variables of type B, where B is a base class for C. Consider the following example[2]: given the classes square and circle, which are subclasses of the base class shape

```
class shape {
 protected:
  char *name;
 public:
  void display()
  { ... }};
```

```
class square {
 float side;
 public:
  ....
  void display()
  { ... }};
```

```
class circle {
 float radius;
 public:
  ...
  void display()
  { ... }};
```

In an application program it is possible to define an array of pointers to shape:

```
shape *array[5];
circle little_circle(2.0);
square big_square(100.0);
```

and successively assign to it values taken from the subclasses circle and square

```
array[0] = &little_circle; array[1] = &big_square;
```

The execution of the methods `array[0]->display()` and `array[1]->display()` should lead to the execution of the correct methods (i.e., those associated to the classes circle and square respectively). The impossibility of determining statically which method should be called implies the need to dynamically detect the correct applicable method at runtime. This process of dynamically detecting the correct method to execute is referred to as *late (or dynamic) binding*. Most of the current object-oriented languages (e.g., C++ and Java) rely on late binding (either implicitly, as Java, or explicitly, as for the *virtual* functions of C++).

Compilers of languages that allow only a static structuring of the class hierarchies can efficiently realize late binding. For example, in C++, a simple table of pointers to methods is associated with each class instance (the *VTable*); at runtime determining the applicable method just involves a simple table lookup. However, there are other languages which allow *dynamic* extension of the classes hierarchy at runtime (e.g., Dynamic Object Oriented languages such as *CLOS* and Smalltalk), allowing creation of new classes on the fly. In this framework, smarter mechanisms are needed to organize class methods so that the correct applicable method can be determined for a given call. This determination may now, however, involve search of the class hierarchy at run-time. How to efficiently perform this search is the problem we address and analyze in this paper. Similar situation occurs (with greater frequency) in *prototype based object-oriented languages* [2,4,29]; in these frameworks the concept of class is not present, rather,

[2] For simplicity we adopt a C++-like syntax, even if C++ is not a dynamic language.

it is replaced by the idea of *prototypes*. Objects and prototypes are created on the fly and each object/prototype can be cloned to create new objects; *parent* attributes are used to connect the objects into hierarchies; execution of methods is based either on *delegation*—i.e., messages that do not match a local method are delegated to the parent object (e.g., in Self [5])—or on inheritance (e.g., Omega [2]). The focus of this work is mainly towards these classes of languages, characterized by heavy run-time requirements and limited compiler intervention.

The problem becomes more complex when multiple inheritance is considered. In multiple inheritance a class is allowed to inherit attributes from different base classes—thus generating possible inheritance conflicts. Existing multiple inheritance systems adopt different methodologies to solve these problems:

- *linearization* of the inheritance DAG, i.e. the DAG is topologically sorted and the list of nodes is traversed during name-resolution (e.g., as in CLOS);
- *tree inheritance* where the inheritance DAG is converted into a tree by duplicating (some of the) nodes reachable via different paths;
- *graph-based approach* where the DAG is not transformed and, in presence of conflicts, the programmer specifies how they should be resolved.

Approaches based on linearization will lead to situations that are analogous to those of single inheritance. Tree-inheritance generates a tree which is rooted at the most defined class and which has to be traversed in a predefined order.

1.3 Search Problems in Other Areas

The situation illustrated in the previous subsections is not unique to object-oriented programming; similar types of search operations occur in various different contexts. Our interest in the problem of implementing inheritance mechanisms was in fact spawned by our previous research [21] in the area of logic programming and search in Artificial Intelligence. Various forms of execution of logic programming (e.g., or-parallel execution of logic programming) progress by developing a computation tree (in parallel), where the different branches represent alternative computation paths, leading to possibly different solutions to the original query. Variables created before a branching point can be instantiated to distinct values along different branches; every time a variable is referenced, the correct value of the variable has to be determined (which is the value assigned to that variable along the current branch of execution). This problem closely resembles the one we are considering here for object-oriented programming—with the exception that in logic programming a variable can be assigned only once along a branch, while in OOP programming an attribute can be redefined multiple times along one branch of the hierarchy.

2 Pointer Machines

A *Pure Pointer machine (PPM)* consists of a finite but expandable collection R of *records*, and a finite collection of *registers*. Each record is uniquely identified through an address (let $\mathcal{N}$ be the set of all the addresses). A special address *nil* is used to denote an invalid address. Each record is a finite collection of named

fields. All the records in the global memory have the same structure, i.e. they all contain the same number of fields, with the same names. Each field may contain only an address (i.e., an element from $\mathcal{N}$). The pointer machine is also supplied with a finite collections of registers, $r_1, r_2, \ldots$ (pointer registers). Each register r_i can contain one element of $\mathcal{N}$. A program P is a finite, numbered sequence of instructions. The instructions allow to move addresses between registers and between registers and records' fields. The only "constant" which can be explicitly assigned to a register is *nil*. Special instructions are used to create a new record and to perform conditional jumps. The only condition allowed in the jumps is equality comparison between two pointer registers. In terms of analysis of complexity, each instruction has a unit cost. For further details on the structure of the pure pointer machine the reader is referred to [18,31,27]. Comparisons of the relative power of PPM and RAM have been presented in [27].

Even though RAM is the most commonly used model in studies of complexity of sequential algorithms, Pointer Machines have received increasing attention. The Pointer Machine model is simpler, thus making it more suitable for analysis of lower bounds of time complexity [31]. Furthermore, RAM hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ (n being the size of the input) to be treated as constant-time operations. PPM instead makes these (*relevant*) costs explicit. From this point of view, PPMs provide a more realistic model for measure of time complexity [27].

3 Abstraction of the Problem

3.1 Preliminary Definitions

A tree $\mathcal{T} = \langle N, E \rangle$ is a connected, directed, and acyclic graph. The tree contains a distinguished node, root which has no incoming edges. Each node in the tree has bounded degree. Trees are manipulated through three instructions: *(i)* create_tree() which creates a tree containing only the root; *(ii)* expand(n, b_1) which, given a node n and a label b_1, creates a new node and adds it as new children of n (b_1 can be thought of as the "name" of the new node); *(iii)* remove(n) which, given a leaf n of the tree, removes it from the tree. These three operations are assumed to be the only ones available for modifying the "physical structure" of the tree. The tree implements a partial ordering between the nodes in the tree. Given two nodes n and n', we write $n \preceq n'$ if n is an ancestor of n'; $n \prec n'$ additionally says that $n \neq n'$.

3.2 Abstracting Inheritance

The abstraction of an OOP execution should account for the various issues present in OOP, (e.g., definition of new classes/objects, inheritance of attributes etc.). Previous research, as well as experience gained from the development of practical implementations [30,26,13,34] shows that management of attributes is indeed one of the key issue in the implementation of an OOP system. The impact of the efficiency of attribute management on an OOP system has been

experimentally demonstrated [16,35,10,11]; furthermore, exploitation of parallelism and adoption of more complex approaches to object orientation make the problem of attribute management even more complex and difficult to implement. Thus, efficient implementation of attribute management is an important problem in implementation of OOP languages, if not the most fundamental one.

As described earlier, one of the main issues in dealing with inheritance in OOP is the search of a given attribute. This search may be needed when a class/object is created (in order to detect which variables should be included in each object) or during execution (in order to detect what is the correct method to execute in response to a given message). In the rest of the discussion we will adopt the following assumptions: the hierarchy is represented by a single-rooted tree which can dynamically grow and shrink; each node either introduces one attribute or redefines one. each tree has a bounded degree—thus, without loss of generality, we can focus on binary trees. In section 6 we will illustrate how most of these assumptions can be lifted, without affecting the results obtained. The tree representing the hierarchy to be searched will be referred to as the *o-tree*.

The development of the OOP computation takes place as described in the previous section. A hierarchy—here assumed to be represented by a dynamic tree of either classes (class-based system), or objects (prototype-based system)—is dynamically created. Each new leaf created in the tree refines the object(s) defined at its parent node, by adding attributes and/or shadowing existing ones. The structure (set of attributes) of an object at node N (denoted by $S(N)$) is thus determined by a combination $d(N, d(N_1, d(N_2, \ldots)))$ of the definitions present in the nodes N_i, where N_i is the ancestor of N at distance i from node N. d is a function with signature

$$d : Nodes \times 2^{Nodes \times Attr} \rightarrow 2^{Nodes \times Attr}$$

where $Attr$ is the set of possible attributes and $Nodes$ is the set of nodes in the hierarchy. Each object is described by a set of pairs $\langle N, \psi \rangle$, where each pair identifies one attribute composing the object; ψ determines the attribute and N determines the node (class/prototype) from where we are going to inherit the definition of ψ. If $\gamma(N)$ is the attribute in node N, then $d(N, \mathcal{S})$ is defined as

$$d(N, \mathcal{S}) = \{\langle N, \gamma(N) \rangle\} \cup \{\langle M, \psi \rangle \mid \langle M, \psi \rangle \in \mathcal{S} \ \wedge \ \psi \neq \gamma(N)\}$$

We can assume that a total of m attributes are available (i.e., there are m variables and methods). If the computation tree has size n, then we can assume that $m = O(n)$ (see also section 6). If $\langle M, \psi \rangle \in S(M)$ and there are no further definitions of ψ in the sequence of path from M to a descendent node N, then the path from M to N is *ψ-free*.

At each node N two operations on attributes are possible:

1. assign an attribute ψ to a node N; as a consequence, every reference to ψ in a node $M \succeq N$ (such that the path from N to M is ψ-free) will produce N as result (i.e., identifies that $\langle N, \psi \rangle$ is present in $S(M)$).
2. dereference an attribute ψ at a node N—i.e., identify the node $max_{\preceq}\{M \mid M \text{ defines } \psi \ \wedge \ M \preceq N\}$

In the formalization proposed here we assume an additional operation:

3. alias two attributes ψ_1 and ψ_2; This means that every reference to ψ_1 (ψ_2) in a node $M \succeq N$ will produce the same result as ψ_2 (ψ_1).

The alias operation is slightly unusual; we introduce it in this abstraction in order to allow the representation of constructions like the ability to access redefined properties by using superclass names as qualifiers (as in C++), and the *super* (Smalltalk) and *resend* (Self), as well as the "inverted" behaviour of languages like Beta (with the *inner* construct). The alias operation subsumes both these behaviours. In the rest of this work we will deal briefly with this operation; most of the complexity results are *independent* from the presence of the alias operation—i.e., the cost of the alias operation is negligible with respect to the complexity of the remaining operations. In the previous abstraction we have assumed the presence of one attribute per node. This restriction can be made without loss of generality if we can assume that the number of attributes in the node is bound by a program dependent constant. Section 6 considers possible relaxations of this condition. Let us refer to the problem of supporting these dynamic tree operations (assign, dereference, and alias) as the $\mathcal{OOP}$ problem. Later we are going to discuss the problems of *aliasing* and *dereferencing* separately. Figure 1 illustrates an example of an o-tree.

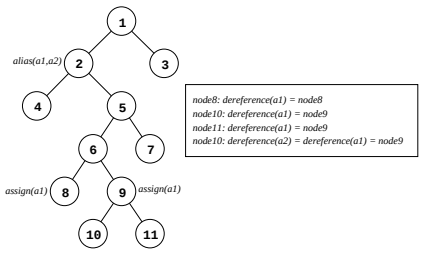


Fig. 1. Operations on o-tree

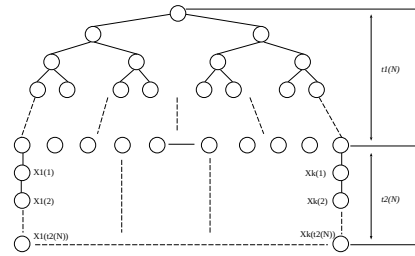


Fig. 2. Complex Tree

All the operations need to be implemented efficiently to guarantee adequate performance. Our abstraction of the problem properly accounts for *all* the costs incurred during execution and is directly related to the computation in the hierarchy. Thus, even schemes which move part of the cost of managing search during object creation are properly represented in our abstraction—such costs will be accounted for in the abstraction during the expansion of a node.

4 Lower Bound for $\mathcal{OOP}$

In this section we will develop a lower bound on the complexity of the operations required in our abstraction of OOP. The initial goal was to demonstrate that a dynamic late-binding OOP system cannot be maintained with a constant-time cost per operation. This result, as shown below, can be easily proved just based

on the behaviour of the alias operation alone. Nevertheless, the real source of complexity arises from the need to maintain dynamic references that, in each branch of the hierarchy, allow to resolve every attribute reference. This produces a considerably higher lower bound.

It is quite easy to observe that along each branch of the tree, attributes are managed in a way very similar to in the union-find problem. Initially all attributes are distinct; given an attribute X, let us denote with $\phi(X)$ the set of all the attributes that are currently (on the branch at hand) aliased to X. If a new aliasing between X_1 and X_2 takes place, then the set ϕ is modified as $\phi(X_1) = \phi(X_1) \cup \phi(X_2)$. If we assume that the value of an attribute X is always associated to the representative of $\phi(X)$, then any dereference operation is based on a find operation in order to detect the representative. We also assume (as addition to the union-find model) that the representative is selected as the node in $\phi(X)$ which is closer to the root. This observation can be used to provide a proof of the non-constant time nature of the computation involved in the management of attributes in a class-hierarchy—by showing that the $\mathcal{OOP}$ problem subsumes the union-find problem.

Theorem 1. *The amortized time complexity of $\mathcal{OOP}$ is $\Omega(n + m\alpha(m, n))$, where n is the number of attributes and m is the number of tree operations.*

This result relies on the execution of aliasing operations. Nevertheless, it is possible to show that the result holds even when we disallow aliasing during OOP execution. In fact, the inability to achieve constant-time behavior is inherent in the management of the attributes in the tree and *does not* depend on the presence of the alias operation.

Theorem 2. *On pointer machines, the worst case time complexity of $\mathcal{OOP}$ is $\Omega(\lg n)$ per operation even without aliasing.*

Detailed proof can be found elsewhere [20]. The above result is stronger than theorem 1, and gives us a greater insight into why it is impossible to support OOP with constant-time overhead.

5 Upper Bound for $\mathcal{OOP}$

In this section we develop an upper bound for the $\mathcal{OOP}$ problem. Most of the implementation schemes proposed in the literature either restrict the capabilities of the OOP system (allowing to collect sufficient information during compile-time and reduce run-time search to simple table lookups) or can be shown to have a worst case complexity of $O(n)$ per operation. Currently, the best result we have been able to achieve is the following:

Theorem 3. *$\mathcal{OOP}$ with no aliasing can be solved on a pointer machine with a single operation worst–case time complexity $O(\sqrt[3]{n}(\lg n)^k)$ for a small k.*

The $\mathcal{OOP}$ problem can be cast in the following terms. The computation is described in terms of operations on a dynamic tree. For the sake of simplicity we

will focus on a growing only tree. Thus, the only operation allowed on the tree is expand(v, X_1, X_2) which expands the leaf v by appending two successors u_1 and u_2, respectively labeled with X_1 and X_2. Here we are assuming, as mentioned above, that each node in the tree has bounded degree; thus, without loss of generality, we can as well restrict our attention to binary trees. Section 6 consider generalizations of this case. Also, for simplicity we assume that X_1 and X_2 are the actual attributes associated to such nodes—i.e., we avoid explicit use of assign. Only leaves can be expanded. The tree is *labeled*; thus a tree is coupled with a labeling function: l: *Nodes* $\rightarrow \Gamma$, where *Nodes* is the set of all the nodes in the tree and Γ is a finite set of labels. Given this, we can define the function

$$\text{dereference: } \Gamma \times \mathit{Nodes} \rightarrow \mathit{Nodes} \cup \{\bot\}$$

which given a label l and a node n returns the lowest (i.e., closer to n) node m (if any) on the path (from the root) ending in n which is labeled l. Our problem now reduces to efficient implementation of the two operation expand and dereference.

Let us start developing a solution under the assumption that the number N of nodes to be inserted in the tree is known in advance. For our purposes, we can think the labels to be stored directly in the nodes of the tree. Let us first consider the naive scheme. The expand operation simply expands a leaf attaching the two new successors. The dereference operation starts with a label and a leaf, and walks back towards the root searching for the desired label. Clearly, in this naive implementation, the expand operation is $O(1)$, but the dereference operation can take time $O(N)$ in the worst case, since the tree may be completely unbalanced. Alternatively, on the other extreme, we could require expand to create a complete summary the branch up to that point ($\tilde{O}(n)$), allowing a very fast dereference operation ($\tilde{O}(1)$). In both cases, a sequence of n operations will lead to a worst–case complexity of $\tilde{O}(n^2)$.

The general idea in our improved schemes is to balance the amount of work done during the expand and the dereference operations, by making the expand operation more complex in order to make the search more efficient. We first start by providing a simple $\tilde{O}(\sqrt{N})$ solution; we later improve the solution to $\tilde{O}(\sqrt[3]{N})$. This last solution is considerably more involved than the first one, but uses some of the ideas in the first one.

An $\tilde{O}(N^{\frac{1}{2}})$ scheme: The first algorithm that we are describing will lead to a complexity[3] $O(\sqrt{N}(\lg N)^k)$.

The behavior of dereference can be improved by creating at various points of each branch a "table" that summarizes the actions performed on that segment of the branch. This is illustrated in figure 3. Table *T1* summarizes the assignment of labels to nodes for the part of the branch between nodes `n1` and `n2`; table *T2* summarizes the assignment of labels to nodes for the part of the branch between nodes `n3` and `n4`. Nodes that are associated to tables are linked together. Note that, although we talk conceptually about tables, these tables are actually implemented as binary search trees on pointer machines. We can do this because we can assume the existence of a total order between the labels (we can remove this assumption by paying an $O(\lg N)$ penalty). The execution of dereference

[3] For the sake of simplicity in the following analysis we will ignore the $(\lg N)^k$ factor.

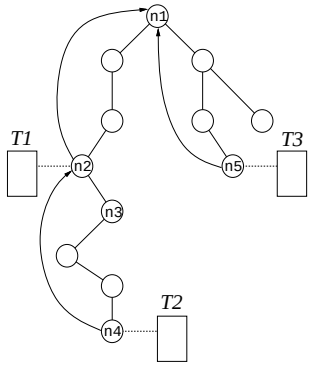


Fig. 3. Use of Tables

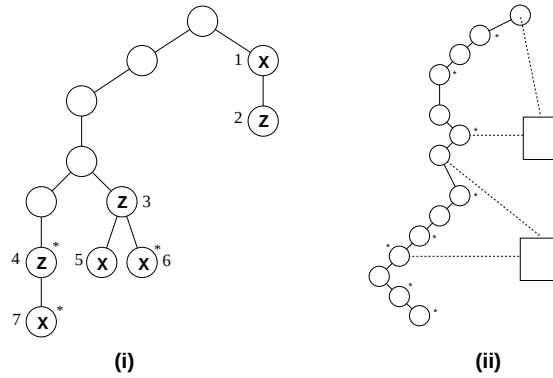


Fig. 4. Examples of Trees

requires linearly scanning the branch from the leaf up to the first table node. If the desired label is not found in such part of the branch, then the table is examined. If the label is not found, then the search is repeated in the preceding table, and so on. If the root is reached, then the searched label has not been assigned to any node on that branch.

The complexity of this process is the following: let us assume that the tables are created every $t(N)$ nodes.

- the initial linear search will not visit more than $t(N)$ nodes;
- searching inside a table will require $O(\lg t(N))$;
- since there are at most $O(N)$ nodes on the branch, then the search process will visit at most $O(\frac{N}{t(N)})$ tables.

Combining these elements, we can state that the complexity of this implementation of dereference is

$$O(t(N) + \lg t(N)\frac{N}{t(N)})$$

The expand operation, on the other hand, has a worst-case time complexity $\tilde{O}(t(N))$ per operation (occurring at the time of table creation). Choosing $t(N) = \sqrt{N}$ the total cost[4] is $O(\sqrt{N} \lg N)$.

An $\tilde{O}(N^{\frac{1}{3}})$ Solution: The above scheme can be improved by making use of the solution to the ancestor problem—the problem of determining if one node is ancestor of the other in a dynamic tree. This is a well-studied problem with very efficient solutions [12,14,18,32]. Let $\beta(n)$ denote the complexity of solving the ancestor problem for a dynamic tree with n nodes. We know from the literature [32,14] that it is possible to solve the ancestor problem for two nodes x, y in time $\approx O(\lg \min(depth(x), depth(y)))$. Thus we can assume that $\beta(N) = O(\lg N)$.

The intuition behind this second solution is the following: if a certain label l has been assigned to very few nodes—let's say k nodes $m_1, \ldots, m_k$—it is pos-

[4] We choose to ignore the polylog factors in making an optimum choice for $t(N)$.

sible to solve the problem dereference(n,l) by simply considering each of the k nodes m_i and verifying whether m_i is an ancestor of n. This is possible only if k is sufficiently small. Observe that, in the worst case, the number of nodes associated to l can be $O(N)$. On the other hand, only "few" labels will be used very frequently. We can try to take advantage of these facts by collecting the assignments of labels to nodes into *groups*. Each group is supposed to have sufficiently small cardinality to allow its exhaustive search.

The general idea in the search scheme (dereference) would be to first determine the group of the label being searched, and then perform an exhaustive search within the group, using the solution to the ancestor problem. Let $t(N)$ be the cardinality of a group. Thus, we associate to each label γ a set of groups, containing all the nodes n in the tree such that $l(n) = \gamma$. During the construction of the tree, every time we create a node m with label γ, we insert m in the current group for γ. If the current group is full—i.e., it contains $t(N)$ elements—then a new group for γ is created, and m is added to such group. Nodes which lead to the creation of a new group are called *boundary nodes*. Each boundary node is properly marked, and contains a pointer to the new group.

Additionally, if the node n is inserted for the label γ, and γ has ever before received a new group, then n is a *used node*, and it is also marked. More formally:

Definition 1. *Given a label γ, we indicate with $\#(\gamma)$ the number of groups currently existing for γ. $G(\gamma, i)$ indicates the i^{th} group for γ (if $1 \leq i \leq \#(\gamma)$). $|G(\gamma, i)|$ indicates the cardinality of the ith group of γ.*

The used and boundary nodes can be defined as follows.

Definition 2. *A node n such that $l(n) = \gamma$ is said to be a* **boundary node** *if, at the moment of its creation, $|G(\gamma, \#(\gamma))| = t(N)$. A node n such that $l(n) = \gamma$ is said to be a* **used node** *if $\#(\gamma) \geq 2$.*

Figure 4(i) schematically illustrates these concepts with a simple example. The example assumes $t(N) = 2$. The numbers indicate the order in which the indicated nodes have been inserted. Node 4 is a boundary node as it requires the creation of a new group for `Z`; for this reason it has been marked. Node 6 is also a boundary node, for label `X`. Node 7 is a used node, since it is inserting a label for which there has already been a group change (due to node 6); for this reason the node is also marked.

Observe that, since there are N nodes in the tree, and a group has size $t(N)$, then during the insertion there will be at most $\frac{N}{t(N)}$ groups. Observe also that given that there are N nodes, there can be at most $\frac{N}{t(N)}$ labels having more than one group. This means that along each branch of the tree, we cannot have more than $O(N/t(N))$ nodes that have been marked (either used or boundary).

Considering the organization of label assignments into groups, a strategy to implement the dereference operation would be to simply visit the marked nodes along the current branch (we can assume that they are linked together). If a

node with the correct label is found, then we have an immediate solution—determined in $O(N/t(N))$. If a node with the correct label is not found, there are two possibilities left:

- the label has been used in the branch and the assignment was generated before filling the first group of that label; thus the assignment has been recorded in the first group for that label. The correct node can be found in time $O(t(N)\beta(N))$.
- the label has not been used at all in the branch. Again, if we search the nodes in the first group ($O(t(N)\beta(N))$) we are going to determine this situation.

Thus, the process will lead to a solution in $O(t(N)\beta(N) + N/t(N))$ steps.

The search can be improved by adding, as in the scheme described in the previous section, tables to summarize the group numbers associated to the used/boundary nodes in the previous section of the branch. Considering that we need to keep track of at most $N/t(N)$ nodes per branch—the maximum number of used and boundary nodes—using the results from the previous section we can determine that the best behavior can be achieved by placing a table at every $\sqrt{N/t(N)}^{th}$ used/boundary node along a branch. The effect of the table will be to summarize the previous $\sqrt{N/t(N)}$ used and boundary nodes generated along the current branch. Figure 4(ii) illustrates the situation. Marked nodes represent used/boundary nodes. Using this scheme, a search for a label starting from a given leaf will proceed as follows:

- starting from the leaf, the various used/boundary nodes are accessed up to the first table. If the label is found within these nodes, then we have immediately an answer to the problem. Considering the assumption made before, this first phase will traverse at most $O(\sqrt{N/t(N)})$ nodes.
- if the label is not found in the used/boundary nodes at the end of the branch, then the first table is searched for the considered label. This search will cost $O(\lg(N/t(N)))$.
- if the label is found in the table, then we can assume that the entry will directly refer to the correct group. Otherwise the search has to be repeated in the preceding table. At most $\sqrt{N/t(N)}$ tables are going to be searched, leading to a total time of $O(\sqrt{N/t(N)}\lg(N/t(N)))$.
- finally, if the label is not found in any of tables, then the group of this label never changed (hence is 1). A final search has to be performed in the initial group for the label, which will imply a cost of $O(t(N)\beta(N))$.

Thus, we obtain a total complexity for the dereference operation of:

$$O(\sqrt{N/t(N)} + \sqrt{N/t(N)}\lg(N/t(N)) + t(N)\beta(N))$$

Choosing $t(N) = \sqrt[3]{N}$, noting that $\beta(N)$ is $O(lgN)$ we get that the complexity of dereference is $\tilde{O}(\sqrt[3]{N})$.

To conclude the argument, we need to tackle the two assumptions made at the beginning of the proof:

1. the argument was focused on growing-only trees, i.e., we ignored the remove operation. It is relatively easy to observe that this operation is applied only to leaves of the tree and hence does not affect the validity of the results.

2. we assumed at the beginning that the number of nodes N is known beforehand. One can show that the use of the *current* number of nodes in place of the *total* number of nodes is going to produce variations only in the polylog part of the complexity. We made this assumption to keep the presentation of the proof simple and easy to follow.

It is interesting to observe that the average case behavior can be improved by adopting a better structuring of the nodes in each group. In particular the nodes in a group can be organized as a tree which reflects the ancestor relationship present in the original dynamic tree—which would, on average, improve the search in a group. Also, a different criteria can be used to switch to a new group: a new group is created whenever the tree present in the current one has reached a specified height. This criteria would guarantee the same worst case behavior as before but may allow to place a larger number of nodes within the same group, potentially leading to a reduction in the time required to scan the main tree. □

The above scheme allows to gain good efficiency on all the requested operations. The scheme increases memory consumption; nevertheless the space complexity is in the order of $O(2 \times n)$.

6 Generalizing the Lower and Upper Bounds

There are various assumptions that we made in the beginning. These assumptions were made to keep the presentation simple, and to derive the lower and upper bounds. Relaxing these assumptions will not affect the lower bound, as the problem will become more complex if an assumption is relaxed. However, relaxing an assumption can increase the upper bound. The effect of relaxing the assumptions is discussed next.

- **Number of Attributes per Node** The treatment of the previous sections was based on the assumption that a single attribute is associated to each node in the tree. As long as the total number of attributes defining a class/object is bounded by a given constant, the lower and upper bounds change only up to a constant. Introducing arbitrary numbers of attributes in a node will add a penalty for searching the attribute; if we assume k to be the number of attributes per node, then a $O(\lg k)$ penalty per operation will be incurred.
- **Degree of the Nodes** In most of the previous treatment we have assumed the use of a bounded degree tree (in particular most of the discussion focused on binary trees). Nevertheless, by looking at the structure of the various proofs we can observe that the issue does not affect the final result. The various formulae proposed are based on the independent structure of each separate path, and it is not affected by the degree of the various nodes. Thus this assumption can be easily relaxed without affecting the final results.
- **Arbitrary Insertion Points** We assumed in deriving the upper bound that the o-tree grows only at the leaf nodes. This assumption is related to the previous one. However, note that adding new children to any node in the tree (even internal nodes) does not affect the existing paths in the tree. Thus, the schemes here provided can be easily modified to accommodate a more general version of the expand operation—the only change required in the $O(\sqrt[3]{n})$

scheme is keeping an additional pointer with each node which points to the most recent boundary/used node. This can be easily maintained with a constant additional overhead.

- **Modification of Internal Nodes** Various dynamic object oriented systems allows modification of existing classes or prototypes, by adding new attributes or changing existing ones. This would imply, in our model, the ability of modifying attributes of internal nodes of the tree. If we allow such operations, then the upper bound is affected.
  The operation of replacing an individual attribute β in an internal node v with an attribute γ can be implemented by updating the information stored in node v and by maintaining an external table which records the replacement operations performed. A search for γ will require two traversal of the data structure, one searching directly for γ, and one for β (where β has been detected from the global replacement table).
  In general, if k replacements have been performed in the whole data structure, then we have a degradation of complexity to $O(k\sqrt[3]{n})$ per operation.

7 Conclusions

In this paper we studied the complexity of supporting single inheritance (or multiple inheritance with linearization), in late-binding dynamic OOP systems. We focused on the complexity of the operations associated with determining the correct attributes to be used with a given class/object. We developed lower and upper bounds for the $\mathcal{OOP}$ problem, i.e., the problem of implementing the basic operations required for correctly managing attribute detection.

The lower bound produced, $O(\lg n)$, is a confirmation that an ideal situation where all the required operations can be implemented in constant-time is not feasible. This seems to confirm the results achieved for similar problems by other authors [25,23]. The upper bound, $\tilde{O}(\sqrt[3]{n})$, even if far from the lower bound, is of great importance, as it suggests that current implementation mechanisms are sub-optimal and better implementation schemes are feasible. In fact, from the implementation proposed for various systems, it seems that mostly naive schemes (typically requiring $O(n)$ time per operation) have been adopted. The scheme presented handle the $\mathcal{OOP}$ without the use of the alias operation. Nevertheless, from [3] we can infer that the alias operation can be implemented with worst–case complexity $O(\frac{\lg n}{\lg \lg n})$, thus adding just a lower order component to $\sqrt[3]{n}$.

The results presented in this work are mainly related to worst–case analysis of the problems. Nevertheless, all the worst–case situations depicted are reasonable and we expect the average–case complexity (which is heavily dependent on the specific application) to be sufficiently close to the results derived here.

An open problem of interest is the development of better upper bounds for the $\mathcal{OOP}$ in case of no aliasing between variables. We have shown that this problem has a non-constant time lower bound, but an optimal upper bound is still a subject of research. Our conjecture is that the problem admits a solution with a worst case time complexity $O((\lg n)^k)$ (for a small k) per operation.

Finally, it is important to observe that the $\mathcal{OOP}$ problem is actually a very general problem, which occurs in almost the same form in various different frameworks. For example, [24,21] illustrates how the same problem arises in the context of parallel execution of logic programming and in various artificial intelligence applications. Thus, the results presented here are very general and can be applied to problems outside of OOP.

References

1. E. Amiel, O. Gruber, and E. Simon. Optimizing Multi-method Dispatch using Compressed Dispatch Tables. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1994. 215
2. G. Blaschek. *Object Oriented Programming with Prototypes*. Springer Verlag, 1994. 214, 216, 217
3. N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4), 1986. 227
4. A. Borning. Classes versus Prototypes in Object Oriented Languages. In *Fall Joint Computer Conference*. ACM/IEEE, 1986. 214, 216
5. D. Ungar C. Chambers and E. Lee. An Efficient Implementation of SELF. In *OOPSLA*. ACM, 1989. 217
6. W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989. 216
7. W.R. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proc. OOPSLA*. ACM, 1989. 216
8. P. Deutch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings POPL*. ACM Press, 1984. 215
9. K. Driesen. Selector Table Indexing and Sparse Arrays. In *Proceedings OOPSLA*. ACM Press, 1993. 215
10. K. Driesen and U. Holzle. Minimizing Row Displacement Dispatch Tables. In *Conf. on OO Progr. Systems, Lang., and Applications*. ACM Press, 1995. 219
11. K. Driesen and U. Holzle. The Direct Cost of Virtual Function Calls in C++. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1996. 219
12. H.N. Gabow. Data structures for weighted matching and nearest common ancestor. In *ACM Symp. on Discrete Algorithms*, 1990. 214, 223
13. A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983. 218
14. D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2), 1984. 214, 223
15. U. Holzle, C. Chambers, and D. Ungar. Optimizing Dynamically-typed Object-Oriented Languages with Polymorphic Inline Caches. In *Proceedings ECOOP*. Springer Verlag, 1991. 215
16. U. Holzle and D. Ungar. A Third Generation SELF Implementation. In *Conf. on Object Oriented Progr. Systems, Languages, and Applications*. ACM Press, 1994. 219
17. S.E. Keene. *Object-oriented Programming in Common Lisp*. Addison Wesley, 1988. 214
18. D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968. 214, 218, 223

19. A.N. Kolmogorov and V.A. Uspenskii. On the Definition of an Algorithm. *Uspehi Mat. Nauk*, 13, 1958. 214
20. E. Pontelli, D. Ranjan, and G. Gupta. Late-binding in Dynamic Object Oriented Systems. Technical Report, New Mexico State Univ., 1998. 221
21. E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In *Procs Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*. Springer Verlag, 1997. 214, 215, 217, 228
22. H. La Poutré. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996. 214
23. G. Ramalingam and H. Srinivasan. A Member Lookup Algorithm for C++. In *Programming Languages Design and Implementation*. ACM Press, 1997. 215, 227
24. D. Ranjan, E. Pontelli, and G. Gupta. Dynamic Data Structures in Advanced Programming Languages Implementation. Technical report, Dept. Computer Science, New Mexico State University, 1997. 215, 228
25. J.G. Rossie and D.P. Friedman. An Algebraic Semantics of Subobjects. In *Conf. on Object-Oriented Progr. Systems, Languages, and Applications*. ACM Press, 1995. 215, 227
26. J.H. Saunders. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, 1(6), 1989. 218
27. A. Schönhage. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980. 214, 218
28. D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26, 1983.
29. L. Stein. Delegation is Inheritance. In *OOPSLA*. ACM, 1987. 214, 216
30. A. Taivalsaari. On the Notion of Inheritance. *Computing Surveys*, 28(3), 1996. 216, 218
31. R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18), 1979. 218
32. A.K. Tsakalidis. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25, 1988. 223
33. J. Vitek, R. Nigel Horspool, and A. Krall. Efficient Type Inclusion Tests. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1997. 215
34. P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1), 1990. 218
35. O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1997. 219

A Compiler for Rewrite Programs in Associative-Commutative Theories

Pierre-Etienne Moreau and Hélène Kirchner

LORIA-CNRS & INRIA-Lorraine
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
moreau,hkirchne@loria.fr

Abstract. We address the problem of term normalisation modulo associative-commutative (AC) theories, and describe several techniques for compiling many-to-one AC matching and reduced term construction. The proposed matching method is based on the construction of compact bipartite graphs, and is designed for working very efficiently on specific classes of AC patterns. We show how to refine this algorithm to work in an eager way. General patterns are handled through a program transformation process. Variable instantiation resulting from the matching phase and construction of the resulting term are also addressed. Our experimental results with the system ELAN provide strong evidence that compilation of many-to-one AC normalisation using the combination of these few techniques is crucial for improving the performance of algebraic programming languages.

Keywords: compilation, rewrite systems, AC theories, AC many-to-one matching.

1 Introduction

In the area of formal specifications and algebraic programming languages, rewrite programs (i.e. sets of rewrite rules) are used for prototyping computations in user-defined data types. In this context, mathematic and algebraic structures often involve properties of function symbols, such as associativity and commutativity. For example, let us consider polynomials with integer coefficients simplify monomials with a null coefficient. This is simply achieved by using a rule $P + (0 * M) \rightarrow P$ where P is a polynomial and M a monomial. This rule takes advantage of the associativity and commutativity property of operators $+$ and $*$: all permutations needed to find a reducible monomial are performed by an AC matching algorithm. Clearly AC operators lead to more concise and elegant rewrite programs, however providing this facility in rule-based languages requires specific implementation techniques. Straightforward implementations of AC matching and normalisation can lead to inefficiency that discourages the programmer to use this feature. So it is a real challenge, addressed in this work, to achieve efficient computations in AC theories.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 230–249, 1998.

A first idea is that efficient execution of rewrite programs can be obtained by designing specific techniques for frequently used patterns. It is natural then to examine which patterns are the most usual ones in rule-based specifications, handle these cases first and use a more powerful, but in general more expensive, method only when needed. This led us to design many-to-one AC matching for classes of patterns. The general case is handled through a preliminary transformation technique of programs so that they fall into the previous classes, and a combination of our many-to-one approach with a general one-to-one matching algorithm [9]. More details and proofs can be found in the PhD Thesis of the first author.

Construction of substitutions and the result are also operations involved in normalisation, that must be carefully designed for efficiency. The difficulty of achieving a compiler for AC rewriting is not only to improve each technical step (often starting from some solutions already proposed in the literature) but also to combine all these steps together to achieve an efficient implementation. We give experimental results and comparisons with other systems to provide some evidence of efficiency.

After introducing preliminary concepts in Section 2, we present our compilation techniques, in Section 3 for matching, and in Section 4 for normalisation. Experimental results are provided in Section 5, before concluding with a few perspectives in Section 6.

2 Preliminary Concepts

We assume the reader familiar with basic definitions of term rewriting and AC theories, given for instance in [26,17].

$\mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a set $\mathcal{X}$ of variables. Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the top-most position. The subterm of t at position ω is denoted $t_{|\omega}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term*. A term t is said to be *linear* if no variable occurs more than once in t. A *substitution* is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. Given a binary function symbol f_{AC}, let AC be the set of associativity and commutativity axioms

$$f_{AC}(x, f_{AC}(y,z)) = f_{AC}(f_{AC}(x,y),z) \quad \text{and} \quad f_{AC}(x,y) = f_{AC}(y,x).$$

All AC function symbols that satisfies these two axioms, are indexed by AC in the following. We write $s =_{AC} t$ to indicate that the terms s and t are equivalent modulo associativity and commutativity. $\mathcal{F}_\emptyset$ is the subset of $\mathcal{F}$ made of function symbols which are not AC, and are called *free* function symbols. A term is said to be *syntactic* if it contains only free function symbols.

A *conditional rewrite rule* is a triple of terms denoted $l \to r$ **if** c such that $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, c is a boolean term, and $\mathcal{V}ar(c) \cup \mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. c is called the *condition*, l the *left-hand side* or *pattern* and r the *right-hand side*. A rewrite rule

is said to be *syntactic* if the left-hand side is a syntactic term. To apply a syntactic rule to a subject t, we look for a match, i.e. a substitution σ satisfying $l\sigma = t$. The algorithm which provides the unique substitution σ whenever it exists is called *syntactic matching*. When the condition $c\sigma$ reduces to the boolean term *true*, the subterm $l\sigma$ is replaced by $r\sigma$ in the subject. Otherwise, the application of this rule fails.

When the left-hand side of the rule contains AC function symbols, then AC matching is invoked. The term l is said to AC match another term t if there exists a substitution σ such that $l\sigma =_{AC} t$. Since AC matching can return several solutions, this introduces a need for backtracking for conditional rules: as long as there is a solution to the AC matching problem for which the condition is not satisfied, another solution has to be extracted. Only when no solution is remaining, the application of this rule fails. Conditional rewriting requires AC matching problems to be solved in a particular way: the first solution has to be found as fast as possible, and the others have to be found "on request".

AC matching has already been extensively studied, for instance in [16,2,20,1,21,9]. Usually all terms in the same AC equivalence class are represented by their canonical form [16,9]. Given a total ordering $>$ on the set of function symbols $\mathcal{F}$ and variables, the canonical form is obtained by flattening nested occurrences of the same AC function symbol, recursively computing the canonical forms and sorting the subterms, and replacing α identical subterms t by a single instance of the subterm with multiplicity α, denoted by t^α. A term in canonical form is said to be *almost linear* if the term obtained by forgetting the multiplicities of variable subterms is linear, i.e. if a variable occurs more than once in a flattened term, it must be directly under the same AC function symbol. For a term t in canonical form, the *top layer* $\hat{t}$ is obtained from t by removing subterms below the first AC symbol in each branch.

In order to support intuition throughout this paper, we choose the following running example of two rewrite rules whose right-hand sides are irrelevant:

$$\begin{array}{ll} f_{AC}(z, f(a,x), g(a)) & \to r_1 \textbf{ if } z = x \\ f_{AC}(f(a,x), f(y, g(b))) & \to r_2 \end{array}$$

3 Many-to-One AC Matching

In programming with rewrite rules, it frequently happens that programs contain several rules whose left-hand sides begin with the same top function symbol. This often corresponds to definition by cases of functions. In this context, many-to-one pattern matching is quite relevant.

The many-to-one (AC) matching problem is the following: given a set of terms $P = \{p_1, \ldots, p_n\}$ called patterns, and a ground term s, called subject, find one (or more) patterns in P that (AC) matches s. Patterns and subject are assumed to be in canonical form. Efficient many-to-one matching algorithms (both in the syntactic case and in AC theories) are based on the general idea of factorising patterns to produce a matching automaton. The discrimination net approach [13,6,24] is of this kind. In the case of AC theories, the decomposition of

matching problems gives rise to a hierarchically structured collection of standard discrimination nets, called an AC discrimination net [1,14].

3.1 Description of the Algorithm

The skeleton of our many-to-one AC matching algorithm is similar to the algorithm presented in [1]. Given a set of patterns P,

1. Transform rules to fit into a specific class of patterns. In particular, patterns are converted to their canonical forms, rules with non-linear left-hand sides are transformed into a conditional rule with a linear left-hand side and a condition expressing equality between variables (of the form $x = y$);

2. Compute the AC discrimination net associated to $P = \{p_1, \ldots, p_n\}$ and the corresponding matching automata. Figure 1 shows the AC discrimination net for the set of patterns in our running example.

$$P = \{f_{AC}(z, f(a, x), g(a)), f_{AC}(f(a, x), f(y, g(b)))\}$$

where only f_{AC} is an AC symbol. The net is composed of two parts:

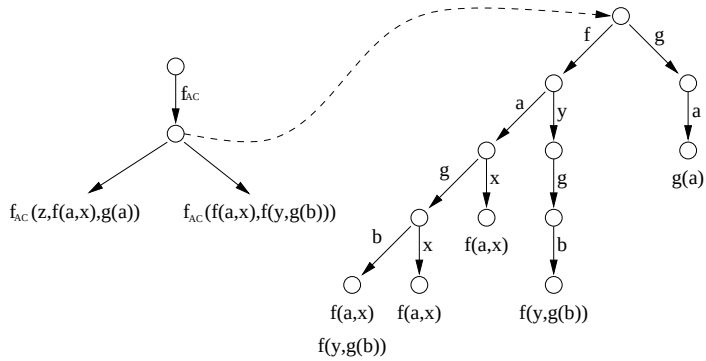


Fig. 1. Example of discrimination net

- a discrimination net for the top layers $\hat{P} = \{\hat{p_1}, \ldots, \hat{p_n}\}$ which determines which rules can be applied, and a link to the sub-automaton used to match subterms of AC function symbols (f_{AC});
- the sub-automaton itself that implements a many-to-one syntactic matching algorithm [13] for the set of syntactic terms: $f(a, x), f(y, g(b))$ and $g(a)$.

This decomposition is iterated if there are more AC function symbols.

The previous steps only depend on the set of rewrite rules. They can be performed once and for all at compile time. At run-time the subject is known and the following steps are performed.

Given a ground term $s = f_{AC}(s_1, \ldots, s_p)$,

3. Build the hierarchy of bipartite graphs according to the given subject s in canonical form. For each subterm $p_{i|\omega}$, where ω is a position of an AC function symbol in $\hat{p_i}$, an associated bipartite graph is built. For an AC matching problem from $f_{AC}(t_1, \ldots, t_m)$ to $f_{AC}(s_1, \ldots, s_p)$, where for some k, $0 \leq k \leq m$, no $t_1, \ldots, t_k$ is a variable, and all $t_{k+1}, \ldots, t_m$ are variables, the associated bipartite graph is $BG = (V_1 \cup V_2, E)$ whose sets of vertices are $V_1 = \{s_1, \ldots, s_p\}$ and $V_2 = \{t_1, \ldots, t_k\}$, and edges E consists of all pairs $[s_i, t_j]$ such that $t_j\sigma$ and s_i are equal modulo AC for some substitution σ.
This construction is done recursively for each subterm of $p_{i|\omega}$ whose root is an AC symbol.

For example, given the ground term $s = f_{AC}(f(a,a), f(a,c), f(b,g(b)), f(g(c), g(b)), g(a))$, the two bipartite graphs given in Figure 2 can be built (one for each rule);

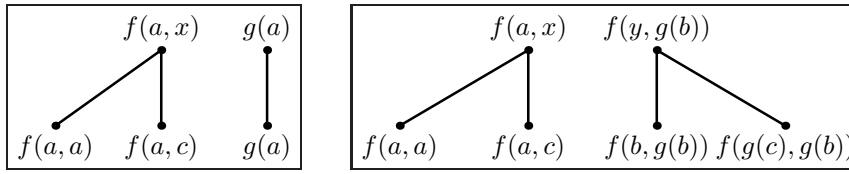


Fig. 2. Examples of bipartite graphs

4. Find a set of solutions to the hierarchy of bipartite graphs and construct a Diophantine equational system which encodes the constraints on the remaining unbound variables: in order to match m variables $x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}$ to n remaining subterms $s_1^{\beta_1}, \ldots, s_n^{\beta_n}$ one looks for non-negative integer solutions of the system $\bigwedge_{i=1..n} \beta_i = \alpha_1 X_i^1 + \cdots + \alpha_m X_i^m$ with $\Sigma_{j=1}^n X_j^i \geq 1$ in order to get solutions of the form $x_k = f_{AC}(s_1^{X_1^k}, \ldots, s_n^{X_n^k})$ for $k = 1..m$.

5. Solve the Diophantine equational system to get a matching substitution.

Starting from this quite general algorithm, our goal was to improve its efficiency by lowering the cost of some steps, such as traversing the levels of the hierarchy of discrimination nets, building and solving bipartite graphs, or solving Diophantine equational systems. The idea is to apply these costly steps on specific patterns for which they can be designed efficiently, or simply skipped. The classes of patterns presented in the following section cover a large class of rewrite programs. For the general case, a pre-processing of the rewrite program is done, as explained in Section 3.4.

3.2 Classes of Patterns

All terms in the pattern classes are assumed to be in canonical form and almost linear. The pattern classes C_0, C_1, C_2 contain respectively terms with no AC function symbol, at most one and at most two levels of AC function symbols.

Definition 1. *Let $\mathcal{F}_\emptyset$ be the set of free function symbols, $\mathcal{F}_{AC}$ the set of AC function symbols and $\mathcal{X}$ the set of variables.*

- *The pattern class C_0 consists of linear terms $t \in \mathcal{T}(\mathcal{F}_\emptyset, \mathcal{X}) \backslash \mathcal{X}$.*
- *The pattern class C_1 is the smallest set of almost linear terms in canonical form that contains C_0, all terms t of the form $t = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$, with $f_{AC} \in \mathcal{F}_{AC}$, $0 \leq n$, $t_1, \ldots, t_n \in C_0$, $x_1, \ldots, x_m \in \mathcal{X}$, $\alpha_1, \ldots, \alpha_m \geq 0$, $m \geq 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_\emptyset$, $t_1, \ldots, t_n \in C_1 \cup \mathcal{X}$.*
- *The pattern class C_2 is the smallest set of almost linear terms in canonical form that contains C_1, all terms t of the form $t = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$, with $f_{AC} \in \mathcal{F}_{AC}$, $0 \leq n$, $t_1, \ldots, t_n \in C_1$, $x_1, \ldots, x_m \in \mathcal{X}$, $\alpha_1, \ldots, \alpha_m \geq 0$, $m \geq 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_\emptyset$, $t_1, \ldots, t_n \in C_2 \cup \mathcal{X}$.*

In our example, the patterns $f_{AC}(z, f(a, x), g(a))$ and $f_{AC}(f(a, x), f(y, g(b)))$ belong to the class C_1. The pattern

$$f_{AC}(x_1^3, g(f_{AC}(z, f(a, x_2), g(a))), g(f_{AC}(f(a, x_3), f(y, g(b)))))$$

belongs to C_2.

It should be emphasised that for completeness of AC rewriting, extension variables have to be added [26,17]. Those variables store the context and allow us to do rewrite steps in subterms. Adding extension variables amounts to considering additional rule patterns of the form $f_{AC}(x, l_1, \ldots, l_n)$ for each rule with a left-hand side of the form $l = f_{AC}(l_1, \ldots, l_n)$. Note that the rule and its extension belong to the same pattern class.

3.3 Many-to-One AC Matching Using Compact Bipartite Graphs

The AC matching techniques described in this section are restricted to the class of patterns presented in Section 3.2. This restriction leads to several improvements of the general approach described in Section 3.1.

- Thanks to the restriction put on patterns, the hierarchy of bipartite graphs has at most two levels. Thus, the construction can be done without recursion;
- We use a new compact representation of bipartite graphs, which encodes, in only one data structure, all matching problems relative to the given set of rewrite rules;
- No Diophantine equational system is generated when there is at most one or two variables, with (restricted) multiplicity, under an AC function symbol in the patterns. Instantiating these variables can be done in a simple and efficient way;

– A preliminary syntactic analysis of rewrite rules can determine that only one solution of an AC matching problem has to be found to apply some rule. This is the case for unconditional rules or for rules whose conditions do not depend on a variable that occurs under an AC function symbol in the left-hand side. Taking advantage of the structure of compact bipartite graphs, a refined algorithm is presented to handle those particular (but frequent) cases.

Compact Bipartite Graph. Given a set of patterns with the same syntactic top layer, all subterms with the same AC top function symbol are grouped to build a particular bipartite graph called a *Compact Bipartite Graph* described below. Given a subject, the compact bipartite graph encodes all matching problems relative to the given set of rewrite rules. From this compact data structure, all bipartite graphs that the general algorithm would have to construct, can be generated. In general, the syntactic top layer may be not empty and several AC function symbols may occur. In this case, a compact bipartite graph has to be associated to each AC function symbol and their solutions have to be combined to solve the matching problem.

Such a decomposition leads us to focus our attention on sets of patterns $p_1, \ldots, p_n$ defined as follows:

$$\begin{array}{ccc} p_1 = f_{AC}(\ p_{1,1}\ , \ldots ,\ p_{1,m_1}\ ) \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ p_n = f_{AC}(\ p_{n,1}\ , \ldots ,\ p_{n,m_n}\ ) \end{array}$$

where for some k_j, $0 \leq k_j \leq m_j$, no $p_{j,1}, \ldots, p_{j,k_j}$ is a variable, and all $p_{j,k_j+1}, \ldots, p_{j,m_j}$ are variables. Given a subject $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$, the associated Compact Bipartite Graph is $CBG = (V_1 \cup V_2, E)$ where $V_1 = \{s_1, \ldots, s_p\}$, $V_2 = \{p_{j,k} \mid 1 \leq j \leq n, 1 \leq k \leq k_j\}$, and E consists of all pairs $[s_i, p_{j,k}]$ such that $p_{j,k}\sigma$ and s_i are equal (modulo AC) for some substitution σ.

Syntactic subterms $p_{j,k}$ are grouped together and a discrimination net that encodes a many-to-one syntactic matching automaton is built. This automaton is used to find pairs of terms $[s_i, p_{j,k}]$ that match together and build the compact bipartite graph as described above.

To handle patterns with two levels of AC function symbols, a two-level hierarchy of bipartite graphs has to be built. To each edge of the graph issued from an AC pattern, is attached the corresponding AC subproblem. This hierarchy is represented by a matrix of bipartite graphs where rows correspond to subject subterms, and columns correspond to patterns $p_{j,k}$.

Solving a Compact Bipartite Graph. Finding a pattern that matches the subject usually consists of selecting a pattern p_j, building the associated bipartite graph BG_j and finding a maximum bipartite matching [15,11]. Instead of building a new bipartite graph BG_j each time a new pattern p_j is tried, in our approach, the bipartite graph BG_j is extracted from the compact bipartite graph

$CBG = (V_1 \cup V_2, E)$ as follows:

$$BG_j = (V_1 \cup V_2', E') \text{ where } \begin{cases} V_2' = \{p_{j,k} \mid p_{j,k} \in V_2 \text{ and } 1 \leq k \leq k_j\} \\ E' = \{[s_i, p_{j,k}] \mid [s_i, p_{j,k}] \in E \text{ and } p_{j,k} \in V_2'\} \end{cases}$$

Given the subject $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ and a fixed j, S_j is a solution of BG_j if:

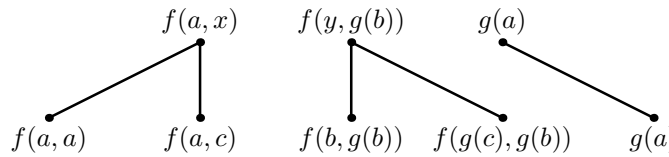
$$\begin{cases} S_j \subseteq E' \text{ and } \forall k \in \{1, \ldots, k_j\}, \exists \text{ a unique } [s_i, p_{j,k}] \in S_j \\ card(\{[s_i, p_{j,k}] \in S_j \mid 1 \leq i \leq p\}) \leq \alpha_i \end{cases}$$

This solution corresponds to a maximum bipartite matching for BG_j. If S_j does not exist, the next bipartite graph BG_{j+1} (associated to p_{j+1}) has to be extracted. The main advantage of this approach is that common syntactic subterms are matched only once, even if they appear in several rules (the information is saved once in the presented compact bipartite graph). Moreover, extraction can be performed efficiently with an adapted data structure: in our implementation, the compact bipartite graph is represented by a set of bit vectors. For each subterm $p_{j,k}$, a bit vector is associated, the i^{th} bit is set to 1 if $p_{j,k}$ matches to s_i. Encoding compact bipartite graphs by a list of bit vectors has two main advantages: the memory usage is low and the bipartite graph extraction operation is extremely cheap, since only bit vectors are selected.

Example. Considering our running example, an analysis of subterms with the same AC top function symbol f_{AC} gives three distinct non-variable subterms up to variable renaming: $p_{1,1} = f(a, x)$ and $p_{1,2} = g(a)$ for $f_{AC}(z, f(a, x), g(a)) \rightarrow r_1$ **if** $z = x$, and $p_{2,1} = f(a, x)$, $p_{2,2} = f(y, g(b))$ for $f_{AC}(f(a, x), f(y, g(b))) \rightarrow r_2$. Variable subterms ($z$ in this example) are not involved in the compact bipartite graph construction. They are instantiated later in the substitution construction phase described in Section 4.1. Let us consider the subject:

$$s = f_{AC}(f(a, a), f(a, c), f(b, g(b)), f(g(c), g(b)), g(a)).$$

After trying to match all subterms $f(a, a)$, $f(a, c)$, $f(b, g(b))$, $f(g(c), g(b))$ and $g(a)$ by $f(a, x), f(y, g(b))$ and $g(a)$, the following compact bipartite graph is built:



The compact bipartite graph is exploited as follows. In order to normalise the subject, a rule has to be selected, for instance $f_{AC}(f(a, x), f(y, g(b))) \rightarrow r_2$. The bipartite graph that should have been created by the general method can be easily constructed by extracting edges that join $f(a, x)$ and $f(y, g(b))$ to subject subterms, which gives the "classical" bipartite graph presented in the right part

of Figure 2. To check if the selected rule can be applied, a maximum bipartite matching has to be found. The bipartite graph has four solutions:

$$[f(a,a), f(a,x)], [f(b,g(b)), f(y,g(b))] \quad [f(a,c), f(a,x)], [f(b,g(b)), f(y,g(b))]$$
$$[f(a,a), f(a,x)], [f(g(c),g(b)), f(y,g(b))] \quad [f(a,c), f(a,x)], [f(g(c),g(b)), f(y,g(b))]$$

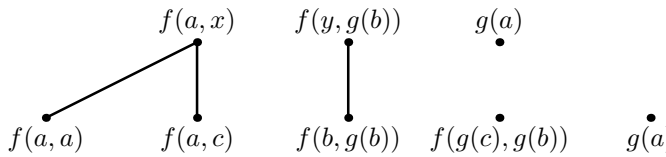
The given example of compact bipartite graph is represented by only three bit vectors: `11000`, `00110` and `00001`. The first one: `11000`, means that the corresponding pattern $f(a,x)$ matches the two first subterms: $f(a,a)$ and $f(a,c)$, and similarly for the other ones. Extracting the bipartite graph is done by only selecting bit vectors associated to $f(a,x)$ and $f(y,g(b))$: `11000` and `00110`.

Eager matching. An AC matching problem usually has more than one solution. But for applying a rule without conditions or whose conditions do not depend on a variable that occurs under an AC function symbol of the left-hand side, there is no need to compute a set of solutions: the first found match is used to apply the corresponding rule. Those rules are called *eager rules*. This remark leads us to further specialise our algorithm to get an *eager matching* algorithm, which tries to find a match for eager rules, as fast as possible. The idea consists of incrementally building the whole compact bipartite graph and adding to each step a test to check whether a bipartite graph associated to an eager rule has a solution. This test is not performed for non-eager rules and no check is necessary on a bipartite graph if no modification occurred since the last applied satisfiability test (i.e. no edge has been added). Using those two remarks, the number of checked bipartite graphs is considerably reduced.

Let us consider our running example. With the first method presented above (called the main algorithm), five matching attempts were done to completely build the compact bipartite graph (corresponding to its five edges). Only after this building phase, bipartite graphs are extracted and solved. Assuming that subterms are matched from left to right, it is sufficient to match only three subterms (with the eager algorithm), to find the first suitable solution:

$$S_2 = \{[f(a,a), f(a,x)], [f(b,g(b)), f(y,g(b))]\}.$$

This solution is found as soon as the following partial compact bipartite graph is built:



In practice, eager matching considerably reduces the number of matching attempts and there is only a small time overhead, due to the test, when no eager rule is applied. In the main algorithm, the number of matching attempts is linear in the number of subterms of the subject. In the eager algorithm, this number also depends on the pattern structure. Using two examples (Prop and

Bool3) described in section 5, some experimental results that compare the number of matched subterms with and without the eager algorithm are given in the following table:

	Main algorithm	Eager algorithm	Gain
Prop	50,108	32,599	17,509 (−35%)
Bool3	44,861	8,050	36,811 (−82%)

Note however that eager matching is not compatible with rewriting with priority, since the eager rule chosen by the eager matching algorithm may not correspond to the first applicable rule in the set of rules ordered by the programmer.

3.4 Handling Other Patterns

In order to handle rules whose patterns are not in C_2, a program transformation is applied. It transforms those rules into equivalent ones whose left-hand sides are in the class C_2. In order to preserve the semantics, the definition of conditional rewrite rules has to be enriched with a notion of local assignment. Such a construction is already allowed in the syntax of ELAN, and a similar one is called matching condition in ASF+SDF. Our compiler can handle rules of the following form: $l \to r$ **if** v **where** $p := u$, where p is a term possibly reduced to a single variable. Let σ be the matching substitution from l to the given subject. If p matches $u\sigma$, its variables are instantiated and this extends the main matching substitution σ. If the pattern p contains AC function symbols, an AC matching procedure is called.

The transformation of a rule with a general pattern into a rule with local assignments and satisfying our pattern restrictions is done according to two cases for the pattern l: either the top symbol of l is an AC symbol or not.

- Let $l = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_k, t_{k+1}, \ldots, t_n)$ with $x_1, \ldots, x_m \in \mathcal{X}$, $t_1, \ldots, t_k \in C_1$ and $t_{k+1}, \ldots, t_n \notin C_1$, where $k < n$.
  If $l' = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_k, y)$. The rule

  $$l' \to r \textbf{ where } f_{AC}(t_{k+1}, \ldots, t_n) := y$$

  is equivalent to the previous one. Recall that when computing the local assignment, y is instantiated by the matching substitution from l' to the ground subject.
- Let $l = f(t_1, \ldots, t_n)$ with some $t_i \notin C_2$. Let Λ be an abstraction function that replaces each maximal non-variable subterm of l which is not in C_2, say u_j, by a new variable x_j, for $j = 1..k$, where k is the number of such subterms. Let $l' = \Lambda(f(t_1, \ldots, t_n))$. The rule

  $$\begin{aligned} l' \to r \textbf{ where } & u_1 := x_1 \\ & \vdots \\ \textbf{where } & u_k := x_k \end{aligned}$$

  is equivalent to $l \to r$.

It is worth noticing that when computing such local assignments, only one-to-one matching problems occur. If the pattern p contains AC function symbols, a general one-to-one AC matching procedure [9] can be called. In the worst case, many-to-one AC matching is not used and the program transformation builds a rewrite rule system where AC problems are solved with a one-to-one AC matching procedure in the **where** parts, helped by a full indexing for the topmost free function symbol layer. We get back a frequently implemented matching technique, used in Maude [7] for instance.

4 Compiling the Normalisation Process

It is essential to have a good AC matching algorithm to get an efficient AC rewriting engine. But, matching is not the only operation in a normalisation process. Computing a normal form of a term consists of repeatedly selecting a rule that matches the subject, computing the current substitution and applying it to the right-hand side of the selected rule to build the reduced term. A global consideration of the whole process is crucial: all data structures and sub-algorithms have to be well-designed to cooperate without introducing any bottleneck. In this section we complete the description of the general architecture of our compiler by giving details on how instances of variables are computed and how the reduced term is built.

In the construction on the reduced term, it is usually possible to reuse parts of the left-hand side to construct the instantiated right-hand side. At least, instances of variables that occur in the left-hand side can be reused. More details can be found in [28] for the syntactic case. Similar techniques have been developed in our compiler, but we do not discuss them here, but rather focus on the construction of substitutions.

4.1 Construction of Substitutions

For constructing the reduced term, the substitution which encodes a solution to the matching problem has to be built. At this stage, two problems can be addressed: how to instantiate the remaining variables in AC patterns? How to optimise the substitution construction?

Variable instantiation. Variables that occur in patterns just below an AC function symbol are not handled in the previously described phases of the compiler. This problem is delayed until the construction of substitutions. When only one or two distinct variables (with multiplicity) appear directly under each AC function symbol, their instantiation does not need to construct a Diophantine equational system. Several cases can be distinguished according to the syntactic form of patterns.

– For $f_{AC}(x_1, t_1, \ldots, t_n)$, once $t_1, \ldots, t_n$ are matched, all the unmatched subject subterms are captured by x_1.

- For $f_{AC}(x_1^{\alpha_1}, x_2, t_1, \ldots, t_n)$, let us first consider the case where $\alpha_1 = 1$. Then once $t_1, \ldots, t_n$ are matched, the remaining subject subterms are partitioned into two non-empty classes in all possible ways. One class is used to build the instance of x_1, the other for x_2.
  When $\alpha_1 > 1$, once $t_1, \ldots, t_n$ are matched, one tries to find in all possible ways α_1 identical remaining subjects to match x_1 and then, all the remaining unmatched subject subterms are captured by x_2.
- For $f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$, once $t_1, \ldots, t_n$ are matched, a system of Diophantine equations is solved for computing instances of $x_1, \ldots, x_m$.

Considering our running example, and the rule $f_{AC}(z, f(a, x), g(a)) \to r_1$ **if** $z = x$. Once matching has been performed, and the two solutions for x (a and c) have been found, the variable z can be instantiated respectively by $f_{AC}(f(a, c), f(b, g(b)), f(g(c), g(b)))$ or $f_{AC}(f(a, a), f(b, g(b)), f(g(c), g(b)))$. The condition $z = x$ is never satisfied with those substitutions, so application of this rule fails.

Compiling the substitution construction In the syntactic case, the matching substitution is easily performed by the discrimination net, since there is at most one solution. In the AC case, there may be many different instantiations for each variable. It would be too costly to record them in a data structure for possible backtracking. Furthermore, the construction of this dynamic data structure is not necessary when the first selected rule is applied, because all computed substitutions are deleted. Our approach consists of computing the substitution only when a solution of the bipartite graph is found. For each subterm $p_{j,k}$, variable positions are known at compile time and used to construct an access function $access\_p_{j,k}$ from terms to lists of terms. This function takes a ground term as argument and returns the list of instances of variables of $p_{j,k}$ as a result. Given $S_j = \{[s_i, p_{j,k}]\}$ a solution of BG_j, the set $I_j = \{access\_p_{j,k}(s_i) \mid [s_i, p_{j,k}] \in S_j\}$ of variable instantiations can be computed.

Given the rule $f_{AC}(f(a, x), f(y, g(b))) \to r_2$, the functions $access\_f(a, x)(t) = t_{|2}$ and $access\_f(y, g(b))(t) = t_{|1}$ are defined. Starting from $S_2 = \{[f(a,a), f(a,x)], [f(b, g(b)), f(y, g(b))]\}$, the set $I_2 = \{a, b\}$ is easily computed, and we get the substitution $\sigma = \{x \mapsto a, y \mapsto b\}$.

4.2 Optimisations

Normalised substitutions. In the case of the leftmost-innermost reduction strategy, all variables instantiated by syntactic matching are irreducible by construction. Nested function calls are such that before a matching phase, each subterm is in normal form w.r.t. the rewrite rule system. This is no longer the case in AC rewriting: for instance, in our running example, the variable z can be instantiated by $f_{AC}(f(a, c), f(b, g(b)), f(g(c), g(b)))$ which is reducible by the rule $f_{AC}(f(a, x), f(y, g(b))) \to r_2$. To ensure that instances of variables that appear immediately under an AC top function symbol are irreducible, they are normalised before using them to build the right-hand side. Moreover, if the considered rule has a non-linear right-hand side, this normalisation substitution step

allows us to reduce the number of further rewrite steps: the irreducible form is computed only once. Without this optimisation, normalisation of identical terms frequently occurs even if a shared data structure is used because flattening can create multiple copies. As illustrated in section 5.2, in practice, the number of applied rules is significantly reduced.

Maintaining canonical forms. The compact bipartite graph construction (and thus the matching phase) supposes that both pattern and subject are in canonical form. Instead of re-computing the canonical form after each right-hand side construction, one can maintain this canonical form during the reduced term construction. Whenever a new term t is added as a subterm of $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$, if an equivalent subterm s_i already exists, its multiplicity is incremented, else, the subterm t (which is in normal form by construction) is inserted in the list $s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}$ at a position compatible with the chosen ordering. If t has the same AC top symbol f_{AC}, a flattening step is done and the two subterm lists are merged with a merge sort algorithm. A detailed algorithm is given in [23].

Term structure and sharing. A specific data structure for terms has been adopted, assuming that terms are maintained in some canonical form. In implementations of algebraic programming language, the representation of first-order terms is generally based on trees. An alternative representation of terms which is linear rather than tree-like has been proposed by Jim Christian [6]. Those *flatterms*, represented by a doubly-linked list data structure, yield in practice simpler and faster traversal algorithms than the conventional tree representation. But in the flatterm representation, subterms stored in the doubly-linked list cannot be shared. In our compiler, two different tree based representations are used. Subterms of free function symbols are stored in an array, while subterms of AC function symbols are stored in a simply-linked-list. This data structure has been easily extended to handle multiplicities and represent terms in canonical form.

4.3 Summary

To give a better understanding of the approach and an overview of the compiler, we summarise in Figure 3 which data structures and operations are used during the compilation process and during the execution time.

5 Experimental Results

Our own interest in compilation of AC normalisation comes from the development of the ELAN system [19]. ELAN is an environment for prototyping and combining different deduction systems described using rewrite rules and user-defined strategies for controlling rewriting [3,4]. A first ELAN compiler was designed and presented in [28], but did not handle AC function symbols. However, remarkable

	Compile Time	**Run Time**
Data Structure	rewrite rules discrimination tree/automata position of variables	term compact bipartite graph bit vector substitution
Operations	linearisation of patterns transformation of rules compilation of discrimination trees pre-construction of CBG compilation of variable access functions	canonical form maintenance syntactic matching CBG construction BG solving variable instantiation sharing creation and detection

Fig. 3. Summary of defined data structures and operations

performances obtained by this compiler were a strong encouragement to design an extension to handle AC theories.

The version of the ELAN compiler used in the experiments below does not handle ELAN's strategies. This ELAN compiler is implemented with approximatively 10,000 lines of Java. A runtime library has been implemented in C to handle basic terms and AC matching operations. This library contains more than 7,000 lines of code. To illustrate the power of our compilation techniques, let us consider examples that make a heavy use of AC normalisation.

5.1 Examples of Specifications

▶ The Prop example implements the two basic AC operators *and*, *xor*, and four syntactic rules that transform *not*, *implies*, *or* and *iff* functions into nested calls of *xor* and *and*. The rewrite system is defined by the following rules:

$and(x, \top)$	$\rightarrow x$	
$and(x, \bot)$	$\rightarrow \bot$	$xor(x, \bot) \rightarrow x$
$and(x, x)$	$\rightarrow x$	$xor(x, x) \rightarrow \bot$
$and(x, xor(y, z))$	$\rightarrow xor(and(x, y), and(x, z))$	
$implies(x, y)$	$\rightarrow not(xor(x, and(x, y)))$	
$not(x)$	$\rightarrow xor(x, \top)$	
$or(x, y)$	$\rightarrow xor(and(x, y), xor(x, y))$	
$iff(x, y)$	$\rightarrow not(xor(x, y))$	

The benchmark consists of normalising the following term:

```
implies(and(iff(iff(or(a1,a2),or(not(a3),iff(xor(a4,a5),not(not(
not(a6))))))),not(and(and(a7,a8),not(xor(xor(or(a9,and(a10,a11)),a2),
and(and(a11,xor(a2,iff(a5,a5))),xor(xor(a7,a7),iff(a9,a4)))))))),
implies(iff(iff(or(a1,a2),or(not(a3),iff(xor(a4,a5),not(not(not(a6))))))),
not(and(and(a7,a8),not(xor(xor(or(a9,and(a10,a11)),a2),and(and(a11,
xor(a2,iff(a5,a5))),xor(xor(a7,a7),iff(a9,a4)))))))),not(and(implies(and(
a1,a2),not(xor(or(or(xor(implies(and(a3,a4),implies(a5,a6)),or(a7,a8)),
```

```
xor(iff(a9,a10),a11)),xor(xor(a2,a2),a7)),iff(or(a4,a9),xor(not(a6),
a6))))),not(iff(not(a11),not(a9)))))))),not(and(implies(and(a1,a2),
not(xor(or(or(xor(implies(and(a3,a4),implies(a5,a6)),or(a7,a8)),
xor(iff(a9,a10),a11)),xor(xor(a2,a2),a7)),iff(or(a4,a9),xor(not(a6),
a6))))),not(iff(not(a11),not(a9))))))
```

The normalisation of this term quickly produces a very large term. The expected result of the evaluation is $\top$.

► The Bool3 example (designed by Steven Eker) implements computation in a 3-valued logic. The rewrite system is defined by the following rules, where $+$ and $*$ are AC.

$x + 0$	$\to x$	$x * 0$	$\to 0$
$x + x + x$	$\to 0$	$x * x * x$	$\to x$
$(x + y) * z$	$\to (x * z) + (y * z)$	$x * 1$	$\to x$
$and(x, y)$	$\to (x * x * y * y) + (2 * x * x * y)+$ $(2 * x * y * y) + (2 * x * y)$		
$or(x, y)$	$\to (2 * x * x * y * y) + (x * x * y)+$ $(x * y * y) + (x * y) + (x + y)$		
$not(x)$	$\to (2 * x) + 1$		
2	$\to 1 + 1$		

The benchmark consists in normalising the two following terms, and compare their normal forms:

$$and(and(and(a_1, a_2), and(a_3, a_4)), and(a_5, a_6))$$

and

$$not(or(or(or(not(a_1), not(a_2)), or(not(a_3), not(a_4))), or(not(a_5), not(a_6))))$$

► In [8], a rewrite system modulo AC for natural arithmetic was presented: Nat10. This system contains 56 rules rooted by the AC symbol $+$, 11 rules rooted by the AC symbol $*$, and 82 syntactic rules. The authors conjecture in their paper that compilation techniques and many-to-one matching should improve their implementation. We used this rewrite system to compute the 16^{th} Fibonacci number.

5.2 Benchmarks

The first two benchmarks (Prop and Bool3) seem to be trivial because they contain a small number of rules. But after several rewrite steps, the term to be reduced becomes very large (several MBytes) and contains a lot of AC symbols. It is not surprising to see a system spending several hours (on a fast machine) before finding the result or running out of memory.

The execution of the Nat10[1] example does not generate such large terms, but the rewrite system contains a lot a rules. This illustrates the usefulness of many-to-one matching techniques.

[1] This last example was originally implemented in C*i*ME which is a theorem prover rather than a programming environment. To compute $Fib(16)$, C*i*ME applies 10,599 rules in 16,400 seconds.

Those three examples have been tested with Brute[2], Maude [7], OBJ [12], OTTER [22], RRL [18], ReDuX [5] and the new ELAN compiler on a Sun Ultra-Sparc 1 (Solaris). Unfortunately, we did not find any other compiler for normalisation with AC rewrite systems. It may be questionable to compare a compiled system with an interpreted systems such as OBJ. However, there is no evidence that a compiled system should be significantly faster than a good interpreter when a lot of AC symbols are involved: compared to the syntactic case, few precomputed data structures and functions can be generated and the main costly operations involve many dynamic data structures and general functions from the runtime library. In this sense, even with a compiler, AC matching operations are partially interpreted.

The number of applied rewrite rules (rwr) and the time spent in seconds (sec) are given in the following array:

	Prop		Bool3		Nat10	
	rwr	sec	rwr	sec	rwr	sec
OBJ	12,837	1,164	-	> 24h	26,936	111
OTTER	167,621	6.17	-	> 10min[3]	-	-
ReDuX	18,292	180	268,658	1,200	-	-
RRL	-	> 24h	-	> 4h [3]	-	-
Brute	23,284	1.78	34,407	2.25	-	-
Maude	12,281	0.47	4,854	0.25	25,314	0.32
ELAN compiler	12,689	0.43	5,282	0.18	15,384	0.15

The following statistics give an overview of the time spent in specialised AC matching operations compared to the total execution time (the total is not equal to 100% because many other functions are involved in the normalisation process):

	Prop	Bool3	Nat10
CBG build	12.76%	14.59%	7.39%
BG extraction	0.3%	0.39%	4.31%
BG solve	3%	3.19%	9.38%
substitution build	3.74%	3.77%	4.52%
canonical form maintenance	24.1%	23.9%	1.7%

On the three presented examples, less than 21% of the total execution time is spent in building and solving the bipartite graph hierarchy. When the number of AC rules increases (Nat10), the time spent on extracting and solving bipartite graphs slightly increases. The CBG construction is cheaper in Nat10 because the size of the involved subject is smaller: the number of matching attempts is reduced. As expected, the time spent on building substitutions does not significantly depend on the example nor from the number of solutions of

[2] available at `ftp://ftp.sra.co.jb/pub/lang/CafeOBH/brute-X.Y.tar.gz`

[3] More than 70 MBytes and 115MBytes respectively were used.

the solved AC matching problem. This clearly shows the interest of our compiled substitution construction approach. To conclude, even with a well-suited term data structure and an optimised canonical form maintenance algorithm, the time spent in maintaining a term in canonical form can be really important on examples in which very large terms are involved.

What is really interesting in the compiled approach is that we can combine interesting performances of the proposed AC rewriting process with extremely good results brought by the syntactic compiler: up to 10,000,000 rewrite steps per second can be performed on simple specifications, and an average of 500,000 rwr/sec when dealing with large specifications that involve complex non-deterministic strategies. The best interpreters can perform up to 400,000 rwr/sec in the syntactic case. Compared with the compiled approach, this is a serious bottleneck when specifying a constraint solver or a theorem prover, for example.

On the two first examples, Maude gives very interesting results. The third example tends to show that in a more realistic situation (with many rewrite rules) the many-to-one approach may be an advantage to efficiently execute large specifications.

6 Conclusion

Experimental results show that the combination of techniques presented in this paper leads to a significant improvement of performance of AC normalisation. Another characteristic of the implementation, not yet mentioned, is to provide modular compilation, in the sense that each function or strategy corresponds to one compiled code module. The idea is that when a function or a strategy is modified, only the corresponding module is recompiled. This is extremely important for compilation of large programs.

The presented compilation techniques are not exclusively designed for the ELAN language. In a recent project in cooperation with the ASF+SDF group, we have designed a translator from a sub-language of ASF+SDF (namely μASF) to the intermediate code used by our compiler. This experiment has clearly shown that our compilation techniques can be used in a more general context than ELAN itself.

To conclude this paper, let us mention further work.

► **User-defined strategies.** The strategy language of ELAN provides predefined constructors for strategies, and also gives the possibility to the user to define his own strategies in a very flexible way using the same paradigm of rewriting [3,4]. Compilation techniques for this powerful strategy language are under development.

► **List matching.** The expressivity offered by associative operators is extremely interesting when designing large specifications. The recent experiment in compiling ASF+SDF specifications reveals the need to have an efficient matching algorithm for such list operations. We are working on a specialised version of the presented AC matching algorithm, using the same compact data structure,

in order to handle some new classes of patterns that contain associative function symbols.

▶ **Combination of theories.** AC theories are the most frequent ones in mathematical and algebraic structures, but a programmer may be interested in mixing in his specifications AC function symbols with others that may be only associative, or commutative, or idempotent or with a unit. Although theoretical problems related to combination of matching algorithms have already been explored [25,27], providing an efficient matching algorithm for the combination of such symbols is a challenging open problem first attacked in [10].

Acknowledgements

We sincerely thank Peter Borovanský, Claude Kirchner, Christophe Ringeissen, Michaël Rusinowitch and Laurent Vigneron for helpful discussions and comments.

References

1. L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 61–74, Orsay, France, Apr. 13–17, 1993. Springer-Verlag. 232, 233
2. D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, Apr. 1987. 232
3. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), Sept. 1996. 242, 246
4. P. Borovanský, C. Kirchner, and H. Kirchner. Rewriting as a unified specification tool for logic and control:the elan language. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, Sept. 1997. Springer-Verlag. 242, 246
5. R. Bündgen. Reduce the redex ← ReDuX. In C. Kirchner, editor, *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, LNCS 690, pages 446–450, Montreal, Canada, June 16–18, 1993. Springer-Verlag. 245
6. J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, Feb. 1993. 232, 242
7. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), Sept. 1996. Electronic Notes in Theoretical Computer Science. 240, 245
8. E. Contejean, C. Marché, and L. Rabehasaina. Rewrite systems for natural, integral, and rational arithmetic. In H. Comon, editor, *Proceedings of 8-th International Conference Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 98–112, Sitges, Spain, 1997. Springer-Verlag. 244

9. S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995. 231, 232, 240
10. S. Eker. Fast matching in combinations of regular equational theories. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), Sept. 1996. Electronic Notes in Theoretical Computer Science. 247
11. K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. Technical Report B-225, Department of Information Sciences, Tokyo Institute of Technology, Oh-okayama, Meguro-ku, Tokyo 152, Japan, 1989. 236
12. J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, Aug. 1988. 245
13. A. Gräf. Left-to-rigth tree pattern matching. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 323–334. Springer-Verlag, Apr. 1991. 232, 233
14. P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996. 233
15. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973. 236
16. J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980. 232
17. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984. 231, 235
18. D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In *Proceedings 9th International Conference on Automated Deduction, Argonne (Ill., USA)*, volume 310 of *Lecture Notes in Computer Science*, pages 768–769. Springer-Verlag, 1988. 245
19. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995. 242
20. E. Kounalis and D. Lugiez. Compilation of pattern matching with associative commutative functions. In *16th Colloquium on Trees in Algebra and Programming*, volume 493 of *Lecture Notes in Computer Science*, pages 57–73. Springer-Verlag, 1991. 232
21. D. Lugiez and J.-L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994. 232
22. W. W. McCune. Otter 3.0: Reference manual and guide. Technical Report 6, Argonne National Laboratory, 1994. 245
23. P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In A. Sellink, editor, *Second International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing, eWiC web site: http://ewic.springer.co.uk/, Amsterdam, Sept. 1997. Springer-Verlag. 12 pages. 242

24. N. Nedjah, C. Walter, and E. Eldrige. Optimal left-to-right pattern-matching automata. In M. Hanus, J. Heering, and K. Meinke, editors, *Proceedings 6th International Conference on Algebraic and Logic Programming, Southampton (UK)*, volume 1298 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, Sept. 1997. 232
25. T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, pages 633–653, 1991. 247
26. G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981. 231, 235
27. C. Ringeissen. Combining decision algorithms for matching in the union of disjoint equational theories. *Information and Computation*, 126(2):144–160, May 1996. 247
28. M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag. 240, 242

Solution to the Problem of Zantema on a Persistent Property of Term Rewriting Systems

Takahito Aoto

Department of Computer Science, Faculty of Engineering, Gunma University
Tenjincho 1-5-1, Kiryuu, Gunma, 376-8515, Japan
aoto@cs.gunma-u.ac.jp

Abstract. A property P of term rewriting systems is persistent if for any many-sorted term rewriting system R, R has the property P iff its underlying term rewriting system $\Theta(R)$, which results from R by omitting its sort information, has the property P. It is shown that termination is a persistent property of many-sorted term rewriting systems that contain only variables of the same sort. This is the positive solution to a problem of Zantema, which has been appeared as Rewriting Open Problem 60 in literature.

1 Introduction

Sort introduction technique in term rewriting has caught attention recently [2,8]. We prove in this paper a conjecture which opens up a possibility of new applications of this technique. The conjecture reads: for any terminating many-sorted term rewriting system $\mathcal{R}$, if $\mathcal{R}$ contains only variables of the same sort then $\Theta(\mathcal{R})$ is also terminating. Here, $\Theta(\mathcal{R})$—called the underlying term rewriting system of $\mathcal{R}$—is the term rewriting system obtained from $\mathcal{R}$ by omitting its sort information. The conjecture was raised by Zantema and adapted in [4] as Rewriting Open Problem 60.

A property $\mathcal{P}$ of TRSs is said to be *persistent* if $\mathcal{R}$ has the property $\mathcal{P}$ iff $\Theta(\mathcal{R})$ has the property $\mathcal{P}$; this notion is due to Zantema [10]. We say $\mathcal{P}$ *is persistent for a class* $\mathcal{C}$ *of many-sorted TRSs* if for any $\mathcal{R} \in \mathcal{C}$, $\mathcal{R}$ has the property $\mathcal{P}$ iff $\Theta(\mathcal{R})$ has the property $\mathcal{P}$. Thus the conjecture above equals persistency of termination for the class of many-sorted TRSs that contain only variables of the same sort. It is known that confluence is persistent for the class of many-sorted TRSs [3] and that termination is persistent for the class of many-sorted TRSs that do not contain both duplicating and collapsing rules [10]. Persistent properties in equational rewriting have been studied in [9].

A property ϕ of TRSs is said to be *modular for the direct sum* if $\phi(\mathcal{R}_1)$ and $\phi(\mathcal{R}_2)$ imply $\phi(\mathcal{R}_1 \cup \mathcal{R}_2)$ for any two TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ sharing no function symbols. For component closed properties ϕ, persistency of ϕ for the class of many-sorted TRSs implies modularity of ϕ for the direct sum of TRSs [10]. Similarly, persistency of property ϕ for a subclass $\mathcal{C}$ of many-sorted TRSs often implies modularity of ϕ for the direct sum of TRSs from the class $\{\Theta(\mathcal{R}) \mid \mathcal{R} \in$

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 250–265, 1998.

$\mathcal{C}\}$. Indeed, if we assign sorts $1 \times \cdots \times 1 \to 1$ to function symbols in $\mathcal{R}_1$ and $2 \times \cdots \times 2 \to 2$ to those in $\mathcal{R}_2$, modularity of ϕ for the direct sum of $\mathcal{R}_1$ and $\mathcal{R}_2$ is a particular case of persistency of ϕ provided that $\mathcal{R}_1 \cup \mathcal{R}_2$ under this sort assignment is contained in $\mathcal{C}$. Thus, for example, persistency of termination for the class of many-sorted TRSs that do not contain both duplicating and collapsing rules implies the corresponding modularity result for the direct sum of TRSs. To the contrary, our requirement on the sorts of variables does not carry over for TRSs and therefore it is hard to formalize the corresponding modularity result. This contrasts sharply with other persistency results obtained so far.

Our result is useful to show that termination is preserved under suitable translations of TRSs. More precisely, it follows that coding a function symbol $f(,\ldots,)$ by a suitable term $t(,\ldots,)$ does not affect termination behavior of TRSs.

The rest of this paper is organized as follows. In Section 2, we fix notations on many-sorted term rewriting and its underlying term rewriting. Section 3 is devoted to the proof of the Zantema's conjecture. In Section 4, applications of our theorem and related works are discussed. We conclude our result in Section 5.

2 Preliminaries

We assume familiarity with basic notions in term rewriting. In what follows, we recall some less common definitions and fix notations used in this paper.

Let $\mathcal{S}$ be a set of sorts (denoted by $\alpha, \beta, \gamma, \ldots$), $\mathcal{F}$ a set of $\mathcal{S}$-sorted function symbols (denoted by $f, g, h, \ldots$), $\mathcal{V}$ a set of $\mathcal{S}$-sorted variables (denoted by $x, y, z, \ldots$). We write $f : \alpha_1 \times \cdots \times \alpha_n \to \beta$ when $f \in \mathcal{F}$ has sort $\alpha_1 \times \cdots \times \alpha_n \to \beta$. We assume that there are countably infinite variables of sort α for each sort $\alpha \in \mathcal{S}$.

We denote by $\mathcal{T}$ (and $\mathcal{T}^\alpha$) the set of terms (of sort α, respectively). For a term t, $\mathcal{V}(t)$ is the set of variables that appear in t. Syntactical equality is denoted by $\equiv$. We write $t \trianglelefteq s$ ($t \triangleleft s$) when t is a *(proper) subterm of* s, or equivalently s *(properly) contains* t.

For each sort α, the *hole* of sort α is written as $\Box^\alpha$. A *context* is a term possibly containing holes. We denote by $\mathcal{C}$ the set of contexts. We write $C : \alpha_1 \times \cdots \times \alpha_n \to \beta$ when $C \in \mathcal{C}$ has sort β (as a term) and has n holes $\Box^{\alpha_1}, \ldots, \Box^{\alpha_n}$ from left to right in it. If $C : \alpha_1 \times \cdots \times \alpha_n \to \beta$ and $t_1 \in \mathcal{T}^{\alpha_1}, \ldots, t_n \in \mathcal{T}^{\alpha_n}$ then $C[t_1, \ldots, t_n]$ is the term obtained from C by replacing holes with $t_1, \ldots, t_n$ from left to right. A context C is written as $C[\,]$ when C contains precisely one hole. A context is said to be *empty* when $C \equiv \Box^\alpha$ for some $\alpha \in \mathcal{S}$.

We denote by $\mathrm{Pos}(t)$ the set of *positions* of a term t; by t/p the subterm of t at a position $p \in \mathrm{Pos}(t)$. $\mathrm{Pos}_{\mathcal{V}}(t)$ stands for $\{p \in \mathrm{Pos}(t) \mid t/p \in \mathcal{V}\}$. The *empty* (or *root*) position is denoted by Λ. For $u, v \in \mathrm{Pos}(t)$, we write $u \leq v$ when u is a prefix of v and $u \mid v$ when u and v are incomparable. For a context $C[\,]$, we write $C[\,]_p$ when $C[\,]/p$ is a hole. When $s \equiv C[u]_p$, we denote by

$s[p \leftarrow v]$ the term $C[v]$. When $p \notin \mathrm{Pos}(s)$, we define $s[p \leftarrow v] \equiv s$. We abbreviate $(\cdots(s[p_1 \leftarrow v_1])\cdots[p_n \leftarrow v_n])$ to $s[p_i \leftarrow v_i \mid 1 \leq i \leq n]$.

A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathcal{T}$ such that x and $\sigma(x)$ have the same sort. A substitution is extended to a homomorphism from $\mathcal{T}$ to $\mathcal{T}$ in the obvious way; $t\sigma$ stands for $\sigma(t)$ for a substitution σ and a term t.

A *(many-sorted) rewrite rule* is a pair $l \to r$ of terms such that (1) l and r have the same sort, (2) $l \notin \mathcal{V}$, (3) $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. A *many-sorted term rewriting system* (*STRS*, for short) is a set of rewrite rules. For an STRS $\mathcal{R}$, its reduction relation $s \to_{\mathcal{R}} t$ is defined as usual. Note that s and t have the same sort whenever $s \to_{\mathcal{R}} t$. The transitive-reflexive closure and the transitive closure of $\to_{\mathcal{R}}$ are denoted by $\overset{*}{\to}_{\mathcal{R}}$ and $\overset{+}{\to}_{\mathcal{R}}$, respectively.

When $\mathcal{S} = \{*\}$, an STRS is called a TRS. Given an arbitrary STRS $\mathcal{R}$, by identifying each sort with $*$, we obtain the TRS $\Theta(\mathcal{R})$—called the underlying TRS of $\mathcal{R}$. According to the definition of rewrite relation, $\Theta(\mathcal{R})$ acts on the set of terms possibly non-well-sorted with respect to the sorts of function symbols in $\mathcal{R}$. Fixing a STRS $\mathcal{R}$, we will denote by $\Theta(\mathcal{T})$ and $\Theta(\mathcal{C})$ the sets of terms and contexts possibly non-well-sorted with respect to the sorts of function symbols in $\mathcal{R}$ respectively. On the other hand, we preserve notation $C : \alpha_1 \times \cdots \times \alpha_n \to \beta$ only for well-sorted contexts; more precisely, we will write $C : \alpha_1 \times \cdots \times \alpha_n \to \beta$ only when C is well-sorted with respect to the sorts of function symbols in $\mathcal{R}$.

3 A Proof of the Conjecture

We prove in this paper the following theorem conjectured by Zantema.

Theorem 1. *Let $\mathcal{R}$ be an STRS that contains only variables of the same sort. Then $\mathcal{R}$ is terminating if and only if $\Theta(\mathcal{R})$ is terminating.*

For this purpose, *we assume in the sequel that there exists a special sort $0 \in \mathcal{S}$, and fix an STRS $\mathcal{R}$ such that every variable occurring in rules of $\mathcal{R}$ has sort* 0.

3.1 Sort introduction basics

Terms in $\Theta(\mathcal{T})$ can be partitioned into well-sorted components; and this partition yields a natural layered structure in each term. In this subsection, we present basic notion and notations related to this layered structure, and present a few key observations that follow from our assumption above.

From now on terms in $\Theta(\mathcal{T})$ are often referred to just terms.

Definition 1. *1. The* sort *of a term t is defined by*

$$\mathrm{sort}(t) = \begin{cases} \alpha & \text{if } t \equiv x^{\alpha}, \\ \beta & \text{if } t \equiv f(t_1, \ldots, t_n) \text{ with } f : \alpha_1 \times \cdots \times \alpha_n \to \beta. \end{cases}$$

We extends the definition of sort *for contexts in the obvious way by putting* $\mathrm{sort}(\Box^{\alpha}) = \alpha$.

2. *Let* $t \equiv C[t_1, \ldots, t_n]$ *(*$n \geq 0$*) be a term with non-empty* C*. We write* $t \equiv C[\![t_1, \ldots, t_n]\!]$ *if (1)* $C : \alpha_1 \times \cdots \times \alpha_n \to \beta$ *and (2)* $\mathrm{sort}(t_i) \neq \alpha_i$ *for* $i = 1, \ldots, n$*. We call the terms* $t_1, \ldots, t_n$ principal subterms *of* t*. We denote by* $\mathrm{cap}(t)$ *the well-sorted context* $C[\Box^{\alpha_1}, \ldots, \Box^{\alpha_n}]$*. Clearly, every term* t *is uniquely written as* $C[\![t_1, \ldots, t_n]\!]$ *by some* $C \in \mathcal{C}$ *and terms* $t_1, \ldots, t_n$*.*
3. *The* rank *of a term* t *is defined by:*

$$\mathrm{rank}(t) = \begin{cases} 1 & \text{if } t \in \mathcal{T}, \\ 1 + \max\{\mathrm{rank}(t_i) \mid 1 \leq i \leq n\} & \text{if } t \equiv C[\![t_1, \ldots, t_n]\!] \ (n > 0). \end{cases}$$

4. *A subterm* u *of* t *is said to be* special *(in* t*) if either* $u \equiv t$ *or* u *is a principal subterm of a special subterm of* t*.*

Convention 1. We denote $C[u]$ by $C[!u]$ if this occurrence of u is a proper special subterm of $C[u]$. We indicate $C[u_1, \ldots, u_n]$ as $C[u_1, \ldots, (u_i)_\gamma, \ldots, u_n]$ when $\Box^\gamma$ is not special in $C[u_1, \ldots, \Box^\gamma, \ldots, u_n]$. In the sequel, we assume every hole in non-empty $C \in \Theta(\mathcal{C})$ is not a special subterm of C; in this spirit, we often omit the superscript α of $\Box^\alpha$.

We next formalize the notion of peak and leaf, which are essential in our proof.

Definition 2. 1. *For a non-empty position* p *in a term* t*, the sort of the position* p *in* t *is defined as:* $\mathrm{sort}(t, p) = \gamma$ *when* $t \equiv C[(u)_\gamma]_p$ *for some* C *and* u*.*
2. *For a term* t*, we define the set of* disconnections in t *as follows:*

$$\mathrm{Dcn}(t) = \{p \in \mathrm{Pos}(t) \mid t/p \text{ is a special subterm of } t\}.$$

For a context C*,* $\mathrm{Dcn}(C)$ *is defined similarly. It is clear that for every non-empty position* p*, we have* $p \in \mathrm{Dcn}(t)$ *iff* $\mathrm{sort}(t, p) \neq \mathrm{sort}(t/p)$*.*
3. *Let* $t \equiv C[!u]_q$*. The* peak of u *is the position* p *defined by*

$$p = \max\{o \leq q \mid \mathrm{sort}(t, o) = 0 \text{ or } o \in \mathrm{Dcn}(t) \backslash \{q\}\}.$$

Clearly, C/p *is again a context; we call it the* leaf *of* C *and denote it by* $\mathrm{leaf}(C)$*.*

Let us illustrate our notation and concepts by an example.

Example 1. Let $\mathcal{F} = \{f : 0 \times 1 \to 0, g : 1 \to 1, h : 1 \to 0, a : 0, b : 0, c : 1\}$. Let $s \equiv f(h(f(h(a), g(b))), c)$. Let $C_1 \equiv f(h(\Box^1), c)$. Then we can write s as $C_1[(!f(h(a), g(b)))_1]$. We have $\mathrm{sort}(s, 1.1) = \mathrm{sort}(s, 1.1.2) = 1$ and $\mathrm{Dcn}(s) = \{\Lambda, 1.1, , 1.1.1.1, 1.1.2.1\}$. The peak of a in s is 1.1.1 and that of b in s is 1.1. Let $C_2 \equiv f(h(f(h(\Box^1), g(b))), c)$. Then $s \equiv C_2[!a]$ and $\mathrm{leaf}(C_2) \equiv h(\Box^1)$.

Definition 3. *A rewrite step* $s \to t$ *is said to be* inner *(written as* $s \to^i t$*) if*

$$s \equiv C[\![s_1, \ldots, C'[l\sigma], \ldots, s_n]\!] \to C[s_1, \ldots, C'[r\sigma], \ldots, s_n] \equiv t$$

for some terms $s_1, \ldots, s_n$*, a substitution* σ*,* $l \to r \in \mathcal{R}$*, and* $C' \in \Theta(\mathcal{C})$*; otherwise it is* outer *(written as* $s \to^o t$*). The redex of an inner reduction is a* inner redex*; that of an outer reduction is an* outer redex*.*

The next lemma is the first key observation arising from our assumption.

Lemma 1. *Let $l \to r \in \mathcal{R}$ and $l \equiv C_l[x_1, \ldots, x_k]_{o_1,\ldots,o_k}$ where C_l is a (non-empty) context containing no variables. Suppose $s \equiv C[l\sigma]_o$ and $l\sigma$ is an outer redex in s. Then,*

1. $\mathrm{sort}(s, o.o_i) = 0$ *for all* $i = 1, \ldots, k$;
2. *for the peak p of each principal subterm of s, we have either $o.o_i \leq p$ for some i, $p < o$ or $p \mid o$. (See Figure 1[1].)*

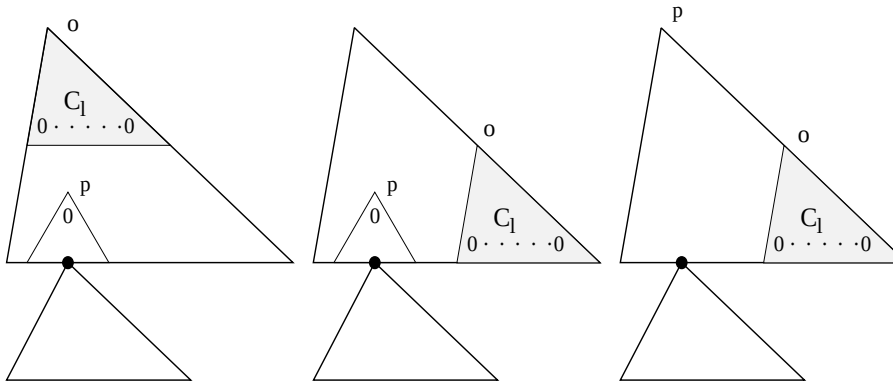


Fig. 1. The peak of a principal subterm and an outer redex

Definition 4. *A rewrite step $s \to^o t$ is said to be* destructive at level 1 *if* $\mathrm{sort}(s) \neq \mathrm{sort}(t)$*; and $s \to^i t$ is destructive at level $k+1$ if*

$$s \equiv C[\![s_1, \ldots, s_j, \ldots, s_n]\!] \to^i C[s_1, \ldots, t_j, \ldots, s_n] \equiv t$$

with $s_j \to t_j$ destructive at level k.

The following lemma appears often (see e.g. [2]).

Lemma 2. *A rewrite step $s \to^o t$ is destructive if and only if $t \equiv \sigma(x)$ and $s \equiv C[\![s_1, \ldots, \sigma(x), \ldots, s_n]\!]$ for some terms $s_1, \ldots, s_n$, substitution σ, and $C \in \mathcal{C}$ such that $C[s_1, \ldots, \Box, \ldots, s_n] \equiv C'\sigma$ for some $C'[x] \to x \in \mathcal{R}$.*

The second key observation derived from our assumption concerns with destructive rewrite steps.

Lemma 3. *Let $s \equiv C[\![s_1, \ldots, s_n]\!]_{q_1,\ldots,q_n}$, and suppose that $s \to s_i$ is a destructive rewrite step of level 1, and p_j is the peak of s_j in s for each $j = 1, \ldots, n$. Then,*

1. $\mathrm{sort}(s) = 0$, $\mathrm{sort}(s, q_i) = 0$ *and* $\mathrm{sort}(s_i) \neq 0$;
2. $p_i = q_i$ *and p_i is minimal in* $\{p_1, \ldots, p_n\}$.

[1] In this and the succeeding figures, • denotes a non-empty disconnection.

3.2 Active subterms and $\rightrightarrows$-reduction

A crucial step of our proof is to define a particular mapping from terms to terms that maps a $\Theta(\mathcal{R})$-reduction to another with some "good" property. This kind of technique is popular in modularity proofs (e.g. pile and delete technique [6], top-down labelling technique [2]). Our mapping results from a confluent and terminating reduction (called $\rightrightarrows$-reduction), which we introduce in this subsection.

We first introduce a notion of active subterms.

Definition 5. *1. Let $s \equiv C[\![s_1, \ldots, s_n]\!]$, and suppose that p_i is the peak of s_i for each $i = 1, \ldots, n$. The term s is said to be* root-active *if* $\mathrm{sort}(s) = 0$ *and s_i is root-active whenever $p_i = \Lambda$.*

2. A subterm of s is said to be active (in s) *if (1) it is a proper special subterm of s and (2) it is root-active.*

Definition 6. *Let $s \equiv C[(!u)_\gamma]_q$ and $u \equiv C'[\![u_1, \ldots, u_n]\!]_{q_1,\ldots,q_n}$. Suppose p is the peak of u in s and p_i is the peak of u_i in s for each $i = 1, \ldots, n$. Then, this occurrence of u is said to be* properly active (in s) *if (1)* $\mathrm{sort}(u) = 0$ *and (2) $q < p_i$ for all i $(1 \le i \le n)$. When u is properly active, u_i $(1 \le i \le n)$ is called an* activator of u (in s) *if (1) p_i is minimal in $\{p_1, \ldots, p_n\}$ and (2)* $\mathrm{sort}(u, q_i) = 0$*; if further it satisfies (3)* $\mathrm{sort}(u_i) = \gamma$*, then u_i is called a* connectable activator of u (in s). *Activators that are not connectable are* inconnectable activators. *(See Figure 2.)*

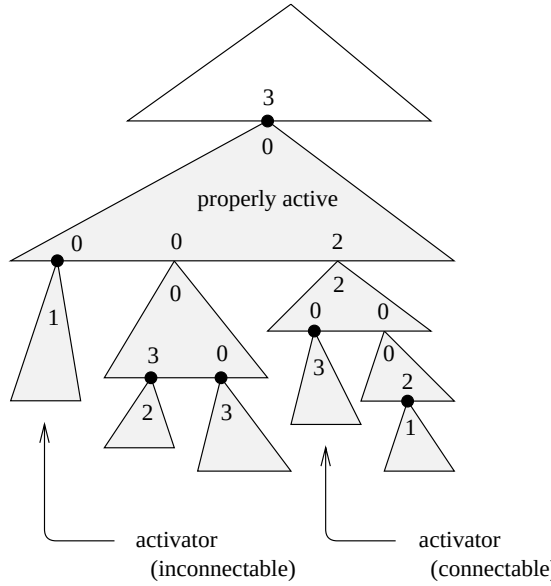


Fig. 2. A properly active subterm and activators

It is readily checked that if $s \equiv C[(!u)_\gamma]_q$, $u \equiv C'[\![u_1, \ldots, u_n]\!]_{q_1,\ldots,q_n}$, p is the peak of u in s and p_i is the peak of u_i in s for each $i = 1, \ldots, n$, then u is active iff (1) $\mathrm{sort}(u) = 0$ and (2) u_i is active in s whenever $q = p_i$. Thus, every properly active subterm is active.

Intuitively, properly active subterms u of a term s are special subterms whose top layer might collapse by a reduction; activators are principal subterms of u that might come up when the top layer of u collapses. Active subterms of a term are special subterms whose descendants might become properly active.

Note that if u is an active subterm of $s \equiv C[!u]_q$ then, since $\mathrm{sort}(u) = 0$ and $q \in \mathrm{Dcn}(s)$, we have $\mathrm{sort}(s, q) \neq 0$, and hence $\mathrm{leaf}(C)$ is non-empty.

Example 2. In Example 1, subterms of s that are root-active are s, $h(f(h(a), g(b)))$ $f(h(a), g(b))$, $h(a)$, a and b; active subterms in s are $f(h(a), g(b))$, a and b; properly active subterms in s are a and b. Let t be the term obtained by replacing b in s with $f(c, c)$, i.e. $t \equiv s[1.1.2.1 \leftarrow f(c, c)]$. Then $f(c, c)$ is a properly active subterm of t, which has a connectable activator c.

The next lemma is an easy consequence of the definition above, Lemma 1 and Lemma 3.

Lemma 4. *Let $s \to t$ a rewrite step destructive at level $k > 1$. Then there exists a subterm u of s properly active in s and an activator u_i of u in s and a context C such that $u \to u_i$, $s \equiv C[u]$ and $t \equiv C[u_i]$.*

Now we are in a position to define the $\rightrightarrows$-reduction.

Definition 7. *Let $s \equiv C[!u]$ and p the peak of u in s.*

1. *This occurrence of u in s is said to be a* $\rightrightarrows$-redex *if (1) u is properly active in s and (2) s/p is root-active.*
2. *The $\rightrightarrows$-redex u is a* left-most inner-most $\rightrightarrows$-redex *if (3) every proper special subterms of s/p with the peak different from p is not a $\rightrightarrows$-redex and (4) every principal subterms of* $\mathrm{leaf}(C)$ *placed on the left to the hole is not a $\rightrightarrows$-redex if its peak equals p.*
3. *Suppose that the $\rightrightarrows$-redex u is a left-most inner-most $\rightrightarrows$-redex. Let $\tilde{u} \equiv C'[\mathrm{leaf}(C)[u_1], \ldots, \mathrm{leaf}(C)[u_n]]$, where $u \equiv C'[u_1, \ldots, u_n]$ with all activators $u_1, \ldots, u_n$ of u displayed. Let $t \equiv s[p \leftarrow \tilde{u}]$. Then we say that s* $\rightrightarrows$-reduces *to t (symbolically $s \rightrightarrows t$) or t is a* $\rightrightarrows$-reduct *of s; we call $s \rightrightarrows t$ a $\rightrightarrows$-rewrite step. (See Figure 3.) The term u is called the* $\rightrightarrows$-redex *of this $\rightrightarrows$-rewrite step and we write $s \rightrightarrows_u t$ when this is the case. A* $\rightrightarrows$-normal *term is a term s such that $s \rightrightarrows t$ for no term t.*

It is clear that every $\rightrightarrows$-normal term contains no $\rightrightarrows$-redex.

Example 3. In Example 2, $t \equiv f(h(f(h(a), g(f(c, c)))), c)$
$\rightrightarrows f(h(f(a, g(f(c, c)))), c) \rightrightarrows f(h(f(f(a, g(c)), c)), c) \rightrightarrows f(f(f(a, g(c)), c), c)$.

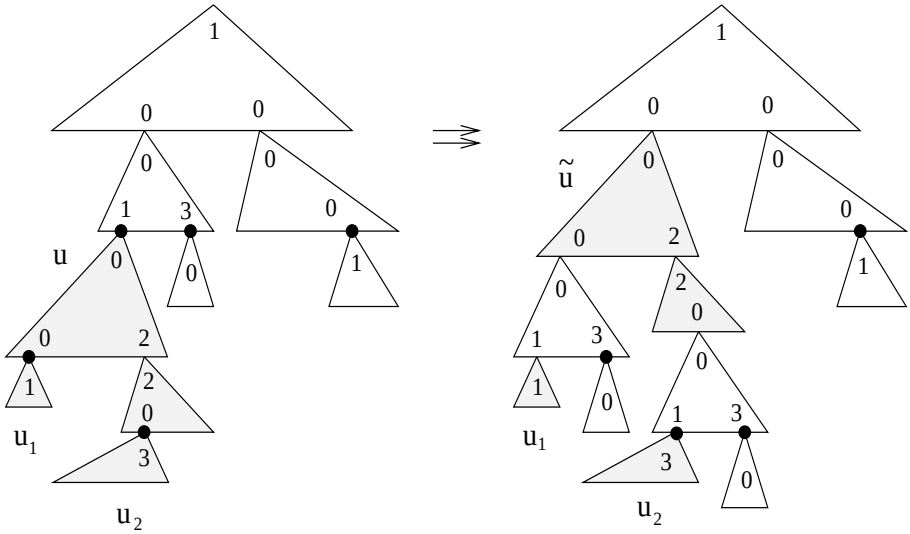


Fig. 3. A $\rightrightarrows$-rewrite step

We note that one can no longer expect a confluent $\rightrightarrows$-reduction if we define $\rightrightarrows$-reduction based on arbitrary $\rightrightarrows$-redexes instead of left-most inner-most $\rightrightarrows$-redexes.

We will need a notion of descendants of $\rightrightarrows$-reduction in the succeeding proofs.

Definition 8. *Let $A : s \rightrightarrows_u t$ be a $\rightrightarrows$-rewrite step. Suppose $s \equiv C[u]_q$ and p is the peak of u in s. For $o \in \mathrm{Pos}(s)$, the set $A(o)$ of* descendants of *o by A is defined as follows: Let $u_1, \ldots, u_n$ $(n \geq 0)$ be activators of u with $u \equiv C[u_1, \ldots, u_n]_{q_1,\ldots,q_n}$. Then,*

$$A(o) = \begin{cases} \{o\} & \text{if } o < p \text{ or } o \mid p, \\ \{p.q_i.(o\backslash p) \mid 1 \leq i \leq n\} & \text{if } p < o \text{ and } q \not\leq o, \\ \{p.(o\backslash q)\} & \text{if } q < o \text{ and } \forall i\ (q.q_i \not\leq o), \\ \{p.q_j.(p\backslash q).(o\backslash(q.q_j))\} & \text{if } q.q_j < o \text{ for some } j\ (1 \leq j \leq n). \\ \emptyset & \text{if } o \in \{p, q, q.q_1, \ldots, q.q_n\}. \end{cases}$$

For $u \equiv s/o$, the set of descendants of u by A is $\{t/o' \mid o' \in A(o)\}$.

One can readily check the next lemma.

Lemma 5. *Let $A : s \rightrightarrows_u t$ be a $\rightrightarrows$-rewrite step. Suppose $s \equiv C[u]_q$ and p is the peak of u in s. Let $u_1, \ldots, u_n$ $(n \geq 0)$ be activators of u with $u \equiv C'[u_1, \ldots, u_n]_{q_1,\ldots,q_n}$. Let $P = \bigcup\{A(o) \mid o \in \mathrm{Dcn}(s)\}$ and $Q = \{p.(q_i\backslash q).(q\backslash p) \mid u_i$ is inconnectable, $1 \leq i \leq n\}$. Then,*

$$\mathrm{Dcn}(t) = \begin{cases} P \cup Q & \text{if } p \notin \mathrm{Dcn}(s), \\ \{p\} \cup P \cup Q & \text{if } p \in \mathrm{Dcn}(s). \end{cases}$$

Lemma 6. *Let $A : s \rightrightarrows_u t$ be a $\rightrightarrows$-rewrite step, and p the peak of u in s. Then,*

1. *s/p is active iff t/p is active;*
2. *every descendant of an active subterm is active;*
3. *every active subterm of t different from t/p is a descendant of an active subterm different from u and s/p.*

Proof. By Lemma 5 and definition, it suffices to prove that s/o is active iff t/o is active, where $o = \max\{p' \leq p \mid s/p' \text{ is a special subterm of } s\}$. We distinguish two cases:

1. $o = p$. The case where $p = \Lambda$ is trivial. Otherwise, suppose that s/p is a proper special subterm of s. We show s/p and t/p are active in s and t respectively. Since s/p is root-active, s/p is active in s. We know hence $\mathrm{sort}(s/p) = 0$, $\mathrm{sort}(s, p) \neq 0$ and every principal subterm of s/p with the peak p is active. If u has no activator, then $t/p \equiv u$, which is active in t. Otherwise suppose u has activators $u_1, \ldots, u_n$ $(n > 0)$ with $u \equiv C'[u_1, \ldots, u_n]$. First, $\mathrm{sort}(t/p) = \mathrm{sort}(u) = 0$. By definition, each principal subterm of t/p is either a descendant of a principal subterm of u that is not an activator, or a principal subterm of $\mathrm{leaf}(C')[u_i]$ $(1 \leq i \leq n)$. Principal subterms from the former do not have the peak p in t, because u is properly active in s. Principal subterms from the latter also do not have the peak p, because $\mathrm{sort}(t, p.q_i) = 0$. Thus, t/p is active.
2. $o < p$. Then s/p is not special. Thus, $\mathrm{sort}(s, p) = 0$ and so the principal subterms of s/o contained in s/p can not have the peak o. Also, the principal subterms of s/o not contained in s/p coincide those of t/o not contained in t/p. Therefore, s/o is active in s iff t/o is active in t.

Definition 9. *For each special subterm u of t, we define* $\mathrm{depth}(u)$ *inductively as follows: (1)* $\mathrm{depth}(t) = 0$. *(2) Suppose u is a special subterm of* $\mathrm{depth}(u) = k$ *and $u \equiv C[\![(u_1)_{\gamma_1}, \ldots, (u_n)_{\gamma_n}]\!]$. Then for $i = 1, \ldots, n$*

$$\mathrm{depth}(u_i) = \begin{cases} k \dot{-} 1 & \text{if } \gamma_i = 0, \\ k + 1 & \text{if } \mathrm{sort}(u_i) = 0, \\ k & \text{otherwise.} \end{cases}$$

Lemma 7. *The relation $\rightrightarrows$ is well-founded and confluent.*

Proof. For each $\rightrightarrows$-redex v in s, let $\mathrm{weight}(v) = \langle \mathrm{depth}(v), \mathrm{rank}(v), M \rangle$ where $M = [\langle \mathrm{depth}(v'), \mathrm{rank}(v') \rangle \mid v'$ is an active subterm of s different from v yet has the same peak as $v\,]$. And, for each term s, let $\|s\| = [\mathrm{weight}(u) \mid u$ is a $\rightrightarrows$-redex in $s\,]$.

Let $\succcurlyeq$ be an order defined by: $\langle i_1, j_1 \rangle \succcurlyeq \langle i_2, j_2 \rangle$ iff either (1) $i_1 > i_2$ or (2) $i_1 = i_2$ and $j_1 > j_2$; and $\succcurlyeq\!\!\succcurlyeq$ the multiset extension of $\succcurlyeq$. Let $\succ$ be an order defined by: $\langle i_1, i_2, M_1 \rangle \succ \langle j_1, j_2, M_1 \rangle$ iff either (1) $i_1 > i_2$, (2) $i_1 = i_2$ and $j_1 > j_2$ or (3) $i_1 = i_2$, $j_1 = j_2$ and $M_1 \succcurlyeq\!\!\succcurlyeq M_2$; and $\succ\!\!\succ$ the multiset extension of $\succ$. We are going to verify that $s \rightrightarrows t$ implies $\|s\| \succ\!\!\succ \|t\|$.

Suppose $A : s \rightrightarrows_u t$, and let p be the peak of u in s. By the definition of $\rightrightarrows$-reduction, if v is a $\rightrightarrows$-redex in t then either (1) $v \equiv t/p$, (2) v is a descendant of a $\rightrightarrows$-redex in s different from u and s/p, or (3) v is a descendant of a properly active subterm of s with the peak $q.q_i$ for some i.

If t/p is a $\rightrightarrows$-redex then s/p is special by Lemma 5 and so $\mathrm{depth}(s/p) = \mathrm{depth}(t/p)$. Therefore, we have $\mathrm{weight}(u) \succ \mathrm{weight}(t/p)$, because $\mathrm{depth}(u) = \mathrm{depth}(s/p) + 1$. If v is a descendant of a properly active subterm of s with the peak $q.q_i$ for some i, then we have $\mathrm{weight}(u) \succ \mathrm{weight}(v)$, because $\mathrm{depth}(u) = \mathrm{depth}(v)$ and $\mathrm{rank}(u) > \mathrm{rank}(v)$. Thus, because u has no descendant, it suffices to show that $[\mathrm{weight}(v)] \succcurlyeq [\mathrm{weight}(t/o') \mid o' \in A(v)]$ for any $\rightrightarrows$-redex v of s different from u and s/p.

Let v be a $\rightrightarrows$-redex in s and o the position of v in s. We distinguish two cases:

1. $o < p$ or $o \mid p$. Then $A(o) = \{o\}$ and therefore it suffices to show $\mathrm{weight}(v) \geq \mathrm{weight}(t/o)$. For any subterm s/o' satisfying $o' \leq p$ or $o' \mid p$, the following hold: (1) s/o' is active iff t/o' is active (by Lemma 6); (2) $\mathrm{rank}(s/o') \geq \mathrm{rank}(t/o')$ (for, $\mathrm{rank}(s/p) \geq \mathrm{rank}(t/p)$); (3) $\mathrm{depth}(s/o') = \mathrm{depth}(t/o')$. Therefore, we have $\mathrm{weight}(v) \geq \mathrm{weight}(t/o)$.
2. $p < o$. By definition, $\rightrightarrows$-redexes contained in s/p have the peak p. Thus, for any descendant v' of v, $\mathrm{depth}(v) = \mathrm{depth}(v')$ and $\mathrm{rank}(v) = \mathrm{rank}(v')$. Further we have $\pi_3(\mathrm{weight}(v)) \backslash \pi_3(\mathrm{weight}(v')) = [\langle \mathrm{depth}(u), \mathrm{rank}(u)\rangle]$, and $\mathrm{depth}(u) = \mathrm{depth}(w)$ and $\mathrm{rank}(u) > \mathrm{rank}(w)$ for any $\langle \mathrm{depth}(w), \mathrm{rank}(w)\rangle \in \pi_3(\mathrm{weight}(v')) \backslash \pi_3(\mathrm{weight}(v))$[2]. Thus, we get $\pi_3(\mathrm{weight}(v)) > \pi_3(\mathrm{weight}(v'))$, and so $\mathrm{weight}(v) > \mathrm{weight}(v')$.

Thus $s \rightrightarrows t$ implies $\|s\| \succ\succ \|t\|$. It is readily checked $\succ\succ$ is well-founded; hence so is $\rightrightarrows$.

Next, $\rightrightarrows$ is locally confluent; for, if two left-most inner-most $\rightrightarrows$-redexes occur in a term then their peaks are disjoint. Therefore $\rightrightarrows$ is confluent.

Thus every term s has the unique $\rightrightarrows$-normal form; we denote it by $s{\downarrow\downarrow}$.

3.3 Simulating $\Theta(\mathcal{R})$-reductions in $\rightrightarrows$-normal terms

To show termination of $\mathcal{R}$ implies that of $\Theta(\mathcal{R})$, we derive a contradiction assuming that there exists an infinite $\Theta(\mathcal{R})$-reduction sequence and that $\mathcal{R}$ is terminating. To this end, we next map infinite $\Theta(\mathcal{R})$-reductions to infinite $\Theta(\mathcal{R})$-reductions consisting of $\rightrightarrows$-normal terms.

We write $s \to_{[p,l\to r]} t$ when $s \equiv C[l\sigma]_p$ and $t \equiv C[r\sigma]_p$ for some context C, substitution σ and $l \to r \in \mathcal{R}$.

Definition 10. *Let $A : s \to_{[p,l\to r]} t$ be a rewrite step. For $o \in \mathrm{Pos}(s)$, the set $A(o)$ of* descendants of *o by A is defined as follows:*

$$A(o) = \begin{cases} \{o\} & \text{if } o < p \text{ or } o \mid p, \\ \{p.p_3.p_2 \mid r/p_3 = l/p_1\} & \text{if } o = p.p_1.p_2 \text{ with } p_1 \in \mathrm{Pos}_{\mathcal{V}}(l), \\ \emptyset & \text{otherwise.} \end{cases}$$

[2] Here, $\pi_i\langle a_1, a_2, a_3\rangle = a_i$.

For $u \equiv s/q$, the set of descendants of u by A is the set $\{t/q' \mid q' \in A(q)\}$.

We first need the following lemma.

Lemma 8. *If $s \to t$ and s is $\rightrightarrows$-normal, then t is also $\rightrightarrows$-normal.*

Proof. We distinguish two cases:

1. $s \to t$ is not destructive. Let $s \equiv C[!u]_q$, $t \equiv C[!u']_q$ and $u \to^o u'$. By the definition of the $\rightrightarrows$-redex, it suffices to show that (a) no principal subterm of u' is a $\rightrightarrows$-redex. (b) u' itself is not a $\rightrightarrows$-redex.
   (a) Suppose that there exists a principal subterm v' of u' that is a $\rightrightarrows$-redex. Then there exists a principal subterm v of u with descendant v'. Let the peak of v in s be p and that of v' in t p'. Since $u \to u'$ is outer, v' is properly active iff so is v. Thus, by our assumption, we know t/p' is root-active while s/p is not. But this can not happen by Lemma 1.
   (b) Suppose u' is a $\rightrightarrows$-redex. By our assumption, this implies that u' is properly active while u is not. Thus, since $\mathrm{sort}(u) = \mathrm{sort}(u') = 0$, there exists a principal subterm v of u with peak q. But then, by Lemma 1, there exists a descendant of v with the peak q. This is a contradiction.
2. $s \to t$ is destructive. If $s \to t$ is destructive at level 1 then the statement holds obviously. Suppose $s \to t$ is destructive at level $k > 1$. Then, by Lemma 4, we have $s \equiv C[(!u)_\gamma]$, $t \equiv C[u_i]$ and $u \to u_i$ for some u properly active in s and u_i an activator of u in s. Let p the peak of u in s.
   If $\gamma \neq \mathrm{sort}(u_i)$, then the only possible $\rightrightarrows$-redex in t is u_i, but, since $\mathrm{sort}(u_i) \neq 0$, this can not be a new $\rightrightarrows$-redex. Therefore, suppose otherwise, i.e. $\gamma = \mathrm{sort}(u_i)$. Then the only possible $\rightrightarrows$-redex in t is (a) a principal subterm of u_i and (b) the minimal special subterm of t that contains t/p.
   (a) Suppose that there exists a principal subterm w of u_i that is a $\rightrightarrows$-redex. Then the peak of the descendant of w in t equals p. For otherwise, the peak of w in s is contained in u_i, and therefore it follows that w is a $\rightrightarrows$-redex. Thus, by our assumption, we know that t/p is root-active. But this implies s/p is root-active since u is properly active. Therefore, it follows that u is a $\rightrightarrows$-redex, which is not the case.
   (b) Suppose that w is the minimal special subterm of t that contains t/p and w is a $\rightrightarrows$-redex. Let v be the minimal special subterm of s that contains s/p. From the fact that t/p is properly active while s/p is not, we know $t/p \equiv w$, because u is the only principal subterm of v that has no descendant that is a principal subterm of w. But this implies, since u and w are properly active, s/p is root-active, and hence u is a $\rightrightarrows$-redex. This is a contradiction.

We also need in our proof the fact that our mapping from terms to $\rightrightarrows$-normal terms preserves infiniteness of $\Theta(\mathcal{R})$-reductions. A notion of head reductions is required to show this.

Definition 11. *Let $t \equiv C[\![t_1, \ldots, t_n]\!]$, and p_i the peak of t_i for each $i = 1, \ldots, n$. The* head *of t is defined as:* $\mathrm{head}(t) = t[p_i \leftarrow \Box \mid 1 \leq i \leq n]$. *We write $t \equiv C'[\![u_1, \ldots, u_k]\!]$ when $t \equiv C'[u_1, \ldots, u_k]$ and C' is the head of t.*

Definition 12. *A rewrite step $s \to t$ is said to be* a body reduction *(written as $s \to^{bd} t$) if*

$$s \equiv C[\![s_1, \ldots, C'[l\sigma], \ldots, s_n]\!] \to C[s_1, \ldots, C'[r\sigma], \ldots, s_n] \equiv t$$

for some terms $s_1, \ldots, s_n$, a substitution σ, $l \to r \in \mathcal{R}$, and $C' \in \Theta(\mathcal{C})$; otherwise it is a head reduction *(written as $s \to^{hd} t$). The redex of a body reduction is a* body redex*; that of a head reduction a* head redex*.*

The following commutativity lemma is a basis of our reduction simulation.

Lemma 9. *If $s \to t$ and $s \rightrightarrows s_1$, then there exists s_2, t_1 such that $s_1 \overset{*}{\rightrightarrows} s_2 \overset{*}{\to} t_1 \overset{*}{\leftleftarrows} t$; further if $s \to^{hd} t$ then $s_2 \to^{hd} t_1$.*

Proof. Suppose $A : s \equiv \hat{C}[l\sigma]_o \to \hat{C}[r\sigma] \equiv t$ and $B : s \rightrightarrows_u s_1$. Let $s \equiv C[u]_q$, p the peak of u. We distinguish the cases according to the position of the redex of $s \to t$:

1. $o \mid p$. Then, we have $s_1 \equiv s_2 \to_{[o,l\to r]} t_1 \leftleftarrows_{t/q} t$ for some t_1.
2. $o < p$. Then $o.o_x \leq p$ for some $o_x \in \mathrm{Pos}_{\mathcal{V}}(l)$ by Lemma 1. Let $\{o_1, \ldots, o_l\} = \{o' \in \mathrm{Pos}_{\mathcal{V}}(l) \mid l/o' = l/o_x, o' \neq o_x\}$ and $\{v_1, \ldots, v_k\} = A(s/o_x)$. Then v_i is a $\rightrightarrows$-redex for all $i = 1, \ldots, k$ and so is $s/(o.o_i.(q\backslash o_x))$ for all $i = 1, \ldots, l$. Thus, we have $s_1 \rightrightarrows_{o_1} \cdots \rightrightarrows_{o_l} s_2 \to_{[o,l\to r]} t_1 \leftleftarrows_{v_k} \cdots \leftleftarrows_{v_1} t$ for some t_1.
3. $p \leq o$. Let $u_1, \ldots, u_n$ $(n \geq 0)$ are activators of u and $u \equiv C'[u_1, \ldots, u_n]$.
   (a) $q \mid o$. Then $C[\ ]_q \to_{[o,l\to r]} C^*[\ ]_q$ and $\mathrm{leaf}(C) \to_{[o\backslash p, l\to r]} \mathrm{leaf}(C^*)$ for some C^*. Let $B(o) = \{o_1, \ldots, o_n\}$. Then s_1/o_i is a redex for any $i = 1, \ldots, n$, and so we have $s_1 \equiv s_2 \to_{[o_1,l\to r]} \cdots \to_{[o_n,l\to r]} t_1 \leftleftarrows_{t/q} t$.
   (b) $q.q_i \not\leq o$ for all i $(1 \leq i \leq n)$. If $o = q$ and $s \to t$ is destructive, then $r\sigma$ is an activator by Lemma 4, and so we have $s_1 \to_{[p,l\to r]} t_1 \equiv t$. (See Figure 4.) Otherwise, for any $i = 1, \ldots, n$, either $o \mid q.q_i$ or $o.o_x \leq q.q_i$ for some $o_x \in \mathrm{Pos}_{\mathcal{V}}(l)$. Hence, t/q' is an activator of t/p for any $q' \in \bigcup_i A(q.q_i)$. Also, $s_1/p.(q\backslash o)$ is a $(\mathcal{R}$-)redex, because $u_i \equiv u_j$ implies $\mathrm{leaf}(C')[u_i] \equiv \mathrm{leaf}(C')[u_j]$. Thus, we have $s_1 \to_{[p.(o\backslash q),l\to r]} t_1 \leftleftarrows_{t/q} t$. (Note that there may be the case that $\bigcup_i B(q.q_i) = \emptyset$; however, by the definition of $\rightrightarrows$-reduction, we have $t_1 \leftleftarrows_{t/q} t$ also in this case.)
   (c) $q.q_i \leq o$ for some i $(1 \leq i \leq n)$. Let $u_j \to u'_j$ with $s \equiv C'[u_j]$ and $t \equiv C'[u'_j]$. Since $\mathrm{sort}(u_j) \neq 0$, $u_j \to u'_j$ is not destructive at level 1 by Lemma 3. Hence $\mathrm{sort}(u_j) = \mathrm{sort}(u'_j)$ and so u'_j is an activator of the descendants of u. Thus, we have $s_1 \to_{[p.q_j.(q\backslash p).(o\backslash(q.q_j)),l\to r]} t_1 \leftleftarrows_{t/q} t$.

Finally, observe that $s \to^{hd} t$ only in the cases 1 or 2.

Lemma 10. *If $s \to t$, then $s{\downarrow\downarrow} \overset{*}{\to} t{\downarrow\downarrow}$; further, if $s \to^{hd} t$, then $s{\downarrow\downarrow} \to^{hd} t{\downarrow\downarrow}$.*

Proof. For any term u, let $\sharp u = \max\{n \mid n \text{ is the length of } u \overset{*}{\rightrightarrows} u{\downarrow\downarrow}\}$. Note $\sharp u$ is well-defined since $\overset{*}{\rightrightarrows}$ is terminating (Lemma 7) and every term contains only finitely many $\rightrightarrows$-redexes. Using Lemma 8 and Lemma 9, it is not difficult to show $s \overset{*}{\to} t$ implies $s{\downarrow\downarrow} \overset{*}{\to} t{\downarrow\downarrow}$ by induction on $\sharp s$.

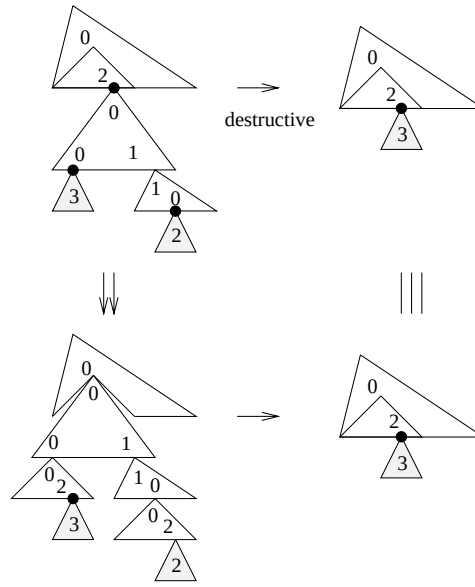


Fig. 4. Destructive step from an active subterm

Definition 13. *Let $s \to t$ be a destructive rewrite step at level $k > 1$. Then by Lemma 4 we have $s \equiv C[!u]$, $t \equiv C[u_i]$ and $u \to u_i$ for some u properly active in s and u_i an activator of u in s. Let p be the peak of u in s. We say $s \to t$* critically destructive *if s/p is root-active.*

The two lemmata below are proved in a straightforward way.

Lemma 11. *Suppose $\mathcal{R}$ is terminating. Every infinite $\Theta(\mathcal{R})$-reduction sequence containing infinitely many head reductions contains infinitely many rewrite steps that are critically destructive.*

Lemma 12. *If $s \to t$ and s is $\rightrightarrows$-normal then $s \to t$ is not critically destructive.*

Lemma 13. *If $\mathcal{R}$ is terminating then so is $\Theta(\mathcal{R})$.*

Proof. Suppose $\mathcal{R}$ is terminating. Let $s_0 \to s_1 \to \cdots$ be an infinite $\Theta(\mathcal{R})$-reduction sequence. Without loss of generality, we can assume that there are infinitely many head reductions in it. Then by Lemma 10, we have a reduction sequence $s_0{\downarrow\downarrow} \xrightarrow{*} s_1{\downarrow\downarrow} \xrightarrow{*} \cdots$, which contains infinitely many head reductions. By Lemma 11, it must contain infinitely many rewrite steps that are critically destructive. As $s_i{\downarrow\downarrow}$ is $\rightrightarrows$-normal, this contradicts Lemma 12.

Now, Theorem 1 immediately follows from Lemma 13.

4 Applications and Related Results

Sort introduction technique based on our result, as well as that based on other known persistency results, is useful to detect a property of TRSs.

Example 4. Let $\mathcal{R} = \{f(g(a), g(b), x) \to f(x, x, x),\ g(x) \to x\}$. To show termination of $\mathcal{R}$, we assume the following sort assignment: $\{f : 0 \times 0 \times 0 \to 1,\ g : 0 \to 0,\ a : 0,\ b : 0\}$. It is clear that $\mathcal{R}$ under this sort assignment is a STRS with only variables of sort 0. Thus, by Theorem 1, it suffices to show termination of $\mathcal{R}$ under this sort assignment. Terms of sort 0 are terminating and confluent, since only applicable rule is $g(x) \to x$. Terms of sort 1 have form $f(t_1, t_2, t_3)$ where t_1, t_2, t_3 are terms of sort 0. Suppose contrary that there exists an infinite reduction sequence of terms of sort 1. Then since terms of sort 0 are terminating, it must contain a reduction at the root position, which has the form $f(g(a), g(b), t) \to f(t, t, t)$ for some term t of sort 0. Since $g(a)$ and $g(b)$ have distinct normal forms and terms of sort 0 are confluent, we never have $g(a) \overset{*}{\leftarrow} t \overset{*}{\to} g(b)$. Thus we know that every reduction sequence starting from $f(t, t, t)$ never has a reduction at the root position. Since t is terminating, this implies $f(t, t, t)$ is terminating, which is a contradiction.

It is known that termination is persistent for the class of many-sorted TRSs that do not contain both collapsing and duplicating rules [10]. However, the argument above does no work with this result, because $\mathcal{R}$ is collapsing and duplicating.

Another kind of applications of our result is to prove that termination is preserved under suitable translations of TRSs.

Example 5 ([4]). For each n-ary function symbol $f \in \mathcal{F}$, let $f_1, \ldots, f_{n-1}$ be new binary function symbols, and let $\hat{\mathcal{F}}$ be the collection of such new function symbols. Define a transformation $\hat{}$ from terms on $\mathcal{F}$ to those on $\hat{\mathcal{F}}$ by:

$$\hat{t} = \begin{cases} t & \text{if } t \in \mathcal{V}, \\ f_1(\hat{t}_1, f_2(\hat{t}_2, f_3(\cdots, f_{n-1}(\hat{t}_{n-1}, \hat{t}_n)\cdots))) & \text{if } t \equiv f(t_1, \ldots, t_n). \end{cases}$$

And, finally let $\hat{\mathcal{R}} = \{\hat{l} \to \hat{r} \mid l \to r \in \mathcal{R}\}$. Using Theorem 1, we can show that $\mathcal{R}$ is terminating if and only if $\hat{\mathcal{R}}$ is terminating.

($\Leftarrow$)-part is trivial. To show ($\Rightarrow$)-part, we introduce a set of sorts by $\mathcal{S} = \{0\} \cup \bigcup\{\delta_f^1, \ldots, \delta_f^{n-2} \mid f \in \mathcal{F},\ f \text{ is } n\text{-ary}\}$, and sort assignment on $\hat{\mathcal{F}}$ as: $f_1 : 0 \times \delta_f^1 \to 0$, $f_i : 0 \times \delta_f^i \to \delta_f^{i-1}$ $(i = 2, \ldots, n-2)$, and $f_{n-1} : 0 \times 0 \to \delta_f^{n-2}$ for each $f \in \mathcal{F}$. It is clear that $\hat{\mathcal{R}}$ is a STRS under this sort assignment with only variables of sort 0. Thus, by Theorem 1, $\hat{\mathcal{R}}$ is terminating if and only if $\hat{\mathcal{R}}$ is terminating on this sort assignment. Suppose $\hat{\mathcal{R}}$ is not terminating. Then there also exists an infinite reduction sequence consisted of terms well-sorted under this sort assignment. Without loss of generality, we assume terms in this reduction sequence have sort 0. It is clear from the sort assignment that if once a function symbol f_i occurs in a well-sorted term of sort 0, it occurs in a form of $f_1(s_1, f_2(s_2, \ldots, f_{n-2}(s_{n-2}, f_{n-1}(s_{n-1}, s_n))\ldots))$ for some

terms $s_1, \ldots, s_n$. Thus, all these terms have form $\hat{t}$ for some t and so we can reversely translate this infinite reduction sequence to that of terms on $\mathcal{F}$. Thus, we know $\mathcal{R}$ is not terminating.

Note that, in the example above, function symbols in $\hat{\mathcal{F}}$ has arity of at most 2, and therefore terms in $\mathcal{T}(\hat{\mathcal{F}}, \mathcal{V})$ have simple structures. Instead, $\mathcal{T}(\hat{\mathcal{F}}, \mathcal{V})$ in general contains much more terms than $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and not all terms in $\mathcal{T}(\hat{\mathcal{F}}, \mathcal{V})$ are images of terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We believe, however, that this kind of coding that preserves termination behavior would be helpful for studying properties of complicated systems by interpreting them in simpler systems.

Another way of simulating a TRS by another TRS containing simpler function symbols is "currying": each function symbol is coded by a constant and a new binary function symbol for "application" is added. Again new terms have simple structures and not all new terms are the images of original terms. This transformation also does not affect termination behavior of TRSs [5,7].

5 Concluding Remarks

We have proved that for any terminating many-sorted TRS $\mathcal{R}$, if $\mathcal{R}$ contains only variables of the same sort then its underlying TRS is also terminating. This is the positive solution to the problem of Zantema that has been appeared as Rewriting Open Problem 60 in [4]. We have also presented some applications of this result.

Acknowledgments

Thanks are due to Hans Zantema and Hitoshi Ohsaki for their comments on the previous version [1] of the paper. The author is grateful to Hans Zantema also for an account on his motivation of the conjecture.

References

1. T. Aoto. A proof of the conjecture of Zantema on a persistent property of term rewriting systems. Research Report IS-RR-98-0008F, School of Information Science, JAIST, 1998. 264
2. T. Aoto and Y. Toyama. On composable properties of term rewriting systems. In *Proceedings of the 6th International Conference on Algebraic and Logic Programming (ALP'97), Southampton, UK,* volume 1298 of *Lecture Notes in Computer Science,* pages 114-128. Springer-Verlag, 1997. 250, 254, 255
3. T. Aoto and Y. Toyama. Persistency of confluence. *Journal of Universal Computer Science,* 3(11):1134-1147. 1997. 250
4. N. Dershowitz, J.-P. Jouannaud, and J. W. Klop. More problems in rewriting. In *Proceedings of the 5th International Conference on Rewriting Techniques and Applications (RTA-93),* volume 690 of *Lecture Notes in Computer Science,* pages 468-487. Springer-Verlag, 1993. 250, 263, 264

5. J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation,* 21:15-39, 1996. 264
6. M. Marchiori. On the modularity of normal forms in rewriting. *Journal of Symbolic Computation,* 22:143-154, 1996. 255
7. A. Middeldorp, H. Ohsaki, and H. Zantema. Transforming termination by self-labelling. In *Proceedings of the 13th International Conference on Automated Deduction (CADE-13),* volume 1104 of *Lecture Notes in Artificial Intelligence,* pages 373-387. Springer-Verlag, 1996. 264
8. H. Ohsaki. *Termination of Term Rewriting Systems: Transformation and Persistence.* PhD thesis, University of Tsukuba, March 1998. 250
9. H. Ohsaki and A. Middeldorp. Type introduction for equational rewriting. In *Proceedings of the 4th International Symposium on Logical Foundations of Computer Science,* volume 1234 of *Lecture Notes in Computer Science,* pages 283-293. Springer-Verlag, 1997. 250
10. H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation,* 17:23-50, 1994. 250, 263

A General Framework for R-Unification Problems

Sébastien Limet[1] and Frédéric Saubion[2]

[1] LIFO, Université d'Orléans, France
Sebastien.Limet@univ-orleans.fr
[2] LERIA, Université d'Angers, France
Frederic.Saubion@univ-angers.fr

Abstract. E-unification (i.e. solving equations modulo an equational theory E) is an essential technique in automated reasoning, functional logic programming and symbolic constraint solving but, in general E-unification is undecidable. In this paper, we focus on R-unification (i.e. E-unification where theories E are presented by term rewriting systems R). We propose a general method based on tree tuple languages which allows one to decide if two terms are unifiable modulo a term rewriting system R and to represent the set of solutions. As an application, we prove a new decidability result using primal grammars.

Keywords: R-unification, Rewrite techniques, Tree languages.

1 Introduction

E-unification (i.e. solving equations modulo an equational theory E) is an essential technique in automated reasoning, functional logic programming and symbolic constraint solving (see [2] for a survey). Unfortunately E-unification is generally undecidable. The problem can be restricted by considering only equational theories presented as confluent term rewriting systems (TRS). In this context, narrowing is a general unification procedure that has been extensively studied [3,8]. But, from an operational point of view, these methods often loop, enumerating infinite sets of answers or computing unproductive branches; such drawbacks challenge their use in programming languages.

In a functional logic programming framework, the computation of whole sets of solutions to an unification problem is, most of the time, not really necessary. It is more interesting to test the existence of solution (which allows one to cut unproductive branches) or to provide a finite representation of infinite sets of answers (to avoid non-terminating enumerations). The purpose of this paper is to propose a general method to decide if two terms can be unified modulo an equational theory and to get compact representations of sets of solutions.

This paper is only devoted to equational theories described by TRS and therefore, it addresses the R-unification problem. Moreover, since we focus on the programming aspects of rewrite techniques, we consider a constructor-based

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 266–281, 1998.

semantics for R-unification, which is more adequate to functional logic programming since it is actually convenient to distinguish the symbols used to represent the data of the program and those used to described operations or relations between these data.

Considering an unification problem $s \stackrel{?}{=}_R t$ (i.e. a goal to be solved modulo a TRS R), the main idea is to describe its ground solutions by a language of ground first order terms. A solution being defined by the instances of the variables of this goal, (i.e. by a tuple of terms and terms being trees) the set of solutions can be viewed as a tree tuple language. To describe this language, we propose to use tree tuple grammars. There exists many different kind of tree grammars that can be used according to the complexity of the solutions to be expressed [5,10,11]. To insure the consistency of this approach, some properties are needed:

- their intersection is a recognized language w.r.t. the classes of grammars used to handle this R-unification problem,
- emptiness is decidable.

The method involves three steps: decomposition of the initial goal into elementary subgoals, representation of sets of solutions for these subgoals and recomposition of the initial goal by intersection or join.

As an application of this work, we extend a decidability result of [14] using primal grammars [10].

2 Preliminaries

2.1 Rewrite Techniques

We recall here rewrite techniques basic notions. We refer the reader to [6] for more details.

Let Σ be a finite set of symbols with arity and $\mathcal{X}$ be an infinite set of variables, $\mathcal{T}(\Sigma, \mathcal{X})$ is the first-order term algebra over Σ and $\mathcal{X}$. Σ is partitioned in two parts : the set $\mathcal{F}$ of definite function symbols (or function symbols), and the set $\mathcal{C}$ of constructor symbols. The terms of $T(\mathcal{C}, \mathcal{X})$ are called data-terms. A term is said linear if it does not contain several times the same variable.

Let t be a term, $\mathcal{O}(t)$ is the set of occurrences of t, $t|_u$ is the subterm of t at occurrence u and $t(u)$ is the symbol that labels the occurrence u of t. $t[u \leftarrow s]$ is the term obtained by replacing in t the subterm at occurrence u by s. A substitution is a mapping from $\mathcal{X}$ into $\mathcal{T}(\Sigma, \mathcal{X})$, which extends trivially to a mapping from $\mathcal{T}(\Sigma, \mathcal{X})$ to $\mathcal{T}(\Sigma, \mathcal{X})$. A data-substitution σ is a substitution such that for each variable x, σx is a data-term.

In the following x, y, z denote variables, s, t, l, r denote terms, f, g, h function symbols, c a constructor symbol, u, v, w occurrences, and σ, θ substitutions.

A term rewrite system (TRS) is a finite set of oriented equations called rewrite rules or rules. lhs means left-hand-side and rhs means right-hand-side. For a TRS R, the rewrite relation is denoted by $\rightarrow_R$ and is defined by $t \rightarrow_R s$ if there exists a rule $l \rightarrow r$ in R, a non-variable occurrence u in t, and a substitution σ,

such that $t|_u = \sigma l$ and $s = t[u \leftarrow \sigma r]$. The reflexive-transitive closure of $\rightarrow_R$ is denoted by $\rightarrow_R^*$, and the symmetric closure of $\rightarrow_R^*$ is denoted by $=_R$. $\rightarrow_R^n$ denotes n steps of the rewrite relation.

A TRS is said confluent if $t \rightarrow_R^* t_1$ and $t \rightarrow_R^* t_2$ implies $t_1 \rightarrow_R^* t_3$ and $t_2 \rightarrow_R^* t_3$ for some t_3. If the lhs (resp. rhs) of every rule is linear the TRS is said left-(resp. right-)linear. If it is both left and right-linear the TRS is said linear. A TRS is constructor based if every rule is of the form $f(t_1, \ldots, t_n) \rightarrow r$ where the t_i's are data-terms.

Since the signatures we deal with follow a constructor discipline, the validity of an equation is defined as a strict equality on terms, in the spirit of functional logic languages like TOY [15] or CURRY [9]. So, as in [1], σ is a solution of the equation $t \stackrel{?}{=} t'$ iff σ is a data substitution and $\sigma t \rightarrow_R^* s$ and $\sigma t' \rightarrow_R^* s$ where s is a data-term [1]. For example, the equation $1 \div x \stackrel{?}{=} 1 \div 0$ where $\div$ is the euclidean division defined by some rewrite rules, is not equivalent to the equation $x \stackrel{?}{=} 0$, since the former is actually not defined.

2.2 Tree Languages

This section presents some basic notions and notations related to tree languages concepts. More details can be found in [5].

As word languages, tree languages can be studied either from the generation point of view (i.e. grammars) or from the recognition point of view (i.e. automata). Tree grammars are very similar to word grammars except that basic objects are trees. Note that, in this paper, we focus on tree grammars where trees are first order terms.

Definition 1. *A tree grammar G is defined by the 4-tuple $(A, N, \mathcal{C}, P)$ where:*

1. *A is the axiom,*
2. *N are the non-terminal symbols with $A \in N$,*
3. *$\mathcal{C}$ is the set of terminal symbols,*
4. *R is a set of production rules[2] $\alpha \Rightarrow \beta$ where $\alpha \in N$, $\beta \in T(\mathcal{C} \cup N)$ [3].*

The derivation relation using a grammar $G = (A, N, \mathcal{C}, P)$ is simply defined by $s \Rightarrow_G t$ iff there exists a rule $\alpha \Rightarrow \beta \in P$ such that $s|_u = \alpha$ and $t = s[u \leftarrow \beta]$. The definition of the language $\mathcal{L}(G)$ generated by G immediately follows from this:

$$\mathcal{L}(G) = \{t \in T(\mathcal{C}) \mid A \stackrel{+}{\Rightarrow}_G t\}$$

where $\stackrel{+}{\Rightarrow}_G$ denotes as usual the transitive closure of $\Rightarrow_G$.

[1] Note that such data-term s is irreducible because there is no relation between constructors, and it is then unique because of confluence.

[2] We use here the $\Rightarrow$ symbol for productions to avoid confusion with the rewrite relation $\rightarrow$.

[3] The standard definition authorizes variables in β but here we consider only ground terms.

As for word grammars, the notion of regular tree grammar is defined as a grammar $G = (A, N, \mathcal{C}, P)$ where all non-terminal symbols have arity 0 and production rules have the form $X \Rightarrow \beta$ where X is a non-terminal and $\beta \in \mathcal{T}(\mathcal{C} \cup N)$.

Extensions of the basic definitions of tree grammars have been proposed in order to express more sophisticated languages. We can mention tree tuples synchronized grammars (TTSG [11]). Primal grammars [10] which are used in this paper have a quite different definition.

From a practical point of view, languages of tuples of finite trees can be recognized by automata (see [5] for more details).[5] mentions different notions of recognizability for tuples of finite trees: by considering products of recognizable sets, by stacking trees or by using Ground Tree Transducers.

In the following, we focus on tree tuples grammars since we are interested in tuples of terms and previous definitions are extended to tree tuple languages (the only difference is that the axiom of the grammar is a tuple).

3 Presentation of the Method

This section describes how, given a TRS R, an unification problem $s \stackrel{?}{=}_R t$[4] can be translated in terms of tree languages. This approach is justified by the fact that it can be much more interesting to determine if an unification problem has at least one solution and then to provide a finite representation of its solutions rather than to enumerate all its solutions. Usually, solving a goal $s \stackrel{?}{=} t$ consists in computing substitutions σ such that $\sigma s =_R \sigma t$. But, the notion of solution can be handled differently by considering the variables $\{x_1, ..., x_n\}$ appearing in s and $\{x_{n+1}, ..., x_m\}$ in t. Then, solutions can be represented by the set of tuples

$$\{(t_1, ..., t_m) \in \mathcal{T}(\mathcal{C})^m \mid s[x_1 \leftarrow t_1, .., x_n \leftarrow t_n] =_R t[x_{n+1} \leftarrow t_{n+1}, .., x_m \leftarrow t_m]\}$$

This set is a language $\mathcal{L}$ of tree tuples (in fact terms tuples). Therefore, decidability of unification is now equivalent to emptiness decidability of $\mathcal{L}$. Obviously, the main difficulty lies in the construction of $\mathcal{L}$ starting from the goal $s \stackrel{?}{=} t$.

Let us consider the following example to illustrate our method. Given a TRS defining two functions (namely the even property and the addition over symbolic integers in an equational logic programming style):

Example 1.

$$\begin{aligned} &e(0) \rightarrow true \\ &e(s(s(x))) \rightarrow e(x) \\ &p(0, y) \rightarrow y \\ &p(s(x), y) \rightarrow s(p(x, y)) \end{aligned}$$

Clearly, the constructors are $\mathcal{C} = \{0, s, true\}$ and therefore the data-terms are symbolic integers and the boolean value *true*. The functions are $\mathcal{F} = \{e, p\}$

[4] The subscript R will be omitted when clear from the context.

which respectively denote a boolean valued function (i.e. a predicate) and an operation over integers.

We want to solve the goal $e(p(x,y)) \stackrel{?}{=} true$ (i.e. $\{(x,y) \mid x+y \bmod 2 = 0\}$). First of all, we decompose the initial goal into three sub-goals: $e(y_2) \stackrel{?}{=} y_1, p(x,y) \stackrel{?}{=} y_2, true \stackrel{?}{=} y_1$. The set of ground solutions of $e(y_2) \stackrel{?}{=} y_1$ can be considered as the following (infinite) set of pairs of ground terms $\mathcal{L}_1 = \{(t_1,t_2) \mid e(t_2) \stackrel{*}{\rightarrow}_R t_1\}$. Similarly, we introduce two languages $\mathcal{L}_2$ and $\mathcal{L}_3$ corresponding to the two other sub-goals and such that: $\mathcal{L}_2 = \{(t_1,t_2,t_3) \mid p(t_2,t_3) \stackrel{*}{\rightarrow}_R t_1\}$ and $\mathcal{L}_3 = \{(true)\}$. These languages have to be formalized thanks to appropriate tree tuple grammars, describing the relation $\stackrel{*}{\rightarrow}_R$.

At this step, we have to reconstruct the global solution of the initial unification problem by combining these different partial solutions. This will be achieved by a join operation (i.e. a kind of intersection). The ground solutions to $e(p(x,y)) = y_1$ will be described by the tree language $\mathcal{L}_4 = \{(t_1,t_2,t_3,t_4) \mid (t_1,t_2) \in \mathcal{L}_1 \text{ and } (t_2,t_3,t_4) \in \mathcal{L}_2\}$. Then, we combine the language $\mathcal{L}_3$ in a similar way to get the final language of solutions $\mathcal{L}_5 = \{(t_1,t_2,t_3,t_4) \mid (t_1,t_2,t_3,t_4) \in \mathcal{L}_4 \text{ and } t_1 = true\}$. Since t_3 and t_4 represent instances of x and y, the result is $e(p(t_3,t_4))$ (represented by t_1) and we impose $t_1 = true$. Therefore, we have a description of the ground substitutions σ such that $\sigma(e(p(x,y)) \stackrel{*}{\rightarrow}_R true$. The only step that remains to be performed is to test if $\mathcal{L}_5$ is empty or not.

We now detail the process step by step.

3.1 Decomposition

Here is the first component of our method. Given a TRS R and a general unification problem $s \stackrel{?}{=} t$ is decomposed according to the following definition.

Definition 2. *Let R be a TRS over a signature $\Sigma = \mathcal{F} \cup \mathcal{C}$ and $\mathcal{X}$ a set of variables, the goal to be solved $s \stackrel{?}{=} t$ is decomposed by applying the following inference rules:*

$$\textit{Variable Introduction } \frac{G \cup \{s \stackrel{?}{=} t)\}}{G \cup \{s \stackrel{?}{=} y, t \stackrel{?}{=} y\}}$$

$$\textit{if } s,t \in \mathcal{T}(\Sigma,\mathcal{X}), y \in \mathcal{X} \setminus (Var(G) \cup Var(s) \cup Var(t))$$

$$\textit{Function Decomposition } \frac{G \cup \{s \stackrel{?}{=} t\}}{G \cup \{s[u \leftarrow y] \stackrel{?}{=} t, s|_u \stackrel{?}{=} y\}}$$

$$\textit{if } s|_u \in \mathcal{F}, y \in \mathcal{X} \setminus (Var(G) \cup Var(s) \cup Var(t))$$

$$\textit{Goal Linearization}\ \frac{G \cup \{s \stackrel{?}{=} t\}}{G \cup \{s[u_1 \leftarrow y_1, ..., u_n \leftarrow y_n] \stackrel{?}{=} t, s|_{u_1} \stackrel{?}{=} y_1, ..., s|_{u_n} \stackrel{?}{=} y_n\}}$$

$$\textit{if } \exists u_1, ..., u_n \in \mathcal{O}(s), n > 1, s|_{u_1}, ..., s|_{u_n} \in Var(s) \textit{ and } s|_{u_1} = ... = s|_{u_n}$$
$$\textit{and } y_1, \ldots, y_n \in \mathcal{X} \setminus (Var(G) \cup Var(s) \cup Var(t)) \textit{ and } u \neq \epsilon$$

Remark 1. Variable introduction allows us to decompose goal into a kind of canonical form: $s \stackrel{?}{=} y$ with $y \in \mathcal{X}$. Therefore, ground solution languages are easier to express.

Function decomposition provides subgoals of the form $f(s) \stackrel{?}{=} t$ with $f \in \mathcal{F}$ such that there is no other function symbols in $f(s)$ (only constructor symbols). This makes the description of the language of solutions easier by looking at the rules of the TRS R defining f (since f will be the only symbol that can be rewritten in this sub-goal).

The last inference rule obviously transforms a goal into a linear goal.

Back to example 1, the goal $e(p(x, y)) \stackrel{?}{=} true$ will be decomposed as:

$$V.I\ \frac{e(p(x,y)) \stackrel{?}{=} true}{e(p(x,y)) \stackrel{?}{=} y_1, true \stackrel{?}{=} y_1}$$
$$F.D\ \frac{e(p(x,y)) \stackrel{?}{=} y_1, true \stackrel{?}{=} y_1}{e(y_2) \stackrel{?}{=} y_1, p(x,y) \stackrel{?}{=} y_2, true \stackrel{?}{=} y_1}$$

At this stage, note that decomposition of the initial goal into several subgoals allows us to represent each sets of subgoal solutions by using different languages. The underlying idea is to choose for each sub problem the adequate grammar to schematize its set of ground solutions and then to recombine them (provided that this reconstruction is possible and that required properties are preserved).

3.2 Description of Sets of Solutions by Means of Tree Languages

In our general framework, we do not consider specific representation or manipulation tools related to languages of m-tuples of trees. These languages can be represented using grammars. Due to remark 1, the work to be done here is to represent the definition in extension of each function defined in the considered TRS by tree tuple languages in order to describe the elementary solutions (i.e. solutions of the sub goals obtained by decomposition). The characteristics of these grammars depend on the complexity of the existing relations between the components of the tuples. For instance the language $\mathcal{L}_1 = \{(s^i(0), s^p(0)) \mid i \text{ is odd and } p \text{ is even}\}$ can be described by a regular grammar with the productions: $\{I \Rightarrow s(0); I \Rightarrow s(s(I)); P \Rightarrow 0; P \Rightarrow s(s(P))\}$ and the axiom (I, P). But, if we consider now $\mathcal{L}_2 = \{(s^n(0), s^{2n}(0))\}$, its representation should differ since it is not a regular language (due to the fact that the second component must

be the double of the first one). Such relations can be handled for instance by introducing synchronizations between productions (see [11]).

We define a working language class $\mathcal{WL}$ as the class of the various tree tuple languages used to solve a given problem. For instance, one may need to combine regular tree tuple languages and languages recognized by tree tuple synchronized grammars to handle unification modulo a particular TRS.

Remark that an emptiness decision procedure is required for this class of languages. Before going on, one has to check that for every language in $\mathcal{WL}$, emptiness is decidable.

Starting from the set of subgoals S obtained by the previous decomposition, we define the notion of set of elementary solutions as:

Definition 3. *Given a set of subgoals* $S_R = \{s_i \stackrel{?}{=}_R y_i\}$*, we define the set:*

$$ES_R = \{(s_i \stackrel{?}{=} y_i, \mathcal{L}_i) \mid \mathcal{L}_i = \{(t_1, ..., t_n) \mid s[x_1 \leftarrow t_2, ..., x_{n-1} \leftarrow t_n] \stackrel{*}{\rightarrow}_R t_1\}\}$$

if $Var(s) = \{x_1, ..., x_{n-1}\}$

At this stage, one has to start from scratch and to choose grammars powerful enough to describe the tuples of solutions but keeping in mind that emptiness must be decidable for this class of languages and moreover that intersections of these languages will be performed in the next step of our method.

3.3 Recomposition of Solutions

The main problem here is to "intersect" languages of tuples with different sizes (i.e languages of n-tuples with languages of m-tuples (see example 1)). This is achieved by a particular join operation.

We generalize the occurrences to tuples in the following way: let $p = (p_1, \ldots, p_n)$ a tuple, $\forall i \in [1, n]\ p|_i = p_i$, and when the p_i's are terms, $p|_{i.u} = p_i|_u$. The following definitions state basic operations over tuples.

Definition 4. *Given two tuples* $(t_1, ..., t_n)$ *and* $(t'_1, ..., t'_m)$*, we define the concatenation operator* $*$ *as:*

$$(t_1, ..., t_n) * (t'_1, ..., t'_m) = (t_1, ..., t_n, t'_1, ..., t'_m)$$

Definition 5. *Given a tuple* $(t_1, ..., t_n)$*, the component elimination is defined as:*

$$(t_1, \ldots, t_i, \ldots, t_n)\backslash_i = (t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$$

Definition 6. *Let* E_1 *be a set of* n_1*-tuples and* E_2 *be a set of* n_2*-tuples. The* k_1, k_2 *join of* E_1 *and* E_2 *is the set of* $n_1 + n_2 - 1$*-tuples defined by*

$$E_1 \bowtie_{k_1,k_2} E_2 = \{tp_1 * (tp_2\backslash_{k_2}) \mid\ tp_1 \in E_1 \text{ and } tp_2 \in E_2 \text{ and } tp_1|_{k_1} = tp_2|_{k_2}\}$$

This operation can also be fully defined w.r.t. the usual intersection of languages.

Proposition 1. *Let E_1 be a set of n-tuples and E_2 be a set of m-tuples*

$$E_1 \bowtie_{k_1,k_2} E_2 = E'_1 \cap E'_2$$

with

$E'_1 = \{(t_1, ..., t_n, t'_1, ..., t'_{k_2-1}, t'_{k_2+1}, ..., t'_m) \mid$
$(t_1, ..., t_n) \in E_1, (t'_1, ..., t'_{k_2-1}, t'_{k_2+1}, ..., t'_m) \in \mathcal{T}(\Sigma)^{m-1}\}$
and
$E'_2 = \{(s'_1, ..., s'_{k_1-1}, s_{k_2}, s'_{k_1+1}, ..., s'_n, s_1, ..., s_{k_2-1}, s_{k_2+1}, ..., s_m)$
$\mid (s_1, ..., s_m) \in E_2, (s'_1, ..., s'_{k_1-1}, s'_{k_1+1}, ..., s'_n) \in \mathcal{T}(\Sigma)^{n-1}\}$

To insure stability of recomposition, one has to check that the join of two languages generates a language that can be handled later in the method (i.e. for two languages in $\mathcal{WL}$, check that their join stays in $\mathcal{WL}$).

More formally, we define a recomposition rule as:

Definition 7. *Given a set of pairs PS of subgoals and languages describing their solutions, solutions are recomposed by applying the following inference rule:*

$$\textit{Recomposition}\ \frac{PS \cup \{(s \stackrel{?}{=} y, \mathcal{L}_1), (t \stackrel{?}{=} y', \mathcal{L}_2)\}}{PS \cup \{(s[y' \leftarrow t] \stackrel{?}{=} y, \mathcal{L}_1 \bowtie_{i,1} \mathcal{L}_2)\}}$$

if y' is represented by the i^{th} component of $\mathcal{L}_1$.

Clearly we start recomposition with the set of elementary solutions ES_R (see definition 3).

4 Application of the Method

In this section, we show how the previous framework can be instantiated with primal grammars as tree language support for representing sets of solutions. First we define primal grammars [10]. Then, we give the class of R unification problems we deal with. The representation of solutions of each elementary subgoal is achieved by primal grammars. The decidability of R-unification comes from decidability of unification (i.e. intersection) of primal grammars.

4.1 Primal Grammars

In this section, we give definitions related to primal grammars which have been introduced by M. Hermann and R. Galbavý in [10]. To avoid too long developments, we restrict the definitions given pp 116-121 in [10] to the subset of primal grammars effectively used here. Moreover, some notations have been modified to fit ours. All missing details can be found in [10].

Definition 8. *(Primal Algebra)*
A counter expression is an expression built over 0*, successor function* s *and a set of counter variable* $\mathcal{C}nt$*.* $s(c)$ *means* $1 + c$ *if* c *is a counter expression. The ground counter expression* $s^n(0)$ *is interpreted as the integer* n*.*

The algebra of primal terms is defined over a set of functions $\tilde{\mathcal{F}}$ *(whose elements* $\tilde{f}$ *have a counter arity* $\mathrm{car}(\tilde{f})$ *and a standard arity* $\mathrm{ar}(\tilde{f})$*), a set of constructor symbols* $\mathcal{C}$*, a set of ordinary variables* $\mathcal{X}$ *and a set of counter variables* $\mathcal{C}nt$*. This is the smallest set such that*

- *each ordinary variable of* $\mathcal{X}$ *is a primal term,*
- *if* $c_1, \ldots, c_k$ *are counter expressions,* $t_1, \ldots, t_n$ *are primal terms and* $\tilde{f} \in \tilde{\mathcal{F}}$ *such that* $car(\tilde{f}) = k$ *and* $ar(\tilde{f}) = k + n$*, then* $\tilde{f}(c_1, \ldots, c_k; t_1, \ldots, t_n)$ *is a primal term,*
- *if* $t_1, \ldots, t_n$ *are primal terms and* c *is a constructor with the arity* $ar(c) = n$ *then* $c(t_1, \ldots, t_n)$ *is a primal term.*

A primal term t *is said to be* regular *if* $\forall u$ *such that* $t|_u = \tilde{f}(t_1, \ldots t_n)$ *where* $\tilde{f} \in \tilde{\mathcal{F}}$*,* t_i *is a data-term, for* $i \in [1, n]$*.*

If a precedence $\prec$ *is defined over*$\tilde{\mathcal{F}}$*,* $Apx(\tilde{f}(\overline{c}; \overline{t})) = \{\tilde{g}(\overline{z}; \overline{u}) | \tilde{f} \succ \tilde{g}$*,* $\overline{z}$ *is a subsequence of* $\overline{c}$ *and* $\overline{u}$ *a subsequence of* $\overline{t}\}$*.*

Definition 9. *(Presburger rewrite system)*
A Presburger rewrite system *contains for each defined functions* $\tilde{f}$ *the following pair of Presburger rewrite rules:*
• *basic rule*

$$\tilde{f}(0, \overline{c}; \overline{x}) \rightarrow_{Pbg} t_1$$

• *and the inductive rule which have the following form:*

$$\tilde{f}(s(n), \overline{c}; \overline{x}) \rightarrow_{Pbg} t_2[u \leftarrow \tilde{f}(n, \overline{c}; \overline{x})]$$

where
• $\overline{c}$ *is a vector of counter variables,* $\overline{x}$ *a vector of ordinary variables,*
• *the rhs of the inductive rewrite rule is a regular primal term.*
• t_1*,* t_2 *are primal terms whose redex belong to the approximation* $Apx(\tilde{f}(s(n), \overline{c}; \overline{x}))$

In [10], two additional conditions are required on t_2 but as discussed p120, any Presburger rewrite system can be transformed to satisfy these two missing restrictions. On the other hand, we omitted one possibility for the inductive rule since it is not used here. In the same way we restrict the recursive application of a function to one occurrence.

Definition 10. *(Primal term grammar)*
A primal term grammar *(or* primal grammar *for short) is a quadruple* $G = (\mathcal{C}, \tilde{\mathcal{F}}, \mathcal{R}, t)$ *where* $\mathcal{C}$ *is a set of constructors,* $\tilde{\mathcal{F}}$ *the set of defined functions,* $\mathcal{R}$ *is a Presburger rewrite system and* t *a primal term called* axiom*.*

The language generated by such a primal term grammar is the set of terms $L(G) = \{\sigma t\downarrow \mid \sigma$ *affects a ground integer value to each counter variable of* t $\}$*.*

$GndL(G)$ is the ground term language recognized by G and is defined as $GndL(G) = \{t \mid t \text{ is a ground term, } t = \theta(t') \text{ and } t' \in L(G)\}$.

Example 2. We want to describe the language of all the integer lists $[n, \ldots, 0]$. Integer are represented with $zero$ and suc and lists are constructed thanks to $*$.

Let G be a primal grammar defined by $\mathcal{C} = \{suc, zero, *\}$, $\tilde{\mathcal{F}} = \{\tilde{f}, \tilde{g}\}$, $\mathcal{R}$ is composed by the 4 Presburger rewrite rules
$\tilde{g}(0) \to_{Pbg} zero$, $\tilde{g}(s(n)) \to_{Pbg} suc(\tilde{g}(n))$,
$\tilde{f}(0) \to_{Pbg} zero$, $\tilde{f}(s(n)) \to_{Pbg} \tilde{g}(s(n)) * \tilde{f}(n)$
and the axiom $\tilde{f}(c)$.

Using the precedence $\tilde{f} \succ \tilde{g}$, our Presburger rules fit definition 9.
$\tilde{f}(1) \to_{Pbg} \tilde{g}(1) * \tilde{f}(0) \to_{Pbg} \tilde{g}(1) * zero \to_{Pbg} suc(\tilde{g}(0)) * zero \to suc(zero) * zero$.
So $L(G) = \{zero, suc(zero) * zero, \ldots, suc^n(zero) * suc^{n-1}(zero) * \ldots * zero, \ldots\}$

4.2 A New Decidability Result Using Primal Grammars

The result proved in this section is in fact a strict extension of a result given by S. Mitra in his Phd Thesis [14]. The proof given by S. Mitra is not detailed on some crucial points. This proof uses a kind of goal decomposition which looks like ours (in fact this decomposition is integrated in a narrowing strategy called lazy narrowing [13]). Then S. Mitra uses (without precisely giving the correspondence) I-Terms of [4]. When we tried to make the proof using our framework, it came out that the restrictions on the TRS were not strict enough to fit the I-term formalism. Therefore the result of Mitra is more restrictive than announced in [14]. In this section, we correct the proof and extend the result by using primal grammars which is a much more powerful term schematization than I-terms.

The following definition gives the class of TRS we deal with.

Definition 11. *(primal TRS)*
A TRS R is called primal TRS *if it is a non-overlapping*[5] *left-linear constructor based TRS which satisfies the following property: each function f is defined*

- *either by a finite set of rewrite rules $f(t_1, \ldots, t_n) \to_R r$ of R such that r is a data-term. f is then called a* non-recursive function
- *or by two rewrite rules $f(t_1, \ldots, t_n) \to_R r$ and $f(t'_1, \ldots, t'_n) \to_R r'$ such that*
  - *$r = C[u \leftarrow f(x_1, \ldots, x_n)]$ where C is a data term and $x_1, \ldots, x_n$ is a set of pairwise different variables such that $\forall i \in [1, n]$, $\{x_i\} = Var(t_i)$.*
  - *r' is either a data term or $r' = C'[u' \leftarrow f'(x'_1, \ldots, x'_{n'})]$ where C' is a data term, $x'_1, \ldots, x'_{n'}$ is a set of pairwise different variables and $f' \neq f$ and f' is defined neither from f nor from non-recursive function.*

  f is then called a recursive function.

[5] A TRS is overlapping if it contains two rules $l_1 \to r_1$ and $l_2 \to r_2$ and a substitution σ s.t. $\sigma(l_1) = \sigma(l_2)$.

Note that mutual recursivity in R is forbidden.

S. Mitra in [14] allows several rules $f(t_1, \ldots, t_n) \rightarrow_R r$ where r is a data-term for recursive functions which cannot be handled by the I-term formalism. On the other hand, only one recursive rewrite rule (i.e. with a function call in the rhs) is admitted and right-linearity is required in contrast with primal TRS.

Here are some technical definitions which are used to deduce Presburger rewrite rules from a primal TRS.

Definition 12. *(technical)*
A TRS R is primal if for a defined function symbol f the depth *of f, denoted $dp(f)$ is*

- *0 if f is a non-recursive function.*
- *1 if f is a recursive function only defined from itself.*
- *$1 + dp(f')$ if f is a recursive function such $f(\overline{t_i}) \rightarrow_R C[u \leftarrow f'(\overline{x_{i'}})]$ and f' is a defined function symbol such that $f \neq f'$.*

The substitution $\sigma_{f(x_1,\ldots,x_n)}$ is defined if f is a recursive function by $\sigma_{f(x_1,\ldots,x_n)} = \{\forall i \in [1,n], x_i \mapsto I_i^f(n_{dp(f)}, \ldots, n_1; V_R)\}$ where $V_R = \bigcup\{Var(l)$ s.t. $l \rightarrow_R r \in R\}$. V_R will also be used instead of the list $y_1, \ldots, y_n$ of the elements of V_R

Now, we are able to define Presburger rewrite rules deduced from a primal TRS R. For this Presburger rewrite system, called $Pbg(R)$, the set of defined functions is $\{\tilde{f} | f \in \mathcal{F}\} \cup \{I_i^f | f \in \mathcal{F}$ and $0 < i \leq Arity(f)\}$. $\tilde{f}$ encodes the set of possible results (i.e. the co-domain) for the function f and I_i^f set of possible values for the i^{th} argument of f. The Presburger rewrite rules of $Pbg(R)$ are deduced only from term rewrite rules of R defining recursive functions.

Definition 13. *($Pbg(R)$)*
The Presburger rewrite system associated to the primal TRS R is denoted by $Pbg(R)$ and is defined by:

- *For each rewrite rule of R, $f(\overline{t_i}) \rightarrow_R r$ where $dp(f) = 1$ and r is a data-term, we deduce the Presburger rewrite rules:*
  - $\tilde{f}(0\,; V_R) \rightarrow_{Pbg} r$
  - *for each* $i \in [1, Arity(f)]$, $I_i^f(0\,; V_R) \rightarrow_{Pbg} t_i$
- *For each rewrite rule of R, $f(\overline{t_i}) \rightarrow_R C[u \leftarrow f(\overline{x'_i})]$ where C is a data-term, we deduce the Presburger rewrite rules:*
  - $\tilde{f}(s(n_{dp(f)}), \ldots, n_1\,; V_R) \rightarrow_{Pbg} \sigma_{f(\overline{x'_i})}(C)[u \leftarrow \tilde{f}(n_{dp(f)}, \ldots, n_1\,; V_R)]$
  - *for each* $i \in [1, Arity(f)]$, $I_i^f(s(n_{dp(f)}), \ldots, n_1\,; V_R) \rightarrow_{Pbg} \sigma_{f(\overline{x'_i})}(t_i)$.
- *For each rewrite rule of R, $f(\overline{t_i}) \rightarrow_R C[u \leftarrow f'(\overline{x'_i})]$ where $f \neq f'$, C is a data-context, we deduce the Presburger rewrite rules:*
  - $\tilde{f}(0, n_{dp(f')}, \ldots, n_1\,; Var_R) \rightarrow_{Pbg} \sigma_{f'(\overline{x'_{i'}})}(C)[u \leftarrow \tilde{f'}(n_{dp(f')}, \ldots, n_1\,; V_R)]$
  - *for each* $i \in [1, Arity(f)]$, $I_i^f(0, n_{dp(f)-1}, \ldots, n_1\,; V_R) \rightarrow_{Pbg} \sigma_{f'(\overline{x'_{i'}})}(t_i)$.

If we consider the precedence $\prec$ such that $\tilde{f} \succ \tilde{f'}$ if $dp(f) > dp(f')$, $\tilde{f} \succ I_i^{f'}$ if $dp(f') \leq dp(f)$ and $I_i^f \succ I_i^{f'}$ if $dp(f) > dp(f')$, we can verify that $Pbg(R)$ fits definition 9 because all defined function symbols appearing in rhs of Presburger rules are less or equal to the function symbol of the lhs according to $\prec$. The counter variables in the rhs are subsequences of the one in lhs. At least for the inductive case, the first counter of the function is decremented from the lhs to the rhs.

Now, we are able to define the primal grammars that generate the definition in extension of each recursive function (to achieved the description step of the method presented in section 3.2).

Definition 14. *($Ext(f)$ for recursive functions)*
Let R be a primal TRS and f a recursive function defined in R. The primal grammar $(\mathcal{C}, \mathcal{D}(R), Pbg(R), ta)$[6] where ta is
$(\tilde{f}(n_{dp(f)}, \ldots, n_1 ; V_R), I_1^f(n_{dp(f)}, \ldots, n_1 ; V_R), \ldots, I^f_{Arity(f)}(n_{dp(f)}, \ldots, n_1 ; V_R))$
is called $Ext(f)$.

The following proposition states the fundamental correspondence between the primal grammar $Ext(f)$ and the definition in extension of the function f according to our semantics.

Proposition 2. *Let R be a primal TRS, for each recursive function f defined in R, $GndL(Ext(f)) = \{(r, t_1, \ldots, t_n) | r, t_1, \ldots, t_n$ are ground data-terms and $f(t_1, \ldots, t_n) \rightarrow_R^* r\}$.*

Proof. First we prove that for each recursive function of R, $GndL(Ext(f)) \subseteq \{(r, t_1, \ldots, t_n) | r, t_1, \ldots, t_n$ are ground data-terms and $f(t_1, \ldots, t_n) \rightarrow_R^* r\}$. This is done on induction on $dp(f)$.

• *Suppose that $dp(f) = 1$,* we prove that for each integer k,
if $(r, t_1, \ldots, t_n) \in (\tilde{f}(k; V_R), \overline{I_i^f(k; V_R)})$ then $f(t_1, \ldots, t_n) \rightarrow_R^{k+1} r$.

▷ *For $k = 0$* we have $(\tilde{f}(0; V_R), I_1^f(0; V_R), \ldots, I_n^f(0; V_R)) \rightarrow_{Pbg} (r', t'_1, \ldots, t'_n)$ and by definition $f(t'_1, \ldots, t'_n) \rightarrow_R r'$ belongs to R.
Since $(r, t_1, \ldots, t_n) \in L((\tilde{f}(0; V_R), I_1^f(0; V_R), \ldots, I_n^f(0; V_R))$, it is a ground instance of $(r', t'_1, \ldots, t'_n)$. Therefore we have $f(t_1, \ldots, t_n) \rightarrow_R^1 r$.

▷ *For $k > 0$,* we suppose the property true for all $k' < k$. As $k > 0$,
$(\tilde{f}(k; V_R), \overline{I_i^f(k; V_R)}) \rightarrow_{Pbg} (\sigma_{f(\overline{x'_i})}(C)[u \leftarrow \tilde{f}(k-1; V_R), \overline{\sigma_{f(\overline{x'_i})}(t'_i)})$. So the rewrite rule $f(t'_1, \ldots, t'_n) \rightarrow_R C[u \leftarrow f(\overline{x'_i})]$ belongs to R and $\forall i \in [1, n]\{x_i\} = Var(t_i)$.

Since $(r, t_1, \ldots, t_n) \in L((\tilde{f}(k; V_R), \overline{I_i^f(k; V_R)}))$, we know that $\forall i \in [1, n]$, t_i is a ground instance of $t'_i[u \leftarrow t''_i])$ where $t''_i \in L(I_i^f(k-1, V_R))$.

Let σ be the substitution defined by $\{\forall i \in [1, n], x'_i \mapsto t''_i\}$. r is a ground instance of $\sigma(C[u \leftarrow r''])$ where $r'' \in L(\tilde{f}(k-1; V_R))$.

[6] $\mathcal{D}(R)$ is the set of defined function symbols of R.

Let θ be the substitution such that $(r, t_1, \ldots, t_n) = \theta(\sigma(C[u \leftarrow r'']), t'_1[u \leftarrow t''_1], \ldots, t'_n[u \leftarrow t''_n])$.

On one hand, we have
$f(\overline{t_i}) = \theta\sigma(f(\overline{t'_i})) \rightarrow_R \theta\sigma(C[u \leftarrow f(\overline{x'_i})]) = \theta\sigma(C[u \leftarrow f(t''_1, \ldots, t''_n)])$.

On the other hand, $(r'', t''_1, \ldots, t''_n) \in GndL((\tilde{f}(k-1; V_R), \overline{I_i^f(k-1; V_R)}))$, so by induction hypothesis $f(t''_1, \ldots, t''_n) \rightarrow_R^k r''$. Therefore
$f(t_1, \ldots, t_n) \rightarrow_R^1 \theta\sigma(C[u \leftarrow f(t''_1, \ldots, t''_n)]) \rightarrow_R^k \theta\sigma(C[u \leftarrow r'']) = r$.

• *Suppose now that* $dp(f) > 1$. Now our induction hypothesis is that $\forall f'$ such that $dp(f') < dp(f)$, if $(r', t'_1, \ldots, t'_{n'}) \in GndL((\tilde{f'}(k, \overline{k'_j}; V_R), \overline{I_i^{f'}(k, \overline{k'_j}; V_R)}))$ then $f'(t'_1, \ldots, t'_{n'}) \rightarrow_R^* r'$.

We prove on induction on k that
if $(r, t_1, \ldots, t_n) \in GndL((\tilde{f}(k, \overline{k'_j}; V_R), \overline{I_i^f(k, \overline{k'_j}; V_R)}))$,
then $f(t_1, \ldots, t_n) \rightarrow_R^{k+1+k''} r$.

▷ *For* $k = 0$, since $dp(f) > 1$ we have
$(\tilde{f}(0, \overline{k'_j}; V_R), \overline{I_i^f(0, \overline{k'_j}; V_R)})) \rightarrow_{Pbg} (\sigma_{f'(\overline{x'_{i'}})}(C)[u \leftarrow \tilde{f'}(\overline{k'_j}; V_R)], \overline{\sigma_{f'(\overline{x'_{i'}})}(t'_i)})$ and $f(t'_1, \ldots, t'_n) \rightarrow_R C[u \leftarrow f'(\overline{x'_{i'}})]$ belongs to R.

Thanks to the substitution $\sigma_{f'(\overline{x'_{i'}})}$, all occurrences of $x'_{i'}$ in C, $t'_1, \ldots, t'_n$ is replaced by $I_{i'}^{f'}(\overline{k'_j}; V_R))$. This means that each occurrences of $x'_{i'}$ corresponds to subterm t''_i of $(r, t_1, \ldots, t_n)$. This term t''_i belongs to $GndL(I_{i'}^{f'}(\overline{k'_j}; V_R))$. Let us call σ the substitution that maps $x'_{i'}$ to $t''_{i'}$.

In the same way, $r|_u$ is the term r'' that belongs to $\tilde{f'}(\overline{k'_j}; V_R)$.
So $(r'', t''_1, \ldots, t''_{n'}) \in GndL((\tilde{f'}(\overline{k'_j}; V_R), \overline{I_{i'}^{f'}(\overline{k'_j}; V_R)}))$ and $dp(f') = dp(f) - 1$ by definition. So by induction hypothesis (for the induction on $dp(f)$) we know that $f'(t''_1, \ldots, t''_{n'}) \rightarrow_R^* r''$.

Moreover, since $(r, t_1, \ldots, t_n) \in L((\tilde{f}(0, \overline{k'_j}; V_R), \overline{I_i^f(0, \overline{k'_j}; V_R)}))$, it is a ground instance of $(\sigma(C[u \leftarrow r'']), \sigma(t'_1), \ldots, \sigma(t'_n))$. Let us call θ the substitution that makes these two tuples equal.
So we have $f(t_1, \ldots, t_n) = \theta\sigma(f(t'_1, \ldots t'_n)) \rightarrow_R \theta\sigma(C[u \leftarrow f'(x'_1, \ldots, x'_{n'})])$ which is $\theta\sigma(C[u \leftarrow f'(t''_1, \ldots, t''_{n'})]) \rightarrow_R^* \theta\sigma(C[u \leftarrow r'']) = r$.

▷ The induction step on k is proved in the same way as the induction step for $dp(f) = 1$.

Now we have to prove that $\{(r, t_1, \ldots, t_n) | r, t_1, \ldots, t_n$ are ground data-terms and $f(t_1, \ldots, t_n) \rightarrow_R^* r\} \subseteq GndL(Ext(f))$. It is proved on induction on the length k (i.e. number of rewriting steps) of the rewrite derivation $f(t_1, \ldots, !t_n) \rightarrow_R^k r$.
• *for* $k = 1$ we have $f(t_1, \ldots, t_n) = \sigma(f(t'_1, \ldots, t'_n))$ and $f(t'_1, \ldots, t'_n) \rightarrow_R r'$ is a rewrite rule of R, r' is a data term such that $\sigma(r') = r$.

So by definition the following Presburger rewrite rules have been deduced: $\tilde{f}(0; V_R) \rightarrow_{Pbg} r'$ and $\forall i \in [1, n]$ $I_i^f(0; V_R) \rightarrow_{Pbg} t'_i$, so the tuple $(r', t'_1, \ldots, t'_n)$

belongs to $L(Ext(f))$. Since $(r, t_1, \ldots, t_n)$ is a ground instance of $(r', t'_1, \ldots, t'_n)$, it is an element of $GndL(Ext(f))$.

- *if* $k > 1$, we have $f(t_1, \ldots, t_n) \to_R \sigma(C[u \leftarrow f'(t'_1, \ldots, t'_{n'})]) \to_R^{k-1} r$.

  Since u is the only occurrence of $\sigma(C[u \leftarrow f'(t'_1, \ldots, t'_n)])$ labeled by a defined function symbol, we have $f'(t'_1, \ldots, t'_{n'}) \to_R^{k-1} r'$ in one hand and in the other hand $r = \sigma(C[u \leftarrow r'])$.

  By induction hypothesis we have $(r', t'_1, \ldots, t'_{n'}) \in GndL(Ext(f'))$ which means that there exists a ground substitution σ_{cnt} for counter variable such that
  $\sigma_{cnt}(\tilde{f}'(n_{dp(f')}, \ldots, n_1 ; V_R), \overline{I_i^{f'}((n_{dp(f')}, \ldots, n_1 ; V_R))}) \to_{Pbg}^* (r''', t'''_1, \ldots, t'''_{n'})$ and $(r', t'_1, \ldots, t'_{n'})$ is a ground instance of this tuple.

  Since $f(t_1, \ldots, t_n) \to_R \sigma(C[u \leftarrow f'(t'_1, \ldots, t'_{n'})])$ there is a rewrite rule $f(t''_1, \ldots, t''_n) \to_R C[u \leftarrow f(x'_1, \ldots, x'_n)]$ in R.

  If $f = f'$ we have $n = n'$ and $Pbg(R)$ contains the following rules

  - $\tilde{f}(s(n_{dp(f)}), \ldots, n_1 ; V_R) \to_{Pbg} \sigma_{f(\overline{x'_i})}(C)[u \leftarrow \tilde{f}(n_{dp(f)}, \ldots, n_1 ; V_R)$
  - for each $i \in [1, Arity(f)]$, $I_i^f(s(n_{dp(f)}), \ldots, n_1 ; V_R) \to_{Pbg} \sigma_{f(\overline{x'_i})}(t_i)$.

  So, if we consider σ'_{cnt} the ground counter substitution such that for all counter variable n_i but $n_{dp(f)}$, $\sigma'_{cnt}(n_i) = \sigma_{cnt}(n_i)$ and $\sigma'_{cnt}(n_{dp(f)}) = s(\sigma_{cnt}(n_{dp(f)}))$. We have
  $\sigma'_{cnt}(\tilde{f}'(n_{dp(f)}, \ldots, n_1 ; V_R), \overline{I_i^f((n_{dp(f)}, \ldots, n_1 ; V_R))}) \to_{Pbg}$
  $\sigma_{cnt}(\sigma_{f(\overline{x'_i})}(C)[u \leftarrow \tilde{f}(n_{dp(f)}, \ldots, n_1 ; V_R)], \overline{\sigma_{f(\overline{x'_i})}(t_i)})$.
  Let $\sigma = \{x'_i \mapsto t'_i, \forall i \in [1, n]\}$. We have
  $\sigma'_{cnt}(\tilde{f}'(n_{dp(f)}, \ldots, n_1 ; V_R), \overline{I_i^f((n_{dp(f)}, \ldots, n_1 ; V_R))}) \to_{Pbg}^* (\sigma(C[u \leftarrow r']), \overline{\sigma(t_i)})$.
  Since $(r, t_1, \ldots, t_n)$ is a ground instance of $(\sigma(C[u \leftarrow r']), \overline{\sigma(t_i)})$, $(r, t_1, \ldots, t_n) \in GndL(Ext(f))$.

  The case $f \neq f'$ is proved in the same way.

Now we have to define $Ext(f)$ and $GndL(Ext(f))$ for non-recursive function.

Definition 15. *($Ext(f)$ and $GndL(Ext(f))$ for non-recursive functions) Let R be a primal TRS and f a non-recursive function of R.*

- $Ext(f) = \{(r, t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n) \to_R r$ *belongs to* $R\}$.
- $GndL(Ext(f)) = \{(r, t_1, \ldots, t_n) \mid (r, t_1, \ldots, t_n)$ *is a ground instance of an element of* $Ext(f)\}$.

$Ext(f)$ can be seen as a set (i.e. a union) of primal grammars with empty Presburger rewrite system. Now, thanks to the proof framework described section 3, and decidability result on intersection of primal grammars of [10], we claim that unification modulo a primal TRS is decidable and that the set of solutions can be represented be a set of primal grammars. In fact, given a primal TRS R, a goal $s \stackrel{?}{=} t$ is decomposed into elementary subgoals according to section 3.1, the elementary solutions are described using proposition 2 and definition 15 and then the global solution is recomposed according to section 3.3.

For instance, the TRS given in example 1 is a primal TRS so R-unification modulo this TRS is decidable. Here is another example of primal TRS.

Example 3. This example defines the function $first_n$ which takes an integer n as argument and returns the list of the n first integers.
$first_n(0) \rightarrow [0]$,
$first_n(s(x)) \rightarrow [s(x), first_n(x)]$.

Thanks to primal grammars we are able to describe the solutions of the equation $first_n(x) = y$. As far as we know, none of the previous decidability results can handle such a TRS because of the non-linearity of the rhs.

5 Conclusion and Future Work

We have defined a general framework to get decidability results for R-unification problems using tree languages. Its advantages are twofold : on one hand the "user" should only focus on proving that the tree languages of interest can represent solutions of very simple goals (in fact most of the time he has to prove that the tree language can represent the definition in extension of each defined function of the TRS), on the other hand the tree language provides a finite representation of the set of solutions. This can be useful to implement a procedure to handle finitely infinite sets of R-unifiers.

The second result of the paper shows how the proof framework can be instantiated with primal grammars to get a new class of decidable R-unification problems. This result is a strict extension of the one given in [14]. Unfortunately, we have not proved that the restrictions on primal TRS define a kind of border between decidability and undecidability like in [11,7]. In fact it seems that our result may be extended because it does not use all the power of primal grammars, so it would be interesting to study how to use all expressivity of primal grammars. On the other hand, it would be interesting to extend the expressivity of primal grammars, by allowing a finite set of basic Presburger rewrite rules $\tilde{f}(\overline{c}, \overline{x}) \rightarrow_{Pbg} r$ where r is a data term, in order to authorize several basic rewrite rules for primal TRS. But one has to be extremely careful because primal grammars are already very powerful and undecidability is round the corner.

Due to lack of space, we have not presented how we could can weaken restrictions given in [11] using regular tree languages. Indeed the result of [11] is only on restricted goals (i.e. the goal have to be linear). Thanks to our framework, it can be shown that when a variable appears several times in a goal and only one occurrence of this variable is defined by a TTSG (all others being defined by regular tree languages), the R-unification problem is still decidable. Roughly it comes from the fact that intersection between a regular tree grammar and a TTSG is a TTSG.

This framework could also be extended to R-disunfication results [12]. For that one has to define a "anti-join" operation on tree languages.

References

1. A. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proceedings 21st ACM Symposium on Principle of Programming Languages, Portland*, pages 268–279, 1994. 268
2. F. Baader and J. Siekmann. Unification Theory. In D.M. Gabbay, C.J. Hogger, and Robinson J.A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford, UK, 1993. 266
3. A. Bockmayr, S. Krischer, and A. Werner. Narrowing strategies for arbitrary canonical systems. *Fundamenta Informaticae*, 24(1,2):125 – 155, 1995. 266
4. H. Comon. On unification of terms with integer exponents. *Math. Systems Theory*, 28:67–88, 1995. 275
5. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 1997. 267, 268, 269
6. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. Van Leuven, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990. 267
7. H. Faßbender and S. Maneth. A Strict Border for the Decidability of E-Unification for Recursive Functions. In *proceedings of the intern. Conf. on Algebraic and Logic Programming.*, number 1139 in LNCS, pages 194–208. Springer-Verlag, 1996. 280
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19 & 20:583–628, May/July 1994. 266
9. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997. 268
10. M. Hermann and R. Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theoretical Computer Science*, 176, 1997. 267, 269, 273, 274, 279
11. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science (http://www.chapmanhall.com/dm)*, 1:69–98, 1997. 267, 269, 272, 280
12. S. Limet and P. Réty. Solving Disequations modulo some Class of Rewrite System. In *Proceedings of 9th Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 121–135. Springer-Verlag, 1998. 280
13. A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. In *procedings of the 20th Colloquium on Trees in Algebra and Programming, LNCS*, 1995. 275
14. S. Mitra. *Semantic Unification for Convergent Rewrite Systems*. Phd thesis, Univ. Illinois at Urbana-Champaign, 1994. 267, 275, 276, 280
15. R. Caballero Rold n, P. L pez Fraguas, and S nchez Hern ndez J. User's manual for toy. Technical Report 57-97, Departamento de Sistemas Inform ticos y Programaci n, Facultad de Matem ticas (UCM),Madrid, 1997. 268

Operational Versus Denotational Methods in the Semantics of Higher Order Languages

Andrew M. Pitts

Cambridge University Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, UK
www.cl.cam.ac.uk/users/ap/

Abstract. In the last few years increasing use has been made of structural operational semantics to study aspects of programming languages which traditionally have been analysed via denotational semantics. (The articles in the recent collection by Gordon and Pitts (1998) [4] are a good illustration of this development and its applications.) Since there are more or less adequate denotational models for programming language features such as higher order functions and procedures, recursive definitions, local state, and data abstraction, one might wonder why one should consider syntactic methods at all. This tutorial talk will attempt to explain why one should.

> *'But once feasibility has been checked by an operational model, operational reasoning should be immediately abandoned; it is essential that all subsequent reasoning, calculation and design should be conducted in each case at the highest possible level of abstraction.'* Hoare (1996, page 182) [6]

The advice in the second half of the above quotation—that one should always strive for the highest possible level of abstraction—is very sound. Nevertheless, I am going to try to persuade you to ignore the first half and embrace, rather than abandon, operational reasoning. The justification for this lies in the claim that operational semantics can (sometimes—we are still learning how) be presented at a sufficient 'level of abstraction' to support quite palatable methods of reasoning, calculation and design. I plan to cover some of the following topics.

- Structural operational semantics—a brief survey, including the use of *structural congruence relations*, arising from work in concurrency theory (Berry and Boudol 1992 [1], Milner 1990 [8]), to simplify the auxiliary syntactic structures (environments, memories, program stacks, multisets of program threads, etc) needed in a semantic specification.
- The use of *evaluation contexts* to aid inductive reasoning about transition relations (Felleisen and Heib 1992 [3], Wright and Felleisen 1994 [13], Harper and Stone 1996 [5]).
- Operational analogues of the domain-theoretic concepts of *least fixed points, continuity*, and the *minimal invariant* property of recursively defined domains (Mason, Smith and Talcott 1996 [7], Smith 1998 [12]).
- Applications of operationally-based *logical relations* (Pitts 1997 [9], Birkedal and Harper 1997 [2], Pitts and Stark 1998 [11], Pitts 1998 [10]).

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 282–283, 1998.

References

1. Berry, G. and G. Boudol (1992). The chemical abstract machine. *Theoretical Computer Science 96*. 282
2. Birkedal, L. and R. Harper (1997). Relational interpretation of recursive types in an operational setting (Summary). In M. Abadi and T. Ito (Eds.), *Theoretical Aspects of Computer Software, 3rd Int. Symp.*, Volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin. 282
3. Felleisen, M. and R. Hieb (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 235–271. 282
4. Gordon, A. D. and A. M. Pitts (Eds.) (1998). *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press. 282
5. Harper, R. and C. Stone (1996). A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, Carnegie Mellon University, Pittsburgh, PA. 282
6. Hoare, C. A. R. (1996). Algebra and models. In I. Wand and R. Milner (Eds.), *Computing Tomorrow. Future research directions in computer science*, Chapter 9, pp. 158–187. Cambridge University Press. 282
7. Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From operational semantics to domain theory. *Information and Computation 128*(1), 26–47. 282
8. Milner, R. (1990). Functions as processes. In *Proc. 17th Int. Coll. on Automata, Languages and Programming*, Volume 443 of *Lecture Notes in Computer Science*, pp. 167–180. Springer-Verlag, Berlin. 282
9. Pitts, A. M. (1997). Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent (Eds.), *Algol-Like Languages*, Volume 2, Chapter 17, pp. 173–193. Birkhauser. 282
10. Pitts, A. M. (1998). Existential types:logical relations and operational equivalence. In *Proceedings ICALP'98*, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin. 282
11. Pitts, A. M. and I. D. B. Stark (1998). Operational reasoning for functions with local state. In (Gordon and Pitts, 1998). 282
12. Smith, S. F. (1998). The coverage of operational semantics. In (Gordon and Pitts, 1998). 282
13. Wright, A. K. and M. Felleisen (1994). A syntactic approach to type soundness. *Information and computation 115*, 38–94. 282

Functional Implementations of Continuous Modeled Animation

Conal Elliott

Microsoft Research
Graphics Group
http://www.research.microsoft.com/~conal

Abstract. Animation is a temporally *continuous* phenomenon, but is typically programmed in terms of a discrete sequence of changes. The use of discreteness serves to accommodate the machine that is *presenting* an animation, rather than the person *modeling* an animation with the help of a computer. Using a continuous model of time for animation allows for natural specification, avoiding some artificial details, but is difficult to implement with generality, robustness and efficiency. This paper presents and motivates *continuous modeled animation*, and sketches out a naive functional implementation for it. An examination of some of the practical problems with this implementation leads to several alternate representations, all of which have difficulties in themselves, some quite subtle. We hope that the insights and techniques discussed in this paper lead to still better representations, so that animation may be specified in natural terms without significant loss of performance.

1 Introduction

A functional approach to animation offers the possibility to make animations much easier and more natural to express, by removing the need to direct the details of *presentation* and allowing the programmer instead to say what an animation is [1,8]. Following the evolution of 3D graphics, we have termed this approach "modeling", as opposed to "presentation" [4].

Given that we want to model animations, what notion of time should we adopt? The first fundamental choice is discrete vs continuous, that is do we think of time as moving forward in a (discrete) sequence of "clock ticks", or a (continuous) flow?

A discrete model of time fits more easily into popular *computational* concepts, because modern computers operate in temporally discrete steps. More specifically, animations are presented as a sequence of frames, delivered to the user at a finite rate.

On the other hand, a continuous model of time seems a more natural fit with our human perception of time. Moreover, we have a rich heritage of mathematical, scientific, and engineering tools for understanding and describing the basic animation concepts of motion, growth, etc, and these tools are based on the continuous notion of time. These tools include polynomials, trigonometry, and

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 284–299, 1998.

calculus, together with their applications to physical motion, governed by Newton's law of motion and its applications to gravitational attraction, springs, etc.

Just as continuous image models such as Postscript [11] naturally give rise to spatial resolution-independence and hence scalability, the continuous time model for animation yields *temporal* resolution-independence and scalability. For instance, one can stretch or squeeze a given animation to fit a desired duration.

Fran (*Functional reactive animation*) is a theory of animation, and implementation as a Haskell library, that is based on a continuous time model. As such, it offers the possibility of allowing people to express animations in ways that match their intuitive understandings and that leverage the wealth of mathematical tools at their disposal.

The cost of offering a continuous time model is a more challenging implementation problem, since continuous animation descriptions must be translated into discrete lower-level software and hardware structures.[1] In fact, we have implemented Fran many times, and have not yet achieved a satisfactory combination of generality, robustness and efficiency. Several subtle difficulties became apparent to us only through implementation and experimentation.

The purpose of this paper is to present the implementation techniques we have explored, together with some insight into practical difficulties, in the hope of facilitating work leading to still better representations.

An expanded version of this paper presents a few more implementation alternatives that we have investigated, but have implemented only partially or not at all [7].

Previous papers on Fran have presented its vocabulary and semantics, its role as an "embedded language", and examples of its use [8,5,4,14]. This paper gives only minimal treatment of these issues.

2 A User Perspective on Fran

Fran is a declarative animation library (or "embedded language") that embodies the continuous time model of animation, and is implemented in Haskell [9].

2.1 Behaviors

For any type `ty`, the Fran type `Behavior ty` represents `ty`-valued animations, i.e., functions from continuous time to `ty`. As an example, consider the following definition of a real-valued animation, which has value 1 at time 0 and grows quadratically:

[1] The translation from continuous to discrete inevitably introduces error. Usually these errors are negligible, but sometimes, as in some systems of differential equations or condition-based events, errors can become significant. These problems are inherent in applying computers to simulate continuous phenomena, regardless of the programming paradigm.

```
type RealB = Behavior Double -- helpful synonym

growQuad :: RealB
growQuad = 1 + time ** 2
```

There is some notational magic going on here. We are using behavior-specific overloadings of familiar numerical operators for addition and exponentiation, and even numeric literals, using Haskell's type class mechanism. The types used in this example are as follows:

```
1, 2 :: RealB
(+), (**) :: RealB -> RealB -> RealB
time :: RealB
```

Literals like 1 and 2 are implicitly converted into behaviors through application of the polymorphic `constantB` function:

```
constantB :: a -> Behavior a
```

For any value `x`, the behavior `constantB x` has value `x` at every time.

The behavior versions of functions like `+` and `**`, as well as `sin`, `log`, etc., are all defined in terms of the non-behavior versions, by using lifting operations.

```
lift1 :: (a -> b) ->
         Behavior a -> Behavior b
lift2 :: (a -> b -> c) ->
         Behavior a -> Behavior b -> Behavior c
lift3 :: (a -> b -> c -> d) ->
         Behavior a -> Behavior b -> Behavior c -> Behavior d
-- etc
```

Lifting is hidden inside of type class instances, with declarations like the following:

```
instance Num a => Num (Behavior a) where
(+) = lift2 (+)
(*) = lift2 (*)
negate = lift1 negate
abs = lift1 abs
fromInt n = constantB (fromInt n)
```

The first line says that for any "number type" `a`, the type of `a`-valued behaviors is also a number type. The second line says that the behavior version of `+` is the binary-lifted version of the unlifted `+` operation. (The definitions appear self-referential, but are not. Overload resolution distinguishes the two different versions of "`+`".) The function `fromInt` is used to resolve literal integer overloading. The reason for this definition, instead of the more obvious one involving `lift1`, is that `fromInt` must still work on numbers, not number-valued behaviors. This requirement is partly desirable, but is also partly due to the restricted type of `fromInt`. In such cases, Fran provides additional definitions with names formed by adding "`B`" or "`*`" to the unlifted version, such as the following.

```
fromIntB :: Num a => Behavior Int -> Behavior a
fromIntB = lift1 fromInt

(==*) :: Eq a => Behavior a -> Behavior a -> Behavior Bool
(==*) = lift2 (==)
```

Because `Behavior` is a type constructor, we can apply it to any type. For instance, when applied to a type `Point2` of 2D static points, the result is the type of motionsn in 2D space.

It is often quite natural to express behaviors in terms of time-varying velocities. For this reason, Fran supports integration over a variety of types, as long as they implement vector space operations.

```
integral :: VectorSpace v => Behavior v -> User -> Behavior v
```

Examples of vector space types are reals, 2D vectors, and 3D vectors. The "user" argument to `integral` has two roles. One is to choose the integral's start time, and the other is to assist choice of step-size in the numerical solution. If a user `u` has start time t_0, then `integral b u` is a behavior that at time t has approximately the value $\int_{t_0}^{t} b(t')dt'$.

For modularity, it is useful to construct a behavior and then, separately, transform it in time. Fran, therefore, has a time transformation operation. Since a time transform remaps time, it is specified as a time-valued behavior, which semantically is a function from time to time.

```
timeTransform :: Behavior a -> Behavior Time -> Behavior a
```

The lifting operators can all be expressed in terms of a single operator:

```
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
```

The name "`$*`" comes from the fact that it is the lifted version of function application, whose Haskell infix operator is "`$`".

The lifting operators are defined simply in terms of `constantB` and "`$*`", as follows.

```
lift0            = constantB
lift1 f b1       = lift0 f $* b1
lift2 f b1 b2    = lift1 f b1 $* b2
lift3 f b1 b2 b3 = lift2 f b1 b2 $* b3
-- etc
```

Note that the basic combinators "`$*`", `constantB`, `time`, and `timeTransform` correspond semantically to type-specialized versions of the classic S, K, I, and B combinators.

2.2 Events

Besides behaviors, the other principle notion in Fran is the *event*. For any type `ty`, the Fran type `Event ty` is the type of `ty`-valued events, each of which is semantically a time-sorted sequence of time/`ty` pairs, also refered to as "occurrences" of the event.

This paper is chiefly concerned with the implementation of continuous *behaviors*, rather than events (which are intrinsically discrete). We will informally describe event operators as they arise.

2.3 Reactive Behaviors

While one can define an infinite set of behaviors using just the behavior combinators given above, such behaviors are not very dynamic. Fran greatly enriches these behaviors by supporting reactivity, via the following primitive, which uses behavior-valued events.

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
```

Given behavior `b` and an event `e`, the behavior `b ‘untilB‘ e` acts like `b` until `e` occurs, and then switches to acting like the behavior that accompanies the event occurrence. Although `untilB` only needs the *first* occurrence of an event, the other occurrences are used by some of the event combinators. This meaning of events turned out to be more convenient than the one in [8], since it enables higher level combinators for constructing reactive behaviors, as illustrated in [4].

3 Implementing Continuous Behaviors

We now turn to the main thrust of our paper, which is an exploration of how to implement continuous behaviors.

Representation A: time-to-value functions. The semantics of behaviors suggest a very simple representation:

```
newtype Behavior a = Behavior (Time -> a)

-- Sample a behavior "at" a given time
at :: Behavior a -> Time -> a
Behavior f ‘at‘ t = f t
```

This definition introduces both a new type constructor and a value constructor, both called `Behavior`. (A simple `type` definition in Haskell would not have allowed overloading.) Using this representation, it is easy to define our simple behavior combinators:

```
constantB x = Behavior (const x)

fb $* xb = Behavior (\ t -> (fb `at` t) (xb `at` t))

time = Behavior (\ t -> t)

timeTransform b tt = Behavior (at b . at tt)
```

To sample a behavior constructed by `b 'untilB' e` at a time t, first check whether the event `e` occurs before t.[2] If so, sample the new behavior, `b'`, that is part of the event occurrence, and if not, sample `b`.

```
b `untilB` e = Behavior sample
  where
    sample t = case (e `occ` t) of
                 Nothing -> b `at` t
                 Just b' -> b' `at` t
```

Here we presume a function `occ`, with the following signature, for checking for an event occurrence before a given time.

```
occ :: Event a -> Time -> Maybe a
```

3.1 The Problem of Non-incremental Sampling

While Representation A given above is appealing in terms of simplicity, it has a serious performance problem. It allows nothing to be remembered from one sampling to another. In animation, however, behaviors are typically sampled at a sequence of display times separated by small differences. For instance, given an integral behavior, it is vital for efficiency that intermediate results are carried from one sampling to the next, so that only a small amount of extra work is required. Similarly, for a *reactive* behavior, i.e., one constructed with `untilB`, it is important to make incremental progress in event detection.

Consider the implementation of `untilB` above, and suppose we want to sample a reactive behavior with times $t_1, t_2, \ldots$. For every t_i, sampling must consider whether the event has occurred, and choose to sample the old behavior or the new one. Moreover, it is frequently the case that the new behavior it itself reactive, and so on, in an infinite chain. In such a case, the time it takes to compute each sample will increase without bound, which is clearly unacceptable. Moreover, consider what is involved in computing "`e 'occ' t`" in `untilB`. There are two possibilities: either event occurrences are being rediscovered, or they are cached somehow. If they are rediscovered, then the cost of sampling increases even more so with each sampling. If, however, they are cached, then the cache itself becomes a large space leak. Together these problems cause what we call a "space-time leak".

[2] By choosing *before*, rather than before *or at*, the sampling time t, animations may be self- or mutually-reactive.

Representation B: residual behaviors. A crucial observation is that typically, the sampling times t_1, t_2, ..., are monotonically increasing. If we assume that this typical case holds, then we can remove the space-time leak.[3] The idea is to have sampling yield not only a value, but also a "residual behavior".

```
newtype Behavior a = Behavior (Time -> (a, Behavior a))

at :: Behavior a -> Time -> (a, Behavior a)
Behavior f `at` t = f t
```

The combinator implementations are not quite as simple as before, but still reasonable. Constant behaviors always return the same pair:

```
constantB x = b
where
b = Behavior (const (x, b))
```

The `time` behavior is also simple.

```
time = Behavior (\ t -> (t, time))
```

The "`$*`" combinator samples its argument behaviors, applies one resulting value to the other, and applies "`$*`" to the residual behaviors.

```
fb $* xb = Behavior sample
 where sample t = (f x, fb' $* xb')
         where (f, fb') = fb `at` t
               (x, xb') = xb `at` t
```

Time transformation can be implemented much like "`$*`".

```
timeTransform b tt = Behavior sample
 where sample t = (x, timeTransform b' tt')
         where (t', tt') = tt `at` t
               (x , b' ) = b `at` t'
```

Time transformation can violate our monotonicity assumption for time streams, if the time transform `tt` is not itself monotonic. It would be possible to check for monotonic sampling dynamically, at least for reactive behaviors, though Fran does not do so. Checking monotonicity "statically", i.e., when a `timeTransform` or `untilB` behavior is constructed does not seem to be feasible.

The `occ` function, used for checking event occurrences, is changed in a way much like behaviors, so that it now yields a residual event along with a possible occurrence.

```
occ :: Event a -> Time -> (Maybe a, Event a)
```

[3] Unfortunately, the cost of this assumption is a significant restriction in the time transforms that may be applied to reactive behaviors.

Next, consider reactivity. Sampling a reactive behavior with a time that is after the event's first occurrence, the newly constructed behavior (a) no longer checks for the event occurrence, thus eliminating the time leak, and (b) no longer holds onto the old behavior, thus eliminating the space leak.

```
b `untilB` e = Behavior sample
 where
   sample t =
     case (e `occ` t) of
       -- No occurrence before t; keep looking
       (Nothing, eNext) -> let (x, bNext) = b `at` t in
                           (x, bNext `untilB` eNext)

       -- Found it; sample new behavior
       (Just bNew, _) -> bNew `at` t
```

Representation C: stream functions. An alternative solution to the problem of non-incremental sampling is to map time streams to value streams. We will see in Section 3.2 that this representation has advantages over Representation B.

```
newtype Behavior a = Behavior ([Time] -> [a])

at :: Behavior a -> [Time] -> [a]
Behavior f `at` ts = f ts
```

Constant behaviors always return the same list containing an infinite repetition of a value:

```
constantB x = Behavior (const (repeat x))
```

The "`$*`" combinator samples the function and argument behaviors, and applies resulting functions to corresponding arguments, using the binary mapping functions `zipWith`.

```
fb $* xb =
  Behavior (\ ts -> zipWith ($) (fb `at` ts) (xb `at` ts))
```

Time is the identity as it was in Representation A, but of a different type:

```
time = Behavior (\ ts -> ts)
```

Reactivity is implemented by scanning through a list of possible event occurrences, while enumerating behavior samples. For convenience, assume a stream sampler function for events:

```
occs :: Event a -> [Time] -> [Maybe a]
```

The implementation of `untilB`:

```
b `untilB` e =
  Behavior (\ ts -> loop ts (b `at` ts) (e `occs` ts))
 where
  -- Non-occurrence. Emit first b sample and continue looking
  loop (_:ts') (x:xs') (Nothing:mbOccs') =
     x : loop ts' xs' mbOccs'
  -- First event occurrence. Discard the rest of the b values
  -- and possible event occurrences, and continue with the
  -- new behavior
  loop ts _ (Just bNew : _ ) = bNew `at` ts
```

A weakness of Representations B and C is that they cause a great deal of construction with each sampling. For this reason, it is important to have a garbage collector that deals very efficiently with the rapid production of short-lived structures, as in generational garbage collection. For instance, the Glasgow Haskell Compiler [12] has such a collector.

3.2 The Problem of Redundant Sampling

Another serious problem with all the representations preceding is that they lead to redundant sampling. As a simple example, consider the following behavior that linearly interpolates between two numerical behaviors `b1` and `b2`, according to the interpolation parameter `a`, so that the resulting behavior is equal to `b1` when `a` is zero and `b2` when `a` is one.

```
interp :: RealB -> RealB -> RealB -> RealB
interp b1 b2 a = (1 - a) * b1 + a * b2
```

The problem is that sampling the behavior generated by `interp` at some time t ends up sampling `a` at t twice, due to the repeated use of `a` in the body of `interp`. If `a` is a complex behavior the redundant sampling is costly. Worse yet, composition of functions like `interp` multiplies the redundancy, as in the following example.

```
doubleInterp :: RealB -> RealB -> RealB -> RealB -> RealB -> RealB
doubleInterp b1 b2 b3 b4 a = interp b1 b2 a'
  where a' = interp b3 b4 a
```

When the result of `doubleInterp` is sampled with a time t, the behavior `a'` will be sampled twice at t, which means `a` will be sampled four times, and the interpolation work for `a'` done twice. If we were to cube the result of `doubleInterp`, via the definition

```
cube :: Num a => a -> a
cube x = x * x * x
```

then sampling work would be multiplied by three, causing `a` to be evaluated twelve times for each sample time t.

One approach to solving the problem of redundant sampling is applying lazy memoization [10,3], which may be supported with a function of the form type.

```
memo :: Eq a => (a -> b) -> (a -> b)
```

Semantically, `memo` is the identity. Operationally, the closure returned contains a mutable "memo table".

Any of the three representations discussed so far may be memoized. For instance, in Representation B one could simply memoize the constructed sampling functions. By way of example, here is an appropriately modified "`$*`". The only change is that the created `sample` function is memoized.

```
fb $* xb = Behavior (memo sample)
 where sample t = (f x, fb' $* xb')
         where (f, fb') = fb `at` t
               (x, xb') = xb `at` t
```

A drawback to this implementation is that the memoization overhead must be paid for each sample time of each component behavior, and so slows down sampling, rather than speeding it up, except in circumstances of extreme redundant sampling.

A more efficient alternative would to start with Representation C, so that memoization works at the level of *lists* of times rather than individual times. Rather than base memoization on the usual elementwise notion of list equality, which would be particularly problematic because our time lists are infinite, it suffices to use pointer equality on the list representations, as recommended in [10].

This is the representation used in the current version of Fran (1.11), except that memo tables are managed explicitly, rather than through a higher-order `memo` function. The reason for this exception is that memo tables need to be "aged", as will explained below.

Our algebra of behaviors is related to, and in some ways inspired by, Backus' language FP [2]. In FP, programs are always expressed at the function level, with application to "object-level" values kept implicit. This property leads to redundant applications of a function to the same argument, similar to the problem discussed in this section. The Illinois Functional Programming Interpreter [13] addressed this problem by using an "expression cache." For some recursive algorithms, expression caching reduced the asymptotic running time. Normally this caching had a more modest effect, speeding up some computations and slowing down others.

3.3 The Problem of Space-Leaking Memo Tables

A subtle but important consideration for any use of memoization is garbage collection. The memoized functions contain tables of domain and range values, and

these tables typically grow without bound. When all references to a memoized function are lost, the contents of its memo table are reclaimed, as described in [3]. For example, consider the following behavior, which uses the `cube` and `interp` functions defined above to stretch a given picture. (The Fran type `ImageB` represents "image-valued behaviors", i.e., two-dimensional animations.)

```
anim1 :: ImageB
anim1 = stretch s1 pic
 where
   s1 = cube (interp 0.5 1.5 (cube time))
```

If `anim1` is displayed, it will be sampled with a time list that depends on the speed of the machine and the time-varying load from other processes. As long as there is no reference to `anim1` besides the one given to the display function, the representations of `anim1` and `s1`, including memo tables will be reclaimed. The list cells in the time and value lists will also be eligible for reclamation in an efficient manner as animation display progresses.

Unfortunately, memory use is not always so well-behaved. One problematic case is that in which the definition of `anim1` is a top-level definition, also known as a "CAF" (constant applicative form). In that case, in the current Haskell implementations we know of, the behaviors will never be reclaimed.[4]

Another problematic situation for reclamation of memo tables arises when a behavior is retained in order to be restarted, as in the following example, which makes a looping behavior out of `s1`, restarting every two seconds.[5]

```
anim2 :: ImageB
anim2 = stretch s2 pic
 where
   s1 = cube (interp 0.5 1.5 (cube time))
   s2 = s1 `untilB` timeIs 2 -=> later 2 s2
```

Here we have used the `later` operator to shift `s2` by two seconds. (Alternatively, the semantics of `untilB` could do this kind of shifting automatically. Although there are some technical difficulties, we intend to alter Fran in this way. The implementation issue discussed below remains, however.) In this case, `s1` cannot released, because it will be used every two seconds to generate a new list of scale values. For each memo table, an entry is added every two seconds, and the evaluated size of each entry grows through lazy evaluation by a time, a scale, and two cons cells every sample, e.g., 10 per second.

For these reasons, it seems necessary for memo tables to have special support from the garbage collector, so that when there are no live pointers to an object other than memo table keys, then the object gets reclaimed and the memo table entries get deleted. This idea is described in [10] and has been in use in some Smalltalk, Lisp and Scheme implementations for quite a while, but as far as we

[4] An upcoming release of the Glasgow Haskell Compiler fixes this problem.

[5] The event "`e -=> v`" occurs whenever `e` occurs and has value `v` at each occurrence. Syntactically, it binds more tightly than `` `untilB` ``.

know it has not yet been implemented in a Haskell run-time system. Some Fran programs leak space for exactly this reason.

A more serious, and more subtle, problem with memoization comes from Fran's `afterE` combinator. This combinator is essentially a dual to `snapshot`. The two signatures are as follows:

```
snapshot :: Event a -> Behavior b -> Event (a, b)
afterE :: GBehavior bv => Event a -> bv -> Event (a, bv)
```

The idea of `e ‘snapshot‘ b` is to pair up the event data from each occurrence of `e` with snapshots of `b` values at the corresponding occurrence time. Dually, `e ‘afterE‘ b` gives access to the *residual* of `b` at each occurrence of `e`. Its purpose is to be able to coordinate with a concurrently running behavior after an event occurrence, but fortunately its use can often be hidden inside of other event and behavior combinators. Because `afterE` need not actually sample, it can be used with "generalized behaviors", a notions that subsumes the actual behavior types.

Fran implements `afterE` in terms of the following more primitive "aging" function:

```
afterTimes :: GBehavior bv => bv -> [Time] -> [bv]
```

The idea is to create a list of times corresponding not only to occurrences of an event, but to closely spaced *non-occurrences* as well. This list is fed into `afterTimes`, which then produces a stream of updated versions of its generalized behavior argument. (Note: it may well be possible and desirable to hide `afterE` beneath a set of more abstract combinators, but the implementation issues remain.)

Now consider the interaction between `afterTimes` and memoization. Suppose we have a behavior `bv` with a memo entry for the time stream $t_1, t_2, \ldots$ and corresponding value stream $x_1, x_2, \ldots$, and we have another time stream $t'_1, t'_2, \ldots$ as an argument to `afterTimes`. How should we initialize the aged behaviors' memo tables? If we use `bv`'s memo table, we will have a space leak, and, moreover, these entries are not likely to get cache hits. If, on the other hand, we start with empty memo tables, we will end up repeating a lot of work. A crucial observation is that these aged behaviors are often sampled with *aged sample time streams*, i.e., suffixes of the time streams that have already been used to sample `bv` itself. Rather than reusing or discarding memo tables in their entirety, the current Fran implementation "ages" the tables, replacing each time- and value-stream pair with corresponding stream suffixes. For instance, if $t_2 < t'_1 \leq t_3$, then the first aged memo pair would be $t_3, t_4, \ldots$ and $x_3, x_4, \ldots$.

Note that if we were to memoize Representation A or B instead of C, then it would become trickier to use a garbage collector to trim the memo tables. If the `Time` type is something like `Float` or `Double` (as in Fran's implementation), then we could easily keep a time/sample pair alive in a memo table by accident. If indeed these accidental retentions happen, a solution would be to introduce a data type to wrap around the time values.

3.4 The Problem of Synchrony

Memoization solves the problem of redundant sampling, but only when the different uses of a behavior are sampled with exactly the same time streams. There are, however, at least three situations in which we would want a behavior to be sampled with different time streams.

The first situation is the application of a time transformation. Consider the following example.

```
b :: RealB
b = b1 + timeTransform b1 (time / pi)
 where
   b1 = cube (interp 0.5 1.5 (cube time))
```

There is a second situation in which we might like to sample a behavior with two different rates, namely when different degrees of precision are needed. As an example, suppose we have an image `im1` moving in a complex motion path `mot`, with an overlaid, much smaller copy of itself:

```
im :: ImageB
im = stretch 0.1 im2 `over` im2
 where
   im2 = move mot im1
```

If we were sampling in order to stay within a error bound measured in 2D distance, rather than *temporal* rate, then the first use of `im2` would require less accuracy from `mot` and `im1` than the second use, because the spatial inaccuracies are reduced by a factor of ten.

A third situation calling for variable sampling rate is detection of events defined by boolean behaviors. As discussed in [8], interval analysis (IA) can be used in a robust nand efficient algorithm to detect occurrences of such events. The sampling patterns are adaptive to the nature of the condition's constituent behaviors.

In all of these cases, lack of synchrony disallows sharing of work between different sampling patterns. We do not have a solution to this problem. Note, however, that the values used for display in between samplings are necessarily approximate. (Fran uses a linearly interpolating engine [6].) As explored in [7], this observation suggests sharing of work among non-synchronous sampling patterns.

3.5 Structural Optimizations

Fran's algebra of behaviors and events satisfies several algebraic properties that are exploited for optimization. Roughly speaking, these identities fall into the categories of "static" and "behavioral" properties.

By a "static" property, we mean one that is lifted to the level of behaviors directly from a property on static values. Examples include the following identity

and distributive properties on images, where `over` is the (associative but not commutative) image overlay operation and `*%` applies a 2D transform to an image.

```
emptyImage `over` im == im

im `over` emptyImage == im

xf *% (im1 `over` im2) == xf *% im1 `over` xf *% im2
```

Another useful property is following one for `condB`, which is the lifted form of if-then-else:

```
condB (constantB True ) b c == b

condB (constantB False) b c == c
```

By a "behavioral" property, we mean one that applies to behaviors over all types. For example, when "`$*`" is applied to constant behaviors, the result is a constant behavior, i.e.,

```
constantB f $* constantB x == constantB (f x)
```

As a consequence, a lifted n-ary function applied to n constant behaviors yields a constant behavior.

Another useful property is distributivity of lifted functions over reactivity. First consider a simple case.

```
lift1 f (b `untilB` e) == lift1 f b `untilB` e ==> lift1 f
```

The event `e ==> h` occurs when `e` occurs, and its occurrence values result from applying the function `h` to the corresponding value from `e`. Syntactically, it binds more tightly than `` `untilB` ``.

Here is an obvious candidate for the general case:

```
fb $* (xb `untilB` e) ==
  fb $* xb `untilB` e ==> \ xb' ->
  fb $* xb'
```

and similarly for the case that `fb` is reactive. If both argument behaviors are reactive, then both rules may be applied sequentially (in either order). Similarly, if in the first rule, `xb` itself is reactive, applying the rule will give rise to another rule application as well.

There is, however, an operational problem with a rule like the one above, namely that it holds onto the behavior `fb` while waiting for the event `e` to occur, thus causing a space leak. Then when `e` finally occurs, `fb` will get sampled starting at the occurrence time. If `fb` has meanwhile undergone many transitions, there may be a lot of catching up to do, which amounts to a "time leak". To fix both of these problems, use `afterE` to give quick access to the residual of `fb`.

```
fb $* (xb `untilB` e) ==
  fb $* xb `untilB` e `afterE` fb ==> \ (xb',fb') ->
  fb' $* xb'
```

The identities above, while widely applicable, do not improve performance by themselves. Their merit is that they often enable other optimizations. For instance, consider the following animation:

```
b :: RealB
b = (time `untilB` timeIs 5 -=> 0) + b2
```

Applying the `$*/untilB` identity yields the following:

```
time + b2 `untilB` (timeIs 5 -=> 0) `afterE` b2 ==>
   \ (b1', b2') -> b1' + b2'
```

When the transition occurs at time 5, the new behavior will be `0 + b2'`, which can be simplified to `b2'`.

4 Conclusions

Modern software and hardware technology are temporally discrete in nature and so encourage discrete software models. In the context of animation, a continuous approach is more natural because it more closely reflects the real-world behaviors being modeled. In this paper, we have explored several functional implementations of continuous animation and some problems that arise. Some of these problems are rather subtle and became apparent only through costly trial and error. See [7] for more alternatives that we have partially explored. We hope that their discussion here will motivate further work in pursuit of the goals of efficiently-executing, naturally-specified interactive animation.

Acknowledgements

Discussions with Sigbjorn Finne and Simon Peyton Jones helped to clarify operational properties of the representations discussed in this paper. Extensive comments from anonymous reviewers greatly improved the presentation.

References

1. Kavi Arya. A functional approach to animation. *Computer Graphics Forum*, 5(4):297–311, December 1986. 284
2. John Backus. Can programming be liberated from the von Neumann style? *Comm. ACM*, 8:613–641, 1978. 293
3. Byron Cook and John Launchbury. Disposable memo functions (extended abstract). In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, page 310, Amsterdam, The Netherlands, June 1997. 293, 294

4. Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *The Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997. USENIX. WWW version at `http://-www.research.microsoft.com/~conal/papers/dsl97/dsl97.html`. 284, 285, 288
5. Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, pages 18–33, July 1998. Expanded version with animated GIFs: `http://www.research.-microsoft.com/~conal/fran/{tutorial.htm,tutorialArticle.zip}`. 285
6. Conal Elliott. From functional animation to sprite-based display (expanded version). Technical Report MSR-TR-98-28, Microsoft Research, July 1998. `http://-www.research.microsoft.com/~conal/papers/spritify/long.ps`. 296
7. Conal Elliott. Functional implementations of continuous modeled animation (expanded version). Technical Report MSR-TR-98-25, Microsoft Research, June 1998. `http://www.research.microsoft.com/~conal/papers/plilpalp98/-long.ps`. 285, 296, 298
8. Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997. 284, 285, 288, 296
9. John Peterson et. al. Haskell 1.3: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996. Current WWW version at `http://haskell.org/-report/index.html`. 285
10. John Hughes. Lazy memo functions. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer Verlag, September 1985. 293, 294
11. Adobe Systems Incorporated. *POSTSCRIPT Language: Tutorial and CookBook*. Addison-Wesley Publishing Company, Inc, 1991. 285
12. Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93. 292
13. Arch D. Robinson. The Illinois functional programming interpreter. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 64–73. ACM, ACM, June 1987. 293
14. Simon Thompson. A functional reactive animation of a lift using Fran. Technical Report 5-98, University of Kent, Computing Laboratory, May 1998. `http://www.cs.ukc.ac.uk/pubs/1998/583/index.html`. 285

Compiling Erlang to Scheme

Marc Feeley and Martin Larose

Université de Montréal
C.P. 6128 succursale centre-ville
Montréal H3C 3J7, Canada
{feeley,larosem}@iro.umontreal.ca

Abstract. The programming languages Erlang and Scheme have many common features, yet the performance of the current implementations of Erlang appears to be below that of good implementations of Scheme. This disparity has prompted us to investigate the translation of Erlang to Scheme. Our intent is to reuse the mature compilation technology of current Scheme compilers to quickly implement an efficient Erlang compiler. In this paper we describe the design and implementation of the **Etos** Erlang to Scheme compiler and compare its performance to other systems. The Scheme code produced by Etos is compiled by the Gambit-C Scheme to C compiler and the resulting C code is finally compiled by `gcc`. One might think that the many stages of this compilation pipeline would lead to an inefficient compiler but in fact, on most of our benchmark programs, Etos outperforms all currently available implementations of Erlang, including the Hipe native code compiler.

1 Introduction

Erlang is a concurrent functional programming language which has been mostly developed internally at Ericsson for the programming of telecom applications. The language is not purely functional because of its support for concurrent processes and communication between processes. Scheme shares many similarities with Erlang: "mostly" functional programming style, mandatory tail-call optimization, dynamic typing, automatic memory management, similar data types (symbols, lists, vectors, etc). Section 2 and Sections 3 briefly describe these languages (a complete description can be found in [3] and [16,7]).

There is growing interest in Erlang in industry but due to its "in-house" development there is a limited choice of compilers. As the implementors of these compilers freely admit [1], "Performance has always been a major problem". On the other hand there are many implementations of Scheme available [18] and the good compilers appear to generate faster code than the Erlang compilers available from Ericsson (for example Hartel *et al.* [13] have shown that the "pseudoknot" benchmark compiled with Ericsson's BEAM/C 6.0.4 is about 5 times slower than when compiled with the Gambit-C 2.3 Scheme compiler).

Because of the strong similarity between Erlang and Scheme and the availability of several good Scheme compilers, we have begun the implementation of

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 300–318, 1998.

an Erlang to Scheme compiler called "Etos". Our goal is to reduce development efforts by exploiting the analyses and optimizations of the Gambit-C Scheme to C compiler. It is reasonable to believe that most of the Gambit-C technology can be reused because the similarities between the languages outweigh the differences (infix vs. prefix syntax, pattern matching vs. access functions, `catch`/`throw` vs. `call/cc`, and concurrency). When we started this project it was not clear however if the many stages of the compilation pipeline would allow efficient code to be generated. In the rest of the paper we explain the major design issues of an Erlang to Scheme compiler and how these are solved in Etos 1.4, and show that its performance is very good when compared to other Erlang compilers.

2 Scheme

This section briefly describes Scheme for those unfamiliar with the language.

Scheme is a lexically scoped dialect of Lisp (invented by Sussman and Steele in 1975 [19] and enhanced regularly since then) which is both small and expressive. It is an expression-based language with garbage-collection and so promotes the functional programming style (but side-effects on variables and data-structures are permitted). The language requires that tail-recursion be implemented properly [6]. Several builtin data types are available, all of which are first-class and have indefinite extent: boolean, character, string, symbol, list, vector (one dimensional array), procedure (of fixed or variable arity), port (file handle), number (unlimited precision integers and rationals (i.e. exact numbers), and floating point and complex numbers). Procedures are closed in their definition environment (i.e. they are "closures" containing a code pointer and environment) and parameters are passed by value. An anonymous procedure is created by evaluating a `lambda` special form (see example below). Scheme is one of the few languages with first-class continuations which represent the "rest of a computation" and a construct, `call/cc`, to transform the current (implicit) continuation into a user-callable procedure. All arithmetic functions are generic, e.g. the addition function can be used to add any mix of number types.

3 Erlang

Erlang, like Scheme, is a garbage-collected expression-based language that is lexically scoped (but with unusual scope rules as explained in Section 8), properly tail-recursive, dynamically typed and which uses call-by-value parameter passing. The data types available are: number (floating-point numbers and unlimited precision integers), atom (like the Scheme symbol type), list, tuple (like the Scheme vector type), function, port (a channel for communicating with external processes and the file system), pid (a process identifier), reference (a globally unique marker), and binary (an array of bytes). Integers are used to represent characters and lists of integers are used to represent strings. Erlang's arithmetic operators are generic (any mix of numbers) and the comparison operators can compare any mix of types.

Erlang's syntax is inspired by Prolog (e.g. `[x,y,z]`, `[]` and `[H|T]` denote lists, variables begin with an uppercase letter and atoms with lowercase, pattern matching is used to define functions and take apart data). Erlang does not provide full unification as in Prolog (i.e. a variable is not an object that can be contained in data). Note also that a guard can be added to a pattern to constrain the match (third clause in the example below). The only way to bind a variable is to use pattern matching (in function parameters, the `case`, `receive`, and *pattern*`=`*expr* constructs). In particular in *pattern*`=`*expr* the expression is evaluated and the result is pattern matched with the pattern, variables bound in the process have a scope which includes the following expressions.

The language was designed to write robust concurrent distributed soft real-time applications in telephony. Local and remote processes are created dynamically with the `spawn` function, and interacted with by sending messages (any Erlang object) to their mailbox which sequentializes and buffers incoming messages. Messages are drained asynchronously from the mailbox with the `receive` construct, which extracts from the mailbox the next message which matches the pattern(s) specified by the `receive` (a timeout can also be specified).

Exceptions are managed using the forms `throw` *expr* and `catch` *expr*. Evaluating a `throw` *X* transfers control to the nearest dynamically enclosing `catch`, which returns *X*. Predefined exceptions exist for the builtin functions.

Erlang supports a simple module system, which provides namespace management. Each module specifies its name, the functions it exports and the functions it imports from other modules. The form `lists:map` indicates the function `map` in the module `lists`.

Here is a small contrived example of an Erlang function definition showing off some of the features of Erlang:

```
f(green,_)              -> 1.5;               % ignore second parameter
f([H|_],Y)              -> T=Y+1, {H,T*T};    % return a two tuple
f(X,Y) when integer(X) -> lists:reverse(Y);  % X must be an integer
f(X,Y)                  -> lists:map(fun(Z) -> [Z,X+Z] end, Y).
```

This is roughly equivalent[1] to the following Scheme definition:

```
(define f
  (lambda (x y) ; parameters of f are x and y
    (cond ((eq? x 'green) 1.5) ; return 1.5 if x is the symbol green
          ((pair? x)
           (let ((t (+ y 1))) ; bind t to y+1
             (vector (car x) (* t t))))
          ((integer? x)
           (reverse y)) ; y better be a list
          (else
           (map (lambda (z) ; pass an anonymous procedure to map
                  (list z (+ x z))) ; create a list
                y))))) ; y better be a list of numbers
```

[1] There are subtle differences such as `(integer? 2.0)` is true in Scheme, but `2.0` is not an integer in Erlang.

4 Portability vs Efficiency

Early on we decided that portability of the compiler was important in order to maximize its usefulness and allow experiments across platforms (different target machines but also different Scheme implementations). Etos is written in standard Scheme [7] and the generated programs conform fairly closely to the standard.

It is clear however that better performance can be achieved if non-standard features of the target Scheme implementation are exploited. For example, the existence of fast operations on fixed precision integers, i.e. fixnums, is crucial to implement Erlang arithmetic efficiently. Fixnums are not part of the Scheme standard but all of the high performance Scheme compilers have some way to manipulate them. To exploit these widespread but not truly standard features, the generated code contains calls to Scheme macros whose definition depends on the target Scheme implementation. The appropriate macro definition file is supplied when the Scheme program is compiled. Not all Scheme implementations implement the same macro facilities, but this is not a problem because each macro file is specific to a particular Scheme implementation. This approach avoids the need to recompile the Erlang program from scratch when the target Scheme implementation is changed. For example, the Erlang addition operator, which is generic and supports arguments of mixed float and unlimited precision integer types, is translated to a Scheme call of the `erl-add` macro. The macro call `(erl-add x y)` may simply expand to a call to a library procedure which checks the type of `x` and `y` and adds them appropriately or signals a run time type error, or if fixnum arithmetic is available, it may expand to an inline expression which performs a fixnum addition if `x` and `y` are fixnums (and the result doesn't overflow) and otherwise calls the generic addition procedure.

Using a macro file also allows to move some of the code generation details out of the compiler and into the macro file, making it easy to experiment and tune the compiler. For example the representation of Erlang data types can easily be changed by only modifying the macro definitions of the operations on that type.

5 Direct Translation

We also wanted the translation to be direct so that Erlang features would map into the most natural Scheme equivalent. This has several benefits:

- Erlang and Scheme source code can be mixed more easily in an application if the calling convention and data representation are similar. Special features of Scheme (such as first-class continuations and assignment) and language extensions (such as a C-interface and special libraries) can then be accessed easily. For this reason, adding extra parameters to all functions to propagate an exception handler and/or continuation would be a bad idea.
- A comparison of compiler technology between Erlang and Scheme compilers will be fairer because the Scheme compiler will process a program with roughly the same structure as the Erlang compiler.
- The generated code can be read and debugged by humans.

When a direct translation is not possible, we tried to generate Scheme code with a structure that would be compiled efficiently by most Scheme compilers. Nevertheless there is often a run time overhead in the generated Scheme code that makes it slower than if the application had been written originally in Scheme. For example, Erlang's "`<`" operator is generic (it works on numbers as well as lists and other data types) but in most application programs it is only used to compare numbers. The code generated by Etos can't use Scheme's "`<`" primitive directly because it works on numbers only.

6 Data Types

The most important Erlang data types have a direct equivalent in Scheme, as explained in this section.

6.1 Numbers

Scheme numbers are organized into a class hierarchy: integer ⊆ rational ⊆ real ⊆ complex. Independently of their class, numbers have an "exactness". For instance `2.0` denotes the inexact number 2 and `1/2` denotes the exact number 0.5. Scheme exact integers correspond to Erlang integers. In both Scheme and Erlang, integers can be of limited range. The Erlang specification requires at least 24 bit integers but all available compilers support unlimited precision integers by using a bignum representation when the integers are larger than can fit in a fixnum. Scheme inexact reals correspond to Erlang floats.

An unfortunate consequence of this representation is that testing for an Erlang integer or float translates into two tests in standard Scheme (i.e. `(and(integer? x)(exact? x))`tests if `x` is an exact integer). The test `(integer? x)` is typically quite expensive because it must return true on both exact integers and on inexact reals which happen to have a null fractional part. Again, some non-standard features can help to do this quicker, for example in Gambit-C: `(or (##fixnum? x) (##bignum? x))`.

6.2 Atoms

Scheme symbols can be used to represent Erlang atoms. Both can contain arbitrary characters and symbols can be compared for equality efficiently with the `eq?` predicate (which is simply a pointer comparison in many implementations of Scheme). The Scheme procedures `string->symbol` and `symbol->string` are equivalent to the Erlang built-in functions `list_to_atom` and `atom_to_list` except that the former deals with strings (which is a separate data type in Scheme).

One complication is that Scheme is a case-insensitive language and Erlang is case-sensitive. Variable names and symbols in the source of Scheme programs are stripped of their case. A simple solution for converting Erlang function and variable names is to prefix uppercase letters with an escape character (i.e. `^`), so that the Erlang variable `ListOfFloats` becomes `^list^of^floats` in Scheme.

Atoms are handled differently. The only way to force a particular case for symbols in Scheme is to use the procedure `string->symbol`. This means that Erlang constants containing atoms (e.g. the constant list `[1,tWo]`) must be created at run time using `string->symbol`. This is done by storing the objects created into global variables once in the initialization phase of the Scheme program and references to these globals replace references to the constants. Constants not containing atoms get converted to Scheme constants. For example, the Erlang call `f([1,tWo],[3,4])` gets converted to:

```
(define const1 (string->symbol "tWo")) ; global definitions
(define const2 (list 1 const1))
... (f const2 '(3 4)) ...
```

Alternative representations for atoms which were rejected are:

- Strings: no special treatment for uppercase letters is needed but the equality test is much more expensive.
- Symbols with escape character for uppercase letters: requires an unnatural and inefficient translation of `list_to_atom` and `atom_to_list`.

Gambit-C provides a (non-standard) notation for symbols that preserves case (e.g. `|tWo|`) so it was possible to reference atoms literally in code and constants.

6.3 Lists

Both languages handle lists similarly. In Scheme, lists are made up of the empty list (i.e. `()`) and pairs created with the `cons` primitive or the variable arity `list` primitive. The primitives `car` and `cdr` extract the head and tail of a list.

6.4 Tuples

Scheme vectors are the obvious counterpart of tuples. Vectors are constructed either with the variable arity `vector` primitive (Erlang's `{...,...}`), the `list->vector` primitive (Erlang's `list_to_tuple`), or the `make-vector` primitive (which creates a vector of length computed at run time).

A minor incompatibility is that tuples are indexed from 1 (with the `element` builtin function) and Scheme vectors are indexed from 0 (using `vector-ref`).

A more serious problem is that lists and vectors are the only compound data structures in standard Scheme. Since the Erlang data types port, pid, reference, and binary don't have a direct counterpart in Scheme, they must be implemented using lists or vectors. We have used vectors to implement these data types (as well as tuples and functions) because their content can be accessed in constant time. The first element of the vector is a symbol which indicates the type and the data associated with the type is in the remaining elements. Thus the tuple `{1,2,3}` is represented by the Scheme vector `#(tuple 1 2 3)`. Note that with this representation, tuple indexing does not require a run time decrement of the index to access an element. However, an Erlang type test translates to two Scheme

tests. Thus `(and (vector? x) (eq? (vector-ref x 0) 'tuple))` tests if `x` is a tuple (we need not worry about the `vector-ref` being out of bound because empty vectors are never created by Etos).

A more compact representation which is based on the ability to test object identity with `eq?` is to use no tag for tuples and a special tag for non-tuples:

```
(define pid-tag (vector 'pid))
(define make-pid (lambda (...) (vector pid-tag ...)))
(define pid?
  (lambda (x)
    (and (vector? x)
         (> (vector-length x) 0)
         (eq? (vector-ref x 0) pid-tag))))
```

This representation was not used because type testing (a frequent operation in pattern matching) is more expensive in this representation. One more test is required for non-tuples as shown above and many more tests for tuples (we must check that the first element is not one of the tags `pid-tag`, etc).

6.5 Functions

Scheme procedures are the obvious counterpart of Erlang functions. Erlang functions are of fixed arity so the variable arity mechanism of Scheme is not necessary. Both Erlang and Scheme can create and call functional objects.

Unfortunately, this direct representation does not support error detection. Erlang's general function calling mechanism needs to ensure that the function that is being called is of the appropriate arity, and signal a run time error if it isn't. Because there is no standard way in Scheme to extract the arity of a procedure or to trap the application of a procedure to the wrong number of arguments, functional objects are represented as a tagged vector which contains the function's arity and the corresponding Scheme closure.

Toplevel functions of a module contain the arity information in their name so no arity test is needed when they are called. For example the function `bar` of arity 2 in module `foo` is translated to a Scheme lambda-expression of arity 2 bound to the global variable `foo:bar/2` (a valid variable name in Scheme). A call such as `foo:bar(1,2)` is then translated to a Scheme call to `foo:bar/2` which is guaranteed to be bound to a procedure of arity 2.

6.6 Ports, Pids, References and Binaries

The remaining Erlang data types can be represented with tagged Scheme vectors as shown above. Ports, which allow interaction with external processes (such as device drivers written in C), will clearly have to be built with some implementation specific extension to Scheme (i.e. a foreign function interface). There are no raw binary array data types in standard Scheme so a space inefficient vector based representation must be used. Scheme strings can't be used because there is no constraint on the size of characters and the `integer->char` procedure may

not implement a natural encoding (such as ASCII). A compact representation is possible in Gambit-C by using bytevectors (arrays of 8, 16 and 32 bit integers).

7 Front End

To ensure compatibility with existing Erlang compilers, Etos' parser specification was derived from the one for the JAM interpreter and processed by our own Scheme parser generator [8,5]. The original parser constructs a parse tree built of tuples. Because Etos needs to attach semantic information on the nodes of the parse tree, a conversion phase was added to extend the tree nodes with additional fields. This conversion also computes the bound variables at each node and performs constant propagation and constant folding. Constant propagation and folding are mainly needed to avoid allocation of structures which are constant, such as in the definition `f(X) -> Y={1,2}, [X,Y,3,4].` which gets compiled as though it were: `f(X) -> [X|[{1,2},3,4]].` The list `[{1,2},3,4]` is represented internally as the Scheme constant list `(#(tuple 1 2) 3 4)`.

Following this, the free variables before and after each node are computed. This is done as a separate pass because the bound variable analysis requires a left-to-right traversal of the parse tree, whereas the free variable analysis requires a right-to-left traversal. The free variables are needed to efficiently translate `case`, `if`, and `receive` expressions, which is explained in the next section.

8 Binding and Pattern Matching

8.1 Binding in Erlang

Erlang's approach for binding variables is a relic of its Prolog heritage. Binding is an integral part of pattern matching. Once it is bound by a pattern matching operation, a variable can be referenced in the rest of a function clause but can't be bound again (unless it has become an "unsafe" variable, see below). For example, in `f({A,B}) -> [X,X,X]=A, B+X.` the function `f` will pattern match its sole argument with a two-tuple. In the process, the variables `A` and `B` get bound to the first and second element respectively. After this, `A` is referenced and pattern matched with a list containing three times the same element. Note that the first occurrence of `X` binds `X` to the first element of the list and the remaining occurrences reference the variable.

8.2 Binding in Scheme

In Scheme the basic binding construct is the lambda-expression and binding occurs when a procedure is called, as in `((λ (x) (* x x)) 3)`. Here the variable `x` is bound to `3` when the closure returned by evaluating the lambda-expression is called with `3`. Scheme also has the binding constructs `let`, `let*` and `letrec` but these are simply syntactic sugar for lambda-expressions and calls. For example the previous expression is equivalent to `(let ((x 3)) (* x x))`.

Erlang syntactic categories: ⟨`const`⟩: constant

⟨`ubvar`⟩: unbound variable
⟨`bvar`⟩: bound variable
⟨`exp1`⟩, ⟨`exp2`⟩: arbitrary expressions
⟨`pat1`⟩, ⟨`pat2`⟩: arbitrary patterns
⟨`fn`⟩: function name

Expression translation: $E(\langle\texttt{const}\rangle, k) = \texttt{(}k\ C(\langle\texttt{const}\rangle)\texttt{)}$

$E(\langle\texttt{bvar}\rangle, k) = \texttt{(}k\ N(\langle\texttt{bvar}\rangle)\texttt{)}$
$E(\langle\texttt{pat1}\rangle\texttt{=}\langle\texttt{exp1}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\texttt{(v1)(}P(\langle\texttt{pat1}\rangle, \texttt{(}k\texttt{v1)}, \texttt{(erl-exit-badmatch))v1)))}$
$E(\langle\texttt{exp1}\rangle\texttt{,}\langle\texttt{exp2}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ E(\langle\texttt{exp2}\rangle, k)\texttt{))}$
$E(\langle\texttt{exp1}\rangle\texttt{+}\langle\texttt{exp2}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\texttt{(v1)}E(\langle\texttt{exp2}\rangle, \texttt{(}\lambda\texttt{(v2)\!(}k\texttt{(erl-add v1 v2))))))}$
$E(\langle\texttt{fn}\rangle\texttt{(}\langle\texttt{exp1}\rangle\texttt{)}, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(}k\ \texttt{(}N(\langle\texttt{fn}\rangle)\texttt{/1 v1))))}$

Pattern matching translation: $P(\langle\texttt{ubvar}\rangle, s, f) = \texttt{(}\lambda\ \texttt{(}N(\langle\texttt{ubvar}\rangle)\texttt{)}\ s\texttt{)}$

$P(\langle\texttt{bvar}\rangle, s, f) = \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(if (erl-eq-object? v1}\ N(\langle\texttt{bvar}\rangle)\texttt{)}\ s\ f\texttt{))}$
$P(\texttt{[]}, s, f) = \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(if (erl-nil? v1)}\ s\ f\texttt{))}$
$P(\texttt{[}\langle\texttt{pat1}\rangle\texttt{|}\langle\texttt{pat2}\rangle\texttt{]}, s, f) =$
```
(λ (v1)
  (if (erl-cons? v1)
```
$(P(\langle\texttt{pat1}\rangle, (P(\langle\texttt{pat2}\rangle, s, f)\texttt{(erl-tl v1\!)\!)}), f)$
```
      (erl-hd v1))f))
```

Auxiliary functions: $C(const)$: translate an Erlang constant to Scheme

$N(name)$: translate an Erlang variable or function name to Scheme
Note:
vn stands for a freshly created variable which will not conflict with other variables.

Fig. 1. Simplified translation algorithm for a subset of Erlang.

8.3 Translation of Binding and Pattern Matching

To translate an Erlang binding operation to Scheme it is necessary to nest the evaluation of the "rest of the function clause" inside the binding construct. This can be achieved by performing a partial CPS conversion, as shown in Figure 1.

The translation function E has two parameters: the Erlang expression to translate (e) and a Scheme lambda-expression denoting the continuation which receives the result of the Erlang expression (k). E returns a Scheme expression.

E makes use of the function P to translate pattern matching. P's arguments are: the pattern to match and the success and failure Scheme expressions. P returns a one argument Scheme lambda-expression which pattern matches its argument to the pattern, and returns the value of the success expression if there is a match and returns the value of the failure expression otherwise.

When an Erlang function is translated, E is called on each function clause to translate the right hand side with the initial continuation `(λ (x) x)` (i.e. the identity function). Note that the continuation k and all lambda-expressions gen-

erated in the translation are always inserted in the function position of a call. This implies that in the resulting Scheme code all the lambda-expressions generated can be expressed with the `let` binding construct (except for those generated in the translation of functional objects, which is not shown). To correctly implement tail-calls, an additional translation rule is used to eliminate applications of the identity function, i.e. $((\lambda\ (\mathtt{x})\ \mathtt{x})\ Y) \rightarrow Y$.

The translation algorithm is not a traditional CPS conversion because function calls remain in direct style (i.e. translated Erlang functions do not take an additional continuation argument). This partial CPS conversion is only used to translate Erlang binding to Scheme binding. A remarkable property of function E is that it embeds k in the scope of all Scheme bindings generated in the translation of e. This is important because k may have free variables which must resolve to variables bound in e. This is achieved by inserting k inside $E(e, k)$ at a place where the variables are in scope. Similarly, P always embeds s (the success expression) in the scope of all Scheme bindings generated. This is useful to handle expressions such as `Z=(X=1)+(Y=2), X+Y+Z` which reference variables bound inside previous expressions (here `X` and `Y`).

Now consider the Erlang expression `[X|Y]=foo:f(A), X+bar:g(Y)`. This is translated to the following Scheme expression (if we assume that `A` is bound):

```
(let ((v7 ^a))
  (let ((v5 (foo:f/1 v7)))
    (let ((v6 v5))
      (if (erl-cons? v6)
          (let ((^x (erl-hd v6)))
            (let ((^y (erl-tl v6)))
              (let ((v1 v5))
                (let ((v2 ^x))
                  (let ((v4 ^y))
                    (let ((v3 (bar:g/1 v4)))
                      (erl-add v2 v3)))))))
          (erl-exit-badmatch)))))
```

There are many useless bindings in this code. In the actual implementation, the translator keeps track of constants, bound variables and singly referenced expressions and propagates them to avoid useless bindings. With this improvement the code generated is close to what we would expect a programmer to write:

```
(let ((v5 (foo:f/1 ^a)))
  (if (erl-cons? v5)
      (erl-add (erl-hd v5) (bar:g/1 (erl-tl v5)))
      (erl-exit-badmatch))))
```

8.4 Translation of Conditionals

The `case`, `if`, and `receive` constructs perform conditional evaluation. The `case` construct evaluates an expression and finds the first of a set of patterns which matches the result, and then evaluates the expression associated with that pattern. The `receive` construct is like `case` except that the object to be matched is implicit (the next message in the process' mailbox), no error is signaled if no pattern matches (it simply moves to the following message in a loop until a match is found), and a timeout can be specified. The `if` construct is a degenerate `case` where each clause is only controlled by a boolean guard (no pattern matching is done).

These conditional constructs must be handled carefully to avoid code duplication. Consider this compound Erlang expression containing `X*Y` preceded by a `case` expression: `case X of 1->Y=X*2; Z->Y=X+1 end, X*Y`.

This `case` expression will select one of the two bindings of `Y` based on the value of `X`. After the `case`, `Y` is a bound variable that can be referenced freely. On the other hand `Z` is not accessible after the `case` because it does not receive a value in all clauses of the `case` (it is an "unsafe" variable after the `case`).

The `case` construct could be implemented by adding to the translation function E a rule like Figure 2a. Note that the continuation k is inserted once in the generated code for each clause of the `case`. This leads to code duplication which is a problem if the `case` is not the last expression in the function body and the `case` has more than one clause. If the function body is a sequence of n binary `case` expressions, some of the code will be duplicated 2^n times.

This code explosion can be avoided by factoring the continuation so that it appears only once in the generated code. A translation rule like Figure 2b would almost work. The reason it is incorrect is that k is no longer nested in the scope of the binding constructs generated for the `case` clauses, so the bindings they introduce are not visible in k.

A correct implementation has to transfer these bindings to k. This can be done by a partial lambda-lifting of k as shown in Figure 2c. The arguments of the lambda-lifted k (i.e. `vk`) are the result of the `case` (i.e. `vr`) and the set of bound variables that are added by the clauses of the `case` and referenced in k (i.e. AV). Each clause of the case simply propagates these bindings to `vk`. AV can be computed easily from the free variables (it is the difference between the set of free variables after the `case` and the set of free variables after the selector expression). The lambda-lifting is partial because `vk` may still have free variables after the transformation.

This lambda-lifting could be avoided by using assignment. Dummy bindings to the variables AV would be introduced just before the first pattern matching operation. Assignment would be used to set the value of these variables in the clauses of the `case`. This solution was rejected because many Scheme systems treat assignment less efficiently than binding (due to generational GC and the assignment conversion traditionally performed to implement `call/cc` correctly).

In the actual implementation of the pattern matching constructs, the patterns are analyzed to detect common tests and factor them out so that they are only

```
E( case ⟨exp0⟩ of      ,k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                  (P(⟨pat1⟩,
    ⟨pat2⟩ -> ⟨exp2⟩                     E(⟨exp1⟩,k),   ;;; duplication of k
   end                                   (P(⟨pat2⟩,
                                            E(⟨exp2⟩,k),   ;;; duplication of k
                                            (erl-exit-case-clause))
                                          v0))
                                       v0)))
```

a) Inefficient translation of the `case` construct.

```
E( case ⟨exp0⟩ of      ,k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                  (let ((vk k))   ;;; k not in right scope
    ⟨pat2⟩ -> ⟨exp2⟩                     (P(⟨pat1⟩,
   end                                      E(⟨exp1⟩,vk),
                                            (P(⟨pat2⟩,
                                               E(⟨exp2⟩,vk),
                                               (erl-exit-case-clause))
                                             v0))
                                          v0))))
```

b) Incorrect translation of the `case` construct.

```
E( case ⟨exp0⟩ of      ,k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                  (let ((vk (λ (vr AV...) (k vr))))
    ⟨pat2⟩ -> ⟨exp2⟩                     (P(⟨pat1⟩,
   end                                      E(⟨exp1⟩, (λ (vr) (vk vr AV...))),
                                            (P(⟨pat2⟩,
                                              E(⟨exp2⟩, (λ (vr) (vk vr AV...))),
                                              (erl-exit-case-clause))
                                             v0))
                                          v0))))
```

Where *AV...* is the set of bound variables that are added by the clauses of the `case` and referenced in k.

c) Correct translation of the `case` construct.

Fig. 2. Translation of the `case` construct.

executed once (using a top-down, left-right pattern matching technique similar to [4]). For example the translation of the following `case` expression will only contain one test that `X` is a pair:

```
case X of  [1|Y]->...;  [2|Z]->...  end
```

9 Errors and `catch/throw`

The traditional way of performing non-local exits in Scheme is to use first-class continuations. A `catch` is translated to a call to Scheme's `call/cc` procedure which captures the current continuation. This "escape" continuation is stored in the process descriptor after saving the current escape continuation for when the `catch` returns. A `throw` simply calls the current escape continuation with its argument. When control resumes at a `catch` (either because of a normal return or a `throw`), the saved escape continuation is restored in the process descriptor.

10 Concurrency

First-class continuations are also used to implement concurrency. The state of a process is maintained in a process descriptor. Suspending a process is done by calling `call/cc` to capture its current continuation and storing this continuation in the process descriptor. By simply calling a suspended process' continuation, the process will resume execution.

Three queues of processes are maintained by the runtime system: the ready queue (processes that are runnable), the waiting queue (processes that are hung at a `receive`, waiting for a new message to arrive in their mailbox), and the timeout queue (processes which are hung at a `receive` with timeout). The timeout queue is a priority queue, ordered on the time of timeout, so that timeouts can be processed efficiently.

There is no standard way in Scheme to deal with time and timer interrupts. To simulate preemptive scheduling the runtime system keeps track of the function calls and causes a context switch every so many calls. When using the Gambit-C Scheme system, which has primitives to install timer interrupt handlers, a context switch occurs at the end of the time slice, which is currently set to 100 msecs.

11 Performance

11.1 Benchmark Programs

To measure the performance of our compiler we have used mostly benchmark programs from other Erlang compilers. We have added two benchmarks (`ring` and `stable`) to measure the performance of messaging and processes.

- `barnes` (10 repetitions): Simulates gravitational force between 1000 bodies.
- `fib` (50 repetitions): Recursive computation of 30th Fibonacci number.
- `huff` (5000 repetitions): Compresses and uncompresses a 38 byte string with the Huffman encoder.
- `length` (100000 repetitions): Tail recursive function that returns the length of a 2000 element list.
- `nrev` (20000 repetitions): Naive reverse of a 100 element list.

- `pseudoknot` (3 repetitions): Floating-point intensive application taken from molecular biology [13].
- `qsort` (50000 repetitions): Sorts 50 integers using the Quicksort algorithm.
- `ring` (100 repetitions): Creates a ring of 10 processes which pass around a token 100000 times.
- `smith` (30 repetitions): Matches a DNA sequence of length 32 to 100 other sequences of length 32. Uses the Smith-Waterman algorithm.
- `stable` (5000 repetitions): Solves the stable marriage problem concurrently with 10 men and 10 women. Creates 20 processes which send messages in fairly random patterns.
- `tak` (1000 repetitions): Recursive integer arithmetic Takeuchi function. Calculates `tak(18,12,6)`.

11.2 Erlang Compilers

Etos was coupled with the Gambit-C Scheme compiler version 2.7a [10]. We will first briefly describe the Gambit-C compiler.

The Gambit programming system combines an interpreter and a compiler fully compliant to R^4RS and IEEE specifications. The Gambit-C compiler translates Scheme programs to portable C code which can run on a wide variety of platforms. Gambit-C also supports some extensions to the Scheme standard such as an interface to C which allows Scheme code to call C routines and vice versa.

The Gambit-C compiler performs many optimizations, including automatic inlining of user procedures, allocation coalescing, and unboxing of temporary floating point results. The compiler also emits instructions in the generated code to check for stack overflows and external events such as user or timer interrupts. The time between each check is bound by a small constant, which is useful to guarantee prompt handling of interprocess messages.

Gambit-C includes a memory management system based on a stop and copy garbage collector which grows and shrinks the heap as the demands of the programs change. The user can force a minimum and/or maximum heap size with a command line argument. Scheme objects are encoded in a machine word (usually 32 bits), where the lower two bits are the primary type tag. All heap allocated objects are prefixed with a header which gives the length and secondary type information of the object. Characters and strings are represented using the Unicode character set (i.e. 16 bit characters). Floating point numbers are boxed and have 64 bits of precision (like the other Erlang compilers used).

The implementation of continuations uses an efficient lazy copying strategy similar to [15] but of a finer granularity. Continuation frames are allocated in a small area called the "stack cache". This area is managed like a stack (i.e. LIFO allocation) except when the `call/cc` procedure is called. All frames in the stack cache upon entry to `call/cc` can no longer be deallocated. When control returns to such a frame, it is copied to the top of the stack cache. Finally, when the stack cache overflows (because of repeated calls to `call/cc` or because of a deep recursion), the garbage collector is called to move all reachable frames from the stack cache to the heap.

We have compared Etos version 1.4 [9] with three other Erlang compilers:

- Hipe version 0.27 [17], an extension of the JAM bytecode compiler that selectively compiles bytecodes to native code;
- BEAM/C version 4.5.2 [14], compiles Erlang code to C using a register machine as intermediate;
- JAM version 4.4.1 [2], a bytecode compiler for a stack machine.

11.3 Execution Time

The measurements were made on a Sun UltraSparc 143 MHz with 122 Mb of memory. Each benchmark program was run 5 times and the average was taken after removing the best and worse times.

The Scheme code generated by Etos is compiled with Gambit-C 2.7a and the resulting C code is then compiled with `gcc` 2.7.2 using the option `-O1`. The executable binary sets a fixed 10 Mb heap.

Program	Etos (secs)	Time relative to Etos		
		Hipe	BEAM/C	JAM
`fib`	31.50	1.15	1.98	8.33
`huff`	9.74	1.48	5.01	24.81
`length`	11.56	2.07	3.44	34.48
`smith`	10.79	2.17	3.37	13.06
`tak`	13.26	1.12	4.37	11.09
`barnes`	9.18	2.08	–	4.07
`pseudoknot`	16.75	2.37	–	3.18
`nrev`	22.10	.84	1.83	10.98
`qsort`	14.97	.96	3.88	15.38
`ring`	129.68	.30	.31	1.92
`stable`	21.27	1.16	.64	2.43

Fig. 3. Execution time of benchmarks

The results are given in Figure 3. They show that Etos outperforms the other Erlang compilers on most benchmarks. If we subdivide the benchmarks according to the language features they stress, we can explain the results further:

- `fib`, `huff`, `length`, `smith` and `tak`, which are integer intensive programs, take advantage of the efficient treatment of fixnum arithmetic in Gambit-C and from the inlining of functions. Etos is up to two times faster than Hipe, 5 times faster than BEAM/C, and 35 times faster than JAM.
- On the floating point number benchmarks, `barnes` and `pseudoknot`, Etos is also faster than the other Erlang implementations. In this case Etos is a little over two times faster than Hipe. These programs crashed when compiled with BEAM/C.

- List processing is represented by `nrev` and `qsort`. On these programs Hipe is a little faster than Etos (4% to 16%), which is still roughly two to four times faster than BEAM/C. Etos' poor performance is only partly attributable to its implementation of lists:
  1. Gambit-C represents lists using 3 word long pairs as opposed to 2 words on the other systems. Allocation is longer and the GC has more data to copy.
  2. Gambit-C guarantees that interrupts are checked at bound intervals [11] which is not the case for the other systems. For example, the code generated by Gambit-C for the function `app` (the most time consuming function of the `nrev` benchmark) tests interrupts twice as often as Hipe (i.e. on function entry and return).
  3. The technique used by Gambit-C to implement proper tail-recursion in C imposes an overhead on function returns as well as calls between modules. For `nrev` the overhead is high because most of the time is spent in a tight non-tail recursive function. Independent experiments [12] have shown that this kind of program can be sped up by a factor of two to four when native code is generated instead of C.
- Finally `ring` and `stable` manipulate processes. Here we see a divergence in the results. Hipe is roughly three times faster than Etos on `ring`. Etos performs slightly better than Hipe on `stable` but is not as fast as BEAM/C. We suspect that our particular way of using `call/cc` to implement processes (and not the underlying `call/cc` mechanism) is the main reason for Etos' poor performance:
  1. When a process' mailbox is empty, a `receive` must call the runtime library which then calls `call/cc` to suspend the process. These inter-module calls are rather expensive in Gambit-C. It would be better to inline the `receive` and `call/cc`.
  2. Scheme's interface to `call/cc` (which receives a closure, which will typically have to be allocated, and must allocate a closure to represent the continuation) adds considerable overhead to the underlying `call/cc` mechanism which requires only a few instructions.

12 Future Work

Etos 1.4 does not implement Erlang fully. Most notably, the following features of Erlang are not implemented:

1. Macros, records, ports, and binaries.
2. Process registry and dictionary.
3. Dynamic code loading.
4. Several built-in functions and libraries.
5. Distribution (all Erlang processes must be running in a single user process).

We plan to add these features and update the compiler so that it conforms to the upcoming Erlang 5.0 specification.

An interesting extension to Etos is to add library functions to access Gambit-C's C-interface from Erlang code. Interfacing Erlang, Scheme and C code will then be easy.

The Gambit-C side of the compilation can also be improved. In certain cases the Scheme code generated by Etos could be compiled better by Gambit-C (its optimizations were tuned to the style of code Scheme programmers tend to write). It is worth considering new optimizations and extensions specifically designed for Etos's output. In particular, a more efficient interface to `call/cc` will be designed. Moreover we think the performance of Etos will improve by a factor of two on average when we start using a native code back-end for Gambit. We are also working on a hard real-time garbage collector and a generational collector to improve the response time for real-time applications.

13 Conclusions

It is unfortunate that inessential mismatches between Erlang and Scheme cause many small difficulties in the translation of Erlang to standard Scheme. Specifically the translation would be easier and more efficient if Scheme: was case-sensitive, did not separate the numeric class (integer, ...) and exactness of numbers, allowed testing the arity of procedures or a way to trap arity exceptions, had the ability to define new data types, had a raw binary array data type, a foreign function interface and a more efficient interface to `call/cc`. Fortunately, mature implementations of Scheme, and Gambit-C in particular, already include many of these extensions to standard Scheme so in practice it is not a big problem because such features can be accessed on an implementation specific basis by using a file of macros tailored to the Scheme implementation.

When coupled with Gambit-C, the Etos compiler shows promising results. It performs very well on integer and floating point arithmetic, beating all other currently available implementations of Erlang. Its performance on list processing and process management is not as good but we think this can be improved in a number of ways. This is a remarkable achievement given that the front-end of the compiler was implemented in less than a month by a single person. It shows that it is possible to quickly reuse existing compiler technology to build a new compiler and that a compiler with a deep translation pipeline (i.e. Erlang to Scheme to C to machine code) need not be inefficient. Of course Etos' success is to a great extent due to the fact that Scheme and Erlang offer very similar features (data types, functional style, dynamic typing) and that Scheme's `call/cc` can be used to simulate Erlang's escape methods and concurrency. Our work shows that Scheme is well suited as a target for compiling Erlang.

Acknowledgements

This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and the Fonds pour la formation de chercheurs et l'aide à la recherche.

References

1. J. L. Armstrong. The development of erlang. In *Proceedings of the International Conference on Functional Programming*, pages 196–203, Amsterdam, June 1997. 300
2. J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real-time applications. In *Proceedings of Software Engineering for Telecommunication Switching Systems*, Florence, April 1992. 314
3. J. L. Armstrong, S. R. Virding, C. Wikström, and M. C. Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996. 300
4. Lennart Augustsson. Compiling Pattern Matching. In Jean-Pierre Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy, France, LNCS 201*, pages 368–381. Springer Verlag, 1985. 311
5. D. Boucher. Lalr-scm. Available at `ftp.iro.umontreal.ca` in `pub/parallele/boucherd`. 307
6. W. Clinger. Proper Tail Recursion and Space Efficiency. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 174–185, Montréal, June 1998. ACM Press. 301
7. W. Clinger and J. Rees [editors]. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991. 300, 303
8. D. Dubé. SILex, user manual. Available at `ftp.iro.umontreal.ca` in `pub/parallele`. 307
9. M. Feeley. Etos version 1.4. Compiler available at `ftp.iro.umontreal.ca` in `pub/parallele/etos/etos-1.4`. 314
10. M. Feeley. Gambit-C version 2.7a, user manual. Compiler available at `ftp.iro.umontreal.ca` in `pub/parallele/gambit/gambit-2.7`. 313
11. M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the Functional Programming and Computer Architecture Conference*, pages 179–187, Copenhagen, June 1993. 315
12. M. Feeley, J. Miller, G. Rozas, and J. Wilson. Compiling Higher-Order Languages into Fully Tail-Recursive Portable C. Technical Report 1078, Département d'informatique et de recherche opérationelle, Université de Montréal, 1997. 315
13. P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. Van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996. 300, 313
14. B. Hausman. Turbo Erlang: approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994. 314
15. R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, 1990. 313
16. IEEE Standard for the Scheme Programming Language. IEEE Standard 1178-1990, IEEE, New York, 1991. 300
17. E. Johansson, C. Jonsson, T. Lindgren, J. Bevemyr, and H. Millroth. A pragmatic approach to compilation of Erlang. UPMAIL Technical Report 136, Uppsala University, Sweden, July 1997. 314

18. The Internet Scheme Repository.
`http://www.cs.indiana.edu/scheme-repository`. 300
19. Gerald Jay Sussman and Guy Lewis Steele Jr. SCHEME, an interpreter for extended lambda calculus. AI Memo 349, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., December 1975. 301

From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View

Silvia Breitinger, Ulrike Klusik, and Rita Loogen*

Philipps-Universität Marburg, D-35032 Marburg, Germany
{breiting,klusik,loogen}@mathematik.uni-marburg.de

Abstract. The explicitly parallel programming language Eden adds a coordination level to the lazy functional language Haskell. This paper describes how a compiler and runtime system for Eden can incrementally be built on the basis of a compiler and runtime system for the computation language. The modifications needed in the compiler are restricted to specific orthogonal extensions. We show that Eden's design for distributed memory systems proves beneficial for the construction of a lean parallel runtime system.

1 Introduction

Due to the side effect freedom of the reduction semantics of functional languages it is possible to evaluate independent subexpressions in arbitrary order or in parallel. This *implicit parallelism* is semantically transparent and allows the automatic parallelization of functional programs. In available parallel functional systems like the Glasgow parallel Haskell (GpH) system [17] or the Nijmegen CLEAN system [19] the programmer is asked to place annotations in programs. These annotations show the compiler which subexpressions can potentially be evaluated in parallel. They only affect the runtime behaviour of programs.

Apart from the exploitation of implicit parallelism, explicit parallel constructs can be added in order to obtain more expressive parallel functional languages and to carry the benefits of functional languages into the world of parallel programming. Caliban [9] e.g. supports annotations for partitioning a program into a static net of processes which communicate via head-strict lazy streams. In *data-parallel functional languages* like Sisal [16], Nesl [2] and SCL [7] special parallel structures and operations are provided to express data-parallel algorithms on a high-level of abstraction. *Concurrent functional languages* like Facile [8], Concurrent ML [14], Erlang [1] and Concurrent Haskell [12] add primitive constructs for the dynamic creation of concurrent processes (spawn or fork) and the exchange of messages between such processes (send and receive) to the functional kernel language. The main purpose of concurrent languages is however the definition of reactive systems and not the speeding up of computations by parallelism.

* The authors have been supported by the DAAD (Deutscher Akademischer Austauschdienst) in the context of a German-Spanish Acción Integrada.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 318–335, 1998.

The parallel functional language Eden [6,5] enables the programmer to explicitly express systems consisting of processes and interconnecting channels. This provides control over the granularity of processes and the communication topology. The placement of processes and the sending and receiving across communication channels will efficiently be handled by the system.

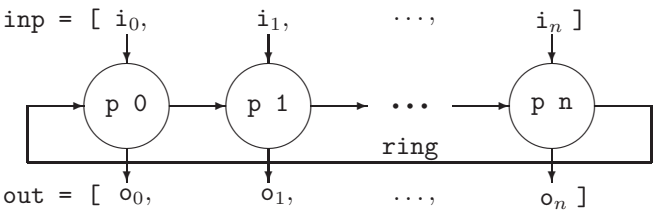
Eden is defined by adding a coordination language with explicit processes to the computation language Haskell [11]. It is tailored for distributed memory systems. The coordination language contains *process abstractions*, which define schemes for processes in a functional style, and *process instantiations*, which are expressions that generate processes and yield the respective outputs as their result. A process that maps inputs $in_1, \ldots, in_m$ to outputs $out_1, \ldots, out_n$ can be specified by the process abstraction

$$\texttt{process}\ (in_1, \ldots, in_m)\ \texttt{->}\ (out_1, \ldots, out_n)$$
$$\texttt{where}\ equation_1 \ldots equation_r$$

The optional `where` part of this expression is used to define auxiliary functions and common subexpressions which occur within the output expression.

Process abstractions have the type `Process a b` where `Process` is a newly introduced type constructor and the type variables `a` and `b` represent the input and output interface of the process, respectively. A process can have as input (respectively, output) tuples of channels and data structures of channels [4]. In the latter case, an annotation `<.>` is used to mark channels in types without changing the types themselves.

Example 1 (Process ring). Consider the following process structure which occurs in many parallel algorithms:



The whole system can easily be defined in Eden as a process that maps a list of input channels to a list of output channels and takes a process abstraction as a parameter that defines the behaviour of the component processes. The number of ring processes corresponds to the length of the input list:

```
procRing :: (Int -> Process (i,r) (o,r)) -> Process [<i>] [<o>]
procRing    p
  = process inp -> out
    where (out, ring) = createRing p 0 inp ring
```

Note that the type of the process abstraction makes clear that the component processes have two input and two output channels and that the integer identifier

is not communicated but passed as a parameter before process creation. For the definition of the function `createRing` we need the concept of process creation.

Process instantiations create processes and their input and output channels. In such an expression, a process abstraction p is applied to a tuple of input expressions, yielding a tuple of outputs. The child process uses n independent threads of control in order to produce these outputs. Likewise, the parent introduces m additional threads that evaluate $input\_exp_1, \ldots, input\_exp_m$:

$$(out_1, \ldots, out_n) = p \ \# \ (input\_exp_1, \ldots, input\_exp_m)$$

Example 1 (cont'd): The function `createRing` can now be defined as follows:

```
createRing                    :: (Int -> Process (i,r) (o,r)) ->
                                 Int -> [i] -> r -> ([o],r)
createRing p k []     ringIn =  ([], ringIn)
createRing p k (i:is) ringIn =  (o:os, ringOut)
   where (o, ringMid) = p k # (i,ringIn)  -- process instantiation
         (os,ringOut) = createRing p (k+1) is ringMid
```

The function `createRing` builds up the process ring element by element, as long as further inputs are available. Finally, the two ends of the ring connection are fused. The ring processes are created together with their communication interface by the process instantiations `p k # (i,ringIn)`.

Communication channels are unidirectional $1:1$ connections which automatically transmit output expressions which have been evaluated to normal form before. Lists are transmitted in a *stream*-like fashion, i.e piecemeal. Upon process creation, communication channels for the transmission of inputs and outputs between parent and child will be established. An evaluation will be suspended if it depends on input that has not yet been received.

The evaluation of Eden processes is driven by the evaluation of the outport expressions, for which there is always demand. This rule overrides normal lazy evaluation in favour of parallelism. There will be a separate concurrent thread of execution for each output channel. Thus, we find in Eden two levels of concurrency: the concurrent or parallel evaluation of processes and the concurrent evaluation of different threads within processes. A process with all its outputs closed will terminate immediately. On termination its input channels will be eliminated and the corresponding outports in the sender processes will be closed.

Furthermore, each process immediately evaluates all *top level* process instantiations. This may lead to speculative parallelism and again overrules lazy evaluation to some extent, but speeds up the generation of new processes and the distribution of the computation.

No implicit or explicit sharing of information among processes is possible. All kinds of communication and synchronization have to be performed using communication channels. If a process abstraction contains global (free) variables on process creation, they will be replaced by their bindings. As the bindings may

be unevaluated, this may lead to the duplication of work. Complex parameters should therefore be replaced by inputs.

The coordination constructs introduced up to now preserve referential transparency. This is not true for the two further constructs which support the definition of reactive systems in Eden but will not be handled in this paper: dynamic reply channels and predefined nondeterministic processes (see [5] for details).

In this paper we show how to extend the Glasgow Haskell compiler[1] (GHC), a publicly available fast implementation of Haskell, for Eden's parallel implementation. We describe the compilation of Eden and present a lean parallel runtime system which implements DREAM, the DistRibuted Eden Abstract Machine [3]. In the Eden parallel runtime system the process structures underlying parallel algorithms can exactly be modelled. Process abstractions determine the process interface and the environment in which the evaluation of the processes takes place. There is no need for a (virtual) shared memory or global address space. This simplifies the whole memory management including garbage collection. Several parts of the GUM (Graph reduction for a Unified Machine model) runtime system [18] for Glasgow parallel Haskell programs [17] have been reused for our parallel system. Our work is not restricted to Eden, a similar approach might be taken for other orthogonal extensions of sequential languages by a coordination language.

2 The Eden Compiler

The key idea of the GHC which forms the basis for the Eden compiler is "compilation by program transformation" [15]. Haskell programs are first transformed into Core Haskell, a minimal functional language in which all other Haskell constructs can be expressed. Core Haskell is mapped to a still functional abstract machine language STGL which has a simple denotational and operational semantics. The operational semantics is defined by the Spineless Tagless G-machine (STGM) which is an abstract description of the run-time system [13]. C is used as a portable target language of the compiler. Figure 1 shows the structure of GHC and the following extensions:

- The **parser** transforms the Eden constructs into Haskell expressions which contain functions defined in a Haskell module `EdenBase`.
- An additional compiler pass is introduced for the transformation of **STG to PEARL**. PEARL is an extension of the STG language by primitive parallel constructs. It enables Eden-specific optimizations and transformations.
- During **code generation** special code for the new PEARL expressions is generated.

2.1 Passing Eden Programs through the GHC Front End

From an implementation point of view, Eden's extensions to Haskell can be seen as "syntactic sugar" in the front end. We introduce for each parallel construct

[1] see `http://www.dcs.gla.ac.uk/fp/software/ghc`

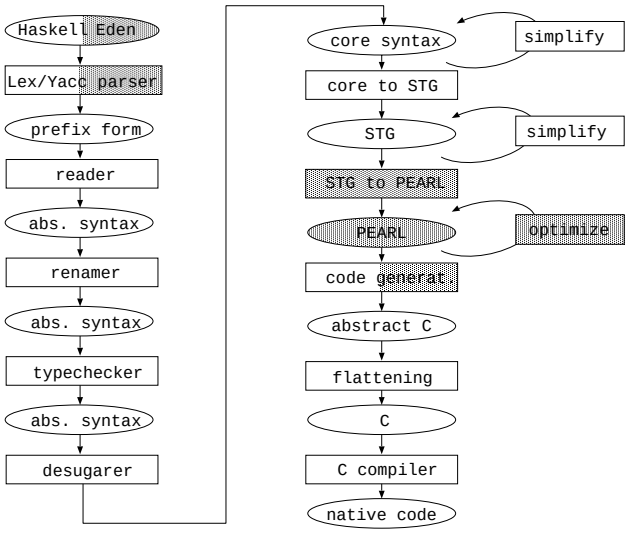


Fig. 1. Overview of GHC's compiler passes and our extensions

a *Haskell function* with the types chosen appropriately to ensure Eden's type restrictions.

We specify the Eden data type `Process a b` to distinguish between processes and functions and introduce Haskell functions `process` and `(#)`:

```
process :: (a -> b) -> Process a b
(#)     :: Process a b -> a -> b
```

In order to force the evaluation of output expressions to normal form and to model the sending of data via communication channels, the following classes are included in the module `EdenBase`:

```
class Eval a => NFData a where
  rnf   :: a -> ()
  rnf x =  x `seq` ()

class NFData a => Transmissible a where
  sendChan   :: a -> a
  sendChan x =  rnf x `seq` sendVal x `seq` x
```

The class `NFData` has been introduced in [17]. Its method `rnf` reduces expressions to normal form. The default implementation is the head normal form evaluation of the class `Eval`, which is sufficient for basic data types. For algebraic data types, the normal form is produced by forcing the normal form evaluation of all component expressions. The instance derivation tool described in [20] already provides rules for creating such instances.

The class `Transmissible` provides an overloaded function `sendChan` which is used to make explicit the sending of values across communication channels. In each of its instances, `sendChan` forces the evaluation of its argument and relies on primitive functions `sendVal`, `sendHead` and `closeStrm` to perform the communication[2]. The primitive functions are implemented by routines of the underlying message passing system. The default implementation of `sendChan` is used for all data types except lists. To communicate lists as streams the following instance declaration is used:

```
instance Transmissible a => Transmissible [a] where
  sendChan xs = sendStream xs `seq` xs
sendStream        :: Transmissible a => [a] -> ()
sendStream []     =  closeStrm
sendStream (x:xs) =  rnf x `seq` sendHead x `seq` sendStream xs
```

The parser recognizes a process abstraction of the form

```
process (i1, ..., im) -> (o1, ..., on)
where decls
```

and transforms it into the following Haskell expression using the definitions in the module `EdenBase`:

```
process ( \ (i1, ..., im) -> let decls
                             in  (sendChan o1, ..., sendChan on) )
```

A process instantiation of the form `pabs # (e1,...,em)` is transformed into `pabs # (sendChan e1,..., sendChan em)`.
The other Eden constructs are handled similarly.

To sum up, we have only modified the parser in order to embed Eden's additional expressions in Haskell syntax. The resulting Haskell program is passed through the subsequent, unchanged compiler passes, until STG syntax is reached. In particular, the original type inference algorithm is used to check the types of Eden programs.

2.2 Transforming STG into Pearl

Our intermediate language PEARL (Parallel Eden Abstract Reduction Language) is an extension of the STG language by parallel constructs. PEARL's operational semantics is given by DREAM (DistRibuted Eden Abstract Machine), an orthogonal extension of the STGM, formally specified in [3].

In PEARL, multithreading is expressed explicitly. The following syntax is used:

[2] The communication functions need no argument for the destination of the communication, because this is implicitly given in the corresponding outport definition.

```
{expr || ... || expr}                          -- multithreading
{frees} \p {args} body                         -- process abstraction
{var, ..., var} = p # {expr || ... || expr}
                                               -- process instantiation
```

The multithreading expression means that the component expressions should be evaluated concurrently, each by a separate newly created thread. It is used in process instantiations and in the body of process abstractions to implement the generation of threads. In the runtime system a primitive function `createThreads` is provided for that purpose. A process abstraction is represented by a lambda abstraction with free variables `frees`, flag `p` and parameters `args`. The process instantiations occur as new `letrec`-bindings in PEARL.

In fact, the definition of primitive functions for `process` and `(#)` would have been sufficient to generate process trees, in which a process only communicates with its parent or children. To implement arbitrary process topologies and to support the simultaneous instantiation of several connected processes, a special code generation is necessary. Such opportunities for optimizations are difficult to detect in Haskell or STG programs. This is the reason for introducing PEARL and the additional compiler pass. It allows to re-collect the information about processes and their interconnections.

The following translation schemes introduce the PEARL expressions for process abstraction and instantiation.

```
STG syntax:   var = process lf
              lf  = { frees } \n { args } body_exp
              lf :: t -> (t1, ..., tn)
-->
PEARL syntax: var    = { scheme, frees } \p { args }
                       let body' = { frees, args } \n { } body_exp
                       in  scheme body'
              scheme = {} \n x -> case x of
                                    (x1,..., xn) -> {x1 ||...|| xn}
```

The original process abstraction is transformed into a lambda abstraction with flag `p`. In order to introduce the multithreading expression in the body of the process abstraction, we extract the number of output channels from the type of the original lambda form[3] and apply a general scheme for the process body to the actual body expression.

```
STG syntax:   var = { p, arg } \n { } (#) p arg
              arg = { vi_1, ..., vi_n } \n { } (vi_1, ..., vi_n)
              p :: Process (it_1, ..., it_n) (ot_1, ..., ot_m)
-->
PEARL syntax: var = { vo_1,...,vo_m } \n { } (vo_1,...,vo_m)
              {vo_1,...,vo_m} = p # {vi_1 ||...|| vi_n}
```

[3] In the GHC, the type information is still accessible in the STG program.

In STGL, the function `(#)` indicates a process instantiation. The input tuple of this instantiation will be defined separately, as functions are applied to variables only. The type of the process abstraction is used to determine the number of outputs of the instantiation. PEARL's process instantiation provides variables for the inputs and outputs. The multithreaded evaluation is made explicit. Of course, the output tuple of the process instantiation has to be reassigned to the original variable `var` as a tuple.

Naturally, the most important part of the back end is the translation into C. The compilation of STGL expressions and declarations remain unchanged. The PEARL transformation is determined by the parallel run-time system which is discussed in the following.

3 Eden's Parallel Runtime System

In this section we give an outline of the parallel runtime system of Eden which is based on the abstract machine DREAM [3]. We will show the advantages of the design decisions underlying Eden with respect to efficiency and ease of implementation in a distributed setting.

In **DREAM**, an Eden process consists of one or more concurrent threads of control. These evaluate different output expressions which are independent of each other and use a common heap that contains shared information (cf. Fig. 2).

Likewise, the state of a *process* includes information common to all threads and the states of the threads. The shared part includes the heap and an inport table, because input is shared among all threads. The state of a *thread* comprises the state of a sequential abstract machine and a specification of the associated outport referencing the connected inport. The interface of a process is visible in the following way: the outports are associated to threads (runnable or blocked) and the inports are enumerated in the inport table.

Communication channels are only represented by inports and outports which are connected to each other. An outport is just the global reference to an inport. These are the only global references needed. There are no explicit channel buffers, but the data is transferred from the producer's heap to the consumer's heap using the inport to find the location where the data should be stored.

We have already shown how we can benefit from our distinction of communication and coordination language by reusing a "state of the art" compiler for the computation language Haskell. In addition, we can take advantage of the runtime system GUM [18] for an implicitly parallel version of Haskell.

In the **parallel functional runtime system GUM**, the units of computation are called threads. Each *thread* evaluates an expression to weak head normal form. A thread is represented by a heap-allocated Thread State Object (TSO) containing slots for the thread's registers and a pointer to heap-allocated Stack Objects (SO). Each processor element (PE) has a pool of runnable threads.

The memory management in GUM is especially involved as it has to manage a virtual shared memory in which the shared program/data graph resides.

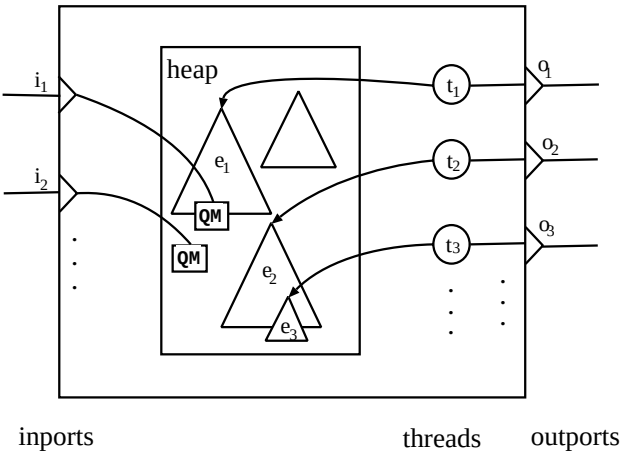


Fig. 2. A DREAM process

The virtual shared memory requires global addresses which are (PE, local identifier) pairs. Local identifiers are mapped to local heap addresses using a Global Indirection Table (GIT).

In order to transfer a subgraph from one PE to another, GUM uses sophisticated packing/unpacking algorithms, which guarantee that all the links back to the original graph are maintained and that the duplication of work is avoided. GUM uses a passive work distribution scheme, where PEs looking for work send out requests for work.

In principle, the GUM system is more complex than what is needed for Eden, as it supports a virtual shared memory and is demand driven. But the overall organization of the system, the interleaved evaluation of independent threads on a single processor, and the graph packing and unpacking algorithms can be incorporated in the Eden system, although they are used in different contexts.

The following aspects are fundamental in the design of a parallel runtime system:

1. communication between processes
2. multithreading inside of processes and scheduling
3. management of parallel activities
4. garbage collection

3.1 Communication

In contrast to most other parallel systems based on lazy functional languages, message data is evaluated and transmitted eagerly in Eden. This means that a one-message protocol is used for communication, i.e. one that dispenses with

request messages for remote data. This feature is very important for the efficient implementation on distributed memory systems with considerable communication latencies. It furthermore alleviates the overlapping of computation and communication. The threads will be suspended less frequently than in a two message protocol, where a thread *must* be blocked after requesting the remote data. Nevertheless, message passing will not be done blindly, but mechanisms for controlling the flow of messages will be provided.

The sending of data across communication channels is initiated in special routines which correspond to the predefined communication functions in PEARL. The receipt of any messages is handled by a communication unit which puts the message contents into the heap and updates the inport table.

Graph Packing/Unpacking. To pack the result of an evaluation into a packet that can be sent across the network, GUM's graph packing and unpacking algorithms can be used. In Eden, it is ensured that a subgraph to be output contains only normal form data. This allows a substantial simplification of GUM's packing algorithm which checks for each closure node whether it contains normal form data or not in order to avoid the duplication of work. The effort for the globalization of shared non-evaluated closures (thunks) is saved. The GUM algorithms however have to be extended by the possibility to use more than one packet for the transmission of large subheaps.

3.2 Multithreading

In order to prevent multiple threads of the same process from evaluating the same shared expression, a synchronization mechanism is used, which can be found in most of the parallel runtime systems for functional languages. The thread that is the first to access a thunk, overwrites it with a so-called *black hole* or `queueMe`-closure. Subsequent threads attempting to access it will be blocked in a special queue and released when the first thread overwrites the `queueMe`-closure with the result of the computation.

GUM's thread management is perfectly appropriate for Eden's threads. The sequential evaluation of threads and the synchronization of threads within the same process need not be changed. The heap-allocated TSOs are extended by the outport specification, i.e. the global reference to the inport to which the result of the thread's computation must be sent.

Scheduling: Eden needs a fair scheduler for the evaluation of concurrent threads. Every logical PE runs a scheduler, which has the following tasks:

- perform local garbage collection (see Section 3.4)
- run the communication unit, which processes incoming messages
- run the active threads according to the scheduling strategy

The scheduling strategy is selected in a way such that it balances the speed of execution between senders and consumers. For efficiency reasons, it is vital to adapt the amount of output produced to the demand for information. In particular, we throttle senders which threaten to flood the heap of slower receivers.

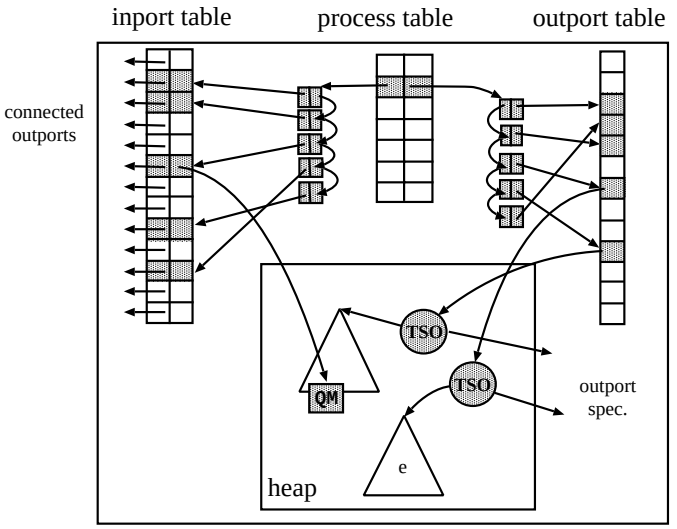


Fig. 3. Runtime tables

Assigning Eden Processes to PEs. In order to map arbitrary process systems on a finite machine, we have to use interleaving of multiple processes on the same PE. In order to keep process creation as cheap as possible, we provide only one instance of the abstract machine per PE, which can execute several Eden processes concurrently. These processes share the heap, the scheduler and the runtime tables.

3.3 Management of Parallel Activities

Runtime Tables. In order to establish the communication structure of the Eden process systems, new runtime structures have to be introduced. In every PE, we use the following tables to represent the communication channels and process interface information (see Fig. 3):

- The *inport table* maps locally unique identifiers of inports to heap addresses and the global references to the connected outports. Until the corresponding input data becomes available, a `queueMe`-closure is stored at the heap address. The outport references are used for the propagation of termination information.
- The *outport table* maps locally unique identifiers of outports to the corresponding thread (TSO) address. The outport table is used for system management, i.e. garbage collection, termination, error detection, etc.
- The *process table* contains for each process identifier the linked lists of inport and outport identifiers. Several Eden processes share the same abstract machine, and there is one entry in the process table for every process.

A *global reference* consists of the identifier of a remote PE and an identifier which is unique within this PE and will be used to index a runtime table.

Process Creation. The evaluation of a process instantiation leads to the creation of a child process which will be spawned on a processor element selected by the load balancing mechanism. The local result of the process creation is the tuple of new inports, which will be filled by the child process. The code generated for process abstractions and process instantiations causes the following sequence of actions (see Fig. 4):

I The **parent** PE evaluates a thread that encounters a process instantiation `pabs # (in1, ..., inm)`. The process abstraction `pabs` and the tuple of input expressions `in1`, ..., `inm` will be available as heap closures, the addresses of which are given on the stack within the TSO of this thread.

II The **parent** PE

- generates `queueMe`-closures for the new *inports*, to which the outports of the new process will point, provides new identifiers for them and inserts them into the inport and process tables.
- provides new identifiers for the new *outports*, which have to be connected to the inports of the new process, inserts them into the outport and process tables correspondingly.
- composes a **createProcess** *message*, which contains the processor identifier, the whole subheap of the process abstraction[4] and the identifiers for the inports and outports of the process. On process creation the *complete* subheap representing the process abstraction is duplicated to build the heap of the new abstract machine of the new process. This ensures that the data dependencies between processes are restricted to the ones represented in the process interfaces.
  The global references to the inports and outports are pairs with the processor identifier and the respective port identifiers. The subheap can be packed using our simplified packing algorithm, because thunks can be copied.
- sends this message to the "child PE" determined by the process distribution and load balancing algorithm.
- creates new threads $t_1, \ldots, t_m$ for the evaluation of the input expressions for the new process, which are suspended until the new process has returned an acknowledgement message with the global references to the corresponding inports.
- returns the tuple with the *heap addresses* of the `queueMe`-closures which represents the (not yet available) result of the subprocess evaluation.

On receipt of the process creation message, the **child** PE

[4] If the subheap contains a `queueMe`-closure which represents a *shared expression* currently under evaluation by another thread, we copy the original closure nevertheless. This is possible because we use the so-called *revertable* black holes already implemented for GUM. If it however depends on an *inport*, blocking cannot be avoided.

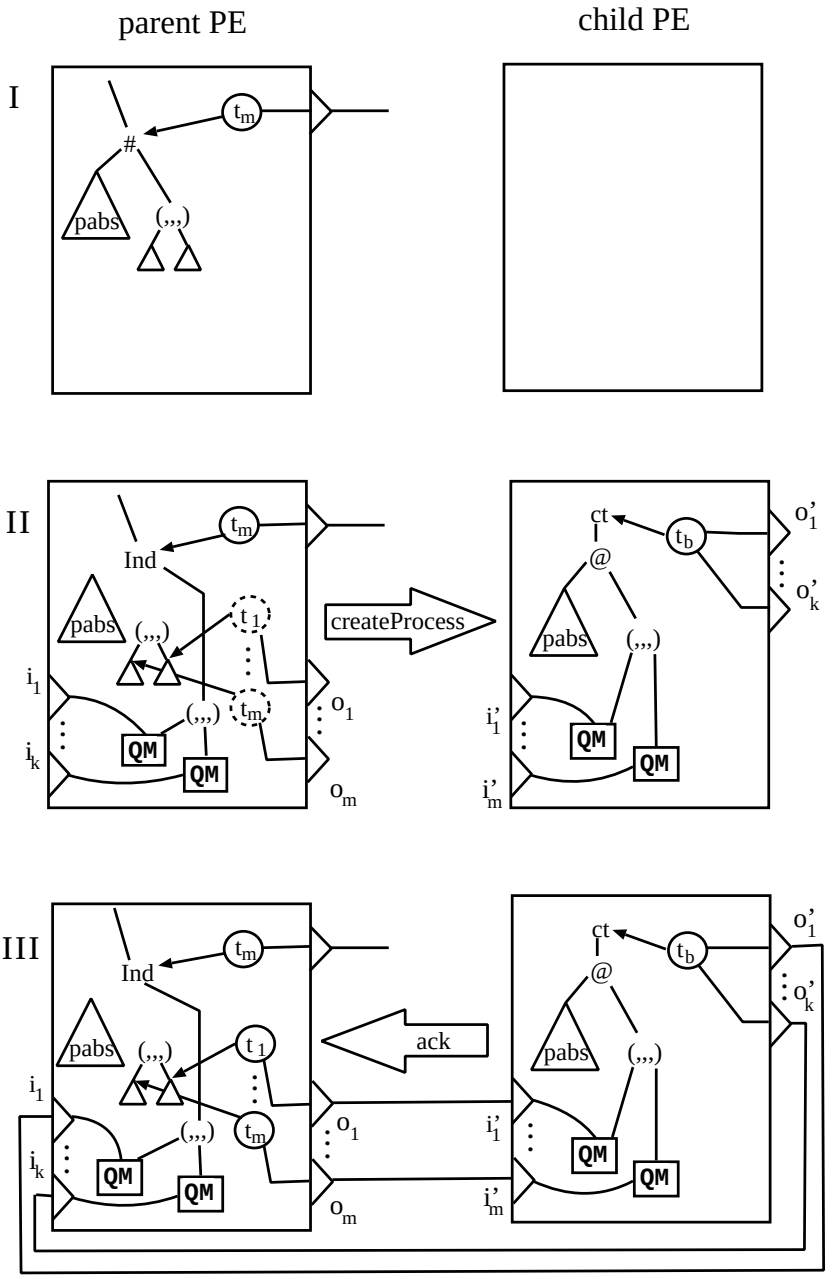


Fig. 4. Process instantiation

- generates `queueMe`-closures for the *inports* of the new process and adapts the inport and process table. The latter is extended by an entry for the new process. Note that the child PE can immediately store the connected outports of the parent PE in its inport table.
- generates new identifiers for the *outports* and extends the outport and process table accordingly.
- unpacks the *process abstraction* and starts the initial thread (t_b) of the process evaluation which will build up the process environment and then spawn the threads for the evaluation of the outport expressions via the primitive function `createThreads` (ct). In the meantime the outport specifications of these threads are stored in the state of the initial thread.

III The **child** PE sends an `ack` *message* to the parent process which contains a mapping between the global addresses of connected inports and outports. On receipt of the `ack` message, the **parent** PE writes the global references to the child's inports into the thread state objects of the outports thereby reactivating the suspended threads. Additionally, it stores the connection of the inports in its inport table.

Process Distribution and Load Balancing. Eden requires an active process distribution algorithm. In the first version of the parallel runtime system we use a random distribution algorithm, i.e. a processor that instantiates a new process sends the `createProcess` message to a randomly selected PE. We plan to investigate more elaborate versions of distribution and balancing algorithms, which use bookkeeping about processor loads and information about process topologies and which choose a suitable substitute in case the selected PE is unable to accept the `createProcess` message.

3.4 Garbage Collection and Termination

There is no need for (general) global garbage collection of heap closures in the Eden runtime system, because there is no global address space. The local garbage collection proceeds as in the sequential system, but with additional roots in the evacuation phase. This extension is similar to the treatment in the GUM system where the entries of the global indirection table (GIT) are used as additional roots for local garbage collection.

An obvious thing to do would be to use the inports as additional roots in the Eden system, because these represent the only global references to the local heap. But this approach hinders the detection of input that is no longer consumed by any thread. For this reason, we decided to use the outports, or to be precise the TSO addresses within the outport table, as additional roots for local garbage collection. This corresponds to our view that there is always demand for output and it facilitates to detect that an inport (i.e. its heap location) is not referenced any more by any thread. In this case we can terminate the sending thread and sometimes even break dependency cycles between processes (see below).

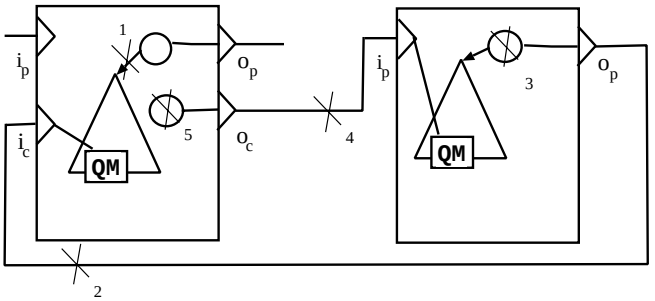


Fig. 5. Garbage collection of inports

Process Termination. Processes without any outports will be terminated as they cannot contribute to the overall result of the program. This situation is detected, when an outport is closed, by checking whether the outport list of the corresponding process in the process table is empty. The inports of the process will be closed and termination messages will be sent to the connected outports.

There are two situations in which *an outport* is closed: if its thread terminates or if a termination message from the receiver process arrives. In the latter case the PE closes the addressed outport and kills the corresponding thread. This may lead to the termination of further processes. Likewise, there are two situations in which *an inport* is closed: if its process terminates or if it is not referenced any more. The latter is detected by the local garbage collection which checks whether the `queueMe`-closures of the inports have been reached from the roots.

The closing of inports whose `queueMe`-closures can no longer be accessed is an important vehicle for the termination of subsystems of processes which are no longer needed, but maintain mutual input/output connections. Many of such systems will shut down by the closing of inports, when the outports which connect these systems with the main system will be closed. Fig. 5 visualizes a scenario where a thread does no longer consume input data (1). The inport will be closed after garbage collection and a termination message will be sent to the corresponding outport (2). This leads to the termination of the associated thread (3) and so on.

4 Related Work

In parallel functional runtime systems like Haskell's GUM [18] and CLEAN's PABC (Parallel ABC machine) [10] the task of a process is the evaluation of an expression to weak head normal form. The whole computation is demand driven. The systems provide a global address space, sometimes called virtual shared memory. Whenever the evaluation needs the contents of a global address, a request message is sent to the processor element holding that address. In the PABC machine such request messages trigger the evaluation of the expression at

the global address. In GUM the graph at the global address is packed and transferred to the requesting processor element. Also process distribution is usually done on demand, i.e. processor elements with empty work pools send requests for work to other processor elements.

In the specification of a parallel process graph in a functional language with parallelism annotations the communication channels will be modelled by function parameters, in implementation terms global addresses, and the request/answer mechanism, a two message protocol, will be adopted for "communicating values along the communication lines" of a process graph. It depends on the runtime system where expressions will be evaluated and when communication takes place. The Caliban system [9] tries to overcome this loss of topology information by extracting process net information at compile time and translating it into a call to a special system primitive called `procnet` which implements the run-time parallelism. This is only possible because the Caliban system is restricted to static process systems.

Data-parallel languages like Sisal [16], NESL [2] and SCL [7] provide only predefined parallel operations and skeletons, which can be implemented by adding parallel primitives to a sequential implementation. The development of a specific parallel abstract machine is not required. Concurrent functional languages like Facile [8], Concurrent ML [14], Erlang [1] and Concurrent Haskell [12] are based on low level parallel primitives, which are implemented on a shared memory base using fair schedulers to switch between the concurrent activities. Distributed implementations are feasible, but require the implementation of a virtual shared memory as in the runtime systems of functional languages with implicit parallelism. The low level process model of these languages obstructs the abstraction of a communication structure which could easily be mapped on a distributed system.

5 Conclusions

As Eden extends Haskell by a coordination language which allows the explicit definition of parallel process systems, an obvious approach to the implementation of Eden is to extend a Haskell compiler and runtime system. In this paper we have shown that the front end of the Glasgow Haskell compiler can be reused with minor extensions for the compilation of Eden into STGL. The back end of the compiler and the runtime system require changes, which are however designed as orthogonal additions to the existing implementation. Kernel parts of the parallel functional runtime system GUM could be reused and even simplified. The Eden parallel runtime system has recently been implemented using MPI. A detailed elaboration of this implementation, together with specific optimizations, is left for future work.

Acknowledgements

We thank Kevin Hammond, Hans-Wolfgang Loidl, Phil Trinder, Yolanda Ortega, Ricardo Peña, Steffen Priebe, and Mari Plümacher for valuable discussions.

References

1. J. Armstrong, M. Williams, and S. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 318, 333
2. G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996. 318, 333
3. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM - the DistRibuted Eden Abstract Machine. In C. Clack, T. Davie, and K. Hammond, editors, *Symposium on the Implementation of Funct. Lang. 1997, St. Andrews, selected papers*, LNCS 1467. Springer, 1998. 321, 323, 325
4. S. Breitinger and R. Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Functional Programming*, 1997. 319
5. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996. 319, 321
6. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997. 319
7. J. Darlington, Y.-K. Guo, H. To, and J. Yang. Functional skeletons for parallel coordination. In *EURO-PAR '95 Parallel Processing*, LNCS 966. Springer, 1995. 318, 333
8. A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Programming*, 18(2), 1989. 318, 333
9. P. Kelly. *Functional Programming for Loosely Coupled Multiprocessors*. Pitman, 1989. 318, 333
10. M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. PhD thesis, Katholieke Universiteit Nijmegen, 1996. 332
11. J. Peterson and K. Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical Report YALEU/DCS/RR-1106, Yale University, 1997. 319
12. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symp. on Principles of Programming Languages (POPL)*, 1996. 318, 333
13. S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2), 1992. 321
14. J. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991. 318, 333
15. A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995. 321
16. S. K. Skedzielewski. Sisal. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991. 318, 333

17. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998. 318, 321, 322
18. P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996. 321, 325, 332
19. M. van Eekelen and R. Plasmeijer. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993. 318
20. N. Winstanley. Reflections on Instance Derivation. In *Glasgow Workshop on Functional Programming*, 1997. 322

Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine

David Wakeling

School of Engineering and Computer Science
University of Exeter, Exeter, EX4 4PT, UK
http://www.dcs.exeter.ac.uk/~david

Abstract. This paper shows how lazy functional programs can be made mobile by compiling them for the Java Virtual Machine. The Haskell compiler it describes is based on the ⟨ν,G⟩-machine, which is designed for implementing lazy functional languages on parallel processors. Although this is not the obvious thing to do, it leads to a particularly elegant set of translation rules. Sadly though, the speed of the resulting Java Virtual Machine code programs is disappointing due to the surprisingly high cost of memory allocation/reclamation in current implementations of the Java Virtual Machine. In future work, we intend to find out whether this high cost is a property of the Java Virtual Machine's design or its implementation.

1 Introduction

Java [AG96] programs are *safe* because they can be verified to ensure that they will not damage the computer that they run on, and *mobile* because they can be compiled for a widely available abstract machine [LY96]. As a result, Java can be used for Internet programming; that is, for creating documents with executable content. Haskell [PH97] programs are also safe because strong type-checking ensures that they will not "go wrong" on the computer that they run on, but they are not mobile because there is no widely available abstract machine that they can be compiled for. As a result, Haskell cannot be used for Internet programming, which is a shame.

This paper shows how lazy functional programs can be made mobile by compiling them for the same abstract machine used by Java. The Haskell to Java Virtual Machine code compiler it describes is novel because it is based on the ⟨ν,G⟩-machine [AJ89], which is designed for implementing lazy functional languages on *parallel* processors, rather than the G-machine [Joh84], which is designed for implementing them on sequential ones.

The paper is organised as follows. Section 2 introduces our version of the ⟨ν,G⟩-machine, and Section 3 shows how a small core functional language can be compiled for it. Section 4 introduces the Java Virtual Machine, and Section 5 shows how ⟨ν,G⟩-machine code can be converted to Java Virtual Machine code. Section 6 presents some benchmark figures. Section 7 mentions some closely related work, and Section 8 concludes.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 335–351, 1998.

2 The ⟨ν,G⟩-machine

This section describes the machine architecture and instruction set of our version of the ⟨ν,G⟩-machine [AJ89].

2.1 Machine Architecture

The ⟨ν,G⟩-machine works by evaluating an expression to normal form. Expressions are represented by *graphs* with three kinds of node (see Figure 1). A *con-*

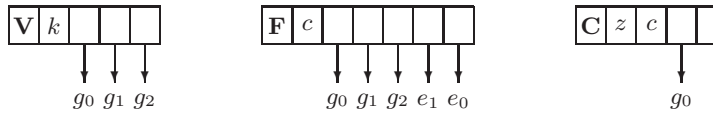


Fig. 1. Constructed value, frame and canonical application nodes.

structed value node represents the application of a data value constructor to some arguments. The node stores the constructor number, k, and has slots for pointers to the graphs representing the arguments, g_i. Basic values, such as integers and characters, are represented by constructed value nodes with appropriate values of k and no argument slots. A *frame node* represents the application of a function to all of its arguments. The node stores the function's code, c, and has slots for pointers to the graphs representing the arguments, g_i. Additional space is also provided for a small *evaluation stack* of pointers to graphs, e_j, used when the function is applied. A *canonical application node* represents the application of a function to the first few of its arguments. The node stores the number of arguments that are missing, z, the function's code, c, and has slots for pointers to the graphs representing the arguments that are present, g_i.

Expressions are evaluated by *graph reduction*. Constructed value and canonical application nodes are *irreducible*, and there is nothing to do. Frame nodes are *reducible*, and can be rewritten by executing their code and updating them with the normal form node that results. As the graph is reduced, new nodes are attached to it and existing nodes are detached from it. From time-to-time, a *garbage collector* recovers the memory occupied by the detached nodes. For us, the most important feature of the ⟨ν,G⟩-machine architecture is that there are many small evaluation stacks instead of a single large one. This makes it possible for the Java Virtual Machine garbage collector to recover all of the memory occupied by detached nodes, so avoiding the space leaks that beset our earlier G-machine-based implementation [Wak97].

2.2 Instruction Set

Each ⟨ν,G⟩-machine instruction can be described by one or more *state transition rules* of the form $s_1 \Rightarrow s_2$. A state maps names to graph nodes, with the name of the frame node where reduction is currently taking place, $\underline{\nu_n}$, underlined. In order to make the rules easier to read, we assume that any graph nodes not explicitly mentioned in s_2 are unchanged from those in s_1.

Figure 2 gives state transition rules for the various NEW and UPD instructions that allocate and initialise graph nodes. In our version of the ⟨ν,G⟩-machine, node allocation and initialisation are separated because the Java Virtual Machine separates object allocation and initialisation.

$$\left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{NEWVAL}\ k\ m : c)\ a\ s \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ ?\ ? \end{array} \right\}$$

$$\left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{NEWFRM}\ f : c)\ a\ s \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{F}\ ?\ ?\ ? \end{array} \right\}$$

$$\left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{NEWCAP}\ f : c)\ a\ s \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{C}\ ?\ ?\ ? \end{array} \right\}$$

$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{UPDVAL}\ k\ m : c)\ a\ (p_1 \cdots p_m : \nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ ?\ ? \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ k\ [p_1 \cdots p_m] \end{array} \right\}$$

$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{UPDFRM}\ f\ m : c)\ a\ (p_1 \cdots p_m : \nu_2 : s) \\ \nu_2 \mapsto \mathbf{F}\ ?\ ?\ ? \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{F}\ code(f)\ [p_1 \cdots p_m]\ [\,] \end{array} \right\}$$

$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{UPDCAP}\ f\ m : c)\ a\ (p_1 \cdots p_m : \nu_2 : s) \\ \nu_2 \mapsto \mathbf{C}\ ?\ ?\ ? \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{C}\ (arity(f) - m)\ code(f)\ [p_1 \cdots p_m] \end{array} \right\}$$

Fig. 2. Instructions for allocating and initialising graph nodes.

Figure 3 gives state transition rules for the GETSLOT and PUTSLOT instructions that move pointers between the argument slots and the evaluation stack of a frame node.

Figure 4 gives the state transition rules for the DO instruction, which makes a general tail-call by applying the canonical application node at the top of the evaluation stack to the n arguments below it. There are three possibilities to consider here. First, if the canonical application node is missing more than n arguments, the result is another canonical application node. Second, if the canonical application node is missing exactly n arguments, the result is a frame node. Finally,

$$\left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{GETSLOT}\ i : c)\ [\ldots a_i \ldots]\ s \right\} \quad \Rightarrow \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ [\ldots a_i \ldots]\ (a_i : s) \right\}$$
$$\left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{PUTSLOT}\ i : c)\ [\ldots a_i \ldots]\ (p : s) \right\} \Rightarrow \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ [\ldots p \ldots]\ s \right\}$$

Fig. 3. Instructions for moving pointers between argument slots and evaluation stack.

if the canonical application node is missing less than n arguments, a frame node must be constructed and reduced to a canonical application node by an EVAL instruction before DO can be tried again.

$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{DO}\ n : c_1)\ a_1\ [\nu_2, p_1 \cdots p_n] \\ \nu_2 \mapsto \mathbf{C}\ z\ c_2\ a_2 \end{array} \right\}, (z > n)$$
$$\Rightarrow \left\{ \underline{\nu_1} \mapsto \mathbf{C}\ (z - n)\ c_2\ (a_2 {+\!+} [p_1 \cdots p_n]) \right\}$$
$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{DO}\ n : c_1)\ a_1\ [\nu_2, p_1 \cdots p_n] \\ \nu_2 \mapsto \mathbf{C}\ z\ c_2\ a_2 \end{array} \right\}, (z = n)$$
$$\Rightarrow \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c_2\ (a_2 {+\!+} [p_1 \cdots p_n])\ [\,] \right\}$$
$$\left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{DO}\ n : c_1)\ a_1\ [\nu_2, p_1 \cdots p_n] \\ \nu_2 \mapsto \mathbf{C}\ z\ c_2\ a_2 \end{array} \right\}, (z < n)$$
$$\Rightarrow \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{EVAL} : \mathsf{DO}(n - z) : c_1)\ a_1\ [\nu_3, p_{z+1} \cdots p_n] \\ \nu_3 \mapsto \mathbf{F}\ c_2\ (a_2 {+\!+} [p_1 \cdots p_z])\ [\,] \end{array} \right\}$$

Fig. 4. The instruction for performing a general tail-call.

Figure 5 gives the state transition rules for the instructions that evaluate a graph to normal form. These rules require the addition of a *dump stack*, d, to the machine state. The EVAL instruction examines the node at the top of the evaluation stack. If it is a constant value node or a canonical application node then EVAL does nothing. If it is a frame node then EVAL saves the current point of reduction on the dump stack before moving it to the frame node. The RET instruction restores the current point of reduction from the dump stack, and leaves a pointer to a result node on its evaluation stack.

Figure 6 gives state transition rules for the instructions that perform case-analysis. The DUP instruction duplicates a pointer to a constructed value node on the evaluation stack, and the TAG instruction pops this pointer and pushes the node's constructor number onto the dump stack. The CASE instruction uses the constructor number to choose which branch to execute next. The SPLIT instruction then provides access to the arguments of the constructed value node

$$
\begin{array}{ll}
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{EVAL} : c_1)\ a_1\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ k\ a_2 \end{array} \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c_1\ a_1\ (\nu_2 : s) \right\}, d\rangle \\
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{EVAL} : c_1)\ a_1\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{C}\ z\ c_2\ a_2 \end{array} \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c_1\ a_1\ (\nu_2 : s) \right\}, d\rangle \\
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{EVAL} : c_1)\ a_1\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{F}\ c_2\ a_2\ [\,] \end{array} \right\}, d\rangle & \Rightarrow \langle \left\{ \begin{array}{l} \nu_1 \mapsto \mathbf{F}\ c_1\ a_1\ s \\ \underline{\nu_2} \mapsto \mathbf{F}\ c_2\ a_2\ [\,] \end{array} \right\}, (\nu_1 : d)\rangle \\
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{RET} : c_1)\ a_1\ (p : s_1) \\ \nu_2 \mapsto \mathbf{F}\ c_2\ a_2\ s_2 \end{array} \right\}, (\nu_2 : d)\rangle & \Rightarrow \langle \left\{ \begin{array}{l} \nu_1 \mapsto \mathbf{F}\ c_1\ a_1\ s_1 \\ \underline{\nu_2} \mapsto \mathbf{F}\ c_2\ a_2\ (p : s_2) \end{array} \right\}, d\rangle
\end{array}
$$

Fig. 5. Instructions for performing evaluation.

by copying them into free argument slots of the frame node at the current point of reduction. A JMP instruction is sometimes used to join up control again.

$$
\begin{array}{ll}
\langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{DUP} : c)\ a\ (p : s) \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ (p : p : s) \right\}, d\rangle \\
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{TAG} : c)\ a_1\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ k\ a_2 \end{array} \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ a_1\ s \right\}, (k : d)\rangle \\
\langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{CASE}\ l_0 \cdots l_n : \cdots l_k : c_k)\ a\ s \right\}, (k : d)\rangle & \\
\qquad \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c_k\ a\ s \right\}, d\rangle & \\
\langle \left\{ \begin{array}{l} \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{SPLIT}\ n\ k : c)\ a_1\ (\nu_2 : s) \\ \nu_2 \mapsto \mathbf{V}\ k\ a_2 \end{array} \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ (a_1 {+\!+} a_2)\ (\nu_2 : s) \right\}, d\rangle \\
\langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ (\mathsf{JMP}\ l_k : \cdots l_k : c)\ a\ s \right\}, d\rangle & \Rightarrow \langle \left\{ \underline{\nu_1} \mapsto \mathbf{F}\ c\ a\ s \right\}, d\rangle
\end{array}
$$

Fig. 6. Instructions for performing case-analysis.

Of course, there are also instructions for manipulating basic values. The ADD instruction, for example, pops two integers from the dump stack and pushes their sum.

3 Compilation Rules

As in [AJ89], functional programs are assumed to be transformed into the core functional language shown in Figure 7. A program is a set of functions of varying arity. An expression is an argument or function applied to zero or more other expressions, a data value constructor applied to all of its arguments, or a **case**-expression with simple patterns.

For convenience, Figure 8 collects together the schemes used to compile core functional programs into $\langle \nu, \mathrm{G} \rangle$-machine code. In these schemes, ρ is an environment mapping variables to argument slots, and n is the number of the next

$$
\begin{array}{lll}
p ::= & f_1^{m_1}\ x_{11} \cdots x_{1m_1} = e_1 & \\
& \vdots & \\
& f_n^{m_n}\ x_{n1} \cdots x_{nm_n} = e_n & \\
\\
e ::= & x\ e_1 \cdots e_m & (m \geq 0) \\
\mid & f_i^k\ e_1 \cdots e_m & (m \geq 0) \\
\mid & c_k\ e_1 \cdots e_m & (m \geq 0) \\
\mid & \textbf{case}\ e\ \textbf{in}\ c_{k_1}\ x_1 \cdots x_m\ :\ e_1 \parallel \cdots\ \textbf{end} &
\end{array}
$$

Fig. 7. The core language.

free slot. Cases that appear not to be covered by the schemes are assumed to have been transformed away before compiling to ⟨ν,G⟩-machine code.

Only the $\mathcal{R}$ scheme is significantly different from that of the original paper. Here, its result need *not* be in normal form because tail calls are dealt with by returning a frame node representing the call, instead of making the call directly. This frame node can be driven to normal form by a loop that is a variant of Steele's "UUO handler" [Ste78]. The $\mathcal{R}$ scheme is the only one that can encounter a tail call, and doing things this way avoids overflowing the Java Virtual Machine's control stack (see Section 5).

As in the original ⟨ν,G⟩-machine, operations on basic values, such as integers and characters, are compiled using a $\mathcal{B}$ scheme similar to that of the ordinary G-machine [Joh84] (not shown in Figure 8).

4 The Java Virtual Machine

This section describes the machine architecture and instruction set of the Java Virtual Machine [LY96].

4.1 Machine Architecture

A Java program is organised into *classes*, which have *methods* for performing computation and describe the structure of *objects*. For every class, the Java compiler produces a file containing Java Virtual Machine code for the methods and a *constant pool* of literals, such as numbers and strings, used by this code. The local state of a method invocation is stored on the Java Virtual Machine stack. It consists of the actual parameters and local variables, and a small operand stack for the intermediate results of expression evaluations. An object is a record whose fields may be either scalar values, methods or references to other objects. Storage for objects is allocated from heap memory and later recovered by garbage collection.

$$
\begin{array}{l}
\mathcal{F}\ [\![\ f_i^k\ x_1 \cdots x_k = e\ ]\!] = \mathcal{R}\ [\![\ e\ ]\!]\ [\ x_1 = 0 \cdots x_k = (k-1)\ ]\ k \\
\\
\mathcal{R}\ [\![\ x\ ]\!]\ \rho\ n = \mathsf{GETSLOT}\ \rho(x);\ \mathsf{RET} \\
\mathcal{R}\ [\![\ x\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \mathcal{C}\ [\![\ e_m\ ]\!]\ \rho\ n; \cdots \mathcal{C}\ [\![\ e_1\ ]\!]\ \rho\ n; \mathcal{E}\ [\![\ x\ ]\!]\ \rho\ n; \mathsf{DO}\ m; \mathsf{RET} \\
\mathcal{R}\ [\![\ f_i^k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \begin{cases} \mathcal{C}[\![\ f_i^k\ e_1 \cdots e_m]\!]\ \rho\ n; \mathsf{RET} & (m \le k) \\ \\ \mathcal{C}[\![\ e_m]\!]\ \rho\ n;\ \cdots\ \mathcal{C}[\![\ e_{k+1}]\!]\ \rho\ n; & (m > k) \\ \mathcal{E}[\![\ f_i^k\ e_1 \cdots e_k]\!]\ \rho\ n; \mathsf{DO}\ (m-k); \mathsf{RET} \end{cases} \\
\mathcal{R}\ [\![\ c_k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \mathcal{C}\ [\![\ c_k\ e_1 \cdots e_m\ ]\!]\ \rho\ n; \mathsf{RET} \\
\mathcal{R}\ [\![\ \mathbf{case}\ e\ \mathbf{in}\ c_{k_1}\ x_1 \cdots x_m\ :\ e_1 \| \cdots\ \mathbf{end}\ ]\!]\ \rho\ n \\
\qquad = \mathcal{E}\ [\![\ e\ ]\!]\ \rho\ n; \mathsf{DUP}; \mathsf{TAG}; \mathsf{CASE}\ l_1, \cdots l_k; \\
\qquad\ \ l_1 : \mathsf{SPLIT}\ n\ m; \\
\qquad\qquad \mathcal{R}[\![\ e_1]\!]\ (\rho + [x_1 = n, \cdots x_m = n+m-1])\ (n+m) \\
\qquad\ \ \cdots \\
\\
\mathcal{E}\ [\![\ x\ ]\!]\ \rho\ n = \mathsf{GETSLOT}\ \rho(x); \mathsf{EVAL}; \mathsf{DUP}; \mathsf{PUTSLOT}\ \rho(x) \\
\mathcal{E}\ [\![\ f_i^k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \mathcal{C}\ [\![\ f_i^k\ e_1 \cdots e_m\ ]\!]\ \rho\ n; \mathsf{EVAL} \\
\mathcal{E}\ [\![\ c_k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \mathcal{C}\ [\![\ c_k\ e_1 \cdots e_m\ ]\!]\ \rho\ n \\
\mathcal{E}\ [\![\ \mathbf{case}\ e\ \mathbf{in}\ c_{k_1}\ v_1 \cdots v_m\ :\ e_1 \| \cdots\ \mathbf{end}\ ]\!]\ \rho\ n \\
\qquad = \mathcal{E}\ [\![\ e\ ]\!]\ \rho\ n; \mathsf{DUP}; \mathsf{TAG}; \mathsf{CASE}\ l_1, \cdots l_k; \\
\qquad\ \ l_1 : \mathsf{SPLIT}\ n\ m; \\
\qquad\qquad \mathcal{E}[\![\ e_1]\!]\ (\rho + [x_1 = n, \cdots x_m = n+m-1])\ (n+m); \\
\qquad\qquad \mathsf{JMP}\ l_x \\
\qquad\ \ \cdots \\
\qquad\ \ l_x : \\
\\
\mathcal{C}\ [\![\ x\ ]\!]\ \rho\ n = \mathsf{GETSLOT}\ \rho(x) \\
\mathcal{C}\ [\![\ f_i^k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \begin{cases} \mathsf{NEWCAP}\ f; \mathcal{C}[\![\ e_m]\!]\ \rho\ n;\ \cdots\ \mathcal{C}[\![\ e_1]\!]\ \rho\ n; & (m < k) \\ \mathsf{UPDCAP}\ f\ m & \\ \\ \mathsf{NEWFRM}\ f; \mathcal{C}[\![\ e_m]\!]\ \rho\ n;\ \cdots\ \mathcal{C}[\![\ e_1]\!]\ \rho\ n; & (m = k) \\ \mathsf{UPDFRM}\ f\ m & \end{cases} \\
\mathcal{C}\ [\![\ c_k\ e_1 \cdots e_m\ ]\!]\ \rho\ n = \mathsf{NEWVAL}\ k\ m; \mathcal{C}\ [\![\ e_m\ ]\!]\ \rho\ n; \cdots \mathcal{C}\ [\![\ e_1\ ]\!]\ \rho\ n; \mathsf{UPDVAL}\ k\ m
\end{array}
$$

Fig. 8. The compilation schemes.

4.2 Instruction Set

Although the Java Virtual Machine has many instructions, we need only a few of them here. Each Java Virtual Machine instruction can be described by a *state transition rule* of the form $s_1 \rightarrow s_2$. A state is a five-tuple consisting of the *instruction stream* c, the *object heap*, h, the *local variables*, v, the *operand stack*, s, and the *control stack* d.

Figure 9 gives the state transition rules for the `new` and `checkcast` instructions that allocate an object in the heap and check that the object is of the expected type.

$$\langle\ (\texttt{new}\ n : c),\ h,\ v,\ s,\ d\ \rangle \rightarrow \langle\ c,\ h + [x \mapsto n],\ v,\ (x : s),\ d\ \rangle$$
$$\langle\ (\texttt{checkcast}\ n : c),\ [\ldots x \mapsto n \ldots],\ v,\ (x : s),\ d\ \rangle \rightarrow \langle\ c,\ [\ldots x \mapsto n \ldots],\ v,\ (x : s),\ d\ \rangle$$

Fig. 9. Instructions for allocating and checking the type of an object.

Figure 10 gives the state transition rules for the `aload` and `astore` instructions that move object references between the local variables and the operand stack.

$$\langle\ (\texttt{aload}\ i : c),\ h,\ [\ldots v_i \ldots],\ s,\ d\ \rangle \rightarrow \langle\ c,\ h,\ [\ldots v_i \ldots],\ (v_i : s),\ d\ \rangle$$
$$\langle\ (\texttt{astore}\ i : c),\ h,\ [\ldots v_i \ldots],\ (x : s),\ d\ \rangle \rightarrow \langle\ c,\ h,\ [\ldots x \ldots],\ s,\ d\ \rangle$$

Fig. 10. Instructions for moving objects between local variables and the stack.

Figure 11 gives the state transition rules for the `dup`, `swap` and `sipush` instructions that duplicate, exchange and push values on the operand stack.

$$\langle\ (\texttt{dup}{:}c),\ h,\ v,\ (x : s),\ d\ \rangle \rightarrow \langle\ c,\ h,\ v,\ (x : x : s),\ d\ \rangle$$
$$\langle\ (\texttt{swap}{:}c),\ h,\ v,\ (x : y : s),\ d\ \rangle \rightarrow \langle\ c,\ h,\ v,\ (y : x : s),\ d\ \rangle$$
$$\langle\ (\texttt{sipush}\ i : c),\ h,\ v,\ s,\ d\ \rangle \rightarrow \langle\ c,\ h,\ v,\ (i : s),\ d\ \rangle$$

Fig. 11. Instructions for manipulating the operand stack.

Figure 12 gives the state transition rules for the `goto` and `lookupswitch` instructions that transfer control within the Java Machine Code for a method.

$$\langle\ (\texttt{goto}\ l_k \ldots l_k : c_k \ldots c),\ h,\ v,\ s,\ d\ \rangle \rightarrow \langle\ c_k,\ h,\ v,\ s,\ d\ \rangle$$
$$\langle\ (\texttt{lookupswitch}\ l_0 \ldots l_n \ldots\ l_k : c_k),\ h,\ v,\ (k : s),\ d\ \rangle \rightarrow \langle\ c_k,\ h,\ v,\ s,\ d\ \rangle$$

Fig. 12. Instructions for transferring control.

Figure 13 gives the state transition rules for the instructions that invoke a method and return its result. The instruction `invokespecial` p invokes the initialisation method `<init>` with signature p of an object, and the instruction `invokevirtual` (m, p) invokes method m with signature p of an object. The `areturn` instruction returns an object reference from a method invocation.

$$\langle\ (\texttt{invokespecial}\ ((p_1 \ldots p_n)p_r) : c),\ h,\ v,\ (a_n \ldots a_1 : x : s),\ d\ \rangle$$
$$\rightarrow \langle\ jvmcode(p.\texttt{<init>}),\ h,\ [a_1 \ldots a_n],\ [\,],\ ((c, v, s) : d)\ \rangle$$
$$\langle\ (\texttt{invokevirtual}\ (m, (p_1 \ldots p_n)p_r) : c),\ h,\ v,\ (a_n \ldots a_1 : x : s),\ d\ \rangle$$
$$\rightarrow \langle\ jvmcode(x.m),\ h,\ [a_1 \ldots a_n],\ [\,],\ ((c, v, s) : d)\ \rangle$$
$$\langle\ (\texttt{areturn}{:}c),\ h,\ v,\ (x : s),\ ((c_1, v_1, s_1) : d)\ \rangle$$
$$\rightarrow \langle\ c_1,\ h,\ v_1,\ (x : s_1),\ d\ \rangle$$

Fig. 13. Instructions for invoking methods and returning results.

Figure 14 gives the state transition rule for the `getfield` $(f.r)$ instruction that accesses a field f with type r of an object x known only at run-time.

$$\langle\ (\texttt{getfield}\ (f, r) : c),\ [\ldots x \ldots],\ v,\ (x : s),\ d\ \rangle \rightarrow \langle\ c,\ [\ldots x \ldots],\ v,\ (x.f : s),\ d\ \rangle$$

Fig. 14. Instruction for accessing a field of an object.

5 Conversion to Java Virtual Machine Code

The conversion of ⟨ν,G⟩-machine code to Java Virtual Machine code uses the Java class for graph nodes, `N`, shown in Figure 15, with subclasses for constructed value, frame and canonical application nodes.

5.1 Constructed Value Node Classes

Constructed value node classes are provided as part of the run-time system, written in Java. Figure 16 shows the class `V3`. The constructor method initialises k and the argument slots. Both the evaluation method, `eval`, and the evaluation step method, `ev1`, are trivial because the node is in normal form. The `tag` method just returns the constructor number. All other methods are illegal.

```
abstract public class N {
    public abstract N eval();
    public abstract N ev1();
    public abstract N do1(N g);
    public abstract int tag();
    public abstract N cpy();
}
```

Fig. 15. Java code for the `N` class.

5.2 Frame Node Classes

A frame node class is produced by the compiler for every function in the program. Figure 17 shows the Java equivalent of the class `f_frm` produced for the function `f x y z = z`. The constructor method just initialises the argument slots. Two methods are used to perform evaluation. The first, the evaluation step method, `ev1`, evaluates the frame node upto a constructor or a tail call. The second, the evaluation method, `eval`, uses the evaluation step method and a loop to evaluate the frame node to normal form. The loop is a version of Steele's "UUO handler" [Ste78]. By returning a node object whose `ev1` method is to be invoked instead of invoking this method directly, one can implement tail calls without overflowing the Java Virtual Machine's control stack. The evaluation step method has three stages. First, the frame node is *blackholed* by setting all of its argument slots to `null`. Black-holing is a way of avoiding spack leaks through nodes that will eventually be updated [Jon92]. Next, the result node is obtained by running the frame node's code. Finally, the frame node is updated by setting the indirection field, `ind` to point to the result node. The more common way of updating, by copying the normal form node over the application node, is impossible because the Java Virtual Machine provides no way to copy one object over another (the merits of both approaches are considered in [Pey87]). The frame node's code appears as a class method, obtained by translating $\langle\nu,\mathrm{G}\rangle$-machine-code into Java Virtual Machine code using the rules given in Figure 18. Of course, our compiler uses the more efficient versions of the Java Virtual Machine instructions, such as `aload0` and `bipush`, whenever possible.

To preserve laziness, special treatment must be given to *constant applicative forms* (or CAFs). The indirection field of the frame node for a CAF is made a *class variable* shared between all instances of the class, so that as soon as one instance of the CAF has been updated, all the others "feel the benefits".

5.3 Canonical Application Node Classes

A canonical application node class is also produced for every function in the program. Figure 19 shows the Java equivalent of the class `f_cap` produced for

```
final public class V3 extends N implements Cloneable {
    public int k; public N s0, s1, s2;

    public V3(N a2, N a1, N a0, int k) {
        this.k = k; this.s0 = a0; this.s1 = a1; this.s2 = a2;
    }

    public N eval() { return this; }

    public N ev1() { return this; }

    public N do1(N g) { RT.Stop("V3.do1()"); return null; }

    public N cpy() { RT.Stop("V3.cpy()"); return null; }

    public int tag() { return this.k; }
}
```

Fig. 16. Java code for the `V3` class.

the function `f` above. Constructor methods are provided for the different numbers of arguments that the function `f` could take with some still missing (here, zero, one and two). Each initialises the missing argument count, `z`, and the first few argument slots. Again both the evaluation method, `eval`, and the evaluation step method, `ev1`, are trivial because the node is in normal form. The do step method, `do1`, extends a copy of a canonical application node with another argument, and returns either the canonical application node or a frame node if there are no longer any arguments missing. This is a rather clumsy way to implement the DO instruction, but it is not supposed to happen very much [AJ89].

6 Benchmarks

A Haskell to Java Virtual Machine code compiler has been constructed using the ideas described above, based on version 0.9999.4 of the Chalmers HBC compiler. Table 1 compares the execution speed of Haskell programs run by version 1.2 Beta 2 of the SUN Java Virtual Machine with those run by version 1.3 of the Nottingham Hugs interpreter. The SUN Java Virtual Machine can either interpret Java Virtual Machine code (INT), or compile it to SPARC machine code "just-in-time" (JIT). These figures were recorded on a SUN UltraSPARC 1 workstation with a 140MHz processor, 64Mbytes of memory. and version 2.5.1 of the Solaris operating system. The first three benchmarks are small programs; the remainder are large ones from the `nofib` suite [Par92]. For our compiler, the timings are the best of three runs after an initial "warm up" run; for Hugs, they are the best of three runs, each made immediately after starting the interpreter.

```
final public class f_frm extends N implements Cloneable {
    N ind, s0, s1, s2;

    public f_frm(N a2, N a1, N a0) {
        this.s0 = a0; this.s1 = a1; this.s2 = a2;
    }

    public N eval() {
        N x, y;
        x = this.ev1(); do { y = x; x = x.ev1(); } while (y != x);
        return x;
    }

    public N ev1() {
        if (this.ind != null) return this.ind.ev1();
        else {
            N a0 = this.s0; this.s0 = null;
            N a1 = this.s1; this.s1 = null;
            N a2 = this.s1; this.s2 = null;
            return this.ind = this.code(a0, a1, a2);
        }
    }

    public N do1(N g) { return this.eval().do1(g); }

    public N cpy() { RT.Stop("f_frm.cpy()"); return null; }

    public int tag() { RT.Stop("f_frm.tag()"); return 0; }

    static N code(N a0, N a1, N a2) {
        return a2; /* generated code for function body */
    }
}
```

Fig. 17. Java code for the `f_frm` class.

$\mathcal{T}$ ⟦ CASE $l_0, \cdots l_k$ ⟧ = `lookupswitch` $l_0, \cdots l_k$
$\mathcal{T}$ ⟦ DO n ⟧ = `invokevirtual (cpy,()N)`
`swap; invokevirtual (do1,(N)N)` (this line repeated n times)
$\mathcal{T}$ ⟦ DUP ⟧ = `dup`
$\mathcal{T}$ ⟦ EVAL ⟧ = `invokevirtual (eval,()N)`
$\mathcal{T}$ ⟦ GETSLOT i ⟧ = `aload` i
$\mathcal{T}$ ⟦ JMP l ⟧ = `goto` l
$\mathcal{T}$ ⟦ NEWCAP f ⟧ = `new` f`_cap; dup`
$\mathcal{T}$ ⟦ NEWFRM f ⟧ = `new` f`_frm; dup`
$\mathcal{T}$ ⟦ NEWVAL k m ⟧ = `new V`m`; dup`
$\mathcal{T}$ ⟦ PUTSLOT i ⟧ = `astore` i
$\mathcal{T}$ ⟦ RET ⟧ = `areturn`
$\mathcal{T}$ ⟦ SPLIT n m ⟧ = `checkcast V`m;
`dup; getfield (s`$_0$`,N); astore` n;
...
`dup; getfield (s`$_{m-1}$`,N); astore` $(n+m-1)$
$\mathcal{T}$ ⟦ TAG ⟧ = `invokevirtual (tag,()I)`
$\mathcal{T}$ ⟦ UPDCAP f_k m ⟧ = `invokespecial ((N`$_1 \cdots$ `N`$_m$`)V)`
$\mathcal{T}$ ⟦ UPDFRM f_k m ⟧ = `invokespecial ((N`$_1 \cdots$ `N`$_m$`)V)`
$\mathcal{T}$ ⟦ UPDVAL k m ⟧ = `sipush` k; `invokespecial (V`m`,(N`$_1 \cdots$ `N`$_m$`I)V)`

Fig. 18. Instruction translation rules.

Benchmark	Hugs Time (s)	Ours (SUN INT) Time (s)	Ours (SUN JIT) Time (s)
calendars	2.1	3.2	8.1
clausify	4.0	5.9	8.3
soda	0.5	1.2	1.0
bspt	22.7	81.6	89.3
infer	6.5	125.0	119.1
parser	11.7	47.6	56.1
prolog	8.7	73.6	79.2
reptile	3.8	10.7	*fails*

Table 1. Benchmark figures for the SUN Java Virtual Machine.

```
final public class f_cap extends N implements Cloneable {
    int z; N s0, s1;

    public f_cap() { this.z = 3; }

    public f_cap(N a0) { this.z = 2; this.s0 = a0; }

    public f_cap(N a1, N a0) { this.z = 1; this.s0 = a0; this.s1 = a1; }

    public N eval() { return this; }

    public N ev1() { return this; }

    public N do1(N g) {
        switch (--this.z) {
        case 0 : return new f_frm(g, this.s1, this.s0);
        case 1 : this.s1 = g; return this;
        case 2 : this.s0 = g; return this;
        }
        return null; /* not reached */
    }

    public N cpy() {
        try { return (N) super.clone(); }
        catch (CloneNotSupportedException e) {
            RT.Stop("f_cap.cpy()"); return null;
        }
    }

    public int tag() { RT.Stop("f_cap.tag()"); return 0; }
}
```

Fig. 19. Java code for the `f_cap` class.

Disappointingly, our compiler produces programs that run between 3 and 18 times more slowly than with Hugs when the Java Virtual Machine code is interpreted, and often run more slowly still when it is compiled "just-in-time". There are several reasons for this. Most importantly, we have done small experiments that show that memory allocation/reclamation is around *an order of magnitude* more expensive in the SUN Java Virtual Machine than in the Hugs run-time system. Since lazy functional language implementations allocate so much memory, this really hurts performance.

Next, the Java Virtual Machine performs/requires run-time checks that are unnecessary for strongly-typed functional languages. Although memory is never accessed through a `null` pointer, and the code that splits apart a constructed

data value never gets one of the wrong sort, this is checked anyway as the program runs. Also, there are the run-time support routines. In the Hugs implementation they are written in C and compiled once; in our implementation they are written in Java and either interpreted or compiled "just-in-time", which is more expensive.

Finally, the Java Virtual Machine uses dynamic linking: at the start of each program run, method references are resolved to memory addresses by loading files from disk. In our benchmarking, we have tried to reduce the cost of dynamic linking as much as possible be getting the computer's operating system to cache class files in memory during the "warm-up" run, so that it does not have to access the disk again.

One might hope that other "just-in-time" compilers would do better than the one from SUN, but unfortunately this is not so. Table 2 compares the execution speed of Haskell programs run with the Symantec "just-in-time" compiler (part of SUN's version 1.1.6 Java distribution) and the Microsoft "just-in-time" compiler (part of Microsoft's version 1.1 Visual J++ system). These figures were recorded on a Gateway 2000 Solo notebook computer with a 166MHz processor, 48Mbytes of memory, and version 4.0 of the Windows NT workstation operating system. A full investigation must wait for another time, but clearly these compilers are optimising for a very different instruction mix from the one that they are getting.

Benchmark	Hugs Time (s)	Ours (ST JIT) Time (s)	Ours (MS JIT) Time (s)
`calendars`	1	7	54
`clausify`	3	5	52
`soda`	5	2	3
`bspt`	16	198	2,237
`infer`	5	267	1,743
`parser`	8	118	865
`prolog`	7	203	1,299
`reptile`	4	19	228

Table 2. Benchmark figures for the Symantec and Microsoft Java Virtual Machines.

7 Related Work

In a previous paper, we described a Haskell to Java Virtual Machine code compiler using the G-machine [Wak97]. The problem with that work was the big Java array used for the pointer stack. It proved to be a severe source of space leaks because none of the nodes accessible from the array could be recovered by the Java Virtual Machine garbage collector. Looking for a way to avoid space leaks by splitting the big stack array into a linked-list of small ones, we soon arrived at an abstract machine remarkably like the $\langle \nu, \mathrm{G} \rangle$-machine.

Through personal communications, we have become aware of a number of other researchers who have considered compiling lazy functional languages to Java Virtual Machine code. At the University of Nijmegen, Rinus Plasmeijer tried some experiments with a view to developing a compiler for the lazy functional language Clean, but concluded that the Java Virtual Machine was not yet efficient enough to make it worthwhile. A prototype Haskell compiler based on the Glasgow Spineless Tagless G-machine was produced by Mark Tullsen at Yale University [Tul97]. But for even the small benchmarks he tried, Tullsen found it hard to get good performance from the Java Virtual Machine, and is no longer working on the compiler. Gary Meehan and Mike Joy have produced a G-machine based compiler for their pure, lazy, weakly-typed language, Ginger [MJ98]. This compiler produces Java Virtual Machine code programs with the same order of performance as our own, but which do not appear to be properly tail-recursive.

There has also been some work on compiling strict functional languages for the Java Virtual Machine. Nick Benton of Persimmon IT has a compiler for Standard ML, and Cygnus Support have a compiler for Scheme.

Aside from Java, other high-level languages have been compiled to Java Virtual Machine code. Intermetrics have a compiler for Ada [Taf96], and the GNU Project has a compiler for Eiffel [CC98]. Thorn surveys a number of other languages and systems for mobile code in [Tho97].

8 Conclusions

In this paper we have shown how lazy functional programs, such as Haskell, can be made mobile by compiling them for the Java Virtual Machine. Basing the compiler on a version of the $\langle \nu, \mathrm{G} \rangle$-machine is not the obvious thing to do, but it leads to a particularly elegant set of translation rules. Indeed, we find it hard to imagine a cleaner implementation. Currently, the speed of the resulting Java Virtual Machine code programs is disappointing — programs run between 3 and 30 times more slowly than with an ordinary interpreter. This is largely due to the surprisingly high cost of memory allocation/reclamation in current implementations of the Java Virtual Machine. Other functional language implementors have built prototype compilers and found (but sadly not documented) much the same problem. In future work, we intend to find out whether this high cost is a property of the Java Virtual Machine's design or its implementation.

Acknowledgements

Our thanks to Lennart Augustsson and Thomas Johnsson, whose work on the HBC compiler forms the basis of our own, and to the anonymous referees who suggested many useful improvements to the paper.

This work was supported by Canon Research Centre Europe Limited.

References

AG96. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, May 1996. 335

AJ89. L. Augustsson and T. Johnsson. Parallel Graph Reduction with the ⟨ν,g⟩-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 202–213. ACM Press, 1989. 335, 336, 339, 345

CC98. D. Colnet and S. Collin. SmallEiffel: the GNU Eiffel Compiler, April 1998. 350

Joh84. T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 58–69, June 1984. 335, 340

Jon92. R. Jones. Tail Recursion Without Space Leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992. 344

LY96. T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, September 1996. 335, 340

MJ98. G. Meehan and M. Joy. Compiling Lazy Functional Programs to Java Bytecode. Technical report, Department of Computer Science, University of Warwick, May 1998. submitted for publication. 350

Par92. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer-Verlag, 1992. 345

Pey87. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. 344

PH97. J. Peterson and K. Hammond, editors. *Haskell 1.4: A Non-strict, Purely Functional Language*. The Haskell Committee, April 1997. 335

Ste78. G. L. Steele. Rabbit: A Compiler for Scheme. Technical Report AI-TR-474, MIT Laboratory for Computer Science, 1978. 340, 344

Taf96. S. Tucker Taft. Programming the Internet in Ada95. In *Ada Europe Conference*, June 1996. 350

Tho97. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, September 1997. 350

Tul97. M. Tullsen. 690 Project: Compiling Haskell to Java, September 1997. 350

Wak97. D. Wakeling. A Haskell to Java Virtual Machine Code Compiler. In *Draft Proceedings of the 1997 Workshop on the Implementation of Functional Languages*, pages 71–84, September 1997. 336, 349

Program Analysis in λProlog

John Hannan

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
http://www.cse.psu.edu/~hannan

High-level languages support a rich syntactic structure involving bound variables and a variety of binding constructs. This structure contributes significantly to the need for sophisticated program analyses during compilation. Providing high-level, declarative descriptions of these analyses is an important step towards designing these analyses and implementing them efficiently. Deductive systems, utilizing a rich collection of logical constants, provide a suitable formalism for specifying such descriptions, and the programming language λProlog provides a means for implementing and experimenting with such deductive systems.

The suitability of λProlog for this task can be traced to two aspects of the language. First, λProlog supports a high-level abstract syntax for programs in which all binding and scoping constructs can be represented using λ-terms. This representation supports a uniform treatment of α-conversion and substitution for all such constructs. Second, λProlog supports the direct encoding of deductive systems as programs. The language exploits the structure of the high-level abstract syntax and supports the construction of programs which manipulate bound variables and substitutions in a simple and uniform manner. Hypothetical and parametric reasoning allow complex tasks to be specified without the use of explicit contexts or side conditions. The resulting programs offer clear, concise, and logical descriptions of program analysis tasks, including semantic definitions, optimizations, and translations. The logical nature of these specifications supports reasoning about their correctness.

We study the use of λProlog to specify the abstract syntax of simple programming languages, the operational semantics of these languages, and various translations and static analyses for these languages. Additionally, we discuss how to reason about the correctness of some of these specifications. We focus on a simple, functional language, but these techniques can be applied to a wide range of languages.

References

1. John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, January 1991. Available as MS-CIS-91-09.
2. John Hannan. Implementing λ-calculus reduction strategies in extended logic programming languages. In L. Hallnäs, editor, *Proceedings of the Second Workshop on Extensions to Logic Programming*, pages 193–219. Springer-Verlag LNCS 596, 1992.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 353–354, 1998.

3. John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, 1993.
4. John Hannan and Patrick Hicks. Higher-order unCurrying. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, January 1998.
5. John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.
6. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
7. Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, pages 379–388. IEEE Press, 1987.
8. Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 810–827. MIT Press, 1988.

A Game Semantics Foundation for Logic Programming
(Extended Abstract)

Roberto Di Cosmo[1], Jean-Vincent Loddo[2], and Stephane Nicolet[1]

[1] DMI-LIENS
Ecole Normale Supérieure
45, Rue d'Ulm, 75005 Paris, France
[2] Université de Paris VII
2, Place Jussieu, 75005 Paris, France

Abstract. We introduce a semantics of Logic Programming based on an classical Game Theory, which is proven to be sound and complete w.r.t. the traditional operational semantics and Negation as Failure. This game semantics is based on an abstract reformulation of classical results about two player games, and allows a very simple characterization of the solution set of a logic program in terms of approximations of the value of the game associated to it, which can also be used to capture in a very simple way the traditional "negation as failure" extensions. This approach to semantics also opens the way to a better understanding of the mechanisms at work in parallel implementations of logic programs and in the operational semantics of logic programs with negative goals. [1]

1 Introduction

Game theory has found, in recent years, various applications in the research field of programming languages semantics, so that game theory is a very active research subject in computer science. After the preliminary works of Lamarche [12], Blass [5] and Joyal [11] in the early 90s, the works of Abramsky, Malacaria and Jagadeesan [2] lead to the first fully abstract semantics for functional (PCF) or imperative (Idealized Algol) languages. Then, more recently, specialists of Linear Logics got interested in links between games and the geometry of interaction [16], whereas Curien and Herbelin showed that certain classical abstract machines could be interpreted in terms of games [7].

On the down side, all these relevant works seem to use the *vocabulary* of games (player, move, game, strategy) more than the results and the techniques of traditional Game Theory: typically, nobody is interested to know, in those games, if there is a winner, and what he wins; the focus there is on the dynamic aspect of player *interaction*, and *game composition*, not on the *gain*.

[1] A full version of the paper is available as
http://www.dmi.ens.fr/~dicosmo/plgames.ps.gz

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 355–373, 1998.

This should not be taken as a criticism, but as proof of the richness of Game Theory, which can be useful even when one only takes its vocabulary: the generality of the concepts it manipulates (arenas, multiple and independent agents, strategies of cooperation or of non-cooperation, quantification of the remuneration after each game) and their intuitive nature, already provide a powerful metalanguage that allows many of the aspects of modern programming languages to be tackled.

Nevertheless, it seems to us of the highest interest to be able to show, for the first time, that classical notions like payoff, propagation functions and evaluation of a game tree are not sterile in the semantics of programming languages. In this paper, the *gain* will be the central notion, while the interaction part, like in classical Game Theory, will be rather trivial.

We will first present in this paper a simple framework for classical two player games, in which well known results from Game Theory, like the monotonicity of evaluation by subgames, can be reformulated in an abstract way, via the notion of *evaluation structure* and then take advantage of this abstract reformulation to apply these results no longer to real numbers representing monetary payoffs, but to algebraic structures built out of a notion which is at the core of logic programming: idempotent substitutions.

We will see how a logic program can be interpreted as a set of rules for a two player game and how traditional semantics like the minimum Herbrand model, the computed answers, and even negation as failure can all be recovered by means of optimistic or pessimistic approximation of the value of the game tree over a suitable domain of values (typically a set of sets of idempotent substitutions). We will also hint at some new possible semantics suggested by taking as value domain the complete lattice of idempotent substitutions described in [15].

Let us say first of all that our goal is not to prove again some well established theorems in Logic Programming, but to show that a simple, natural, intuitive framework based on the classical notion of game can be used to recover them, in an unified framework.

The slogan *"each logic program is a game"* arose by a sort of a side effect: while working on bisimulation games for concurrency theory [13], we implemented a Prolog prototype which turned out to be very simple (only six lines of code). Actually, it was *too* simple: it was impossible not to have the clear feeling that we had come across, in one way or the other, a structural shift of the problem, so we looked more closely at the mechanisms at work, and the outcome of this investigation was the foundation of a game semantics for logic programming that we present in this paper.

Some Basic Notions of Game Theory

Game theory is the branch of mathematics which tries to model and to understand the behavior that rational agents should choose in a competitive arena. We want to informally recall here some basic intuitive notions of game theory which form the basis of the next section and corresponds, mainly, to the intu-

itive idea of randomized strategic games and refer the interested reader to the comprehensive handbook [1] for a recent introduction to the topic.

Classically, a *game* is given extensionally as a tree in which the root represents the *initial position* of a play, the nodes are *positions* and edges are *moves*. In a general n-players game, there are $n+1$ players (n human players plus an (optional) special one, the *chance* player, used to model random processes, like throwing a dice), and nodes are labeled with the name of the player whose turn is to play in that position. The *payoff function* is a function giving terminal positions p an n-ary vector of *values* (usually reals), in which the j^{th} component represents the amount paid (or received) by player j when the game ends in position p. A *strategy* for player j is a function which associates to each of the (human) player's node exactly one of his moves in that player's position: it describes the way he will behave in a play. The most studied games in Game Theory have been the two players, zero-sum games: for them, the *Minimax Theorem* (von Neumann, 1928 [17]) states that the dual approaches of *maximizing* one's gains or *minimizing* one's losses lead to the same expected value for each player, so each game has *one* optimal value, the *minimax* value. This can be rephrased in terms of game trees by saying that one can compute the value of a two-player, zero-sum game by assigning the payoff values to the terminal nodes of the game tree, and propagating them bottom-up applying the *Max* (resp. *Min*) *propagation functions* to the values of the player (resp. opponent) subtrees.

It is important to note that the choice of the *Max* and *Min* functions depends on the game: usually they are the maximum and minimum over a set of reals, but there are well-known cases where a different function is needed. For example, in the presence of random events (dices etc.), the mean value is used instead of the maximum. Similarly, the game presented in this paper uses substitutions as values and intersection of substitutions as the *Max* function. In the next section, we will rephrase and formalize these notions for two player games in an abstract setting, and clearly separate the notion of game rules, and game tree from the notion of evaluation.

Structure of the Paper

The paper is organized as follows: in Section 2 we set up the formal framework for abstract two player games, and reformulate and prove in this framework some classical results related to approximations and strategies. Then we introduce the game associated to a logic program in Section 3, and turn to show how, with the proper choice of evaluation structures defined in Section 4, several classical semantics can be recovered (Sections 5 and 6). Finally, we hint at some new semantics and various applications in the Conclusions 7.

2 An Abstract Framework for Two Player Games

In this section we introduce a general framework which is quite convenient to describe the large class of two-player games. We will introduce in an abstract way the general notions of a *game tree*, *evaluation of a game tree*, *strategy* and *winning strategy*, and a few key lemmas that are at the core of the approximation procedures of infinite (or very large) game trees. This will allow us later on to easily deal with various variants of games and valuations specialized for Logic Programming.

2.1 Game Tree

The intuitive notion of a game is best described as a tree, which is a convenient way, given the rules of the game, to describe the set of *all* possible plays starting from a given initial position, so that a particular play is just a branch of the tree. Informally, a (possibly infinite) Game Tree is a tree whose nodes are divided into two sets of *Player* and *Opponent* nodes, representing Player and Opponent positions, and whose edges are labeled by moves. This tree is usually given implicitly by defining the "rules of the game", which is just a function P assigning to each Player or Opponent position the set of legal moves, and an initial position G. From these elements, it is easy to build the whole tree $\Gamma(P, G)$. We will denote game trees using (possibly) infinite *terms* built via the constructors $\texttt{PLeaf}(G)$ and $\texttt{OLeaf}(G)$ (where G is a terminal position) , the n-ary constructors $\texttt{P}$ and $\texttt{O}$, and the binary constructor $\texttt{Label}$ used to label an edge in the tree with the corresponding move of the game (we will use often a more convenient notation for labeling egdges, writing for example $\texttt{P}(\frac{m_1}{\Gamma_1}, .., \frac{m_n}{\Gamma_n})$ instead of $\texttt{P}(\texttt{Label}(m_1,\Gamma_1), .., \texttt{Label}(m_n,\Gamma_n))$. Typically, when drawing such trees, a Player node is depicted by a circle, while an Opponent node is depicted by a square.

2.2 Game Evaluation

The major aim of game theory is to study the *value* associated to a game, traditionally representing a sort of "best" gain Player can expect to get given an initial position. This *value of the game* is obtained by combining in some way all possible gains Player can obtain in all possible plays starting from the given initial position. Traditionally, this is done by assigning a gain (or **payoff**) from a given domain D, usually the real numbers, only to the final positions, by means of a function $EvaluateTerminal$, and these values are then propagated bottom up to the root by means of two functions $\sqcup$ and $\sqcap$ which give the value of an internal Player or Opponent node, knowing the values of its sons. We require that the domain of values be a partial order $(D, \leq)$ with a maximum and a minimum element $\top, \bot$. We will additionally require $(D, \leq)$ to be ω-chain complete (i.e. any ascendent or descendent chain has *lub* and *glb* respectively) to deal with infinite game trees, but for finite game trees this is not necessary. The propagations functions $\sqcup$ and $\sqcap$ are associative and commutative variable n-ary

operations ($n \geq 1$) that can be seen as generated by their binary version, so we will often give only the binary version in what follows.
In general it is important to take into account not only the strength of the terminal position, but also the way by which this position has been reached: for this, we add to the payoff of a terminal position a component which depends on the path to that node, *i.e* the sequence of moves of the play. This can be stated in an elegant way by saying that, at each turn, a player has to pay something to stay in the game, so that each move of the game has a local payoff (given by a function $Price$), which gets accumulated as the play proceeds by means of a function $+$. More formally, an *evaluation structure* for a game is a 9-uple collecting the above elements:

$$(D,\ \top,\ \bot,\ \leq,\ \sqcup,\ \sqcap,\ +,\ EvaluateTerminal,\ Price)$$

Notation 1 *When interested in the value of a game tree, we will often need to label edges not only with the moves, but also with their associated local payoffs. Since we are more interested in payoffs than move names, we will just annotate the edges with their local payoffs, and, with a slight abuse of notation, write* $\mathtt{P}(\frac{Price_1}{\Gamma_1}, .., \frac{Price_n}{\Gamma_n})$ *instead of* $\mathtt{P}(\frac{(m_1,Price_1)}{\Gamma_1}, .., \frac{(m_n,Price_n)}{\Gamma_n})$.

Given an evaluation structure, it is possible to compute the value of a finite game tree as follows:

Definition 1 (Value of a finite game tree).
The value of a finite game tree is defined in a bottom-up fashion from the leaves to the root, by first giving values to the terminal nodes and then combining the values of the subtrees as follows:

- $\mathbf{val}\,(t) = EvaluateTerminal\,(t)$ *if* t *is a leaf.*
- $\mathbf{val}\,(t) = \sqcup(x_1 + \mathbf{val}\,(t_1), \cdots, x_n + \mathbf{val}\,(t_n))$
  if the root of t *is a* `P` *node and* $x_i = Price(m_i)$ *are the prices of the moves* m_i *leading to opponent positions* t_i.
- $\mathbf{val}\,(t) = \sqcap(x_1 + \mathbf{val}\,(t_1), \cdots, x_n + \mathbf{val}\,(t_n))$
  if the root of t *is an* `O` *node and* $x_i = Price(m_i)$ *are the prices of the moves* m_i *leading to player positions* t_i.

For example, the famous Minimax algorithm simply computes the value obtained by choosing as evaluation structure $(R, +\infty, -\infty, \leq, max, min, +, h, zero)$, where $zero$ is the constant zero function and h is the function, often named *heuristic* for historical reasons, that evaluates the leaves for the given game. The game presented in this paper (see section 3 for details) will also fit into this general framework, with the construction of the game tree imposed by the rules of the logic program (this can been seen as a syntaxic analysis of the logic program) and the evaluation of the game giving the semantics of the program. In fact, we will show that different choices for the evaluations structure allow us to recover different traditional semantics (see section 4 for two usual semantics, namely the denotation of computed answers and the SLDNF extension), and suggest new ones (see 7).

2.3 Infinite Game Trees and Approximations of a Game Value

If the propagation functions used to evaluate a game tree have good regularity properties (if they are all monotone, for instance), it is possible to get *approximations* of the true value of a game tree by giving approximate values to some of the leaves or some of the internal nodes, instead of fully evaluating the tree. This is the key idea for defining the value of infinite game trees as the limit of a sequence of approximations, and comes directly from traditional practice of games : for instance, the size of the chess game tree is so big that it can be considered infinite for all practical purpose, so real chess playing programs cut it at a certain level and provide an approximation evaluation of the tree using appropriate heuristics for the values of cut nodes. In general, if we do not know anything about the heuristic function used, we cannot have any idea of the quality of the approximation. However, if we know that the bias it introduces has always the same sign (that is, if we know the heuristic function always overestimate or always underestimate the true value of the node), then we can know the bias of the approximation of the root value with respect to the true (unknown) value, by using the following key lemma :

Lemma 1 (monotonicity of sub-games).
If the functions $+$, $\sqcup$ *and* $\sqcap$ *are monotone (non decreasing) w.r.t.* $\leq$, *then for any context tree* $\Gamma[.]$ *and any game trees* Γ_1 *and* Γ_2,

$$\mathbf{val}\,\Gamma_1 \leq \mathbf{val}\,\Gamma_2 \Longrightarrow \mathbf{val}\,\Gamma[\Gamma_1] \leq \mathbf{val}\,\Gamma[\Gamma_2]$$

Proof. The composition of non decreasing function is still a non decreasing function.

Using the above lemma, it is possible to approximate the value of an infinite or very large game tree with a sequence of finite approximations. An approximation is built in two steps: first one applies a function $Cut(k, \_)$ (which is formally defined in the long version of the paper) to the tree to cut it at the k^{th} Opponent level (that is, at depth $2k$), replacing all non-leaf subtrees with a special constant Ω, which represents the interruption of the tree development process.

Then one approximates the values of the Ω nodes using values with known bias and propagate them to the root to obtain an approximation of the tree value. For example, using $\top$ (resp. $\bot$) one gets the greatest (resp. lowest) approximation, that we call here optimistic (resp. pessimistic).

Definition 2 (Pessimistic and optimistic approximations of the game value).
The function $\mathbf{val}_{pess}$ *is an extension of the function* $\mathbf{val}$ *on the game trees containing the constant* Ω, *which is evaluated by the worst possible value, that is by setting* $\mathbf{val}_{pess}\Omega = \bot$. *The dual optimistic version* $\mathbf{val}_{opt}$ *evaluates* Ω *with the best possible value, that is by setting* $\mathbf{val}_{opt}\Omega = \top$. *Given a game tree* Γ, *we denote* $\mathbf{val}^k_{pess}\Gamma$ *the approximation* $\mathbf{val}_{pess}Cut(k, \Gamma)$ *and* $\mathbf{val}^k_{opt}\Gamma$ *the dual approximation* $\mathbf{val}_{opt}Cut(k, \Gamma)$.

Two fundamental properties of the approximations are the following

Lemma 2 (Monotonicity of progressive approximations). *If the functions* $+$, $\sqcup$ *and* $\sqcap$ *are non decreasing, then the function* $\mathbf{val}^k_{pess}$ *of a game tree is increasing (monotone) with respect to the cut level* k*, whereas the function* $\mathbf{val}^k_{opt}$ *is decreasing.*

Proof. Using lemma 1.

When a game tree Γ is finite then exists a minimum natural number h such that $\Gamma = Cut(h, \Gamma)$, and we will call this integer the *depth* of Γ.

Evaluation of an Infinite Game Tree We can now consider the problem of defining the value of an infinite game-tree. We have established the monotonicity of finite approximations, so we could define the value of an infinite tree as the limit of the approximate values, but then we may get different results depending on the choice of the value for the Ω nodes. Indeed, we will define two approximate values, one pessimistic, and the other optimistic, that may or may not coincide. Notice that to ensure the existence of such limits, we need to require our value domain $(D, \leq)$ to be ω-chain complete (this is the case for R and for the evaluation structures we will use for Logic Programming).

Lemma 3. *If the functions* $+$, $\sqcup$ *and* $\sqcap$ *are non decreasing, then*

- *if the game tree* Γ *is finite and has depth* h*, then*

$$\mathbf{val}^k_{pess}\Gamma \leq \mathbf{val}\,\Gamma \leq \mathbf{val}^k_{opt}\Gamma \qquad \forall k \geq 0$$

$$\mathbf{val}^k_{pess}\Gamma = \mathbf{val}\,\Gamma = \mathbf{val}^k_{opt}\Gamma \qquad \forall k \geq h$$

- *if the game tree* Γ *is infinite, then*

$$\mathbf{val}^k_{pess}\Gamma \leq \mathbf{val}^k_{opt}\Gamma \qquad \forall k \geq 0$$

so, assuming $(D, \leq)$ *is an* ω*-chain complete domain, if we define* $\mathbf{val}^\infty_{pess}\Gamma$ *and* $\mathbf{val}^\infty_{opt}\Gamma$ *as the limits of corresponding chains, we obtain*

$$\mathbf{val}^\infty_{pess}\Gamma \leq \mathbf{val}^\infty_{opt}\Gamma$$

2.4 Strategies

Finally, we recall another central notion of game theory, which relates game trees to rational behaviour: that of a *strategy* for Player or for Opponent. Intuitively, a strategy for one of the players determines uniquely a move to play in any given position. Formally, a Player (Opponent) strategy is a subtree of a game tree which is deterministic on the Player (Opponent) levels. We write $\Phi(t)$ for the Player strategies in a game tree t and $\Psi(t)$ for the Opponent strategies in a game tree t.

Remark 1 (Depth and value of a finite strategy). Finite strategies are just finite game trees of a special shape, so the definition of depth and of the value of a game tree extends naturally to strategies.

Now, we introduce the notion of *winning* strategy for the Player and the dual notion of *winning* strategy for the Opponent.

Definition 3 (Winning Strategies).

- *A winning strategy for Player in a (possibly infinite) game tree* Γ *is a finite player strategy* $\varphi \in \Phi(\Gamma)$ *such that* $\mathbf{val}\,\varphi \neq \bot$. *The set of all winning strategies of depth* k *will be denoted by* W_k^Γ, *and* $W^\Gamma = \bigcup_k W_k^\Gamma$.
- *A winning strategy for Opponent in a (possibly infinite) game tree* Γ *is a finite opponent strategy* $\psi \in \Psi(\Gamma)$ *such that* $\mathbf{val}\,\psi \neq \top$. *The set of all winning strategies of depth* k *for the Opponent will be denoted by* L_k^Γ, *and* $L^\Gamma = \bigcup_k L_k^\Gamma$.

A strategy fixes the behaviour of one of the players, so the value of a strategy provides an approximation from below of the value of the game for that player, and under some additional assumptions on the propagation functions (weak or strong distributivity), this approximated value can be tightly related to the approximation obtained by cutting the trees at a fixed depth.

Definition 4 (Distributive properties of propagation functions). *We will say that a binary operator* g $\leq$-distributes *(resp.* $\geq$-distributes*) with respect to another binary operator* f *if*

$$(\mathrm{x}\ \mathrm{f}\ \mathrm{y})\ \mathrm{g}\ \mathrm{z} \leq (\mathrm{x}\ \mathrm{g}\ \mathrm{z})\ \mathrm{f}\ (\mathrm{y}\ \mathrm{g}\ \mathrm{z}) (\textit{resp.} (\mathrm{x}\ \mathrm{f}\ \mathrm{y})\ \mathrm{g}\ \mathrm{z} \geq (\mathrm{x}\ \mathrm{g}\ \mathrm{z})\ \mathrm{f}\ (\mathrm{y}\ \mathrm{g}\ \mathrm{z})),$$

and we will say that g strongly distributes *(or distribute, for short) with respect to* f *if it both* $\leq$*-distributes and* $\geq$*-distributes w.r.t.* g.

For example, *min* and *max* are functions that mutually strongly distribute.

Theorem 2 (Distributivity and strategies).

- *if* $\bot$ *is the neutral element of* $\sqcup$ *and* $\sqcap$ $\geq$-distributes *w.r.t.* $\sqcup$, *then* $\bigsqcup_{\varphi \in W_k^\Gamma} \mathbf{val}\,\varphi \leq \mathbf{val}^k_{pess}\Gamma$
- *if* $\top$ *is the neutral element of* $\sqcap$ *and* $\sqcup$ $\leq$-distributes *w.r.t.* $\sqcap$, *then* $\mathbf{val}^k_{opt}\Gamma \leq \bigsqcap_{\psi \in L_k^\Gamma} \mathbf{val}\,\psi$

Moreover, if the operations strongly distribute, then we can write equality in place of inequality.

Proof. By induction on k using the distributivity properties of $\sqcup$ and $\sqcap$.

3 The Rules of the Game

There are traditionally two classical views of a logic program, depending on whether each clause $B \leftarrow A_1, \ldots, A_n$ in it is interpreted from right to left or from left to right:

- the logical vision: if $A_1 \wedge \cdots \wedge A_n$ then B
- the effective vision: to compute B, one must first compute $A_1, \cdots, A_n$. Another way of saying it is that B is a procedure whose body is $A_1, \cdots, A_n$

Our game theoretic alternative will be the following: to each logic program we associate a simple two player game, in which one of the players tries to prove the goal and his opponent tries to disprove it: the beauty of the approach is that the logical rules written by the programmer are *just* the rules of the game, and this allows us to get a simpler understanding of the mechanisms involved in logic programming.

Nevertheless, the game we define here give rise, for a given goal G, to a game tree which is *not* the naïve one obtained by taking the tree of all possible SLD derivations starting from G; we want to clearly separate in our framework the different features making up an SLD derivation: the construction of the tree, the visit of the tree and its evaluation. For this, we use player positions consisting of just one atom, and check the consistency of solutions using the propagation function in the opponent nodes of the game tree.

We can now give a formal definition for the game $\Gamma(P)$ associated to a logic program P and the game tree $\Gamma(P, G)$ for the game $\Gamma(P)$ starting at position G.

Definition 5 (Game of a logic program).
The two-player game associated to a logic program P has the following components:

Positions : *a* player position *is an* atomic formula A*, while an* opponent position *is a* sequence *of atomic formulae.*

Initial position : *only opponent positions are legal starting positions of a game, and Opponent begins.*

Player moves : *Player moves by choosing a (variant of a) rule $H \leftarrow A_1, .., A_n$ in P whose head H unifies with the current position A. This leads to an opponent position $(A_1\theta, .., A_n\theta)$, where θ is the mgu of A and H. If no rule unifies with the current position, the game stops in a Player leaf.*

Opponent moves : *Opponent moves by selecting one of the subgoals A_i of an opponent position $G = A_1, .., A_n$, and the game continues in the player position A_i. If the current opponent position is the empty sequence, the game stops in a Opponent leaf.*

Now that we know what are the Player and Opponent nodes, and we know how to build the sons of a Player and an Opponent position, we immediately

know what is the game tree associated to a given program P and starting position G, we will write $\Gamma(G, P)$ with a little abuse of notation (confusing the logic program P with the "game rules" function P of the previous section).

Notice that the game tree built this way is *not* a complete SLD-tree, which can be obtained as the game tree of a *different* game where player and opponent positions are *both* sequences of formulae and the rules are resolution steps over sequence of atomic formulae (this different game is hinted at in [9]). Our game tree is much more compact, as shown in 5.3: in fact, each SLD-tree describes a particular *visit* of the game tree, given the chosen selection rule which imposes the order of the visit of the sub-game trees at Opponent's nodes.

Notice that, as a consequence of this compactness of representation, it is not enough, for a goal to be provable, that Player always arrives to a terminal winning position: he also needs to have used compatible payoffs. In other terms, it will be the "value" of the game tree, for a given evaluation structure, that is related to provability. We can see all this in the following example, where we use informally the evaluation structure which will be formally defined in 4.

Example 1 (A simple example). Consider the logic program P consisting of the following three rules

```
1.  path(X,Z) :- arc(X,Y), path(Y,Z)
2.  path(X,X)
3.  arc(a,b)
```

This program P defines the game $\Gamma(P)$. If we try to satisfy the goal path(X,b) in P, we use $\Gamma(P)$ with the initial position $path(X, b)$ and get the game tree $\Gamma(P, path(X, b))$, which will be the regular infinite tree of figure 1, where the

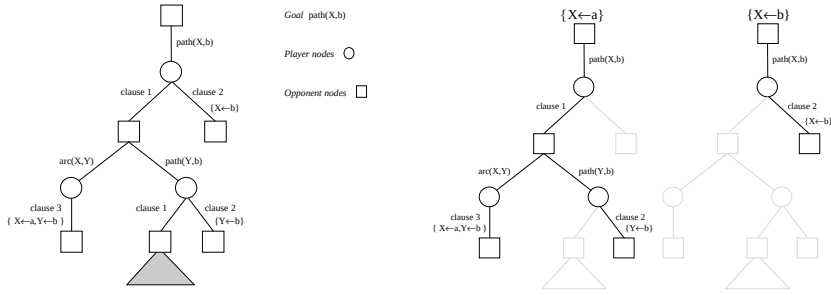


Fig. 1. An example game tree and two winning strategies.

gray subtree is equal to $\Gamma(P, path(X, b))$ up to the renaming of X to Y.

Remark 2. As usual, we use variants of program rules obtained by replacing the free variables of the rule with fresh variables. Here, we will also require that each application of a node formation rule uses a different supply of fresh variables.

4 Evaluation Structures for Traditional Semantics

It is now time to introduce the evaluation structures that can be used to recover the different traditional semantics. To recover the computed answer semantics and SLDNF, we choose as values sets of idempotent substitutions, ordered by inclusion

It is worth noticing that a slightly different evaluation structure, where values are multisets of idempotent substitutions can be used to give the semantics of actual prolog machines.

4.1 An Evaluation Structure for Computed Answers Semantics and SLDNF

We are interested here in handling the calculated substitutions of a query of a logic program, so we will construct our domain using the *complete lattice of idempotent substitutions* (introduced in [10] and [15]), in which each point of the lattice is a class of equivalent (up to renaming) idempotent substitutions.

domain of values : $\top$ will be the (infinite) set of all idempotent substitutions, $\bot$ will be the empty set $\emptyset$, and D will be just the powerset $(2^{\top}, \subseteq)$ of subsets of $\top$ ordered by inclusion.

payoffs : The first player unable to move will *lose*, so $EvaluateTerminal$ is defined as $EvaluateTerminal(\texttt{PLeaf}(A)) = \bot = \emptyset$ independently from the player terminal position A, and $EvaluateTerminal(\texttt{OLeaf}(\square)) = \top$. The local payoff of the moves are defined as follow :

- the Player has to perform an unification step to choose a rule of the logic program, so the *payoff* of his move will just be the *mgu* θ used in this unification.
- the Opponent has no price to pay, so his local payoff is always ϵ, the identity substitution, and we will omit the labeling of Opponent moves altogether in what follows

accumulation of payoffs : The accumulation function is the usual composition of substitution, extended to sets of substitutions, that is $S_1 \cdot S_2 = \{\sigma_1 \cdot \sigma_2) \mid \sigma_1 \in S_1, \sigma_2 \in S_2\}$, so that $\top = \{\epsilon\} \cdot \top$ and $\emptyset = \emptyset \cdot \top$.

propagation functions : The propagation functions $\sqcup$ and $\sqcap$ are defined as $\sqcup = \cup$ (the usual union of sets) and $\sqcap = \uparrow$, the extension to sets of substitutions of the operator of parallel composition of substitions defined in [15].

We say that two substitutions θ_1 and θ_2 are *compatible* if $\theta_1 \uparrow \theta_2 \neq \emptyset$. As usual when working with logic programing, we will assume that we only keep the relevant parts of the game values.

Remark 3. It can been seen that the value of a finite game (and also the finite approximations of an infinite game) is either $\emptyset$ or can be put in the form $\{\theta_1, \ldots, \theta_n\} \cdot \top$ with $n \geq 1$. In particular, the value of a winning strategy for Player is always $\{\theta\} \cdot \top$, and we will allow us a slight abuse of notation by saying shortly that its value is θ. Since the *EvaluateTerminal*fonction is independent from terminal positions, we can forget they and denote leaves only with the constructors `PLeaf`and `OLeaf`.

Using theorem 2, we immediately obtain the following

Proposition 1 (Approximations by cut vs. strategies).

- $\mathbf{val}^k_{pess}\Gamma = \bigcup_{\varphi \in W^\Gamma_k} \mathbf{val}\,\varphi$
- $\mathbf{val}^k_{opt}\Gamma \subseteq \uparrow \bigcup_{\psi \in L^\Gamma_k} \mathbf{val}\,\psi$

Proof. Because $\uparrow$ distributes strongly w.r.t. $\cup$, and $\cup \subseteq$-distributes w.r.t. $\uparrow$.

5 Recovering the Least Herbrand Model through the Pessimistic Approximations

It is well known that some of the traditional semantics of logic programming, like the least Herbrand model, have operational characterizations in terms of SLD-derivations. We show here that it is possible to give an equivalent definition of these denotations in terms of winning strategies. Since the initial positions of the game are just atomic goals, we can freely use in the following the terms "atomic, ground, Herbrand, conjunctive, etc." to characterize them.

The least Herbrand model denotation (success set) of a program P becomes now the set of winning Herbrand positions for Player, that is the Herbrand positions for which he has a winning strategy:

$$W_P = \{A \mid A \text{ Herbrand position s.t. } \exists \text{ a winning player strategy in } \Gamma(P, A)\}$$

For the denotation of computed answers (S-semantics in [4]), we have the most general atomic positions (*i.e.* positions with the form $A(X_1, .., X_n)$), instantiated by the computed answers:

$$W^S_P = \{A\theta \mid A \text{ most general position s.t.} \exists \text{ a winning player strategy in} \Gamma(P, A) \text{ with value } \theta\}$$

We say that W_P and W^S_P are the two *winning position denotations* of a program P, and prove the soundness and the completeness of these game-based denotations with respect to the least Herbrand model O_P and the s-semantics O^S_P, by relating them to the well-known operational characterizations:
$O_P = \{A \mid A \text{ Herbrand atom and } \exists\ A \underset{P}{\Longrightarrow}^* \square\}$
and
$O^S_P = \{A\theta \mid A \text{ most general atom and } \exists\ A \overset{\theta}{\underset{P}{\Longrightarrow}}^* \square\}$.

5.1 Fusion and Splitting of SLD Derivations

In order to relate strategies to SLD-derivations, we need two key results about SLD-derivations, namely that we can fuse two successful derivations and that we can split a successful derivation starting from a conjunctive goal into several successful derivations starting from the components of the conjunction.
These are stated in the following lemmas, which are immediate consequences of work in [15] and [8], but, for completeness, we provide also a self-contained full proof of a more general result in the appendix of the full paper.

Lemma 4 (Goal fusion for successful SLD-derivations).
Suppose that θ_1 *and* θ_2 *are compatible, that* $G_1 \stackrel{\theta_1}{\Longrightarrow}^* \square$ *and that* $G_2 \stackrel{\theta_2}{\Longrightarrow}^* \square$, *then* $(G_1, G_2) \stackrel{\theta_1 \uparrow \theta_2}{\Longrightarrow}^* \square$.

Lemma 5 (Goal split for successful SLD-derivation).
Conversely, if $(G_1, G_2) \stackrel{\theta}{\underset{P}{\Longrightarrow}}^* \square$ *then there exist two shorter SLD-derivations* $G_1 \stackrel{\theta_1}{\underset{P}{\Longrightarrow}}^* \square$, $G_2 \stackrel{\theta_2}{\underset{P}{\Longrightarrow}}^* \square$ *such that* $\theta = \theta_1 \uparrow \theta_2$.

5.2 Soundness and Completeness of the Winning Positions Denotations

We show here first that our game based denotations are sound, that is, whenever a winning strategy for Player exists, we can find a successful SLD-derivation. We then show that they are also complete, that is, that whenever a successful SLD-derivation exists, we can find a winning strategy for Player. What is more, the computed answer and the value of the strategy are the same.

Theorem 3 (From winning strategies to successful SLD-derivations (soundness)).
If there exists a winning strategy for Player with value θ *in* $\Gamma(P, G)$ *then there exists a successful SLD-derivation of* G *in* P *with computed answer* θ.

Proof. The idea of the proof is to construct the SLD-derivation piece by piece, while performing a visit of the winning strategy. Formally, we need an induction on the structure of the strategy and, in fact, since a winning strategy φ is a finite Player-Opponent tree that has no player leaf, we can prove the thesis by induction on the number k of opponent levels of the strategy. Complete details are given in the full version of the paper.

Theorem 4 (From successful SLD-derivations to winning strategies (completeness)).
If $G \stackrel{\theta}{\underset{P}{\Longrightarrow}}^* \square$ *then there exists a winning strategy for Player with value* θ *in the game* $\Gamma(P, G)$.

Proof. Let ξ be a SLD-derivation $G \overset{\theta}{\underset{P}{\Longrightarrow}}{}^{*} \square$. We shall construct a winning strategy in $\Gamma(P, G)$ The proof is by induction on the length l of ξ. Complete details are given in the full version of the paper.

Since we know, by Proposition 1, that $\mathbf{val}^{k}_{pess}\Gamma = \bigcup_{\varphi \in W^{\Gamma}_{k}} \mathbf{val}\,\varphi$, we immediately get the main result of this section:

Theorem 5 (Equivalence of computed answer semantics and the pessimistic value). *The pessimistic approximation of the value of the game associated to a logic program P on an initial position G is exactly the computed answer semantics of G in P.*

One can now trivially recover the least Herbrand model as usual.

5.3 Expressive Power of Winning Strategies

Before turning to Negation as Failure, we pause for a moment to consider the amount of SLD-derivations which are subsumed in a game tree. It turns out that a strategy represents exactly a natural class of equivalent SLD derivations, derived from the notion of *switching* introduced in a classical technical lemma on SLD-derivations (see [3]), which says, informally, that two SLD-derivation steps can be exchanged provided that in the second step an instance of an "old" atom is selected.

Definition 6 (Equivalence of SLD-derivation). *If ξ and ξ' are two SLD-derivations which satisfy the conditions of the switching lemma, then we say that ξ' is a* switch *of ξ and we write $\xi' \leftrightarrow \xi$. By definition $\leftrightarrow$ is a symmetric relation, so we define the equivalence $\approx$ as the reflexive and transitive closure of $\leftrightarrow$.*

Looking better at the proofs of the soundness and completeness results, we see that a winning strategy represents precisely the class of switch equivalent successful SLD-derivations. For a formal proof, we should reconstruct the strategy implicitly associated to an SLD-derivation and prove that any equivalent SLD-derivation corresponds to the same strategy, but here we only want to remark that equivalent SLD-derivations have the same skeleton and the strategy is indeed this skeleton.

It is possible to exactly determine the number of different (but switch-equivalent) SLD-derivations denoted by a strategy. Observe that if an SLD-derivation is the switch of another then they consist of the same number of SLD-steps (the same holds for $\approx$), so all derivations in an equivalence class have the same length. We can then compute the length $\ell(\varphi)$ and the number $\sharp(\varphi)$ of successful SLD-derivations that a strategy denotes.

$$\ell(\varphi) = \begin{cases} 0 & \text{if } \varphi = \texttt{OLeaf}, \\ \sum_{j=1}^{n} (\ell(\varphi_j) + 1) & \text{if } \varphi = \texttt{O}(\texttt{P}(\frac{\theta_1}{\varphi_1}), .., \texttt{P}(\frac{\theta_n}{\varphi_n})) \end{cases}$$

$$\sharp(\varphi) = \begin{cases} 1 & \text{if } \varphi = \texttt{OLeaf} \\ \prod_{j=1}^{n} \sharp(\varphi_j) \cdot \frac{\ell(\varphi)!}{\prod_{j=1}^{n} (\ell(\varphi_j)+1)!} & \text{if } \varphi = \texttt{O}(\texttt{P}(\frac{\theta_1}{\varphi_1}), .., \texttt{P}(\frac{\theta_n}{\varphi_n})) \end{cases}$$

Example 2. Consider a strategy with the structure shown in the following picture (figure 2). We can calculate that this strategy alone represents 13440 different (swith-equivalent) successful SLD-derivations.

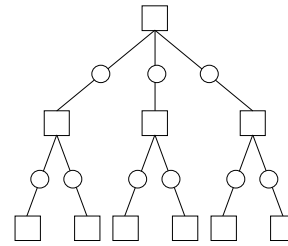


Fig. 2. A strategy with three Opponent levels

SLD Trees, AND/OR Trees Game Trees

Let us use a few words to clarify the differences between our game trees, with their winning strategies, and traditional objects in logic programming like the very old AND/OR trees which were abandoned in favor of SLD trees. First of all, it is true that a *specific* SLD tree, with a fixed selection rule, is as concise as our game trees in representing the full search space, due to the well known independence on the selection rules. But this conciseness comes, for SLD trees, at the price of the introduction of an arbitrary element: the selection rule, precisely. There is no "canonical" SLD tree, while the game tree *is* a canonical object.

On the other hand, one could hold as a criticism to game trees, like to the old AND/OR trees, the fact that there are cases where the game tree is infinite while some specific SLD tree is not, for example, the goal `p(X),q(X)` produces a finite SLD tree with the leftmost selection rule in the program with clauses `p(a)` and `q(b) :- q(b)`, while the game tree is infinite, as it contains a branch for the loop generated by the second clause that the leftmost rule avoids (while a rightmost rule would not).

But the game tree is just a mathematical object that we are interested in evaluating, and an infinite tree *can* sometimes be evaluated in finite time: in this example, the optimistic approximation yields $\emptyset$ already at the first opponent level, so we know the answer is $\emptyset$ *without* building the whole tree.

The advantage of separating the tree as a canonical object from the evaluation, which is the real semantics, is to allow a great deal of flexibility in studying the semantic properties, but one must keep in mind that, in practice, nobody will build the whole tree before evaluating it, but rather incrementally refine an initial evaluation while incrementally building the part of the tree needed for the evaluation: SLD trees are just an example of such exploration procedures.

6 Recovering SLDNF through the Optimistic Approximations

We have seen in the previous section that the pessimistic approximations of the game value (using its characterisation in terms of winning strategies for Player of Theorem 2) allows us to recover traditional semantics of logic programming. Turning our attention to the treatment of negation, we can now show that the sequence of *optimistic* values can be used to describe successive approximations of the greatest fix-point of the logic consequences operator T_P, thus allowing us to use the *very same* game and evaluation structure as above to recover the semantics of negation as failure. More precisely, we will prove that the process of going from an approximation level k to $k+1$ has the same pruning power as an application of T_P to $T_P^k(H)$, starting from the Herbrand base H (or, put in another way, that an application of T_P corresponds exactly to an alternation Player-Opponent in the game process). In the following, we will say that a goal G is *game-refuted at depth* k if $\mathbf{val}^k_{opt}\Gamma(P,G)=\emptyset$. We will use $T_P \downarrow k$ to denote the k^{th} application of T_P, starting from the Herbrand base H.

Let's start by showing that game refutations at depth k are included in finite failures exhibited after k applications of T_P, which gives soundness. We need two lemmas.

Lemma 6 (Monotonicity by generalisation). *If* $\mathbf{val}^k_{opt}\Gamma(P,G) = \emptyset$, *then for all substitution* θ *we have* $\mathbf{val}^k_{opt}\Gamma(P,G\theta) = \emptyset$.

Lemma 7 (Game value of ground positions as binary answers). *If* G *is a ground position, then* $\mathbf{val}^k_{opt}\Gamma(P,G) \in \{\emptyset, \top\}$.

The previous lemma implies that, if $A_1,..,A_n$ are ground positions such that the value $\mathbf{val}^k_{opt}\Gamma(P,(A_1,..,A_n))$ is $\emptyset$, then there exists at least one A_i s.t. $\mathbf{val}^k_{opt}\Gamma(P,A_i) = \emptyset$.

Theorem 6 (Soundness of optimistic semantics with respect to $T_P\downarrow\omega$). *If* A *is an Herbrand atom game-refutated at depth* k, *then* $A \notin T_P \downarrow k$

Proof. The proof proceeds by induction on the approximation level $k \geq 1$ of the game tree. Complete details are given in the full version of the paper.

Having established the soundness, let us turn now to prove completeness. In the following, if S is a set of atoms, we will use the notation $[S]$ to indicate the union of the *ground instances* of the atoms of S. The following lemma is classical :

Lemma 8. *If A unifies with the head of the program clause $B \leftarrow B_1 \cdots B_n$ with the mgu θ and σ is a substitution, then $[A\theta\sigma] \subseteq T_P(\bigcup_{j=1}^{n} B_j\theta\sigma)$*

Theorem 7 (Optimistic approximations *survive* T_P).
Let $v = \mathbf{val}^k_{opt}\Gamma(P, G)$ be the k^{th} optimistic approximation of the value of a game with initial position $G = A_1, .., A_n$. Then $[G \cdot v] \subseteq T_P \downarrow k$

Proof. The proof is by induction on the approximation level $k \geq 0$ of the game tree. Complete details are given in the full version of the paper.

This allows us to finally conclude

Corollary 1 (Completness of optimistic semantics with respect to $T_P \downarrow \omega$). *If A is an Herbrand atom such that $A \notin T_P \downarrow k$, then A is game-refuted at depth k.*

Proof. By contradiction: if we suppose that $v = \mathbf{val}^k_{opt}\Gamma(P, A)$ is not the empty set, then, by previous theorem, $[A \cdot v] \subseteq T_P \downarrow k$. Since A is ground, we have $[A \cdot v] = \{A\}$, so we get $A \notin T_P \downarrow k$, which contradicts the hypothesis.

7 Conclusion and Future Work

We have presented in this paper a game theoretic approach to the semantics of Logic Programming, by providing an abstract framework for two-player games based on the notion of evaluation structure, which allows us to abstractly prove general properties of finite approximations, and we have stated the conditions under which the value of the game can be equivalently formulated using the value sets of strategies or the limit of successive approximations obtained by cutting the tree at deeper and deeper levels. By choosing the proper evaluation structure, this framework can be instantiated to capture various forms of semantics proposed in the literature: from the minimum Herbrand model to the computed answers, to a very elegant treatment of SLDNF using sets of idempotent substitutions, and computed answers with multiplicity using multisets of idempotent substitutions. The key notion here is that of winning strategy, which concisely represents a very large class of equivalent SLD derivations, thus providing a powerful tool to investigate properties of derivations.

This showed a little of the potential of the game theoretic approach, but we are convinced that much more can be achieved by pursuing this direction of research :

– For lack of space, we can only hint here at another evaluation structure, where the domain is precisely the complete lattice of idempotent substitutions introduced by Palamidessi in [15], equipped with $\uparrow$ and $\downarrow$ instead of $\uparrow$ and $\cup$. This domain seems very well adapted to perform abstract interpretation of logic programs to obtain an approximation of the real values.

- The game values of programs are AND-compositional (since it can be seen that $\mathbf{val}_{pess}\Gamma(P,(G_1,G_2)) = \mathbf{val}_{pess}\Gamma(P,G_1) \uparrow \mathbf{val}_{pess}\Gamma(P,G_2)$), but they are not OR-compositional, since $\mathbf{val}_{pess}\Gamma((P \cup Q),G) \neq \mathbf{val}_{pess}\Gamma(P,G) \cup \mathbf{val}_{pess}\Gamma(P,G)$. This leaves open the problem to find an OR-compositional game-based denotation, hopefully more abstract and intuitive than the known ones.
- The relation we have established here between various traditional semantics and two player games could be the basis of a very promising transfer of results from traditional game theory to logic programming, especially in the field of parallel execution, where game theory provides a wealth of efficient execution algorithms that could now be carried over to the Logic Programming community (see [14,6] for comprehensive bibliographies about parallelisation of Logic Programming and parallelisation game-search algorithms)
- To efficiently handle the problem of repeated position in a game, clever programs use a hash-based mechanism to check the linear history of the play for already visited positions. Could this technique be adapted to logic programs? Intuitively, the program $Q(X) \leftarrow Q(X)$ will loop forever for the goal $Q(X)$ because it will go through an infinite sequence of variants of the same position $Q(X)$, although $Q(X)$ is not included in the least fix point semantics of the program. If we code a clause using deBrujin indexes, variants of a same clause are mapped to the same term, so hashing could be applied, like in game programs. Could this be made into a nice semantic framework?
- Finally, the two ways, either pessimistic or optimistic, of defining a cut-approximation computation of game values yields a duality clearly similar to the least fix-point *vs* greatest fix-point duality used to deal with positive and negative interrogation in logic programming, but in a unified framework, so we hope this will provide a means of giving a clean semantics to logic programs with negative goals. After all, the intuitive semantics of $not(A)$ is quite simple in term of games: when Opponent challenges us to show that A is not provable, we just turn the table, give him A, and say "your turn"!

We believe that the game semantic framework holds interesting promises for the Logic Programming community, if it accepts to play the game.

References

1. Aumann and Hart (eds). *Handbook of Game Theory with Economic Applications*. 1992. 357
2. Samson Abramsky and Rhada Jagadeesan. Games and full completeness for multiplicative linear logic. *The Journal of Symbolic Logic*, 59(2):543–574, 1994. 355
3. Krzysztof Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997. 368
4. 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. 366
5. A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:pages 183–220, 1992. 355

6. M.G. Brockington. A taxonomy of parallel game-tree search algorithms. *Journal of the International Computer Chess Association*, 19(3):162–174, 1996. 372
7. P.L. Curien and H. Herbelin. Computing with abstract bohm trees. 1996. 355
8. Levi Comici and Meo. Compositionality of SLD-derivations and their abstractions. *ILPS*, 1995. 367
9. K. Doets. Levationis Laus. *Journal of Logic Computation*, 3(5):pages 487–516, 1993. 364
10. E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, (1):31–46, 1985. 365
11. A. Joyal. Free lattices, communication and money games. *Proceedings of the 10th International Congress of Logic, Methodology, and Philosophy of Science*, 1995. 355
12. F. Lamarche. Game semantics for full propositional linear logic. *LICS*, pages 464–473, 1995. 355
13. Jean Loddo and Stéphane Nicolet. Theorie des jeux et langages de programmation. Technical report TR-98-01, ENS, 45, Rue d'Ulm, 1998. 356
14. G. Levi and F. Patricelli. *Prolog : Linguaggio Applicazioni ed Implementazioni*. Scuola Superiore G. Reiss Romoli, 1993. 372
15. C. Palamidessi. Algebraic properties of idempotent substitutions, *ICALP*, *LNCS*, 443, 1990. 356, 365, 367, 371
16. V. Danos P. Baillot and T. Ehrhard. Believe it or not, AJM's games model is a model of classical linear logic. *LICS*, pages 68–75, 1997. 355
17. J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathaematische Annalen*, (100):195–320, 1928. 357

Controlling Search in Declarative Programs*

Michael Hanus and Frank Steiner

RWTH Aachen, Informatik II, D-52056 Aachen, Germany
{hanus,steiner}@i2.informatik.rwth-aachen.de

Abstract. Logic languages can deal with non-deterministic computations via built-in search facilities. However, standard search methods like global backtracking are often not sufficient and a source of many programming errors. Therefore, we propose the addition of a single primitive to logic-oriented languages to control non-deterministic computation steps. Based on this primitive, a number of different search strategies can be easily implemented. These search operators can be applied if the standard search facilities are not successful or to encapsulate search. The latter is important if logic programs interact with the (non-backtrackable) outside world.
We define the search control primitive based on an abstract notion of computation steps so that it can be integrated into various logic-oriented languages, but to provide concrete examples we also present the integration of such a control primitive into the multi-paradigm declarative language Curry. The lazy evaluation strategy of Curry simplifies the implementation of search strategies, which also shows the advantages of integrating functions into logic languages.

1 Introduction

Computing solutions to partially instantiated goals and dealing with non-deterministic computations via built-in search facilities is one of the most important features of logic languages. Standard logic languages like Prolog use a global backtracking strategy to explore the different alternatives of a computation. This is often not sufficient and a source of many problems:

- If a top-level predicate fails, all alternatives of previously called predicates are also explored. This may lead to an unexpected behavior and makes the detection of programming errors difficult (e.g., if the backtracking is caused by a missing alternative in the top-level predicate). This problem is often solved by inserting "cuts" which, however, decreases the readability of programs.
- Depth-first search is an incomplete strategy. Although this drawback can be managed by experienced programmers, it causes difficulties for beginners (who frequently use predicates like commutativity or left-recursive clauses in the beginning). As a consequence, one is forced to talk about Prolog's

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 374–391, 1998.

depth-first search strategy too early in logic programming courses. This can have a negative impact on the declarative style of programming.
– In larger application programs (e.g., combinatorial problems), other strategies than the standard depth-first search are often necessary. In such cases the programmer is forced to program her own strategies (e.g., by using meta-programming techniques). The possible interaction with the standard strategy can lead to errors which are difficult to find.

These problems can be solved if there is a simple way to replace the standard search strategy by other strategies and to implement new search strategies fairly easy. In this paper we show that this is possible by adding a single primitive operation to control non-deterministic computation steps. This primitive, which is a generalization of Oz's search operator [15], evaluates the program as usual but immediately stops if a non-deterministic step occurs. In the latter case, the different alternatives are returned so that the programmer can determine the way to proceed the computation. Based on this primitive, a number of different search operators, like depth-first search, breadth-first search, `findall`, or the Prolog shell, can be easily implemented. These operators also allow the encapsulation of possible search in local predicates. This feature is important if logic programs interact with the (non-backtrackable) outside world, like file accesses or Internet applications.

In contrast to Oz's search operator [15], which is directly connected to a syntactic construct of the language (disjunctions), our control operator is based on an abstract notion of basic computation steps. Thus, it can be considered as a meta-level construct to control (don't know) non-deterministic computation steps which could be added to logic-oriented languages provided that they offer constraints or equations to represent variable bindings and existential quantification to distinguish variables which can be bound in a local computation. Moreover, we provide a formal connection between the search trees of the base language and the results computed by our search operators. Hence, soundness and completeness results for the base language carry over to corresponding results for particular search strategies based on our control operator.

The next section introduces our notion of computation steps of the base language. The primitive to control non-deterministic computations is described in Section 3. Based on this primitive, we show the implementation of different search strategies in Section 4. The relations of these search strategies with the search trees of the base language are established in Section 5. We show the advantages of a base language with lazy evaluation to provide a simple implementation of search strategies in Section 6. Section 7 compares our techniques with related work, and Section 8 contains our conclusions. Due to lack of space, we omit some details and the proofs of the theorems which can be found in [4,5].

2 Operational Semantics of the Base Language

As mentioned above, the search primitive should control the different non-deterministic steps occurring in a derivation. To abstract from the operational

model of the concrete base language, we only assume that a computation step of the base language reduces an expression (goal) to a disjunction consisting of a sequence of pairs of substitutions (bindings) and expressions (goals), i.e, the *operational semantics of the base language* is defined by a one step relation

$$e \;\Rightarrow\; \sigma_1, e_1 \mid \cdots \mid \sigma_n, e_n$$

where $n \geq 0$, e, e_1,...,e_n are expressions, $\sigma_1, \ldots, \sigma_n$ are substitutions on the free variables in e, and "|" joins different alternatives to a disjunction. A *substitution* is a mapping from variables into terms and we denote it by $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ is the *domain* of σ and $\mathcal{VR}an(\sigma) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$ is its *variable range*, where $\mathcal{V}ar(e)$ denotes the set of all free variables occurring in an expression e. The identity substitution (i.e., the substitution id with $\mathcal{D}om(id) = \emptyset$) is often omitted in computation steps. We call the evaluation step *deterministic* if $n = 1$ and *non-deterministic* if $n > 1$. The case $n = 0$ corresponds to a *failure* and is also written as $e \Rightarrow \mathit{fail}$.

This notion of a computation step makes the possible don't know non-determinism of the base language explicit which will be controlled by our search primitive. A possible don't care non-determinism (e.g., in a concurrent base language) corresponds to an indeterminate definition of "⇒" and will not be controlled by our search primitive. Furthermore, note that this notion of a computation step covers a variety of declarative languages. In functional programming, n is at most 1 (i.e., no non-deterministic step occurs) and all substitutions are the identity since unbound variables do not occur during a computation. In logic programming, e is a goal, $e_1, \ldots, e_n$ are all resolvents of this goal and $\sigma_1, \ldots, \sigma_n$ are the corresponding unifiers restricted to the goal variables (for constraint logic programming, the notion of substitutions must be replaced by constraints).

Since our search control operator will be based on this abstract notion of a computation step of the base language (in contrast to Oz [15]), it is applicable to a variety of (functional, constraint) logic languages. To provide concrete examples and to show the advantages of integrating lazily evaluated functions into a logic language, we present the addition of the search control operator to Curry [3,5], a multi-paradigm declarative language aiming to amalgamate functional, logic, and concurrent programming paradigms. Therefore, we outline in the rest of this section Curry's computation model (details can be found in [3,5]).

Values in Curry are, similarly to functional or logic languages, *data terms* constructed from constants and data constructors. These are introduced through *data type declarations* like[1]

```
data bool    = true | false
data nat     = z    | s(nat)
data list(A) = []   | [A|list(A)]
```

[1] In the following we use a Prolog-like syntax which is slightly different from the actual Curry syntax.

`true` and `false` are the Boolean constants, `z` and `s` are the zero value and the successor function to construct natural numbers,[2] and polymorphic lists (`A` is a type variable ranging over all types) are defined as in Prolog.

A *data term* is a well-typed[3] expression containing variables, constants and data constructors, e.g., `s(s(z))`, `[true|Z]` etc. *Functions* (*predicates* are considered as Boolean functions for the sake of simplicity) operate on data terms. Their meaning is specified by *rules* (or *equations*) of the form l `|` $\{c\}$ `=`r (the condition part "`|` $\{c\}$" is optional) where l is a *pattern*, i.e., l has the form $f(t_1,\ldots,t_n)$ with f being a function, $t_1,\ldots,t_n$ data terms and each variable occurs only once, and r is a well-formed *expression* containing function calls, constants, data constructors and variables from l and c. The *condition* c is a *constraint* which consists of a conjunction of equations and optionally contains a list of locally declared variables, i.e., a constraint can have the form `let` $v_1,\ldots,v_k$ `free in` $\{eq_1,\ldots,eq_n\}$ where the variables v_i are only visible in the equations $eq_1,\ldots,eq_n$. If a local variable v of a condition should be visible also in the right-hand side, the rule is written as l `|` $\{c\}$ `=`r `where` v `free`. A rule can be applied if its condition is satisfiable. A *head normal form* is a variable, a constant, or an expression of the form $c(e_1,\ldots,e_n)$ where c is a data constructor. A *Curry program* is a set of data type declarations and equations.

Example 1. The addition on natural numbers (type `nat` above) is defined by

```
add(z   ,Y) = Y
add(s(X),Y) = s(add(X,Y))
```

The following rules define the concatenation of lists and functions for computing the first and the last element of a list ("`_`" denotes an anonymous variable):

```
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
first([X|_]) = X
last(Xs) | {append(_,[X])=Xs} = X  where X free
```

If the equation `append(_,[X])=Xs` is solvable, then `X` is the last element of `Xs`. □

From a functional point of view, we are interested in computing the *value* of an expression, i.e., a data term which is equivalent (w.r.t. the program rules) to the initial expression. The value can be computed by applying rules from left to right. For instance, to compute the value of `add(s(z),s(z))`, we apply the rules for addition to this expression:

`add(s(z),s(z))` ⇒ `s(add(z,s(z)))` ⇒ `s(s(z))`

[2] Curry has also built-in integer values and arithmetic functions. We use here the explicit definition of naturals only to provide some simple and self-contained examples.

[3] The current type system of Curry is a Hindley/Milner-like system with parametric polymorphism, e.g., a term like `s(true)` is ill-typed and thus excluded.

A *strategy* selects a single function call for reduction in the next step. Curry is based on a *lazy* (*leftmost outermost*) strategy. This also allows the computation with infinite data structures and provides more modularity, as we will see in Section 6. Thus, to evaluate the expression `add(add(z,s(z)),z)`, the first subterm `add(z,s(z))` is evaluated to head normal form (in this case: `s(z)`) since its value is required by all rules defining `add` (such an argument is also called *demanded*). On the other hand, the evaluation of the subterm `append([z],[])` is not needed in the expression `first([z|append([z],[])])` since it is not demanded by `first`. Therefore, this expression is reduced to `z` by one outermost reduction step.

Since Curry subsumes logic programming, it is possible that the initial expression may contain variables. In this case the expression might not be reducible to a single value. For instance, a logic programming system should find values for the variables in a goal such that it is reducible to `true`. Fortunately, it requires only a slight extension of the lazy reduction strategy to cover non-ground expressions and variable instantiation: if the value of a variable argument is demanded by the left-hand sides of program rules in order to proceed the computation, the variable is non-deterministically bound to the different demanded values.

Example 2. Consider the function `f` defined by the rules

```
f(a) = c
f(b) = d
```

(`a`, `b`, `c`, `d` are constants). Then the expression `f(X)` with the variable argument `X` is evaluated to `c` or `d` by binding `X` to `a` or `b`, respectively. Thus, this nondeterministic computation step can be denoted as follows:
`f(X)` $\Rightarrow$ $\{$`X` $\mapsto$ `a`$\}$`c` | $\{$`X` $\mapsto$ `b`$\}$`d`. □

A single *computation step* in Curry performs a reduction in exactly one (unsolved) expression of a disjunction. For inductively sequential programs [1] (these are, roughly speaking, function definitions without overlapping left-hand sides), this strategy, called *needed narrowing* [1], computes the shortest possible successful derivations (if common subterms are shared) and a minimal set of solutions, and it is fully deterministic if variables do not occur.[4]

Functional logic languages are often used to solve equations between expressions containing defined functions. For instance, consider the equation {`add(X,z)=s(z)`} w.r.t. Example 1. It can be solved by evaluating the left-hand side `add(X,z)` to the answer expression {`X` $\mapsto$ `s(z)`}`s(z)` (here we omit the other alternatives). Since the resulting equation is trivial, the equation is valid w.r.t. the computed answer {`X` $\mapsto$ `s(z)`}. In general, an *equation* or *equational constraint* $\{e_1{=}e_2\}$ is satisfied if both sides e_1 and e_2 are reducible to the same data term. Operationally, an equational constraint $\{e_1{=}e_2\}$ is solved by evaluating e_1 and e_2 to unifiable data terms where the lazy evaluation of the expressions is interleaved with the binding of variables to constructor terms [10].

[4] These properties also show some of the advantages of integrating functions into logic programs, since similar properties for purely logic programs are not known.

Thus, an equational constraint $\{e_1{=}e_2\}$ without occurrences of defined functions has the same meaning (unification) as in Prolog.[5] Note that constraints are solved when they appear in conditions of program rules in order to apply this rule or when a search operator is applied (see Section 3). Conjunctions of constraints can also be evaluated concurrently but we omit this aspect here (see [3,5] for more details).

3 Controlling Non-deterministic Computation Steps

Most of the current logic languages are based on global search implemented by backtracking, i.e., disjunctions distribute to the top-level (i.e., a goal $A \wedge B$, where A is defined by $A \leftrightarrow A_1 \vee A_2$, is logically replaced by $(A_1 \wedge B) \vee (A_2 \wedge B)$). As discussed in Section 1, this must be avoided in some situations in order to control the exploration of the search space.

For instance, consider the problem of doing input/output. I/O is implemented in most logic languages by side effects. To handle I/O in a declarative way, as done in Curry, one can use the *monadic I/O* concept [18] where an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). Thus, actions are functions of the type

$$World \rightarrow pair(\alpha, World)$$

(where $World$ denotes the type of all states of the outside world). This function type is also abbreviated by $IO(\alpha)$. If an action of type $IO(\alpha)$ is applied to a particular world, it yields a value of type α and a new (changed) world. For instance, `getChar` of type $IO(Char)$ is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. The important point is that values of type $World$ are not accessible to the programmer — she/he can only create and compose actions on the world. For instance, the action `getChar` can be composed with the action `putChar` (which writes a character to the terminal) by the sequential composition operator `>>=`, i.e., "`getChar >>= putChar`" is a composed action which prints the character typed in the keyboard to the screen (see [18] for more details).

An action, obtained as a result of a program, is executed when the program is executed. Since the world cannot be copied (note that the world contains at least the complete file system or the complete Internet in web applications), an interactive program having a disjunction as a result makes no sense. Therefore, all possible search must be encapsulated between I/O operations. In the following, we describe a primitive operation to get control over non-deterministic computations so that one can encapsulate the necessary search for solving goals.

[5] We often use the general notion of a *constraint* instead of equations since it is conceptually fairly easy to add other constraint structures than equations over Herbrand terms.

3.1 Search Goals and a Control Primitive

Since search is used to find solutions to some constraint, we assume that search is always initiated by a constraint containing a *search variable* for which a solution should be computed.[6] A difficulty is that the search variable may be bound to different solutions (by different alternatives in non-deterministic steps) which should be represented in a single expression for further processing. As a solution, we adapt the idea of Oz [15] and abstract the search variable w.r.t. the constraint to be solved, which is possible in a language providing functions as first-class objects.[7] Therefore, a *search goal* has the function type $\alpha \to Constraint$ where α is the type of the values which we are searching for and *Constraint* is the type of all constraints (e.g., an equation like {`add(X,z)=s(z)`} is an expression of type *Constraint*, cf. [5]). In particular, if c is a constraint containing a variable x and we are interested in solutions for x, i.e., values for x such that c is satisfied, then the corresponding search goal has the form `\`x`->`c (this is the notation for lambda abstractions, e.g., `\X->3*X` denotes an anonymous function which multiplies its argument with `3`). For instance, if we are interested in values for the variable `X` such that the equation `append([1],X)=[1,2]` holds, then the corresponding search goal is `\X->{append([1],X)=[1,2]}`.

To control the non-deterministic steps performed to find solutions to search goals, we introduce a function[8] `try` of type

$$(\alpha \to Constraint) \to list(\alpha \to Constraint)$$

i.e., `try` takes a search goal as an argument and produces a list of search goals. The idea is that `try` attempts to evaluate the constraint of the search goal until the computation finishes or does a non-deterministic step. In the latter case, the computation is immediately stopped and the different constraints obtained by the non-deterministic step are returned. Thus, an expression of the form `try(`g`)` can have the following outcomes:

`try(`g`) = []:` The empty list indicates that the search goal g has no solution. For instance, the expression

```
try(\X -> {1=2})
```

reduces to `[]`. Note that a failure of the search can now be handled explicitly because it does not lead to a failure of the whole computation as it would do without the search operator.

[6] The generalization to more than one search variable is straightforward by the use of tuples.

[7] If the base language does not provide functions as first-class objects, one has to introduce a special language construct to denote the search variable, like in Prolog's `setof` or `findall` predicate.

[8] If the base language does not provide functions, like Prolog, we can also implement `try` as a binary predicate where the second argument denotes the result.

`try`(g) = `[`g'`]`: A one-element list indicates that the search goal g has a single solution represented by the element of this list. For instance, the expression

```
try(\X -> {[X]=[0]})
```

reduces to `[\X->{X=0}]`. Note that a solution, i.e., a binding for the search variable like a substitution $\{x \mapsto t\}$, can always be represented as an equational constraint $\{x{=}t\}$. In the following, we denote by $\overline{\sigma}$ the *equational representation* of the substitution σ.

`try`(g) = `[`g_1`,...,`g_n`]`, $n > 1$: If the result list contains more than one element, the evaluation of the search goal g requires a non-deterministic computation step. The different alternatives immediately after this non-deterministic step are represented as elements of this list, where the different bindings of the search variable are added as constraints. For instance, if the function `f` is defined as in Example 2, then the expression

```
try(\X -> {f(X)=d})
```

reduces to the list `[\X->{X=a,c=d}, \X->{X=b,d=d}]`. This example also shows why the search variable must be abstracted: the alternative bindings cannot be actually performed (since a variable is only bound to at most one value in each computation thread) but are represented as equational constraints in the search goal. Note that the search goals in the result list are not further evaluated. The further evaluation can be done by a subsequent application of `try` to the list elements. This allows the explicit control of the strategy to explore the search tree. It will be discussed in more detail in Section 4.

3.2 Local Variables

Some care is necessary if free variables occur in a search goal, as in

```
\E -> {append(L,[E])=[3,4,5]}                                    (*)
```

To compute the last element `E` of the list `[3,4,5]` with this goal, the variable `L` must be instantiated which is problematic since `L` is free. There are different possibilities to deal with this case. In Prolog's `bagof`/`setof` predicates, free variables are (possibly non-deterministically!) instantiated and then remain instantiated with this value, which does not help to really encapsulate all search and sometimes leads to unexpected results. Another ad-hoc method is to consider a `try` application to a search goal containing free variables as a run-time error. Since Curry as well as most Prolog systems is equipped with coroutining facilities, we take another solution and require that the `try` operator *never* binds free variables of its search goal. If it is necessary to bind a free variable in order to proceed a `try` evaluation, the computation suspends. Thus, a `try` application to the search goal $(*)$ cannot be evaluated and suspends until the variable `L` is bound.

$$\texttt{try}(g) = \begin{cases} \texttt{[]} & \text{if } \{c\} \Rightarrow \mathit{fail} \\ \texttt{[}g'\texttt{]} & \text{if } \{c\} \Rightarrow \sigma\,\{\} \text{ (i.e., } \sigma \text{ is a mgu for all equations in } c\text{) with} \\ & \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\}, \\ & g' = \texttt{\textbackslash}x\texttt{->let}\, x_1, \ldots, x_n\, \texttt{free in}\, \{\overline{\sigma}\} \\ \texttt{try(}g'\texttt{)} & \text{if } \{c\} \Rightarrow \sigma\,\{c'\} \text{ with } \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\} \\ & \text{and } g' = \texttt{\textbackslash}x\texttt{->let}\ x_1, \ldots, x_n, y_1, \ldots, y_m\ \texttt{free in}\ \{\overline{\sigma}, c'\} \\ & \text{where } \{y_1, \ldots, y_m\} = \mathcal{VR}an(\sigma) \setminus (\{x, x_1, \ldots, x_n\} \cup \mathcal{V}ar(g)) \\ \texttt{[}g_1\texttt{,...,}g_k\texttt{]} & \text{if } \{c\} \Rightarrow \sigma_1\,\{c_1\}\,|\cdots|\,\sigma_k\,\{c_k\},\ k > 1, \text{ and, for } i = 1, \ldots, k, \\ & \mathcal{D}om(\sigma_i) \subseteq \{x, x_1, \ldots, x_n\} \text{ and} \\ & g_i = \texttt{\textbackslash}x\texttt{->let}\ x_1, \ldots, x_n, y_1, \ldots, y_{m_i}\ \texttt{free in}\ \{\overline{\sigma_i}, c_i\} \\ & \text{where } \{y_1, \ldots, y_{m_i}\} = \mathcal{VR}an(\sigma_i) \setminus (\{x, x_1, \ldots, x_n\} \cup \mathcal{V}ar(g)) \\ \textit{suspend} & \text{otherwise} \end{cases}$$

Fig. 1. Operational semantics of the `try` operator for $g = \texttt{\textbackslash}x\texttt{->let}\, x_1, \ldots, x_n\, \texttt{free in}\, \{c\}$

To allow possible bindings of unbound variables during a local search, they can be declared as local to the constraint so that they might have different bindings in different branches of the search. For instance, we start the evaluation of

```
try(\E -> let L free in {append(L,[E])=[3,4,5]})          (**)
```

to compute the last element of the list `[3,4,5]`. Now the variable `L` is only visible inside the constraint (i.e., existentially quantified) and can be bound to different values in different branches. Therefore, the expression (∗∗) evaluates to

```
[\E -> let L      free in {L=[], [E]=[3,4,5]},
 \E -> let L,X,Xs free in {L=[X|Xs],[X|append(Xs,[E])]=[3,4,5]\}]
```

The new variables `X` and `Xs` (introduced by unification) are also added to the list of local variables so that they can be further instantiated in subsequent steps.

The exact behavior of the `try` operator is specified in Figure 1. Thus, the search goal is solved (second case) if the constraint is solvable without bindings of global variables. In a deterministic step (third case), we apply the `try` operator again after adding the newly introduced variables to the list of local variables. Note that the free variables $\mathcal{V}ar(g)$ occurring in g must not be declared as local because they can appear also outside of g, and therefore they have to be removed from $\mathcal{VR}an(\sigma)$. In a non-deterministic step (fourth case), we return the different alternatives as the result. If a computation step on the constraint tries to bind a free variable, the evaluation of the `try` operator suspends. In order to ensure that an encapsulated search will not be suspended due to necessary bindings of free variables, the search goal should be a closed expression when a search operator is applied to it, i.e., the search variable is bound by the lambda abstraction and all other variables are existentially quantified by local declarations.

Note that the operational semantics of the `try` operator depends only on the meaning of computation steps of the underlying language. Thus, it can be introduced in a similar way to other logic-oriented languages than Curry.[9] Although the management and testing of local variable bindings look complicated, it can be efficiently implemented by decorating each variable with a declaration level and checking the level in binding attempts (similarly to the implementation of scoping constructs in logic languages [12]). Moreover, the equational representations $\overline{\sigma_i}$ of the substitutions need not be explicitly implemented but can be implicitly represented by binding lists for the variables.

4 Search Strategies

The search control operator `try` introduced in the previous section is a basis to implement powerful and easily applicable search strategies. This section demonstrates the use of `try` to implement some search strategies in Curry. These strategies can be defined in a similar way in other declarative languages. However, we will show in Section 6 that Curry's lazy evaluation strategy can be exploited to simplify the application of search operators.

The following function defines a **depth-first search** strategy which collects all solutions of a search goal in a list:

```
all(G) = collect(try(G))
       where collect([])         = []
             collect([G])        = [G]
             collect([G1,G2|Gs]) = concat(map(all,[G1,G2|Gs]))
```

The auxiliary function `collect` applies recursively `all` to all resulting alternatives of a non-deterministic step and appends all solutions in a single list (`concat` concatenates a list of lists to a single list and `map` applies a function to all elements of a list). Thus, the expression

```
all(\L -> {append(L,[1])=[0,1]})
```

reduces to `[\L->{L=[0]}]`.

Due to the laziness of Curry, search goals with infinitely many solutions cause no problems if one is interested only in finitely many of them. A function which computes only the first solution w.r.t. a depth-first search strategy can be simply defined by

```
once(G) = first(all(G))
```

Note that `once` is a partial function, i.e., it is undefined if `G` has no solution.

The value computed for the search variable in a search goal can be easily accessed by applying it to an unbound variable. For instance, the evaluation of the applicative expression

[9] For concurrent languages, one could modify the definition of `try` such that non-deterministic steps lead to a suspension as long as a deterministic step might be enabled by another computation thread. This corresponds to *stability* in AKL [8] and Oz [15].

```
once(\L -> {append(L,[1])=[0,1]}) @ X
```

(`F@E` denotes the application of a function `F` to some `E`, where `F` and `E` can be arbitrary expressions) binds the variable `X` to the value `[0]`, since the first subexpression evaluates to `\L->{L=[0]}` and the constraint `{X=[0]}` obtained by the application of this expression to `X` can only be solved by this binding. Based on this idea, we can define a function `unpack` that takes a list of solved search goals and computes the list of the corresponding values for the search variable:

```
unpack([]) = []
unpack([G|Gs]) | \{G@X\} = [X|unpack(Gs)]  where X free
```

Now it is simple to define a function similarly to Prolog's `findall` predicate:

```
findall(G) = unpack(all(G))
```

For a search goal without free variables, `findall` explores the search tree (depth first) and collects all computed values for the search variable in a list.

A **bounded search** strategy, where search is performed only up to a given depth n in the search tree, can also be easily implemented when we consider search trees containing only the nodes for non-deterministic steps. This means that search will not end after n arbitrary reduction steps but only after n non-deterministic steps. The following function is very similar to the function `all` but explores the search goal `G` only up to depth `N`.

```
all_bounded(N,G) = if N>1 then collect(try(G)) else []  where
  collect([])         = []
  collect([G])        = [G]
  collect([G1,G2|Gs]) = concat(map(all_bounded(N-1),[G1,G2|Gs]))
```

Note that the algorithm may not terminate if an infinite deterministic reduction occurs (which is seldom in practical search problems) because the search operator will never return a result in this case. The same can happen with the next algorithm implementing a **breadth-first search** strategy that traverses the search tree level by level and each level from left to right, regarding as level n all goals obtained from the search goal after n non-deterministic steps.

```
all_bfs(G) = trygoals([G])  where
   trygoals([])     = []
   trygoals([G|Gs]) = splitgoals(map(try,[G|Gs]),[])
   splitgoals([]             ,Ugs) = trygoals(Ugs)
   splitgoals([[]|Gs]        ,Ugs) = splitgoals(Gs,Ugs)
   splitgoals([[G]|Gs]       ,Ugs) = [G|splitgoals(Gs,Ugs)]
   splitgoals([[G1,G2|G3]|Gs],Ugs) = splitgoals(Gs,
                                        append(Ugs,[G1,G2|G3]))
```

The function `trygoals` applies the search operator to the list of remaining alternatives and scans the result (a list of lists) using the function `splitgoals`, which

removes failures and returns all solutions computed so far. Then the remaining goals, which result from non-deterministic steps, are recursively explored further. Similarly, one can also implement other search strategies like depth-first iterative deepening or best solution search with branch and bound [15]. Moreover, a parallel fair search for the first or all solutions can be implemented with our search primitive and a committed choice [15] (which is also available in Curry). To show the use of encapsulated search to control the failure of computations, we define a function on constraints which implements **negation as finite failure** known from logic programming:

```
naf(C) = {all(\_->{C}) = []}
```

Thus, if `C` is a constraint where all variables are existentially quantified, then `naf(C)` is solvable iff the search space of solving `C` is finite and does not contain any solution.

5 Search Trees and Search Operators

In this section we sketch the connection between the search trees of the base language and the results computed by some of the search operators defined above. More details can be found in [4].

The notion of a *search tree w.r.t.* $\Rightarrow$ can be defined as in logic programming [9], i.e., each node is marked with a constraint, and if an *inner node* N is marked with c and $c \Rightarrow \sigma_1, c_1 \mid \cdots \mid \sigma_k, c_k$ is a computation step of the base language, then N has k sons $N_1, \ldots, N_k$ where N_i is marked with c_i and the edge from c to c_i is marked with σ_i $(i = 1, \ldots, k)$. In case of logic programming, where $\Rightarrow$ denotes a resolution step with all resolvents for a goal, search trees w.r.t. $\Rightarrow$ are similar to SLD-trees [9]. *Leaves* are nodes marked with a constraint c that cannot be further derived. The leaf is *successful* if c is the empty constraint (in this case we call the composition of all substitutions marked along the branch from the root to this leaf a $\Rightarrow$*-computed answer* for the constraint at the root of the tree). The leaf is *failed* if $\{c\} \Rightarrow \mathit{fail}$. All other leaves are *suspended*.

The following theorems relate search trees w.r.t. $\Rightarrow$ to results computed by some of the search operators (here we assume the functional definition as given in the previous section, but these properties can be also transferred to other definitions, e.g., in a relational style). To simplify the formulation of the theorems, we represent a *search goal* as a triple (V, σ, c) where $V = \{x_1, \ldots, x_n\}$ is a set of variables, σ is a substitution and c is a constraint. This corresponds to `\_->let` $x_1, \ldots, x_n$ `free in` $\{\overline{\sigma}, c\}$ in the representation introduced in Section 3, i.e., here we ignore the special rôle of the search variable since it is not important for the results in this section. In order to avoid the problem of suspension due to necessary bindings of free variables, we consider only initial search goals where all variables are existentially quantified.

The first theorem states the soundness of the `all` operator.

Theorem 1 (Soundness of "`all`"). *Let c be a constraint and $g = (\mathcal{V}ar(c), id, c)$. If* `all`$(g)$ *evaluates to a list* `[`(V_1, σ_1, c_1)`,`(V_2, σ_2, c_2)`,...]`*, then each c_i is an empty constraint, each σ_i is a $\Rightarrow$-computed answer for c and $\mathcal{V}ar(c) \cup \mathcal{VR}an(\sigma_i) \subseteq V_i$.*

The converse result does not hold in general due to infinite branches in the search tree, since `all` implements a depth-first search through the tree. However, we can state a completeness result for the case of finite search trees.

Theorem 2 (Completeness of "`all`" for finite search trees). *Let c be a constraint and σ be a $\Rightarrow$-computed answer for c. If the search tree with root c is finite, then* `all((`$\mathcal{V}ar(c), id, c$`))` *evaluates to a list* `[`(V_1, σ_1, c_1)`,...,`(V_n, σ_n, c_n)`]`*, where $\sigma_i = \sigma$ for some $i \in \{1, \ldots, n\}$.*

A corollary of this theorem is the completeness of the negation-as-failure operator.

Corollary 1 (Completeness of "`naf`" for finite search trees). *Let c be a constraint. If the search tree with root c is finite and contains only failed leaves, then* `naf(`c`)` *is a solvable constraint.*

A further interesting result is the completeness of the breadth-first search strategy `all_bfs`. As already discussed, this strategy may be incomplete in case of infinite deterministic evaluations. Therefore, we call a search tree *deterministically terminating* if there is no infinite branch where each inner node has exactly one successor. Excluding this case, which is seldom in practical search problems, we can state the following completeness result.

Theorem 3 (Completeness of "`all_bfs`"). *Let c be a constraint and σ be a $\Rightarrow$-computed answer for c. If the search tree with root c is deterministically terminating, then* `all_bfs((`$\mathcal{V}ar(c), id, c$`))` *evaluates to a (possibly infinite) list* `[`(V_1, σ_1, c_1)`,`(V_2, σ_2, c_2)`,...]`*, where $\sigma_i = \sigma$ for some $i > 0$.*

6 Exploiting Laziness

We already exploited the advantages of Curry's lazy evaluation strategy by defining the search for the first solution (`once`) based on the general depth-first search strategy `all`. This shows that lazy evaluation can reduce the programming efforts. Furthermore, it is well known from functional programming that lazy evaluation provides more modularity by separating control aspects [7]. We want to emphasize this advantage by an implementation of Prolog's top-level shell with our search operator.

The interactive command shell of a Prolog interpreter roughly behaves as follows. If the user types in a goal, a solution for this goal is computed by the standard depth-first search strategy. If a solution is found, it is presented to the user who can decide to compute the next solution (by typing '`;`' and `<return>`) or to ignore further solutions (by typing `<return>`). This behavior can be easily implemented with our search operator:

```
prolog(G) = printloop(all(G))

printloop([])     = putStr("no") >> newline
printloop([A|As]) = browse(A) >> putStr("? ")
                              >> getChar >>= evalAnswer(As)

evalAnswer(As,';')  = newline >> printloop(As)
evalAnswer(As,'\n') = newline >> putStr("yes") >> newline
```

Here we make use of the monadic I/O concept discussed at the beginning of Section 3. The result of `browse(A)` is an action which prints a solution on the screen. Similarly, `putStr` and `newline` are actions to print a string or an end-of-line. `>>` and `>>=` are the sequential composition operators for actions [18]. The second argument of `>>=` must be a function which takes the result value of the first action and maps this value to another action. The expression "`evalAnswer(As)`" is a partially applied function call, i.e., it is a function which takes a further argument (a character) and produces an action: if the character is `';'`, the next solution is computed by a call to `printloop(As)`, and if the character is a `<return>` (`'\n'`), then the computation finishes with an action to print the string `"yes"`. Note that disjunctions do not occur in the `printloop` evaluation since potential non-deterministic computation steps of the goal `G` are encapsulated with `all(G)`.

Since the solutions for the goal are evaluated by `all` in a lazy manner, only those solutions are computed which are requested by the user. This has the advantage that the user interface to present the solutions (`printloop`) can be implemented independently of the mechanism to compute solutions. In an eager language like Prolog, the computation of the next solution must be interweaved with the print loop, otherwise all solutions are computed (which may not terminate) before the print loop is called, or only one standard strategy can be used. Our implementation is independent of the particular search strategy, since the following functions use the same top-level shell but bounded and breadth-first search to find the solutions:

```
prolog_bounded(N,G) = printloop(all_bounded(N,G))
prolog_bfs(G)       = printloop(all_bfs(G))
```

7 Related Work

This section briefly compares our operator for controlling non-deterministic computations with some related methods.

Prolog provides built-in predicates for computing all solutions, like `bagof`, `setof`, or `findall`. As shown in Section 4, they can be easily implemented with our control primitive, provided that all variables are existentially quantified. On the other hand, the search strategy in these predicates is fixed to Prolog's depth-first search and they always compute all solutions, i.e., they do not terminate if there are infinitely many solutions. In particular, they cannot be used in situations where not all solutions are immediately processed, like in an interactive shell where a demand-driven computation becomes important (cf. Section 6).

The lazy functional language **Haskell** [6] supports the use of list comprehensions to deal with search problems. List comprehensions allow the implementation of many generate-and-test programs, since logic programs with a strict data flow ("well-moded programs") can be translated into functional programs by the use of list comprehensions [17]. On the other hand, list comprehensions are much more restricted than our search operators, since purely functional programs do not allow the use of partially instantiated structures, and list comprehensions fixes a particular search strategy (diagonalization of the generators) so that other strategies (like best solution search) cannot be applied.

The higher-order concurrent constraint language **Oz** [16] provides a primitive operator to control search [15] similarly to ours. Actually, our operator `try` generalizes Oz's operator since `try` is not connected to a construct of the language (like `or` expressions in Oz) but its semantics is defined on the meaning of computation steps of the base language. This has an important consequence of the programming style and causes a significant difference between both concepts which should be explained in the following. An Oz programmer must explicitly specify in the program whether a search operator should later be applicable or not. A non-deterministic step can be performed in Oz only if an explicit disjunction (`or` or `choice`, see [14,15]) occurs in a procedure. As a consequence, programs must be written in different ways depending on the use of search operators. The following simplified example explains this fundamental difference to our approach in more detail. Consider the multiplication with zero defined by the following rules:

```
mult(X,z) = z
mult(z,X) = z
```

Then expressions like `mult(z,z)` or `mult(add(z,z),z)` can be reduced to `z` with one deterministic reduction step using the first rule.[10]

In Oz, there are two implementation choices by using a conditional (`multc`) or a disjunction (`multd`):

```
proc {multc A B C}
  if B=z then C=z
  [] A=z then C=z
  fi
end
```

```
proc {multd A B C}
    or B=z then C=z
    [] A=z then C=z
    ro
  end
```

Conditionals commit to single computation branches, e.g., {`multc z z X`} reduces to the constraint `X=z`. However, we cannot use `multc` if we want to compute solutions to a goal like {`multc X Y z`} since the conditions in an `if` are only checked for entailment. Thus, we have to take the disjunctive formulation `multd` where we can compute a solution using some search operator [15]. On the other

[10] Although one could also apply the second rule in this situation, sophisticated operational models for functional logic programming exploit the determinism property of functions: if a function call is reducible (i.e., a rule is applicable without instantiating arguments), then all other alternative rules can be ignored [2,11].

hand, the advantages of deterministic reductions are lost in `multd`, since the expression {`multd z z X`} is only computable with a search operator (a disjunction is not reduced until all but at most one clause fails [15]). Therefore, one has to implement `mult` twice to combine the deterministic reduction and search possibilities.

In contrast to Oz, the definition of our control operator is based on the meaning of computation steps, i.e., the possible application of search operators does not influence the way how the basic functions or predicates are defined. This property keeps the declarative style of programming, i.e., function definitions describe the meaning of functions and control aspects are covered by search operators. Thus, functions or predicates can be defined independently of their later application, and explicit disjunctions are not necessary. The latter property also allows to write more predicates as functions which leads to potentially more efficient executions of programs. Furthermore, the laziness of Curry allows the implementation of search strategies independently of their application, e.g., demand-driven variants of search strategies (see [15]) are not necessary in our framework since the user interface can be implemented independently of the search strategy, as shown in Section 6.

8 Conclusions

We have presented a new primitive which can be added to logic languages in order to control the exploration of the search tree. This operator, which can be seen as a generalization of Oz's search operator [15], can be added to any logic-oriented language which supports equational constraints and existential quantification. In this paper, we have added it to the multi-paradigm language Curry and we have shown the advantages of Curry's lazy evaluation strategy to simplify the implementation of the different search operators. Since the search operators can be applied to any expression (encapsulated in a constraint), there is no need to translate functions into predicates or to use explicit disjunctions as in other approaches.

Since the definition of our control primitive is only based on an abstract view of computation steps (deterministic vs. non-deterministic steps), it can be applied to a variety of programming languages with a non-deterministic computation model, like pure logic or constraint logic languages (extended by existential quantifiers like in Prolog's `bagof/setof` construct), higher-order logic languages like λProlog [13] which already has explicit scoping constructs for variables, or the various functional logic languages which often differ only in the definition of a computation step. The general connection between search trees of the base language and the results computed by the search operators, which is also provided in this paper, supports the transfer of soundness and completeness results for the base language to corresponding results for the search operators.

The use of search operators supports the embedding of logic programs into other application programs where backtracking is not possible or too complicated (e.g., programs with side effects, input/output) since search operators allow the

local encapsulation of search. Furthermore, they contribute to an old idea of logic programming by separating logic and control: the specification of functions or predicates becomes more independent of their use since the same function can be used for evaluation (computing values) or for searching (computing solutions) with various strategies without the necessity to define them in different ways. As shown in Section 6, this feature enables to simply replace the standard depth-first search by a bounded or breadth-first search in the user interface. This is quite useful to teach logic programming without talking about backtracking too early.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994. 378
2. M. Hanus. Lazy Narrowing with Simplification. *Computer Languages*, Vol. 23, No. 2–4, pp. 61–85, 1997. 388
3. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997. 376, 379
4. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. Technical Report, RWTH Aachen, 1998 375, 385
5. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/ hanus/curry`, 1998. 375, 376, 379, 380
6. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell (Version 1.2). *SIGPLAN Notices*, Vol. 27, No. 5, 1992. 388
7. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topcis in Functional Programming*, pp. 17–42. Addison Wesley, 1990. 386
8. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 International Logic Programming Symposium*, pp. 167–183. MIT Press, 1991. 383
9. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987. 385
10. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993. 378
11. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995. 388
12. G. Nadathur, B. Jayaraman, and K. Kwon. Scoping Constructs in Logic Programming: Implementation Problems and their Solution. *Journal of Logic Programming*, Vol. 25, No. 2, pp. 119–161, 1995. 383
13. G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988. 389
14. C. Schulte. Programming Constraint Inference Engines. In *Proc. of the Third International Conference on Principles and Practice of Constraint Programming*, pp. 519–533. Springer LNCS 1330, 1997. 388

15. C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pp. 505–520. MIT Press, 1994. 375, 376, 380, 383, 385, 388, 389
16. G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995. 388
17. P. Wadler. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 1985. 388
18. P. Wadler. How to Declare an Imperative. In *Proc. of the 1995 International Logic Programming Symposium*, pp. 18–32. MIT Press, 1995. 379, 387

Encapsulating Data in Logic Programming via Categorical Constraints

James Lipton and Robert McGrail

Dept. of Mathematics and Computer Science
Wesleyan University
http://www.cs.wesleyan.edu/

Abstract. We define a framework for writing executable declarative specifications which incorporate categorical constraints on data, Horn Clauses and datatype specification over finite-product categories. We construct a generic extension of a base syntactic category of constraints in which arrows correspond to resolution proofs subject to the specified data constraints.

1 Introduction

Much of the research in logic programming is aimed at expanding the expressive power and efficiency of declarative languages without compromising the logical transparency commitment: programs should (almost) read like specifications. One approach is to place more expressive power and more of the control components into the logic itself, possibly by expanding the scope of the underlying mathematical formalism. This has been the goal of constraint logic programming (CLP, Set constraints, Prolog III), and extensions to higher-order and linear logic, to name a few such efforts. This paper is a step in this direction. Rather than expand the logic itself, we consider two extensions of the underlying *syntactical* foundation, using fairly simple categorical tools. Categorical syntax and proof theory for logic programming permits a powerful extension of conventional logic programming to be built in a manner that does not compromise declarity and that admits a similar semantical treatment to conventional logic programming.

The encoding dilemma. The main contribution of the paper is a new approach to defining syntactic extensions to specify data types and constraints as independent components, somewhat in the spirit of modules in functional programming.

This information is processed by extending the logic programming interpreter dynamically with new rules. In traditional logic programming, this kind of information is coded directly into the Horn logic, often very cleverly, in a way that may obscure the intended meaning of the code, despite the fact that it is logical. Soundness and completeness of the target logic says nothing about the correctness of the encoding. We propose expanding the specification formalism in a

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 391–411, 1998.

way that admits direct definitions of data and constraint information that are already mathematically clear, or verified elsewhere using appropriate tools. We consider a datatype defintion construct and a very liberal notion of constraint based on finite product categories, of which equational constraint systems are a special case. We make use of a categorical foundation developed in [9], and similar in spirit to, e.g. [1,4,5,21]. Although the framework admits extensions to Hereditarily Harrop or Higher-order logic, we will only consider Horn logic here.

An Example: It will be useful to give an example of the kind of code that is admitted by the extensions discussed in this paper.

```
begin module "list"
 datatype 'a list =  nil | cons of 'a * 'a list
  begin
    fun length nil = (0:int)
     |  length (cons (a,t)) = 1 + length t
  end;
 even(0:int).
 even(X) :- even(X - 2).

 length_of(Z, length Z).
end module;
? - list even (cons(2,(cons(4,(cons(6,nil)))))).
-  yes
? - pred[length](cons (2,(cons (4,(cons (5,nil))))),A).
-  A = 3
? - length_of((cons (2,(cons (4,(cons (5,nil)))))),A).
-  A = 3
```

It is assumed that a datatype `int`, together with operators `+,*,-` of type `int -> int`, constants `0,1,2,...:int` and equational constraints are defined elsewhere.

The predicate `pred[length]` is not specified by the user, but is built up automatically from the function definition for `length`, whereas (for no good reason other that to illustrate the difference) `length_of` is a predicate with the same meaning *hand coded* by the user[1] The query

```
 ?-length_of((cons (2,(cons (4,(cons (5,nil)))))),A+2).
```

yields A=1 by unification of `((cons (2,(cons (4,(cons (5,nil)))))),A+2)` with `(Z,length Z)` which is carried out in the appropriate category by taking the pullback of the two corresponding arrows, as described below. The predicate `list even` is also not defined directly by the user, only `even`.

In this paper we show how to define a category with arrows corresponding to resolution proofs with a generalized notion of unification with constraints such as:

[1] The point of providing an automatically generated predicate `pred[length]` is that the function `length` could be made *private* to the data type. This is in keeping with the view that adding datatypes should not necessarily mean turning logic programming into a functional programming language.

$$\texttt{length\_of((cons (2,(cons (4,(cons (5,nil)))))),A+2)} \overset{A=1}{\rightsquigarrow} \square.$$

and with new proof steps that incorporate data type information

$$\texttt{list } X \texttt{ (cons(a,t))} \rightsquigarrow X\texttt{(a),list } X \texttt{ (cons(a,t))}$$

which are treated as dynamic updates of the basic Horn clause interpreter.

What we will not discuss here are proof strategies: algorithms for searching the category of proofs or the category of non-deterministic resolutions defined in the paper, or for computing pullbacks. This is of course extremely relevant, and quite a harder matter to resolve than the subject matter of this paper. What we show here is that in principle a correct interpreter can be dynamically generated from constraint data expressed in a modular discipline that is both a specification at top level, and correctly executable code.

Related Work: Module proposals, and data type definitions for logic programming have appeared since the early 1980's, notably [12,2]. Our contribution here is to introduce a new technique for lifting certain constraints on data to predicates on that data and proofs between them, in the general framework of a categorically defined constraint domain.

Categorical approaches to logic programming features appeared in the mid 1980's in Rydeheard and Burstall's categorical treatment of unification [24] based on ideas of Burstall and Goguen. Since then research using categories to analyze or generalize different facets of declarative programming has been carried out by Corradini, Asperti and Montanari [1,5,4], Panangaden, Scott, Seely, Saraswat, [19], Power and Kinoshita [21] Diaconescu [8], Pym [22], Orejas, Ehrig and Pino [18], and Finkelstein, Freyd and Lipton [9,10].

2 Logic Programming Categories

In the remainder of this paper, we build on the basic categorical framework defined in [9], based on standard approaches to categorical logic, e.g. [16,11]. It is briefly sketched here.

Categorical Logic: In a nutshell, the categorical representation of the syntax of logic programs is as follows: an FP category (category with finite products) $\mathbb{C}$ is taken as a generalization of to the Herbrand Universe. It serves as a generalized program signature as well: it supplies the basic sorts (objects), function symbols of sort (σ, ρ) (arrows $\sigma \xrightarrow{f} \rho$), and constants of sort ρ (arrows of the form $\mathbf{1} \xrightarrow{a} \rho$, where $\mathbf{1}$ is the terminal object). Since it is closed under allowed compositions, it supplies the terms as well. Often, only a distinguished class of arrows (closed under composition) interest us as program terms, and are so identified. Predicates are subobjects of their sort, with instances given by pulling back along term arrows. In order to make the right kind of subobjects available, special categories $\mathbb{C}[X_1, \ldots, X_n]$ or $\mathbb{C}[\sigma]$ of generic predicates will be built on top of $\mathbb{C}$ subject to different constraints depending on the extension we are

trying to formalize. In these categories, predicates are functors generated by spans emanating from their sort.

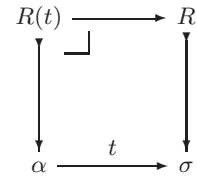
We can recover traditional logic programming over the Herbrand universe in this framework, if we pick $\mathbb{C}$ to be the so-called the Lawvere Algebraic Theory for a one-sorted signature with at least one constant (considered in e.g. [9]) and with no equations. Here, arrows correspond to (projections of vectors of) terms of the Herbrand Universe. We call this the *Herbrand Category* for the signature. The more general framework of arbitrary finite-product categories and generic predicate extensions allows us to *build-in* constraint information and data types into the syntax, as well as into the the structure of predicates. In such categories, unification of two arrows u and v with a common target (corresponding to two terms of the same sort which are *standardized apart*) amounts to finding a pair of substitutions (arrows) with a common source θ, ψ making a commuting square, i.e. such that $\psi v = \theta u$.

Most general unifiers yield pullback diagrams, when they exist. Note that separating the domains of terms u and v is what standardizes them apart[2]. In extensions of Horn logic, such as λ-prolog, queries and programs may share variables, which requires explicit sharing of domains. Unification then reduces to equalizing the arrows (see [10])

For basic concepts of category theory, we refer the reader to [11], and for the elements of indexed category theory, see [6]. We note that compositions of arrows are written in diagrammatic order: $A \xrightarrow{f} B$ composed with $B \xrightarrow{g} C$ is $A \xrightarrow{fg} C$.

2.1 Generic Predicates and Unordered Goals

A predicate symbol R of type σ may be modelled in a category with finite products by a monic arrow with target σ. Instantiation (and hence, substitution) by terms of sort σ can be modelled by pullbacks along the appropriate arrow:



Any distinguished family of monics can play the role of predicates in a logic program, but if they are to make sense in logic program syntax such a collection should be closed under pullbacks[3] and they should not be chosen to conflict with the meaning of the logic program in which they will be used. Unlerss one wishes to constrain program predicates in advance (a question we will take up

[2] If the ambient category is the Herbrand Category, the occurs-check will be automatically enforced.

[3] If the predicates are not stable under pullbacks of arrows designated as program terms, then certain instances may not even exist in the syntax, which is a rather unusual form of failure for a logic program query.

subsequently), the predicate A should not be true of any instance $A(t)$ in the syntactic category, until information about the program is somehow incorporated into the categorical structure. In short: they should behave like second-order predicate variables. We thus seek a notion of *generic* or *freely adjoined predicate*.

Definition 1 (Generic Predicates) *Let X be a subobject of some object b in a finite product category $\mathbb{C}$, and let $\mathcal{D}$ be a family of arrows in $\mathbb{C}$ targeted at b. We say X is a* **generic subobject** *of b with respect to the maps $\mathcal{D}$ if*

- *For every arrow t in $\mathcal{D}$ the pullback $t^{\#}(X)$ exists.*
- *No such pullback is an isomorphism.*

Definition 2 *Let $\mathbb{C}$ be an FP category and $\boldsymbol{b} = b_1, \ldots, b_n$ a sequence of objects of $\mathbb{C}$. Then $\mathbb{C}[X_1, \ldots, X_n]$, the* category obtained from $\mathbb{C}$ by freely adjoining indeterminate subobjects of $\boldsymbol{b}$ *is defined as follows:*

objects: *pairs $\langle A, S\rangle$ where $A \in |\mathbb{C}|$ and S is a sequence $S_1, \ldots, S_n$ of finite sets $S_i \subset Hom_{\mathbb{C}}(A, b_i)$,*

arrows: *are triples $\langle A, S\rangle \xrightarrow{f} \langle B, T\rangle$ where $A \xrightarrow{f} B$ is an arrow in $\mathbb{C}$ and $fT \subset S$, that is to say, for every i, $(1 \leq i \leq n)$ and every $t \in T_i$, $ft_i \in S_i$. The arrow f in $\mathbb{C}$ is called the* **label** *of $\langle A, S\rangle \xrightarrow{f} \langle B, T\rangle$. Composition of arrows is inherited from $\mathbb{C}$. Two arrows $\langle A, S\rangle \xrightarrow{f} \langle B, T\rangle$ and $\langle A', S'\rangle \xrightarrow{f'} \langle B', T'\rangle$ are* **equal** *if they have the same domain and range and if $f = f'$ in $\mathbb{C}$.*

Given an object $\langle A, S\rangle$ we will use the notation tS, where t is an arrow in $\mathbb{C}$ with target A, to mean the sequence $tS_1, \ldots, tS_n$ where $tS_i = \{ts : s \in S_i\}$. Notice that an arrow in $\mathbb{C}[X_1, \ldots, X_n]$, may have an identity arrow in $\mathbb{C}$ as a label, and not even be an isomorphism in $\mathbb{C}[X_1, \ldots, X_n]$. We will be paying special attention to a certain class of such arrows.

Theorem 3 *Let $\mathbb{C}$ be a finite product category. The category $\mathbb{C}[X_1, \ldots, X_n]$ has*

- *a terminal object $\langle \mathbf{1}, \boldsymbol{\emptyset}\rangle$, where $\boldsymbol{\emptyset}$ is the sequence $\emptyset, \ldots, \emptyset$ of length n,*
- *products: $\langle A, S\rangle \times \langle B, T\rangle = \langle A \times B, \pi_1 S \cup \pi_2 T\rangle$ where $A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$ is a product in $\mathbb{C}$.*

Furthermore, the functor $\mathbb{C} \xrightarrow{\iota} \mathbb{C}[X_1, \ldots, X_n]$ given by mapping objects A to $\langle A, \boldsymbol{\emptyset}\rangle$ and arrows $A \xrightarrow{f} B$ to $\langle A, \boldsymbol{\emptyset}\rangle \xrightarrow{f} \langle B, \boldsymbol{\emptyset}\rangle$, is a limit-preserving, full embedding.

Limit preservation follows from the fact that ι has a left adjoint, namely the forgetful functor U taking objects $\langle A, S\rangle$ to A and arrows to their labels.

Lemma 4 *Addition of indeterminate subobjects simultaneously, sequentially, or in permuted order results in isomorphic categories. More precisely:*

1. $\mathbb{C}[X_1, \ldots, X_n] \simeq \mathbb{C}[X_1] \cdots [X_n]$.
2. *Let σ be a permutation of the first n positive integers. Then $\mathbb{C}[X_1, \ldots, X_n] \simeq \mathbb{C}[X_{\sigma(1)}, \ldots, X_{\sigma(n)}]$.*

Proof. Straightforward. □

Definition 5 *In* $\mathbb{C}[X_1, \ldots, X_n]$ *define the* indeterminate subobjects $X_1, \ldots, X_n$ *of sorts* $b_1, \ldots, b_n$ *respectively, to be the subobjects* $\langle b_i, J^i \rangle \xrightarrow{id_{b_i}} \langle b, \emptyset \rangle$, *where the* J^i *are the* "basis vectors"

$$(J^i)_k = \begin{cases} \emptyset & \text{if} \quad i \neq k \\ \{id_{b_i}\} & \text{o.w.} \end{cases}$$

Theorem 6 *The indeterminate subobjects* X_i *of* b_i *are generic with respect to the maps (targeted at* $\langle b_i, \emptyset \rangle$*) in the image of* $\mathbb{C} \xrightarrow{\iota} \mathbb{C}[X_1, \ldots, X_n]$.

Proof. The following diagram is a pullback for any arrow $\langle A, \emptyset \rangle \xrightarrow{t} \langle b_i, \emptyset \rangle$:

$$\begin{array}{ccc} \langle A, tJ^i \rangle & \xrightarrow{t} & \langle b_i, \{id\} \rangle \\ {\scriptstyle id_A}\downarrow & & \downarrow{\scriptstyle id_{b_i}} \\ \langle A, \emptyset \rangle & \xrightarrow[t]{} & \langle b_i, \emptyset \rangle \end{array}$$

so $X(t) = \langle A, tJ^i \rangle \xrightarrow{id_A} \langle A, \emptyset \rangle$ exists for all appropriate t. This arrow cannot be an isomorphism in $\mathbb{C}[X_1, \ldots, X_n]$: its inverse, which would have to be labelled with id_A, would have to satisfy $id_A t \in \emptyset$. □

Objects of the form $X(t) = \langle A, tJ^i \rangle$ will be called **atomic** predicates. If A is an object of $\mathbb{C}$, we say that the monic $\langle B, S \rangle \xrightarrow{f} \langle A, \emptyset \rangle$ is a canonical (representative of a) subobject of $\langle A, \emptyset \rangle$ if B is A and the monic f is id_A. Observe that every object $\langle A, S \rangle$ of $\mathbb{C}[X_1, \ldots, X_n]$ is a canonical subobject of "its sort" $\langle A, \emptyset \rangle$. This allows us to define a natural indexed structure [6] for $\mathbb{C}[X_1, \ldots, X_n]$ over $\mathbb{C}$.

Definition 7 *For each object* A *of* $\mathbb{C}$, *let* $\mathbb{C}_A[X_1, \ldots, X_n]$ *be the category whose objects are arrows in* $\mathbb{C}[X_1, \ldots, X_n]$ *of the form* $\langle A, S \rangle \xrightarrow{id_A} \langle A, \emptyset \rangle$, *and with morphisms given by arrows between their sources labelled by the identity on* A *in* $\mathbb{C}$. *Then let* $p : \mathbb{C} \longrightarrow \mathbb{CAT}$ *be the strict indexed category given by*

- $p(A) = \mathbb{C}_A[X_1, \ldots, X_n]$
- $p(A \xrightarrow{f} B) = f^{\#} : \mathbb{C}_B[X_1, \ldots, X_n] \longrightarrow \mathbb{C}_A[X_1, \ldots, X_n]$

Notice that the pullback operation referred to by $f^{\#}$ maps $\langle B, S \rangle \xrightarrow{id_B} \langle B, \emptyset \rangle$ to $\langle A, fS \rangle \xrightarrow{id_A} \langle A, \emptyset \rangle$. Thus $(fg)^{\#}$ is precisely $g^{\#} f^{\#}$ on the nose. Now we define a "canonical intersection" functor for the indexed category p:

Definition 8 *Let* $\cap : p \times p \longrightarrow p$ *be defined by* $\langle A, S \rangle \cap_A \langle A, T \rangle = \langle A, S \cup T \rangle$.

By the remarks immediately preceding the definition, it is immediate that $\cap$ commutes with pullback, i.e. is a natural transformation.

The following theorems make precise the fact that $\mathbb{C}[X_1, \ldots, X_n]$ is called the category obtained by freely adjoining the indeterminate subobjects of the sorts $b_1, \ldots, b_n$.

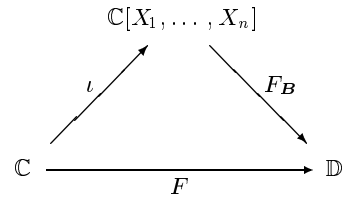
Lemma 9 *Every object $\langle A, S\rangle$ is representable as (i.e. equal on the nose to) the canonical intersection*

$$\bigcap\{t^{\#}(X_i) : t \in S_i, 1 \leq i \leq n\}$$

where the pullbacks are canonical: $t^{\#}(X_i) = \langle A, tJ^i\rangle = \langle A, \emptyset \cdots \emptyset \underbrace{\{t\}}_{i} \emptyset \cdots \emptyset\rangle$.

Proof. Immediate: Since $S = \bigcup\{\{t\} : t \in S\}$, the indicated canonical intersection is precisely $\langle A, S\rangle$. □

Theorem 10 (Universal Mapping Property) *Suppose $F : \mathbb{C} \longrightarrow \mathbb{D}$ is a limit preserving functor from the finite-product category $\mathbb{C}$ to the finitely complete category $\mathbb{D}$, and that $F(b_i) = d_i$ for $1 \leq i \leq n$. Furthermore, let $B_1, \ldots, B_n$ be a sequence of subobjects of $d_1, \ldots, d_n$ respectively, in $\mathbb{D}$. Then there is a limit-preserving functor $F_{\boldsymbol{B}} : \mathbb{C}[X_1, \ldots, X_n] \longrightarrow \mathbb{D}$, unique up to isomorphism, such that the following diagram commutes.*



$F_{\boldsymbol{B}}$ is called the **evaluation** *functor induced by the B_i.*

Proof. Define $F_{\boldsymbol{B}}$ on objects by $F_{\boldsymbol{B}}(\langle A, S\rangle) = \varprojlim\{F(t)^{\#}(B_i) : t \in S_i, 1 \leq i \leq n\}$. The universal mapping property of limits gives us the action on arrows: if $\langle A, S\rangle \xrightarrow{f} \langle A', S'\rangle$ is an arrow in $\mathbb{C}[X_1, \ldots, X_n]$ then $F_{\boldsymbol{B}}(\langle A, S\rangle)$, the limit of the family of monics $\{F(t)^{\#}(B_i) : t \in S_i, 1 \leq i \leq n\}$ targeted at FA, is also, by composing with $F(A \xrightarrow{f} B)$ and using properties of pullbacks and of arrows in $\mathbb{C}[X_1, \ldots, X_n]$, a cone over the family of monics $\{F(t)^{\#}(B_i) : t \in S'_i, 1 \leq i \leq n\}$. There is therefore a unique induced arrow $F\langle A, S\rangle \xrightarrow{\theta} F\langle A', S'\rangle$ which is the value of $F(\langle A, S\rangle \xrightarrow{f} \langle A', S'\rangle)$. The details, and those of the proof of limit preservation, are left to the perseverant reader. □

We are interested in a category $\mathbb{D}$ with richer structure, in which case we are able to sharpen this result a bit.

Corollary 11 *Assume the category $\mathbb{D}$ in the preceding theorem is $\mathsf{Set}^{\mathbb{C}^o}$ and that F is the Yoneda embedding. Choose the sequence of subobjects B_i of $Fb_i = Hom_{\mathbb{C}}(\_, b_i)$ to be* canonical*, that is to say, pointwise subsets of Fb_i, and take limits in $\mathsf{Set}^{\mathbb{C}^o}$ to be given pointwise (not just up to isomorphism, but on the nose). Then the evaluation functor $F_{\boldsymbol{B}}$ of the preceding theorem is unique.*

Unordered Goals The most elementary notion of query, or of state in a logic program, is that of a conjunction of atoms

$$X_{i_1}(t_1), \ldots, X_{i_n}(t_n) \tag{1}$$

where the X_{i_j} are program predicate symbols. We call these *basic* goals.

A first approximation to syntactic goals is already present in the category $\mathbb{C}[X_1, \ldots, X_n]$ whose objects are effectively *unordered goals*. By the representation lemma (9) above, every object $\langle A, S\rangle$ in $\mathbb{C}[X_1, \ldots, X_n]$ is an intersection

$$\bigcap\{t^{\#}(X_i) : t \in S_i, 1 \leq i \leq n\}. \tag{2}$$

and can be thought of as a non-deterministic image of the corresponding ordered goal (1) above.

These intersections are free in the sense that one can recover all components $t^{\#}(X_i)$ from them, i.e. by reading off the arrows in the S_i, and in the sense of theorem 10. Since the S_i are sets, they cannot capture order or repititions of atoms within goals. An ordered counterpart will be defined below.

Definition 12 *Let $\mathbb{C}$ be a finite product category, and $X_1, \ldots, X_n$ a sequence of generic predicates over $\mathbb{C}$ of sorts $b_1, \ldots, b_n$. An* **interpretation** *is an evaluation functor extending the Yoneda embedding, assigning to each X_i some canonical subobject B_i of $Hom_{\mathbb{C}}(\_, b_i)$ as in corollary 11.*

In other words, an interpretation is a functor

$$[\![\,]\!] : \mathbb{C}[X_1, \ldots, X_n] \longrightarrow \mathsf{Set}^{\mathbb{C}^o}$$

- agreeing with the Yoneda embedding on $\mathbb{C}$, and
- mapping $\langle A, S\rangle$ to $\bigcap\{([\![t]\!])^{\#}(B_i) : t \in S_i, 1 \leq i \leq n\}$,

where $[\![t]\!]$ means the Yoneda image of t.

It is easy to check that interpretations form a complete lattice under the pointwise order.

2.2 Two Proof-Theoretic Categories

Fix a sequence of sorts $\sigma = \sigma_1, \ldots, \sigma_n$, $\sigma_i \in |\mathbb{C}|$ and a family $X_1, \ldots, X_n$ of generic predicates $X_1, \ldots, X_n$ with X_i of sort σ_i, as in the construction of the categories in the preceding section. We will refer to objects $X_i(t_i) = t_i J^i$ of sort A, where $A \xrightarrow[t]{} \sigma_i$ is an arrow in $\mathbb{C}$, as atomic goals or predicates (of sort A), objects $\langle A, S\rangle$ of $\mathbb{C}[X_1, \ldots, X_n]$ as unordered (or non-deterministic) goals (of sort A), and sequences $X_{i_1}(t_1), \ldots, X_{i_m}(t_m)$, where each atomic predicate $X_{i_j}(t_j)$ is of sort A, as ordered goals of sort A.

Note that there is a "forgetful" function β which takes ordered goals to the underlying unordered ones:

$$\beta(X_{i_1}(t_1), \ldots, X_{i_m}(t_m)) = \langle A, S\rangle, \tag{3}$$

where $S_k = \{t_j : i_j = k\}$ that is to say, the set of all terms occurring as arguments to the kth generic predicate, wherever it occurs (0 or more times) in $X_{i_1}(t_1), \ldots, X_{i_m}(t_m)$. Thus $\langle A, S\rangle = \bigcap\{t^{\#}(X_i) : t \in S_i, 1 \leq i \leq n\}$ as discussed in the previous section.

It will be convenient below to describe the following operation on ordered and unordered goals, both given the same name del_t^k, which removes one occurrence (the first one, in the ordered case) of the atomic goal $X_k(t)$ if it exists, and returns the original goal unchanged, otherwise. More formally, in the unordered case: $\mathsf{del}_t^k\langle A, S\rangle = \langle A, T\rangle$ where $T_i = S_i$ if $i \neq k$ and $T_k = S_k \setminus \{t\}$.

Definition 13 *A Horn Program over $\mathbb{C}$ (in the predicates $X_1, \ldots, X_n$) is a finite set of triples $\langle A, G, X(t)\rangle$ where G is an ordered goal, and $X(t)$ an atomic goal, both of sort A. An unordered Horn program is a similar set of triples $\langle A, G, X(t)\rangle$, but where G is unordered. The triples are called (ordered or unordered) clauses, and may be written $G \Rightarrow X(t)$ when the sort is understood from context.*

Note that the "forgetful" function β in (3) extends naturally to a map from ordered clauses to unordered ones.

Definition 14 *Let P be a Horn program. A P-SLD proof step between ordered goals G_1 and G_2, of sorts A_1 and A_2, respectively, with substitution θ, and program clause $c = \langle A, H, X_k(u)\rangle$, denoted*

$$G_1 \overset{\theta,c}{\rightsquigarrow} G_2$$

is a 4-ary relation (relating G_1, θ, c, G_2) defined by:

- *$A_2 \xrightarrow{\theta} A_1$ is an arrow in $\mathbb{C}$, and*
- *there is an arrow $A_1 \xrightarrow[t]{} \sigma_k$ in $\mathbb{C}$, such that $X_k(t)$ is an atomic subgoal of G_1, that is to say, $G_1 = G, X_k(t), G'$,*
- *there is an arrow $A_2 rT o^{\psi} A$ such that $\theta t = \psi u$ and*
- *$G_2 = \theta G, \psi H, \theta G'$.*

If $\langle A_1, S\rangle, \langle A_2, T\rangle$ are unordered goals, an sld-step $\langle A_1, S\rangle \overset{\theta,c}{\rightsquigarrow} \langle A_2, T\rangle$ between them exists when a clause c and unifying arrows θ and ψ exist, as above, and $T = \theta(\mathsf{del}_t^k S) \cap \psi H$. In both the ordered and unordered cases, the arrow θ is called the substitution of the SLD step, and c its clause.

We define an SLD-*sequence* between goals, $G \overset{\theta}{\rightsquigarrow} \cdots \rightsquigarrow G'$ to be the ternary relation that obtains when there is a sequence of goals $G = G_0, \ldots, G_n = G'$ and SLD-steps between each G_{i-1} and G_i with substitution θ_i, and θ is the composition $\theta_n \cdots \theta_1$. The "reflexive case" $n = 0$ and $\theta = id$ is allowed.

Definition 15 *The category $\mathbb{C}_P^{SLD}$ of unordered SLD proofs over program P has the objects of $\mathbb{C}[X_1, \ldots, X_n]$ as its objects, and arrows $\langle A, S\rangle \xrightarrow{\theta} \langle B, T\rangle$ where $\langle B, T\rangle \overset{\theta}{\rightsquigarrow} \cdots \rightsquigarrow \langle A, S\rangle$ is an SLD sequence. We call $A \xrightarrow{\theta} B$ in $\mathbb{C}$ the* label *of this arrow of $\mathbb{C}_P^{SLD}$.*

The Operational Category $\mathbb{C}_\mathsf{P}$ We now modify the generic predicate construction $\mathbb{C}[X_1, \ldots, X_n]$ to produce a category of predicates $\mathbb{C}_\mathsf{P}$ which are generic *modulo the clauses in program* P. $\mathbb{C}_\mathsf{P}$ is a categorical (and constraint-sensitive) counterpart to the operational C-semantics of Levi et. al. [15], in the sense that objects are goals modulo operational equivalence, and arrows are open substitutions between them. Later we will study how datatype information acts on both $\mathbb{C}_\mathsf{P}$ and $\mathbb{C}_P^{SLD}$ to produce dynamically updated predicates, operational semantics and proofs.

Definition 16 *Let A be an object in $\mathbb{C}$, and T a monotone operator on sets of arrows with a common source in $\mathbb{C}$. Then a family S of arrows with source A is said to be closed under T if $T(S) = S$. S is T-***generated** *by a family of arrows emanating from A if $S = T(U)$. We will also write $\langle U\rangle$ for $T(U)$ when the closure operator is understood from context.*

Definition 17 *Let $\mathbb{C}$ be a finite-product category, and P a Horn clause program over $\mathbb{C}[X_1, \ldots, X_n]$. The category $\mathbb{C}_\mathsf{P}$ is given by the following data:*

objects: *Pairs (A, S) where A is an object of $\mathbb{C}$, and S is a sequence $S_1, \ldots, S_n$, where each S_i is a set of arrows from A to σ_i, closed* with respect to the following **clausal** closure condition:
For each clause of sort α in P

$$X_{i_1}(t_1), \ldots, X_{i_n}(t_n) \Rightarrow X(t) \tag{4}$$

where X, X_{i_k} are generic predicates of sort σ_j, σ_{i_k}, $\alpha \xrightarrow{t} \sigma$ and $\alpha \xrightarrow{t_k} \sigma_{i_k}$ arrows in $\mathbb{C}$, and for each arrow $A \xrightarrow{\varphi} \alpha$ satisfying

$$\varphi t_1 \in S_{i_1}, \ldots, \varphi t_n \in S_{i_n} \tag{5}$$

we must have $\varphi t \in S_j$.

arrows *are triples* $(A,S) \xrightarrow{f} (B,V)$ *such that* $A \xrightarrow{f} B$ *(the* label*) is an arrow in* $\mathbb{C}$*, and* $fV \subset S$*. Composition is defined by composing labels.*

If we let T_{P}^{A} be the operator on sets of arrows with a common source A given by

$$T_{\mathsf{P}}^{A}(S) = \bigcup_{c \in \mathsf{P}} \{\varphi t : A \xrightarrow{\varphi} \alpha \text{ and } \varphi t_1 \ldots, \varphi t_n \in S\}$$

then the requirement on S in the preceding definition is equivalent to being T_{P}^{A}-generated.

Lemma 18 *There is a faithful functor* $\iota_{\mathsf{P}} : \mathbb{C}[X_1, \ldots, X_n] \longrightarrow \mathbb{C}_{\mathsf{P}}$ *given by* $(A,U) \mapsto (A, \langle U \rangle)$ *on objects, and which preserves labels of arrows.*

Proof. Since the closure operator T_{P}^{A} is monotone, ι_{P} maps arrows to arrows. Fidelity and functoriality is immediate. □

2.3 Semantics

Let P be a program, $\mathcal{G}^{\mathsf{P}}$the collection of ordered goals over P, and $[\![\ ]\!]$ an interpretation into $\mathsf{Set}^{\mathbb{C}^o}$ as in definition (12), and β the forgetful map from ordered to unordered goals. Then observe that $\beta[\![\ ]\!]$ is a function $\mathcal{G}^{\mathsf{P}} \longrightarrow \mathsf{Set}^{\mathbb{C}^o}$. We sometimes overload the symbol $[\![\ ]\!]$ to denote $\beta[\![\ ]\!]$, since context will always make clear what is meant. We call the composition $\beta\iota_{\mathsf{P}} : \mathcal{G}^{\mathsf{P}} \longrightarrow \mathbb{C}_{\mathsf{P}}$ the representation of goals in the category $\mathbb{C}_{\mathsf{P}}$.

Note that by corollary (11), *the value of* $[\![\ ]\!]$ *on goals is completely determined by its value on generic predicates.*

We now give a quick sketch of some results on semantics from [9,10]. We give a categorical analogue of the Kowalski-van Emden bottom-up semantics. Proofs can be founded in the cited references, and are, for the most part, omitted.

Definition 19 *An interpretation* $[\![\ ]\!]$ *is a model of program* P *if for every clause* $tl_{cl} \Rightarrow hd_{cl}$ *we have* $[\![tl]\!] \subset [\![hd]\!]$*. A goal* G *of sort* α *is said to be* **true** *in the interpretation if the image* $[\![\beta(G)]\!] \in \mathsf{Set}^{\mathbb{C}^o}$ *of the monic*

$$\beta(G) \xrightarrow{id_\alpha} (\alpha, \emptyset)$$

is an isomorphism.

In the following discussion we will use the notation $cl \in \mathsf{P}$ to refer to the fact that cl is an ordered clause in the program P. We also refer to the empty goal of sort σ by $\Box_\sigma$. The image of this goal in $\mathbb{C}[X_1, \ldots, X_n]$ under β is $(\sigma, \emptyset)$, which is the entire subobject $(\sigma, \emptyset) \Longrightarrow (\sigma, \emptyset)$. Since functors preserve identity arrows, it must be mapped by any interpretation to the identity $\mathbb{C}(\_, \sigma) \Longrightarrow \mathbb{C}(\_, \sigma)$.

Lemma 20 (Soundness) *Let* $[\![\ ]\!]$ *be a model of program* P*, and* G_1 *and* G_2 *goals of sorts* α_1, α_2 *respectively, and* $G_1 \rightsquigarrow \overset{\theta,c}{\cdots} \rightsquigarrow G_2$ *a resolution proof with answer substitution* $\alpha_2 \xrightarrow{\theta} \alpha_1$*. Then* $[\![G_2]\!] \subset [\![\theta G_1]\!]$*. In particular, if* G_2 *is* $\square_{\alpha_2}$ *then* θG_1 *is true in the model.*

We now define a categorical analogue of the T_P operator of Kowalski and Van Emden, an operator E_P on the lattice of interpretations.

Definition 21 *Let* $[\![\ ]\!]$ *be an interpretation and* $X_1, \ldots, X_n$ *the sequence of generic predicates in program* P*. Then*

$$E_P([\![\ ]\!])(X_i) = \bigcup_{tl \Rightarrow X_i(t) \in P} \mathsf{Im}_{[\![t]\!]}([\![\mathsf{tl}_{\mathsf{cl}}]\!])$$

where, for each t *occurring in the head of a clause in* P *of the form* $tl \Rightarrow X_i(t)$*,* $\mathsf{Im}_{[\![t]\!]}$ *denotes the image along the arrow* $[\![t]\!]$ *in* $\mathsf{Set}^{\mathbb{C}^o}$*.*

In [9,10] it is shown that E_P is a continuous operator on the lattice of interpretations, with a least fixed point $[\![\ ]\!]^*$ (called the Herbrand interpretation for P).

Lemma 22 *An interpretation* $[\![\ ]\!]$ *is a model of program* P *if and only if it is a pre-fixed point of* E_P *(that is to say,* $E_P([\![\ ]\!]) \subseteq [\![\ ]\!]$*) and hence, if and only if* $[\![\ ]\!]^* \subseteq [\![\ ]\!]$*.*

The following is established in [9,10]

Theorem 23 (Completeness) *If* P *is a program,* $[\![\ ]\!]^*$ *its Herbrand interpretation and* G *a goal,* $[\![G]\!]^*$ *is an isomorphism if and only if there is an SLD proof* $G \rightsquigarrow \overset{id}{\cdots} \rightsquigarrow \square$

We now establish some properties of the category $\mathbb{C}_{\mathsf{P}}$. Since it is a finite-product category, arrows in $\mathbb{C}_{\mathsf{P}}$ are not, strictly speaking, SLD proof steps (in reverse), but a bit more. Arrows in $\mathbb{C}_{\mathsf{P}}$ correspond to *extended* resolution steps that include weakening of goals and instantiation of goals, as well as products of basic SLD sequences that can be thought of as parallel resolutions of multiple goals. But the present category suffices for our purposes, since the only *global* proofs $\square_\alpha \xrightarrow{\theta} \beta\iota_{\mathsf{P}}(G)$ correspond to real SLD proofs $G \rightsquigarrow \overset{\theta}{\cdots} \rightsquigarrow \square_\alpha$.

Theorem 24 (Soundness of $\beta \circ \iota_{\mathsf{P}}$) *Let* $G_1 \rightsquigarrow \overset{\theta}{\cdots} \rightsquigarrow G_2$ *be an SLD sequence, and let* $(\alpha_1, \langle U_1 \rangle)$ *and* $(\alpha_2, \langle U_2 \rangle)$ *be the images in* $\mathbb{C}_{\mathsf{P}}$ *of* G_1 *and* G_2 *under* $\beta \circ \iota_{\mathsf{P}}$*. Then* $G_2 \xrightarrow{\theta} G_1$ *is an arrow in* $\mathbb{C}_{\mathsf{P}}$*.*

Proof. Observe that the closure condition (5) of definition (17) guarantees that for each clause $tl \Rightarrow hd$ of P, the arrow $(\alpha, \langle S_{tl} \rangle) \xrightarrow{id_\alpha} (\alpha, \langle S_{hd} \rangle)$ is in $\mathbb{C}_{\mathsf{P}}$, for $\beta\iota_{\mathsf{P}}(hd) = (\alpha, \langle S_{hd} \rangle)$ and $\beta\iota_{\mathsf{P}}(tl) = (\alpha, \langle S_{tl} \rangle)$, the requirement for arrows in $\mathbb{C}_{\mathsf{P}}$ being $S_{tl} \subseteq S_{hd}$ which is precisely the closure condition (5). This proves the

claim for the trivial one-step resolution from the head of a clause to its tail. Using the fact that for any arrow θ targeted at α $(\alpha, \theta\langle S_{tl}\rangle) \xrightarrow{id_\alpha} (\alpha, \theta\langle S_{hd}\rangle)$ is also in $\mathbb{C}_{\mathsf{P}}$, and that $G_1 \overset{\theta}{\rightsquigarrow} \cdots \rightsquigarrow G_2$ implies $\theta G_1 \overset{id}{\rightsquigarrow} \cdots \rightsquigarrow G_2$, it is straightforward to show that any resolution sequence with answer substitution θ maps to an arrow with label θ in the opposite direction. □

Theorem 25 (Completeness of $\beta \circ \iota_{\mathsf{P}}$) *If $\Box_\alpha \xrightarrow{\theta} \beta\iota_{\mathsf{P}}G$ is an arrow in $\mathbb{C}_{\mathsf{P}}$, then there is an SLD proof $G \overset{\theta}{\rightsquigarrow} \cdots \rightsquigarrow \Box_\alpha$.*

Proof. The empty goal of sort α is represented by the object $(\alpha, \langle\emptyset\rangle)$ in $\mathbb{C}_{\mathsf{P}}$. Suppose $\beta\iota_{\mathsf{P}}G = (\gamma, \langle U\rangle)$. That is to say G is of the form $X_{i_1}(t_1), \ldots, X_{i_n}(t_n)$ and U is

$$\bigcap \{t^{\#}(X_i) : t \in U_i, 1 \leq i \leq n\}. \tag{6}$$

Then if $\Box_\alpha \xrightarrow{\theta} \beta\iota_{\mathsf{P}}G$ is an arrow in $\mathbb{C}_{\mathsf{P}}$, $\theta\langle U\rangle \subseteq \langle\emptyset\rangle$, that is to say, by the remarks about T_{P} following definition (17), every $X_{i_j}(t_j)$ is true in the Herbrand interpretation $[\![\ ]\!]^*$ of P for $(1 \leq j \leq n)$. By completeness for $[\![\ ]\!]^*$ (theorem 23) there is an SLD proof of θG from P, hence a proof of G with computed answer substitution θ. □

3 The Categories $\mathbb{C}[\boldsymbol{\sigma}]$, $\mathbb{C}_{\mathsf{P}}[\boldsymbol{\sigma}]$ and $\mathbb{C}_{\boldsymbol{P}}^{\boldsymbol{SLD}}[\boldsymbol{\sigma}]$

Next, we generalize definition 2 so that the resultant construction will allow lifting of functorial datatypes and certain diagrams from the base category to the category of predicates. For the duration of this section $\mathbb{C}$ is an FP category, $\mathbb{D}$ is a complete category, $\mathbb{J}$ is a category, and $\mathcal{D}$ is a collection of product diagrams in $\mathbb{J}$, which we will call distinguished products, or **$\mathcal{D}$-products**.

Definition 26 ($\mathcal{D}$-Product Preservation) *We will say that the functor $H : J \longrightarrow \mathbb{B}$ preserves $\mathcal{D}$-products if H sends each $\mathcal{D}$-product to a product diagram in $\mathbb{B}$.*

For the rest of this paper $\sigma : \mathbb{J} \longrightarrow \mathbb{C}$ will be a functor that preserves $\mathcal{D}$-products and σ_i will denote the $\mathbb{C}$-object $\sigma(i)$.

Definition 27 ($\mathcal{D}$-Closure) *Let $A \in |\mathbb{C}|$, and $\mathcal{D}$ be a set of product diagrams in $\mathbb{C}$. A subfunctor S of $\mathbb{C}(A, \sigma(-))$ is $\mathcal{D}$-closed if it preserves $\mathcal{D}$-products. In other words, for every $\mathcal{D}$-product diagram for $i_1 \times \ldots \times i_n$, if there are arrows $f_k \in S(i_k)$ for each $k = 1, \ldots, n$, then $\langle f_1, \ldots, f_n\rangle \in S(i_1 \times \ldots \times i_n)$.*

In particular, if $\mathbf{1}_{\mathbb{J}} \in \mathcal{D}$, then $\sigma_{\mathbf{1}_{\mathbb{J}}} = \mathbf{1}_{\mathbb{C}}$ and $S(\mathbf{1}_{\mathbb{J}}) = \{!_A\}$.

Definition 28 ($\mathcal{D}$-Generating Sets) *Let Y be a collection of arrows emanating from A, each with target in the range of σ. Y is said to $\mathcal{D}$-generate S if S is the smallest $\mathcal{D}$-closed subfunctor of $\mathbb{C}(A, \sigma(-))$ containing all the arrows of Y. In that case we write $\ll Y \gg$ for S, and call it the $\mathcal{D}$-closure of Y.*

Definition 29 *Let* $\mathbb{C}[\sigma]$ *be defined as follows.*

objects: *pairs* $\langle A, S\rangle$ *where* $A \in |\mathbb{C}|$ *and* S *is a* $\mathcal{D}$*-closed subfunctor of* $\mathbb{C}(A, \sigma(-))$.

arrows: *are triples* $\langle A, S\rangle \xrightarrow{h} \langle B, T\rangle$ *where* $A \xrightarrow{h} B$ *is an arrow in* $\mathbb{C}$ *and* $hT \subset S$*, i.e. for any* $j \in |\mathbb{J}|$ *and* $f \in T(j)$*,* $hf \in S(j)$*. Again, we define a* **label** *as before, and compostion is inherited from* $\mathbb{C}$.

Observe that the functors S are no longer finitely generated.

A routine verification shows that if $\mathbb{J}$ is the discrete category on the set $\{1, \ldots, n\}$ and $\mathcal{D} = \emptyset$ then $\mathbb{C}[\sigma] = \mathbb{C}[X_1, \ldots, X_n]$. Hence Definition 2 is a special case of Definition 29.

Proposition 30 *If* $\mathbb{C}$ *is an FP category [resp. cartesian] then* $\mathbb{C}[\sigma]$ *is an FP category [resp. cartesian] and the functor* $\mathbb{C} \xrightarrow{\iota} \mathbb{C}[\sigma]$ *given by mapping objects* A *to* $\langle A, \ll \emptyset \gg\rangle$*, and arrows* $A \xrightarrow{f} B$ *to* $\langle A, \ll \emptyset \gg\rangle \xrightarrow{f} \langle B, \ll \emptyset \gg\rangle$*, is a full and faithful limit-preserving embedding.*

The proof is straightforward, but lengthy. The reader is referred to [17] for details.

Definition 31 *Let* $\mathbb{C}(\sigma)$ *stand for the full subcategory of* $\mathbb{C}[\sigma]$ *whose objects are*

$$|\iota(\mathbb{C})| \bigcup \{\langle \sigma_j, \ll Id_{\sigma_j} \gg\rangle | j \in |\mathbb{J}|\}.$$

Also, let $X : \mathbb{J} \longrightarrow \mathbb{C}(\sigma)$ *be the functor taking objects* $j \in |\mathbb{J}|$ *to the pair* $\langle \sigma_j, \ll Id_{\sigma_j} \gg\rangle$ *and arrows* $i \xrightarrow{h} j$ *to* $\langle \sigma_i, \ll Id_{\sigma_i} \gg\rangle \xrightarrow{\sigma_h} \langle \sigma_j, \ll Id_{\sigma_j} \gg\rangle$.

Finally, let $m : X \longrightarrow \iota \circ \sigma$ *be the natural transformation defined by* $m_j \equiv X_j \xrightarrow{id_{\sigma_j}} \iota(\sigma_j)$ *where* $j \in |\mathbb{J}|$.

It is straightforward to show the composite functor $\mathbb{J} \xrightarrow{X} \mathbb{C}(\sigma) \hookrightarrow \mathbb{C}[\sigma]$ preserves $\mathcal{D}$-products.

The monic m_j (and by abuse of language, its source X_j) will be called the **generic predicate of sort** σ_j since the family m exhibits similar characteristics to the generic predicates of Definition 1. In particular, for every $j \in |\mathbb{J}|$, $t \in \mathbb{C}(A, \sigma_j)$ the diagram

$$\begin{array}{ccc} \langle A, \ll t \gg\rangle & \xrightarrow{t} & X_j \\ {\scriptstyle Id_A}\downarrow & & \downarrow{\scriptstyle m_j} \\ \langle A, \ll \emptyset \gg\rangle & \xrightarrow[t]{} & \iota\sigma_j \end{array}$$

is a pullback in $\mathbb{C}[\sigma]$. However, it may be the case that some of these pullbacks are indeed isomorphisms: the constraint information contained in σ may make

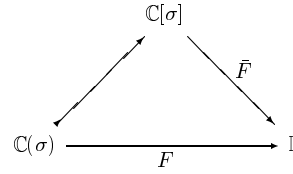
some generic predicates true. For example, suppose $\mathbf{1}_{\mathbb{J}}$ is in $\mathcal{D}$. Then it follows that $X_{\mathbf{1}_{\mathbb{J}}} = \sigma_{\mathbf{1}_{\mathbb{J}}} = \mathbf{1}_{\mathbb{C}}$ so that $X_{\mathbf{1}_{\mathbb{J}}} \xrightarrow{m_{\mathbf{1}_{\mathbb{J}}}} \sigma_{\mathbf{1}_{\mathbb{J}}}$ is iso.

The "genericity" of the family m is made precise in the following theorem.

Theorem 32 (Universal Mapping Property II) *Suppose $\mathbb{D}$ is complete category and $F : \mathbb{C}(\sigma) \longrightarrow \mathbb{D}$ is a functor such that*

- *$F \circ \iota : \mathbb{C} \longrightarrow \mathbb{D}$ is a cartesian representation;*
- *$F \circ X : \mathbb{J} \longrightarrow \mathbb{D}$ preserves $\mathcal{D}$-products; and*
- *$F(m) : F \circ X \longrightarrow F \circ \iota(\sigma)$ is a monic natural transformation.*

Then there exists a cartesian functor $\bar{F} : \mathbb{C}[\sigma] \longrightarrow \mathbb{D}$ unique up to isomorphism making the triangle



commute.

Proof(sketch). For $A \xrightarrow{t} \sigma_j$, let $\tilde{F}(t)$ be the pullback of $X_j \xrightarrow{Fm_j} F\iota\sigma_j$ along $F\iota(A) \xrightarrow[F(t)]{} F\iota\sigma_j$.

Define $\bar{F}\langle A, S\rangle$ on objects by $\bar{F}\langle A, S\rangle = \bigcap\{\tilde{F}(t) | t \in S(j), j \in |\mathbb{J}|\}$. $\bar{F}$ on arrows is just a consequence of the limit definition of $\bar{F}$ on objects. It follows that $\bar{F}$ has the stated properties. We refer the reader to [17] for the details. □

Corollary 33 *Assume the category $\mathbb{D}$ in Theorem 32 is $\mathsf{Set}^{\mathbb{C}^o}$ and that $F \circ \iota$ is the Yoneda embedding. Also, for each $i \in |\mathbb{J}|$ take $F(X_i) \xrightarrow{F(m_i)} F(\sigma_i)$ to be a pointwise subset of $F(\sigma_i)$, and take limits in $\mathsf{Set}^{\mathbb{C}^o}$ to be given pointwise. Then the functor $\bar{F}$ of Theorem 32 is unique. In this case we call the functor a σ-interpretation by analogy with definition 12, and write it $[\![\,]\!]$.*

The category $\mathbb{C}_{\mathsf{P}}[\sigma]$: We have given two constructions, one yielding a category $\mathbb{C}_{\mathsf{P}}$ of substitutions corresponding to proofs with respect to program P, and one yielding arrows that serve as proof steps for data information encapsulated in the index category $\mathbb{J}$ and the functor σ. Finally we define a category of σ-proofs $\mathbb{C}_{\mathsf{P}}[\sigma]$ by merging the two constructions. The objects are pairs (A, S), A in $|\mathbb{C}|$ and S a subfunctor of $\mathbb{C}(A, \sigma(-))$ closed under *both* the conditions 5 of definition 17 (clausal closure) and that of definition 27 ($\mathcal{D}$-closure). Arrows are induced by labels from $\mathbb{C}$ as before.

The category $\mathbb{C}_P^{SLD}[\sigma]$: Given the data $\mathbb{C}$, $\mathbb{J}$ and σ, we define an (unordered) σ-goal to be an object of $\mathbb{C}[\sigma]$, a σ-clause to be a triple $\langle A, G, X(t)\rangle$, where $X(t)$ is an atomic σ-goal (a pullback of a generic σ-predicate) and G a σ-goal, both of sort $A \in |\mathbb{C}|$, and a σ-program P a set of σ-clauses. We can construct a category

of generalized resolution proofs directly from $\mathbb{C}_{\mathsf{P}}[\sigma]$ and P by taking, as objects, the objects of $\mathbb{C}[\sigma]$, and as arrows, triples

$$\langle A, S\rangle \xrightarrow{\ h\ } \langle B, T\rangle$$

where $\langle A, \langle S\rangle_{\mathcal{D}}\rangle \xrightarrow{\ h\ } \langle B, \langle T\rangle_{\mathcal{D}}\rangle$ is an arrow in $\mathbb{C}_{\mathsf{P}}[\sigma]$, ($\langle S\rangle_{\mathcal{D}}$ being the $\mathcal{D}$-closure of S), and composition is given by composition of labels from $\mathbb{C}$. In other words, $\mathbb{C}_P^{SLD}[\sigma]$ is the object part of the comma category (κ, κ) induced by the functor $\kappa : \mathbb{C}[\sigma] \longrightarrow \mathbb{C}_{\mathsf{P}}[\sigma]$ which maps objects $\langle A, S\rangle$ to $\langle A, \langle S\rangle_{\mathcal{D}}\rangle$ and labels to themselves. In this category, arrows correspond to SLD-resolutions induced by program P, weakening, and proof steps induced by the datatype information encoded in σ. We then have soundness and completeness in the following sense

Theorem 34 *Let* P *be a* σ*-program and* G *be a* σ*-goal.* $[\![G]\!]$ *is true for every* σ*-interpretation* $[\![\ ]\!]$ *that is a model of* P *if and only if there is an arrow* $\square_A \xrightarrow{\ id_\alpha\ } G$ *in* $\mathbb{C}_P^{SLD}[\sigma]$.

4 Lifting Data Types to Predicates and Proofs

In this section we give several examples of the use of the machinery developed in preceding sections to incorprate datatype definitions into logic programs.

We take the following syntax for a datatype declaration.

```
datatype 'a foo = c0|c1 ... |k1 of E_1('a)| ... |kn of E_n('a)
```

with `'a` possibly a sequence of variables, and where each E_i(`'a`) is a type term, e.g. `'a * 'a list`, and `ki` the appropriate constructor (e.g. `cons`).

The category $\mathbb{C}$*:* We assume a finite product category $\mathbb{C}$ of data has been specified with objects corresponding to all required sorts together with an endofunctor $\mathsf{foo}\colon \mathbb{C} \longrightarrow \mathbb{C}$ and endofunctors $E_1, \ldots, E_n$ induced by the type terms E_i in the datatype declaration (e.g. $\lambda X.X \times \mathsf{list}\ (X)$) such that:

1. For each constant $\mathtt{c}_i$ in the datatype declaration there is a natural transformation $\lambda X.\mathbf{1} \xrightarrow{\ (c_i)\ } id$.
2. For each constructor $\mathtt{k}_i$ there is a natural transformation k_i from E_i to foo
3. The object $D(\alpha) = \mathbf{1} + \ldots + \mathbf{1} + E_1(\alpha) + \ldots E_n(\alpha)$ exists (as a coproduct) in $\mathbb{C}$, for every object α of $\mathbb{C}$.
4. The arrow
$$D(\mathsf{foo}\ \alpha) \xrightarrow{\ [c_0 \cdots c_m \_ k_1 \cdots k_n]\mathsf{foo}\ \alpha\ } \mathsf{foo}\ \alpha$$
is an initial algebra in $\mathbb{C}$ for every object α.

The techniques for building the appropriate category $\mathbb{C}$ are well known, and discussed in the literature, so we will not dwell upon this question here. The reader should consult e.g. [20,13,14,23,3].

Definition 35 (The Index category $\mathbb{J}$) *Let $\mathcal{S} = \{\gamma_1, \ldots, \gamma_n\}$ be the* multi-*set of sorts of predicates in the program (some sorts may be repeated) and let $\mathbb{N}$ be the preorder on the set $\{1, \ldots, n\}$ such that there is precisely one arrow of each type, i.e. all objects of $\mathbb{N}$ are isomorphic. $\mathbb{J}$ will be the smallest subcategory of $\mathbb{C} \times \mathbb{N}$ subject to the following conditions:*

1. *$(\mathbf{1}_{\mathbb{C}}, l), (\gamma_l, l) \in |\mathbb{J}|$ for all $l = 1, \ldots, n$;*
2. *for each $(\alpha, l) \in |\mathbb{J}|$ and $l' = 1, \ldots, n$, we have*

$$!_{(\alpha,l)} \in \mathbb{J}((\alpha, l), (\mathbf{1}_{\mathbb{C}}, l'));$$

3. *for each $(h, f) \in \mathbb{J}((\alpha, l), (\beta, l'))$, the following diagrams are included in $\mathbb{J}$;*

$$(\mathsf{foo}(\alpha), l) \xrightarrow{(\mathsf{foo}(h), f)} (\mathsf{foo}(\beta), l') \tag{7}$$

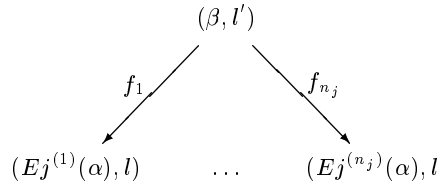
4. *for each constant ci in the declaration, $(\alpha, l) \in |\mathbb{J}|$, and $l' = 1, \ldots, n_j$,*

$$(ci, l' \Rightarrow l) \in \mathbb{J}((\mathbf{1}_{\mathbb{C}}, l'), (\mathsf{foo}(\alpha), l)),$$

   where $l' \Rightarrow l$ is the unique arrow in $\mathbb{N}(l', l)$;
5. *for each constructor kj in the datatype declaration and object $(\alpha, l) \in |\mathbb{J}|$, all product diagrams*

$$(Ej(\alpha), l) \xrightarrow{\pi_k} (Ej^{(k)}(\alpha), l) \qquad (1 \leq k \leq n_j) \tag{8}$$

   are included in $\mathbb{J}$.
   Given a $\mathbb{J}$-span



(9)

   it must be the case that

$$\langle f_1, \ldots, f_{n_J} \rangle \in \mathbb{J}((\beta, l'), (Ej(\alpha), l)).$$

The special products in $\mathcal{D}$ are precisely:

1. *the terminators $(\mathbf{1}_{\mathbb{C}}, l)$; and*
2. *the products of Diagram 8.*

Let $\sigma : \mathbb{J} \longrightarrow \mathbb{C}$ be the projection functor onto the first coordinate.

The construction implements the appropriate extension of SLD resolution in the following sense.

Theorem 36 *Given the data $\mathbb{J} \xrightarrow{\sigma} \mathbb{C}$ described above, the category $\mathbb{C}_{\mathsf{P}}[\sigma]$ contains all instances of the following arrows, for $\gamma \in \mathcal{S}$ and each constructor k_i:*

– $X_{E_i(\gamma)} \xrightarrow{(k_i)_\gamma} \mathsf{foo}(X_\gamma)$ *(where k_i is a label from $\mathbb{C}$)*

– $X_{E_i(\gamma)}(\langle t_1, \ldots, t_n\rangle) \xrightarrow{id_\gamma} \mathsf{foo}\ \mathsf{X}_\gamma(k_i(t_1, \ldots, t_n))$
– *All SLD proofs over the Horn Clauses in* P.

Observe that the predicate $\mathsf{foo}(X_\gamma)$ means $X_{\mathsf{foo}\gamma}$, which in the category $\mathbb{C}_{\mathsf{P}}[\sigma]$ is $(\mathsf{foo}\gamma, \ll id_{\mathsf{foo}\gamma} \gg)$. It is easily shown foo extends to an endofunctor on $\mathbb{C}_{\mathsf{P}}[\sigma]$ (The case where foo is a monad is studied in [17]).

In the case of e.g. the list datatype, the arrow $X_{E_1(\gamma)} \xrightarrow{k_1} \mathsf{foo}\ \mathsf{X}_\gamma$ becomes $X_{\gamma\times\mathsf{list}(\gamma)} \xrightarrow{cons} \mathsf{list}(X_\gamma)$ and $X_{E_i(\gamma)}(\langle t_1, \ldots, t_n\rangle) \xrightarrow{id_\gamma} \mathsf{foo}(X_\gamma)(k_i(t_1, \ldots, t_n))$ becomes $X_{\gamma\times\mathsf{list}(\gamma)}(\langle t_1, t_2\rangle) \xrightarrow{cons} \mathsf{list}(X_\gamma)(cons(t1, t2))$ which corresponds to the the proof rule

$$\mathsf{list}(X_\gamma)(cons(t1, t2)) \overset{id}{\rightsquigarrow} X_\gamma(t1), \mathsf{list}(X_\gamma)(t_2)$$

4.1 Encapsulation

Finally we show how to lift a datatype definition with member functions

```
datatype 'a foo = c0|c1 ... |k1 of E_1('a)| ... |kn of
  E_n('a)
begin
  fun f1 ...
  ...
end
```

where the functions between the `begin...end` are arrows $\alpha \xrightarrow{f} \rho$ in the original term category $\mathbb{C}$.

All we need to change is the definition of $\mathbb{J}$, which now must be a disjoint union of the $\mathbb{J}_\gamma$ and the diagrams $\alpha\times\alpha \xrightarrow{id_\alpha\times f} \alpha\times\rho$. The generic predicate $m_j \equiv X_j \xrightarrow[Id_{\sigma_j}]{} (\alpha\times\rho \ll id_{\alpha\times\rho} \gg)$ resulting from this extra component of $\mathbb{J}$ is called `pred[f]` in the program syntax. It will automatically satisfy the condition that `pred[f](X,f(X))` is an isomorphism in the categories $\mathbb{C}_{\mathsf{P}}[\sigma]$ and $\mathbb{C}_P^{SLD}[\sigma]$. That is to say, the construction "hard-wires" the graph of the function f into the binary relation `pred[f]`. This makes it possible for an implementation to supply the predicate without supplying the function, which would remain hidden. When interpreted in any model `pred[f](X,f(X))` will be true, and in any resolution proof in $\mathbb{C}_P^{SLD}[\sigma]$ this goal is isomorphic to the empty goal of its sort. We illustrate with an example based on the code in the introduction. We consider the `length` function given in the fragment

```
begin module "list"
datatype 'a list =  nil | cons of 'a * 'a list
  begin
    fun length nil = (0:int)
     |  length (cons (a,t)) = 1 + length t
  end;
```

which will give rise to an arrow $\mathsf{list}(\gamma) \xrightarrow{length} int$ in $\mathbb{J}$ (and in $\mathbb{C}$), and to the arrow $\mathsf{list}(\gamma) \xrightarrow{\langle id, length\rangle} \mathsf{list}(\gamma) \times int$ in $\mathbb{J}$, mapped by σ to itself in $\mathbb{C}_{\mathsf{P}}[\sigma]$. The objects in category $\mathbb{C}_{\mathsf{P}}[\sigma]$ are pairs (A, S) where S must satisfy the closure requirement:

$$for\ every\ arrow A \xrightarrow{\varphi} \mathsf{list}(\gamma), \varphi \circ \langle id, length\rangle\ is\ in\ S(\mathsf{list}(\gamma) \times int) \tag{9}$$

since S is a subfunctor of $\mathbb{C}(A, \sigma(\_))$. The predicate `pred[length]` is represented in $\mathbb{C}_{\mathsf{P}}[\sigma]$ and in $\mathbb{C}_P^{SLD}[\sigma]$ as the generic

$$(\mathsf{list}(\gamma) \times int, \ll id \gg) \xrightarrow[id]{} (\mathsf{list}\gamma \times int, \ll \emptyset \gg)$$

Its instantiation (pullback) along $\langle id, length\rangle$ is

$$(\mathsf{list}(\gamma), \ll \langle id, length\rangle \gg) \xrightarrow[id]{} (\mathsf{list}\gamma, \ll \emptyset \gg)$$

which has the inverse $(\mathsf{list}(\gamma), \ll \langle id, length\rangle \gg) \xleftarrow[id]{} (\mathsf{list}\gamma, \ll \emptyset \gg)$ since, perhaps suprisingly at first, $\ll \langle id, length\rangle \gg \subseteq \ll \emptyset \gg$ by condition (9) with $A = \mathsf{list}(\gamma)$ and φ the identity. Therefore the (reverse SLD step corresponding to the) arrow $\square_\sigma \xrightarrow{id}$ `pred[length]`$(\langle id, length\rangle)$ is in $\mathbb{C}_P^{SLD}[\sigma]$.

5 Conclusion and Future Work

We have described a construction of a category of predicates over a base category $\mathbb{C}$, generic up to satisfaction of program clauses and datatype definitions, which gives a category of non-deterministic resolution proofs that implement extended resolution steps capturing program, datatype and base-category information. The framework is proposed here as a blueprint for incorporating certain types of extensions into constraint logic programming. It has been used elsewhere to extend Hereditarily Harrop logic programming with constraints on terms [10,7] and with monads [17]. We hope it will prove a natural vehicle for declarative approach to the *limited* amounts of control and state that are needed in logic programming as well.

Conspicuously absent in this study is a systematic approach to a deterministic implementation of an abstract machine based on this framework, a set of categorical narrowing rules for execution of the code, and reduction rules for computation of pullbacks (or a canonical choice of unifiers) in the base category $\mathbb{C}$, when $\mathbb{C}$ is suitably chosen (for example with recursively enumerable pullbacks).

References

1. A. Asperti and S. Martini. Projections instead of variables, a category theoretic interpretation of logic programs. In *Proc. 6^{th} ICLP*, pages 337–352. MIT Press, 1989. 392, 393

2. R. Barbuti, M Bellia, and G. Levi. *Leaf: a Language which Integrates Logic Equations and Functions*. Prentice-Hall, 1984. 393
3. Michael Barr and Charles F. Wells. *Category Theory for Computing Science*. Prentice-Hall International, Englewood Cliffs, NJ, USA, 1990. 406
4. A. Corradini and A. Asperti. A categorical model for logic programs: Indexed monoidal categories. In *Proceedings REX Workshop '92*. Springer Lecture Notes in Computer Science, 1992. 392, 393
5. Andrea Corradini and Ugo Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Theoretical Computer Science*, 103:51–106, 1992. 392, 393
6. Roy Crole. *Categories for Types*. Cambridge University Press, 1993. 394, 396
7. Mary de Marco. *Dynamic Constraints and Hereditarily Harrop Programming*. PhD thesis, Wesleyan University, 1999. 409
8. R. Diaconescu. *Category Semantics for Equational and Constraint Logic Programming*. PhD thesis, Oxford University, 1994. 393
9. Stacy E. Finkelstein, Peter Freyd, and James Lipton. Logic programming in tau categories. In *Computer Science Logic '94, LNCS 933*. Springer, 1995. 392, 393, 394, 401, 402
10. Stacy E. Finkelstein, Peter Freyd, and James Lipton. A new framework for declarative programming. *Theoretical Computer Science*, To appear. Technical report available as `www.cs.wesleyan.edu/~lipton/jftp/dist96-nfdp.ps`. 393, 394, 401, 402, 409
11. Peter Freyd and Andre Scedrov. *Categories, Allegories*. North-Holland, 1990. 393, 394
12. Joseph Goguen and José Meseguer. *Equality Types and Generic Modules for Logic Programming*. Prentice-Hall, 1984. 393
13. Tatsuya Hagino. *A Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, Department of Computer Science, 1987. CST-47-87 (also published as ECS-LFCS-87-38). 406
14. Bart Jacobs and Jacob Rutten. *A Tutorial on (Co)algebras and (Co)induction*, volume 62, pages 222–259. 1997. 406
15. M. Martelli M. Falaschi, G. Levi and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *TCS*, 69(3), 1989. 400
16. Michael Makkai and Gonzalo Reyes. *First Order Categorical Logic*, volume 611 of *Lecture Notes in Mathematics*. Springer-Verlag, 1977. 393
17. R. McGrail. *Modules,Monads and Control in Logic Programming*. PhD thesis, Wesleyan University, 1998. To appear. 404, 405, 408, 409
18. F. Orejas, E. Pino, and H. Ehrig. Algebraic methods in the compositional analysis of logic programs. In P. Ruzicka I. Privara, B. Rovan, editor, *Proc. MFCS 94*, volume 841 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. 393
19. P. Panangaden, V. Saraswat, P.J. Scott, and R.A.G. Seely. A hyperdoctrinal view of constraint systems. In *Lecture Notes in Computer Science 666*. Springer Verlag, 1993. 393
20. A. Poigné. Algebra categorically. In *Category Theory and Computer Programming*. Springer, 1986. 406
21. J. Power and Y. Kinoshita. A new foundation for logic programming. In *Extensions of Logic Programming '96*. Springer Verlag, 1996. 392, 393
22. D. Pym. Functorial kripke models of the $\lambda\pi$-calculus. Lecture at Newton Institute Semantics Programme, Workshop on Category Theory and Logic Programming, Cambridge, September 1995, 1995. 393

23. H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987. 406
24. D.E. Rydeheard and R.M. Burstall. A categorical unification algorithm. In *Category Theory and Computer Programming*, 1985. 393

Constructive Negation Using Typed Existence Properties

John G. Cleary and Lunjin Lu

Department of Computer Science
University of Waikato
Hamilton, New Zealand
Phone: +64-7-8384627, Fax: +64-7-8384155
{jcleary,lunjin}@cs.waikato.ac.nz

Abstract. A method for extracting positive information from negative goals is proposed. The method makes use of knowledge about the existence of solutions of predicates and about the types of variables. It strictly generalises earlier work in this area. It can be used as a construction rule or as a simplification rule. As a construction rule it does not involve an SLD-derivation of the negated sub-goal and so is applicable during compilation. As a simplification rule it prunes unsatisfiable goals without doing an explicit satisfiability test.

Keywords: constructive negation, *existence* properties, type system

1 Introduction

The "negation by failure" rule, NAF for short, may lead to floundering when negative goals contain variables. The reason is that NAF doesn't allow negative goals to bind variables. To overcome this limitation, Chan introduced the "constructive negation" rule which allows non-ground negative goals to bind variables in the same way as positive ones [2,3]. The basic idea is that answers to $\neg\ Q$ are obtained by negating answers to Q. Given $\neg\ Q$, a frontier of a derivation tree for Q is first obtained. Answers to $\neg\ Q$ are then obtained from the frontier as first-order formulae which are interpreted in Clark's equality theory (CET). Chan's method was formulated for logic programs in the Herbrand universe and involves introducing disequality constraints over the Herbrand universe. An answer to a goal by Chan's operational semantics SLD-CNF is a set of equality and disequality constraints. Originally, Chan's method applied only to negative goals with finite sub-derivation trees and worked by negating answers to the negated sub-goal [2]. Chan later extended his method by negating a frontier of a derivation tree for the negated sub-goal [3]. The simplification procedure in Chan's method relies on the following property of the Herbrand universe.

$$\neg\exists\mathbf{y},\mathbf{z}.(x = s(\mathbf{y}) \wedge Q(\mathbf{y},\mathbf{z})) \leftrightarrow \forall\mathbf{y}.(x \neq s(\mathbf{y})) \vee \exists\mathbf{y}.(x = s(\mathbf{y}) \wedge \neg\exists\mathbf{z}.Q(\mathbf{y},\mathbf{z}))$$

where x is a free variable, bold letters denote vectors of *different* variables, $\mathbf{y}$ and $\mathbf{z}$ are disjoint and variables in $\mathbf{z}$ don't occur in $\mathbf{y}$.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 411–426, 1998.

Foo et. al propose an approach for constructive negation for Datalog programs [9]. A Datalog program with negation is first transformed by adding typing goals before its negative goals. Each argument of a negated sub-goal corresponds to a typing goal, called the relevant type of the argument. The typing goal enumerates all possible values for the argument. In this way, the negative goals in the transformed program are always ground when it is selected and floundering is avoided. The relevant type for a program is a superset of the success set of the program and is derived from the program.

Małuszyński and Näslund put forward another approach to constructive negation which allows a negative goal to directly return fail substitutions, as its answers [13]. Since answers to negative goals cannot in general be represented by a finite number of substitutions, Małuszyński and Näslund's approach sometimes needs to return an infinite number of fail substitutions.

Drabent defines SLDFA resolution over the Herbrand universe [7]. Chan's first method works only when the negated sub-goal has a finite number of answers. SLDFA overcomes this by constructing answers for the negative goal from a finite number of answers to the negated sub-goal.

Fages proposes a simple concurrent pruning mechanism over standard SLD derivation trees for constructive negation in constraint logic programs [8]. Two derivation trees are concurrently constructed. The computed answers from one of the trees are used to prune the nodes of the other. Fages' method admits an efficient implementation as it is not necessary to deal with complex goals with explicit quantifiers outside the constraint part.

Stuckey provides a constructive negation method for constraint logic programs over arbitrary structures [17]. Stuckey's method which is sound and complete with respect to the three-valued consequences of the completion of the program can be thought of as a generalisation of Chan's. Stuckey uses the following property of logic formulae in his simplification procedure.

$$\neg\exists \mathbf{y}.(c \wedge Q) \leftrightarrow \neg\exists \mathbf{y}.c \vee \neg\exists \mathbf{y}.(c \wedge Q)$$

where c is a constraint and Q is a conjunction of goals. The method needs to do a satisfiability test when combining $\neg\exists \mathbf{y}.c$ with other constraints.

Cleary makes use of domain knowledge and *existence* properties of arithmetic constraints to construct answers to negative goals [4]. There are usually functional dependencies between arguments to an arithmetic constraint. Let $add(x, y, z)$ denote addition on the domain of integers, for any integers x and y, then there is a unique z such that $add(x, y, z)$ is true. This is called an *exists unique* property. It implies that $\neg\exists z.add(x, y, z)$ is unsatisfiable and that $\neg\exists z.(add(x, y, z) \wedge q(z))$ can be directly simplified to $add(x, y, z) \wedge \neg q(z)$. Another kind of property is called the *exists sometimes* property which corresponds to partial functional dependencies between arguments to an arithmetic constraint. Let $log(y, x)$ denote $y = 10^x$ on the domain of integers. Then there is at most one x such that $log(y, x)$ is true. So, we can directly simplify $\neg\exists x.(log(y, x) \wedge q(x))$ to $\neg\exists x.log(y, x) \vee log(y, x) \wedge \neg q(x)$. Yet another kind of property is called the *exists* property. For instance, for any integer x, there are integers y and z such

that $add(x, y, z)$ is true. *Exists* properties can be used to derive falsity of negative goals such as $\neg\exists y, z.add(x, y, z)$. [4] offers a rewrite rule for each of these three kinds of property. The simplification procedure in [4] consists of rewrite rules for these kinds of property plus some miscellaneous rewrite rules such as $\neg(x > y) \leftrightarrow (x \leq y)$ provided x and y range over the domain of numbers.

Unlike other constructive negation methods [3,7,8,13,17], the method in [4] does not necessarily rely on SLD resolution to obtain a frontier of the negated sub-goal, and can be used both as a constructive rule and as a simplification rule. Though the method was proposed for arithmetic constraints, the basic principles behind the method carries over to other goals so long as they satisfy some available *existence* properties.

The prerequisite that a functional or partial functional dependency exists between arguments to a predicate (arithmetic constraints in [4]) is over restrictive. Consider $sq(x, y)$ in the domain of real numbers. For every x, there is a unique y such that $sq(x, y)$ is true. However, for every $y > 0$, there are two x's such that $sq(x, y)$ is true. The rewrite rule for *exists unique* properties in [4] doesn't apply directly when it comes to constructing answers to $\neg\exists x.(sq(x, y) \wedge b(x))$. This problem is resolved by inserting a tautology $(x \geq 0 \vee x < 0)$ into the negative goal and transforming $\neg\exists x.(sq(x, y) \wedge b(x))$ into $\neg\exists x_1.(sq(x_1, y) \wedge x_1 \geq 0 \wedge b(x_1)) \wedge \neg\exists x_2.(sq(x_2, y) \wedge x_2 < 0 \wedge b(x_2))$ and then applying the rewrite rule for *exists unique* properties to the two negative sub-goals. This causes difficulty because we need to have *exists unique* properties for complex constraints $(sq(x_1, y) \wedge x_1 \geq 0)$ and $(sq(x_2, y) \wedge x_2 < 0)$. Moreover, inserting a correct tautology, say $(x \geq 0 \vee x < 0)$, into the negative goal before rewriting is involved and difficult to mechanise.

Types are essential in any practical implementation of the constructive negation method in [4] as it is necessary to constrain the range of input and output values in *existence* properties. Though [4] allows the use of types in *existence* properties in the context of arithmetic constraints, it does not provide a theoretical framework for doing so.

This paper generalises the constructive negation method in [4] by generalising the notion of an *existence* property and incorporating a type system. In a generalised *existence* property, an input value may now correspond to multiple output values provided each of the output values can be isolated into a subdomain. Thus, the generalised method is applicable to more negative goals than the original one. A typed version of *existence* properties and rewrite rules are formulated by expressing (sub-)domains as types. Types also allow *existence* properties to be more concise and precise. The generalised method can be used either as a construction rule or as a simplification rule. As a construction rule, it does not involve sub-SLD derivation of the negated sub-goal. As a simplification rule, it prunes unsatisfiable goals without doing an explicit satisfiability test. We also note that Chan's simplification rule can be fully characterised by a set of *existence* properties.

The rest of the paper is organised as follows. Section 2 informally presents the type system. Section 3 formulates the typed version of *existence* properties

and rewrite rules and concludes with a number of examples. Section 4 concludes the paper. Proofs are omitted due to limited space.

2 Type System

This section informally describes a type system which supports inclusion, parametric and overloading polymorphism. The type system is expressive and decidable. The reader is referred to [16] for more details on type systems in logic programming.

A *type* is a finite expression denoting a possibly infinite set of ground terms. Specifically, a type denotes a set of ground terms which is a regular term language. We first introduce the notion of a type program. A type term is a term constructible from a denumerable set Para of *type parameters* β, β_i, and a set Cons of *type constructors*. Cons is divided two groups: Cons_f containing type constants with fixed denotations and Cons_d containing type constructors whose denotations are determined by type definitions. Cons_f contains two special type constants none denoting the empty set of ground terms and all denoting the Herbrand universe $\mathcal{HU}$.

Example 1. The following will be used to illustrate the type system. Let

$$\begin{aligned}\mathsf{Cons}_f = \{&\mathbb{R}, \mathbb{Z}, \mathsf{none}, \mathsf{all}\}\\ &\cup\{\mathbb{Z}_{[L,H]} \mid L, H \textit{ are integers} \wedge L \leq H\}\\ &\cup\{\mathbb{R}_{[L,H]} \mid L, H \textit{ are real numbers} \wedge L \leq H\}\end{aligned}$$

The type constants in Cons_f have the following fixed denotations. $\mathbb{R}$ denotes $\{-\infty, \infty\}$ plus the set of reals, $\mathbb{Z}$ denotes $\{-\infty, \infty\}$ plus the set of integers, $\mathbb{Z}_{[L,H]}$ denotes the integer interval $[L..H]$ and $\mathbb{R}_{[L,H]}$ denotes the closed real interval $[L..H]$.

Following [10] and [12], we define types by type clauses. A type clause is either of the form $f(x_1, \cdots, x_n) : c(\beta_1, \cdots, \beta_m) \leftarrow x_1 : \mathcal{G}_1, \cdots, x_n : \mathcal{G}_n$ or of the form $x : c(\beta_1, \cdots, \beta_m) \leftarrow x : d(\beta'_1, \cdots, \beta'_l)$ where $m, n, l \geq 0$, $c/m \in \mathsf{Cons}_d, d/l \in \mathsf{Cons}$, f/n is a function symbol, $\beta_1, \cdots, \beta_m$ are different type parameters, $x_1, \cdots, x_n$ are different program variables, each β'_j is a type parameter, and each $\mathcal{G}_j$ is either a type parameter or a type constructor applied to type parameters. Type clauses are required to be *type preserving* [18] in that any type parameter occurring in the body also occurs in the head.

Example 2. Let $\mathsf{Cons}_d = \{\mathsf{nat}, \mathsf{man}, \mathsf{woman}, \mathsf{person}, \mathsf{list}\}$. The programmer may define types by the following.

$$\begin{array}{lll}
0 : \mathsf{nat} \leftarrow & \% & D1\\
s(x) : \mathsf{nat} \leftarrow x : \mathsf{nat} & \% & D2\\
[\,] : \mathsf{list}(\beta_1) \leftarrow & \% & D3\\
[h|l] : \mathsf{list}(\beta_2) \leftarrow & \% & D4\\
\quad h : \beta_2, l : \mathsf{list}(\beta_2) & &
\end{array}
\qquad
\begin{array}{lll}
\mathit{peter} : \mathsf{man} \leftarrow & \% & D5\\
\mathit{john} : \mathsf{man} \leftarrow & \% & D6\\
\mathit{mary} : \mathsf{woman} \leftarrow & \% & D7\\
x : \mathsf{person} \leftarrow x : \mathsf{man} & \% & S1\\
x : \mathsf{person} \leftarrow x : \mathsf{woman} & \% & S2
\end{array}$$

D1 says that 0 is a member of type nat. D2 means that if x has type nat then $s(x)$ has type nat. D3 states that [ ] is a member of type list(β_1) for any type β_1 while D4 reads that, for any type β_2, if h is a member of type β_2 and l a member of type list(β_2) then $[h|l]$ is a member of type list(β_2). D5 and D6 mean that *peter* and *john* are members of man while D7 means that *mary* is a member of woman. S1 and S2 declare that man and woman are subtypes of person.

User-defined type clauses induce a type program Φ as in [6]. Φ consists of the user-defined type clauses, as well as $f(x_1, \cdots, x_n) : \mathsf{all} \leftarrow x_1 : \mathsf{all}, \cdots, x_n : \mathsf{all}$ for each function symbol f/n, and type clauses $t : c_f \leftarrow$ for any other $c_f \in \mathsf{Cons}_f$ and each constant t of type c_f.

Monomorphic type terms, denoted by $\mathcal{M}$,$\mathcal{N}$, are type terms without type parameters. The meaning $\mathrm{\not\exists}_{\Phi}(\mathcal{M})$ of a monomorphic type term $\mathcal{M}$ with respect to a type program Φ is defined as the set of ground terms t such that $t : \mathcal{M}$ is a logical consequence of Φ.

$$\mathrm{\not\exists}_{\Phi}(\mathcal{M}) \stackrel{def}{=} \{t \in \mathcal{HU} \mid \Phi \models t : \mathcal{M}\}$$

$\mathrm{\not\exists}_{\Phi}(\mathcal{M})$ is a regular term language, according to the following lemma.

Lemma 1. *Given a type program Φ, $\mathrm{\not\exists}_{\Phi}(\mathcal{M})$ is a regular term language for any monomorphic type term $\mathcal{M}$.*

In the sequel, we shall assume a fixed Φ and drop Φ from $\mathrm{\not\exists}_{\Phi}$.

We now introduce two mutually recursively defined notions of a typed term and a type. A term is typed by typing its variables. Let $x{:}\sigma$ denote that variable x is of type σ. $x : \sigma$ means that any term assigned to the variable x must be a member of σ. As its variables are constrained by types, possible instances of a term are restricted. For example, $f(a)$ is not a legitimate instance of $f(x : \mathbb{Z})$ since a is not an integer. If a variable in a term is not typed, it can be thought of as being typed by all.

A special function value maps a typed term to a type, called a singleton type. value(T) with T being a typed term is treated as a set expression [1].

$$\mathrm{\not\exists}(\mathsf{value}(x : \sigma)) \stackrel{def}{=} \mathrm{\not\exists}(\sigma)$$

$$\mathrm{\not\exists}(\mathsf{value}(f(T_1, \cdots, T_n))) \stackrel{def}{=} \{f(t_1, \cdots, t_n) \mid t_i \in \mathrm{\not\exists}(\mathsf{value}(T_i))\}$$

$\mathrm{\not\exists}(\mathsf{value}(T))$ is a regular term language of ground terms, according to the following lemma.

Lemma 2. *For any typed term T, if the types of variables in T are regular term languages then $\mathrm{\not\exists}(\mathsf{value}(T))$ is a regular term language.*

A type is an expression consisting of monomorphic type terms, singleton types, and boolean operators and, or and not. Notice that the use of boolean operators are restricted by the definition of a type clause. For instance, list(person or nat) and list(not($\mathbb{Z}$)) are not types. or, and and not are defined respectively as set union, set intersection and set complement.

$$\begin{aligned} \not\ni(\sigma_1 \text{ and } \sigma_2) &\stackrel{def}{=} \not\ni(\sigma_1) \cap \not\ni(\sigma_2) \\ \not\ni(\sigma_1 \text{ or } \sigma_2) &\stackrel{def}{=} \not\ni(\sigma_1) \cup \not\ni(\sigma_2) \\ \not\ni(\text{not}(\sigma)) &\stackrel{def}{=} \mathcal{HU} \setminus \not\ni(\sigma) \end{aligned}$$

Define $\sigma \sqsubseteq \eta \stackrel{def}{=} \not\ni(\sigma) \subseteq \not\ni(\eta)$ and $\sigma \equiv \eta \stackrel{def}{=} \not\ni(\sigma) = \not\ni(\eta)$. Types are regular term languages since monomorphic type terms and singleton types are regular term languages which are are closed under set intersection, union and complement. The inclusion and equivalence problems of regular term languages are known to be decidable [11]. So, the validity of $\sigma \sqsubseteq \eta$ and $\sigma \equiv \eta$ can be effectively tested.

Example 3. Continuing examples 1 and 2, we have

$$\begin{aligned} \not\ni(\text{person}) &= \{peter, john, mary\} \\ \not\ni(\text{list(none)}) &= \{[\,]\} \\ \not\ni(\text{list(all)}) &= \{[\,], \cdots, [x], \cdots, [y, s(0)], \cdots\} \end{aligned}$$

$\mathbb{R} \equiv \mathbb{R}_{[-\infty,\infty]}$, $\mathbb{Z} \equiv \mathbb{Z}_{[-\infty,\infty]}$, and $\mathbb{Z}_{[0,\infty]}$ denotes the set of non-negative integers. $[\,]$ is in $\not\ni(\text{list}(\mathcal{M}))$ for any monomorphic type term $\mathcal{M}$. Note that $\text{nat} \not\equiv \mathbb{Z}_{[0,\infty]}$ as $\not\ni(\text{nat})$ and $\not\ni(\mathbb{Z}_{[0,\infty]})$ are different sets of terms. Open real intervals can be expressed by type expressions. For instance, $\mathbb{R}_{[0,1]}$ and not(value(0)) is a type expression denoting the set of real numbers from 0 to 1 but excluding 0.

The type system is powerful and yet decidable. It supports inclusion, parametric and overloading polymorphism. Types are regular term languages. This is the same as in [6]. However, [6] does not support parametric polymorphism as type rules are not parameterised. Other type systems that support parametric polymorphism [10,15,19,14] use types to denote tuple-distributive sets of terms which are strictly less powerful than regular term languages. We have completed an efficient implementation of a subset of the type system and are working towards a full and efficient implementation.

In the next section, we shall use the following abbreviations for type expressions. $\mathbb{R}_{>0}$ stands for ($\mathbb{R}_{[0,\infty]}$ and not value(0)) that denotes the set of real numbers greater than zero, $\mathbb{R}_{<0}$ for ($\mathbb{R}_{[-\infty,0]}$ and not value(0)) that the set of real numbers less than zero, $\mathbb{R}_{\geq 0}$ for $\mathbb{R}_{[0,\infty]}$, $\mathbb{Z}_{>0}$ for $\mathbb{Z}_{[1,\infty]}$, and $\mathbb{Z}_{<0}$ for $\mathbb{Z}_{[-\infty,-1]}$.

3 Rewrite Rules

This section revises Cleary's rewrite rules for constructive negation [4]. We first revise these rewrite rules by relaxing their application conditions and then formalise a typed version of the revised rewrite rules. The rewrite rule making use

of *exists unique* properties is revised as follows. For every input value, the predicate holds for a fixed number of output values each of which can be isolated into a sub-domain. For instance, each positive number has two square roots one of which is positive and the other is negative. The rewrite rule making use of *exists sometimes* properties is revised in the same manner, so that every input value has at most one output value on each of a fixed number of sub-domains.

3.1 Basic Rewrite Rules

Exists unique The rewrite rule for *exists unique* properties requires that, for every $\mathbf{u}$, there be exactly one $\mathbf{x}$ such that $A(\mathbf{ux})$ holds. In other words, "A" must be a function from the domain of $\mathbf{u}$ to that of $\mathbf{x}$. $\mathbf{u}$ and $\mathbf{x}$ can be viewed as input and output parameters respectively. The predicate "A" may satisfy more than one *exists unique* property with different groups of input and output parameters.

The revised rewrite rule for *exists unique* properties requires that there be sub-domains such that, for any $\mathbf{u}$, there is exactly one $\mathbf{x}$ in each of those sub-domains such that $A(\mathbf{ux})$ holds. It also requires that, for any $\mathbf{u}$, any $\mathbf{x}$ such that $A(\mathbf{ux})$ holds lies in one of those sub-domains. Let $D_i(\mathbf{x}), i \in I$ be sub-domains (type constraints) where I is a finite set of indices. Then

$$\forall \mathbf{ux}.[A(\mathbf{ux}) \rightarrow \vee_{i \in I} D_i(\mathbf{x})] \tag{1}$$

$$\forall \mathbf{u}.\ \wedge_{i \in I}\ \exists! \mathbf{x}_i.[A(\mathbf{ux}_i) \wedge D_i(\mathbf{x}_i)] \tag{2}$$

where $\exists!$ means "there is exactly one".

The revised rewrite rule is obtained as follows. From (1), we have $A(\tilde{\mathbf{u}}\mathbf{x}) \leftrightarrow A(\tilde{\mathbf{u}}\mathbf{x}) \wedge [\vee_{i \in I} D_i(\mathbf{x})]$ where bold letters with tildes denote vectors of terms that are not necessarily different. Hence $A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q \leftrightarrow \vee_{i \in I} A(\tilde{\mathbf{u}}\mathbf{x}) \wedge D_i(\mathbf{x}) \wedge Q$. Distributing $\exists$ over $\vee$, renaming existentially quantified variables within their scopes and applying De Morgan's law result in the following:[1]

$$\neg\exists \mathbf{xy}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q] \leftrightarrow \wedge_{i \in I} \neg\exists \mathbf{x}_i\mathbf{y}.[A(\tilde{\mathbf{u}}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i) \wedge Q[\mathbf{x} := \mathbf{x}_i]]$$

where $Q[\mathbf{x} := \mathbf{x}_i]$ is the result of substituting $\mathbf{x}_i$ for $\mathbf{x}$ in Q. From (2), we obtain the following revised rewrite rule where $\mathbf{V}_E$ is the set of variables in E.

Q	Given (1) and (2) and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\mathbf{x} \cup \mathbf{y}) = \emptyset$
	$\neg\exists \mathbf{xy}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q] \leftrightarrow \wedge_{i \in I}(A(\tilde{\mathbf{u}}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i) \wedge \neg\exists \mathbf{y}.Q[\mathbf{x} := \mathbf{x}_i])$

The condition $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\mathbf{x} \cup \mathbf{y}) = \emptyset$ ensures that $\tilde{\mathbf{u}}$ does not contain existentially quantified variables also called *local* variables. That this is necessary is shown as follows. Assume the *exists unique* property for integer addition in the introduction, $\neg\exists y.(add(x, y, y) \wedge q(y))$ cannot be simplified to $add(x, y, y) \wedge \neg q(y)$ because $\neg\exists y.add(x, y, y)$ holds for $x \neq 0$. The fact that the second argument y to add is

[1] $\mathbf{x}$ has been renamed into $\mathbf{x}_i$ for each sub-domain D_i.

a local variable invalidates the condition. The sub-domains needn't be disjoint though it is computationally more efficient if they are.

Example 4. The fact that, in the domain of real numbers, a positive number has exactly one negative square root and exactly one positive square root can be expressed as the following *exists unique* property.

$$\forall y > 0.\forall x.(sq(x,y) \rightarrow x < 0 \vee x > 0)$$
$$\forall y > 0.(\exists! x_1.(sq(x_1,y) \wedge x_1 < 0) \wedge \exists! x_2.(sq(x_2,y) \wedge x_2 > 0))$$

Suppose $y > 0$, the negative goal $\neg\exists x.(sq(x,y) \wedge b(x))$ is rewritten into $sq(x_1,y) \wedge x_1 < 0 \wedge \neg\, b(x_1) \wedge sq(x_2,y) \wedge x_2 > 0 \wedge \neg\, b(x_2)$.

Exists sometimes The revision of the rewrite rule for *exists sometimes* properties is similar to that for *exists unique* properties. (2) is now replaced by the following requirement.

$$\forall \mathbf{u}.\ \wedge_{i\in I}\ \exists?\mathbf{x}_i.[A(\mathbf{u}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i)] \tag{3}$$

where $\exists?$ denotes "there is at most one". (3) requires that, for each $\mathbf{u}$, there is at most one $\mathbf{x}$ in each sub-domain such that $A(\mathbf{u}\mathbf{x})$ holds. The revised rewrite rule for *exists sometimes* properties follows.

S	Given (1) and (3) and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\mathbf{x} \cup \mathbf{y}) = \emptyset$
	$\neg\exists \mathbf{x}\mathbf{y}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q] \leftrightarrow$
	$\wedge_{i\in I}(\neg\exists \mathbf{x}_i.[A(\tilde{\mathbf{u}}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i)] \vee A(\tilde{\mathbf{u}}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i) \wedge \neg\exists \mathbf{y}.Q[\mathbf{x} := \mathbf{x}_i])$

Example 5. The fact that, in the domain of integer numbers, a positive number has at most one negative square root and at most one positive square root can be expressed as the following *exists sometimes* property.

$$\forall y > 0.\forall x.(sq(x,y) \rightarrow x < 0 \vee x > 0)$$
$$\forall y > 0.(\exists? x_1.(sq(x_1,y) \wedge x_1 < 0) \wedge \exists? x_2.(sq(x_2,y) \wedge x_2 > 0))$$

Suppose $y > 0$, the negative goal $\neg\exists x.(sq(x,y) \wedge b(x))$ is rewritten into $(\neg\exists x_1.(sq(x_1,y) \wedge x_1 < 0) \vee sq(x_1,y) \wedge x_1 < 0 \wedge \neg\, b(x_1)) \wedge (\neg\exists x_2.(sq(x_2,y) \wedge x_2 > 0) \vee sq(x_2,y) \wedge x_2 > 0 \wedge \neg\, b(x_2))$.

Exists There is no revision to *exists* properties and corresponding rewrite rules at this stage as the knowledge of subdomains in which output values lie is irrelevant.

E	Given $\forall \mathbf{u}.\exists \mathbf{x}.A(\mathbf{u}\mathbf{x})$ and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap \mathbf{x} = \emptyset$
	$\neg\exists \mathbf{x}.A(\tilde{\mathbf{u}}\mathbf{x}) \leftrightarrow$ false

Miscellaneous The miscellaneous rewrite rule makes use of properties $\forall \mathbf{u}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ to rewrite a negative goal into a positive one.

R	Given $\forall \mathbf{u}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$
	$\neg A(\tilde{\mathbf{u}}) \leftrightarrow B(\tilde{\mathbf{u}})$

3.2 Typed Rewrite Rules

Types make concise the expression of an *existence* property as domains are types and sub-domains are subtypes.

Example 6. The *exists unique* property in example 4 can be expressed as follows. [2]

$$\forall y{:}\mathbb{R}_{>0}.\forall x.(sq(x,y) \rightarrow x \in \mathbb{R}_{>0} \vee x \in \mathbb{R}_{<0})$$
$$\forall y{:}\mathbb{R}_{>0}.(\exists! x_1{:}\mathbb{R}_{>0}.sq(x_1,y) \wedge \exists! x_2{:}\mathbb{R}_{<0}.sq(x_2,y))$$

Example 7. The fact that the square of any real number is a positive real number is expressed as follows.

$$\forall x{:}\mathbb{R}.\forall y.(sq(x,y) \rightarrow y \in \mathbb{R}_{\geq 0})$$
$$\forall x{:}\mathbb{R}.\exists! y{:}\mathbb{R}_{\geq 0}.sq(x,y)$$

Note that we have restricted the domain of y into $\mathbb{R}_{\geq 0}$ rather than $\mathbb{R}$, which helps avoid the introduction of local variables in some cases as explained later.

The typed version of an *exists unique* property is expressed by the following two equations.

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\forall \mathbf{x}.[A(\mathbf{ux}) \rightarrow \vee_{i\in I}\mathbf{x} \in \boldsymbol{\theta}_i] \quad (4)$$
$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\ \wedge_{i\in I}\ \exists! \mathbf{x}_i{:}\boldsymbol{\theta}_i.A(\mathbf{ux}_i) \quad (5)$$

Domains in (1) and (2) are replaced by types in (4) and (5). Each $\boldsymbol{\theta}_i$ is called a *solution subtype* of the output parameter $\mathbf{x}$. Furthermore, input parameters are typed, which uses a type expression to express the condition under which a specific property holds. The typed version of an *exists sometimes* property is expressed by (4) and the following equation.

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\ \wedge_{i\in I}\ \exists ? \mathbf{x}_i{:}\boldsymbol{\theta}_i.A(\mathbf{ux}_i) \quad (6)$$

An example of typed *exists sometimes* properties can be found in example 10.

A typed *exists* property $\forall \mathbf{u}{:}\boldsymbol{\sigma}.\exists \mathbf{x}{:}\boldsymbol{\theta}.A(\mathbf{ux})$ states that for every $\mathbf{u}$ of type $\boldsymbol{\sigma}$ there are some $\mathbf{x}$ of type $\boldsymbol{\theta}$ such that $A(\mathbf{ux})$ holds. For instance, the *append*/3

[2] The type of a variable in a conjunctive formula follows just the first occurrence of the variable.

program satisfies $\forall z{:}\mathsf{list}(\beta).\exists x{:}\mathsf{list}(\beta).y{:}\mathsf{list}(\beta).append(x,y,z)$ which states that every list z can be split into two lists x and y.

A typed miscellaneous property $\forall \mathbf{u} : \boldsymbol{\sigma}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ states that, for every $\mathbf{u}$ of type $\boldsymbol{\sigma}$, $\neg A(\mathbf{u})$ can be replaced by $B(\mathbf{u})$. For instance, we have $\forall x{:}\mathbb{Z}.y{:}\mathbb{Z}.(\neg(x < y) \leftrightarrow (x \geq y))$.

As types and subtypes are associated with variables, the constraints that express sub-domains can be eliminated from the right-hand sides of rewrite rules for *existence* properties. So, more concise residual formulae can be obtained by making use of typed *existence* properties. As each variable is now associated with a type, it is necessary to make sure that a local variable is of the right type as an output argument in order to avoiding incorrect rewriting. This is guaranteed by requiring that if the output value lies in a solution subtype of an output parameter then the type of the local variable in the corresponding argument position is a supertype of that solution subtype.

Example 8. The following is an *exists unique* property in the domain of integers.

$$\forall x{:}\mathbb{Z}.y{:}\mathbb{Z}.\forall z.(add(x,y,z) \rightarrow z \in \mathbb{Z})$$
$$\forall x{:}\mathbb{Z}.y{:}\mathbb{Z}.\exists! z{:}\mathbb{Z}.add(x,y,z)$$

It states that, for any integers x and y, there is a unique integer z such that $add(x,y,z)$ is true. It would be wrong to use the *exists unique* property to rewrite $\neg\ \exists z{:}\mathbb{Z}_{[-\infty,10]}.(add(10,y{:}\mathbb{Z},z) \wedge b(z))$ into $add(10,y{:}\mathbb{Z},z{:}\mathbb{Z}_{[-\infty,10]}) \wedge \neg\ b(z)$. This is because z can take any value in $\mathbb{Z}$ and $\mathbb{Z}_{[-\infty,10]}$ is not a supertype of $\mathbb{Z}$.

The number of solutions to be negated is limited by the number of solution subtypes of the output parameter. Some solution subtypes are not relevant for a particular negative goal. A solution subtype is relevant if and only if it intersects with the type of the local variable in the negative goal. We call an index a *relevant index* if its corresponding solution subtype is relevant. We only need to consider relevant solution subtypes when rewriting the negative goal.

Example 9. Let the negative goal to rewrite be the following.

$$\neg\exists x{:}\mathbb{R}_{\geq 0}.(sq(x,y{:}\mathbb{R}_{>0}) \wedge b(x))$$

From example 6, $sq(x,y{:}\mathbb{R}_{>0})$ has two solutions for x, one of them is in $\mathbb{R}_{<0}$ and the other is in $\mathbb{R}_{>0}$. This suggests that there are two solutions to be negated. But, the type $\mathbb{R}_{\geq 0}$ of the local variable x doesn't intersect with $\mathbb{R}_{<0}$, that is, only solution subtype $\mathbb{R}_{>0}$ is relevant for the negative goal. The negative goal is rewritten to

$$sq(x_1{:}\mathbb{R}_{>0},y{:}\mathbb{R}_{>0}) \wedge \neg\ b(x_1)$$

since the type $\mathbb{R}_{\geq 0}$ of x is a supertype of the relevant solution subtype $\mathbb{R}_{>0}$.

Let $iarg(\tilde{\mathbf{u}},\boldsymbol{\sigma},\tilde{\mathbf{z}}) \stackrel{def}{=} \tilde{\mathbf{u}} \in \boldsymbol{\sigma} \wedge \mathbf{V}_{\tilde{\mathbf{u}}} \cap \tilde{\mathbf{z}} = \emptyset$ where $\tilde{\mathbf{z}}$ is a set of variables which are local when $iarg(\tilde{\mathbf{u}},\boldsymbol{\sigma},\tilde{\mathbf{z}})$ is invoked. The above considerations lead to the following typed rewrite rules for *exists unique* and *exists sometimes* properties.

QT

Given (4), (5) and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, (\mathbf{x} \cup \mathbf{y})) \wedge \forall i \in J.(\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta})$
where $J = \{i \in I \mid (\boldsymbol{\eta} \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$

$$\neg\exists \mathbf{x}{:}\boldsymbol{\eta}\mathbf{y}{:}\boldsymbol{\tau}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q] \leftrightarrow \wedge_{j\in J}[A(\tilde{\mathbf{u}}\mathbf{x}_j{:}\boldsymbol{\theta}_j) \wedge \neg(\exists \mathbf{y}{:}\boldsymbol{\tau}.Q[\mathbf{x} := \mathbf{x}_j])]$$

ST

Given (4), (6) and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, (\mathbf{x} \cup \mathbf{y})) \wedge \forall i \in J.(\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta})$
where $J = \{i \in I \mid (\boldsymbol{\eta} \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$

$$\neg\exists \mathbf{x}{:}\boldsymbol{\eta}\mathbf{y}{:}\boldsymbol{\tau}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge Q] \leftrightarrow \wedge_{j\in J} \left[\begin{array}{c} \neg\exists \mathbf{x}_j{:}\boldsymbol{\theta}_j.A(\tilde{\mathbf{u}}\mathbf{x}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\mathbf{x}_j{:}\boldsymbol{\theta}_j) \wedge \neg(\exists \mathbf{y}{:}\boldsymbol{\tau}.Q[\mathbf{x} := \mathbf{x}_j]) \end{array}\right]$$

The condition $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, (\mathbf{x} \cup \mathbf{y}))$ in the above two rewrite rules ensures that an input argument is of the type of the corresponding input parameter and it doesn't contain any local variables. The rewrite rules only generate subformulae for relevant solution subtypes which are collected by $J = \{i \in I \mid (\boldsymbol{\eta} \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$. For these relevant solution subtypes, the type of a local variable needs be a supertype of the type of the corresponding output parameter. This is guaranteed by $\forall i \in J.(\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta})$. Note that both $(\boldsymbol{\eta} \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}$ and $\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta}$ can be effectively tested.

The following rewrite rule makes uses of typed *exists* properties. It verifies that an input argument is of the type of the corresponding input parameter and that the type of an output argument is a supertype of the type of the corresponding output parameter.

ET

Given $\forall \mathbf{u}{:}\boldsymbol{\sigma}.\exists \mathbf{x}{:}\boldsymbol{\theta}.A(\mathbf{ux})$ and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \mathbf{x}) \wedge \boldsymbol{\theta} \sqsubseteq \boldsymbol{\eta}$

$$\neg\exists \mathbf{x}{:}\boldsymbol{\eta}.A(\tilde{\mathbf{u}}\mathbf{x}) \leftrightarrow \mathsf{false}$$

The following miscellaneous rewrite rule verifies that an input argument is of the type of the corresponding input parameter.

RT

Given $\forall \mathbf{u}{:}\boldsymbol{\sigma}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ and $\tilde{\mathbf{u}} \in \boldsymbol{\sigma}$

$$\neg A(\tilde{\mathbf{u}}) \leftrightarrow B(\tilde{\mathbf{u}})$$

3.3 Variable Introduction

The (QT) and (ST) rewrite rules require that input arguments are terms that do not contain any local variables and that are of type $\boldsymbol{\sigma}$. They also require that output arguments are local variables of type $\boldsymbol{\eta}$ such that $\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta}$ for each relevant solution subtype $\boldsymbol{\theta}_i$. If the requirement on output arguments is not met then it is possible to introduce new local variables so these rewrite rules can be applied.

Let $\neg\exists \mathbf{l} : \boldsymbol{\phi}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q]$ be the negative goal. It can be rewritten to $\neg\exists \mathbf{l} : \boldsymbol{\phi}.\mathbf{x}.[A(\tilde{\mathbf{u}}\mathbf{x}) \wedge (\tilde{\mathbf{x}} = \mathbf{x}) \wedge Q]$ where $\mathbf{x}$ be a vector of new local variables each for an element of $\tilde{\mathbf{x}}$. Recall that a bold letter denotes a vector of different variables and a bold letter with a tilde denotes a vector of terms that are not necessarily different.

Using (4) and (5), we can rewrite the above negative goal to $\wedge_{j\in J}[A(\tilde{\mathbf{u}}\mathbf{x}_j : \boldsymbol{\theta}_j) \wedge \neg\exists\mathbf{l}{:}\boldsymbol{\phi}.((\tilde{\mathbf{x}} = \mathbf{x}_j) \wedge Q)]$ with J being the set of relevant indices provided that $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \mathbf{l})$ is true. Let $\mathbf{w}{:}\boldsymbol{\zeta}$ be the vector of those elements of $\mathbf{l}$ that occur in $\tilde{\mathbf{x}}$ and $\mathbf{y}{:}\boldsymbol{\tau}$ be the vector of those elements of $\mathbf{l}$ that do not occur in $\tilde{\mathbf{x}}$. Then $\neg\exists\mathbf{l}{:}\boldsymbol{\phi}.((\tilde{\mathbf{x}} = \mathbf{x}_j) \wedge Q)$ for each $j \in J$ is equivalent to

$$\neg\exists\mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j) \vee (\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j) \wedge \neg\exists\mathbf{y}{:}\boldsymbol{\tau}.Q[\mathbf{w} := \mathbf{w}_j]$$

So, we have the following temporary rewrite rule for *exists unique* properties.

* Given (4), (5) and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \mathbf{l})$

$$\neg\exists\mathbf{l}{:}\boldsymbol{\phi}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow$$
$$\wedge_{j\in J}\begin{pmatrix} A(\tilde{\mathbf{u}}\mathbf{x}_j{:}\boldsymbol{\theta}_j) \wedge \neg\exists\mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\mathbf{x}_j{:}\boldsymbol{\theta}_j) \wedge (\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j) \wedge \neg\exists\mathbf{y}{:}\boldsymbol{\tau}.Q[\mathbf{w} := \mathbf{w}_j] \end{pmatrix}$$

where $J = \{i \in I \mid (typeof(\tilde{\mathbf{x}})$ and $\boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$

$\mathbf{w}{:}\boldsymbol{\zeta}$ *be the vector of those elements of* $\mathbf{l}$ *that occur in* $\tilde{\mathbf{x}}$

$\mathbf{y}{:}\boldsymbol{\tau}$ *be the vector of those elements of* $\mathbf{l}$ *that do not occur in* $\tilde{\mathbf{x}}$

where $typeof(E)$ denotes the type of E with E being a term or a vector of terms. As each variable in a term is typed, the type of the term is defined. In the above rewrite rule, $\mathbf{x}_j$ and $\mathbf{w}_j$ are variables that do not occur in the left-hand side of the rewrite rule. $\mathbf{x}_j$ is typed with j^{th} solution subtype while $\mathbf{w}_j$ inherits the type of $\mathbf{w}$. Disequality constraints $\neg\exists\mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j)$ can be dealt with by augmenting Chan's method with types.

A new local variable is introduced for each output argument of A in the above rewrite rule. As the cost of simplifying disequality constraint $\neg\exists\mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{x}}[\mathbf{w} := \mathbf{w}_j] = \mathbf{x}_j)$ increases with the number of equations in it, it is desirable to avoid introducing new local variables whenever possible. In order to avoid introducing a new local variable for an output argument, the output argument must be a local variable and if its type intersects with a solution subtype of its corresponding output parameter then its type must be a supertype of that solution subtype.

Let $\tilde{\mathbf{x}}$ be a vector of terms, $\tilde{\mathbf{y}}$ a sub-vector of $\tilde{\mathbf{x}}$ and $\boldsymbol{\sigma}$ a vector of types of the same length as $\tilde{\mathbf{x}}$. $\boldsymbol{\sigma}^{\tilde{\mathbf{y}}\lhd\tilde{\mathbf{x}}}$ denotes the vector of elements in $\boldsymbol{\sigma}$ that correspond in position to elements of $\tilde{\mathbf{y}}$ in $\tilde{\mathbf{x}}$. The above considerations lead to the following rewrite rule for *exists unique* properties.

QVT Given (4), (5) and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \mathbf{l})$

$$\neg\exists\mathbf{l}{:}\boldsymbol{\phi}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow$$
$$\wedge_{j\in J}\begin{pmatrix} A(\tilde{\mathbf{u}}\mathbf{z}_j{:}\boldsymbol{\theta}_j{}^{\tilde{\mathbf{s}}\lhd\tilde{\mathbf{x}}}\mathbf{r}_j{:}\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}}) \wedge \neg\exists\mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{s}}[\mathbf{r}\mathbf{w} := \mathbf{r}_j\mathbf{w}_j] = \mathbf{z}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\mathbf{z}_j{:}\boldsymbol{\theta}_j{}^{\tilde{\mathbf{s}}\lhd\tilde{\mathbf{x}}}\mathbf{r}_j{:}\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}}) \wedge (\tilde{\mathbf{s}}[\mathbf{r}\mathbf{w} := \mathbf{r}_j\mathbf{w}_j] = \mathbf{z}_j) \wedge \neg\exists\mathbf{y}{:}\boldsymbol{\tau}.Q[\mathbf{r}\mathbf{w} := \mathbf{r}_j\mathbf{w}_j] \end{pmatrix}$$

where $J = \{i \in I \mid (typeof(\tilde{\mathbf{x}})$ and $\boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$

$\mathbf{r}{:}\boldsymbol{\gamma}$ *be a subvector of* $\tilde{\mathbf{x}}$ *such that* $\forall j \in J.(\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}} \sqsubseteq \boldsymbol{\gamma})$

$\tilde{\mathbf{s}}$ *be the vector of the elements of* $\tilde{\mathbf{x}}$ *other than those in* $\mathbf{r}$

$\mathbf{w}{:}\boldsymbol{\zeta}$ *be the vector of those elements of* $\mathbf{l}$ *that occur in* $\tilde{\mathbf{s}}$

$\mathbf{y}{:}\boldsymbol{\tau}$ *be the vector of those elements of* $\mathbf{l}$ *that do not occur in* $\tilde{\mathbf{x}}$

The choice of $\mathbf{r}{:}\boldsymbol{\gamma}$ in the (QVT) rewrite rule is nondeterministic so long as it is subvector of $\tilde{\mathbf{x}}$ and it satisfies $\forall j \in J.(\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}} \sqsubseteq \boldsymbol{\gamma})$. In the extreme case, $\mathbf{r}{:}\boldsymbol{\gamma}$ is of zero length and the (QVT) rewrite rule degenerates to the (*) rewrite rule.

The same considerations as in the case for *exists unique* properties lead to the following rewrite rule for *exists sometimes* properties.

SVT

Given (4), (6) and $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \mathbf{l})$

$$\neg\exists \mathbf{l}{:}\boldsymbol{\phi}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow$$

$$\wedge_{j\in J} \left(\begin{array}{c} \neg\exists \mathbf{z}_j{:}\boldsymbol{\theta}_j{}^{\tilde{\mathbf{s}}\lhd\tilde{\mathbf{x}}}\mathbf{r}_j{:}\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}}.A(\tilde{\mathbf{u}}\mathbf{z}_j\mathbf{r}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\mathbf{z}_j{:}\boldsymbol{\theta}_j{}^{\tilde{\mathbf{s}}\lhd\tilde{\mathbf{x}}}\mathbf{r}_j{:}\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}}) \wedge \neg\exists \mathbf{w}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{s}}[\mathbf{rw} := \mathbf{r}_j\mathbf{w}_j] = \mathbf{z}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\mathbf{z}_j{:}\boldsymbol{\theta}_j{}^{\tilde{\mathbf{s}}\lhd\tilde{\mathbf{x}}}\mathbf{r}_j{:}\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}}) \wedge (\tilde{\mathbf{s}}[\mathbf{rw} := \mathbf{r}_j\mathbf{w}_j] = \mathbf{z}_j) \wedge \neg\exists{:}\boldsymbol{\tau}.Q[\mathbf{rw} := \mathbf{r}_j\mathbf{w}_j] \end{array} \right)$$

where $J = \{i \in I \mid (typeof(\tilde{\mathbf{x}}) \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$

$\mathbf{r}{:}\boldsymbol{\gamma}$ *be a subvector of* $\tilde{\mathbf{x}}$ *such that* $\forall j \in J.(\boldsymbol{\theta}_j{}^{\mathbf{r}\lhd\tilde{\mathbf{x}}} \sqsubseteq \boldsymbol{\gamma})$

$\tilde{\mathbf{s}}$ *be the vector of the elements of* $\tilde{\mathbf{x}}$ *other than those in* $\mathbf{r}$

$\mathbf{w}{:}\boldsymbol{\zeta}$ *be the vector of those elements of* $\mathbf{l}$ *that occur in* $\tilde{\mathbf{s}}$

$\mathbf{y}{:}\boldsymbol{\tau}$ *be the vector of those elements of* $\mathbf{l}$ *that do not occur in* $\tilde{\mathbf{x}}$

Example 10. The following is the typed version of the *exists sometimes* property in example 5.

$$\forall y{:}\mathbb{Z}_{>0}.\forall x.(sq(x,y) \rightarrow x \in \mathbb{Z}_{<0} \vee x \in \mathbb{Z}_{>0})$$
$$\forall y{:}\mathbb{Z}_{>0}.(\exists ?x_1{:}\mathbb{Z}_{<0}.sq(x_1,y) \wedge \exists ?x_2{:}\mathbb{Z}_{>0}.sq(x_2,y))$$

The local variable x in the negative goal $\neg\exists x{:}\mathbb{Z}_{[0,20]}.(sq(x, y{:}\mathbb{Z}_{>0}) \wedge b(x))$ has a type $\mathbb{Z}_{[0,20]}$ which is not a supertype of the sole relevant solution subtype $\mathbb{Z}_{>0}$ of the corresponding output parameter. Therefore, a new variable z_2 of type $\mathbb{Z}_{>0}$ is introduced and the negative goal is rewritten to the following.

$$\begin{array}{l} \neg\exists z_2{:}\mathbb{Z}_{>0}.sq(z_2, y{:}\mathbb{Z}_{>0}) \\ \vee\ sq(z_2{:}\mathbb{Z}_{>0}, y{:}\mathbb{Z}_{>0}) \wedge\ \neg\exists x{:}\mathbb{Z}_{[0,20]}.(x = z_2) \\ \vee\ sq(z_2{:}\mathbb{Z}_{>0}, y{:}\mathbb{Z}_{>0}) \wedge (x{:}\mathbb{Z}_{[0,20]} = z_2) \wedge \neg\ b(z_2) \end{array}$$

Solving the typed equality and the typed disequality constraints rewrites the above formula to the following.

$$\begin{array}{l} \neg\exists z_2{:}\mathbb{Z}_{>0}.sq(z_2, y{:}\mathbb{Z}_{>0}) \\ \vee\ sq(z_2{:}\mathbb{Z}_{[21,\infty]}, y{:}\mathbb{Z}_{>0}) \\ \vee\ sq(z_2{:}\mathbb{Z}_{[1,20]}, y{:}\mathbb{Z}_{>0}) \wedge \neg\ b(z_2) \end{array}$$

There is no rewrite rule with introduction of local variables for *exists* properties because introducing local variables won't lead to simplification. Introduction of local variables is irrelevant to the miscellaneous rewrite rule as miscellaneous properties have no output parameters.

The above rewrite rules can be used to extract positive information from negative goals.

Example 11. The *append*/3 program satisfies the following *exists unique* property.

$$\forall\beta.\left[\begin{array}{c}\forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta).\forall z.(append(x,y,z)\rightarrow z\in\mathsf{list}(\beta))\\ \wedge\\ \forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta)\exists!z{:}\mathsf{list}(\beta).append(x,y,z)\end{array}\right]$$

which satisfy (4) and (5). Using (QVT) allows

$$\neg\exists z{:}\mathsf{list}(\beta).(append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z), p(z))$$

to be rewritten as

$$append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z{:}\mathsf{list}(\beta)), \neg\ p(z)$$

Both Chan's method and Stuckey's first construct an SLD derivation tree of $append(x,y,z), p(z)$ and collect a frontier of the SLD derivation, say,

$$\{(x=[], y=z, p(z)), (x=[h|x'], y=y', z=[h|z'], append(x',y',z'), p(z))\}$$

Then the negation of this frontier is simplified and put into its disjunctive normal form. This gives rise to the following four conjunctive formulae.

(1) $x\neq[], \forall h,x'.(x\neq[h|x'])$
(2) $x\neq[], x=[h|x'], \neg\exists z'.(append(x',y,z'), p([h|z']))$
(3) $x=[], \forall h,x'.(x\neq[h|x']), \neg p(y)$
(4) $x=[], x=[h|x'], \neg p(y), \neg\exists z'.(append(x',y,z'), p([h|z']))$

Stuckey's method derives (2) and (3) because the constraint parts of (1) and (4) are unsatisfiable. Chan's method derives (1),(2) and (3) as it only tests satisfiability of atomic constraints. The constraint part of (4) is failed by unification in Chan's method as [] is not unifiable with $[h|x']$. Neither of these methods is effective as (2) is as complex as the original goal. The *exists unique* property allows us to obtain a simpler derived goal without making use of SLD derivation, and to eliminate unsatisfiable derived goals without satisfiability tests.

As well as rewriting the original goal, our rules can also be used for simplification. This will result in a more efficient simplification procedure since unsatisfiable goals are pruned without doing an explicit satisfiability test.

Example 12. We have $\forall y{:}\mathsf{all}.x.(x=s(y)\rightarrow x{:}\mathsf{all})$ and $\forall y{:}\mathsf{all}.\exists!x{:}\mathsf{all}.(x=s(y))$ in the Herbrand universe. Consider the following program.

```
p(y).
r(y) :- x=s(y),q(x).
```

The goal $p(y{:}\mathsf{all}), \neg\ r(y)$ is reduced to $p(y), \neg\exists x{:}\mathsf{all}.(x=s(y{:}\mathsf{all}), q(x))$ which is then simplified directly into $x{:}\mathsf{all}=s(y), p(y{:}\mathsf{all}), \neg\ q(x)$ using the above property. Without using this property, $\neg\exists x{:}\mathsf{all}.(x=s(y{:}\mathsf{all}), q(x))$ is simplified to

$$\forall x{:}\mathsf{all}.(x\neq s(y{:}\mathsf{all}))\vee(x{:}\mathsf{all}=s(y{:}\mathsf{all}), \neg\ q(x))$$

and a satisfiability test is then used to eliminate $\forall\ x{:}\mathsf{all}.(x\neq s(y{:}\mathsf{all}))$. In this sense, the satisfiability test is pushed into the simplification procedure by the *exists unique* property.

Chan's simplification rule can be formalised by a set of *exists sometimes* properties as follows.

$$\forall x{:}\mathsf{all}.y_1{:}\mathsf{all}\cdots y_n{:}\mathsf{all}.(x = s(y_1,\cdots,y_n) \rightarrow y_1 \in \mathsf{all} \wedge \cdots \wedge y_n \in \mathsf{all})$$
$$\forall x{:}\mathsf{all}.\exists ?y_1{:}\mathsf{all}\cdots y_n{:}\mathsf{all}.(x = s(y_1,\cdots,y_n))$$

These satisfy (4) and (6) and allow (SVT) to be applied.

The constructive negation method is sound as it rewrites a formula to another logically equivalent formula. The issue of completeness doesn't arise when the method is used as a simplification rule in other constructive methods [3,17,8]. Used stand-alone, the completeness of the method depends on the completeness of the available typed existence properties. It is part of our ongoing work to develop an algorithm for checking the completeness of the method with respect to a given program and a set of typed existence properties which can be either declared by the programmer or inferred from the program by static analysis [5].

4 Conclusion

The use of *existence* properties in [4] for the constructive negation of arithmetic goals has been generalised. This was done by generalising the notion of *existence* properties and by explicitly using types in generalised *existence* properties. This in turn allows negation of predicates that have multiple possible solutions. It also allows application of the rules to any user-defined predicates whose typed *existence* properties are known. The method need not do an SLD-derivation of the negated sub-goal in order to extract positive information. Because of this, it can be applied at compile time. It can also be used to simplify a frontier of an SLD-derivation tree. When used for simplification, it removes the need for an explicit satisfiability test. We have also integrated a powerful type system into the constructive negation method.

References

1. A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 329–340. The IEEE Computer Society Press, 1992. 415
2. D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 111–125. The MIT Press, 1988. 411
3. D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 477–496, 1989. 411, 413, 425
4. J.G. Cleary. Constructive Negation of Arithmetic Constraints Using Data-Flow Graphs. *Constraints*, 2: 131-162, 1997. 412, 413, 416, 425
5. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992. 425

6. P.W. Dart and J. Zobel. Efficient Run-Time Type Checking of Typed Logic Programs. *Journal of Logic Programming*, 14(1-2):31–69, 1992. 415, 416
7. W. Drabent. What Is Failure? An Approach to Constructive Negation. *Acta Informatica*, 32:27–59, 1995. 412, 413
8. F. Fages. Constructive Negation by Pruning. *Journal of Logic Programming*, 32(2):85–118, 1997. 412, 413, 425
9. N. Foo, A. Rao, A. Taylor, and A. Walker. Deduced Relevant Types and Constructive Negation. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 126–139. The MIT Press, 1988. 412
10. T. Fruhwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. The IEEE Computer Society Press, 1991. 414, 416
11. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, 1984. 416
12. M. Kifer and J. Wu. A First-Order Theory of Types and Polymorphism in Logic Programming. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 310–321. The IEEE Computer Society Press, 1991. 414
13. J. Małuszyński and T. Näslund. Fail Substitutions for Negation as Failure. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 461–476. The MIT Press, 1989. 412, 413
14. P. Mishra. Towards a Theory of Types in Prolog. In *Proceedings of the IEEE international Symposium on Logic Programming*, pages 289–298. The IEEE Computer Society Press, 1984. 416
15. A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984. 416
16. Frank Pfenning, editor. *Types in Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992. 414
17. P.J. Stuckey. Negation and Constraint Logic Programming. *Information and Computation*, 118:12–33, 1995. 412, 413, 425
18. J. Tiuryn. Type Inference Problems: A Survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120. Springer-Verlag, 1990. 414
19. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10(2):125–153, 1991. 416

Optimal Reduction of Functional Expressions

Andrea Asperti

Dipartimento di Scienze dell'Informazione
Via di Mura Anteo Zamboni 7, Bologna, Italy
asperti@cs.unibo.it

Abstract. The usual implementations of functional programming languages based on a weak evaluation paradigm (no reduction inside a lambda), betray the very spirit, i.e. the higher-order nature, of lambda-calculus.

1 Introduction

Consider the term

$$M = (n\ two\ I\ I)$$

where n and *two* are church integers, and I is the identity. Obviously, M reduces to I and, in this case, innermost reduction gives you the normal form in a number of steps which is linear in n. On the other side, every reduction strategy based on a weak evaluation paradigm (either eager or lazy) requires *exponential* time (check it as an exercise).

As it is clear by the previous example, there is a clear, dramatic problem with *sharing* in the usual implementations. The usual point of functional programming hackers is that situations like the one depicted above never occur in practice. This essentially amounts to say that *functions* are never used as real first class citizens; in particular, they are never used as real computational data. This obviously contradicts the very spirit of λ-calculus, where every datum is eventually coded as a function, i.e. by control information.

This talk will be about *optimal reduction* in functional programming languages, that exactly provides an *optimal* implementation of *sharing*. Optimal reduction joins the benefit of *every* possible reduction strategy (even considering "strategies" in the most liberal acception, i.e. not confined to the restrictive universe of λ-terms).

In particular, the formal notion of sharing we shall deal with was formalized in the seventies by Lévy in terms of "families" of redexes with a same origin—more technically, in terms of sets of redexes that are residuals of a unique (virtual) redex. The goal of his foundational research was exactly to characterize what an *optimally efficient* reduction strategy for the λ-calculus would look like, even if the technology for its implementation was at the time lacking.

In particular, optimal and correct implementations were known for recursion schemas, but not ones where higher-order functions could be passed as first-class values. Actually, more than ten years elapsed between Lévy's definition of optimality and its first feasible implementation, by Lamping.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 427–428, 1998.

The complex problem, brilliantly solved by Lamping, is that having a unique representation for all the redexes in a same family requires the use of a *very sophisticated* sharing technique, where all the computational and expressive power of higher-order languages really shines. Note in particular that two redexes of a same family can nest one into the other (for instance, given $R = (\lambda x.M)\,N$, its subterms M and N may contain redexes in the family of R). Therefore, all the traditional implementation techniques for functional languages (mostly based on supercombinators, environments or continuations) fail in avoiding useless repetitions of work: no machinery in which the sharing is exploited at the (first order) level of subterms can be Lévy's optimal.

Lamping's breakthrough was a technique to share contexts, that is, to share terms with an unspecified part, say a hole. Each instance of a context may fill its holes in a distinct way. Hence, for any shared context, there are several access pointers to it and, for each hole, a set of possible choices for how to fill it—a choice for each instance of the context. Lamping's solution moved from the idea that some control operator should manage the matching between the instances of a context and the ways in which its holes may be filled. As a result, Lamping extended λ-graphs by two sharing nodes called fans: a "fan-in" node collecting the pointers to a context; a "fan-out" node collecting the ways in which a hole can be filled (the exit pointers from the context)). Assuming that each pointer collected by a fan-in or fan-out is associated to a named port, the correspondence between entering and exiting pointers is kept provided to have a way to pair fan-in's and fan-out's: the instance of the context corresponding to the pointer connected to a fan-in port with name a fills its holes with the subgraphs accessed through the ports with name a of the matching fan-out's.

In this talk we shall try to give a friendly introduction to the topic of optimal reduction and Lamping's algorithm, pointing out the main open issues for further research.

References

AG98. A. Asperti, S.Guerrini *The Optimal Implementation of Functional Programming Languages*. To appear in the *"Cambridge Tracts in Theoretical Computer Science"* Series, Cambridge University Press, 1998.

Embedding Multiset Constraints into a Lazy Functional Logic Language*

P. Arenas-Sánchez, F.J. López-Fraguas, and M. Rodríguez-Artalejo

Universidad Complutense de Madrid, Fac. de CC. Matemáticas
Departamento de Sistemas Informáticos y Programación
Av. Complutense s/n, 28040 Madrid, Spain
{puri,fraguas,mario}@sip.ucm.es

Abstract. In recent works, we have proposed a general framework for lazy functional logic programming with algebraic polymorphic types, i.e., parametric datatypes whose data constructors fulfill a given set of equational axioms. The aim of this paper is to investigate implementation techniques for an extended instance of this framework, namely, lazy functional logic programming with multisets and constraints. We consider a language (named Seta) which supports a polymorphic datatype $Mset(\alpha)$ along with specific constraints for multisets: strict equality (already present in the general framework), disequality, membership and non-membership. We describe a quite readable Prolog-based implementation which can be executed on top of any Prolog system that provides the ability to solve simple arithmetic constraints.

1 Introduction

The combination of different declarative paradigms (specially functional and logic programming) has been widely treated in the literature (see [12] for a survey). Many approaches to the integration of functional and logic programming take constructor-based conditional term rewriting systems as programs. In order to deal properly with non-strict functions, lazy functional logic languages use *strict equality*, which regards two expressions as equal iff they have the same constructor normal form. Moreover, *lazy narrowing* (a notion introduced in [20] and refined in [1,17]) is often chosen as the goal solving mechanism.

As shown in [10], classical equational logic does not supply an adequate semantics for lazy functional logic programming, and a suitably defined *constructor based rewriting logic* provides a more convenient semantic framework. In two recent papers [3,4]

We have extended the approach from [10] with algebraic polymorphic types, i.e., parametric datatypes whose data constructors are governed by a given set $\mathcal{C}$ of equational axioms.

The aim of the present paper is to investigate implementation techniques for an extended instance of the framework from [3,4]. More precisely, we consider

* This research has been partially supported by the the Spanish National Project TIC95-0433-C03-01 "CPD" and the Esprit BRA Working Group EP-22457 "CCLII".

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 429–445, 1998.

a language Seta[1] which provides the algebraic datatype $Mset(\alpha)$, the data constructors $\{\!\!\{\ \}\!\!\} :\rightarrow Mset(\alpha)$ (to build the empty multiset) and $\{\!\!\{\cdot|\cdot\}\!\!\} : (\alpha, Mset(\alpha)) \rightarrow Mset(\alpha)$ (to add an element to a multiset), required to fulfill the equational axiom $\{\!\!\{x, y|zs\}\!\!\} \approx \{\!\!\{y, x|zs\}\!\!\}$. Here, we have used $\{\!\!\{x, y|zs\}\!\!\}$ as abbreviation for $\{\!\!\{x|\{\!\!\{y|zs\}\!\!\}\}\!\!\}$. In the sequel we will continue using such notation.

As an extension w.r.t. [3,4] (where constraints were absent, with the exception of strict equations), Seta supports also disequality constraints for all datatypes, as well as membership and non-membership constraints for multisets. Some others extensions of logic programming, as e.g. [14,11,9,16], use (multi) set unification as a particular instance of unification modulo equations [15]; but no concern on programming with lazy functions is found in these proposals. Being able to combine lazy functions with multiset unification and constraints, Seta turns out to be a very expressive language for any kind of problem related to the general idea of multiset rewriting. This is known to be useful for different applications, including the Gamma programming model [5], action and change problems [19,11], and the formal definition of visual languages [18].

Borrowing ideas from several previous works, mainly [17,2], we have developed a Prolog-based implementation which can be easily understood as an executable specification of Seta's operational semantics. Our implementation includes many substantially new techniques, since goal solving in Seta requires a non-trivial combination of lazy narrowing and unification modulo the equational axioms for multisets. As in the case of AC1-unification (see [15]), strict equations involving only constructor terms may have several incomparable unifiers. Moreover, function symbols defined by rewrite rules give rise to additional difficulties. For instance, assuming the rewrite rules $ms \rightarrow \{\!\!\{A|ns\}\!\!\}$ and $ns \rightarrow \{\!\!\{B|ms\}\!\!\}$, we can see that matching ms with the multiset pattern $\{\!\!\{B|xs\}\!\!\}$ cannot be solved simply by reducing ms to *head normal form* (i.e., variable or outermost constructor), as in lazy functional languages. This shows that lazy evaluation is much harder to implement in the presence of non-free data constructors.

These and other similar considerations motivate the need of a clever representation of multisets, to be used by the Prolog implementation. In fact, we have chosen to represent a multiset by keeping information about finitely many expressions known to be elements, another expression of type multiset corresponding to the rest of elements, and some simple arithmetic constraints imposing a lower bound to the multiset's cardinality (and eventually supplying the exact value of that cardinality). Therefore, our implementation must be executed on top of some Prolog system which can handle arithmetic constraints. We have chosen Sicstus Prolog 3.3. [21], which provides a *solver for real constraints* developed by C. Holzbaur [13].

The rest of the paper is organized as follows: Sect. 2 describes the language Seta, including a simple but illustrative programming example, within the realm of *action and change problems*. Sect. 3 presents a Prolog-based implementation for Seta. Some topics for future research are pointed in the concluding Sect. 4.

[1] Seta is not an acronym, but simply the spanish word for *mushroom*.

2 The Language Seta

Our presentation of Seta follows the lines of [3], with suitable extensions to incorporate constraints. We assume a countable set $TVar$ of *type variables* α, β etc., and a countable ranked alphabet $TC = \bigcup_{n\geq 0} TC^n$ of *type constructors* which includes the multiset type constructor $Mset \in TC^1$. *Polymorphic types* $\tau, \tau' \in T_{TC}(TVar)$ are built as $\tau ::= \alpha | T(\tau_1, \ldots, \tau_n)$, where $T \in TC^n$, $\alpha \in TVar$ and $\tau_i \in T_{TC}(TVar)$, $1 \leq i \leq n$. The set of type variables occurring in τ is written $tvar(\tau)$. On top of a given TC, we define a *polymorphic signature* Σ as $\Sigma = \langle TC, DC, FS, CS \rangle$, where:

- DC is a set of type declarations $C : (\tau_1, \ldots, \tau_n) \to \tau_0$ for *data constructors*, with $\bigcup_{i=1}^{n} tvar(\tau_i) \subseteq tvar(\tau_0)$. Of course, since Seta handles multisets, the type declarations: $\{\!\{\ \}\!\} :\to Mset(\alpha)$ and $\{\!\{\cdot|\cdot\}\!\} : (\alpha, Mset(\alpha)) \to Mset(\alpha)$ must be contained in DC, where $\{\!\{\ \}\!\}$ represents the *empty multiset* and $\{\!\{\cdot|\cdot\}\!\}$ is the *multiset constructor*. The intended meaning of $\{\!\{x|zs\}\!\}$ is to add a new copy of the element x to the multiset zs. The multiset constructor is governed by the multiset *commutativity* equation: $\{\!\{x, y|zs\}\!\} \approx \{\!\{y, x|zs\}\!\}$, which states that the order of elements in a multiset is irrelevant. All the other data constructors are free.
- FS is a set of type declarations $f : (\tau_1, \ldots, \tau_n) \to \tau_0$ for *defined function symbols*.
- $CS \equiv \{==, /= : (\alpha, \alpha), \in, \notin : (\alpha, Mset(\alpha))\}$ is a set of type declarations for *atomic constraint symbols*, where $==$ and $/=$ stand for *strict equality* and *disequality* respectively, whereas $\in, \notin$, are specific constraint symbols for multisets, representing *membership* and *non-membership* respectively.

We require that Σ does not include multiple type declarations for the same symbol. The types given by declarations in $DC \cup FS$ are called *principal types*. We will write $h \in DC^n \cup FS^n$ to indicate the arity of a symbol according to its type declaration in Σ. The notation $\diamondsuit \in CS$ will indicate that $\diamondsuit \in \{==, /=, \in, \notin\}$.

Assuming another countable set $DVar$ of *data variables* x, y, etc., we build *expressions* $e, r, l \ldots \in Expr_\Sigma(DVar)$ as $e ::= x | h(e_1, \ldots, e_n)$, where $h \in DC^n \cup FS^n$, $e_i \in Expr_\Sigma(DVar)$, $1 \leq i \leq n$. *Data terms* $Term_\Sigma(DVar) \subseteq Expr_\Sigma(DVar)$ are built by using variables and data constructors only. In the sequel, we reserve t, s to denote data terms. An *atomic constraint* φ has the form $e \diamondsuit e'$, where $e, e' \in Expr_\Sigma(DVar)$ and $\diamondsuit \in CS$. We write $ACon_\Sigma(DVar)$ for the set of all atomic constraints.

An *environment* is defined as any set V of type-annotated data variables $x : \tau$, such that V does not include two different annotations for the same variable. The set $Expr^\tau_\Sigma(V)$ of all expressions that admit type τ w.r.t. V is defined in the usual way; see [3]. $Expr^\tau_\Sigma(V)$ has a subset $Term^\tau_\Sigma(V)$, that is defined in the natural way.

An atomic constraint $e \diamondsuit e'$, $\diamondsuit \in \{==, /=\}$, is *well-typed* w.r.t. an environment V iff $e, e' \in Expr^\tau_\Sigma(V)$, for some $\tau \in T_{TC}(TVar)$. If $\diamondsuit \in \{\in, \notin\}$, then $e \diamondsuit e'$ is *well-typed* iff $e \in Expr^\tau_\Sigma(V)$, and $e' \in Expr^{Mset(\tau)}_\Sigma(V)$, for some $\tau \in T_{TC}(TVar)$.

Program rules are constructor-based rewrite rules for defined functions. More precisely, assuming a principal type declaration $f : (\tau_1, \ldots, \tau_n) \to \tau \in FS$, a *defining rule* for f must have the form: $f(t_1, \ldots, t_n) \to r \Leftarrow \varphi_1, \ldots, \varphi_m$, where the left-hand side is *linear* (i.e., without multiple occurrences of variables), $t_i \in$

$Term_\Sigma(DVar)$, $1 \leq i \leq n$, $r \in Expr_\Sigma(DVar)$, all variables occurring in r occur also in the left-hand side, and $\varphi_j \in ACon_\Sigma(DVar)$, $1 \leq j \leq m$. Furthermore, such a defining rule is *well-typed* iff there is some environment V such that $t_i \in Term^{\tau_i}_\Sigma(V)$, $1 \leq i \leq n$, $r \in Expr^{\tau}_\Sigma(V)$ and for all φ_j, $1 \leq j \leq m$, φ_j is well-typed w.r.t. V.

Note that, similarly to [10,3,4], neither termination nor confluence are assumed for program rules. Therefore, defined functions can be non-deterministic. This feature supports some useful programming techniques, as discussed in [10].

Programs are pairs $\mathcal{P} = \langle \Sigma, \mathcal{R} \rangle$, where Σ is a polymorphic signature and $\mathcal{R}$ is a finite set of defining rules for defined functions symbols in Σ. We will say that a program $\mathcal{P}$ is *well-typed* iff all program rules in $\mathcal{R}$ are well-typed.

Goals G have the form $\varphi_1, \ldots, \varphi_m$, where $\varphi_j \in ACon_\Sigma(DVar)$, $1 \leq j \leq m$. Furthermore, G is *well-typed* iff there exists an environment V such that φ_j, $1 \leq j \leq m$, are well-typed w.r.t. V. In the following, we will implicitly assume that all expressions, programs and goals we use are well-typed.

Solutions for Seta goals must sometimes refer to multiset cardinalities. Therefore, terms and constraints occurring in a solution can include what we call *multiset variables*, of the form $ys : L$, where $ys \in DVar$ and L (standing for ys's *cardinality*) is either an integer number or a variable. In the last case, we speak of a *cardinal variable*. A *solution* S for a goal G is a triple $\langle SS, \phi, Con \rangle$, where:

- $SS \equiv \{x_1 = t_1, \ldots, x_n = t_n\}$ is a system of equations in *solved form*, i.e., each variable x_i occurs only once in SS and t_i are data terms, $1 \leq i \leq n$. The intended meaning of equations of the form $xs = ys : L \in SS$ is that xs represents a multiset of cardinality L.
- ϕ is a set of constraints in *solved form* $x/{=}t$ (with t different from x) or $t \notin xs$.
- Con is a set of arithmetic constraints referring to the cardinal variables L occurring in SS.

Note that SS represents an idempotent substitution of terms for data variables. As we will show in Section 3, Seta's Prolog-based implementation is designed in such a way that the substitution described by SS is computed by Prolog's unification, whereas the set Con of arithmetic constraints is computed by a constraint solver. Multiset variables are not allowed to appear in user given programs and goals, but they help to improve the conciseness and generality of solutions. For instance, given the goal $\{\!|\, x|xs \,|\!\}/{=}\{\!|\, A, B \,|\!\}$, Seta can compute the solution $\langle \{xs = ys : L\}, \emptyset, \{\mathtt{L} = \setminus = \mathtt{1}\} \rangle$ (expressing that $\{\!|\, x|xs \,|\!\}$ and $\{\!|\, A, B \,|\!\}$ have different cardinality), as well as other solutions, including e.g. $\langle \{xs = ys : 1\}, \{x/{=}B, B \notin ys : 1\}, \emptyset \rangle$.

Goal solving in Seta uses a combination of lazy narrowing and unification modulo the multiset *commutativity* equation $\{\!|\, x, y|zs \,|\!\} \approx \{\!|\, y, x|zs \,|\!\}$. Narrowing is needed to solve any kind of goal involving defined functions. It works by means of unification with the left-hand sides of program rules, which must be done modulo multiset commutativity. As in [10,4], goals of the form $e{==}e'$ are solved by narrowing e and e' to some data terms which are equivalent modulo multiset commutativity (which amounts to syntactical identity for terms involving no multisets). Disequality goals between expressions whose main constructor symbol is different from $\{\!|\, \cdot|\cdot \,|\!\}$ have the same treatment as in [2], i.e., they are solved by narrowing both expressions to a sufficient extent for detecting disagreement of constructor symbols at the same position. Disequality goals between expres-

sions of type multiset are solved by narrowing both expressions to a sufficient extent to detect either a discrepancy of cardinality, or equality of cardinalities and discrepancy in the multiplicity of some element. A more detailed discussion is given in Subsection 3.6. Membership and non-membership goals have the natural, expected treatment.

The expressive power of multisets in Seta can be used to tackle any kind of problem which is related to the widely applicable idea of *multiset rewriting*; see e.g. [5,19,10,18]. In particular, multiset rewriting is known to be useful for solving *action and change problems* declaratively, while avoiding the so-called *frame problem*. Several known approaches to this problem are surveyed in [19]. In particular, it is known that *action and change problems* can be modeled by means of *equational logic programs* [11], using a binary AC1 operation $\circ$ to represent *situations* as multisets of *facts* $\mathsf{fact}_1 \circ \ldots \circ \mathsf{fact}_n$, and a ternary predicate $\mathsf{execPlan}(\mathsf{initialSit}, \mathsf{plan}, \mathsf{finalSit})$ to model the transformation of an *initial situation* into a *final situation* by the execution of a *plan* (sequence of *actions*). In Seta we can follow the same idea quite naturally, while non-deterministic functions and constraints help to improve expressivity. Let us illustrate this by means of an action and change problem related to a fictitious case of *microorganism evolution*.

Example 1. Microorganisms correspond to facts in our example; they are described by an *identifier* (natural number) and a multiset of *genes* (natural numbers). Natural numbers and microorganisms are built by the data constructors *Zero*, S and O respectively, declared as:

$$Zero :\rightarrow Nat \qquad S : Nat \rightarrow Nat \qquad O : (Nat, Mset(Nat)) \rightarrow Micoorganism$$

Our *situations* are *populations of microorganisms*, built by the data constructor:

$$P : (Nat, Mset(Microorganism)) \rightarrow Population$$

In a population $P(i, xs)$, i represents the least natural number which is not used as identifier by the microorganisms in xs. Populations can evolve through *events* (which correspond to *actions* in action and change problems). We assume two kinds of events (which are members of a datatype *Event*) described by the data constructors: $Mut : Nat \rightarrow Event$ and $Cross : (Nat, Nat) \rightarrow Event$, where:

- $Mut(i)$ represents a mutation of the microorganism with identifier i. A microorganism can *mutate* by duplicating one of its genes provided that there was a single occurrence of that gene. The new microorganism replaces the old one.
- $Cross(i, j)$ represents a cross of the microorganisms with identifiers i, j. Two microorganisms can *cross* by exchanging one gene, provided that they have the same number of different genes (ignoring duplications). Two new microorganisms do arise.

The performance of a given event causes a non-deterministic transformation of a given population. We express such a transformation by means of the following non-deterministic function:

$$\begin{array}{l}\mathsf{perf\_event} : (Event, Population) \rightarrow Population \\ \quad perf\_event(Mut(i), P(n, \{\!\{ O(j, \{\!\{ x|xs \}\!\})|oths \}\!\})) \rightarrow P(n, \{\!\{ O(i, \{\!\{ x, x|xs \}\!\})|oths \}\!\}) \\ \qquad\qquad \Leftarrow i{=}{=}j, x \notin xs \\ \quad perf\_event(Cross(i, j), P(n, \{\!\{ O(i_1, \{\!\{ x|xs \}\!\}), O(j_1, \{\!\{ y|ys \}\!\})|oths \}\!\})) \rightarrow \\ \qquad P(S(S(n)), \{\!\{ O(i, \{\!\{ x|xs \}\!\}), O(j, \{\!\{ y|ys \}\!\}), O(n, \{\!\{ x|ys \}\!\}), O(S(n), \{\!\{ y|xs \}\!\})|oths \}\!\}) \\ \qquad\qquad \Leftarrow i{=}{=}i_1, j{=}{=}j_1, size(\{\!\{ x|xs \}\!\}){=}{=}size(\{\!\{ y|ys \}\!\})\end{array}$$

$\mathsf{size} : Mset(\alpha) \rightarrow Nat$
$size(\{\!\{\ \}\!\}) \rightarrow Zero$
$size(\{\!\{ x|xs \}\!\}) \rightarrow size(xs) \Leftarrow x \in xs$
$size(\{\!\{ x|xs \}\!\}) \rightarrow S(size(xs)) \Leftarrow x \notin xs$

Plans in action and change problems will be *lists of events* here, where lists are described by the data constructors: $[\,] :\rightarrow List(\alpha)$ and $[\cdot|\cdot] : (\alpha, List(\alpha)) \rightarrow List(\alpha)$. The performance of a plan also transforms a given population. For that, we use a non-deterministic function (which corresponds to the ternary predicate execPlan in the equational logic programming approach to planning) defined by the rules:

$\mathsf{perform\_plan} : (List(Event), Population) \rightarrow Population$
$perform\_plan([\,], pop) \rightarrow pop$
$perform\_plan([first|rest], pop) \rightarrow perform\_plan(rest, perf\_event(first, pop))$

Assume now that we are looking for a plan that transforms the population $P(2, \{\!\{ O(0, \{\!\{ 1 \}\!\}), O(1, \{\!\{ 2 \}\!\}) \}\!\})$ (initial situation in action and change problems), into a new population containing two microorganisms with genomes $\{\!\{ 1,1,1 \}\!\}$ and $\{\!\{ 2,2,2 \}\!\}$ (final situation in action and change problems), where 2 is an abbreviation of $S(S(Zero))$, etc. Then, we can solve the goal:

$$perform\_plan(plan, P(2, \{\!\{ O(0, \{\!\{ 1 \}\!\}), O(1, \{\!\{ 2 \}\!\}) \}\!\}) == P(n, microo),$$
$$O(i, \{\!\{ 1,1,1 \}\!\}) \in microo, O(j, \{\!\{ 2,2,2 \}\!\}) \in microo$$

One possible computed answer would be:

$$n = 6, i = 5, j = 4,$$
$$microo = \{\!\{ O(0, \{\!\{ 1,1 \}\!\}), O(1, \{\!\{ 2,2 \}\!\}), O(2, \{\!\{ 1,1,2 \}\!\}),$$
$$O(3, \{\!\{ 2,2,1 \}\!\}), O(4, \{\!\{ 2,2,2 \}\!\}), O(5, \{\!\{ 1,1,1 \}\!\}) \}\!\},$$
$$plan = [Mut(0), Mut(1), Cross(0, 1), Mut(2), Mut(3), Cross(2, 3)]$$

Of course, other solutions can also be computed. ∎

3 A Prolog-Based Implementation for Seta

Along this section we describe the essentials of a Prolog-based implementation which can be understood as an executable specification of Seta's operational semantics (following [3,4], with suitable extensions to deal with constraint solving). The resulting executable specification borrows ideas from [2] but with a substantial novelty: we have introduced unification modulo the multiset commutativity equation (predicate mutate in Subsect. 3.3) and multiset constraints (predicates $==$, $\in$, $/=$, $\notin$ in Subsects. 3.4, 3.5, 3.6, 3.7, respectively).

Those expressions whose principal type is $Mset(\tau)$, for some $\tau \in T_{TC}(TVar)$ (shortly, *multiset expressions* in what follows), will be translated into Prolog terms which keep information about cardinalities. More precisely, we will use some simple arithmetic constraints to represent this information. For this reason, our implementation must be executed on top of some Prolog system which provides a solver for arithmetic constraints (we have chosen Sicstus Prolog 3.3. [21]).

Unless otherwise specified, every time a piece of code depends on a symbol `c` (resp. f), we assume that C (resp. `f`) ranges over $DC^n - \{\{\!\{\ \}\!\}, \{\!\{\cdot|\cdot\}\!\}\}$ (resp. FS^n), $n \geq 0$. Abusing of notation, we use predicate names not allowed

in Sicstus, such as ==, /=, ∈, etc.., subscripts to write Prolog variables[2], and $\mathtt{f}(\overline{\mathtt{E}_\mathtt{n}})$, $\mathtt{c}(\overline{\mathtt{E}_\mathtt{m}})$ to denote $\mathtt{f}(\mathtt{E}_1,\ldots,\mathtt{E}_\mathtt{n})$ and $\mathtt{c}(\mathtt{E}_1,\ldots,\mathtt{E}_\mathtt{m})$, respectively.

3.1 Prolog Representation of Seta Expressions

As in [7,17,2] our implementation relies on a translation of Seta expressions, programs and goals into Prolog. The arithmetical constraints over cardinalities occurring in the translations of multiset expressions will play a crucial role for the early detection of failure and success, sometimes avoiding infinite computations. For example, consider the goal $\{\!|\, f \,|\!\}=\!=\{\!|\, f, f \,|\!\}$, where f is defined by the program rule $f \to f$. Since both multisets have different cardinalities, the goal should fail immediately without attempting the evaluation f, which would cause an infinite computation. Dually, the disequality goal $\{\!|\, f \,|\!\}/=\{\!|\, f, f \,|\!\}$ should succeed immediately.

Let $\mathcal{P} = \langle \Sigma, \mathcal{R} \rangle$ be some given well-typed Seta program. Any expression e occurring in the program will be translated into a pair $PT(e) = \langle \mathtt{te}, \mathtt{const} \rangle$, where $\mathtt{te}$ is a Prolog term and $\mathtt{const}$ is a set of arithmetic constraints. We can safely assume (see [8]) that the (statically computed) principal types of all the expressions occurring in the Seta program are available during the translation. Moreover, we can also assume that all the occurrences of one and the same Seta variable are given identical translations. For the sake of simplicity, this assumption is not formally reflected in the following recursive definition of $PT(e)$:

- $PT(x) = \langle \mathtt{X}, \emptyset \rangle$, for all $x \in DVar$ whose type is not of the form $Mset(\tau)$. $\mathtt{X}$ is a fresh Prolog variable.
- $PT(xs) = \langle \mathtt{Xs} : \mathtt{L}, \{\mathtt{L} >= \mathtt{0}\} \rangle$, for all $xs \in DVar$ whose type is of the form $Mset(\tau)$. $\mathtt{Xs}, \mathtt{L}$ are two fresh Prolog variables.
- $PT(\{\!|\ |\!\}) = \langle \mathtt{empty} : \mathtt{0}, \emptyset \rangle$.
- $PT(\{\!|\, e | e' \,|\!\}) = \langle \mathtt{ms}(\mathtt{te}, \mathtt{k} : \mathtt{L}_1) : \mathtt{L}, \mathtt{const} \cup \{\mathtt{L} = \mathtt{L}_1 + \mathtt{1}\} \rangle$, where $PT(e) = \langle \mathtt{te}, \mathtt{const}' \rangle$, $PT(e') = \langle \mathtt{k} : \mathtt{L}_1, \mathtt{const}'' \rangle$ and $\mathtt{const} = \mathtt{const}' \cup \mathtt{const}''$.
- $PT(C(e_1,\ldots,e_n)) = \langle \mathtt{c}(\mathtt{te}_1,\ldots,\mathtt{te}_\mathtt{n}), \bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{const}_\mathtt{i} \rangle$, for every $C \in DC^n$, $n \geq 0$, C different from the multiset constructors, where C is translated as a Prolog data constructor $\mathtt{c}$ and $PT(e_i) = \langle \mathtt{te}_\mathtt{i}, \mathtt{const}_\mathtt{i} \rangle$, $1 \leq i \leq n$.
- $PT(f(e_1,\ldots,e_n)) = \langle \mathtt{f}(\mathtt{te}_1,\ldots,\mathtt{te}_\mathtt{n}, \mathtt{R}, \mathtt{S}), \bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{const}_\mathtt{i} \rangle$, for all $f \in FS^n$, $n \geq 0$, such that the type of $f(e_1,\ldots,e_n)$ is not of the form $Mset(\tau)$, where $PT(e_i) = \langle \mathtt{te}_\mathtt{i}, \mathtt{const}_\mathtt{i} \rangle$, $1 \leq i \leq n$. The arguments $\mathtt{R}$ and $\mathtt{S}$ are two fresh Prolog variables used to implement *sharing* following a technique introduced in [7]. Argument $\mathtt{S}$ controls if the expression has already been evaluated to head normal form (i.e., to an expression which is either a variable or has a data constructor symbol at head). In such a case, the result of the evaluation is kept in $\mathtt{R}$.
  $PT(f(e_1,\ldots,e_n)) = \langle \mathtt{f}(\mathtt{te}_1,\ldots,\mathtt{te}_\mathtt{n}, \mathtt{R}, \mathtt{S}) : \mathtt{L}, \bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{const}_\mathtt{i} \cup \{\mathtt{L} >= \mathtt{0}\} \rangle$, $\mathtt{L}$ is a fresh Prolog variable and $f(e_1,\ldots,e_n)$ has type $Mset(\tau)$.

Let us show a simple example illustrating several translations of expressions.

Example 2. Suppose that we have $\mathtt{f} : Nat \to Mset(Nat) \in FS$, and also that $Zero :\!\to Nat$ and $S : Nat \to Nat$ belong to DC. Then:

[2] *Warning*: some occurrences of == in the code refer to the Prolog metapredicate that checks syntactic identity!

- $PT(\{\!|\, Zero \,|\!\}) = \langle \mathtt{ms(zero, empty : 0) : L_1}, \{\mathtt{L_1 = 0 + 1}\}\rangle$.
- $PT(\{\!|\, Zero|xs \,|\!\}) = \langle \mathtt{ms(zero, Xs : L_1) : L_2}, \{\mathtt{L_2 = L_1 + 1, L_1 >= 0}\}\rangle$.
- $PT(\{\!|\, x|f(S(x)) \,|\!\}) = \langle \mathtt{ms(X, f(s(X), R, S) : L_1) : L_2}, \{\mathtt{L_2 = L_1 + 1, L_1 >= 0}\}\rangle$.

Note that in the first item, the arithmetic constraint solver of Sicstus Prolog would bind immediately $\mathtt{L_1}$ to 1. More generally, our Prolog translation keeps the exact cardinality n for any multiset expression of the form $\{\!|\, e_1, \ldots, e_n \,|\!\}$. Otherwise, our representation keeps a lower bound of the cardinality. For instance, in the second and third items above, the arithmetic constraints at hand entail $\mathtt{L_2} \geq 1$. Notice also how in the third item both occurrences of the variable x have been translated into the same Prolog variable X. ∎

3.2 Constraint Store and Prolog Translation of Seta Goals

As commented in Section 2, a solution S for a goal G is given by a data substitution (δ), a set of Seta constraints in solved form (ϕ) and a set of arithmetic constraints over those cardinal variables occurring in S (*Con*). Along a computation, Prolog unification together with the arithmetic constraint solver will compute δ and *Con* respectively. In order to calculate ϕ, the implementation uses a *constraint store*, always accessible during a computation.

The Prolog structure of the constraint store is similar to that used in [6], i.e., a Prolog list of the form $[\mathtt{v_1}\#\mathtt{c_1}, \ldots, \mathtt{v_n}\#\mathtt{c_n}]$, where each so-called *constrained variable* $\mathtt{v_i}$, $1 \leq i \leq n$, is either a Prolog variable X or a Prolog term $\mathtt{X : L}$ (representing a *multiset variable*), where X is a Prolog variable and L is an integer number or a Prolog variable. Each $\mathtt{c_i}$, $1 \leq i \leq n$, is a Prolog list of the form $[\mathtt{c}^{\mathtt{i}}_{\notin}, \mathtt{c}^{\mathtt{i}}_{/=}]$, where $\mathtt{c}^{\mathtt{i}}_{\notin}$ and $\mathtt{c}^{\mathtt{i}}_{/=}$ are again Prolog lists of terms, representing the non-membership and disequality constraints associated to the constrained variable $\mathtt{v_i}$, respectively. For those constrained variables in the constraint store which are not multiset variables $\mathtt{c}^{\mathtt{i}}_{\notin}$ reduces to the empty list. As an example of a constraint store we have $[\mathtt{Xs_1 : L_1}\#[[\mathtt{c}, \mathtt{d}], [\mathtt{Xs_2 : L_2}]], \mathtt{Xs_2 : L_2}\#[[\mathtt{c}], [\mathtt{Xs_1 : L_1}]]]$, which represents the following sequence of constraints in solved form: $C \notin xs_1, D \notin xs_1, C \notin xs_2, xs_1 / = xs_2$.

The translation $PT(G)$ of a well-typed goal $G \equiv e_1 \diamondsuit e'_1, \ldots, e_n \diamondsuit e'_n$, where $\diamondsuit \in CS$, is defined as: $PT(G) = \langle (\mathtt{te_1} \diamondsuit \mathtt{te'_1}, \mathtt{te_2} \diamondsuit \mathtt{te'_2}, \ldots, \mathtt{te_n} \diamondsuit \mathtt{te'_n}), \mathtt{const}\rangle$, where $PT(e_i) = \langle \mathtt{te_i}, \mathtt{const_i}\rangle$, $PT(e'_i) = \langle \mathtt{te'_i}, \mathtt{const'_i}\rangle$, $1 \leq i \leq n$, and $\mathtt{const} = \bigcup_{\mathtt{i=1}}^{\mathtt{n}} \mathtt{const_i} \cup \mathtt{const'_i}$. Remark again here, that all occurrences of one and the same (multiset) variable x in G are translated into the same (multiset) Prolog variable.

Example 3. Consider the goal $G \equiv \{\!|\, x|xs \,|\!\} == \{\!|\, x, x|xs \,|\!\}$. Then $PT(G)$ will be equal to $\langle (\mathtt{ms(X, Xs : L) : L_1 == ms(X, ms(X, Xs : L) : L_2) : L_3}), \{\mathtt{L >= 0, L_1 = L + 1, L_2 = L + 1, L_3 = L_2 + 1}\}\rangle$.

Thanks to the fact that both occurrences of xs have been translated into the same Prolog multiset variable $\mathtt{Xs : L}$, the arithmetic constraints occurring in $PT(G)$ entail that $\mathtt{L_1}$ can not be equal to $\mathtt{L_3}$, i.e., G's failure can be detected very quickly. ∎

Once a goal G has been translated into $PT(G) = \langle \mathtt{C}, \mathtt{const}\rangle$, the following Prolog call is generated: $\mathtt{const, solve(C, [\,], FCS)}$, where const activates the arithmetic constraint solver, and predicate solve solves consecutively each one

of the constraints occurring in C. The last two arguments of the predicate solve correspond to the initial and final constraint store. Thus, at the beginning of the computation the initial constraint store is empty (i.e., [ ]), and, after the execution of solve, the final constraint store FCS will contain the constraints in solved form composing a solution for G. In general, all Prolog predicates used in our Seta implementation have two additional arguments (which always appear as the two last positions). The first one represents the initial constraint store, whereas the second one represents the final constraint store obtained after a successful computation. The Prolog code for predicate solve is the following:

$\texttt{solve}((\texttt{Const},\texttt{Rest}),\texttt{ICS},\texttt{FCS})$:- $\texttt{solve}(\texttt{Const},\texttt{ICS},\texttt{CS}_1),\texttt{solve}(\texttt{Rest},\texttt{CS}_1,\texttt{FCS}).$

$\texttt{solve}(\texttt{E}\diamondsuit\texttt{R},\texttt{ICS},\texttt{FCS})$:- $\diamondsuit(\texttt{E},\texttt{R},\texttt{ICS},\texttt{FCS})$. % for each $\diamondsuit \in \{\texttt{==}, \texttt{/=}, \in, \notin\}$

In the sequel we will make use of several Prolog predicates to handle the constraint store. Due to lack of space, we give only an informal explanation of their behaviour. The first predicate is $\texttt{extract}(\texttt{ICS},\texttt{V},\texttt{Cv},\texttt{FCS})$, that, given an initial constraint store ICS and a (multiset) variable V, returns in Cv the constraints associated to V in ICS, removing from ICS the element V#Cv and returning the resulting constraint store in FCS. If the (multiset) variable V does not appear in ICS as a constrained variable, then Cv is bound to [[ ], [ ]]. The second predicate that we will use is $\texttt{addconstraints}(\texttt{CS}_1,\texttt{CS}_2,\texttt{FCS})$, that, given two constraint stores $\texttt{CS}_1$ and $\texttt{CS}_2$, returns in FCS the merge of $\texttt{CS}_1$ and $\texttt{CS}_2$, i.e., for all (multiset) variable V occurring as constrained variable in $\texttt{CS}_1$ (with constraints $[\texttt{C}_1,\texttt{C}_2]$) and $\texttt{CS}_2$ (with constraints $[\texttt{C}'_1,\texttt{C}'_2]$), we keep in FCS the constrained variable $\texttt{V}\#[\texttt{C}''_1,\texttt{C}''_2]$, where $\texttt{C}''_\texttt{i}$, $\texttt{1} \leq \texttt{i} \leq \texttt{2}$, is the concatenation of $\texttt{C}_\texttt{i}$ and $\texttt{C}'_\texttt{i}$. Otherwise, if V occurs as constrained variable in $\texttt{CS}_1$ (with constraints C) but not in $\texttt{CS}_2$ or vice versa, then we keep in FCS the constrained variable V#C.

3.3 Head Normal Forms and Prolog Translation of Program Rules

An expression is in *head normal form* (shortly *hnf*) iff it is a variable or its outermost symbol is a data constructor. As in [17,2], we will calculate hnfs by applying program rules. In contrast to functional programming, a given expression may have several different hnfs, due to the occurrence of free variables (affected by narrowing) and/or non-deterministic function symbols.

Predicate $\texttt{hnf}(\texttt{E},\texttt{H},\texttt{ICS},\texttt{FCS})$ returns in H one of the possible head normal forms of the expression E. The code associated to this predicate is the following:

$\texttt{hnf}(\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{R},\texttt{S}),\texttt{H},\texttt{ICS},\texttt{FCS})$:- $\texttt{S} == \texttt{on}, !, \texttt{H} = \texttt{R}, \texttt{ICS} = \texttt{FCS}.$
$\texttt{hnf}(\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{R},\texttt{S}),\texttt{H},\texttt{ICS},\texttt{FCS})$:- $!, \#\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{H},\texttt{ICS},\texttt{FCS}), \texttt{S} = \texttt{on}, \texttt{R} = \texttt{H}.$
$\texttt{hnf}(\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{R},\texttt{S}) : \texttt{L},\texttt{H},\texttt{ICS},\texttt{FCS})$:- $\texttt{S} == \texttt{on}, !, \texttt{H} = \texttt{R}, \texttt{ICS} = \texttt{FCS}.$
$\texttt{hnf}(\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{R},\texttt{S}) : \texttt{L},\texttt{H},\texttt{ICS},\texttt{FCS})$:- $!, \#\texttt{f}(\overline{\texttt{E}}_\texttt{n},\texttt{H},\texttt{ICS},\texttt{FCS}), \texttt{S} = \texttt{on}, \texttt{R} = \texttt{H}.$
$\texttt{hnf}(\texttt{E},\texttt{H},\texttt{ICS},\texttt{FCS})$:- $\texttt{H} = \texttt{E}, \texttt{ICS} = \texttt{FCS}.$

Note that, the clauses of hnf distinguish function calls which are known to return a multiset (third and fourth clauses) from other function calls (first and second clauses). In both situations, given a function call whose hnf is not yet computed (second and fourth clauses), the hnf must be computed by using the program rules. Therefore, these must be translated into Prolog. More precisely,

for each program rule $f(t_1, \ldots, t_n) \to r \Leftarrow C$, we produce the following Prolog clause:

$$\#\mathtt{f}(\overline{\mathtt{E}}_\mathtt{n}, \mathtt{H}, \mathtt{CS}_0, \mathtt{CS}_{n+2}) \text{ :- } \mathtt{const}, \mathtt{unif}(\mathtt{E}_1, \mathtt{t}'_1, \mathtt{CS}_0, \mathtt{CS}_1), \ldots, \mathtt{unif}(\mathtt{E}_n, \mathtt{t}'_n, \mathtt{CS}_{n-1}, \mathtt{CS}_n), \mathtt{solve}(\mathtt{C}', \mathtt{CS}_n, \mathtt{CS}_{n+1}), \mathtt{hnf}(\mathtt{t}_r, \mathtt{H}, \mathtt{CS}_{n+1}, \mathtt{CS}_{n+2}).$$

where $PT(t_i) = \langle \mathtt{t}'_i, \mathtt{const}_i \rangle$, $1 \leq i \leq n$, $PT(r) = \langle \mathtt{t}_r, \mathtt{const}_{n+1} \rangle$, $PT(C) = \langle \mathtt{C}', \mathtt{const}_{n+2} \rangle$ and $\mathtt{const} = \bigcup_{i=1}^{n+2} \mathtt{const}_i$. Remark that $PT(C)$ is defined similarly to $PT(G)$, where G is a goal.

This translation of program rules implements a quite naïve narrowing strategy, rather than the more efficient strategy known as *needed narrowing* [1,17]. Actually, needed narrowing cannot be applied straightforwardly to a language with algebraic data constructors. The investigation of some suitable extension is left for future research.

Predicate $\mathtt{unif}(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{FCS})$ defined immediately below, unifies an expression E given as actual parameter for some function call with a linear term T coming from the left-hand side of some program rule. This unification will reduce E as much as needed to match the constructors occurring in T. Predicate $\mathtt{isvar}(\mathtt{T})$ invoked in the first clause of unif succeeds iff T is a Prolog variable or T is a multiset variable. The code for predicate unif is the following:

$$\begin{array}{l}
\mathtt{unif}(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{isvar}(\mathtt{T}), !, \mathtt{T} = \mathtt{E}, \mathtt{ICS} = \mathtt{FCS}. \\
\mathtt{unif}(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{hnf}(\mathtt{E}, \mathtt{H}, \mathtt{ICS}, \mathtt{CS}_1), \mathtt{unifhnf}(\mathtt{H}, \mathtt{T}, \mathtt{CS}_1, \mathtt{FCS}). \\
\\
\mathtt{unifhnf}(\mathtt{X}, \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{isvar}(\mathtt{X}), !, \mathtt{extract}(\mathtt{ICS}, \mathtt{X}, \mathtt{Cx}, \mathtt{CS}_1), \mathtt{X} = \mathtt{T}, \\
\quad \mathtt{propag}(\mathtt{Cx}, \mathtt{X}, \mathtt{CS}_1, \mathtt{FCS}). \\
\mathtt{unifhnf}(\mathtt{c}(\overline{\mathtt{E}}_\mathtt{n}), \mathtt{c}(\overline{\mathtt{T}}_\mathtt{n}), \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{unif}(\mathtt{E}_1, \mathtt{T}_1, \mathtt{ICS}, \mathtt{CS}_1), \ldots, \mathtt{unif}(\mathtt{E}_n, \mathtt{T}_n, \mathtt{CS}_{n-1}, \mathtt{FCS}). \\
\mathtt{unifhnf}(\mathtt{empty} : 0, \mathtt{empty} : 0, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{ICS} = \mathtt{FCS}. \\
\mathtt{unifhnf}(\mathtt{ms}(\mathtt{E}, \mathtt{Es}) : \mathtt{L}, \mathtt{ms}(\mathtt{R}, \mathtt{Rs}) : \mathtt{L}_1, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \{\mathtt{L} = \mathtt{L}_1\}, \\
\quad \mathtt{mutate}(\mathtt{ms}(\mathtt{E}, \mathtt{Es}) : \mathtt{L}, \mathtt{ms}(\mathtt{E}_1, \mathtt{Es}_1) : \mathtt{L}, \mathtt{ICS}, \mathtt{CS}_1), \\
\quad \mathtt{unif}(\mathtt{E}_1, \mathtt{R}, \mathtt{CS}_1, \mathtt{CS}_2), \mathtt{unif}(\mathtt{Es}_1, \mathtt{Rs}, \mathtt{CS}_2, \mathtt{FCS}). \\
\\
\mathtt{propag}([[\,], [\,]], \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \mathtt{ICS} = \mathtt{FCS}, !. \\
\mathtt{propag}([[\mathtt{E}|\mathtt{R}], [\,]], \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } !, \notin(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{CS}_1), \mathtt{propag}([\mathtt{R}, [\,]], \mathtt{T}, \mathtt{CS}_1, \mathtt{FCS}). \\
\mathtt{propag}([[\,], [\mathtt{E}|\mathtt{R}]], \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } !, /=(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{CS}_1), \mathtt{propag}([[\,], \mathtt{R}], \mathtt{T}, \mathtt{CS}_1, \mathtt{FCS}). \\
\mathtt{propag}([[\mathtt{E}|\mathtt{R}], [\mathtt{E}_1|\mathtt{R}_1]], \mathtt{T}, \mathtt{ICS}, \mathtt{FCS}) \text{ :- } \notin(\mathtt{E}, \mathtt{T}, \mathtt{ICS}, \mathtt{CS}_1), \\
\quad /=(\mathtt{E}_1, \mathtt{T}, \mathtt{CS}_1, \mathtt{CS}_2), \mathtt{propag}([\mathtt{R}, \mathtt{R}_1], \mathtt{T}, \mathtt{CS}_2, \mathtt{FCS}).
\end{array}$$

Some comments are needed in order to clarify the behaviour of unif. The first clause covers the case in which T is a (possibly multiset) variable, and unification succeeds immediately. Otherwise, E is reduced to hnf and the predicate unifhnf is invoked. In the first clause for unifhnf we can observe a new predicate propag/4, which must be invoked whenever a (possibly multiset) variable X is bound to a term T. The task of propag is to impose on T all the constraints associated to X in the current constraint store. This might lead to failure.

The last clause for predicate unifhnf takes care of unification modulo the multiset commutativity axiom $\{\!\{ x, y | zs \}\!\} \approx \{\!\{ y, x | zs \}\!\}$. As explained in the Introduction, this causes multiplicity of unifiers as well as complications related to the computation of hnfs. These matters are dealt with predicate mutate/4 defined below, which reorganizes the representation of a multiset in different ways, trying to bring all the elements at the head position. For instance, predicate mutate applied to the multiset $\{\!\{ e_1, e_2, e_3 \}\!\}$, where $e_i \in \mathit{Expr}_\Sigma(\mathit{DVar})$, $1 \leq i \leq 3$,

would generate the following three mutations: $\{\!\{ e_1, e_2, e_3 \}\!\}$, $\{\!\{ e_2, e_1, e_3 \}\!\}$ and $\{\!\{ e_3, e_1, e_2 \}\!\}$. For multiset expressions including function calls of type multiset, as e.g. $\{\!\{ e_1 | f(e) \}\!\}$, `mutate` may force further computations to hnf, as shown by its third clause. This is done in order to extract new elements to be brought at the head. The last clause for `mutate` also establishes constraints which update the lower bounds kept in multiset representations. Moreover, the first clause of predicate `mutate` calls to predicate `propag`, because the multiset variable `Xs : L` becomes bounded to `MX`. Predicate `mutate` is described by the following clauses:

```
mutate(Xs : L, MX, ICS, FCS) :- var(Xs), !, extract(ICS, Xs : L, Cx, CS1),
      Xs : L = MX, propag(Cx, Xs : L, CS1, FCS).
mutate(ms(E, Es) : L, Ms, ICS, FCS) :- Ms = ms(E, Es) : L, ICS = FCS.
mutate(ms(E, Es : L) : L1, ms(R, ms(E, Rs : L2) : L3) : L4, ICS, FCS) :-
    hnf(Es : L, HEs, ICS, CS1), mutate(HEs, ms(R, Rs : L2) : L3, CS1, FCS),
    {L2 >= 0, L = L2 + 1, L3 = L2 + 1, L4 = L3 + 1}.
```

3.4 Solving Strict Equations

Predicate == defined below solves strict equations $E{==}R$ between expressions. Its associated code is:

```
==(E, R, ICS, FCS) :- hnf(E, HE, ICS, CS1), hnf(R, HR, CS1, CS2),==hnf(HE, HR, CS2, FCS).
==hnf(X, T, ICS, FCS) :- isvar(X), !, ==var(X, T, ICS, FCS). % and symmetric case
==hnf(c(En), c(Rn), ICS, FCS) :- ==(E1, R1, ICS, CS1), ..., ==(En, Rn, CSn-1, FCS).
==hnf(empty : 0, empty : 0, ICS, FCS) :- ICS = FCS.
==hnf(ms(E, Es) : L, ms(R, Rs) : L1, ICS, FCS) :- {L = L1},
    mutate(ms(E, Es) : L, ms(E1, Es1) : L, ICS, CS1),
    ==(E1, R, CS1, CS2), ==(Es1, Rs, CS2, FCS).
==var(X, Y, ICS, FCS) :- isvar(Y), X==Y, !, ICS = FCS.
==var(X, Y, ICS, FCS) :- isvar(Y), !, extract(ICS, X, Cx, CS1),
    X = Y, propag(Cx, Y, CS1, FCS).
==var(X, c(En), ICS, FCS) :- !, occursnot(X, c(En)), extract(ICS, X, Cx, CS1),
    X = c(Vn), propag(Cx, X, CS1, CS2),
    ==(V1, E1, CS2, CS3), ..., ==(Vn, En, CSn+1, FCS). % Vn fresh variables
==var(X, empty : 0, ICS, FCS) :- extract(ICS, X, Cx, CS1),
    X = empty : 0, propag(Cx, X, CS1, FCS).
==var(X, ms(E, Es) : L, ICS, FCS) :- occursnot(X, ms(E, Es) : L),
    extract(ICS, X, Cx, CS1), X = ms(Y, Ys : L1) : L, {L = L1 + 1, L1 >= 0},
    propag(Cx, X, CS1, CS2), ==(Y, E, CS2, CS3),
    ==(Ys : L1, Es, CS3, FCS). % Y, Ys, L1 fresh variables
```

The last two clauses for predicate ==`hnf` refer to the resolution of equalities between multisets. Note that in the fourth clause, predicate `mutate` (defined in Subsect. 3.3) is used only once. To mutate both multisets would only generate redundant solutions. Note also the following: `mutate`, by itself, does not enumerate all the permutations of a given multiset representation; but the recursive use of `mutate` within the clauses for the predicate == guarantees that all the solutions are eventually computed.

A strict equation with one side being a (multiset) variable is solved by predicate ==`var`. In particular, when both sides of a strict equation are different

(possibly multiset) variables `X`, `Y`, the equation must succeed iff both variables are not constrained to be different. Furthermore, since both variables will be bound, only one of them must remain in the constraint store, containing simultaneously all constraints for `X` and `Y`. This task is realized by the second clause of predicate ==`var`. Note that in this clause, predicate `extract` is invoked in order to remove `X` from the constraint store, remaining variable `Y`. After binding `X` to `Y`, predicate `propag` will add all constraints associated to `X` to the constraints for `Y` in the constraint store, since all non-membership and disequality constraints to be solved are in solved form (see predicates $/=$ and $\notin$ in Subsects. 3.6 and 3.7, respectively).

In order to bind a (possibly multiset) variable `X` to an expression `E`, we must ensure that `X` does not occur in `E` at some position whose ancestor positions are all occupied by constructor symbols. This is checked by predicate `occursnot`/2 appearing in the third and last clause[3] of predicate ==`var`, whose Prolog code is not presented due to lack of space.

3.5 Solving Membership Constraints

The membership of an element `E` to a multiset `R` is implemented by means of the predicate $\in$ defined as follows:

$\in(\mathtt{E},\mathtt{R},\mathtt{ICS},\mathtt{FCS})$:- $\mathtt{hnf}(\mathtt{R},\mathtt{HR},\mathtt{ICS},\mathtt{CS}_1),\in\mathtt{hnf}(\mathtt{E},\mathtt{HR},\mathtt{CS}_1,\mathtt{FCS})$.

$\in\mathtt{hnf}(\mathtt{E},\mathtt{Xs}:\mathtt{L},\mathtt{ICS},\mathtt{FCS})$:- $\mathtt{var}(\mathtt{Xs}),!,\in\mathtt{var}(\mathtt{E},\mathtt{Xs}:\mathtt{L},\mathtt{ICS},\mathtt{FCS})$.
$\in\mathtt{hnf}(\mathtt{E},\mathtt{ms}(\mathtt{E}_1,\mathtt{Es}):\mathtt{L},\mathtt{ICS},\mathtt{FCS})$:- $==(\mathtt{E},\mathtt{E}_1,\mathtt{ICS},\mathtt{FCS})$.
$\in\mathtt{hnf}(\mathtt{E},\mathtt{ms}(\mathtt{E}_1,\mathtt{Es}):\mathtt{L},\mathtt{ICS},\mathtt{FCS})$:- $\in(\mathtt{E},\mathtt{Es},\mathtt{ICS},\mathtt{FCS})$.

$\in\mathtt{var}(\mathtt{E},\mathtt{Xs}:\mathtt{L},\mathtt{ICS},\mathtt{FCS})$:- $==(\mathtt{E},\mathtt{Y},\mathtt{ICS},\mathtt{CS}_1),\mathtt{extract}(\mathtt{CS}_1,\mathtt{Xs}:\mathtt{L},\mathtt{Cx},\mathtt{CS}_2)$,
$\mathtt{Xs}:\mathtt{L}=\mathtt{ms}(\mathtt{Y},\mathtt{Ys}:\mathtt{L}_1):\mathtt{L},\{\mathtt{L}_1>=0,\mathtt{L}=\mathtt{L}_1+1\},\mathtt{propag}(\mathtt{Cx},\mathtt{Xs}:\mathtt{L},\mathtt{CS}_2,\mathtt{FCS})$.
% $\mathtt{Y},\mathtt{Ys},\mathtt{L}_1$ `fresh variables`

Note that predicate $\in$ does not require to compute a hnf for `E` in advance, because a strict equation between `E` and some member of the multiset should eventually succeed. Moreover, solving membership constraints does not require to mutate the multiset, because the validity of this constraint is independent of the syntactic order of elements in a multiset representation.

The membership of an expression `E` to a multiset variable $\mathtt{Xs}:\mathtt{L}$ (predicate $\in$ `var` defined above) forces the evaluation of `E` to normal form by invoking to predicate == with its first argument `E` and the second one a fresh Prolog variable. After this, `Xs` is bound to a pattern representing an arbitrary multiset one of whose members is `Y` (which contains the normal form `E`), and appropriate constraints are imposed, as usual.

3.6 Solving Disequality Constraints

Disequality constraints are solved by predicate $/=$ defined below. Considering that disequality constraints involving free constructors are solved by techniques

[3] Of course, these two clauses could be optimizated as in [2] by using an incremental occurs-check.

borrowed from [2], here we limit ourselves to discuss disequalities between multisets. We want to implement the following notion of multiset disequality: Two multisets Ms and $\mathtt{Ms}_1$ verify $\mathtt{Ms} /= \mathtt{Ms}_1$ iff one of the following conditions holds:

(a) Ms and $\mathtt{Ms}_1$ have different cardinality;
(b) Ms and $\mathtt{Ms}_1$ have the same cardinality but one of the following items holds:
  (i) there exists $\mathtt{a} \in \mathtt{Ms}$ such that $\mathtt{a} \notin \mathtt{Ms}_1$ or vice versa;
  (ii) there exists $\mathtt{a} \in \mathtt{Ms} \cap \mathtt{Ms}_1$ such that the *multiplicity* of a in Ms (i.e., the number of occurrences of a as element of Ms) is different from the multiplicity of a in $\mathtt{Ms}_1$.

For implementing disequality according to this definition, the following case distinction turns out to be helpful: We say that a multiset expression is *closed* iff it is of the form $\{\!\!\{ e_1, \ldots, e_n \}\!\!\}$, *open* iff it is of the form $\{\!\!\{ e_1, \ldots, e_n | xs \}\!\!\}$ and *pending* iff it is of the form $\{\!\!\{ e_1, \ldots, e_n | f(e'_1, \ldots, e'_m) \}\!\!\}$, where $xs \in DVar$ and $f \in FS^m$. We will assume the existence of the predicate tail(Ms, Tail) (whose code is not presented due to lack of space) which returns in Tail which kind of multiset expression Ms is. More precisely: Tail is the atom "c" if Ms is closed, the atom "o" if Ms is open, or the term "p(n)" if Ms is pending with n elements before the tail. Let us now present the code for predicate /=:

```
/=(E,R,ICS,FCS) :- hnf(E,HE,ICS,CS1),hnf(R,HR,CS1,CS2),/=hnf(HE,HR,CS2,FCS).

/=hnf(X,T,ICS,FCS) :- isvar(X),!,/=var(X,T,ICS,FCS).  % and symmetric case

/=hnf(c(Ē_n),d(R̄_m),ICS,FCS) :- ICS = FCS.
  % C,D with the same principal type, up to variants

/=hnf(c(Ē_n),c(R̄_n),ICS,FCS) :- (/=(E1,R1,ICS,FCS);...;/=(En,Rn,ICS,FCS)).
/=hnf(Ms,Ms1,ICS,FCS):-tail(Ms,V),tail(Ms1,V1),/=hnftail(V,V1,Ms,Ms1,ICS,FCS).


/=hnftail(Id1,Id2,Ms:L,Ms1:L1,ICS,FCS) :- ({L = \ = L1},ICS = FCS ;
        {L = L1},/=hnfbasic(Ms:L,Ms1:L1,ICS,FCS)).   % Id1,Id2 ∈ {c,o}
/=hnftail(c,p(N),Ms:L,Ms1:L1,ICS,FCS) :- (\ + {L = L1},!,ICS = FCS ;
     getnewelement(Ms1:L1,Ms2,T,ICS,CS1),/=hnftail(c,T,Ms:L,Ms2,CS1,FCS)).
          % and symmetric case
/=hnftail(o,p(N),Ms:L,Ms1,ICS,FCS) :- ({L < N},ICS = FCS ;
     getnewelement(Ms1,Ms2,T,ICS,CS1),/=hnftail(o,T,Ms:L,Ms2,CS1,FCS)).
          % and symmetric case
/=hnftail(p(M),p(N),Ms,Ms1,ICS,FCS) :-
     getnewelement(Ms,Ms2,T,ICS,CS1),/=hnftail(p(N),T,Ms1,Ms2,CS1,FCS).

/=hnfbasic(Ms,Ms1,ICS,FCS) :-
     mutate(Ms,ms(E,Es):L,ICS,CS1),∉(E,Ms1,CS1,FCS).  % and symmetric case
/=hnfbasic(Ms,ms(E,Es):L,ICS,FCS) :-
     mutate(Ms,ms(E1,Es1):L1,ICS,CS1),==(E1,E,CS1,CS2),/=(Es1,Es,CS1,FCS).

/=var(X,Y,ICS,FCS) :- isvar(Y),!,X\ == Y,
        addconstraints([X#[[ ],[Y]],Y#[[ ],[X]]],ICS,FCS).
/=var(X,T,ICS,FCS) :- isterm(T),!,addconstraints([X#[[ ],[T]]],ICS,FCS).
/=var(X,c(Ē_n),ICS,FCS):-extract(ICS,X,Cx,CS1),X = d(Ū_m),propag(Cx,X,CS1,FCS).
   % C,D with the same principal type, Ū_m fresh Prolog variables
/=var(X,c(Ē_n),ICS,FCS) :- extract(ICS,X,Cx,CS1),X = c(Ū_n),
  propag(Cx,X,CS1,CS2),/=(X,c(Ē_n),CS2,FCS).   % Ū_n fresh Prolog variables
```

```
/=var(X, ms(E, Es) : L, ICS, FCS) :- extract(ICS, X, Cx, CS1), X = empty : 0
        propag(Cx, X, CS1, FCS).
/=var(X, ms(E, Es) : L, ICS, FCS) :- extract(ICS, X, Cx, CS1), X = ms(C, R : L1) : L2,
    {L2 = L1 + 1, L1 >= 0}, propag(Cx, X, CS1, CS2), /=(X, ms(E, Es) : L, CS2, FCS).
  % C, R, L1, L2 fresh Prolog variables
```

The fourth clause for /=`hnf` states that for solving a disequality between two multiset expressions in hnf `Ms` and $\texttt{Ms}_1$, their tails are firstly examined and then /=`hnftail` performs a case distinction depending on the results:

(1) If `Ms` and $\texttt{Ms}_1$ are not pending (the first clause for /=`hnftail`, which is a clause schema abbreviating the 4 clauses obtained by choosing values for Id_1, Id_2 among "c", "o"), then we open two alternatives which correspond to the cases (a) and (b) of the definition of /=.

(1.a) The cardinalities `L` and $\texttt{L}_1$ of `Ms` and $\texttt{Ms}_1$ are required to be different. In some cases the condition $\{\texttt{L} = \backslash = \texttt{L}_1\}$ behaves as a test of incompatibility of cardinals, as e.g. when $\texttt{Ms} \equiv \texttt{ms(a, empty:0):1}$ and $\texttt{Ms}_1 \equiv \texttt{ms(a, ms(b, Xs : L) : L}_1\texttt{) : L}_2$ along with $\{\texttt{L} >= 0, \texttt{L}_1 = \texttt{L} + 1, \texttt{L}_2 = \texttt{L}_1 + 1\}$. But in other cases, $\{\texttt{L} = \backslash = \texttt{L}_1\}$ adds new information to the constraint store and can contribute to the solution, as e.g. like for $\texttt{Ms} \equiv \texttt{ms(a, Xs : L) : L}_1$ and $\texttt{Ms}_1 \equiv \texttt{ms(a, Ys : L')} : \texttt{L}'_1$, with additional previous constraints $\{\texttt{L} >= 0, \texttt{L}_1 = \texttt{L} + 1, \texttt{L}' >= 0, \texttt{L}'_1 = \texttt{L}' + 1\}$. In this case, we obtain the solution $\langle\{xs = zs : L, ys = ws : L'\}, \emptyset, \{\texttt{L} = \backslash = \texttt{L}'\}\rangle$, which cannot be covered by any finite set of solutions not containing arithmetic constraints over cardinalities.

(1.b) This closely follows the item (b) of the definition of /= above (predicate /=`hnfbasic`). Unfortunately, the behaviour of this process may depend critically on the particular order in which mutations are generated. To see this, consider $\texttt{Ms} = \{\!\{[A|f], [B]\}\!\}$ and $\texttt{Ms}_1 = \{\!\{[A|f], [C]\}\!\}$, where A, B and C are different constant constructors and f is defined by the program rule $f \rightarrow [A|f]$. Then $\texttt{Ms}/=\texttt{Ms}_1$ can succeed by picking $[B]$ among the elements of `Ms` and checking that $[B] \notin \texttt{Ms}_1$. However, if we pick $[A|f]$ first, then we will have to solve either $[A|f] \notin \texttt{Ms}_1$ or else a strict equality between $[A|f]$ and some element of $\texttt{Ms}_1$; any of these alternatives will lead to an infinite computation. In summary, disequalities between multisets of the same cardinality are very hard to compute, unless all the expressions occurring as elements can be evaluated to normal form in finitely many steps.

(2) If `Ms` is closed and $\texttt{Ms}_1$ is pending (second clause of /=`hnftail`), it is still possible that the condition $\backslash + \{\texttt{L} = \texttt{L}_1\}$ (where $\backslash +$ is the symbol for negation as failure in Sicstus Prolog) entails that both multisets may never be equal. For instance, if $\texttt{Ms} \equiv \texttt{ms(a, empty : 0) : 1}$ and $\texttt{Ms}_1 \equiv \texttt{ms(a, ms(b, e : L) : L}_1\texttt{) : L}_2$, the associated constraints $\{\texttt{L} >= 0, \texttt{L}_1 = \texttt{L} + 1, \texttt{L}_2 = \texttt{L}_1 + 1\}$ entails $\texttt{L}_2 >= 2$, i.e., the condition $\backslash + \{\texttt{1} = \texttt{L}_2\}$ succeeds without evaluating `e` (which could even diverge). The behaviour in this case is "lazy" in the sense that the evaluation of expressions occurring as elements is avoided.
As a second alternative, `getnewelement/5` (whose code is not presented due to lack of space) extracts a new element from the tail of $\texttt{Ms}_1$ and then a recursive call to /=`hnftail` is performed.

(3) If Ms is open with cardinality L and $\texttt{Ms}_1$ is pending with tail information p(N) (third clause of /=hnftail), a lazy behaviour similar to the case (2) above can obtain a solution by imposing the condition $\{\texttt{L} < \texttt{N}\}$. There is also a second alternative, exactly as in item (2).

(4) If Ms and $\texttt{Ms}_1$ are both pending (fourth clause of /=hnftail), the information about their cardinalities is useless for the moment. Therefore, we extract one new element from the tail of Ms and recursively call to /=hnftail, but switching the arguments in order to give chances to progress in the evaluation of both multiset tails.

With respect to /=var, remark that a disequality constraint between to different variables X and Y adds X as disequality constraint for Y and vice versa. On the other hand, all the disequality constraints stored for variables must correspond to data terms; this is checked by predicate isterm whose code has been omitted. So, when solving a constraint X/=E, where E is not a data term, X becomes bound to an imitation binding. This contributes to the lazy behaviour of disequality solving.

3.7 Solving Non-membership Constraints

Non-membership is solved by predicate $\notin$ defined below. Differently to the rest of constraints discussed so far, the non-membership of an element E to a multiset Ms can succeed without any evaluation of E whenever Ms is the empty multiset. The code for this predicate is the following:

```
∉(E,R,ICS,FCS) :- hnf(R,HR,ICS,CS1), ∉ hnf(E,HR,CS1,FCS).

∉ hnf(E,X,ICS,FCS) :- isvar(X), !, ∉ var(E,X,ICS,FCS).
∉ hnf(E,empty:0,ICS,FCS) :- !, ICS = FCS.
∉ hnf(E,ms(E1,Es):L,ICS,FCS) :- /=(E,E1,ICS,CS1), ∉ (E,Es,CS1,FCS).

∉ var(T,X,ICS,FCS) :- isterm(T), !, addconstraints([X#[[T], [ ]]], ICS, FCS).
∉ var(E,X,ICS,FCS) :- extract(ICS,X,Cx,CS1), X = empty:0, propag(Cx,X,CS1,FCS).
∉ var(E,X,ICS,FCS) :- extract(ICS,X,Cx,CS1), X = ms(C,R:L):L1,
      {L1 = L + 1, L >= 0}, propag(Cx,X,CS1,CS2), ∉ (E,X,CS2,FCS).
    % C,R,L,L1 fresh Prolog variables
```

In order to understand the behaviour of the clauses above, consider the program rule $f \rightarrow Zero$ and the goal $G \equiv S(f) \notin xs$. The most general solution for such a goal would be $S(Zero) \notin xs$. However, the clauses for $\notin$ var will not compute this solution; rather, the following infinite sequence of more particular solutions will be enumerated: $xs = \{\!\{\ \}\!\}$; $xs = \{\!\{ Zero \}\!\}$; $xs = \{\!\{ Zero, Zero \}\!\}$; ... The point is that Seta's implementation has been designed to delay the evaluation of function calls as much as possible. Therefore, goals of the form $e \notin xs$ where e is not a data term will often lead to an expensive enumeration of solutions. This is good from a "laziness" point of view. For instance, suppose now that the function f is defined by the rule $f \rightarrow S(f)$. Then the enumeration of solutions has sense since the evaluation to normal form of $S(f)$ would lead to an infinite computation without providing any solution.

4 Conclusions

Following previous work on a general framework for lazy functional programming with algebraic polymorphic types [3,4], we have presented Seta, a language for functional logic programming with multisets and constraints. The fragment of Seta using only strict equality constraints has a firm theoretical foundation, provided by [3,4]. Seta turns out to be very expressive for any kind of problems related to the general idea of multiset rewriting; we have shown an illustrative programming example within the realm of action and change problems.

We have also described a Prolog-based implementation for Seta, easy to understand as specification of the operational semantics, and actually executable on top of any Prolog system which supports simple arithmetic constraints (as e.g. Sicstus Prolog 3.3. [21] which provides a solver for real constraints [13]). Our implementation extends previous related approaches with substantially new techniques, tailored to deal with the combination of lazy evaluation and unification modulo multisets.

Currently we are working out an extension of [3,4], in order to obtain a formal model for Seta's semantics. We also plan to extend Seta's design, with real numbers as a predefined type and allowing a richer repertoire of constraints (including arithmetic constraints) in program rules and goals.

References

1. Antoy S., Echahed R., Hanus M.: *A Needed Narrowing Strategy.* In Proc. POPL'94, pp. 268–279, 1994. 429, 438
2. Arenas-Sánchez P., Gil-Luezas A., López-Fraguas F.J.: *Combining Lazy Narrowing with Disequality Constraints.* In Proc. PLILP'94, Springer LNCS 844, pp. 385–399, 1994. 430, 432, 434, 435, 437, 440, 441
3. Arenas-Sánchez P., Rodríguez-Artalejo M.: *A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types.* In Proc. TAPSOFT'97, Springer LNCS 1214, pp. 453–464, 1997. 429, 430, 431, 432, 434, 444
4. Arenas-Sánchez P., Rodríguez-Artalejo M.: *A Lazy Narrowing Calculus for Functional Logic Programming with Algebraic Polymorphic Types.* In Proc. ILPS'97, the MIT Press, pp. 53–69, 1997. 429, 430, 432, 434, 444
5. Banâtre, J.P. and Le Métayer, D.: *The Gamma model and its discipline of programming.* Science of Computer Programming 15, pp. 55–77, 1990. 430, 433
6. Caballero-Roldán R., Sánchez-Hernández J., López-Fraguas F.J: *User's Manual for $\mathcal{TOY}$.* Tech. Rep. SIP 97/57, Universad Complutense de Madrid, 1997. 436
7. Cheong P.H., Fribourg, L.:*Implementation of Narrowing: The Prolog-Based Approach.* In Apt, de Bakker, Rutten (eds), *Logic Programming languages: constraints, functions and objects*, the MIT Press, pp. 1–20, 1993. 435
8. Damas L., Milner R.: *Principal type schemes for functional programs.* In Proc. POPL'82, pp. 207-212, 1982. 435
9. Dovier A., Rossi G.: *Embedding Extensional Finite Sets in CLP.* In Proc. ILPS'93, the MIT Press, pp. 540–556, 1993. 430
10. González-Moreno J.C., Hortalá-González T., López-Fraguas F.J, Rodríguez-Artalejo M.: *A Rewriting Logic for Declarative Programming.* In Proc. ESOP'96, Springer LNCS 1058, pp. 156–172, 1996. Full version available as TR DIA95/10, http://mozart.sip.ucm.es 429, 432, 433

11. Große G., Hölldobler J., Schneeberger J., Sigmund U., Thielscher M.: *Equational Logic Programming, Actions, and Change.* In Proc. ICLP'92, the MIT Press, pp. 177–191, 1992. 430, 433
12. Hanus M.: *The Integration of Functions into Logic Programming.* A Survey. JLP (19:20). Special issue *Ten Years of Logic Programming*, pp. 583–628, 1994. 429
13. Holzbaur C.: *OFAI clp(Q,R) Manual.*. Edition 1.3.3., Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995. 430, 444
14. Jayaraman B., Plaisted D.A.: *Programming with Equations, Subsets, and Relations.* In Proc. ICLP'89, Vol. 2, the MIT Press, pp. 1051–l068, 1989. 430
15. Jouannaud J.P., Kirchner C.: *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification.* In J.L. Lassez and G. Plotking (eds.), Computational Logic, Essays in Honor of Alan Robinson. The MIT Press, pp. 257–321, 1991. 430
16. Legèard B.: *Programmation en Logique avec Contraintes sur les Ensembles, Multi-ensembles et Séquences. Utilisation en prototypage de logiciels.* PhD thesis, Université de Franche-Comté, 1994. 430
17. Loogen R., López-Fraguas F.J., Rodríguez-Artalejo M.: *A Demand Driven Computation Strategy for Lazy Narrowing.* In Proc. PLILP'93, Springer LNCS 714, pp 184–200, 1993. 429, 430, 435, 437, 438
18. Marriott, K.: *Constraint multiset grammars.* In Proc. IEEE Symposium on Visual Languages, IEEE Computer Society Press, pp. 118–125, 1994. 430, 433
19. Martí-Oliet N., Meseguer J.: *Action and Change in Rewriting Logic.* In R. Pareschi & B. Fronhöfer (eds.). Theoretical Approaches to Dynamic Worlds in Computer Science and Artificial Intelligence. Cambridge M.P., 1995. 430, 433
20. Reddy U.: *Narrowing as the Operational Semantics of Functional Languages.* In Proc. IEEE Symposium on Logic Programming, pp. 138–151, 1985. 429
21. *Sicstus Prolog User's Manual.* Programming Systems Group. Swedish Institute of Computer Science, 1995. 430, 434, 444

A Hidden Herbrand Theorem*

Joseph Goguen[1], Grant Malcolm[2], and Tom Kemp[3]

[1] Dept. of Computer Science & Engineering
University of California at San Diego
goguen@cs.ucsd.edu
[2] Connect Centre, Dept. of Computer Science
University of Liverpool
grant@csc.liv.ac.uk
[3] Oxford

Abstract. The benefits of the object, logic (or relational), functional, and constraint paradigms can be combined, by providing existential queries over objects and their attributes, subject to constraints. This paper provides a precise mathematical foundation for this novel programming paradigm, and shows that it is computationally feasible by reducing it to familiar problems over term algebras (i.e., Herbrand universes). We use the formalism of hidden logic, and our main result is a version of Herbrand's Theorem for that setting. By extending a result of Diaconescu, we lift our results from equational logic to Horn clause logic with equality.

* The research reported in this paper has been supported in part by the Science and Engineering Research Council, the EC under ESPRIT-2 BRA Working Groups 6071, IS-CORE and 6112, COMPASS, Fujitsu Laboratories Limited, and a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program 'New Models for Software Architectures,' sponsored by NEDO (New Energy and Industrial Technology Development Organization).

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 445–462, 1998.

1 Introduction

The object paradigm has many practical advantages, including its support for reuse through inheritance, its intuitive appeal, and its affinity for data abstraction [30]. However, it has not been integrated with the complementary advantages of the logic (or perhaps more accurately, relational) and functional paradigms. The advantages of these paradigms include clean declarative semantics, and (for the relational case) natural integration with database query languages and constraint formalisms. Following [19], we believe that the best way to combine paradigms is to combine their underlying logics; in this paper we extend the relational and functional paradigms to the object paradigm by extending Horn clause logic with equality [19,20] to *hidden* Horn clause logic with equality, building on prior work on *hidden algebra* as a foundation for the object paradigm [10,13,17]. We first study existential queries in a hidden equational setting, and obtain a Herbrand theorem that allows solutions to be constructed in a term algebra. We then extend this theorem to hidden Horn clause logic.

All this provides a semantic foundation for a novel programming style, in which framing a query can activate methods that change the world so that a solution actually comes to exist [21,15]. For example, consider a query about a holiday package, with constraints on cost, flight times, etc.; a solution to this query would be an actual package, with tickets, etc., satisfying the constraints.

The hidden algebra approach to the object paradigm [17] is a natural extension of the initial algebra approach to abstract data types [23,9], and allows reasoning about systems with state. The hidden algebraic approach differs from classical algebraic approaches in that some sorts are declared to be *hidden*, and are used to model the states of objects; intuitively, states cannot be observed directly, but only indirectly through the attributes of objects. The hidden paradigm builds on work of Goguen and Meseguer on abstract machines [18,29]; hidden algebra differs from this mainly in its use of *behavioural satisfaction* for equations, an idea first introduced by Reichel [31]. Later, Reichel [32] introduced the related idea of behavioural equivalence for states, which is also used here.

Section 2 gives a condensed review of overloaded many sorted algebra, and Sections 3 and 4 introduce hidden algebra and present basic results that support reasoning about specifications. An important result in this section states that for certain classes of reachable models, behavioural satisfaction of an equation reduces to satisfaction by an initial algebra; moreover, for ground equations the restriction to reachable models is not required.

The classical Herbrand Theorem [25] says that for the models of a set of Horn clauses, existential queries can be answered by examining a term model, called the Herbrand universe. This result has been generalised to Horn clause logic with equality by Goguen and Meseguer, who showed that in this case also it suffices to examine a term model [19,20]. A hidden Herbrand Theorem is given in Section 5, stating that if a query is behaviourally satisfied by a certain term algebra, then it is behaviourally satisfied by all algebras. Section 6 generalises a result of Diaconescu [5], allowing us to lift results in hidden algebra to hidden Horn clause logic with equality.

2 Prerequisites, Notation and Preliminaries

Our presentation of hidden algebra is based on the notion of algebras as collections of structured sets. We assume familiarity with the 'overloaded' approach to many sorted algebra. To establish notation, we briefly review the main concepts and results; for compatible expositions with more detail, see [16,29].

An **S-sorted set** A is a family of sets A_s indexed by elements $s \in S$. An **S-sorted function** $f\colon A \to B$ is a family $\langle f_s\colon A_s \to B_s \mid s \in S\rangle$; similarly, an S-sorted relation $R \subseteq A{\times}B$ is a family $\langle R_s \subseteq A_s{\times}B_s \mid s \in S\rangle$.

A **signature** (S, Σ) is an $S^*{\times}S$-sorted set $\Sigma = \langle \Sigma_{w,s} \mid w \in S^*,\ s \in S\rangle$; we often write just Σ instead of (S, Σ). Notice that this definition permits *overloading*, in that the sets $\Sigma_{w,s}$ need *not* be disjoint; this can be useful in many applications. A **signature morphism** ϕ from a signature (S, Σ) to a signature (S', Σ') is a pair (f, g) consisting of a map $f\colon S \to S'$ of sorts and an $S^*{\times}S$-sorted family of maps $g_{w,s}\colon \Sigma_{w,s} \to \Sigma'_{f^*(w),f(s)}$ on operation symbols, where $f^*\colon S^* \to S'^*$ is the extension of f to strings defined by $f^*([\,]) = [\,]$ and $f^*(ws) = f^*(w)f(s)$, for w in S^* and s in S.

A **Σ-algebra** A consists of an S-sorted set A (the **carrier** sets) and for each $\sigma \in \Sigma_{w,s}$, a function $A_\sigma\colon A_w \to A_s$ (for $w = s_1 ... s_n \in S^*$, we let $A_w = A_{s_1}{\times}\cdots{\times}A_{s_n}$; in particular, we let $A_{[\,]} = \{\star\}$, some singleton set). A **Σ-homomorphism** from a Σ-algebra A to another B is an S-sorted function $f\colon A \to B$ such that $f_s(A_\sigma(a_1, ..., a_n)) = B_\sigma(f_{s_1}(a_1), ..., f_{s_n}(a_n))$ for each $\sigma \in \Sigma_{w,s}$ with $w = s_1 \ldots s_n$ and $a_i \in A_{s_i}$ for $i = 1, \ldots, n$. (When $w = [\,]$, the condition is simply that $f(A_\sigma) = B_\sigma$.) Let $\mathbf{Alg}_\Sigma$ denote the category with Σ-algebras as objects and Σ-homomorphisms as morphisms.

Given a subsignature $\Psi \subseteq \Sigma$, there is a **reduct** functor, $\restriction_\Psi\colon \mathbf{Alg}_\Sigma \to \mathbf{Alg}_\Psi$, that sends a Σ-algebra A to $A\restriction_\Psi$, which is A viewed as a Ψ-algebra by forgetting about any sorts and operations in Σ that are not in Ψ; similarly, if $f\colon A \to B$ is a Σ-homomorphism, then $f\restriction_\Psi\colon A\restriction_\Psi \to B\restriction_\Psi$ is the Ψ-homomorphism obtained by restricting f to the sorts in Ψ.

Given a many sorted signature Σ and an S-sorted set (of variable symbols) X (where the sets X_s and $\Sigma_{[\,],s}$ are disjoint), we let $T_\Sigma(X)$ denote the **term algebra** with operation symbols from Σ and variable symbols from X; it is the **free** Σ-algebra[1] generated by X, in the sense that any **assignment**, $\theta\colon X \to A$ to a Σ-algebra A, has a unique extension to a Σ-homomorphism $\theta^*\colon T_\Sigma(X) \to A$. We often use the following property of free extensions:

Lemma 1. *Given an assignment $\theta\colon X \to A$ and a Σ-homomorphism $f\colon A \to B$, then $(f \circ \theta)^* = f \circ \theta^*\colon T_\Sigma(X) \to B$.*

We let T_Σ denote the **initial** term Σ-algebra, $T_\Sigma(\emptyset)$; this means there is a unique Σ-homomorphism $!_A\colon T_\Sigma \to A$ for any Σ-algebra A. Given a **ground Σ-term** $tt \in T_\Sigma$, let t_A denote the element $!_A(t)$ in A. Call A **reachable** iff $!_A$ is surjective, i.e., iff each element of A is 'named' by some ground term.

[1] Strictly speaking, the usual term algebra is not free if some terms have multiple parses; however, even then, a closely related term algebra, with operations annotated by their sort, is free.

A **conditional Σ-equation** consists of a variable set X, terms $t, t' \in T_\Sigma(X)_s$ for some sort s, and terms $t_j, t'_j \in T_\Sigma(X)_{s_j}$ for $j = 1, ..., m$, and is written in the form $(\forall X)\, t = t'$ **if** $t_1 = t'_1, ..., t_m = t'_m$. An (**unconditional**) **equation** has $m = 0$, and is written $(\forall X)\, t = t'$. A **ground equation** has $X = \emptyset$. A Σ-algebra A **satisfies** a conditional equation of the form above iff for all $\theta : X \to A$, we have $\theta^*(t) = \theta^*(t')$ whenever $\theta^*(t_j) = \theta^*(t'_j)$ for $j = 1, ..., m$. Given a set E of (possibly conditional) Σ-equations, let $\mathbf{Alg}_{\Sigma,E}$ denote the full subcategory of $\mathbf{Alg}_\Sigma$ with objects the Σ-algebras that satisfy E; we call these (Σ, E)-algebras. If A satisfies an equation e, we write $A \models_\Sigma e$

A Σ-**congruence** on a Σ-algebra A is an S-sorted family of relations, $\equiv_s$ on A_s, each of which is an equivalence relation and also satisfies the **substitutive property**, that given $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, and given $a_i, a'_i \in A_{s_i}$ for $i = 1, ..., n$, then $A_\sigma(a_1, ..., a_n) \equiv_s A_\sigma(a'_1, ..., a'_n)$ whenever $a_{s_i} \equiv_{s_i} a'_{s_i}$ for $i = 1, ..., n$. The **quotient** of A by a Σ-congruence $\equiv$, denoted $A/_\equiv$, has $(A/_\equiv)_s = A_s/_{\equiv_s}$ and inherits a Σ-algebra structure by defining $(A/_\equiv)_\sigma([a_1], \ldots, [a_n]) = [A_\sigma(a_1, \ldots, a_n)]$, where $[a]$ denotes the $\equiv$-equivalence class of a.

3 Hidden Algebra

This section summarises the comprehensive introduction to hidden algebra given in [17]. Hidden algebra captures the fundamental distinction between data values and internal states by modeling the former with 'visible' sorts and the latter with 'hidden' sorts. In order to communicate, the various components of a particular system should share the same representations for data; hence we work with a fixed collection of data values, which can be bundled together to form a fixed algebra. Our assumptions and notation for data values are given in the following:

Definition 1. *Assume a fixed algebra D of data values, let Ψ be its signature and let V be its sort set; assume Ψ is such that for each $d \in D_v$ with $v \in V$ there is some $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D; for simplicity, we assume that $D_v \subseteq \Psi_{[],v}$ for each $v \in V$. We call (V, Ψ, D) the* **visible data universe**.

Signatures in hidden algebra are defined with respect to a fixed visible data universe, which may be thought of as containing standard abstract data types such as the numbers, Booleans, lists, etc.

Definition 2. *A* **hidden signature** *(over (V, Ψ, D)) is a pair (H, Σ) such that $(V \cup H, \Sigma)$ is a many sorted signature with $\Psi \subseteq \Sigma$ and $H \cap V = \emptyset$, and such that:*

1. *if $w \in V^*$ and $v \in V$, then $\Sigma_{w,v} = \Psi_{w,v}$;*
2. *for each $\sigma \in \Sigma_{w,s}$, at most one element of w is in H.*

We often write (H, Σ) for Σ, and S for $V \cup H$. The elements of V are called as **visible sorts**, *and elements of H* **hidden sorts**. *If $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a* **method** *if $s \in H$, and an* **attribute** *if $s \in V$.*

Condition (1) expresses *data encapsulation*, in that if $\Psi \subseteq \Sigma$ is an inclusion of modules, then it does not allow new operations on old data. Condition (2) says that methods and attributes act on single objects.

Definition 3. *Given hidden signatures Σ and Σ', a* **hidden signature morphism** $\Phi\colon \Sigma \to \Sigma'$ *is a signature morphism $\Phi = (f, g)\colon \Sigma \to \Sigma'$ such that:*

1. $f(v) = v$ *for* $v \in V$;
2. $f(H) \subseteq H'$ *(where H' is the hidden sort set of Σ');*
3. $g(\psi) = \psi$ *for* $\psi \in \Psi$; *and*
4. *if* $\sigma' \in \Sigma'_{w',s'}$ *and w' has a sort in $f(H)$, then $\sigma' = g(\sigma)$ for some $\sigma \in \Sigma$.*

The first three conditions say that hidden signature morphisms preserve visibility and hiddenness, while the fourth condition expresses encapsulation, in the sense that no new methods or attributes can be defined on an imported class.

Definition 4. *A* **hidden Σ-algebra** *is a Σ-algebra A such that $A\restriction_\Psi = D$. A* **hidden Σ-homomorphism** $f\colon A \to A'$ *is a Σ-homomorphism such that $f\restriction_\Psi = 1_D$. Let* $\mathbf{HAlg}_\Sigma$ *denote the category of all hidden Σ-algebras.*

An equation is satisfied in hidden algebra if the left and right sides are indistinguishable by any experiment that produces a visible result. We make this precise by defining *contexts*:

Definition 5. *A* **Σ-context** *of sort s is a visible sorted Σ-term having one occurrence of a new symbol z of sort s. A context is* **appropriate** *for a term t iff t has the same sort as z, and $c[t]$ denotes the result of substituting t for z in c. We let $T_\Sigma[z]$ denote the V-sorted set of contexts using the variable z, and we sometimes write a context c as $c[z_s]$ to indicate that z has sort s.*

Definition 6. *A hidden Σ-algebra A* **behaviourally satisfies** *a Σ-equation $(\forall X)\, t = t'$ iff for all appropriate contexts $c \in T_\Sigma[z]$, we have $A \models_\Sigma (\forall X)\, c[t] = c[t']$. Similarly, A* **behaviourally satisfies** *a conditional equation e of the form $(\forall X)\, t = t'$* **if** *$t_1 = t'_1, ..., t_m = t'_m$ iff for every interpretation $\theta\colon X \to A$, we have*

$$\theta^*(c[t]) = \theta^*(c[t'])$$

for all appropriate contexts c whenever $\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$ for $j = 1, ..., m$, and for all appropriate contexts c_j. We denote behavioural satisfaction by $A \models\!\!\models_\Sigma e$ or sometimes just $A \models\!\!\models e$.

A **hidden theory** *is a triple (H, Σ, E), where (H, Σ) is a hidden signature and E is a set of Σ-equations; we write (Σ, E) for short. A* **hidden (Σ, E)-algebra** *is a hidden Σ-algebra A that behaviourally satisfies each equation in E. We let $\mathbf{HAlg}_{\Sigma,E}$ denote the full subcategory of $\mathbf{HAlg}_\Sigma$ whose objects are hidden (Σ, E)-algebras.*

A hidden (Σ, E)-algebra can be seen as one way of implementing objects in the class defined by the specification. The hidden (Σ, E)-algebras give *all possible* implementations. A standard example is that of stack objects; this is an ubiquitous example, but provides a good benchmark for specification formalisms:

Example 1. Here we assume that the data universe specified in `DATA` contains at least the natural numbers of sort `Nat`. We use the notation of OBJ [24], but intend hidden semantics for sorts declared outside `DATA`.

```
th STACK is pr DATA .
  sort Stack .
  op push : Nat Stack -> Stack .
  op top_ : Stack -> Nat .
  op pop_ : Stack -> Stack .
  op empty : -> Stack .
  var S : Stack .
  var N : Nat .
  eq pop push(N,S) = S .
  eq top push(N,S) = N .
endth
```

The first line gives the name `STACK` to the theory, and imports the data specification `DATA`. A hidden sort `Stack` is declared, and the next four lines declare the familiar stack operations, whose behaviour is described by the given equations. The algebras for `STACK` need only appear to satisfy its equations when observed through contexts. For example, the usual implementation of a stack by a pointer and an array does not actually satisfy the equation $\texttt{pop push}(\texttt{N},\texttt{S}) = \texttt{S}$, although it does satisfy it behaviourally, and hence it is a `STACK`-algebra.

4 Behavioural Equivalence

This section states some results concerning behavioural satisfaction that will be useful in following sections. In [17] it is shown that hidden satisfaction can be defined using a smaller class of contexts; that is, we can restrict the number of 'experiments' that are used to distinguish states to what we call *local* contexts:

Definition 7. *A $\Psi(X)$-term is* **local** *iff it is in D or in X; a $\Sigma(X)$-term that is not a $\Psi(X)$-term is* **local** *iff all visible sorted proper subterms are either in D or else in X. A context is* **local** *iff it is a local $\Sigma(\{z\})$-term. We write L_Σ for the S-sorted set of local Σ-terms, and $L_\Sigma[z]$ for the V-sorted set of local contexts.*

Proposition 1. *A conditional equation $(\forall X)\ t = t'$* **if** *$t_1 = t'_1, \ldots, t_m = t'_m$ is behaviourally satisfied by a hidden algebra A iff for all $\theta : X \to A$, if $\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$ for $j = 1, \ldots, m$ and all appropriate local contexts c_j, then $\theta^*(c[t]) = \theta^*(c[t'])$ for all appropriate local contexts c.*

We now consider when two elements of an algebra behave the same way in all experiments; this gives a semantic notion of *behavioural equivalence*:

Definition 8. *Given a hidden signature Σ with hidden subsignature $\Delta \subseteq \Sigma$, and given a hidden Σ-algebra A, elements $a, a' \in A_s$ are* **behaviourally Δ-equivalent** *iff $A_c(a) = A_c(a')$ for all appropriate local Δ-contexts $c \in L_\Delta[z]$ built from operations in Δ, where A_c is the function obtained by interpreting each σ in c as A_σ; in this case, we write $a \equiv_{\Delta,s} a'$, or just $a \equiv_\Delta a'$ if s is*

clear. When we want to emphasise that behavioural Σ-equivalence is defined on a particular hidden Σ-algebra A, we write $\equiv_A$ instead of $\equiv_\Sigma$.

A fundamental property of behavioural equivalence is given in the following

Lemma 2. *Given $\Delta \subseteq \Sigma$ and a hidden Σ-homomorphism $f\colon A \to B$, then $a \equiv_\Delta a'$ in A iff $f(a) \equiv_\Delta f(a')$ in B for all a, a' in A.*

Proof. By definition, $a \equiv_\Delta a'$ means $A_c(a) = A_c(a')$ for all $c \in L_\Delta[z]$, which is equivalent to $f(A_c(a)) = f(A_c(a'))$ for all $c \in L_\Delta[z]$, because f is the identity on visible sorts; moreover, because f is a homomorphism, this is equivalent to $B_c(f(a)) = B_c(f(a'))$ for all $c \in L_\Delta[z]$, which is by definition $f(a) \equiv_\Delta f(a')$.

A key property of behavioural equivalence, which justifies a number of techniques for proving behavioural satisfaction [14,28,17], becomes clear if we make the following definition:

Definition 9. *Given $\Delta \subseteq \Sigma$, a* **behavioural Δ-congruence** *on a hidden Σ-algebra A is a Δ-congruence $\equiv$ which is equality on visible sorts; that is for $v \in V$ and $a, a' \in A_v = D_v$, we have $a \equiv_v a'$ iff $a = a'$.*

Proposition 2. *$\equiv_\Delta$ is the greatest behavioural Δ-congruence.*[2]

This result is proved in [17] and means that two states can be shown to be Δ-equivalent by finding *any* Δ-congruence that relates them: we call this proof technique *hidden coinduction*. Here we are more concerned with abstract properties of behavioural satisfaction than with proof techniques, and the following lemma is particularly useful, as it links satisfaction to behavioural equivalence:

Lemma 3. *A hidden algebra A behaviourally satisfies a conditional equation $(\forall X)\, t = t'$ **if** $t_1 = t'_1, \ldots, t_m = t'_m$ iff for every assignment $\theta\colon X \to A$, whenever $\theta^*(t_j) \equiv_A \theta^*(t'_j)$ for $j = 1, ..., m$, then $\theta^*(t) \equiv_A \theta^*(t')$.*

The proof follows straightforwardly from Definition 8.

Corollary 1. *Given a hidden Σ-algebra A and a conditional Σ-equation all of whose terms have visible sort, then A satisfies the equation iff it behaviourally satisfies the equation.*

Corollary 2. *If a hidden Σ-algebra A satisfies a conditional Σ-equation all of whose conditions have visible sort, then it behaviourally satisfies that equation.*

*Proof. Suppose A satisfies $(\forall X)\ t = t'$ **if** $t_1 = t'_1, \ldots, t_m = t'_m$, where each t_i and t'_i is of visible sort. If $\theta\colon X \to A$ is such that $\theta^*(t_i) \equiv_A \theta^*(t'_i)$ for $i = 1, \ldots, m$, then because $\equiv_A$ is equality on visible sorts, we have $\theta^*(t_i) = \theta^*(t'_i)$, so $\theta^*(t) = \theta^*(t')$ and therefore $\theta^*(t) \equiv_A \theta^*(t')$, as desired.*

[2] This formulation appeared in a conversation between Grant Malcolm and Rolf Hennicker, for the special case where $\Delta = \Sigma$.

The fixed data algebra D gives a kind of lower bound to the classes of models of hidden theories. The class of models can be reduced by adding equations to the theory; however, as soon as these equations confuse data items, the class of models collapses (since any model A has to have $A{\restriction}_\Psi = D$).

Definition 10. *A hidden theory is* **consistent** *iff it has at least one model.*

We turn now to necessary and sufficient conditions for the existence of standard models. We give constructions for initial and final models of hidden theories, and examine their logical properties.

Definition 11. *Given a hidden theory $P = (H, \Sigma, E)$, a ground Σ-term t is* **defined** *iff for every context c (of appropriate sort), there is some $d \in D$ such that $E \models c[t] = d$. P is* **lexic** *iff all ground terms are defined.*

The following basic result is proved in [17]:

Theorem 1. *A hidden theory has an initial model iff it is consistent and lexic.*

An important result for initial models is given in Theorem 2, whose proof uses the following corollary to Lemmas 2 and 1:

Lemma 4. *Given a hidden homomorphism $f : A \to B$ and an equation e, if $B \models e$ then $A \models e$.*

Theorem 2. *Given a hidden theory P, we have:*

(1) an initial P-algebra behaviourally satisfies an equation iff some P-algebra behaviourally satisfies it;
(2) a final P-algebra behaviourally satisfies an equation iff all P-algebras behaviourally satisfy it.

Lemma 4 states that satisfaction of equations propagates backwards along hidden homomorphisms; the following results state that satisfaction propagates forwards for ground equations, and also along surjective homomorphisms. For reasons of space, we omit the proofs.

Lemma 5. *Given a hidden homomorphism $f : A \to B$ and an equation e, if f is surjective or if e is ground, then $A \models e$ implies $B \models e$.*

Corollary 3. *If $P = (\Sigma, E)$ is a consistent, lexic hidden theory, and if e is a ground Σ-equation, then $I_P \models e$ iff all P-algebras behaviourally satisfy e.*

Corollary 4. *If $P = (\Sigma, E)$ is consistent and lexic, and if e is a Σ-equation, then $I_P \models e$ iff all reachable hidden P-algebras behaviourally satisfy e.*

An immediate corollary to this and Theorem 2 is that for consistent, lexic hidden theories, the behaviour of all reachable algebras is the same.

Corollary 5. *If $P = (\Sigma, E)$ is a consistent, lexic hidden theory, and if e is a Σ-equation behaviourally satisfied by some P-algebra, then all reachable hidden P algebras behaviourally satisfy e.*

A useful technique for showing behavioural satisfaction is equational reasoning. We end this section by showing that equational reasoning is sound for hidden algebra. We begin with the following consequence of Lemma 3:

Lemma 6. *For any hidden Σ-algebra A and Σ-equation e, we have $A/_{\equiv_\Sigma} \models e$ iff $A \models\!\models e$.*

We can state soundness of equational deduction as

Proposition 3. *If a Σ-equation e is derivable from a set of equations E, then all hidden (Σ, E)-algebras behaviourally satisfy e.*

We prove the more general form below, which says that if all behavioural consequences of an equation follow from E, then all behavioural models of E behaviourally satisfy the equation.

Proposition 4. *If for every appropriate context c, the Σ-equation $(\forall X)\ c[t] = c[t']$ is derivable from a set of equations E, then all hidden (Σ, E)-algebras behaviourally satisfy $(\forall X)\ t = t'$.*

Proof. *If $(\forall X)\ c[t] = c[t']$ is derivable from E for each context c, then we have $E \models (\forall X)\ c[t] = c[t']$, so for all hidden (Σ, E)-algebras A, we have $A/_{\equiv_\Sigma} \models E$ and so $A/_{\equiv_\Sigma} \models (\forall X)\ c[t] = c[t']$ for each c, which means that $A \models\!\models (\forall X)\ c[t] = c[t']$ for each c which implies $A \models\!\models (\forall X)\ t = t'$.*

5 Hidden Queries

Suppose we want to know if every object of some class can be put into a state that satisfies certain constraints; for example, we might ask 'can the elements of a stack be put in increasing order?' Our approach to the object paradigm suggests we formalise this by regarding constraints as equations[3], and grouping them together in an *existential query*. A *solution* to the query consists of values for the logical variables such that the equations are behaviourally satisfied by every hidden P-algebra.

To make this computationally feasible, we would like to find a term algebra that is 'representative' for all other P-algebras, in the sense that a solution to a query in this algebra systematically translates to a solution in any other. Our 'hidden Herbrand Theorem' says that this is possible in many interesting cases; in fact, we can use the initial P-algebra. Of course, by Theorem 1, this requires a consistent, lexic theory. However, we also show that without these restrictions, equational deduction, and therefore techniques such as narrowing and paramodulation, are sound for arbitrary hidden theories.

Definition 12. *Given a hidden signature Σ, a* **Σ-query** *is a sentence q of the form $(\exists X)\, t_1 = t'_1, \ldots, t_m = t'_m$ where $t_j, t'_j \in T_\Sigma(X)_{s_j}$ for $j = 1, ..., m$. A hidden Σ-algebra A* **behaviourally satisfies** *q, written $A \models\!\models_\Sigma q$, iff there is an assignment $\theta\colon X \to A$ such that for $j = 1, ..., m$, we have $\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$ for all appropriate contexts c_j. Call such an assignment a* **solution** *for the query.*

[3] Section 6 will show how to use first order predicates in constraints.

Note that X can have both visible and hidden variables.

From the definition of behavioural equivalence we have the following

Proposition 5. *Given a Σ-query q of the form $(\exists X)\, t_1 = t'_1, \ldots, t_m = t'_m$ and a Σ-algebra A, then A behaviourally satisfies q with solution $\theta\colon X \to A$ iff $\theta^*(t_j) \equiv_A \theta^*(t'_j)$ for $j = 1, ..., m$.*

Corollary 6. *Given a behavioural Σ-query and a Σ-algebra A, if A satisfies the query, then A behaviourally satisfies it.*

Example 2. In the setting of Example 1, the behavioural query

$$(\exists S, S')\ \mathtt{push}(3, S) = \mathtt{pop}(S')\ ,$$

asks whether there are two stacks that are related in the indicated way, for any possible way of implementing STACK. One solution (among many) is

$$S = \mathtt{empty}\ , \quad S' = \mathtt{push}(0, \mathtt{push}(3, \mathtt{empty}))\ .$$

The solution in this example can be found in the standard term algebra using narrowing, as in the language Eqlog [19,6]. Then the unique homomorphism from it to any other algebra which *satisfies* STACK gives corresponding values in each of these algebras. However, it is not obvious that this technique can guarantee the *behavioural* satisfaction of the query in all algebras which behaviourally satisfy STACK. The results given below show that techniques such as term rewriting, narrowing and coinduction can indeed solve queries over all (Σ, E)-algebras. We first need the following consequence of Lemma 2:

Lemma 7. *Given $h : A \to B$ and a query q, if $A \models q$ then $B \models q$.*

Theorem 3. *Given a hidden theory P, we have:*

(1) an initial P-algebra behaviourally satisfies a query iff all P-algebras behaviourally satisfy it;
(2) a final P-algebra behaviourally satisfies a query iff some P-algebra behaviourally satisfies it.

Goguen and Meseguer [20,19] give a version of Herbrand's theorem for Horn clause logic with equality. It proves that an existential query is satisfied by the initial model of a specification iff it is satisfied by all models of the specification. Theorems 4 and 5 constitute a Herbrand Theorem for hidden algebra and a proof that techniques based on equational deduction are sound for arbitrary hidden theories. In Section 6 we generalise this to hidden Horn clause logic with equality.

Theorem 4. *(*Herbrand Theorem*) Given a consistent, lexic hidden theory P, and a Σ-query q, then $I_P \models q$ iff every P-algebra behaviourally satisfies q.*

The existence of I_P is given by Theorem 1, and the result follows immediately from Theorem 3. A weaker, but still useful corollary to this and Lemma 6 is

Proposition 6. *Given a consistent, lexic hidden theory P and a query q, if I_P satisfies q then all P-algebras behaviourally satisfy q.*

Corresponding to Proposition 4 we have the following, which justifies equational techniques in finding solutions to queries.

Theorem 5. *Let q be a Σ-query of the form $(\forall X)\ t_1 = t'_1, \ldots, t_m = t'_m$. For a set E of Σ-equations, and assignment $\theta : X \to T_\Sigma$, if $E \models (\forall \emptyset)\ \theta^*(c[t_i]) = \theta^*(c[t'_i])$ for all appropriate c and for $i = 1, \ldots, m$, then q is behaviourally satisfied by every hidden (Σ, E)-algebra.*

Proof. For A a hidden (Σ, E)-algebra, $A/_{\equiv_\Sigma} \models E$, so $A/_{\equiv_\Sigma} \models (\forall \emptyset)\ \theta^*(c[t_i]) = \theta^*(c[t'_i])$ for each i and appropriate c, which means that $!_A(\theta^*(t_i)) \equiv_\Sigma !_A(\theta^*(t'_i))$, and so $A \models\!\models (\forall \emptyset)\ \theta^*(t_i) = \theta^*(t'_i)$ as desired.

This means that the technique of narrowing, as described for example in [4], in conjunction with proof techniques such as coinduction [17], can be used to solve queries.

6 Hidden Horn Clause Logic

The queries considered so far have used equations. In languages such as Prolog and Eqlog [19], sentences are Horn clauses with predicate symbols, which are interpreted as relations in models. This section generalises a theorem of Diaconescu [5] to lift our preceding results to hidden Horn clause logic with equality.

Recall (e.g., from [11]) that a **first order signature** (**with equality**) is a triple (S, Σ, Π) such that (S, Σ) is a many sorted signature and Π is an S^+-sorted family of sets of **predicate** or **relation** symbols. We often write (Σ, Π) for (S, Σ, Π), leaving the sort set implicit. For every sort $s \in S$, there is a distinquished **equality** symbol $= \in \Pi_{s\,s}$. A **morphism** $(f, g, k) \colon (S, \Sigma, \Pi) \to (S', \Sigma', \Pi')$ between two first order signatures consists of: a signature morphism (f, g) together with an S^+-sorted family of maps $k_w \colon \Pi_w \to \Pi'_{f^+(w)}$ on predicate symbols, where f^+ is f^* restricted to non-empty strings. A **model** M of a first order signature (S, Σ, Π) is a Σ-algebra together with an interpretation $M_\pi \subseteq M_w$ for each predicate symbol $\pi \in \Pi_w$, with the equality symbol always interpreted as true identity. A **morphism** $h \colon M \to M'$ between (S, Σ, Π)-models M and M' is a Σ-homomorphism such that for any predicate symbol $\pi \in \Pi_{s_1 \ldots s_n}$, if $(m_1, \ldots, m_n) \in M_\pi$, then $(h_{s_1}(m_1), \ldots, h_{s_n}(m_n)) \in M'_\pi$.

For a first order signature (Σ, Π), let $\mathbf{Mod}_{\Sigma,\Pi}$ denote the category of (Σ, Π)-models and morphisms. If $(V, \Psi, \Upsilon) \subseteq (S, \Sigma, \Pi)$ is an inclusion of first order signatures, then there is a forgetful reduct functor $\restriction_{\Psi,\Upsilon} \colon \mathbf{Mod}_{\Sigma,\Pi} \to \mathbf{Mod}_{\Psi,\Upsilon}$ which maps any (Σ, Π)-model M to the (Ψ, Υ)-model $M\restriction_{\Psi,\Upsilon}$ whose carriers are the V-sorted carriers of M, with operations the Ψ-operations on M, and relations the Υ-relations on M. If $h \colon M \to M'$ is a morphism of (Σ, Π)-models, then $h\restriction_{\Psi,\Upsilon}$ is the (Ψ, Υ)-morphism obtained by restricting h to the sorts in Ψ.

Given a first order signature (Σ, Π) and an S-sorted set X of variables with the sets X_s disjoint, we can build the Σ, Π-**term model** $T_{\Sigma,\Pi}(X)$ with carriers

and operations those of $T_\Sigma(X)$, and with $T_{\Sigma,\Pi}(X)_\pi = \emptyset$ for every $\pi \in \Pi$ except when π is the distinguished equality predicate symbol, which is interpreted as equality in $T_\Sigma(X)$. An **assignment** $\theta\colon X \to M$ is an S-sorted mapping from an S-sorted set of variables X to a (Σ,Π)-model M; it extends uniquely to a morphism $\theta^*\colon T_{\Sigma,\Pi}(X) \to M$. For a ground Σ-term t, we let M_t denote the element $!_M(t)$ of M, where $!_M\colon T_{\Sigma,\Pi}(\emptyset) \to M$ is the extension to a morphism of the unique assignment $\emptyset \to M$.

Definition 13. *Given a many sorted first order signature (Σ,Π), a $\Pi(X)$-***atom** *is a term B of the form $\pi(t_1,\ldots,t_n)$, for π a predicate symbol in $\Pi_{s_1\ldots s_n}$ and $t_i \in T_\Sigma(X)_{s_i}$. Given a (Σ,Π)-model M, an assignment $\theta\colon X \to M$ satisfies an atom $B = \pi(t_1,\ldots,t_n)$, written $\theta \models_X B$, iff $(\theta^*(t_1),\ldots,\theta^*(t_n)) \in M_\pi$.*

*A (Σ,Π)-***Horn clause** *is an expression of the form $(\forall X)\, B$ **if** $B_1,\ldots,B_m$ where $B, B_1,\ldots,B_m$ are all $\Pi(X)$-atoms. A Horn clause of the above form is said to be* **unconditional** *iff $m = 0$, and in that case it is written $(\forall X)\, B$.*

*Given a (Σ,Π)-Horn clause e of the form $(\forall X)\, B$ **if** $B_1,\ldots,B_m$, and a (Σ,Π)-model M, we say that M **satisfies** e, written $M \models_{\Sigma,\Pi} e$, iff for every assignment $\theta\colon X \to M$, we have $\theta \models_X B$ whenever $\theta \models_X B_j$ for $j = 1,\ldots,m$.*

For a set C of (Σ,Π)-Horn clauses, let $\mathbf{Mod}_{\Sigma,\Pi,C}$ denote the full subcategory of $\mathbf{Mod}_{\Sigma,\Pi}$ whose objects are all models which satisfy each clause in C.

*A (Σ,Π)-***query** *is an expression of the form $(\exists X)\, B_1,\ldots,B_m$ where the B_j are $\Pi(X)$-atoms. If q is such a query, then a (Σ,Π)-model M **satisfies** q, written $M \models_{\Sigma,\Pi} q$, iff there is an assignment $\theta\colon X \to M$ such that for each $j = 1,\ldots,m$, we have $\theta \models_X B_j$.*

As with equational logic, these concepts have hidden counterparts. We fix a universe (V,Ψ,Υ,D) of data values, where D is a fixed (V,Ψ,Υ)-model.

Definition 14. *A* **hidden first order signature** *(over (V,Ψ,Υ,D)) is a first order signature (Σ,Π), where Σ is a hidden (equational) signature, $\Upsilon \subseteq \Pi$, and*

1. *$\Pi_w \subseteq \Upsilon_w$ for $w \in V^+$;*
2. *if $\pi \in \Pi_w$ for $w \in V^+$, then w has at most one element in H.*

For convenience, we assume that for any $\pi \in \Pi_w$ that takes a hidden sorted argument, that argument is its first.

Definition 15. *For a hidden first order signature (Σ,Π), a hidden (Σ,Π)-***model** *is a (Σ,Π)-model M such that $M{\restriction}_{\Psi,\Upsilon} = D$. A* **morphism** *$h\colon M \to M'$ between hidden (Σ,Π)-models is a morphism of many sorted models such that $h{\restriction}_{\Psi,\Upsilon} = 1_D$. We let $\mathbf{HMod}^D_{\Sigma,\Pi,\Psi,\Upsilon}$ denote the category of all hidden (Σ,Π)-models and their morphisms. We may write $\mathbf{HMod}_{\Sigma,\Pi}$ when the other elements of the signature are clear from the context.*

The notions of behavioural equivalence and satisfaction extends to Horn clause logic by requiring that no atoms distinguish equivalent states:

Definition 16. *Given a hidden (Σ, Π)-model M and $m, m' \in M_s$, then m and m' are* **behaviourally equivalent**, *written $m \equiv_M m'$, iff either $s \in V$ and $m = m'$, or $s \in H$ with $m \equiv_\Sigma m'$, as in hidden equational logic, and for every predicate symbol $\pi \in \Pi_{s_1 \ldots s_n}$ and appropriate hidden context c, we have $(M_c(m), d_2, \ldots, d_n) \in M_\pi$ iff $(M_c(m'), d_2, \ldots, d_n) \in M_\pi$ for all $d_i \in D_{s_i}$.*

Given a (Σ, Π)-model M, an assignment $\theta \colon X \to M$ **behaviourally satisfies** *a $\Pi(X)$-atom B iff B is of the form $t = t'$ and $\theta^*(t) \equiv_M \theta^*(t')$, or B is of the form $\pi(t_1, \ldots, t_n)$, where π is not the equality symbol, and $\theta \models_X B$. In both cases we write $\theta \models\!\!\models_X B$.*

Given a (Σ, Π)-Horn clause e of the form $(\forall X)\, B$ **if** *$B_1, \ldots, B_m$, and a (Σ, Π)-model M, we say that M* **behaviourally satisfies** *e, written $M \models\!\!\models_{\Sigma,\Pi} e$, iff for every assignment $\theta \colon X \to M$, we have $\theta \models\!\!\models_X B$ whenever $\theta \models\!\!\models_X B_j$ for $j = 1, \ldots, m$.*

Given a (Σ, Π)-query q of the form $(\exists X)\, B_1, \ldots, B_m$ we say that a (Σ, Π)-model M **behaviourally satisfies** *q, written $M \models\!\!\models_{\Sigma,\Pi} q$, iff there is an assignment $\theta \colon X \to M$ such that for each $j = 1, \ldots, m$, we have $\theta \models\!\!\models_X B_j$.*

Diaconescu [5] gives a way of translating a first order signature into an algebraic signature by treating the predicate symbols as function symbols with result sort *Bool*, where *Bool* is a new sort with a single new constant *true*. Here we extend his definition to a translation between hidden signatures.

Definition 17. *For any hidden first order signature (Σ, Π), define a hidden algebraic signature $(\Sigma \cup \Pi^b)$ over a universe $(V^b, \Psi \cup \Upsilon^b, D^b)$ by*

- *$V^b = V \cup \{Bool\}$, where Bool is a new sort name;*
- *Π^b is a signature defined by $\Pi^b_{w,Bool} = \Pi_w$ and $\Pi^b_{w,s} = \emptyset$ for $s \neq Bool$;*
- *Υ^b is a signature defined by $\Upsilon^b_{w,Bool} = \Upsilon_w$*
- *D^b is the $\Psi \cup \Upsilon^b$-algebra with $D^b_v = D_v$ for $v \in V$ and $D_{Bool} = \{true, false\}$, with Ψ-operation symbols interpreted as in D, and with $D^b_\pi(d_1, \ldots, d_n) = true$ if $(d_1, \ldots, d_n) \in D_\pi$ and false otherwise, for $\pi \in \Pi_{s_1 \ldots s_n}$ and $d_i \in D_{s_i}$.*

Again generalising Diaconescu [5], the corresponding translation from algebras to models uses an adjunction:

Definition 18. *Given a hidden first order signature (Σ, Π), define a forgetful functor $W_{\Sigma,\Pi} \colon \mathbf{HAlg}^{D^b}_{\Sigma \cup \Pi^b} \to \mathbf{HMod}^{D}_{\Sigma,\Pi}$ to map a $\Sigma \cup \Pi^b$-algebra A to the (Σ, Π)-model whose S-sorted carriers are those of A, with operations those of A restricted to Σ, and with relations in Π defined by $(a_1, \ldots, a_n) \in W(A)_\pi$ iff $A_\pi(a_1, \ldots, a_n) = true$. If $f \colon A \to A'$ is a morphism in $\mathbf{HAlg}_{\Sigma \cup \Pi^b}$ then $W_{\Sigma,\Pi}(f) = f$. We often write W instead of $W_{\Sigma,\Pi}$.*

It is straightforward to check that $W_{\Sigma,\Pi}$ is well-defined.

Theorem 6. *Given a hidden signature (Σ, Π), $W_{\Sigma,\Pi}$ has an inverse F.*

We sketch the proof: given a (Σ, Π)-model M the carriers of $F(M)$ are those of M. The Σ-operations are interpreted as in M and, for each $\pi \in \Pi_w$, define

$F(M)_\pi(\overline{m}) = \mathit{true}$ if $(\overline{m}) \in M_\pi$ and *false* otherwise. To see that W and F are each other's inverse, note that neither change the carriers or Σ-interpretations of their arguments, and

$$m \in W(F(M))_\pi \text{ iff } F(M)_\pi(m) = \mathit{true} \text{ iff } m \in M_\pi \ .$$

We now consider Horn clause specifications and their models.

Definition 19. *Given a set C of (Σ, Π)-Horn clauses, let $\mathbf{HMod}^D_{\Sigma,\Pi,\Psi,\Upsilon,C}$ denote the full subcategory of $\mathbf{HMod}^D_{\Sigma,\Pi,\Psi,\Upsilon}$ whose objects behaviourally satisfy each clause in C. We shall often write this as $\mathbf{HMod}_{\Sigma,\Pi,C}$ and call its objects $(\Sigma, \Pi, \Psi, \Upsilon, C)$-**models** or just (Σ, Π, C)-**models** if context permits.*

Diaconescu [5] defines a translation from Horn clauses to conditional equations; we extend this to include queries:

Definition 20. *Given a (Σ, Π)-Horn clause e and a (Σ, Π)-query q, define $\alpha(e)$, a conditional $\Sigma \cup \Pi^b$-equation, and $\alpha(q)$, a $\Sigma \cup \Pi^b$-query, as follows:*

1. *Every equation $t_1 = t_2$ is left untouched;*
2. *every atom $\pi(t_1, \ldots t_n)$ not of the above form is translated as $\pi(t_1, \ldots, t_n) = \mathit{true}$.*

Some basic properties of this mapping are:

Lemma 8. *Given a $(\Sigma \cup \Pi^b)$-algebra A and elements $a, a' \in A_s$ for some $s \in S$, we have $a \equiv_\Sigma a'$ iff $a \equiv_{W_{\Sigma,\Pi}(A)} a'$.*

Corollary 7. *Given a $(\Sigma \cup \Pi^b)$-algebra A, for any $\Pi(X)$-atom B and any $\theta : X \to A$, we have $\theta \models_X B$ iff $\theta^*(t) \equiv_\Sigma \theta^*(t')$, where $\alpha(B) = (t = t')$.*

Finally, we can formalise the validity of translating behavioural satisfaction of Horn clauses into behavioural satisfaction of conditional equations:

Proposition 7. *For any $(\Sigma \cup \Pi^b)$-algebra A, we have*

(1) for e a (Σ, Π)-Horn clause, $A \models^{D^b}_{\Sigma \cup \Pi^b} \alpha(e)$ iff $W_{\Sigma,\Pi}(A) \models^D_{\Sigma,\Pi} e$
(2) for q a (Σ, Π)-query, $A \models^{D^b}_{\Sigma \cup \Pi^b} \alpha(q)$ iff $W_{\Sigma,\Pi}(A) \models^D_{\Sigma,\Pi} q$

*Proof. To see (1), let e be a Horn clause of the form $(\forall X)\, B$ **if** $B_1, \ldots, B_m$, and let $\alpha(e)$ be $(\forall X)\, t = t'$ **if** $t_1 = t'_1, \ldots, t_m = t'_m$. Then $A \models e$ iff for every $\theta : X \to A$ we have $\theta^*(t) \equiv_\Sigma \theta^*(t')$ whenever $\theta^*(t_j) \equiv_\Sigma \theta^*(t'_j)$, iff (by Corollary 7) for every $\theta : X \to A$ we have $\theta \models_X B$ whenever $\theta \models_X B_j$ for $j = 1, \ldots, m$, which is equivalent to $W(A) \models e$. The proof of (2) is similar.*

Corollary 8. *Given a (Σ, Π)-Horn clause e and a (Σ, Π)-query q, if M is any (Σ, Π)-model then*

(1) M behaviourally satisfies e iff $F_{\Sigma,\Pi}(M)$ behaviourally satisfies $\alpha(e)$;
(2) M behaviourally satisfies q iff $F_{\Sigma,\Pi}(M)$ behaviourally satisfies $\alpha(q)$.

Proposition 7 and the above corollary means we can check whether a model behaviourally satisfies a Horn clause by testing whether an algebra behaviourally satisfies a conditional equation. We now examine standard models.

Definition 21. *A set C of (Σ, Π)-Horn clauses is* **lexic** *iff $\alpha(C)$ is lexic.*

Using this, we can extend Theorem 4 to obtain a 'hidden Herbrand theorem' for hidden sorted Horn clause logic with equality. Theorem 4 and Corollary 8 give:

Theorem 7. *(*Herbrand Theorem*) Given a consistent lexic set C of (Σ, Π)-Horn clauses, then $I_P \models \alpha(q)$ iff $M \models q$ for every model M in $\mathbf{HMod}_{\Sigma,\Pi,C}$, where $P = (\Sigma \cup \Pi^b, \alpha(C))$.*

In the same way, the equivalence between hidden algebras and first order models allows results from Section 5 to be lifted to first order models. For example, the use of narrowing is justified by Theorem 5 to give

Theorem 8. *Let q be a (Σ, Π)-query such that $\alpha(q)$ is of the form $(\forall X)\ t_1 = t'_1, \ldots, t_m = t'_m$. For a set C of (Σ, Π)-Horn clauses, and assignment $\theta : X \to T_\Sigma$, if $\alpha(C) \models (\forall \emptyset)\ \theta^*(c[t_i]) = \theta^*(c[t'_i])$ for all appropriate c and for $i = 1, \ldots, m$, then q is behaviourally satisfied by every hidden (Σ, Π, C)-model.*

7 Conclusions

The results of this paper lay the foundations for a programming paradigm that combines the advantages of the logic and object paradigms. Our hidden algebraic approach differs from classical algebraic approaches in using a notion of behavioural satisfaction and a fixed interpretation for data sorts. In this it is quite similar to Diaconescu's categorical approach to the constraint paradigm [6,7], which uses a notion of built in data types. However, in hidden algebra these built-ins are protected, and hidden specifications have loose semantics with protected data. It is this loose semantics that allows hidden algebra to capture nondeterminism by underspecification [17]. When a hidden theory is deterministic, an initial, term-based model exists, which behaviourally satisfies an existential query iff all models behaviourally satisfy it. This gives rise to the two Herbrand theorems for hidden equational logic and hidden Horn clause logic, which allow solutions to be constructed in initial term algebras. There is no completeness result for hidden algebra, as solving constraints in hidden specifications can involve arbitrary constraints; however, we have shown that coinduction techniques can considerably simplify proofs, and in many cases reduce behavioural satisfaction to standard satisfaction. Some of these techniques have already been incorporated into the algebraic specification language CafeOBJ [8], and are also used in related *coalgebraic* approaches to the object paradigm [33,26,27,2].

A useful direction for future research is to extend our results to include the kind of subtyping given by order sorted algebra [22,12]. Burstall and Diaconescu [1] have extended the hiding process to many other institutions, and

in particular, to order sorted algebra. Malcolm and Goguen [28] show that hidden order sorted logic forms an institution, using a construction that differs from Burstall and Diaconescu's in its treatment of error-handling; yet another treatment of ordered sorts in hidden algebra preserves the relationship between hidden algebra and coalgebra [3].

References

1. Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*. Prentice-Hall International, 1994. 459
2. Corina Cîrstea. Coalgebra semantics for hidden algebra: parameterized objects and inheritance. In *Proc. 12th Workshop on Algebraic Development Techniques*. Springer-Verlag Lecture Notes in Computer Science, to appear, 1998. 459
3. Corina Cîrstea, Grant Malcolm, and James Worrell. Hidden order sorted algebra: subtypes for objects. Draft, Oxford University Computing Laboratory, 1998. 460
4. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990. 455
5. Răzvan Diaconescu. The logic of Horn clauses is equational. Technical Report PRG-TR-3-93, Programming Research Group, University of Oxford, 1993. Written in 1990. 446, 455, 457, 458
6. Răzvan Diaconescu. *Category-based Semantics for Equational and Constraint Logic Programming*. PhD thesis, Oxford University, 1994. 454, 459
7. Răzvan Diaconescu. A category-based equational logic semantics to constraint programming. *Lecture Notes in Computer Science*, 1130:200–222, 1996. 459
8. Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for Cafeobj. Technical Report IS-RR-96-0024S, Japan Advanced Institute of Science and Technology, 1996. 459
9. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985. 446
10. Joseph A. Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford University Press, 1991. 446
11. Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992. 455
12. Joseph A. Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994. 459
13. Joseph A. Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*. Springer-Verlag Lecture Notes in Computer Science 785, 1994. 446
14. Joseph A. Goguen and Grant Malcolm. Proof of correctness of object representations. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*, chapter 8, pages 119–142. Prentice-Hall International, 1994. 451
15. Joseph A. Goguen and Grant Malcolm. Situated adaptive software: beyond the object paradigm. In *Proc. New Models for Software Architecture (Tokyo)*. Information-technology Promotion Agency, 1995. 446

16. Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996. 447
17. Joseph A. Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, Department of Computer Science and Engineering, University of California at San Diego, 1997. An extended abstract appears in Proc. *Intelligent Systems: a Semiotic Perspective*, 1996. 446, 448, 450, 451, 452, 455, 459
18. Joseph A. Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer-Verlag Lecture Notes in Mathematics 140, 1982. 446
19. Joseph A. Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984. 446, 454, 455
20. Joseph A. Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer-Verlag Lecture Notes in Mathematics 250, 1987. 446, 454
21. Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. 446
22. Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. 459
23. Joseph A. Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80–149. 446
24. Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing obj. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. to appear. Also available as a technical report from SRI International. 450
25. Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III*, 33(128), 1930. 446
26. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in LNCS, pages 210–231. Springer, Berlin, 1996. 459
27. Grant Malcolm. Behavioural equivalence, bisimilarity, and minimal realisation. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types, WADT11. Oslo Norway, September 1995*, pages 359–378. Springer-Verlag Lecture Notes in Computer Science 1130, 1996. 459
28. Grant Malcolm and Joseph A. Goguen. Proving correctness of refinement and implementation. Technical Monograph PRG-114, Programming Research Group, Oxford University, 1994. 451, 460

29. José Meseguer and Joseph A. Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985. 446, 447
30. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997. 446
31. Horst Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *Proceedings, Third Hungarian Computer Science Conference*. Akademiai Kiado, 1981. Budapest. 446
32. Horst Reichel. Behavioural validity of conditional equations in abstract data types. In *Contributions to General Algebra 3*. Teubner, 1985. Proceedings of the Vienna Conference, June 21-24, 1984. 446
33. Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995. 459

Integrating Constraint Propagation in Complete Solving of Linear Diophantine Systems

Farid Ajili* and Hendrik C.R. Lock

[1] IC-Parc, William Penny Laboratory, Imperial College
London SW7 2AZ, United Kingdom
ajili@loria.fr
[2] SAP AG, Logistics Development and Planning
POB 14 61, D-69190. Walldorf, Germany
hendrik.lock@sap-ag.de

Abstract. Several *complete* methods for solving linear Diophantine constraints have been proposed. They can handle *infinite* domains, but their pruning during search is relatively weak. In contrast to those, consistency techniques based constraint propagation provides stronger pruning and have been applied successfully to many combinatorial problems, but are limited to *finite* domains. This paper studies the combination of (1) a complete solver which is based on a geometric interpretation and (2) propagation techniques. We study the pruning potential created through such a combination, both conceptually and experimentally. In addition, it turns out that dynamic variables orderings can be easily embedded in the method. Our result is an extended solver, which is implemented in `Java`, based on which we present some interesting features and a few experimental results.

1 Introduction

Several complete methods for solving linear Diophantine constraints have been developed in recent years (see *e.g.* [2,6,8,10,11,15]). Here complete means computing a finite representation of the solution set, the latter is potentially infinite. An adequate representation is the set of non-decomposable solutions such that any solution can be written as a $\mathbb{N}$-linear combination of these non-decomposable solutions. Most of the various mathematical techniques behind these methods originated in the areas of linear algebra (*e.g.* [10]), automata theory (*e.g.* [5,6,15]) and geometric interpretation (*e.g.* [1,2,6,8]).

An examination of the solving machineries underlying those methods shows that they do not *dynamically* exploit constraints during the search. Compared to approaches based on constraint logic programming (CLP), we observe a lack of strong pruning criterion driven by constraint-based reasoning. In addition, they do not embed the power of problem-specific heuristics.

* This work was done while the first author was in LORIA/INRIA-Lorraine. Nancy, France.

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 463–480, 1998.

The success of constraint-based reasoning for pruning search spaces has been established through many systems that follow the CLP scheme [13,12]. A particular technique called constraint propagation establishes local consistency during search (see *e.g.* [9,12]). Propagation techniques have been applied successfully and with acceptable efficiency to a large number of discrete combinatorial problems. The key idea of propagation is to eliminate inconsistent value pairings with the aim to significantly reduce value domains prior to search steps.

Our work is motivated by the observation that the pruning power of complete solving methods is still too weak for practical reasons. Our idea is to improve pruning by means of constraint propagation. We claim that combining a complete solving method with propagation techniques yields a competitive constraint solver, provided that this combination preserves the advantages of each of its parts. This paper concentrates precisely on how to integrate constraint propagation and heuristics into the family of complete methods based on geometric interpretation. As a representative of this family we choose the algorithm of F. Ajili & E. Contejean [1,2] (named here `ACalg`) since it is a generalisation of the algorithms [6,8] and solves systems of both equations and inequations. The `ACalg` algorithm searches for solutions starting from canonical tuples. A tuple is incremented if a successor generation criterion succeeds. This criterion, which is based on a geometric interpretation, enforces convergence of search and ensures termination.

Our approach combines `ACalg` algorithm with constraint propagation. The combination is achieved by attaching to each search node a local constraint store. This store contains the information necessary to establish stronger pruning conditions. The constraint store is constructed incrementally in the sense that a node inherits the store of the predecessor node, and tells additional information in the form of relevant constraints. At each node, constraint propagation is used to derive stronger information, that is, to identify inconsistencies. On this basis, we can design new pruning rules in charge of cutting off globally inconsistent search branches and skipping inconsistent sub-spaces. Quite interestingly, it turned out that the freezing mechanism used in [1,2,8] can be encoded by means of constraints. This is important since the freezing can closely interact with propagation thus deriving stronger bounds on the domains of variables. Another issue is the embedding of dynamic variable orderings (DVO for short) which considerably influences the search behaviour, such as the *first fail* strategy. The embedding is achieved at the node level by means of the local constraint store. This means that existing DVO strategies which are based on the notion of constraint store can be easily adapted to our method.

The paper is organised as follows. Section 2 introduces some formal preliminaries. In Section 3, we present a generic solving procedure that captures at an abstract level the common ideas of the family of methods based on geometric interpretation. Section 4 presents algorithm `ACalg` as a specific instance of the generic procedure. Section 5 motivates in greater detail the need for stronger pruning rules in `ACalg`. Section 6 makes precise how to integrate constraint store and propagation into complete solving. Section 7 presents new stronger pruning

rules. Section 8 addresses implementation issues and provides benchmarks. The last section concludes and outlines further works. Proofs and technical details can be found in [3].

2 Formal Preliminaries

As usual $\mathbb{N}$ and $\mathbb{Z}$ denote respectively the set of naturals and integers. A vector $v = (v_1, v_2 \ldots, v_q)$ of $\mathbb{N}^q$ is called a tuple. We denote by e_j the j^{th} canonical tuple of $\mathbb{N}^q$. For a given q, $\leq_q$ is the component-wise partial extension of $\leq$ in $\mathbb{N}^q$. $\mathcal{V} = \{x_1, x_2, \ldots\}$ is a set of variables taking values in $\mathbb{N}$. A (linear Diophantine) constraint is $\sum_{j\in[1..n]} c_j\ x_j\ \#\ c_0$, where $c_0,\ c_j \in \mathbb{Z}$, $x_j \in \mathcal{V}$, and $\# \in \{=^?, \leq^?\}$. It is written $cX\ \#\ c_0$, where $c = (c_1, \ldots, c_n)$ and $X = (x_1, \ldots, x_n)$, and called homogeneous if $c_0 = 0$. A linear (homogeneous) constraint system is a conjunction of m (homogeneous) constraints on q variables. It is denoted $CX \odot c$ where C is an $m \times q$ integer coefficient matrix, $\odot \in \{=^?, \leq^?\}^m$ and $c \in \mathbb{Z}^m$. $CX \odot c$ can be clearly split into $AX =^? a \wedge BX \leq^? b$. Let m_B denote the number of rows of B.

Let $[inf..sup]$, where $inf \in \mathbb{N}$ and $sup \in \mathbb{N} \cup \{\infty\}$, represent a usual (possibly infinite) interval of $\mathbb{N}$. For x in $\mathcal{V}$, $x \in^? [inf..sup]$ is said to be a domain (interval) constraint. We abbreviate a conjunction of q domain constraints $\bigwedge_{j\in[1..q]} x_j \in^? [l_j..u_j]$ by $X \in^? [L..U]$ where $L = (l_1, \ldots, l_q)$ and $U = (u_1, \ldots, u_q)$.

In this paper, the computation domain we consider for the constraints is $\mathbb{N}$. Here, $\mathcal{S}ol(\mathcal{C})$ means the set of solutions of the constraint system $\mathcal{C}$ in $\mathbb{N}$. Let $\mathcal{C}_1$ be $AX =^? 0 \wedge BX \leq^? 0$. A solution of $\mathcal{C}_1$ is said to be *non-decomposable* if it is not the sum of any two other non-null solutions of $\mathcal{C}_1$. It is well-known that the set of non-decomposable solutions provides a complete representation of $\mathcal{S}ol(\mathcal{C}_1)$:

Lemma 1. *$\mathcal{S}ol(\mathcal{C}_1)$ is generated by its sub-set of non-decomposable solutions: any solution of $\mathcal{C}_1$ is a $\mathbb{N}$-linear combination of non-decomposable solutions.*

Elements of $\mathcal{S}ol(\mathcal{C}_1)$ are the projections over the first q components of elements of $\mathcal{S}ol(AX =^? 0 \wedge BX + Z =^? 0)$, where Z is an m_B-wide tuple of slack variables such that z_i is the slack variable of the i^{th} inequation:

Lemma 2. *A solution $s \in \mathbb{N}^q$ of $\mathcal{C}_1$ is non-decomposable if and only if its associated solution $(s, -Bs) \in \mathbb{N}^{q+m_B}$ of $\mathcal{S}ol(AX =^? 0 \wedge BX + Z =^? 0)$, is minimal w.r.t. $\leq_{q+m_B}$.*

Note that if $m_B = 0$ then non-decomposability property coincides with the minimality w.r.t. $\leq_q$. Solving techniques presented here extend trivially to the non-homogeneous (in)equations according to the lines of [2,8].

3 A Generic Solving Procedure

Let $\mathcal{C}_1$ be the input constraint system on q variables $x_1, \ldots, x_q$. We define at an abstract level a generic procedure $\mathcal{P}roc$ for solving $\mathcal{C}_1$. $\mathcal{P}roc$ is parametric in

four operations *Wanted*, *Generate*, *Cut*, and ρ. An instance of $\mathcal{P}$roc is completely determined by giving specifications to its parameters.

We further define an abstract data type $Node = (v : \mathbb{N}^q,\ \sigma : Store)$, where *Store* is the constraint language inductively defined by:

$$\begin{array}{rll} Store ::= & Store \wedge Store & \text{conjunction} \\ | & cX \leq^? c_0 & \text{inequation} \\ | & cX =^? c_0 & \text{equation} \\ | & x \in^? [inf..sup] & \text{domain constraint} \\ | & \mathbb{T} & \text{true (1)} \\ | & \mathbb{F} & \text{false (0)} \end{array}$$

Additionally, the abstract data type *Node* provides four operations:

$$\begin{array}{ll} Wanted\ :\ Node \to \{0,1\} & Cut\ : Node \to \{0,1\} \\ Generate\ : Node \times 2^{\{1,2,\ldots,q\}} \to \{0,1\} & \rho\ :\ \ Node \to Node \end{array}$$

A node $y = (v, \sigma)$ of type *Node* consists of a tuple $v(y)$ and a local constraint store $\sigma(y)$. Whenever no ambiguity arises, $v(y)$ and $\sigma(y)$ are written simply v and σ. Then, given a node y, y_j denotes the j-th component of its tuple $v(y)$. By slight abuse of notation, we also use e_j for $(e_j, \sigma(e_j))$. Informally, the role of $\sigma(y)$ is to accumulate information about the current search state.

Let us give an informal description of the abstract machine. $\mathcal{P}$roc maintains two lists of nodes. $\mathcal{P}$ is the list of nodes already reached and pending for development, and $\mathcal{M}$ is the list of collected nodes. *Wanted* specifies nodes that $\mathcal{P}$roc collects in $\mathcal{M}$: this may be solutions, non-decomposable/minimal solutions .... *Cut* characterises nodes whose descendants cannot satisfy *Wanted*. *Generate* defines how to develop a node into its direct descendants. Finally, ρ expresses the processing of the local store.

Initialise	$[\,];\ [\rho(e_{j_1}^{\{j_2,\ldots,j_q\}}), \ldots, \rho(e_{j_{q-1}}^{\{j_q\}}), \rho(e_{j_q}^{\{\}})];$
	if $e_{j_1} \prec_0 \ldots \prec_0 e_{j_{q-1}} \prec_0 e_{j_q}$
Leaf	$\mathcal{M};\ y@\mathcal{P} \longrightarrow \mathcal{M}\ ;\ \mathcal{P}$ if $Cut(y)$
Solution	$\mathcal{M};\ y@\mathcal{P} \longrightarrow y@\mathcal{M}\ ;\ \mathcal{P}$ if $Wanted(y)$
Develop	$\mathcal{M};\ y@\mathcal{P} \longrightarrow \mathcal{M};\ [\rho(y + e_{j_1}^{\mathcal{F} \cup \{j_2,\ldots,j_l\}}), \ldots, \rho(y + e_{j_l}^{\mathcal{F}})]@\mathcal{P}$
	if $Generate(y, \{j'_1, \ldots, j'_p\})$, $\{j'_1, \ldots, j'_p\} \setminus \mathcal{F} = \{j_1, \ldots, j_l\}$ and
	$\rho(y + e_{j_1}) \prec_y \rho(y + e_{j_2}) \prec_y \ldots \prec_y \rho(y + e_{j_l})$

Fig. 1. The Stack Version of $\mathcal{P}$roc(*Wanted*, *Cut*, *Generate*, ρ).

$\mathcal{P}$roc has both breadth-first and depth-first (or stack) versions (see [8] for details). The breadth-first version of $\mathcal{P}$roc constructs a direct acyclic graph (dag for short). A dag contains redundancies: a node may appear as a direct successor of several nodes. Such redundancies can be avoided by turning the dag into a forest, which is achieved by imposing a total local ordering $\prec_y$ on the direct successors of y. Assume that a node y has two successors $\rho(y+e_{j_1})$ and $\rho(y+e_{j_2})$ such that $\rho(y+e_{j_1}) \prec_y \rho(y+e_{j_2})$. To obtain a forest instead of a dag, one forbids to increment x_{j_1} in the subtree rooted at $\rho(y+e_{j_2})$. In this case, x_{j_1} is said *frozen* in all such a subtree. For the ordering $\prec_y$, one can simply take for instance the node-independent ordering: $\rho(y + e_{j_1}) \prec_y \rho(y + e_{j_2})$ iff $j_2 < j_1$. This freezing mechanism [8] associates to each node a *freeze pattern* $\mathcal{F}(y)$ which is a subset of $\{1, \ldots, q\}$. A node y with freeze pattern $\mathcal{F}$ is denoted by $y^{\mathcal{F}}$. Simultaneously with freezing, the forest is developed in a *depth-first* manner by representing $\mathcal{P}$ as a *stack*. For technical simplicity, we add to each forest a "fictitious" root 0 which contains a tuple of zeros and has an empty freeze pattern.

The complete stack-based procedure is shown in Figure 1. When starting $\mathcal{P}$roc, Initialise simply sets the lists $\mathcal{P}$ and $\mathcal{M}$. Then, it proceeds with the 3 remaining rules until $\mathcal{P}$ is empty. At each step, it performs the first applicable rule in *textual order* to the top of $\mathcal{P}$.

4 Solving Linear Diophantine Systems

Algorithms which are specific instances of $\mathcal{P}$roc do exist in the literature. Indeed, the following algorithms can be viewed as instances of $\mathcal{P}$roc. M. Clausen & A. Fortenbacher [6] solve a single homogeneous equation, E. Contejean & H. Devie [8] solve a system of homogeneous equations, and F. Ajili & E. Contejean [1,2] solve a system of homogeneous equations and inequations. The latter algorithm, already called `ACalg`, is a generalisation of the previous approaches since it simultaneously solves equations and inequations. Due to its generality, we have chosen `ACalg` as a starting point, and will provide its detailed description in the following. Thereby, we assume that $\mathcal{C}_1$ is $AX =^? 0 \ \wedge \ BX \leq^? 0$.

In `ACalg`, the abstract data type *Node* contains only a tuple v but no store σ. Therefore, in this section we can drop the distinction between tuple and node. Following Lemma 1, the solution set of $\mathcal{C}_1$ is generated by its sub-set of non-decomposable solutions. Accordingly, *Wanted* specifies the non-decomposability property of collected nodes in $\mathcal{M}$:

$$\mathit{Wanted}(y) = (Ay = 0 \ \wedge \ By \leq 0 \ \wedge \ \nexists y_0 \in \mathcal{M}, \ (y_0, -By_0) \leq_{q+m_B} (y, -By))$$

The operation for processing the constraint store is the identity:

$$\rho(y) = y$$

A central issue in all cited approaches is a criterion with a geometric interpretation which determines the set of descendants of a node. [2] proposed a criterion Ψ which generalises the one of [8] in order to reflect the presence of inequations. A

node y is developed into a descendant $y + e_j$ if the criterion $\Psi(y, j)$ holds. We capture this in the following specification of *Generate*:

$$Generate(y, \{j_1, \ldots, j_p\}) = (\{j_1, \ldots, j_p\} = \{j \mid \Psi(y, j)\})$$

The authors remarked that due to inequations, descendants of a non-decomposable solution can be themselves non-decomposable solutions. Thus, to preserve completeness, one has to distinguish which solution nodes can be deleted from $\mathcal{P}$, and which have to be developed further. A solution node having a null-defect, that is, $By = 0$, can be *pruned* because it naturally subsumes all descendant solutions. This distinction was the reason why $\mathcal{M}$ became split into two disjoint lists: $\mathcal{M} = \mathcal{M}_{=} \cup \mathcal{M}_{<}$, where $\mathcal{M}_{=}$ is the list of tuples with null-defect. *Cut* is then specified accordingly:

$$Cut(y) = (\exists y_0 \in \mathcal{M}_{=},\ y_0 \leq_q y)$$

Moreover, it follows that *Wanted* is a disjunction of two disjoint sub-relations

Initialise	$[\,];\quad [\,];\ [e_{j_1}^{\{j_2,\ldots,j_q\}}, \ldots, e_{j_{q-1}}^{\{j_q\}}, e_{j_q}^{\{\}}]$; if $e_{j_1} \prec_0 \ldots \prec_0 e_{j_{q-1}} \prec_0 e_{j_q}$
Leaf	$\mathcal{M}_{=}; \mathcal{M}_{<}; y^{\mathcal{F}}@\mathcal{P} \longrightarrow \quad \mathcal{M}_{=} \quad ; \mathcal{M}_{<} \quad ; \mathcal{P}$ if $\exists s \in \mathcal{M}_{=}\ s \leq_q y$
Solution$_{=}$	$\mathcal{M}_{=}; \mathcal{M}_{<}; y^{\mathcal{F}}@\mathcal{P} \longrightarrow y@\mathcal{M}_{=}\ ; \mathcal{M}_{<} \quad ; \mathcal{P}$ if $Ay = 0 \wedge By = 0$
Solution$_{<}$	$\mathcal{M}_{=}; \mathcal{M}_{<}; y^{\mathcal{F}}@\mathcal{P} \longrightarrow \mathcal{M}_{=} \quad ; y@\mathcal{M}_{<}\ ; [y + e_{j_1}^{\mathcal{F}\cup\{j_2,\ldots,j_l\}}, \ldots, y + e_{j_l}^{\mathcal{F}}]@\mathcal{P}$ if $Ay = 0 \wedge By < 0,\ \nexists y_0 \in \mathcal{M}_{<}\ (y_0, -By_0) \leq_{q+m_B} (y, -By)$, $\{e_{j_1}, \ldots, e_{j_l}\} = \{e_j \mid (\Psi(y, j)\ \wedge j \notin \mathcal{F})\}$ and $y + e_{j_1} \prec_y \ldots \prec_y y + e_{j_l}$
Develop	$\mathcal{M}_{=}; \mathcal{M}_{<}; y^{\mathcal{F}}@\mathcal{P} \longrightarrow \mathcal{M}_{=} \quad ; \mathcal{M}_{<} \quad ; [y + e_{j_1}^{\mathcal{F}\cup\{j_2,\ldots,j_l\}}, \ldots, y + e_{j_l}^{\mathcal{F}}]@\mathcal{P}$ if $\{e_{j_1}, \ldots, e_{j_l}\} = \{e_j \mid (\Psi(y, j)\ \wedge j \notin \mathcal{F})\}$ and $y + e_{j_1} \prec_y \ldots \prec_y y + e_{j_l}$

Fig. 2. The Stack Version of the `ACalg` Algorithm for Solving $\mathcal{C}_1$.

Wanted$_{=}$ and *Wanted*$_{<}$ depending on which list a solution goes. Thus, two corresponding rules Solution$_{=}$ and Solution$_{<}$ are designed. Solution$_{=}$ loads into $\mathcal{M}_{=}$ a solution having a null-defect. Solution$_{<}$ develops the successors of y according to the criterion Ψ, and loads y into $\mathcal{M}_{<}$ if it is not already subsumed by $\mathcal{M}_{<}$. The complete description of the stack version of `ACalg` is depicted in Figure 2. The already defined arguments for parameters not only show that `ACalg` is an instance of $\mathcal{P}$roc, but also ensure termination, soundness and completeness (see [2]).

5 Motivation of the Work

The aim of this section is to highlight the limited pruning power of the algorithm `ACalg`. As an example, we consider the following constraint system:

$$\mathcal{C}_1 \equiv \begin{pmatrix} 3x_1 - 2x_3 =^? 0 \\ -2x_1 + x_2 - 3x_3 \leq^? 0 \\ x_2 + 2x_3 - x_4 \leq^? 0 \end{pmatrix}$$

We want to point out the weak pruning of `ACalg` even in the presence of unbound variables (no explicit bounds are given). Figure 5 shows a large part of the forest generated by running the stack version of `ACalg` on $\mathcal{C}_1$. Due to lack of space, we do not visualise the tree rooted at node $(1, 1, 0, 0)$. A component y_j of a forest node $(y_1, \ldots, y_q)$ is put into a small box iff x_j is frozen. A node labelled by a non-decomposable solution is entirely boxed. Although the sub-tree rooted at

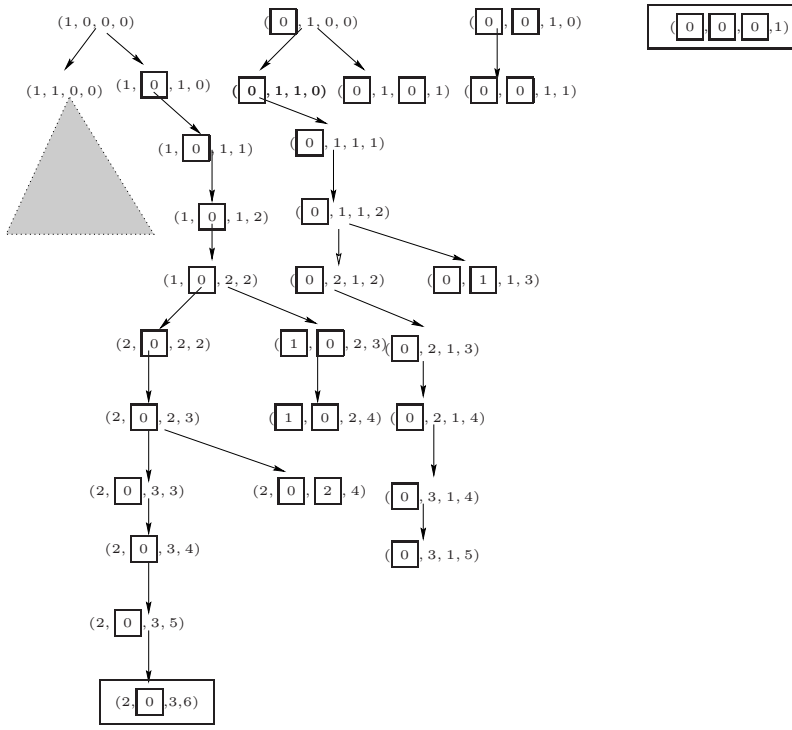


Fig. 3. A Part of the Forest Developed by `ACalg` for Solving $\mathcal{C}_1$.

$(\boxed{0}, \boxed{0}, 1, 0)$ cannot contain solutions to $\mathcal{C}_1$, `ACalg` would continue to increment the components x_3 and x_4. The subtree is failed because any *solution* must satisfy:

$$x_1 =^? 0 \ \wedge \ x_2 =^? 0 \ \wedge \ 1 \leq^? x_3 \ \wedge \ 0 \leq^? x_4 \ \wedge \ 3x_1 - 2x_3 =^? 0.$$

The equation $3x_1 - 2x_3 =^? 0$ implies $2x_3 \leq^? 3x_1$, which by the above implies that all the descendants are inconsistent. The reason of failure is that x_3 violates the upper bound imposed by x_1. However, algorithm `ACalg` cannot discover this fact. With a similar reasoning, one can see that the tree rooted at $(\boxed{0}, 1, 1, 0)$ cannot contain solutions.

Now, let us study another aspect of `ACalg`. Consider the tree rooted at $(1, \boxed{0}, 1, 0)$ which contains the solution labelled by tuple $(2, 0, 3, 6)$. Every node of that tree, which solves $\mathcal{C}_1$, must satisfy:

$$1 \leq^? x_1 \ \wedge \ x_2 =^? 0 \ \wedge \ 1 \leq^? x_3 \ \wedge \ 0 \leq^? x_4 \ \wedge \ 3x_1 - 2x_3 =^? 0$$

From equality $3x_1 - 2x_3 =^? 0$ and $1 \leq^? x_3$, we can deduce $2 \leq^? x_1$. As an immediate consequence, x_3 cannot be in $[1..2]$, since $x_3 \leq^? 2 \wedge 2 \leq^? x_1 \wedge 0 =^? 3x_1 - 2x_3$ is unsatisfiable. In sum, all nodes between $(1, \boxed{0}, 1, 0)$ and its descendants not satisfying $2 \leq^? x_1 \ \wedge \ 3 \leq^? x_3$ are inconsistent w.r.t. $\mathcal{C}_1$. These nodes are inconsistent because they violate lower bounds. We would be able to improve the efficiency of `ACalg` if we could *skip* those nodes and *jump* to the smallest descendant satisfying the lower bounds.

6 Towards the Solving Method

Section 5 motivated why derived lower and upper bounds enhance the pruning power of method `ACalg`. That is, a combination of domain constraints with the freezing mechanism of `ACalg` enables deriving stronger domain constraints. Such information can be exploited to detect inconsistencies at earlier stages of the computation. Compared to this pruning potential, the current relation *Cut* is relatively weak because the `ACalg` algorithm prunes a node only if it is greater or equal to a previous solution having a null defect or if the criterion Ψ fails. In practice, we observed that many useless search branches are blindly constructed. Thus, `ACalg` needs stronger pruning rules.

The approach we adopt consists of combining complete solving with constraint propagation, where the latter plays a central role in the design of powerful pruning rules. Our starting point is the `ACalg` algorithm. Following the design of the generic procedure, the combination is mainly achieved by extending each search node by a local store in field σ.

Adding a constraint store offers the opportunity to add explicit domain constraints to the input constraint system. Hence, we are interested in solving the following homogeneous constraint system in $\mathbb{N}^q$:

$$\mathcal{C}_1 \ \equiv \ AX =^? 0 \ \wedge \ BX \leq^? 0 \ \wedge \ X \in^? [L..U]$$

Note that each variable has a possibly infinite domain. Before continuing the discussion, let us mention that there is a naive approach to combine the `ACalg` solver with *static* upper bounds. The idea is to apply `ACalg` to $\mathcal{C}_1$ and to freeze a variable as soon as its upper bound is reached. E. Contejean [7] proposed such an approach for the case where all variable domains are finite. This approach is

considered naive because it inherits from `ACalg` the need for stronger pruning rules, and, in addition, it does not exploit domain bounds during the solving process.

6.1 Representation of the Solution Set

In general, the solution set of $\mathcal{C}_1$ is not only likely to be large but also infinite. In the case where all domains are finite, it is even necessary to enumerate all solutions. This is the reason why we are interested in a compact representation of $\mathcal{S}ol(\mathcal{C}_1)$.

In order to achieve a complete representation, we first solve a constraint system where we relax the lower bounds L:

$$\mathcal{C}_0 \;\equiv\; AX =^? 0 \;\wedge\; BX \leq^? 0 \;\wedge\; X \in^? [0..U]$$

The "why" is illustrated in the next example:

Example 1. Let $\mathcal{C}_1 \equiv x_1 - x_2 \leq^? 0 \;\wedge\; x_1 \in^? [3..6] \;\wedge\; x_2 \in^? [2..5]$. The non-decomposable solutions of $x_1 - x_2 \leq^? 0$ are $s^1 = (0, 1)$ and $s^2 = (1, 1)$. None of them solves $x_1 \in^? [3..6] \;\wedge\; x_2 \in^? [2..5]$. But, to describe the solution $s = (3, 5)$ of $\mathcal{C}_1$ one needs s^1 and s^2 since $s = 2s^1 + 3s^2$.

Let $\mathcal{E} = \{s^1, \ldots, s^e\}$ denote the set of non-decomposable solutions found when solving $\mathcal{C}_0$. In general, the set of solutions obtained by $\mathbb{N}$-linear combination $\sum_{i\in[1..e]} \lambda_i s^i$ is a superset of the solutions of the original system $\mathcal{C}_1$. The idea is to constrain the parameters λ_i sufficiently such that both solution sets coincide:

Lemma 3. *For any s in $\mathcal{S}ol(\mathcal{C}_1)$ there exist coefficients $\lambda_1, \ldots, \lambda_e \in \mathbb{N}$ such that:*

- $s = \sum_{i\in[1..e]} \lambda_i s^i$
- *The vector $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_e)$ solves the constraint system $\Delta = \bigwedge_{j\in[1..q]} \Psi_j$, where*

$$\Psi_j = \begin{cases} l_j \leq^? \Sigma_{k\in[1..e]} \lambda_k s_j^k & \text{if } u_j = \infty \\ l_j \leq^? \Sigma_{k\in[1..e]} \lambda_k s_j^k \;\wedge\; \Sigma_{k\in[1..e]} \lambda_k s_j^k \leq^? u_j & \text{if } u_j \in \mathbb{N}. \end{cases}$$

The proof is straightforward. In fact, this lemma is a generalisation of Lemma 1 in the presence of domain constraints. In the special case where $\mathcal{C}_1$ contains no domain constraints, the two lemmas coincide.

The variables of Δ are the parameters $\lambda_1, \ldots, \lambda_e$, where e is the size of $\mathcal{E} = \{s^1, \ldots, s^e\}$. Therefore, we are able to represent $\mathcal{S}ol(\mathcal{C}_1)$ by means of the expression $\Theta \equiv \bigwedge_{j\in[1..q]} x_j = \sum_{k\in[1..e]} \lambda_k s_j^k$ and an additional constraint Δ. Θ provides the general form of solutions of $\mathcal{C}_1$, and Δ specifies both necessary and sufficient conditions on $\lambda_1, \ldots, \lambda_e$ that incorporate all domain constraints. Let $\Omega = \Theta || \Delta$ be called a constrained parametric expression. It stands for the following set of tuples in $\mathbb{N}^q$:

$$\{(\sum_{k\in[1..e]} \nu_k s_1^k, \ldots, \sum_{k\in[1..e]} \nu_k s_q^k) \;\mid\; (\nu_1, \ldots, \nu_e) \in \mathcal{S}ol(\Delta)\}.$$

Then, it holds that:

Lemma 4. *The constrained parametric expression Ω is a complete representation of $\mathcal{S}ol(\mathcal{C}_1)$.*

The proof is trivial since by construction $\mathcal{S}ol(\mathcal{C}_1)$ and Ω are equivalent. Note that $\mathcal{S}ol(\mathcal{C}_1)$ is infinite (resp. empty) iff $\mathcal{S}ol(\Delta)$ is infinite (resp. empty).

Example 2 ((1) continued). The constrained parametric expression $\Omega = \Theta||\Delta$ representing $\mathcal{S}ol(\mathcal{C}_1)$ is defined by $\Theta \equiv x_1 = \lambda_1 \ \wedge \ x_2 = \lambda_1 + \lambda_2$ and $\Delta \equiv 3 \leq^? \lambda_1 \leq^? 6 \ \wedge \ 2 \leq^? \lambda_1 + \lambda_2 \leq^? 5$.

Note that in the case where we are concerned by computing *one solution* there is no need to relax lower bounds of $\mathcal{C}_1$. Moreover, it can be easily shown that we can usually reach the case of null lower bounds with the price of getting non-homogeneous system (see Example 3). As said before, the obtained system can be turned into a homogeneous one. The next example emphasises that :

Example 3 ((1) continued). To get non-null lower bounds, one can introduce two new variables z_1 and z_2, and transform $\mathcal{C}_1$ into $\mathcal{C}_0 \equiv z_1 - z_2 \leq^? -1 \ \wedge \ z_1 =^? x_1 - 3 \ \wedge \ z_2 =^? x_2 - 2 \ \wedge \ z_1 \in^? [0..3] \ \wedge \ z_2 \in^? [0..3]$.

6.2 The Notion of Local Constraint Store

As said before, `ACalg` does not make use of the field σ. Section 5 motivated the need for domains in order to detect pruning opportunities. Such information can be kept in the form of a constraint store which is local to each node.

Definition 1 (Local Constraint Store). *Let $y = (y_1, y_2, \ldots, y_q)^{\mathcal{F}}$ be a node generated by* `ACalg`*. Its local constraint store $\sigma(y)$ is defined as follows:*

$$\sigma(y) = \begin{cases} AX =^? 0 \\ BX \leq^? 0 \\ X \in^? [0..U] \\ \bigwedge_{j \notin \mathcal{F}} x_j \geq^? y_j \\ \bigwedge_{j \in \mathcal{F}} x_j =^? y_j \end{cases} \tag{1}$$

Recall that x_j denotes a variable and y_j is the j^{th} component value of the tuple $v(y)$. This definition provides us with a direct construction of the local store as a function depending only on the current node y. It is desirable to design an *incremental* construction of σ. The base case corresponds to the root node, and it holds, by definition 1, that $\sigma(0) = \mathcal{C}_0$ since $\mathcal{F}(0) = \emptyset$. That is:

$$\sigma(0) \ = \ AX =^? 0 \ \wedge \ BX \leq^? 0 \ \wedge \ X \in^? [0..U] \tag{2}$$

The general case deals with nodes generated either by rule $\mathsf{Solution}_<$ or $\mathsf{Develop}$. Let the current node be y, associated with some ordering $\prec_y$, and let $z_{j_k} = y + e_{j_k}$

be its k^{th} direct descendant. According to Figure 2 it holds that $\mathcal{F}(z_{j_k}) = \mathcal{F}(y) \cup \{j_{k+1}, \ldots, j_l\}$. Therefore, we define:

$$\sigma(z_{j_k}) = \sigma(y) \ \wedge \ \phi_{j_k} \ \textit{where}$$
$$\phi_{j_k} = \begin{pmatrix} x_{j_{k+1}} =^? y_{j_{k+1}} \ \wedge \ \ldots \ \wedge \ x_{j_l} =^? y_{j_l} \\ x_{j_k} \geq^? y_{j_k} + 1 \end{pmatrix} \tag{3}$$

It is very important to note that the tuple of z_{j_k} solves the constraints ϕ_{j_k}. This fact is at the heart of our rule soundness proofs. The next lemma follows by a structural induction on nodes.

Lemma 5. *For any node, Definition 1 is equivalent to the incremental construction inductively defined by equations (2) and (3).*

In the following sub-section we show that taking the identity for operation ρ as done in `ACalg` is not the best choice that one can imagine.

6.3 Local Simplification versus Bound Propagation

Using constraint propagation [9,12], we can strengthen the information in the local constraint store to detect inconsistencies earlier. The idea of this technique is that a constraint enforces the elimination of inconsistent value combinations. We only need bound propagation, denoted by `BProp`, which eliminates inconsistent domain bounds but not values inside domains. `BProp` is a fix-point operation with signature *Store* $\rightarrow$ *Store* that ends when changes to variable bounds cease to occur. Let σ_1, σ_2 be two constraints of *Store*. Then, $\sigma_1 \vdash \sigma_2$ means that σ_1 operationally entails σ_2, whereas $\sigma_1 \Longrightarrow \sigma_2$ means that σ_1 logically entails σ_2. σ entails failure (*ie.*, $\sigma \vdash \mathbb{F}$) if some variable domain is empty. We shall use $inf_\sigma(x)$ for the lower bound and $sup_\sigma(x)$ for the upper bound of x in σ.

In general, `BProp` enforces local consistency, but not global consistency, which is reflected by the law that $\sigma_1 \vdash \sigma_2$ implies $\sigma_1 \Longrightarrow \sigma_2$. Our results use the well-known fact saying that `BProp` is solution preserving: $\texttt{BProp}(\sigma) \Longleftrightarrow \sigma$.

We now choose `BProp` for the parameter ρ of the generic solving procedure. That is, for any node $y = (v, \sigma)$ we take:

$$\rho(y) = (v, \texttt{BProp}(\sigma)) \tag{4}$$

For a first look, it may seem that the presence of infinite domains in σ could lead to non-termination of `BProp`. However, this is not the case. Indeed, from the fact that, on one side, our method is aiming at finding non-decomposable solutions and, on the other side, possibly large uniform theoretical bounds for non-decomposable solutions exist (see *e.g.* [2,10]) it follows that the latter bounds on components can simulate infinite domains.

6.4 Encoding Freezing by Constraints

The freezing mechanism can be driven by the constraints of the local store. Two requirements need to be met:

(R_1) A node y needs to *tell* $\sigma(y)$ which components become frozen.
(R_2) A node needs to find all already frozen components by asking $\sigma(y)$.

The incremental construction of the constraint store already meets requirement (R_1), since for a node $y = (v, \sigma)$, σ accumulates constraints $x_j =^? v_j$ if j in $\mathcal{F}$. In order to meet the second requirement, it suffices to test whether $\sigma(y)$ contains some constraint $x_j =^? y_j$. Trivially, if x_j was previously frozen then σ will contain a unary constraint $x_j =^? y_j$.

Driving freezing by the local store is stronger than using $\mathcal{F}$ since a variable will be considered "frozen" either because it was previously frozen, or because it is constrained by freezing other variables. Beyond this role, the local store is the heart of a more powerful pruning engine that is discussed in the next section.

7 Stronger Pruning Rules

Before each rule application, the search engine applies ρ to the current node, and hence `BProp` computes stronger upper and lower bounds. We discuss how these improved bounds can be exploited locally to prune inconsistent subtrees early, and to skip parts of the search tree containing inconsistent nodes. In addition, it is shown how search heuristics can be elegantly embedded into the method.

Let $y = (v, \sigma)$ be the current node (*ie.*, the top of stack $\mathcal{P}$). For any two nodes y and y', we write $y \hookrightarrow y'$ if y' is a direct successor of y developed by `ACalg`. Furthermore, $\overset{*}{\hookrightarrow}$ is the transitive and reflexive closure of relation $\hookrightarrow$. y' is called a descendant of y if $y \overset{*}{\hookrightarrow} y'$.

7.1 Pruning with Partial Information

The local stores are monotonic w.r.t. to the relation $\hookrightarrow$. In addition, the failure of the store means that the corresponding node (*ie.*, its tuple) is not a solution.

Lemma 6. *Let y be a given node developed by* `ACalg`. *Let us denote by $u_0 = 0 \hookrightarrow u_1 \hookrightarrow \ldots \hookrightarrow u_k = y$ ($1 \leq k$) the unique path from the root 0 to y. The following properties hold:*

$\mathtt{P}_1$) $\forall i \in [0..k-1]$: $\sigma(u_{i+1}) \Longrightarrow \sigma(u_i)$
$\mathtt{P}_2$) *If $\sigma(y) \Longrightarrow \mathbb{F}$ then y is not a solution of $\mathcal{C}_0$.*

Detecting a local failure $\sigma(y) \vdash \mathbb{F}$ does not only indicate the inconsistency of y, but also the inconsistency of all nodes of the tree rooted at y. This is cast into a new pruning rule:

$$\mathsf{Prune}\ \mathcal{M}_=; \mathcal{M}_<; y@\mathcal{P} \longrightarrow \quad \mathcal{M}_=; \mathcal{M}_<; \mathcal{P} \quad \textit{if} \quad \sigma(y) \vdash \mathbb{F}$$

Recall that $\vdash$ implies $\Longrightarrow$. Prune preserves termination of `ACalg` since the latter terminates without Prune and $\sigma(y) \vdash \mathbb{F}$ can be decided in finitely many steps. In addition, Prune preserves solutions.

Theorem 1. *Let y be a node developed by* ACalg *algorithm such that $\sigma(y) \vdash \mathbb{F}$. Then, there is no descendant of y which is a solution of $\mathcal{C}_0$.*

Since Prune does not prune solutions then necessarily it does not prune any non-decomposable solution. That is, it preserves the completeness of ACalg. Prune is applied before ACalg rules. The reader is invited to observe that Prune is not subsumed by Leaf and is more powerful than the latter rule. In particular, when $\mathcal{C}_0$ has no solution with null-defect (*ie.*, satisfying $BX =^? 0$), Leaf will never be applicable, but Prune may be potentially applicable. The example of Section 5 exhibits this clearly.

7.2 Skipping Inconsistent Sub-Spaces

We now study the case where a given node does not satisfy the current constraint store σ, but σ is not failed. This can happen after some lower bounds have changed. Suppose that σ' is the local store of y before propagation, that is, $\sigma = \texttt{BProp}(\sigma')$. The set of variables whose lower bounds changed by the execution of BProp is defined by the index set:

$$\mathcal{K} = \{k_1, \ldots, k_p\} = \{k_j \mid j \in [1..p],\ inf_{\sigma'}(x_{k_j}) < inf_{\sigma}(x_{k_j})\}$$

The following characterises those descendant nodes z of y developed by ACalg that do not satisfy the derived lower bounds:

$$y \overset{*}{\hookrightarrow} z \ \wedge\ \exists j \in \mathcal{K},\ z_j < inf_{\sigma(y)}(x_j) \tag{5}$$

As already observed in Section 5, those nodes cannot solve $\mathcal{C}_0$, which is formalised by the next lemma.

Lemma 7. *Let y be a node developed by algorithm* ACalg *satisfying (5). For any node z such that $y \overset{*}{\hookrightarrow} z$: if z satisfies (5), then z does not solve $\mathcal{C}_0$.*

This lemma suggests to skip those nodes that satisfy condition (5). Intuitively, one wants to push lower bounds into components. Furthermore, we need to adjust the constraint store such that it reflects the skip. Let $\alpha = ((\alpha_1, \ldots, \alpha_q), \sigma(\alpha))$ be the node depending on y which is constructed as follows:

$$\forall j \in [1..q],\ \alpha_j = \begin{pmatrix} inf_\sigma(x_j) & \text{if } j \in \mathcal{K} \\ y_j & \text{otherwise} \end{pmatrix}$$

$$\sigma(\alpha) = \begin{pmatrix} \sigma(y) \ \wedge \\ x_{k_1} \geq^? \alpha_{k_1} \ \wedge\ x_{k_2} \geq^? \alpha_{k_2} \ldots \ \wedge\ x_{k_p} \geq^? \alpha_{k_p} \end{pmatrix} \quad \text{if } \mathcal{K} = \{k_1, \ldots, k_p\}$$

Since α is the smallest conflict free node after y, it is taken as the direct descendant of y. We cast this into the following pruning rule:

$$\textsf{Forward}\ \mathcal{M}_=; \mathcal{M}_<; y@\mathcal{P} \longrightarrow \quad \mathcal{M}_=; \mathcal{M}_<; \alpha(y)@\mathcal{P} \quad \text{if } \mathcal{K} \neq \emptyset \text{ and } \sigma(y) \nvdash \mathbb{F}$$

This *deterministic* rule has to be tried before the ACalg rules. Note that Forward preserves the monotonicity property $\mathtt{P}_1$ of Lemma 6 since $\sigma(\alpha)$ entails $\sigma(y)$.

Furthermore, by Lemma 7 it follows that Forward skips only *non-solutions* nodes. This means that the completeness of `ACalg` is preserved. Since $\alpha(\alpha(y)) = \alpha(y)$, it suffices to apply Forward at most once. This together with the fact that no infinite path can be developed from α (see [2]) shows that Forward preserves termination. In general, we can expect that any application of Forward shortens a search path. Furthermore, branching on non-solution nodes can be delayed, thus avoiding multiple unfolding blind paths.

7.3 Embedding Dynamic Variable Orderings

The selection order of variables in tree-search algorithms is known to have substantial influence on the search behaviour and efficiency. Using a variable ordering one can try to obtain shorter search paths. In general, orderings are highly problem specific, may be with exception of the *first-fail* strategy (FF for short) which selects at each search step the variable with the smallest domain. Variable orderings can be classified into static orderings which are fixed prior to search, and the more powerful dynamic ones which are re-evaluated at each search step. F. Bacchus & P. Van Run [4] showed that it is not immediately obvious how a DVO can be embedded in an arbitrary tree-search algorithm.

In our context, the ordering $\prec_y$ which determines the freezing pattern is clearly a variable ordering since it determines the order of descendants on the stack. The order in the current implementation of `ACalg` is node-independent and therefore static. However, the design of `ACalg` theoretically admits any local ordering $\prec_y$, and hence DVOs. Unfortunately, it was not clear so far how to maintain information required to apply a DVO.

Our solution to this problem is surprisingly simple: the constraint store offers the means to maintain and derive the necessary information. For instance, in order to implement the FF strategy we can exploit the domain bounds. This means also that those existing DVO strategies which are based on the notion of constraint store can be easily adapted to our method.

Technically, we define a DVO by means of a function which weights a variable x in a local constraint store. The evaluation depends on the domain of x and the constraint network topology. The weights can be used to determine a total order $\prec_y$ on the variables $x_{j_1}, \ldots, x_{j_l}$ which via $\sigma(y)$ depends on the current node y. This order is lifted to the component level by: $\rho(y + e_{j_i}) \prec_y \rho(y + e_{j_k})$ iff $x_{j_i} \prec_y x_{j_k}$. Among the descendants, the one corresponding to the variable with the lowest weight will be on top of stack $\mathcal{P}$. As defined in Figure 2, this descendant freezes all components belonging to variables of higher weight: $y + e_{j_1}^{\mathcal{F}'}$ where $\mathcal{F}' = \mathcal{F} \cup \{j_2, \ldots, j_l\}$. Thus, $y + e_{j_1}^{\mathcal{F}'}$ becomes the *most constrained* descendant.

The extended method, previously detailed in Sections 6 and 7 preserves desirable properties of `ACalg`. We cast this in the following main theorem:

Theorem 2 (Termination, Soundness and Completeness). *The obtained method preserves* `ACalg` *termination. Furthermore, it is sound and provides a complete representation of the solution set of* $\mathcal{C}_1$ *according to the definition mentioned by Lemma 4.*

In addition, it can be shown that the obtained method is still an instance of the generic solving procedure $\mathcal{P}$roc presented in Section 3.

8 Implementation

In order to investigate the effect of pruning, we have implemented the solver in `Java`. The system, called `ALalg`, solves non-homogeneous linear Diophantine constraint systems. Its architecture basically consists of three main classes: `Solver` which coordinates the search, `Node` which encodes search nodes and `PropagationNode` which extends `Node` by adding local store and propagation methods. We kept the design flexible by delegating the search strategy into a separate class: new strategies can be implemented without modifying the `Node` class methods. `ALalg` is used via a graphical user interface offering different views and navigation on the search tree. Colours and statistics tell additional informations about nodes and applied rules. An alternative view presents the detailed internal state of the selected search node. The bound propagation module is based on computing infimum/supremum terms (see [9,12] for details).

Input	N	`naiveACalg`	`ALalg`		
			LR	RL	FF
$\mathcal{C}_1$	1	8	6	4	4
$\mathcal{C}_2$	8	15	14	17	17
$\mathcal{C}_3$	1	36	16	6	6
$\mathcal{C}_4$	0	204	52	6	6
$\mathcal{C}_5$	48	237	48	80	80
$\mathcal{C}_6$	278	674	620	457	291
$\mathcal{C}_7$	45	112	109	80	80
$\mathcal{C}_8$	57	443	161	113	113
$\mathcal{C}_9$	15	157	23	19	19

Fig. 4. Experimental results.

The central aim of our implementation is to test the impact of propagation based pruning and heuristics in complete solving of Diophantine systems in terms of *search tree size*, but not in terms of run-times. This is why we have compared `ALalg` under a specified search strategy and a `C` implementation of `naiveACalg`.

The allowed search strategies are LR (left to right), RL (right to left) and FF (first fail). Our evaluation considers the constraints depicted in the Appendix and is based on the size of the search tree. Figure 4, where N is the cardinality of the solution set basis, shows that our solver clearly outperforms `naiveACalg`. Currently, our implementation is not well-suited to evaluate efficiency in terms of run-times because `Java` is currently not competitive compared to `C`-like compiled programming languages.

9 Conclusion

This work could be seen as an attempt in closing the gap between existing mathematical solving methods and consistency techniques widely and successfully used in the CLP paradigm [13,12]. Our extension of `ACalg` facilitates and enables the performing of constraint based reasoning and integration of DVO heuristics which reduce significantly the size of the search space. Because of our `Java` implementation, we did not evaluate the impact of possible overheads which may be caused by the call of propagation at *each* search node. In the near future, we would like to investigate such a concern by designing a `C` implementation which would make meaningful a comparison of the two methods in terms of run-times.

In addition, this work offers several interesting followups for future research. One is constraint entailment. Indeed, the compact representation of a (possibly) large or infinite solution set should be deeply exploited in order to design new entailment techniques. Finally, it seems worth-while to record some "simple" local stores and to prune a node as soon as its store is entailed by a previously recorded one. We would like to emphasis such issue since we frequently observed *regularities* in the visualised search tree which indicate further pruning opportunities. Currently, we are seeking for subsumption rules which capture such regularities.

References

1. F. Ajili and E. Contejean. Complete solving of linear diophantine equations and inequations without adding variables. In Montanari and Rossi [14], pages 1–17. 463, 464, 467
2. F. Ajili and E. Contejean. Avoiding slack variables in the solving of linear diophantine equations and inequations. *Theoretical Computer Science*, 173(1):183–208, February 1997. 463, 464, 465, 467, 468, 473, 476
3. Farid Ajili. *Contraintes Diophantiennes Linéaires : résolution et coopération inter-résolveurs*. PhD thesis, Université Henri Poincaré-Nancy I, May 1998. 465
4. F. Bacchus and P. Van Run. Dynamic variable ordering in CSPs. In Montanari and Rossi [14], pages 258–275. 476
5. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, Lecture Notes in Computer Science, 1996. 463

6. M. Clausen and A. Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1 & 2):201–216, 1989. Special issue on unification. Part two. 463, 464, 467
7. E. Contejean. Solving linear diophantine constraints incrementally. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 532–549, Budapest, Hungary, 1993. The MIT Press. 470
8. E. Contejean and H. Devie. An efficient algorithm for solving systems of diophantine equations. *Information and Computation*, 113(1):143–172, August 1994. 463, 464, 465, 467
9. D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press. 464, 473, 477
10. E. Domenjoud. Solving systems of linear diophantine equations: An algebraic approach. In A. Tarlecki, editor, *Proc. 16th Inter. Symp. on Mathem. Foundations of Computer Science, Kazimierz Dolny (Poland)*, volume 520 of *Lecture Notes in Computer Science*, pages 141–150. Springer-Verlag, 1991. 463, 473
11. E. Domenjoud and A. P. Tomás. From Elliott-MacMahon to an algorithm for general linear constraints on naturals. In Montanari and Rossi [14], pages 18–35. 463
12. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989. 464, 473, 477, 478
13. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 & 20:503–582, May 1994. 464, 478
14. U. Montanari and F. Rossi, editors. *Proceedings 1st International Conference on Principles and Practice of Constraint Programming, Cassis (France)*, volume 976 of *Lecture Notes in Computer Science*. Springer Verlag, September 1995. 478, 479
15. J.-F. Romeuf. A polynomial algorithm for solving systems of two linear diophantine equations. Technical report, Laboratoire d'Informatique de Rouen (France) and LITP, 1989. 463

Appendix

$$\mathcal{C}_1 = \begin{bmatrix} 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} X =^? \begin{bmatrix} 0 \\ 0 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..3]} x_i \in^? [0..\infty] \quad \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} X \leq^? \begin{bmatrix} 0 \end{bmatrix}$$

$$\mathcal{C}_2 = \begin{bmatrix} 3 & 2 & -1 & -2 \end{bmatrix} X \leq^? \begin{bmatrix} 0 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..4]} x_i \in^? [0..\infty]$$

$$\mathcal{C}_3 = \begin{bmatrix} 3 & 0 & -2 \\ -2 & 1 & -1 \\ 1 & -2 & 4 \end{bmatrix} X \leq^? \begin{bmatrix} 0 \\ 0 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..3]} x_i \in^? [0..\infty]$$

$$\mathcal{C}_4 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 12 & -36 \\ 7 & -5 \end{bmatrix} X \leq^? \begin{bmatrix} 150 \\ -10 \\ 115 \\ -6 \end{bmatrix} \ \wedge \ x_1 \in^? [0..16] \ \wedge \ x_2 \in^? [0..8]$$

$$\mathcal{C}_5 = \left[\begin{array}{ccc} 6 & -1 & 0 \\ 2 & -1 & -7 \\ 0 & -10 & 3 \\ 1 & 1 & 1 \end{array}\right] X \leq^? \left[\begin{array}{c} 9 \\ -3 \\ 9 \\ 7 \end{array}\right] \ \wedge \ x_1 \in^? [0..3] \ \wedge \ x_2 \in^? [0..5] \ \wedge \ x_3 \in^? [0..9]$$

$$\mathcal{C}_6 = \left[\begin{array}{ccc} 2 & -1 & -7 \\ 1 & -1 & 3 \\ 1 & 1 & 2 \end{array}\right] X \leq^? \left[\begin{array}{c} -3 \\ 5 \\ 20 \end{array}\right] \ \wedge \ x_1 \in^? ![0..\infty] \ \wedge \ x_2 \in^? [0..9] \ \wedge \ x_3 \in^? [0..\infty]$$

$$\mathcal{C}_7 = \begin{bmatrix} 6 & -1 & 0 \\ 2 & -1 & -7 \\ 0 & 1 & 3 \end{bmatrix} X \leq^? \begin{bmatrix} 9 \\ -3 \\ 9 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..3]} x_i \in^? [0..\infty]$$

$$\mathcal{C}_8 = \begin{bmatrix} 2 & -3 & 1 & 7 & -2 \\ -4 & 0 & 6 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} X \leq^? \begin{bmatrix} -2 \\ 5 \\ 5 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..5]} x_i \in^? [0..\infty]$$

$$\mathcal{C}_9 = \begin{bmatrix} 3 & 0 & -2 & 0 \end{bmatrix} X =^? 0 \quad \begin{bmatrix} -2 & 1 & -3 & 0 \\ 0 & 1 & 2 & -1 \end{bmatrix} X \leq^? \begin{bmatrix} 0 \\ 0 \end{bmatrix} \ \wedge \ \bigwedge_{i\in[1..3]} x_i \in^? [0..\infty]$$

Approaches to the Incremental Detection of Implicit Equalities with the Revised Simplex Method

Philippe Refalo*

ILOG, S.A.
9 rue de Verdun - BP 85, 94253 Gentilly, France
refalo@ilog.fr

Abstract. This paper deals with the incremental detection of implicit equalities using the revised simplex method. This algorithm is more efficient and more suitable to practical problems than the tableau method usually applied in constraint logic programming. We describe and discuss the adaptation to the revised simplex of three approaches: the CLP($\mathcal{R}$), the Prolog III, and the quasi-dual one. All of these have been integrated into the constraint logic programming language Athena based on a revised simplex method over exact-precision rationals. This system is used to compare these methods on a set of typical CLP problems over linear constraints.

1 Introduction

Detecting implicit equalities is closely related to solving linear constraints in constraint logic programming (CLP). An implicit equality is an inequation $a_1x_1 + \cdots + a_nx_n \leq a_0$ from a set of constraints S that can be converted into the equation $a_1x_1 + \cdots + a_nx_n = a_0$ without changing the set of solutions of S. For instance in the set $\{x_1 + x_2 \leq 2, 1 \leq x_1, 1 \leq x_2\}$ all inequations are implicit equalities. Indeed this set can be rewritten to $\{x_1 + x_2 = 2, x_1 = 1, x_2 = 1\}$ without changing its set of solutions.

Detecting implicit equalities is one of the fundamental operations in solving generalized linear constraints [15] that include disequations ($\neq$) and strict inequations ($<$). This method for solving constraints is integrated into languages like Prolog III [6] and Prolog IV [8].

This detection enables the discovery of all variables fixed to a single value. This is applied to the delayed solving of non linear constraints [7]. In languages like CLP($\mathcal{R}$) [12] or Prolog III [6], solving the non linear constraint $2xy + 3z + t = 0$ is delayed until one of the variables x or y is fixed by the linear constraint solver. For instance, if x becomes fixed to 3, the constraint $6y + 3z + t = 0$ is then added to the linear constraint solver.

* A part of this work was done when the author was in Laboratoire d'informatique de Marseille, Faculte des sciences de Luminy, 13009 Marseille- France

C. Palamidessi, H. Glaser, and K. Meinke (Eds.): PLILP'98, LNCS 1490, pp. 481–497, 1998.

Detecting implicit equalities is also useful in cooperating constraint solvers which associate a constraint propagation based solver (on intervals or finite domains) with a linear constraint solver. The former gives bounds on variables to the latter while the latter maintains a rational solution of linear constraints and gives the former fixed variables [3,8,1]. For instance, to solve the system $S = \{x-y+z \leq 0,\ y-z \leq 0,\ y \leq 2x+1,\ x \in \{0,1,2\}, y \in \{0,1,2\}, z \in \{0,1,2\}\}$, it is given to a finite domain constraint solver and the set $R = \{x - y + z \leq 0,\ y - z \leq 0,\ y \leq 2x + 1,\ 0 \leq x \leq 2,\ 0 \leq y \leq 2,\ 0 \leq z \leq 2\}$ is given to a linear constraint solver. The latter detect that the inequation $0 \leq x$ is an implicit equality. Consequently x is fixed to 0. With this new bound on x the finite domain solver can reduce the domain of y and z to $\{0,1\}$. This inference is not possible with a standard finite domain constraint solver alone.

Linear constraint solving uses mainly a Gaussian elimination method (for equations and disequations) and a simplex method (for inequations[1]). Implicit equalities are incrementally detected by extensions of the simplex method. Among the first methods developed for this purpose, are the `Prolog III` method [9,6], the `CLP(`$\mathcal{R}$`)` method [18,12], and the `CHIP` method [20], which is now also used in `Prolog IV` [2]. These methods are implemented with a tableau simplex algorithm over non-negative variables since that method is easier to extend to incremental solving of linear constraints.

This work deals with the incremental detection of implicit equalities with the revised simplex method. This method is widely used for solving practical linear optimization problems [5,4]. In comparison with the tableau simplex method, the revised simplex method has the advantages of being much more efficient (many optimizations are possible), of being more numerically stable (accuracy of computations can be controlled), and of using less memory (the rate of non-zero coefficients increases rapidly with the tableau simplex but not with the revised simplex).

The revised simplex method is applied to solving linear constraints in new generation CLP languages like `CLAIRE` [3] or `Athena` [17,16]. Detection of implicit equalities is then based on the quasi-dual formulation of a system of inequations introduced by Lassez [13].

In this article we describe how to incrementally detect the presence of implicit equalities. We also detail adaptations to the revised simplex of the `Prolog III` method, the `CLP(`$\mathcal{R}$`)` method, and the quasi-dual based method. The method using the lexicographic form [20] is not detailed here since it requires the simplex tableau to be applied. For each method, we provide the main theoretical results without providing proofs. They can be found in references which are given instead. We also provide the different algorithms that identify all implicit equalities.

These algorithms have been integrated to the CLP language `Athena`. This makes quantitative comparisons possible. The linear constraint solver of `Athena` is based on a dynamic revised simplex method over exact-precision rationals [17]. It includes advanced linear programming techniques such as LU factorization and

[1] A strict inequation is usually decomposed to an equation and a disequation

smart pivoting techniques. This system is used to compare these approaches on a set of typical CLP problems over linear constraints.

The rest of this paper is organized as follows. Definitions and notations are given in Section 2. The solved form for linear constraints and the revised simplex method are presented in Section 3. Section 4 describes incremental linear constraint solving and detection of implicit equalities. Section 5 details the approach of `CLP(`$\mathcal{R}$`)`. The approach of `Prolog III` is described in Section 6, and the quasi-dual method is described in Section 7. Details about implementation and results from the practical comparison are given in Section 8.

2 Definitions and Notations

Let $\mathcal{R}$ be the set of real numbers and $\overline{\mathcal{R}} = \mathcal{R} \cup \{-\infty, +\infty\}$. Let V be an infinite and countable set of variables. A *linear term* is an expression α, αx or $t_1 + t_2$ where α is a real number, x is a variable from V and t_1 and t_2 are linear terms. A *linear constraint* is an expression $t_1 = t_2$ or $t_1 \leq t_2$ where t_1 and t_2 are linear terms. These constraints are called respectively *equation* and *inequation*. A *constraint system* is a finite set of constraints. A variable has an *occurrence* in a term or constraint if it appears with a non zero coefficient in the term or constraint. Let S be a system of inequations. The system noted $S_=$ is composed of constraints of S where the symbol $\leq$ is replaced by $=$.

An *assignment* is a mapping from set V to set $\mathcal{R}$ that assigns a value to each variable of V. An assignment can be extended naturally to a mapping from the set of terms to $\mathcal{R}$. A *solution* of a constraint is an assignment σ such that σ is a solution of the constraint $t_1 \diamond t_2$ if $\sigma(t_1) \diamond \sigma(t_2)$ where $\diamond \in \{=, \leq\}$. An assignment is a solution of a constraint system if it is a solution of each constraint of the system. A system having at least one solution is *solvable*; otherwise it is *unsolvable*. A variable is *fixed* for a constraint system S if every solution of the system assigns the same value to the variable. An inequation $t_1 \leq t_2$ from a constraint system S is an *implicit equality* if every solution of S is a solution of $t_1 = t_2$.

In the following, matrix notations are used. Let A be an $m \times n$ matrix. The expression A_{i*} represents the i^{th} row of A, the expression A_{*j} represents the j^{th} column of A. Let $P \subset \{1, \ldots, m\}$, the expression A_P represents a submatrix of A whose columns are columns of A whose indexes are in P. The inverse of a square matrix M is denoted by M^{-1}.

If e is a row vector of n real numbers, x a column vector of n variables, b a column vector of m real numbers and β a real number, the expression $ex \leq \beta$ represents an inequality and the expression $Ax = b$ represents a system of equations. We implicitly assume compatibility of the sizes of matrices and vectors when they are multiplied. Matrices and sets are denoted by upper case letters, vectors by lower case letters and real numbers by Greek symbols.

3 Solved Forms and Revised Simplex

3.1 Tableau Solved Form

The tableau solved form is composed of a set of equations and a set of inequations that bound the value of variables. A system of linear constraints in tableau solved form is at first in *basic form*:

$$S = \begin{cases} x_B = Mx_L + M'x_U + b \\ l \leq x \leq u \end{cases} \tag{1}$$

where x is a vector of variables, b a column-vector of reals, M and M' are matrices of reals, l and u are vectors of elements of $\overline{\mathcal{R}}$ and where B, L and U are disjoint sets of indexes such that $l_i \neq -\infty$, for $i \in L$ and $u_i \neq +\infty$, for $i \in U$. The set of indexes B is called the *basis* of the system. Variables from vector x_B are called *basic variables*. Variables from vectors x_L and x_U are called *non-basic variables*. The values l_i and u_i are respectively the *lower bound* and the *upper bound* of the variable x_i. For each basic form S there exists a unique assignment σ_S such that

$$\sigma_S(x_i) = \begin{cases} l_i & \text{if } i \in L \\ u_i & \text{if } i \in U \\ \sigma_S(t) & \text{if } i \in B \text{ and } x_i \text{ is basic in } x_i = t \end{cases}$$

When this assignment is also a solution of S the system is in *tableau solved form* and σ_S is called the *basic solution* of S. It is well known that any solvable system of linear equations and inequations can be mapped to tableau solved form [5].

3.2 Revised Solved Form

The revised solved form is the basis for the revised simplex method. The main difference from the tableau solved form is the basic variables that are not eliminated in the equations. However eliminating basic variables gives a system in tableau solved form. A system in *revised solved form* is a system

$$S' = \begin{cases} A_B x_B + A_L x_L + A_U x_U = d \\ l \leq x \leq u \end{cases} \tag{2}$$

where A_B is a non-singular square matrix and such that the system

$$\mathcal{T}(S') = \begin{cases} x_B = -A_B^{-1} A_L x_L - A_B^{-1} A_U x_U + A_B^{-1} d) \\ l \leq x \leq u \end{cases}$$

where basic variables are eliminated, is in solved form. The matrix A_B of basic variables coefficients is called the *basic matrix* of system S'.

A *regular* system is a system in revised solved form that does not contain implicit equalities. The purpose of this article is to solve and maintain incrementally regular systems.

3.3 The Revised Simplex Method

The revised simplex method considers a linear program of the form

$$\min\{ax_L + a'x_U \mid S\}$$

The system S is in revised solved form and the term $ax_L + a'x_U$ is an objective function where basic variables have been eliminated. It computes a *solved* linear program (i.e. such that $a_i > 0$ and $a'_i < 0$). The basic solution of S is then an optimal solution of the linear program (see [5]).

The revised simplex method first chooses an *entering variable* x_k such that $a_i < 0$ or $a'_i > 0$. A variation of the value $\sigma_S(x_k)$ decreases the value of the objective function.

The method next chooses a *leaving variable* from basic variables in order to make room for the variable x_k entering the basis and to change the value of x_k. To choose the leaving variable, the vector d of coefficients of x_k in the system $\mathcal{T}(S)$ is computed:

$$d = -A_B^{-1} A_{*k} \tag{3}$$

This vector is used to obtain the variation α_i of x_k (in absolute value) allowed by the bounds of the basic variable x_i and by the equation where x_i is basic. The greatest possible variation for x_k is $\alpha_r = \min \alpha_i$. When $\alpha_r = +\infty$ the algorithm stops. The variation of x_k is unbounded and thus the linear program is also unbounded.

In other cases, the system and the objective function are pivoted over the entering variable x_k and the leaving variable x_r. The variable x_r reaches one of its bounds and its index is added to one of the sets L or U. The index of variable x_k leaves the set where it appeared (L or U) and is introduced in the set B. Let B', L' and U' be the new set of indexes after pivoting. The new basis matrix is $A_{B'}$ and the new objective function $fx_{L'} + f'x_{U'}$ where the basic variables are eliminated, is computed as follows:

$$\begin{aligned} f &= e_{L'} - e_{B'} A_{B'}^{-1}\ A_{L'} \\ f' &= e_{U'} - e_B A_{B'}^{-1}\ A_{U'} \end{aligned} \tag{4}$$

It may happen after a pivot that the value of the objective function for the basic solution has not changed. This is the case if $\alpha_r = 0$. The pivot is then *degenerated*.

The efficiency of the revised simplex method hinges around the representation of the matrix A_B^{-1} and the update of this matrix during a pivot. Some of the numerous approaches for this are summarized in [5]. The representation used in `Athena` is an LU factorization of the basis matrix and the update is based on matrix accumulation [17,16].

4 Linear Constraint Solving

This section briefly presents some fundamental results for incremental solving of equation and inequations and the detection of implicit equalities. References containing more details are provided.

Linear constraint solving in constraint programming must be incremental. From a regular system S, and a constraint c, the basic operation is to compute a regular system S' equivalent to $S \cup \{c\}$.

4.1 Determining Solvability and the Presence of Implicit Equalities

Assume that the constraint c is an inequation $ex \leq \beta$. Determining the solvability of the system $S \cup \{c\}$ and the presence of implicit equalities in this system amounts to solving the linear program

$$\gamma = \min\{ex \mid S\}$$

with, for instance, the revised simplex method. The following three cases have to be considered:

- if $\gamma > \beta$ then $S \cup \{c\}$ is unsolvable ;
- if $\gamma < \beta$ then $S \cup \{c\}$ is solvable and does not contain implicit equalities;
- if $\gamma = \beta$ then $S \cup \{c\}$ is solvable and contains implicit equalities.

The absence of implicit equalities when $\gamma < \beta$ is proven in [15]. This fundamental result leads to efficient incremental methods for detecting implicit equalities. In other words, solving a system that does not contain any implicit equalities does not require anything more than searching for a solution of this system.

Observe that when $\gamma = \beta$, the inequation c is an implicit equality itself. Moreover, it is not the only one in the system $S \cup \{c\}$. Note also that iterations of the simplex method can terminate as soon as a solution is reached that gives the objective function a value strictly inferior to β.

If the constraint c is an equation $ex = \beta$, adding c is equivalent to adding the two inequations $ex \leq \beta$ and $\beta \leq ex$. The basic solution σ_S can often discard one of these two inequations. Indeed, if $\sigma_S(ex) < \beta$, it is sufficient to search for a solution of the inequation $\beta \leq ex$. Conversely, if $\sigma_S(ex) > \beta$, it is sufficient to search for a solution of the inequation $ex \leq \beta$. When $\sigma_S(ex) = \beta$, both inequations must be considered since each of them can be an implicit equality. However, since S does not contain any implicit equalities, if $ex \leq \beta$ is an implicit equality then the inequation $\beta \leq ex$ is not an implicit equality and vice versa.

The constraint is added to the system only if $\gamma < \beta$ (we will see later that it is not necessary to add it when $\gamma = \beta$). This problem have been studied for some time with the tableau solved form [12,20,11] and more recently with the revised solved form [17,16,3]. We will not go into further details herein since the purpose of this work is to describe the incremental computation of a regular system.

4.2 Determining All Implicit Equalities

Once the presence of implicit equalities is proven, the next step is to determine all implicit equalities of $S' = S \cup \{c\}$ and to replace them by equations to obtain a regular system.

A naive approach is to solve a linear program for each inequation $\delta x_i \leq \alpha$ of the system S' to decide if it is an implicit equality or not. In [19] Telgen proposes some improvements to this method. He gives some syntactic criteria to identify some implicit equalities in a tableau solved form. For each system obtained by the application of the tableau simplex, the criteria identify additional implicit equalities. Moreover, the basic solution σ_S that is associated to each system is used to discard any inequation verifying $\sigma_S(\delta x_i) < \alpha$ (that is obviously not an implicit equality). It remains that in the worst case, this approach can solve n linear programs for n inequations in the system.

An ideal method would solve a single linear program for each implicit equality found. Most of the incremental approaches that are described herein are very close to the ideal method and sometimes better.

Each of the incremental methods considers first the solved linear program proving the presence of implicit equalities in the system $S \cup \{ex \diamond \beta\}$ where $\diamond \in \{=, \leq\}$:

$$\min\{ax_L + a'x_U \mid S\} \tag{5}$$

This linear program contains informations for determining with certainty a first non empty set of implicit equalities. Let $l \leq x \leq u$ be the set of inequations of system S. The *restrictive inequations* of this linear program are the inequations of S that directly restrict the decrease of the objective function. They are characterized by the set

$$R = \{l_i \leq x_i \mid a_i \neq 0\} \cup \{x_i \leq u_i \mid a'_i \neq 0\}$$

As a fundamental result, each restrictive inequation of the solved linear program is an implicit equality in the system $S \cup \{ex \leq \beta\}$ (see [16]). This result is applied in most of the implementations of CLP languages over linear constraints.

It can be proved that the systems $S \cup \{ex \leq \beta\}$, $S \cup \{ex = \beta\}$ and $S \cup R_=$ are all equivalent (see [16]). Consequently, adding the constraint $ex \leq \beta$ or $ex = \beta$ amounts to fixing the variables appearing in $R_=$ to one of their bounds.

Each algorithm presented in the following considers a regular system S and a constraint $ax_L + a'x_U \leq \beta$ such that the linear program $\min\{ax_L + a'x_U \mid S\}$ is solved and $\sigma_S(ax_L + a'x_U) = \beta$. It returns a regular system S' equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$. It always detects implicit equalities as restrictive inequations of linear programs that are successively solved. The final system is thus built step by step by replacing each variable fixed to a value by this value.

5 Adaptation of the CLP($\mathcal{R}$) Method

The method described here is an adaptation of the incremental method developed in [18]. It has been integrated to the CLP($\mathcal{R}$) language [12] with the tableau solved form over non-negative variables.

5.1 Method Principles

The basic principle of this method is to consider each implicit equality found as an implicit equality in the solved linear program (5) and to add the corresponding equation in the system. If this addition involves the appearance of implicit equalities, the new solved linear program that proves this is used to identify a part of them as restrictive inequations. The corresponding equations are added the same way. When no more inequations need to be reintroduced, the final system is regular.

More precisely, for each restrictive inequation $l_i \leq x_i$ found, the equation $x_i = l_i$ is added to the system to obtain a system equivalent to

$$S \cup \{x_i = l_i\}$$

This system is obviously solvable since $\sigma_S(x_i) = l_i$. It is only necessary to decide if it contains implicit equalities. These are searched for in one of the systems $S \cup \{x_i \leq l_i\}$ or $S \cup \{l_i \leq x_i\}$. Since S is regular the constraint $l_i \leq x_i$ is not an implicit equality in S and we have to decide if $x_i \leq l_i$ is an implicit equality in the system

$$S' = (S - \{l_i \leq x_i \leq u_i\}) \cup \{x_i \leq l_i\}$$

For this purpose the linear program $\min\{x_i \mid S'\}$ is solved (the case where the restrictive inequation is $x_i \leq u_i$ is similar).

If one of the inequations $x_i \leq l_i$ or $u_i \leq x_i$ is an implicit equality, the minimization computes a linear program in revised solved form that gives new implicit equalities as restrictive inequations. The whole process is then repeated for each of them.

5.2 Algorithm

The algorithm described in figure 1 begins by computing the set E of restrictive inequations of the linear program given as input (step 1). An inequation $\delta x_i \leq \alpha$ from E is chosen and the bounds on x_i are replaced by the inequation $\alpha \leq \delta x_i$ (step 2a). The expression δx_i is then minimized (step 2b). If $\min\{\delta x_i \mid S\} = \alpha$, the inequation $\delta x_i \leq \alpha$ is an implicit equality and the restrictive inequations of the solved linear program are added to E (step 2d). The variable x_i is then explicitly fixed in the system S (step 2e). These steps are repeated until $E = \emptyset$.

This algorithm has two main advantages for incrementality. Stuckey emphasizes in [18] that the number of steps of the algorithm, and thus the number of linear programs solved, is equal to the number of implicit equalities found. So this method is equivalent to the ideal method. Moreover, the minimization at step 2b for deciding the presence of implicit equalities in the system $S \cup \{\delta x_i \leq \alpha\}$ is initialized with a system whose basic solution satisfies the constraint $\delta x_i = \alpha$. Consequently, few simplex iterations are necessary to prove that $\min\{\delta x_i \mid S\} = \alpha$ or $\min\{\delta x_i \mid S\} < \alpha$. Assuming that there is no degeneracy, a single pivot is sufficient to decide. This is often the case in practice.

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let E be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. **repeat**
   a. remove a constraint $\delta x_i \leq \alpha$ from E and rewrite S to
   $$S \leftarrow \begin{cases} (S - \{l_i \leq x_i \leq u_i\}) \cup \{x_i \leq l_i\} & \text{if } \frac{\alpha}{\delta} = l_i \\ (S - \{l_i \leq x_i \leq u_i\}) \cup \{u_i \leq x_i\} & \text{if } \frac{\alpha}{\delta} = u_i \end{cases}$$
   b. solve the linear program $\gamma = \min\{\delta x_i \mid S\}$,
   c. let R be the new set of restrictive inequations
   d. **if** $\gamma = \alpha$ **then** $E \leftarrow E \cup R$
   e. $S \leftarrow S \cup \{x_i = \frac{\alpha}{\delta}\}$
   until $E = \emptyset$
3. **return** S
end

Fig. 1. `CLP(`$\mathcal{R}$`)` Method

6 Adaptation of the `Prolog III` Method

The method presented herein was developed for the linear constraint solver of `Prolog III` [6], with the tableau solved form over non-negative variables. The method is briefly described in [9] ; it has not been detailed nor proven. This approach is also very similar to the one developped in [10] for the system `clp(q,r)`.

When the first implicit equalities found as restrictive inequations are replaced by equations, the new system found may contain more implicit equalities. The `Prolog III` methods identifies some constraints that, once removed from this system, lead assuredly to a regular system. Those constraints are then reintroduced one by one in this system. If during the introduction the presence of new implicit equalities is detected, other constraints are removed and reintroduced the same way later on.

6.1 Definitions and Properties

In the `Prolog III` linear constraint solver, the constraints removed are the equations that contain an occurrence of a variable that also appears in the objective function of the solved linear program proving the presence of implicit equalities.

A more efficient approach is to remove some saturated inequations called bounding inequations. A *saturated inequation* of a system S in revised solved form is an inequation $l_i \leq x_i$ such that $\sigma_S(x_i) = l_i$ or an inequation $x_i \leq u_i$ such that $\sigma_S(x_i) = u_i$.

Definition 1. (Bounding Inequation) *The* bounding inequations *of a solved linear program* $\min\{ax_L + a'x_U \mid S\}$ *are the saturated inequations over variables that are basic in an equation of* S *and whose right hand side of the equation shares an occurrence of variables with the objective function.*

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let R be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. let E be the set of bounding inequations of $\min\{ax_L + a'x_U \mid S\}$
3. $S \leftarrow (S - E) \cup R_=$
4. **repeat**
   a. remove a constraint $\delta x_i \leq \alpha$ from E and solve $\gamma = \min\{\delta x_i \mid S\}$
   b. **if** $\gamma = \alpha$
      then $S \leftarrow S \cup \{x_i = \frac{\alpha}{\delta}\}$
      else $S \leftarrow S \cup \{\delta x_i \leq \alpha\}$
   c. let R be the set of restrictive inequations of $\min\{\delta x_i \mid S\}$ solved
   d. let S_B be the set of bounding inequations of $\min\{\delta x_i \mid S\}$ solved
   e. $S \leftarrow (S - S_B) \cup R_=$
   f. $E \leftarrow E \cup S_B$.
   until $E = \emptyset$
4. **return** S
end

Fig. 2. `Prolog III` Method

When restrictive inequations are replaced by equations, the resulting system is not necessarily regular. At this point, the removal of bounding inequations leads with certainty to a different but regular system [16].

Theorem 1. *Let P be a linear program in solved form whose system S is regular. Let R be set of restrictive inequations of P and S_B be its set of bounding inequations. The system $(S - S_B) \cup R_=$ is regular.*

Computing the system S_B requires computing in $\mathcal{T}(S)$ the coefficients of variables that have an occurrence in the objective function. These are obtained by solving the system (3) for each variable x_k that has a non zero coefficient in the objective function.

6.2 Algorithm

The algorithm described in figure 2 begins by initializing to set R of restrictive inequations (step 1) and the set E of bounding inequations to remove and reintroduce (step 2). It then computes the system $(S - E) \cup R_=$ which is regular (step 3). An inequation $\delta x_i \leq \alpha$ is chosen in the set E and is reintroduced in S by minimizing the term δx_i to decide if it is an implicit equality (step 4a). If so, the variable x_i is fixed to the value $\frac{\alpha}{\delta}$, otherwise the inequation is reintroduced in the system (step 4b). Then the algorithm continues until there are no more inequations to reintroduce.

Despite the improvement proposed, this algorithm can remove at each step as many inequations as there are equations. Reintroducing these constraints requires solving the same number of linear programs. This method is thus weakly

incremental. However, the basic solution of the system S at step 2e is a solution of the equation $\delta x_j = \alpha$ and, as in the `CLP(`$\mathcal{R}$`)` method, the minimization requires few iterations in practice. Here again, a single non-degenerate pivot is sufficient to prove that the inequation is not an implicit equality.

In conclusion this method seems less efficient than the `CLP(`$\mathcal{R}$`)` method that solves only a linear program for each implicit equality found. However, both methods are unable to prove in one step the absence of implicit equalities in the system $S \cup R_=$. This is possible with the quasi-dual method.

7 Adaptation of the Quasi-Dual Formulation

The quasi-dual formulation of a system of linear constraints was introduced by Lassez [14]. The use of this formulation for detecting implicit equalities is suggested in [13] among other possible applications. It is close to the first approach to finding implicit equalities proposed in [18]. The quasi dual formulation has been used in the `CLAIRE` language [3] and in the `Athena` language [17].

Let $S = \{Ax \leq b\}$ be a system of m inequations. The quasi-dual formulation of this system is the linear program

$$\eta = \min\{yb \mid yA = 0, y \geq 0, \Sigma y_i = 1\}$$

where y is a row-vector of m variables. If $\eta = 0$ then the system S contains implicit equalities otherwise it does not. With this formulation, the absence of implicit equalities is decided by solving a single linear program. We consider here the dual of the quasi-dual, i.e. the linear program

$$\mu = \max\{x_0 \mid Ax + u.x_0 \leq b, x_0 \geq 0\}$$

where u is a vector of m coefficients, all of them set to 1. If these two linear programs are bounded, the well-known relation $\eta = \mu$ holds. As a consequence, the dual of the quasi-dual can also be solved to detect implicit equalities. This last formulation is more interesting since it involve fewer modifications of the original system S.

In the following, we introduce the (tableau or revised) auxiliary form that is an extension of the dual of the quasi-dual formulation to the (tableau or revised) solved form. The algorithm presented here computes a system in revised auxiliary form from a system in revised solved form by introducing a new variable in some equations. That variable is then minimized. If the optimal value is zero, then the system contains implicit equalities that are identified as restrictive inequations of the solved linear program. Those inequations are replaced by the corresponding equations and the whole process is repeated for the new system to find more implicit equalities or to prove that there are no more.

7.1 Definitions and Properties

Computing the auxiliary form of a tableau solved form S consists of introducing a new variable x_0 in each equation whose basic variable x_b is assigned to one of its

bounds in the basic solution. The variable x_0 is not introduced in every constraint as in the dual of the quasi-dual formulation because the basic solution of S can discard a priori some inequations that are not implicit equalities (inequations $l_i \leq x_i$ or $x_i \leq u_i$ such that $l_i < \sigma_S(x_i)$ or $\sigma_S(x_i) < u_i$).

To clarify this presentation, the adaptation is first presented with the tableau solved form and then with the revised one.

Definition 2. (Tableau Auxiliary Form) *Let S be a system in tableau solved form (1). The* tableau auxiliary form *of S is the system*

$$S_A = \begin{cases} x_B = w.x_0 + Mx_L + M'x_U + b \\ l \leq x \leq u \\ 0 \leq x_0 \leq +\infty \end{cases}$$

where $\sigma_{S_A}(x_0) = 0$ and w is a vector such that

$$w_i = \begin{cases} 0 & \text{if } l_i < \sigma_S(x_i) < u_i \\ 1 & \text{if } \sigma_S(x_i) = l_i \\ -1 & \text{if } \sigma_S(x_i) = u_i \end{cases}$$

Note that S_A is in tableau solved form since its basic solution corresponds to the basic solution of S on all variables except on x_0 which is assigned to zero.

Definition 3. (Revised Auxiliary Form) *Let S' be a system in revised solved form (2). The* revised auxiliary form *of S' is*

$$S'_A = \begin{cases} A_B x_B + w'.x_0 + A_L x_L + A_U x_U = 0 \\ l \leq x \leq u \\ 0 \leq x_0 \leq +\infty \end{cases}$$

where x_0 is a variable assigned to zero in the basic solution of S'_A and w' is a vector such that $w' = -A_B w$ where w is the vector from definition 2.

Assume that $S = \mathcal{T}(S')$ then $S_A = \mathcal{T}(S'_A)$. The vector w' is thus the appropriate vector to introduce in S'_A to verify this relation. The solving of a single linear program can determine the presence or the absence of implicit equalities in the system S'.

Theorem 2. *With notations of definition 2, the system S' contains implicit equalities if and only if* $\min\{-x_0 \mid S'_A\} = 0$.

This result leads to the incremental algorithm of figure 3.

7.2 Algorithm

This algorithm is rather simple. It identifies first the restrictive inequations of the solved linear program (step 1). It fixes the variables appearing in those inequations (step 2a). The auxiliary revised form is computed (step 2b) and

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let E be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. **repeat**
   a. $S \leftarrow S \cup E_=$
   b. let S_A be the revised auxiliary system of S and x_0 be the var. introduced.
   c. **if** $\min\{-x_0 \mid S_A\} < 0$
      then return S.
      else let E be the set of restrictive inequations of $\min\{-x_0 \mid S_A\}$ solved.
   until false

end

Fig. 3. Quasi Dual Method

the linear program of theorem 2 is solved with the simplex method (step 2c). If the optimal value is strictly negative, the system S is regular. Otherwise new restrictive inequalities are found and the algorithm continues. Note that the iterations of the simplex can be stopped as soon as the term $-x_0$ can be assigned to a strictly negative value for the basic solution.

In comparison with the previous methods, the quasi-dual method has many advantages. The computation of the auxiliary form is simple and proving the absence of implicit equalities requires solving a single linear program. In practice this proof needs few simplex iterations since the term $-x_0$ is already assigned to zero in the basic solution of the auxiliary forms.

8 Practical Results

The three approaches above have been integrated to the CLP language `Athena`. The linear constraint solver of `Athena` is based on a revised simplex method on exact-precision rationals. It includes advanced linear programming techniques and an efficient procedure for dynamic backtracking [17].

Various examples have been tested with the three approaches. Most of these examples are typical CLP problems over linear constraints that need to detect implicit equalities for handling disequations or strict inequalities.

The `Donald` problem is the well-known cryptarithm problem [6]. `Periodic` solves the problem of proving that a mathematical sequence is periodic [6]. It contains a single disequation, however finding implicit equalities reduces the search space. The `Square` problem [6] is a two dimensional placement problem that is very combinatorial. It consists of filling a rectangle with squares of different sizes such that no squares overlaps. All pivoting operations done to solve this problem are for detecting implicit equalities. Two instances of this problem are solved: one with 9 squares, the other with 14 squares. `Transport` [21] is a deterministic problem of assigning providers to customers. At the optimum value all of the 80 variables are fixed showing that there is no other optimal solution. `Cut-stock` is a cutting stock problem solved to optimality with a branch

		Quasi-dual meth.			`Prolog III` meth.			`CLP(`$\mathcal{R}$`)` meth.		
Problems	Nb IE	Steps	Pivots	Piv. IE	Steps	Pivots	Piv. IE	Steps	Pivots	Piv. IE
Square 9	933	607	2826	2826	174	2756	2756	410	2618	2618
Square 14	4293	2939	18485	18485	861	17631	17631	1435	16886	16886
Transport	80	1	125	0	0	125	0	80	125	0
Donald	46	52	179	171	28	145	137	46	127	119
Periodic	39	46	96	96	17	72	72	23	67	67
Cut-stock	245	207	909	887	124	693	671	245	587	565

Fig. 4. Pivots and steps requirement

Problems	Quasi-dual meth.	`Prolog III` meth.	`CLP(`$\mathcal{R}$`)` meth.
Square 9	4360	4850	4680
Square 14	42930	47120	45610
Transport	190	260	280
Donald	780	800	770
Periodic	110	130	120
Cut-stock	5000	5020	4650

Fig. 5. Time requirement in ms.

and bound approach [21]. All these problems are non deterministic except for `Transport`.

Figure 4 shows the number of implicit equalities (**Nb IE**) to be found for each problem. For each method, it shows the the number of steps of the algorithm (**Steps**) which equal the number of linear programs solved to detect implicit equalities. It shows the total number of pivotings required (**Pivots**) and the number of pivoting needed to solve those linear programs (**Piv. IE**). Figure 5 shows the the computation times[2] to solve the problem.

From these results we can see that no method is clearly superior to the others.

The `Prolog III` method performs much fewer iterations than the other methods. This result is surprising since it is the worst method in terms of complexity. In terms of computation time, this advantage is negated by the need to compute columns of the tableau solved form to determine bounding inequalities. This is the case for the `Transport` where no steps are performed to find all implicit equalities since there is no bounding inequation, but the column is computed for each fixed variable to prove this fact. This explains the slowdown in comparison with the quasi-dual method.

The `CLP(`$\mathcal{R}$`)` method is slightly faster than the `Prolog III` methods. Note that it does not perform as many iterations as there are implicit equalities be-

[2] In milliseconds on a Sun Sparc 10

cause transforming a restrictive inequation into an equation sometimes leads to a failure due to disequations. Consequently no linear program is solved for the remaining restrictive inequations. This method is the one that performs the least number of pivotings. This can be explained by the fact that only a partial non degenerate pivot is often sufficient to prove the absence of implicit equalities when inequations are reintroduced. These partial pivots are not taken into account in computing the total number of pivots necessary to find all implicit equalities. However the time spent in these partial pivots can be significant since it requires the computation of the objective function (see formula 4) and the solving of the systems (see formula 3).

The quasi-dual method is the best one in terms of computation time, but it is not far better than the others. On average it is 12% faster than the `Prolog III` method and 10% faster than the `CLP(`$\mathcal{R}$`)` method. However, it can do many more iterations than the others, since fewer implicit equalities are discovered at each step. The exception is the `Transport` problem where the implicit equalities are all found at one time as restrictive inequations and the quasi-dual method proves that there are no more of them in only one step.

9 Conclusion

This paper has described how to detect implicit equalities with the revised simplex algorithm. This algorithm is more efficient and more suitable to practical problems than the tableau method that is usually applied in CLP. We have described the adaptation to this simplex of three main approaches for the incremental detection of implicit equalities: the `CLP(`$\mathcal{R}$`)`, the `Prolog III`, and the quasi-dual one.

A practical comparison with the CLP language `Athena` was done on some classical CLP problems. This comparison shows that these methods are almost equivalent from the efficiency standpoint. The quasi-dual method, which was considered the most suitable method for the revised simplex, is not much faster than the others, while the `Prolog III` method, which was considered computationally too costly, is the one that performs the least number of steps.

As a future study it would be interesting to combine the `Prolog III` and the quasi-dual methods in order to use the quick detection of the absence of implicit equalities of the latter and the fewer number of steps of the former.

Acknowledgments

The author is grateful to Pascal Van Hentenryck for discussions about the subject of this work. This research was partially supported by the ESPRIT project ACCLAIM-7195.

References

1. *Ilog Planner, User Manual*. ILOG, S.A., Paris, France, November 1996. 482
2. Frédéric Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea. 482
3. H. Beringer and B. de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In *Logic Programming : Formal Methods and Practical Applications*. Elsevier Science Publishers, 1994. 482, 486, 491
4. D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997. 482
5. V. Chvatal. *Linear Programming*. W.H. Freeman and Company, New York, 1983. 482, 484, 485
6. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–91, July 1990. 481, 482, 489, 493
7. A. Colmerauer. Naive resolution of non-linear constraints. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 89–112. MIT Press, 1993. 481
8. A. Colmerauer. Spécifications de Prolog IV. Technical report, Laboratoire d'Informatique de Marseille, 1996. 481, 482
9. Michel Henrion. Les algorithmes numériques de Prolog III. Technical report, Prologia, 1989. 482, 489
10. C. Holzbaur. A specialized incremental solved form algorithm for systems of linear inequalities. Technical Report TR-94-07, Austrian Research Institute for Artificial Intelligence, Vienna, 1994. 489
11. J.-L. Imbert and P. van Hentenryck. On the handling of disequations in CLP over linear rational arithmetic. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 49–72. MIT Press, 1993. 486
12. J. Jaffar, S. Michaylov, P. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. *Transactions on Programming Languages and Systems*, 14(3), July 1992. 481, 482, 486, 487
13. J-L Lassez. Parametric queries, linear constraints and variable elimination. In *DISCO 90, LNCS 429*, pages 164–173. Springer-Verlag Lecture Notes in Computer Science, 1990. 482, 491
14. J-L Lassez. Querying constraints. In *Proceedings of the ACM Conference on Principles of Database Systems*, Nashville, 1990. 491
15. J-L Lassez and K. McAloon. A canonical form for generalised linear constraints. *Journal of Symbolic Computation*, (1):1–24, 1992. 481, 486
16. P. Refalo. *Resolution et implication de contraintes lineaires en programmation logique par constraintes*. Ph.D Thesis. Laboratoire d'informatique de Marseille, Marseille, 1997. 482, 485, 486, 487, 490
17. P. Refalo and P. van Hentenryck. `CLP`($\mathcal{R}_{lin}$) revised. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 1–14, Bonn, Germany, 1996. 482, 485, 486, 491, 493
18. P. J. Stuckey. Incremental linear constraint solving and implicit equalities. *ORSA Journal of Computing*, 3(4):269–274, 1991. 482, 487, 488, 491
19. J. Telgen. *Redundancy and Linear Programs*. Mathematical Centre Tracts, 1981. number 137. 487

20. P. van Hentenryck and T. Graf. Standard forms for rational linear arithmetics in constraint logic programming. In *The International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1990. 482, 486
21. P. van Hentenryck and V. Ramachandran. Backtracking without trailling in CLP($\mathcal{R}_{lin}$). *ACM Transaction on Programming Langages and Systems*, 1(1):0–0, 1995. 493, 494

Author Index