Lecture Notes in Computer Science 2664
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Michael Leuschel (Ed.)

Logic Based Program Synthesis and Transformation

12th International Workshop, LOPSTR 2002
Madrid, Spain, September 17-20, 2002
Revised Selected Papers

Springer

Volume Editor

Michael Leuschel
University of Southampton
Dept. of Electronics and Computer Science
Highfield, Southampton, SO17 1BJ, UK
E-mail: mal@ecs.soton.ac.uk

Cataloging-in-Publication Data applied for

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

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

CR Subject Classification (1998): F.3.1, D.1.1, D.1.6, I.2.2, F.4.1

ISSN 0302-9743
ISBN 3-540-40438-4 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN 10937549 06/3142 5 4 3 2 1 0

Preface

This volume contains selected papers from LOPSTR 2002, the 12th International Workshop on Logic-Based Program Development and Transformation. Topics of interest to LOPSTR cover all aspects of logic-based program development and, in particular, specification, synthesis, verification, transformation, specialization, analysis, optimization, composition, reuse, component-based and agent-based software development, and software architectures.

LOPSTR 2002 took place at the Technical University of Madrid (Spain) from September 17 to September 20, 2002. Past LOPSTR workshops were held in Manchester, UK (1991, 1992, 1998), Louvain-la-Neuve, Belgium (1993), Pisa, Italy (1994), Arnhem, The Netherlands (1995), Stockholm, Sweden (1996), Leuven, Belgium (1997), Venice, Italy (1999), London, UK (2000), and Paphos, Cyprus (2001).

LOPSTR 2002 was co-located with the International Static Analysis Symposium (SAS 2002), the APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP 2002), and the Joint CoLogNet Workshop on Component-Based Software Development and Implementation Technology for Computational Logic Systems. The LOPSTR community profited from the shared lectures of the invited speakers, and the active scientific discussions enabled by the co-location.

I especially wish to thank Francisco Bueno and the entire team of local organizers from the Technical University of Madrid. They did an outstanding job in organizing the various events and helped make LOPSTR 2002 a success. I also wish to express my gratitude towards the program committee and the many additional referees for their efforts in carefully reviewing the submitted papers and ensuring the quality of LOPSTR. In addition I wish to thank all the authors who submitted their papers to LOPSTR 2002. Special thanks to Stefan Gruner, who helped me with various tasks related to the organization of LOPSTR 2002. Finally, the sponsorship of the Association of Logic Programming (ALP) is gratefully acknowledged, and I thank Springer-Verlag for publishing this volume of selected papers in the LNCS series.

Out of 40 submissions, the program committee selected 22 for presentation at LOPSTR 2002, 15 of which were selected to appear as long papers in this volume. This volume also includes abstracts of the other 7 papers presented at LOPSTR 2002.

The preproceedings of LOPSTR 2002 were printed by the Technical University of Madrid. Further information can be obtained from the LOPSTR 2002 homepage: `http://clip.dia.fi.upm.es/LOPSTR02/`.

March 2003 Michael Leuschel

Program Chair

Michael Leuschel	University of Southampton, UK

Local Chair

Francisco Bueno	Technical University of Madrid, Spain

Program Committee

Jean Raymond Abrial	Consultant, Marseille, France
Elvira Albert	Technical University of Valencia, Spain
Michael Butler	University of Southampton, UK
James Caldwell	University of Wyoming, USA
Bart Demoen	University of Leuven, Belgium
Sandro Etalle	University of Twente, The Netherlands
Laurent Fribourg	ENS Cachan, France
Michael Hanus	University of Kiel, Germany
Andy King	University of Kent, UK
Kung-Kiu Lau	University of Manchester, UK
Michael Leuschel	University of Southampton, UK
C.R. Ramakrishnan	SUNY at Stony Brook, USA
Olivier Ridoux	University of Rennes, France
Sabina Rossi	University of Venice, Italy
Wolfram Schulte	Microsoft Research, USA
Jens Peter Secher	University of Copenhagen, Denmark
Maurizio Proietti	IASI-CNR, Rome, Italy
Germán Puebla	Technical University of Madrid, Spain
Julian Richardson	University of Edinburgh, UK
Michael Wooldridge	University of Liverpool, UK

Local Organizers

Astrid Beascoa	Pedro López
Francisco Bueno	José Morales
Jesús Correas	Oscar Portela
Jose Manuel Gómez	Germán Puebla
Manuel Hermenegildo	Claudio Vaucheret

Additional Referees

Juan C. Augusto
Hendrik Blockeel
Yegor Bryukhov
Veronique Cortier
John Cowles
Wlodzimierz Drabent
Santiago Escobar
Carla Ferreira
Jean-Christophe Filliatre
Fabio Fioravanti
Andy Gravell
Wolfgang Goerigk
Jean Goubault-Larrecq
Tamas Horvath
Frank Huch
Baudouin Le Charlier
Helko Lehmann
Jim Lipton
Thierry Massart
Nancy Mazur
Fred Mesnard
Ginés Moreno
Guy Alain Narboni
Nikolay Pelov
Alberto Pettorossi
Carla Piazza
Jan Ramon
Francesco Ranzato
Jacques Riche
Tom Schrijvers
Francesca Scozzari
Colin Snook
Remko Tronçon
Peter Vanbroekhoven
Bert Van Nuffelen
Claudio Vaucheret
Germán Vidal
Alicia Villanueva

Table of Contents

Verification

Partial Evaluation

Rewriting and Object-Oriented Development

Abstract Diagnosis of Functional Programs*

María Alpuente[1], Marco Comini[2], Santiago Escobar[1],
Moreno Falaschi[2], and Salvador Lucas[1]

[1] Departamento de Sistemas Informáticos y Computación-DSIC
Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,sescobar,slucas}@dsic.upv.es
[2] Dipartimento di Matematica e Informatica
University of Udine, Via delle Scienze 206, 33100 Udine, Italy
{comini,falaschi}@dimi.uniud.it

Abstract. We present a generic scheme for the declarative debugging of functional programs modeled as term rewriting systems. We associate to our programs a semantics based on a (continuous) immediate consequence operator, $T_{\mathcal{R}}$, which models the (values/normal forms) semantics of $\mathcal{R}$. Then, we develop an effective debugging methodology which is based on abstract interpretation: by approximating the intended specification of the semantics of $\mathcal{R}$ we derive a finitely terminating bottom-up diagnosis method, which can be used statically. Our debugging framework does not require the user to either provide error symptoms in advance or answer questions concerning program correctness. We have made available a prototypical implementation in Haskell and have tested it on some non trivial examples.

1 Introduction

Finding program bugs is a long-standing problem in software construction. This paper is motivated by the fact that the debugging support for functional languages in current systems is poor [31], and there are no general purpose, good semantics-based debugging tools available. Traditional debugging tools for functional programming languages consist of tracers which help to display the execution [23,30] but which do not enforce program correctness adequately as they do not provide means for finding bugs in the source code w.r.t. the intended program semantics. Declarative debugging of functional programs [22,21,28] is a semi-automatic debugging technique where the debugger tries to locate the node in an execution tree which is ultimately responsible for a visible bug symptom. This is done by asking the user, which assumes the role of the oracle. When debugging real code, the questions are often textually large and may be difficult to answer. Abstract diagnosis [9,10,11] is a declarative debugging framework which extends the methodology in [16,26], based on using the immediate consequence operator T_P, to identify bugs in logic programs, to diagnosis w.r.t. computed

* Work partially supported by CICYT TIC2001-2705-C03-01, Acciones Integradas HI2000-0161, HA2001-0059, HU2001-0019, and Generalitat Valenciana GV01-424.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 1–16, 2003.

answers. The framework is goal independent and does not require the determination of symptoms in advance.

In this paper, we develop an abstract diagnosis method for functional programming which applies the ideas of [10] to debug a functional program w.r.t. the semantics of normal forms and (ground) constructor normal forms (or *values*). We use the formalism of term rewriting systems as it provides an adequate computational model for functional programming languages where functions are defined by means of patterns (e.g., Haskell, Hope or Miranda) [4,18,25]. We associate a (continuous) immediate consequence operator $T_{\mathcal{R}}$ to program $\mathcal{R}$ which allows us to derive an input-output semantics for $\mathcal{R}$, as in the fixpoint finite/angelic relational semantics of [12]. Then, we formulate an efficient debugging methodology, based on abstract interpretation, which proceeds by approximating the $T_{\mathcal{R}}$ operator by means of a $depth(k)$ cut [10]. We show that, given the intended specification $\mathcal{I}$ of the semantics of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single step of the abstract immediate consequence operator $T^{\kappa}_{\mathcal{R}}$ and, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

The debugging of functional programs via specifications is an important topic in automated program development. For example, in QuickCheck [7], formal specifications are used to describe properties of Haskell programs (which are written as Haskell programs too) which are automatically tested on random input. This means that the program is run on a large amount of arguments which are randomly generated using the specification. A size bound is used to ensure finiteness of the test data generation. A *domain specific* language of *testable specifications* is imposed which is embedded in Haskell, and only properties which are expressible and observable within this language can be considered. The major limitation of Quickcheck is that there is no measurement of structural coverage of the function under test: there is no check, for instance, that every part of the code is exercized as it heavily depends on the distribution of test data.

The debugging methodology which we propose can be very useful for a functional programmer who wants to debug a program w.r.t. a preliminary version which was written with no efficiency concern. Actually, in software development a specification may be seen as the starting point for the subsequent program development, and as the criterion for judging the correctness of the final software product. For instance, the executability of OBJ[1] specifications supports prototype-driven incremental development methods [17]. On the other hand, OBJ languages have been provided with equational proving facilities such as a Knuth-Bendix completion tool which, starting with a finite set of equations and a reduction order, attempts to find a finite canonical system for the considered theory by generating critical pairs and orienting them as necessary [8,13]. However, it might happen that the completion procedure fails because there is a critical pair which cannot be oriented. Thus, in many cases, the original code needs to be manipulated by hand, which may introduce incorrectness or incom-

[1] By OBJ we refer to the family of OBJ-like equational languages, which includes OBJ3, CafeOBJ, and Maude.

pleteness errors in program rules. Therefore, a debugging tool which is able to locate bugs in the user's program and provide validation of the user's intention becomes also important in this context. In general it often happens that some parts of the software need to be improved during the software life cycle, for instance for getting a better performance. Then the old programs (or large parts of them) can be usefully (and automatically) used as a specification of the new ones.

The rest of the paper is organized as follows. Section 3 introduces a novel immediate consequence operator $T_{\mathcal{R}}$ for functional program $\mathcal{R}$. We then define a fixpoint semantics based on $T_{\mathcal{R}}$ which correctly models the values/normal forms semantics of $\mathcal{R}$. Section 4 provides an abstract semantics which correctly approximates the fixpoint semantics of $\mathcal{R}$. In Section 5, we present our method of abstract diagnosis. The diagnosis is based on the detection of *incorrect rules* and *uncovered equations*, which both have a bottom-up definition (in terms of one application of the "abstract immediate consequence operator" $T^{\alpha}_{\mathcal{R}}$ to the abstract specification). It is worth noting that no fixpoint computation is required, since the abstract semantics does not need to be computed. We have developed a prototypical implementation in Haskell (DEBUSSY) which we use for running all the examples we illustrate in this section. Section 6 concludes.

2 Preliminaries

Let us briefly recall some known results about rewrite systems [4,18]. For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [24]. In the paper, syntactic equality of terms is represented by $\equiv$. Throughout this paper, $\mathcal{V}$ will denote a countably infinite set of variables and Σ denotes a set of function symbols, or signature, each of which has a fixed associated arity. $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup \mathcal{V}$ and Σ, respectively. $\mathcal{T}(\Sigma)$ is usually called the Herbrand universe ($\mathcal{H}_\Sigma$) over Σ and it will be denoted by $\mathcal{H}$. $\mathcal{B}$ denotes the Herbrand base, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. A Σ-equation $s = t$ is a pair of terms $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$, or *true*.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t which are rooted by symbols in S. $t|_u$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm at the position u replaced with r. By $Var(s)$ we denote the set of variables occurring in the syntactic object s, while $[s]$ denotes the set of ground instances of s. A *fresh* variable is a variable that appears nowhere else.

A *substitution* is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. We write $\theta_{\restriction s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by ϵ. A *renaming* is a substitution ρ for which there exists the inverse

ρ^{-1}, such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. An equation set E is unifiable, if there exists ϑ such that, for all $s = t$ in E, we have $s\vartheta \equiv t\vartheta$, and ϑ is called a *unifier* of E. We let $mgu(E)$ denote 'the' *most general unifier* of the equation set E [20].

A *term rewriting system* (TRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rule schemes of the form $l \to r$, $l,\ r \in \mathcal{T}(\Sigma, \mathcal{V})$, $l \notin \mathcal{V}$ and $Var(r) \subseteq Var(l)$. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$. For TRS $\mathcal{R}$, $r \ll \mathcal{R}$ denotes that r is a new variant of a rule in $\mathcal{R}$ such that r contains only *fresh* variables, i.e., contains no variable previously met during computation (standardized apart). Given a TRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma := \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} := \{f \mid f(t_1, \ldots, t_n) \to r \in \mathcal{R}\}$ and $\mathcal{C} := \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called constructor terms. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and $\bar{d}$ is a n-tuple of constructor terms. We say that a TRS is *constructor–based* (CB) if the left hand sides of $\mathcal{R}$ are patterns.

A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t, $s \to_{\mathcal{R}} t$, if there exist $u \in O_{\Sigma}(s)$, $l \to r$, and substitution σ such that $s|_u \equiv l\sigma$ and $t \equiv s[r\sigma]_u$. When no confusion can arise, we omit the subscript $\mathcal{R}$. A term s is a *normal form*, if there is no term t with $s \to_{\mathcal{R}} t$. t is the normal form of s if $s \to^*_{\mathcal{R}} t$ and t is a normal form (in symbols $s \to^!_{\mathcal{R}} t$). A TRS $\mathcal{R}$ is *noetherian* if there are no infinite sequences of the form $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \ldots$ A TRS $\mathcal{R}$ is *confluent* if, whenever a term s reduces to two terms t_1 and t_2, both t_1 and t_2 reduce to the same term. The program $\mathcal{R}$ is said to be canonical if $\mathcal{R}$ is noetherian and confluent [18].

In the following we consider functional languages and thus often will refer to the corresponding TRS of a program with the term program itself.

3 The Semantic Framework

The relational style of semantic description associates an input-output relation to a program where intermediate computation steps are ignored [12]. In this section, we consider the finite/angelic relational semantics of [12], given in fixpoint style. In order to formulate our semantics for term rewriting systems, the usual Herbrand base is extended to the set of all (possibly) non-ground equations [14,15]. $\mathcal{H}_{\mathcal{V}}$ denotes the $\mathcal{V}$-*Herbrand universe* which allows variables in its elements, and is defined as $\mathcal{T}(\Sigma, \mathcal{V})/_{\cong}$, where $\cong$ is the equivalence relation induced by the preorder $\leq$ of "relative generality" between terms, i.e. $s \leq t$ if there exists σ s.t. $t \equiv \sigma(s)$. For the sake of simplicity, the elements of $\mathcal{H}_{\mathcal{V}}$ (equivalence classes) have the same representation as the elements of $\mathcal{T}(\Sigma, \mathcal{V})$ and are also called terms. $\mathcal{B}_{\mathcal{V}}$ denotes the $\mathcal{V}$-*Herbrand base*, namely, the set of all equations $s = t$ modulo variance, where $s, t \in \mathcal{H}_{\mathcal{V}}$. A subset of $\mathcal{B}_{\mathcal{V}}$ is called a $\mathcal{V}$-Herbrand interpretation. We assume that the equations in the denotation are renamed apart. The ordering $\leq$ for terms is extended to equations in the obvious way, i.e. $s = t \leq s' = t'$ iff there exists σ s.t. $\sigma(s) = \sigma(t) \equiv s' = t'$.

The concrete domain $\mathbb{E}$ is the lattice of $\mathcal{V}$-Herbrand interpretations, i.e., the powerset of $\mathcal{B}_\mathcal{V}$ ordered by set inclusion.

In the sequel, a semantics for program $\mathcal{R}$ is a $\mathcal{V}$-Herbrand interpretation. In term rewriting, the semantics which is usually considered is the set of normal forms of terms, $Sem_{\mathsf{nf}}(\mathcal{R}) := \{s = t \mid s \to^!_\mathcal{R} t\}$. On the other hand, in functional programming, programmers are generally concerned with an abstraction of such semantics where only the values (ground constructor normal forms) that input expressions represent are considered, $Sem_{\mathsf{val}}(\mathcal{R}) := Sem_{\mathsf{nf}}(\mathcal{R}) \cap \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\mathcal{C})$.

Since our framework does not depend on the particular target semantics, following [12], our definitions are parametric by the set of final/blocking state pairs B. Then if we are interested, for example, in the semantics of values we can take as final/blocking state pairs the set $\mathsf{val} := \{t = t \mid t \in \mathcal{T}(\mathcal{C})\}$. Moreover if we are interested in the semantics of normal forms we can take B as $\mathsf{nf} := \{t = t \mid t$ is a normal form for $\mathcal{R}\}$.

We can give a fixpoint characterization of the shown semantics by means of the following immediate consequence operator.

Definition 1. *Let $\mathcal{I}$ be a Herbrand interpretation, B be a set of final/blocking state pairs and $\mathcal{R}$ be a TRS. Then,*

$$T_{\mathcal{R},B}(\mathcal{I}) := B \cup \{s = t \mid r = t \in \mathcal{I}, s \to_\mathcal{R} r\}$$

The following proposition allows us to define the fixpoint semantics.

Proposition 1. *Let $\mathcal{R}$ be a TRS and B be a set of final/blocking state pairs. The $T_{\mathcal{R},B}$ operator is continuous on $\mathbb{E}$.*

Definition 2. *The least fixpoint semantics of a program $\mathcal{R}$ w.r.t. a set of final/blocking state pairs B, is defined as $\mathcal{F}_B(\mathcal{R}) = T_{\mathcal{R},B} \uparrow \omega$.*

The following result relates the (fixpoint) semantics computed by the $T_\mathcal{R}$ operator with the semantics val and nf.

Theorem 1 (soundness and completeness). *Let $\mathcal{R}$ be a TRS. Then, $Sem_{\mathsf{nf}}(\mathcal{R}) = \mathcal{F}_{\mathsf{nf}}(\mathcal{R})$ and $Sem_{\mathsf{val}}(\mathcal{R}) = \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

Example 1. Let us consider now the following (wrong) program $\mathcal{R}$ expressed in OBJ for doubling.

```
obj ERRDOUBLE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op double : Nat -> Nat .
  var X : Nat .
  eq double(0) = 0 .
  eq double(s(X)) = double(X) .
endo
```

The intended specification is given by the following OBJ program $\mathcal{I}$ which uses addition for doubling:

```
obj ADDDOUBLE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op double : Nat -> Nat .
  op add : Nat Nat -> Nat .
  vars X Y : Nat .
  eq add(0,X) = X .
  eq add(s(X),Y) = s(add(X,Y)) .
  eq double(X) = add(X,X) .
endo
```

According to Definition 2, the val fixpoint semantics of $\mathcal{R}$ is[2] (we omit the equations for the auxiliary function add):

$$\begin{aligned}\mathcal{F}_{\mathsf{val}}(\mathcal{R}) = \{&\mathtt{0=0},\ \mathtt{double(0)=0},\ \mathtt{double(s(0))=0},\ \mathtt{double(s^2(0))=0},\\ &\mathtt{double(s^3(0))=0},\ \mathtt{double(s^4(0))=0},\ \ldots,\ \mathtt{s(0)=s(0)},\\ &\mathtt{s(double(0))=s(0)},\ \mathtt{s(double(s(0)))=s(0)},\\ &\mathtt{s(double(s^2(0)))=s(0)},\ \mathtt{s(double(s^3(0)))=s(0)},\\ &\mathtt{s(double(s^4(0)))=s(0)},\ \ldots,\ \mathtt{s^2(0)=s^2(0)},\\ &\mathtt{s^2(double(0))=s^2(0)},\ \mathtt{s^2(double(s(0)))=s^2(0)},\\ &\mathtt{s^2(double(s^2(0)))=s^2(0)},\ \mathtt{s^2(double(s^3(0)))=s^2(0)},\\ &\mathtt{s^2(double(s^4(0)))=s^2(0)},\ \ldots \qquad\qquad \}\end{aligned}$$

whereas the nf fixpoint semantics of $\mathcal{I}$ is:

$$\begin{aligned}\mathcal{F}_{\mathsf{nf}}(\mathcal{I}) = \{&\mathtt{0=0},\mathtt{X=X},\mathtt{s(0)=s(0)},\mathtt{s(X)=s(X)},\ \mathtt{double(0)=0},\\ &\mathtt{double(X)=add(X,X)},\ \mathtt{s^2(0)=s^2(0)},\ \mathtt{s^2(X)=s^2(X)},\\ &\mathtt{s(double(0))=s(0)},\ \mathtt{s(double(X))=s(add(X,X))},\\ &\mathtt{double(s(0))=s^2(0)},\ \mathtt{double(s(X))=add(s(X),s(X))},\\ &\mathtt{double^2(0)=0},\ \mathtt{double^2(X)=add(add(X),add(X))},\ \mathtt{s^3(0)=s^3(0)},\\ &\mathtt{s^3(X)=s^3(X)},\ \mathtt{s^2(double(X))=s^2(add(X,X))},\\ &\mathtt{s(double(s(0)))=s^3(0)},\ \mathtt{s(double(s(X)))=s(add(s(X),s(X)))},\\ &\mathtt{s(double^2(0))=s(0)},\ \mathtt{s(double^2(X))=s(add(add(X),add(X)))},\\ &\mathtt{double(s^2(0))=s^4(0)},\ \mathtt{double(s^2(X))=s^2(add(X,s^2(X)))},\ \ldots \quad \}\end{aligned}$$

Now, we can "compute in the fixpoint semantics $\mathcal{F}_{\mathsf{val}}(\mathcal{R})$" the denotation of the term $t \equiv \mathtt{double(s(0))}$, which yields $s \equiv \mathtt{0}$, since the denotation $\mathcal{F}_{\mathsf{val}}(\mathcal{R})$ contains the equation $t = s$; note that this value is erroneous w.r.t. the intended semantics of the double operation.

[2] We use the notation $f^n(x)$ as a shorthand for $f(f(\ldots f(x)))$, where f is applied n times.

4 Abstract Semantics

In this section, starting from the fixpoint semantics in Section 3, we develop an abstract semantics which approximates the observable behavior of the program and is adequate for modular data-flow analysis, such as the analysis of unsatisfiability of equation sets.

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call a *term abstraction* $\tau : \mathcal{T}(\Sigma, \mathcal{V}) \to AT$.

We start by choosing as abstract domain $\mathbb{A} := \mathcal{P}(\{a = a' \mid a, a' \in AT\})$, ordered by a set ordering $\sqsubseteq$. We will call elements of $\mathbb{A}$ abstract Herbrand interpretations. The concrete domain $\mathbb{E}$ is the powerset of $\mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$, ordered by set inclusion.

Then we can lift τ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{E}$ by defining

$$\alpha(E) := \{\tau(s) = \tau(t) \mid s = t \in E\}$$
$$\gamma(A) := \{s = t \mid \tau(s) = \tau(t) \in A\}$$

The only requirement we put on τ is that $\alpha(Sem(\mathcal{R}))$ is finite.

Now we can derive the optimal abstract version of $T_{\mathcal{R}}$ simply as $T^{\alpha}_{\mathcal{R}} := \alpha \circ T_{\mathcal{R}} \circ \gamma$. By applying the previous definition of α and γ this turns out to be equivalent to the following definition.

Definition 3. *Let τ be a term abstraction, $X \in \mathbb{A}$ be an abstract Herbrand interpretation and $\mathcal{R}$ be a TRS. Then,*

$$T^{\alpha}_{\mathcal{R},B}(X) = \{\tau(s) = \tau(t) \mid s = t \in B\} \cup \{\tau(s) = \tau(t) \mid \tau(r) = \tau(t) \in X, s \to_{\mathcal{R}} r\}$$

Abstract interpretation theory assures that $T^{\alpha}_{\mathcal{R},B} \uparrow \omega$ is the best correct approximation of $Sem_B(\mathcal{R})$. Correct means $T^{\alpha}_{\mathcal{R},B} \uparrow \omega \sqsubseteq \alpha(Sem_B(\mathcal{R}))$ and best means that it is the maximum w.r.t. $\sqsubseteq$ of all correct approximations.

Now we can define the abstract semantics as the least fixpoint of this (obviously continuous) operator.

Definition 4. *The abstract least fixpoint semantics of a program $\mathcal{R}$ w.r.t. a set of final/blocking state pairs B, is defined as $\mathcal{F}^{\alpha}_{B}(\mathcal{R}) = T^{\alpha}_{\mathcal{R},B} \uparrow \omega$.*

By our finiteness assumption on τ we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists a finite natural number h such that $T^{\alpha}_{\mathcal{R},B} \uparrow \omega = T^{\alpha}_{\mathcal{R},B} \uparrow h$.

4.1 A Case Study: The Domain *depth(k)*

Now we show how to approximate an infinite set of computed equations by means of a *depth*(k) cut [29], i.e., by using a term abstraction $\tau : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})$ which cuts terms having a depth greater than k. Terms are cut by replacing each subterm rooted at depth k with a new variable taken from the set $\hat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent each term obtained by instantiating the variables of $\hat{\mathcal{V}}$ with terms built over $\mathcal{V}$.

First of all we define the term abstraction $t/_k$ (for $k \geq 0$) as the $depth(k)$ cut of the concrete term t. We denote by $T/_k$ the set of $depth(k)$ terms $(\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})/_k)$. The abstract domain $\mathbb{A}$ is thus $\mathcal{P}(\{a = a' \mid a, a' \in T/_k\})$ ordered by the Smyth extension of ordering $\leq$ to sets, i.e. $X \leq_S Y$ iff $\forall y \in Y\ \exists x \in X : (x \leq y)$ [27]. The resulting abstraction α is $\kappa(E) := \{s/_k = t/_k \mid s = t \in E\}$.

We provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Proposition 2. *For $k > 0$, the operator $T^{\kappa}_{\mathcal{R},B} : T/_k \times T/_k \to T/_k \times T/_k$ obtained by Definition 3 holds the property $\widetilde{T}^{\kappa}_{\mathcal{R},B}(X) \leq_S T^{\kappa}_{\mathcal{R},B}(X)$ w.r.t. the following operator:*

$$\widetilde{T}^{\kappa}_{\mathcal{R},B}(X) = \kappa(B) \cup \{\sigma(u[l]_p)/_k = t \mid u = t \in X, p \in O_{\Sigma\cup\mathcal{V}}(u), \\ l \to r \ll \mathcal{R}, \sigma = mgu(u|_p, r)\}$$

Definition 5. *The effective abstract least fixpoint semantics of a program $\mathcal{R}$ w.r.t. a set of final/blocking state pairs B, is defined as $\widetilde{\mathcal{F}}^{\kappa}_{B}(\mathcal{R}) = \widetilde{T}^{\kappa}_{\mathcal{R},B} \uparrow \omega$.*

Proposition 3 (Correctness). *Let $\mathcal{R}$ be a TRS and $k > 0$.*

1. $\widetilde{\mathcal{F}}^{\kappa}_{B}(\mathcal{R}) \leq_S \kappa(\mathcal{F}_B(\mathcal{R})) \leq_S \mathcal{F}_B(\mathcal{R})$.
2. *For all $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ that are ground, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*
3. *For all $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{nf}}(\mathcal{R})$ such that $Var(e) \cap \hat{\mathcal{V}} = \emptyset$, $e \in \mathcal{F}_{\mathsf{nf}}(\mathcal{R})$.*

Example 2. Consider the correct (i.e., intended) version $\mathcal{I}$ of program in Example 1 and take $k = 2$. We have:

$$\mathsf{val}/_2 = \{\texttt{0=0},\ \texttt{s(0)=s(0)},\ \texttt{s(s(x̂))=s(s(x̂))}\}$$

According to the previous definition, the fixpoint abstract semantics is (without equations for `add`):

$$\begin{aligned}\widetilde{\mathcal{F}}^{2}_{\mathsf{val}}(\mathcal{I}) = \{&\texttt{0 = 0}, \texttt{s(0) = s(0)}, \texttt{s(0) = s(0)}, \texttt{s(s(x̂)) = s(s(x̂))},\\ &\texttt{double(0) = 0}, \texttt{s(double(x̂)) = s(0)},\\ &\texttt{s(double(x̂)) = s(s(ŷ))}, \texttt{double(s(x̂)) = s(0)},\\ &\texttt{double(s(x̂)) = s(s(ŷ))}, \texttt{double(double(x̂)) = 0},\\ &\texttt{double(double(x̂)) = s(0)}, \texttt{double(double(x̂)) = s(s(ŷ))}\ \}\end{aligned}$$

In particular, note that all ground equations

$$\{\texttt{0=0}, \texttt{s(0)=s(0)}, \texttt{double(0)=0}\ \}$$

in $\widetilde{\mathcal{F}}^{2}_{\mathsf{val}}(\mathcal{I})$ belong to the concrete semantics $\mathcal{F}_{\mathsf{val}}(\mathcal{I})$.

5 Abstract Diagnosis of Functional Programs

Program properties which can be of interest are Galois Insertions between the concrete domain (the set of Herbrand interpretations ordered by set inclusion) and the abstract domain chosen to model the property. The following Definition 6 extends to abstract diagnosis the definitions given in [26,16,19] for declarative diagnosis. In the following, $\mathcal{I}^\alpha$ is the specification of the intended behavior of a program w.r.t. the property α.

Definition 6. *Let $\mathcal{R}$ be a program and α be a property.*

1. *$\mathcal{R}$ is* partially correct *w.r.t. $\mathcal{I}^\alpha$ if $\mathcal{I}^\alpha \sqsubseteq \alpha(Sem(\mathcal{R}))$.*
2. *$\mathcal{R}$ is* complete *w.r.t. $\mathcal{I}^\alpha$ if $\alpha(Sem(\mathcal{R})) \sqsubseteq \mathcal{I}^\alpha$.*
3. *$\mathcal{R}$ is* totally correct *w.r.t. $\mathcal{I}^\alpha$, if it is partially correct and complete.*

It is worth noting that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(Sem(\mathcal{R}))$ and not in terms of the (possibly less precise) abstract semantics $Sem^\alpha(\mathcal{R})$. This means that $\mathcal{I}^\alpha$ is the abstraction of the intended concrete semantics of $\mathcal{R}$. In other words, the specifier can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations. Note also that our notion of total correctness does not concern termination. We cannot address termination issues here, since the concrete semantics we use is too abstract.

The *diagnosis* determines the "basic" symptoms and, in the case of incorrectness, the relevant rule in the program. This is captured by the definitions of *abstractly incorrect rule* and *abstract uncovered equation.*

Definition 7. *Let r be a program rule. Then r is* abstractly incorrect *if $\mathcal{I}^\alpha \not\sqsupseteq T^\alpha_{\{r\}}(\mathcal{I}^\alpha)$.*

Informally, r is abstractly incorrect if it derives a wrong abstract element from the intended semantics.

Definition 8. *Let $\mathcal{R}$ be a program. $\mathcal{R}$ has abstract uncovered elements if $T^\alpha_{\mathcal{R}}(\mathcal{I}^\alpha) \not\sqsubseteq \mathcal{I}^\alpha$.*

Informally, e is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking the conditions of Definitions 7 and 8 requires one application of $T^\alpha_{\mathcal{R}}$ to $\mathcal{I}^\alpha$, while the standard detection based on symptoms [26] would require the construction of $\alpha(Sem(\mathcal{R}))$ and therefore a fixpoint computation.

In this section, we are left with the problem of formally establishing the properties of the diagnosis method, i.e., of proving which is the relation between abstractly incorrect rules and abstract uncovered equations on one side, and correctness and completeness, on the other side.

It is worth noting that correctness and completeness are defined in terms of $\alpha(Sem(\mathcal{R}))$, i.e., in terms of abstraction of the concrete semantics. On the other hand, abstractly incorrect rules and abstract uncovered equations are defined directly in terms of abstract computations (the abstract immediate consequence

operator $T^{\alpha}_{\mathcal{R}}$). The issue of the precision of the abstract semantics becomes therefore relevant in establishing the relation between the two concepts.

Theorem 2. *If there are no abstractly incorrect rules in $\mathcal{R}$, then $\mathcal{R}$ is partially correct w.r.t. $\mathcal{I}^{\alpha}$.*

Theorem 3. *Let $\mathcal{R}$ be partially correct w.r.t. $\mathcal{I}^{\alpha}$. If $\mathcal{R}$ has abstract uncovered elements then $\mathcal{R}$ is not complete.*

Abstract incorrect rules are in general just a hint about a possible source of errors. Once an abstract incorrect rule is detected, one would have to check on the abstraction of the concrete semantics if there is indeed a bug. This is obviously unfeasible in an automatic way. However we will see that, by adding to the scheme an under-approximation of the intended specification, something worthwhile can still be done.

Real errors can be expressed as incorrect rules according to the following definition.

Definition 9. *Let r be a program rule. Then r is* incorrect *if there exists an equation e such that $e \in T_{\{r\}}(\mathcal{I})$ and $e \notin \mathcal{I}$.*

Definition 10. *Let $\mathcal{R}$ be a program. Then $\mathcal{R}$ has an* uncovered element *if there exist an equation e such that $e \in \mathcal{I}$ and $e \notin T_{\mathcal{R}}(\mathcal{I})$.*

The following theorem shows that if the program has an incorrect rule it is also an abstractly incorrect rule.

Theorem 4. *Any incorrect rule is an abstractly incorrect rule.*

The check of Definition 9 (as claimed above) is not effective. This task can be (partially) accomplished by an automatic tool by choosing a suitable under-approximation $\mathcal{I}^c$ of the specification $\mathcal{I}$, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$ (hence $\alpha(\mathcal{I}) \sqsubseteq \mathcal{I}^c$), and checking the behavior of an abstractly incorrect rule against it.

Definition 11. *Let r be a program rule. Then r is* provably incorrect *using α if $\mathcal{I}^{\alpha} \not\sqsubseteq T^{\alpha}_{\{r\}}(\mathcal{I}^c)$.*

Definition 12. *Let $\mathcal{R}$ be a program. Then $\mathcal{R}$ has* provably uncovered *elements using α if $T^{\alpha}_{\mathcal{R}}(\mathcal{I}^{\alpha}) \not\sqsubseteq \mathcal{I}^c$.*

The name "provably incorrect using α" is justified by the following theorem.

Theorem 5. *Let r be a program rule and $\mathcal{I}^c$ such that $(\alpha\gamma)(\mathcal{I}^c) = \mathcal{I}^c$. Then if r is provably incorrect using α it is also incorrect.*

Thus by choosing a suitable under-approximation we can refine the check for wrong rules. For all abstractly incorrect rules we check if they are provably incorrect using α. If it so then we report an error, otherwise we can just issue a warning.

As we will see in the following, this property holds (for example) for our case study. By Proposition 3 the condition $(\alpha\gamma)(\mathcal{I}^c) = \mathcal{I}^c$ is trivially satisfied by any ground subset of the over-approximation. Thus we will consider the best choice which is the biggest ground subset of the over-approximation.

Theorem 6. *Let $\mathcal{R}$ be a program. If $\mathcal{R}$ has a provably uncovered element using α, then $\mathcal{R}$ is not complete.*

Abstract uncovered elements are provably uncovered using α. However, Theorem 6 allows us to catch other incompleteness bugs that cannot be detected by using Theorem 3 since there are provably uncovered elements using α which are not abstractly uncovered.

The diagnosis w.r.t. approximate properties is always effective, because the abstract specification is finite. As one can expect, the results may be weaker than those that can be achieved on concrete domains just because of approximation. Namely,

- absence of abstractly incorrect rules implies partial correctness,
- every incorrectness error is identified by an abstractly incorrect rule. However an abstractly incorrect rule does not always correspond to a bug. Anyway,
- every abstractly incorrect rule which is provably incorrect using α corresponds to an error.
- provably uncovered equations always correspond to incompleteness bugs.
- there exists no sufficient condition for completeness.

The results are useful and comparable to those obtained by verification techniques (see, for example, [3,2]). In fact, if we consider the case where specifications consist of post-conditions only, both abstract diagnosis and verification provide *a sufficient condition for partial correctness*, which is well-assertedness in the case of verification and absence of incorrect rules in abstract diagnosis. For both techniques there is no sufficient condition for completeness. In order to verify completeness, we have to rely on a fixpoint (the model of a transformed program or the abstraction of the concrete semantics), which, in general, cannot be computed in a finite number of steps. As expected, abstract diagnosis (whose aim is locating bugs rather than just proving correctness) gives us also information useful for debugging, by means of provably incorrect rules using α and provably uncovered equations using α.

5.1 Our Case Study

We can derive an efficient debugger which is based on the notion of over-approximation and under-approximation for the intended fixpoint semantics that we have introduced. The basic idea is to consider two sets to verify partial correctness and determine program bugs: $\mathcal{I}^\alpha$ which over-approximates the intended semantics $\mathcal{I}$ (that is, $\mathcal{I} \subseteq \gamma(\mathcal{I}^\alpha)$) and $\mathcal{I}^c$ which under-approximates $\mathcal{I}$ (that is, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$).

Now we show how we can derive an efficient debugger by choosing suitable instances of the general framework described above. We consider as α the $depth(k)$

abstraction κ of the set of values of the TRS that we have defined in previous section. Thus we choose $\mathcal{I}^\kappa = \mathcal{F}^\kappa_{\mathsf{val}}(\mathcal{I})$ as an over-approximation of the values of a program. We can consider any of the sets defined in the works of [6,10] as an under-approximation of $\mathcal{I}$. In concrete, we take the "ground" abstract equations of $\mathcal{I}^\kappa$ as $\mathcal{I}^c$. This provides a simple albeit useful debugging scheme which is satisfactory in practice.

The methodology enforced by previous results (in particular Theorem 5) has been implemented by a prototype system DEBUSSY, which is available at

`http://www.dsic.upv.es/users/elp/soft.html`

The systems is implemented in Haskell and debugs programs written in OBJ style w.r.t. a formal specification also written in OBJ. The current version only considers the evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$. The tool takes advantage from the sorting information that may be provided within the programs to construct the (least) sets of blocking terms and equations which are used to generate the approximation of the semantics. The user interface uses textual menus which are (hopefully) self-explaining. A detailed description of the system can be found at the same address.

Let us illustrate the method by using the guiding example.

Example 3. Let us reconsider the TRS $\mathcal{R}$ (program `ERRDOUBLE`) and the intended specification $\mathcal{I}$ (program `ADDDOUBLE`) in Example 1 and let us see how it can be debugged by DEBUSSY. The debugging session returns:

```
Incorrect rules in ERRDOUBLE :

double(s(X)) -> double(X)

Uncovered equations from ADDDOUBLE :

add(0,0) = 0
```

where the second rule is correctly identified as erroneous.

In this example, we also show the over and under-approximation computed by the system. Here, we consider a cut at depth 2.

1. Over-approximation $\mathcal{I}^\alpha = \mathcal{I}^2 = \widetilde{\mathcal{F}}^2_{\mathsf{val}}(\mathcal{I}) = \widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow \omega$.

$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 0 = \mathsf{val} = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{s(0)} = \mathtt{s(0)},\ \mathtt{s(s(\hat{x}))} = \mathtt{s(s(\hat{x}))}\ \}$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 1 =$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 0 \cup \{\ \mathtt{add(0,0)} = \mathtt{0},\ \mathtt{add(0,s(\hat{x}))} = \mathtt{s(0)},\ \mathtt{s(add(\hat{x},\hat{y}))}$
$= \mathtt{s(0)},\ \mathtt{add(0,s(\hat{x}))} = \mathtt{s(s(\hat{y}))},\ \mathtt{s(add(\hat{x},\hat{y}))} = \mathtt{s(s(\hat{z}))}\ \}$

$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 2 =$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 1 \cup \{$ double(0) = 0, add(0,add($\hat{x}$,$\hat{y}$)) = 0, add(add($\hat{x}$,$\hat{y}$),0) = 0, add(0,add($\hat{x}$,$\hat{y}$)) = s(0), add(add($\hat{x}$,$\hat{y}$),s($\hat{z}$)) = s(0), s(double($\hat{x}$)) = s(0), add(s($\hat{x}$),$\hat{y}$) = s(0), add(0,add($\hat{x}$,$\hat{y}$)) = s(s($\hat{z}$)), add(add($\hat{x}$,$\hat{y}$),s($\hat{z}$)) = s(s($\hat{w}$)), s(double($\hat{x}$)) = s(s($\hat{y}$)), add(s($\hat{x}$),$\hat{y}$) = s(s($\hat{z}$)) $\}$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 3 =$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 2 \cup \{$ add(0,double($\hat{x}$)) = 0, double(add($\hat{x}$,$\hat{y}$)) = 0, add(add($\hat{x}$,$\hat{y}$),add($\hat{z}$,$\hat{w}$)) = 0, add(double($\hat{x}$),0) = 0, add(0,double($\hat{x}$)) = s(0), add(add($\hat{x}$,$\hat{y}$),add($\hat{z}$,$\hat{w}$)) = s(0), add(double($\hat{x}$),s($\hat{y}$)) = s(0), double(s($\hat{x}$)) = s(0), add(add($\hat{x}$,$\hat{y}$),$\hat{z}$) = s(0), add(0,double($\hat{x}$)) = s(s($\hat{y}$)), add(add($\hat{x}$,$\hat{y}$),add($\hat{z}$,$\hat{w}$)) = s(s($\hat{v}$)), add(double($\hat{x}$),s($\hat{y}$)) = s(s($\hat{z}$)), double(s($\hat{x}$)) = s(s($\hat{y}$)), add(add($\hat{x}$,$\hat{y}$),$\hat{z}$) = s(s($\hat{w}$)) $\}$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 4 =$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 3 \cup \{$ add(add($\hat{x}$,$\hat{y}$),double($\hat{z}$)) = 0, double(double($\hat{x}$)) = 0, add(double($\hat{x}$),add($\hat{y}$,$\hat{z}$)) = 0, add(add($\hat{x}$,$\hat{y}$),double($\hat{z}$)) = s(0), double(add($\hat{x}$,$\hat{y}$)) = s(0), add(double($\hat{x}$),add($\hat{y}$,$\hat{z}$)) = s(0), add(double($\hat{x}$),$\hat{y}$) = s(0), add(add($\hat{x}$,$\hat{y}$),double($\hat{z}$)) = s(s($\hat{w}$)), double(add($\hat{x}$,$\hat{y}$)) = s(s($\hat{z}$)), add(double($\hat{x}$),add($\hat{y}$,$\hat{z}$)) = s(s($\hat{w}$)), add(double($\hat{x}$),$\hat{y}$) = s(s($\hat{w}$)) $\}$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 5 =$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 4 \cup \{$ add(double($\hat{x}$),double($\hat{y}$)) = 0, add(double($\hat{x}$),double($\hat{y}$)) = s(0), double(double($\hat{x}$)) = s(0), add(double($\hat{x}$),double($\hat{y}$)) = s(s($\hat{z}$)), double(double($\hat{x}$)) = s(s($\hat{y}$)) $\}$
$\widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow \omega = \widetilde{T}^2_{\mathcal{I},\mathsf{val}} \uparrow 5$

2. Under-approximation $\mathcal{I}^c$, which is the ground part of $\mathcal{I}^\alpha$. Note that $\mathcal{I}^c = (\alpha\gamma)(\mathcal{I}^c)$.

   $\mathcal{I}^c = \{$ 0 = 0, s(0) = s(0), add(0,0) = 0, double(0) = 0 $\}$

The selection of the appropriate depth for the abstraction is a sensitive point of our approach. The following theorem shows that a threshold depth k exists such that smaller depths are unfeasible to consider.

Theorem 7. *Let $l \to r$ be a program rule and $k > 0$, If $r/_k \ncong r$, then $l \to r$ is not provably incorrect using k.*

Definition 13. *Let $\mathcal{R}$ be a TRS. Depth k is called* admissible to diagnose $\mathcal{R}$ *if for all $l \to r \in \mathcal{R}$, $r/_k \cong r$.*

Obviously, admissible depths do not generally guarantee that all program bugs are recognized, since the abstraction might not be precise enough, as illustrated in the following example. The question of whether an optimal depth exists such that no additional errors are detected by considering deeper cuts is an interesting open problem in our approach which we plan to investigate as further work.

Example 4. Consider the following (wrong) OBJ program $\mathcal{R}$ for doubling and the specification of the intended semantics (program `ADDDOUBLE`) of Example 1.

```
obj ERRDOUBLE2 is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op double : Nat -> Nat .
  var X : Nat .
  eq double(0) = s(0) .
  eq double(s(X)) = s(double(X)) .
endo
```

The execution of DEBUSSY for program `ERRDOUBLE2` and specification `ADDDOUBLE` is:

```
Incorrect rules in ERRDOUBLE2 :

double(0) -> s(0)

Uncovered equations from ADDDOUBLE :

add(0,0) = 0
double(0) = 0
```

Note that 2 is the smaller admissible depth for this program. When depth $k = 1$ is considered, rule `double(0)` $\rightarrow$ `s(0)` can not be proven to be incorrect using the under-approximation $\mathcal{I}^c = \{\texttt{0} = \texttt{0}\}$ since equation `s(0)` = `s(0)` does not belong to $\mathcal{I}^c$. For instance, by using $k = 2$, the debugger is not able to determine that rule `double(s(X))` $\rightarrow$ `s(double(X))` is incorrect.

6 Conclusions

We have presented a generic scheme for the declarative debugging of functional programs. Our approach is based on the ideas of [10,1] which we apply to the diagnosis of functional programs. We have presented a fixpoint semantics $T_{\mathcal{R}}$ for functional programs. Our semantics allows us to model the (evaluation/normalization) semantics of the TRS in a bottom-up manner. Thus, it is a suitable basis for dataflow analyses based on abstract interpretation as we illustrated. This methodology is superior to the abstract rewriting methodology of [5], which requires canonicity, stratification, constructor discipline, and

complete definedness for the analyses. We have developed a prototype Haskell implementation of our debugging method for functional programs, and we have used it for debugging the examples presented in this work. Nevertheless, more experimentation is needed in order to assess how our methodology performs in comparison to other tools for revealing errors in functional programs such as QuickCheck [7].

Some topics for further research are to develop specialized analyses for particular languages, such as those in the OBJ family. We also plan to endow DEBUSSY with some inductive learning capabilities which allow us to repair program bugs by automatically synthesizing the correction from the examples which can be generated as an outcome of the diagnoser.

References

1. M. Alpuente, F. J. Correa, and M. Falaschi. Declarative Debugging of Functional Logic Programs. In B. Gramlich and S. Lucas, editors, *Proceedings of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, North Holland, 2001. Elsevier Science Publishers.
2. K. R. Apt. *From Logic Programming to PROLOG*. Prentice-Hall, 1997.
3. 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.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. D. Bert and R. Echahed. Abstraction of Conditional Term Rewriting Systems. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 162–176, Cambridge, Mass., 1995. The MIT Press.
6. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169, Linköping, Sweden, 1997. University of Linköping Press.
7. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, 35(9):268–279, 2000.
8. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building Equational Proving Tools by Reflection in Rewriting Logic. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 1–32. Elsevier, 2000.
9. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, *Proceedings of Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop (LOMAPS'96)*, volume 1192 of *Lecture Notes in Computer Science*, pages 22–50, Berlin, 1996. Springer-Verlag.
10. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
11. M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 275–287, Cambridge, Mass., 1995. The MIT Press.

12. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
13. F. Durán. Termination Checker and Knuth-Bendix Completion Tools for Maude Equational Specifications. Technical report, Universidad de Málaga, July 2000.
14. 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.
15. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.
16. G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
17. J. A. Goguen and G. Malcom. *Software Engineering with OBJ*. Kluwer Academic Publishers, Boston, 2000.
18. J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.
19. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
20. M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.
21. H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 36 – 47. ACM Press, 1999.
22. H. Nilsson and P. Fritzson. Algoritmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370, 1994.
23. J. T. O'Donell and C. V. Hall. Debugging in Applicative Languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
24. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
25. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, MA, 1993.
26. E. Y. Shapiro. Algorithmic Program Debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.
27. M.B. Smyth. Power Domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.
28. J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional languages. In M. Ducassé, editor, *Proceedings Second International Workshop on Automated and Algorithmic Debugging, AADEBUG'95*, 1995.
29. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. A. Tärnlund, editor, *Proceedings of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
30. I. Toyn. *Exploratory Environments for Functional Programming*. PhD thesis, University of York, U.K., 1987.
31. P. Wadler. Functional Programming: An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, 1998.

A Cut-Free Sequent Calculus for Pure Type Systems Verifying the Structural Rules of Gentzen/Kleene*

Francisco Gutiérrez and Blas Ruiz

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga. Campus Teatinos 29071, Málaga, Spain
{pacog,blas}@lcc.uma.es

Abstract. In this paper, a new notion for sequent calculus (à la Gentzen) for Pure Type Systems (PTS) is introduced. This new calculus, $\mathcal{K}$, is equivalent to the standard PTS, and it has a cut-free subsystem, $\mathcal{K}^{\mathsf{cf}}$, that will be proved to hold non-trivial properties such as the structural rules of Gentzen/Kleene: thinning, contraction, and interchange.
An interpretation of completeness of the $\mathcal{K}^{\mathsf{cf}}$ system yields the concept of *Cut Elimination*, (CE), and it is an essential technique in proof theory; thus we think that it will have a deep impact on PTS and in logical frameworks based on PTS.

Keywords: lambda calculus with types, pure type systems, sequent calculi, cut elimination.

1 Introduction

Pure Type Systems (PTS) [1,2] provide a flexible and general framework to study dependent type system properties. Moreover, PTS also include many interesting systems. These systems are the basis for logical frameworks and proof-assistants that heavily use dependent types [3,4].

The proposed sequent calculi for PTS are based on a correspondence between Gentzen's natural deduction and sequent calculus intuitionistic logics [5]. In order to obtain a sequent calculus from the type inference relation in a PTS, the (apl) rule (or Π elimination rule) that types applications has to be dispensed with, since it eliminates the Π connective (see Fig. 1). For different typing disciplines (intersection types) and non dependent types, other authors [6,7,8,9] employ a modification of Gentzen's $(\to L)$ rule of sequent calculus:

$$(\to L)\ \frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\,.$$

Contrarily, we consider an adaptation of the left rule used by Kleene [10, p. 481] in the G_3 system:

$$\frac{A \to B, \Gamma \vdash A \qquad A \to B, B, \Gamma \vdash C}{A \to B, \Gamma \vdash C}\,.$$

* This research was partially supported by the project TIC2001-2705-C03-02.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 17–31, 2003.

Definition 1 *We consider the rule:*

$$(\mathcal{K})\ \frac{\Gamma \vdash a : A \qquad \Gamma, x : S, \Delta \vdash c : C}{\Gamma, \Delta[x := y\,a] \vdash c[x := y\,a] : C[x := y\,a]} \quad \begin{cases} y : \Pi z : A.B \in \Gamma, \\ S =_\beta B[z := a]. \end{cases}$$

$\mathcal{K}$ (for Kleene) denotes the system obtained by replacing the (apl) rule (see Fig. 1) of the standard PTS *by the (cut) and ($\mathcal{K}$) rules (see Fig. 2). The type inference relation of $\mathcal{K}$ will be denoted by $\vdash_{\mathcal{K}}$. Similarly, $\mathcal{K}^{\mathsf{cf}}$ denotes the system obtained by eliminating the (cut) rule. Its type inference relation will be denoted by $\vdash_{\mathsf{cf}}$.*

By using the Howard-Curry-de Bruijn correspondence, the $\mathcal{K}^{\mathsf{cf}}$ system takes on a role similar to Kleene's G_3 system. In this way, the choice of this rule avoids facing usual problems since contexts will not be sets but sequences of declarations. The reason is that $\mathcal{K}^{\mathsf{cf}}$ satisfies rules similar to Gentzen/Kleene's TCI structural rules for managing contexts: thinning, contraction/condensation, and interchange/permutation. As a consequence of thinning, $\mathcal{K}$ is proved to be equivalent to the standard PTS, and hence, the $\vdash_{\mathsf{cf}}$ typing inference relation is a reformulation of the $\vdash$ inference relation in the Gentzen style.

Using the fact that the $[x := y\,a]$ substitution operator preserves normal forms, it is easily derived by induction on derivations (IDs) that $\mathcal{K}^{\mathsf{cf}}$ yields normalized types, contexts, and terms, i.e.:

$$\Gamma \vdash_{\mathsf{cf}} c : C \Rightarrow \Gamma, c, C \text{ are in normal form.}$$

Can the (*cut*) rule be eliminated? If so, will we obtain the same relation for normalized contexts, subjects, and predicates?

Thus, *cut elimination* is enunciated as:

$$\Gamma, c, C \text{ are in normal form, then } \Gamma \vdash_{\mathcal{K}} c : C \ \Rightarrow\ \Gamma \vdash_{\mathsf{cf}} c : C. \qquad (CE)$$

The previous statement is a result similar to Gentzen's *Hauptsatz*: every LJ derivation can be obtained without using the (*cut*) rule.

Cut elimination is an essential technique in proof theory; and it will have a deep impact on PTS. Thus, CE can be applied to develop proof-search techniques with dependent types, similar to those proposed in [11,9,3].

2 Description and Properties of PTS

In this section we review PTS and their main properties. For further details the reader is referred to [1,2,12,13].

Considering an infinite set of variables $\mathcal{V}$ ($x, y, \cdots \in \mathcal{V}$) and a set of constants or *sorts* $\mathcal{S}$ ($s, s', \cdots \in \mathcal{S}$), the set $\mathcal{T}$ of terms for a PTS is inductively defined as:

$$\begin{array}{ll} a \in \mathcal{V} \cup \mathcal{S} & \Rightarrow\ a \in \mathcal{T}, \\ A, C, a, b \in \mathcal{T} & \Rightarrow\ a\ b,\ \lambda x : A.b,\ \Pi x : A.C \in \mathcal{T}. \end{array}$$

We denote the β-reduction as $\twoheadrightarrow_\beta$ and the equality generated by $\twoheadrightarrow_\beta$ as $=_\beta$. The set of β-normal forms is denoted β-nf and $\mathrm{FV}(a)$ denotes the set of

free variables for a. As usual $A[x := B]$ denotes substitution. The relation $\twoheadrightarrow_\beta$ is *Church-Rosser* (CR). a_β denotes the β-normal form of a. A *context* Γ is a ordered sequence (possibly empty) $\langle x_1 : A_1, \ldots, x_n : A_n\rangle$ of declarations $x_i : A_i$ where $x_i \in \mathcal{V}$ and $A_i \in \mathcal{T}$. We drop the symbols $\langle\rangle$ if there is no ambiguity. We write $x_i : A_i \in \Gamma$ if the declaration $x_i : A_i$ is in Γ, and using the $(\doteq)$ symbol to establish definitions, we set

$$\begin{array}{lcl} \Gamma \subseteq \Gamma' & \doteq & \forall x \in \mathcal{V}\ [x : A \in \Gamma\ \Rightarrow\ x : A \in \Gamma'], \\ \mathrm{Var}(\Gamma) & \doteq & \{x_1, \ldots, x_n\}, \\ \mathrm{FV}(\Gamma) & \doteq & \bigcup_{1\le i\le n} \mathrm{FV}(A_i). \end{array}$$

A PTS is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a set of constants or *sorts*, $\mathcal{A} \subseteq \mathcal{S}^2$ a set of *axioms*, and $\mathcal{R} \subseteq \mathcal{S}^3$ a set of *rules*. The standard notion of derivation $\Gamma \vdash a : A$ is defined by the inductive system shown in Fig. 1. We say that Γ is a legal context (denoted by $\Gamma \vdash$) if $\exists c, C[\Gamma \vdash c : C]$.

We recall elementary properties of PTS:

Lemma 2 (Elementary properties) *If $\Gamma \vdash c : C$, then:*

(*i*) $\mathsf{FV}(c : C) \subseteq \mathsf{Var}(\Gamma)$, *and if* $x_i, x_j \in \mathrm{Var}(\Gamma)$, *then* $i \neq j \Rightarrow x_i \neq x_j$. ($FrVrs$)
(*ii*) $s_1 : s_2 \in \mathcal{A} \Rightarrow \Gamma \vdash s_1 : s_2$. ($TypAx$)
(*iii*) $y : D \in \Gamma \Rightarrow \Gamma \vdash y : D$. ($TypVr$)
(*iv*) *Type correctness:* $\Gamma \vdash c : C \wedge C \notin \mathcal{S}\ \Rightarrow\ \exists s \in \mathcal{S}\ [\Gamma \vdash C : s]$. ($CrTyps$)
(*v*) *Context correctness:* $\Gamma, x : A, \Delta \vdash d : D\ \Rightarrow\ \exists s \in \mathcal{S}\ [\Gamma \vdash A : s]$. ($CrCtx$)

Recall that a PTS is normalizing if it verifies $\Gamma \vdash a : A \Rightarrow a$ is weak normalizing (also, by type correctness, A is weak normalizing).

We also need typical properties of PTS: subject β-reduction ($S\beta$), predicate β-reduction ($P\beta$), the substitution lemma (Sbs), and the thinning lemma ($Thnng$):

$$\frac{\Gamma \vdash a : A \quad a \twoheadrightarrow_\beta a'}{\Gamma \vdash a' : A}\ (S\beta), \qquad \frac{\Gamma \vdash a : A \quad A \twoheadrightarrow_\beta A'}{\Gamma \vdash a : A'}\ (P\beta),$$

$$\frac{\Gamma \vdash d : D \qquad \Gamma, y : D, \Delta \vdash c : C}{\Gamma, \Delta[y := d] \vdash c[y := d] : C[y := d]}\ (Sbs), \quad \frac{\Gamma \vdash b : B \quad \Gamma\ \subseteq\ \Psi \quad \Psi \vdash}{\Psi \vdash b : B}\ (Thnng).$$

Every free *object* on the right hand side of an implication or in the conclusion of a rule is existentially quantified. For example, the $CrCtx$ property of Lemma 2 can be enunciated as: $\Gamma, x : A, \Delta \vdash d : D\ \Rightarrow\ \Gamma \vdash A : s$.
In our proofs we also use the (generation) Lemma 5.2.13 described in [2].

The (cut) rule is obtained taking $\Delta \equiv \langle\rangle$ in Sbs:

$$(cut)\ \frac{\Gamma \vdash d : D \qquad \Gamma, y : D \vdash c : C}{\Gamma \vdash c[y := d] : C[y := d]}\ .$$

$$(ax) \quad \frac{}{\vdash s_1 : s_2} \qquad s_1 : s_2 \in \mathcal{A}$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad x \notin \mathsf{Var}(\Gamma)$$

$$(weak) \quad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \qquad b \in \mathcal{S} \cup \mathcal{V}, x \notin \mathsf{Var}(\Gamma)$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B \ : \ s_3} \qquad (s_1, s_2, s_3) \in \mathcal{R}$$

$$(apl) \quad \frac{\Gamma \vdash f : \Pi x : A.F \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : F[x := a]}$$

$$(\lambda) \quad \frac{\Gamma \vdash \Pi x : A.B : s \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b \ : \ \Pi x : A.B}$$

$$(\beta) \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A' : s}{\Gamma \vdash a : A'} \qquad A =_\beta A'$$

Fig. 1. Inference rules for PTS. $s_1 : s_2 \in \mathcal{A}$ stands for $(s_1, s_2) \in \mathcal{A}$.

The standard PTS will be denoted by $\mathcal{N}$ (for natural) (i.e.: the relation $\vdash$ in Fig. 1). Recall that the (apl) rule can be replaced by the rule:

$$(apl') \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : \Pi z : A.B \qquad \Gamma \vdash B[z := a] : s}{\Gamma \vdash f\ a : B[z := a]}\,.$$

This is easy to prove, by using $CrTyps$, the generation lemma and Sbs. The rule (apl') will be used to simplify the proof of completeness of $\mathcal{K}$ system.

The two lemmas below are rarely referred to in the literature; however, they will be used here to simplify some proofs.

Lemma 3 (van Benthem Jutting [12]) *The set of terms of a* PTS, $\mathcal{T}$, *can be divided into two disjoint classes* T_v *and* T_s, *inductively defined as:*

$$\begin{array}{ll} x \in T_v, & \text{and } A, c \in \mathcal{T} \wedge b \in T_v \ \Rightarrow\ b\,c, \lambda x : A.b \in T_v, \\ s, \Pi x : A.B \in T_s, & \text{and } A, c \in \mathcal{T} \wedge b \in T_s \ \Rightarrow\ b\,c, \lambda x : A.b \in T_s, \end{array}$$

where $x \in \mathcal{V}$ *and* $s \in \mathcal{S}$, *so that*

$$\Gamma \vdash a : A \wedge \Gamma \vdash a : A' \Rightarrow \begin{cases} a \in T_v \ \Rightarrow\ A =_\beta A', \\ a \in T_s \ \Rightarrow\ \exists s, s' \in \mathcal{S}\ [A \twoheadrightarrow_\beta \Pi\Delta.s \wedge A' \twoheadrightarrow_\beta \Pi\Delta.s'], \end{cases}$$

where $\Pi\langle\rangle.M \doteq M$ *and* $\Pi\langle x : X, \Delta\rangle.M \doteq \Pi x : X.(\Pi\Delta.M)$.

Lemma 4 (Legal supercontexts) *For every system (or relation) $\vdash_\star$ satisfying context correctness, the (var) rule, and thinning, it holds that:*

$$\frac{\Gamma, \Delta \vdash_\star \qquad \Gamma' \vdash_\star}{\Gamma', \Delta \vdash_\star}, \qquad \Gamma \subseteq \Gamma', \qquad \mathsf{Var}(\Gamma') \cap \mathsf{Var}(\Delta) = \emptyset.$$

Proof It easily follows by induction on the length of the context Δ. □

3 Sequent Calculi for PTS. Properties

By Definition 1, in order to obtain a sequent calculus, we have considered the $\mathcal{K}$ system obtained from the $\mathcal{N}$ system by removing the (*apl*) rule and then by adding the (κ) and (*cut*) rule (see Fig. 2). In addition, we define the cut-free system $\mathcal{K}^{\mathsf{cf}}$ by removing the (*cut*) rule, obviously, $\mathcal{K}^{\mathsf{cf}} \subseteq \mathcal{K}$ [1].

In this section, we will prove that the $\mathcal{K}$ system is equivalent to the natural $\mathcal{N}$ system. In order to prove this, elementary properties of PTS must be particularized for these sequent calculi. In addition, we will prove that the system $\mathcal{K}^{\mathsf{cf}}$ verifies the structural rules of Gentzen/Kleene.

Lemma 5 *Lemma 2 holds for $\mathcal{K}$ and $\mathcal{K}^{\mathsf{cf}}$ systems.*

Proof We only prove (*iv*) by induction over the derivation (ID) $\Gamma \vdash_{\mathcal{K}} c : C$. If the last applied rule is ($\mathcal{K}$) with $C[x := y\,a] \notin \mathcal{S}$, then $C \notin \mathcal{S}$, and by induction hypothesis (IH), $\Gamma, x : S, \Delta \vdash_{\mathcal{K}} C : s$, and then the ($\mathcal{K}$) rule is applied. The other cases are proved in a similar way. □

The property (*v*) of Lemma 5 allow us to include $\Gamma \vdash S : s$ in the premise of ($\mathcal{K}$) rule.

Lemma 6 (Correctness of sequent calculi) $\mathcal{K}^{\mathsf{cf}} \subseteq \mathcal{K} \subseteq \mathcal{N}$.

Proof Since $\mathcal{N}$ satisfies the substitution lemma, it also satisfies (*cut*), so it suffices to prove that the ($\mathcal{K}$) rule holds for $\mathcal{N}$. Assume:

$$\Gamma \vdash a : A, \qquad \Gamma, x : S, \Delta \vdash c : C, \qquad y : \Pi z : A.B \in \Gamma, \qquad S =_\beta B[z := a].$$

Then, by applying *CrCtx* and *TypVr* (Lemma 2) we follow the derivation:

$$\dfrac{\Gamma \vdash S : s \qquad \dfrac{\Gamma \vdash y : \Pi z : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash y\ a : B[z := a]}\,(apl)}{\dfrac{\Gamma \vdash y\ a : S \qquad\qquad \Gamma, x : S, \Delta \vdash c : C}{\Gamma, \Delta[x := y\ a] \vdash c[x := y\ a] : C[x := y\ a].}\,Sbs}\,(\beta) \quad \square$$

Because of the form of the ($\mathcal{K}$) rule, every object (subject, context, and type) typable in $\vdash_{\mathsf{cf}}$ is normal. In fact,

[1] In the sequel, we use $\mathcal{S}_1 \subseteq \mathcal{S}_2$ to stand for $\vdash_{\mathcal{S}_1} \subseteq \vdash_{\mathcal{S}_2}$ and $\mathcal{S}_1 \equiv \mathcal{S}_2$ for $\vdash_{\mathcal{S}_1} \equiv \vdash_{\mathcal{S}_2}$.

$$(ax)\quad \frac{}{\vdash_{\mathcal{K}} s_1 : s_2} \qquad s_1 : s_2 \in \mathcal{A}$$

$$(var)\quad \frac{\Gamma \vdash_{\mathcal{K}} A : s}{\Gamma, x : A \vdash_{\mathcal{K}} x : A} \qquad x \notin \mathsf{Var}(\Gamma)$$

$$(weak)\quad \frac{\Gamma \vdash_{\mathcal{K}} b : B \qquad \Gamma \vdash_{\mathcal{K}} A : s}{\Gamma, x : A \vdash_{\mathcal{K}} b : B} \qquad b \in \mathcal{S} \cup \mathcal{V}, x \notin \mathsf{Var}(\Gamma)$$

$$(\Pi)\quad \frac{\Gamma \vdash_{\mathcal{K}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathcal{K}} B : s_2}{\Gamma \vdash_{\mathcal{K}} \Pi x : A.B \ : \ s_3} \qquad (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\mathcal{K})\quad \frac{\Gamma \vdash_{\mathcal{K}} a : A \qquad \Gamma, x : S, \Delta \vdash_{\mathcal{K}} c : C}{\Gamma, \Delta[x := y\, a] \vdash_{\mathcal{K}} c[x := y\, a] : C[x := y\, a]} \quad \begin{cases} y : \Pi z : A.B \in \Gamma \\ S =_\beta B[z := a] \end{cases}$$

$$(cut)\quad \frac{\Gamma \vdash_{\mathcal{K}} d : D \qquad \Gamma, y : D \vdash_{\mathcal{K}} c : C}{\Gamma \vdash_{\mathcal{K}} c[y := d] : C[y := d]}$$

$$(\lambda)\quad \frac{\Gamma \vdash_{\mathcal{K}} \Pi x : A.B : s \qquad \Gamma, x : A \vdash_{\mathcal{K}} b : B}{\Gamma \vdash_{\mathcal{K}} \lambda x : A.b \ : \ \Pi x : A.B}$$

$$(\beta)\quad \frac{\Gamma \vdash_{\mathcal{K}} a : A \qquad \Gamma \vdash_{\mathcal{K}} A' : s}{\Gamma \vdash_{\mathcal{K}} a : A'} \qquad A =_\beta A'$$

Fig. 2. Inference rules for the $\mathcal{K}$ system.

Lemma 7 (The shape of types) *In $\mathcal{K}^{\mathsf{cf}}$ system, it holds that:*

(i) $\Gamma \vdash_{\mathsf{cf}} m : M \ \Rightarrow \ \Gamma, m, M \in \beta\text{-}\mathsf{nf}$.
(ii) $a \in T_v \wedge \Gamma \vdash_{\mathsf{cf}} a : A, a : A' \ \Rightarrow \ A \equiv A'$.
(iii) $a \in T_s \wedge \Gamma \vdash_{\mathsf{cf}} a : A, a : A' \ \Rightarrow \ \exists s, s' \in \mathcal{S}\ [A \equiv \Pi\Delta.s \wedge A' \equiv \Pi\Delta.s']$.
(iv) $\Gamma \vdash_{\mathsf{cf}} m : M \ \Rightarrow \ \mathsf{FV}(M) \subseteq \mathsf{FV}(m)$.

Proof (i) It follows by IDs (induction on derivations) using the fact that the $[x := y\, a]$ operator preserves normal forms when $a \in \beta\text{-}\mathsf{nf}$. In order to prove $(ii) - (iii)$, it suffices to apply Lemma 6 and then Lemma 3 and (i). (iv) In [14,13] it is proved that every PTS satisfies the property $\Gamma \vdash m : M \wedge u \notin \mathsf{FV}(m) \Rightarrow M \twoheadrightarrow_\beta M' \wedge u \notin \mathsf{FV}(M')$ that easily follows by IDs. It suffices to apply $\vdash_{\mathsf{cf}} \subseteq \vdash$ together with (i). □

The previous result provides powerful tools to simplify quite a number of proofs: from (i) we obtain that in $\mathcal{K}^{\mathsf{cf}}$ we can dispense with the (β) rule since it does not yield different types. The rest of the lemma characterizes the types as in Lemma 3 but for normalized types.

We will now prove the completeness property: $\mathcal{N} \ \subseteq \ \mathcal{K}$. Our proof uses thinning, a property hard to state since the context Δ occurs in the $(\mathcal{K})$ rule. For this reason, two auxiliary systems are introduced taking $\Delta \equiv \langle\rangle$.

Definition 8 *The system κ is obtained from $\mathcal{K}$ by replacing the $(\mathcal{K})$ rule by:*

$$(\kappa)\ \frac{\Gamma \vdash a : A \qquad \Gamma, x : S \vdash c : C}{\Gamma \vdash c[x := y\,a] : C[x := y\,a]}\ \begin{cases} y : \Pi z : A.B \in \Gamma, \\ S =_\beta B[z := a]. \end{cases}$$

The system κ^{cf} is similarly defined.

The only difference between the $\mathcal{K}$ and κ systems is that the context Δ in the $(\mathcal{K})$ rule does not appear in the (κ) rule. It is easy to prove that Lemma 5 and Lemma 7 hold for κ and κ^{cf} systems. In addition, it is obvious that $k \subseteq \mathcal{K}$ and $\kappa^{\mathsf{cf}} \subseteq \mathcal{K}^{\mathsf{cf}}$.

Lemma 9 (Thinning in κ and κ^{cf}) *For $\vdash_\star \equiv \vdash_\kappa$ or $\vdash_\star \equiv \vdash_{\kappa^{\mathsf{cf}}}$,*

$$\Gamma \vdash_\star m : M \wedge \Gamma \subseteq \Delta \wedge \Delta \vdash_\star \Rightarrow \Delta \vdash_\star m : M.$$

Proof By ID (induction over the derivation) $\varphi \equiv \Gamma \vdash_\star m : M$. Only two cases are shown, the others being standard. By $CrCtx$, we can introduce $\Gamma \vdash_\star S : s$ as a premise in the (κ) rule. If φ has been inferred from the (κ) rule, we apply (var) in order to obtain $\Delta, x : S \vdash_\star$ from IH; the rest is straightforward. If φ has been inferred from the (cut) rule

$$\frac{\Gamma, y : D \vdash_\star c : C \qquad \Gamma \vdash_\star d : D}{\Gamma \vdash_\star c[y := d] : C[y := d]},$$

by IH we have that $\Delta \vdash_\star d : D$; in order to apply the rule one more time we need to prove $\Delta, y : D \vdash_\star s$, $s \in \mathcal{S}$, which is a consequence of $\Delta \vdash_\star D : s$. By applying $CrTyps$ to the derivation $\Delta \vdash_\star d : D$, it suffices to prove the case $D \in \mathcal{S}$. However, since $\Gamma, y : D \vdash_\star c : C$, by context correctness we have that $\Gamma \vdash_\star D : s$, and since $\vdash_\star \subseteq \vdash$, we have that $\Gamma \vdash D : s$. Now, since D is a constant, the generation lemma is applied (in $\vdash$) to obtain $D : s' \in \mathcal{A}$. Finally, since $\Delta \vdash_\star$, we apply Lemma 5(ii) yielding $\Delta \vdash_\star D : s'$. □

Corollary 10 (Completeness of sequent calculus) $\mathcal{N} \equiv \mathcal{K}$.

Proof To complete the proof of $\mathcal{N} \subseteq \kappa \subseteq \mathcal{K} \subseteq \mathcal{N}$, the first inclusion has to be proved. In order to do so, it suffices to prove that (apl') can be applied in κ. For the sake of readability of the proof, we let Π stand for $\Pi z : A.B$ and B_a for $B[z := a]$. Assume $\Gamma \vdash_\kappa f : \Pi$, $\Gamma \vdash_\kappa a : A$ and $\Gamma \vdash_\kappa B_a : s$. To prove $\Gamma \vdash_\kappa f\ a : B_a$ we follow the derivation:

$$\dfrac{\Gamma \vdash_\kappa f : \Pi \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash_\kappa f : \Pi}{\Gamma \vdash_\kappa \Pi : s'}\,\mathrm{CrTyps}}{\Gamma, y : \Pi \vdash_\kappa y : \Pi}\,(var) \qquad \Gamma \vdash_\kappa B_a : s}{\Gamma, y : \Pi \vdash_\kappa B_a : s}\,\mathrm{Thnng} \qquad \dfrac{\dfrac{\Gamma, y : \Pi \vdash_\kappa \Gamma \vdash_\kappa a : A}{\Gamma, y : \Pi \vdash_\kappa a : A}\,\mathrm{Thnng} \qquad \dfrac{\Gamma, y : \Pi \vdash_\kappa B_a : s}{\Gamma, y : \Pi, x : B_a \vdash_\kappa x : B_a}\,(var)}{\Gamma, y : \Pi \vdash_\kappa y\,a : B_a}\,(\kappa)}{\Gamma \vdash_\kappa f\,a : B_a.}\,(cut)$$

□

Lemma 11 (Thinning in $\mathcal{K}^{\mathsf{cf}}$) *The system κ^{cf} satisfies the $(\mathcal{K})$ rule, and therefore $\kappa^{\mathsf{cf}} \equiv \mathcal{K}^{\mathsf{cf}}$. Also, by Lemma 9, $\mathcal{K}^{\mathsf{cf}}$ verifies thinning.*

Proof We now prove the implication:

$$\Rightarrow \quad \begin{array}{l} \Gamma \vdash_{\kappa^{\mathsf{cf}}} a : A \qquad \wedge \qquad \Gamma, x : S, \Delta \vdash_{\kappa^{\mathsf{cf}}} c : C \\ \\ \Gamma, \Delta[x := y\, a] \vdash_{\kappa^{\mathsf{cf}}} c[x := y\, a] : C[x := y\, a] \end{array} \quad \text{if } \begin{cases} y : \Pi z : A.B \in \Gamma, \\ S =_\beta B[z := a]. \end{cases}$$

Let Γ' be $\Gamma, x : S, \Delta$. We proceed by ID $\Gamma' \vdash_{\kappa^{\mathsf{cf}}} c : C$. We will show the most interesting case: when the last rule applied was (κ). Assume:

$$\frac{\Gamma' \vdash_{\kappa^{\mathsf{cf}}} q : Q \qquad \Gamma', x' : S' \vdash_{\kappa^{\mathsf{cf}}} m : M}{\Gamma' \vdash_{\kappa^{\mathsf{cf}}} m[x' := y'\, q] : M[x' := y'\, q]} \quad \begin{cases} y' : \Pi t : Q.R \in \Gamma', \\ S' =_\beta R[t := q]; \end{cases}$$

then we have to prove that:

$$\Gamma, \Delta[x := y\, a] \vdash_{\kappa^{\mathsf{cf}}} m[x' := y'\, q][x := y\, a] : M[x' := y'\, q][x := y\, a]. \tag{1}$$

By applying IH twice we have that:

$$\Gamma, \Delta^\circ \vdash_{\kappa^{\mathsf{cf}}} q^\circ : Q^\circ, \qquad \Gamma, \Delta^\circ, x' : S'^\circ \vdash_{\kappa^{\mathsf{cf}}} m^\circ : M^\circ, \tag{2}$$

where $^\circ$ stands for $[x := y\, a]$. Three cases are now distinguished:

1. $y' \not\equiv x$ with $y' : \_ \in \Delta$. Therefore, $y' : \Pi t : Q^\circ.R^\circ \in \Gamma, \Delta^\circ$, and since we had $S'^\circ =_\beta R^\circ[t := q^\circ]$, by applying the (κ) rule we get

   $$\Gamma, \Delta^\circ \vdash_{\kappa^{\mathsf{cf}}} m^\circ[x' := y'\, q^\circ] : M^\circ[x' := y'\, q^\circ].$$

   However $x' \notin \mathsf{FV}(y\, a)$; we can thus apply the substitution lemma of the untyped λ-calculus [2, Lemma 2.1.6] to get (1).
2. $y' \not\equiv x$ with $y' : \_ \in \Gamma$. The reasoning is similar.
3. $y' \equiv x$. In this case $S \equiv \Pi t : Q.R$, and x does not occur free in $\Pi t : Q.R$. By $CrCtx$ and Lemma 4 in κ^{cf} a fresh variable ϵ can be introduced in the derivations in (2):

   $$\Gamma, \Delta^\circ, \epsilon : \Pi t : Q.R \vdash_{\kappa^{\mathsf{cf}}} q^\circ : Q \qquad \Gamma, \Delta^\circ, \epsilon : \Pi t : Q.R, x' : S'^\circ \vdash_{\kappa^{\mathsf{cf}}} m^\circ : M^\circ.$$

   The (κ) rule can now be applied to get:

   $$\Gamma, \Delta^\circ, \epsilon : \Pi t : Q.R \vdash_{\kappa^{\mathsf{cf}}} m^\circ[x' := \epsilon\, q^\circ] : M^\circ[x' := \epsilon\, q^\circ].$$

   Since $S \equiv \Pi t : Q.R$, we can remove the variable introduced ϵ by applying the (κ) rule:

   $$\Gamma, \Delta^\circ \vdash_{\kappa^{\mathsf{cf}}} m^\circ[x' := \epsilon\, q^\circ][\epsilon := y\, a] : M^\circ[x' := \epsilon\, q^\circ][\epsilon := y\, a]. \tag{3}$$

   It suffices to prove that the sequence of substitutions in (3) is that of (1). $\square$

The following result provides strengthening in cut free sequent calculi. The proof uses the same property as in the $\mathcal{N}$ natural system, together with the generation lemma.

Theorem 12 *The system* $\mathcal{K}^{\mathsf{cf}}$ *satisfies the* **strong strengthening** *property:*

$$\frac{\Delta \vdash_{\mathsf{cf}} m : M}{\Delta\backslash\Psi \vdash_{\mathsf{cf}} m : M} \qquad \mathsf{FV}(m : M) \cup \mathsf{FV}(\Delta\backslash\Psi) \subseteq \mathsf{Var}(\Delta\backslash\Psi). \qquad (Strng_S)$$

Proof It is well-known that the strengthening property holds for every PTS [12,13]:

$$\frac{\Gamma, u : U, \Gamma' \vdash m : M}{\Gamma, \Gamma' \vdash m : M} \qquad u \notin \mathsf{FV}(\Gamma') \cup \mathsf{FV}(m : M). \qquad (Strng)$$

The proviso in $Strng_S$ is equivalent to $v \in \mathsf{Var}(\Psi) \Rightarrow v \notin \mathsf{FV}(m : M) \cup \mathsf{FV}(\Delta\backslash\Psi)$; and hence, by applying $Strng$, every declaration in Ψ is removed from Δ right to left. Then, for $\mathcal{N}$ and $\mathcal{K}^{\mathsf{cf}}$ systems, $Strng_S$ is a consequence of $Strng$. Thus, we need to prove that $Strng$ holds in $\mathcal{K}^{\mathsf{cf}}$. For this purpose, if we suppose the property

$$\frac{\Delta \vdash_{\mathsf{cf}} m : M}{\Delta\backslash\Psi \vdash_{\mathsf{cf}} m : M} \quad \Delta\backslash\Psi \vdash m : M', \qquad (4)$$

then, by Lemma 7(iv), $Strng$ in $\mathcal{K}^{\mathsf{cf}}$ is straightforward. To prove (4) we assume the premise of the rule and take $\Gamma \equiv \Delta\backslash\Psi$. We prove $\Gamma \vdash_{\mathsf{cf}} m : M$ by IDs. The most interesting case is when the last rule applied is (κ) (recall Lemma 11):

$$\frac{\Delta \vdash_{\mathsf{cf}} a : A \qquad \Delta, x : S \vdash_{\mathsf{cf}} c : C}{\Delta \vdash_{\mathsf{cf}} c[x := y\, a] : C[x := y\, a]} \quad \begin{cases} y : \Pi z : A.B \in \Delta, \\ S =_\beta B[z := a]. \end{cases}$$

Assume $\Gamma \vdash c[x := y\, a] : C'$. If $x \notin \mathsf{FV}(c)$, by Lemma 7(iv), $x \notin \mathsf{FV}(C)$, and then we had to prove $\Gamma \vdash_{\mathsf{cf}} c : C$. But this is a consequence of $(\Delta, x : S)\backslash(\Psi, x : S) \vdash c : C'$ together with $\Delta, x : S \vdash_{\mathsf{cf}} c : C$, and IH. Thus, $x \in \mathsf{FV}(c)$ is assumed instead, and hence $y\, a$ is a subterm of $c[x := y\, a]$. Therefore, $\mathsf{FV}(y) \cup \mathsf{FV}(a) \subseteq \mathsf{FV}(c[x := y\ a])$, and by Lemma 5$(i)$:

$$y : \Pi z : A.B \in \Gamma \qquad \mathsf{FV}(a) \subseteq \mathsf{Var}(\Gamma). \qquad (5)$$

But Γ is a legal context and $y : \Pi z : A.B \in \Gamma$, then $\mathsf{FV}(\Pi z : A.B) \subseteq \mathsf{Var}(\Gamma)$ and also

$$\mathsf{FV}(A), \mathsf{FV}(B) \subseteq \mathsf{Var}(\Gamma), \qquad (6)$$

and hence $\mathsf{FV}(a : A) \cup \mathsf{FV}(\Gamma) \subseteq \mathsf{Var}(\Gamma)$. Strengthening (via $(Strng_S)$ in $\mathcal{K}$ system) the derivation $\Delta \vdash a : A$ we get $\Gamma \vdash a : A$, and by IH we have $\Gamma \vdash_{\mathsf{cf}} a : A$. In order to apply the (κ) rule again, we have to prove $\Gamma, x : S \vdash_{\mathsf{cf}} c : C$. The last judgment is a consequence of $(Strgn_S)$ together with $\Delta, x : S \vdash c : C$, and IH. So that strengthening $\Delta, x : S \vdash c : C$ we need to prove

$$\mathsf{FV}(c) \subseteq \mathsf{Var}(\Gamma, x : S), \qquad (7)$$

$$\mathsf{FV}(C) \subseteq \mathsf{Var}(\Gamma, x : S), \tag{8}$$

$$\mathsf{FV}(\Gamma, x : S) \subseteq \mathsf{Var}(\Gamma, x : S). \tag{9}$$

(We assume that x is a fresh variable, and then $\Gamma, x : S \equiv (\Delta, x : S)\backslash\Psi$). However, by $\Gamma \vdash c[x := y\ a] : C'$, we have $\mathsf{FV}(c) \subseteq \mathsf{FV}(c[x := y\ a]) \cup \{x\}$ and it is easy to get (7) using Lemma 5(i).

To prove (8) we need to apply Lemma 7(iv) to the derivation $\Delta, x : S \vdash_{\mathsf{cf}} c : C$ in order to obtain $\mathsf{FV}(C) \subseteq \mathsf{FV}(c)$ and then to apply (7).

Finally, we prove (9). This is a consequence of $\mathsf{FV}(S) \subseteq \mathsf{Var}(\Gamma)$ and the previous discussion. We know $\Delta, x : S \vdash_{\mathsf{cf}} c : C$. Then, by applying Lemma 7(i), we obtain $S \in \beta\text{-}\mathsf{nf}$. On the other hand, we have $S =_\beta B[z := a]$; therefore $\mathsf{FV}(S) \subseteq \mathsf{FV}(B[z := a])$. But $\mathsf{FV}(B[z := a]) \subseteq \mathsf{FV}(B) \cup \mathsf{FV}(a)$. Now, we apply (5) and (6) to get $\mathsf{FV}(B[z := a]) \subseteq \mathsf{Var}(\Gamma)$. □

Theorem 13 (Permutation) *The system $\mathcal{K}^{\mathsf{cf}}$ satisfies*[2]*:*

$$\frac{\Gamma, u : U, \Delta, \Gamma' \vdash_{\mathsf{cf}} m : M}{\Gamma, \Delta, u : U, \Gamma' \vdash_{\mathsf{cf}} m : M}, \qquad u \notin \mathsf{FV}(\Delta).$$

Proof It suffices to prove that $\Gamma, \Delta, u : U, \Gamma' \vdash_{\mathsf{cf}}$ and apply thinning to the premise. For the former to be proved, Lemma 4 has to be applied to $\Gamma, \Delta, u : U \vdash_{\mathsf{cf}}$, that can be proved to be legal assuming $\Gamma, u : U, \Delta, \Gamma' \vdash_{\mathsf{cf}} m : M$, as follows: by correctness of contexts, we have that $\Gamma \vdash_{\mathsf{cf}} U : s''$ and $\Gamma, u : U, \Delta \vdash_{\mathsf{cf}} s : s'$, where $s : s'$ is an axiom. We first apply strengthening to get $\Gamma, \Delta \vdash_{\mathsf{cf}}$, and then thinning to get $\Gamma, \Delta \vdash_{\mathsf{cf}} U : s''$. Finally, we apply ($var$) to obtain $\Gamma, \Delta, u : U \vdash_{\mathsf{cf}}$. □

Theorem 14 (Contraction) *The system $\mathcal{K}^{\mathsf{cf}}$ satisfies:*

$$\frac{\Gamma, y : A, \Delta \vdash_{\mathsf{cf}} c : C}{\Gamma, \Delta[y := x] \vdash_{\mathsf{cf}} c[y := x] : C[y := x]}, \qquad x : A \in \Gamma. \tag{$CnTrc$}$$

Proof We proceed by ID $\Gamma, y : A, \Delta \vdash_{\mathsf{cf}} c : C$. We will show the most interesting case: when the last rule applied was (κ) (Recall by Lemma 11 that $\kappa^{\mathsf{cf}} \equiv \mathcal{K}^{\mathsf{cf}}$). Let Ψ be $\Gamma, y : A, \Delta$. Assume:

$$\frac{\Psi \vdash_{\mathsf{cf}} q : Q \qquad \Psi, x' : S' \vdash_{\mathsf{cf}} m : M}{\Psi \vdash_{\mathsf{cf}} m[x' := y'\ q] : M[x' := y'\ q]} \quad \begin{cases} y' : \Pi t : Q.R \in \Psi, \\ S' =_\beta R[t := q], \end{cases}$$

then we have to prove that:

$$\Gamma, \Delta[y := x] \vdash_{\mathsf{cf}} m[x' := y'\ q][y := x] : M[x' := y'\ q][y := x]. \tag{10}$$

By applying IH twice we have that:

$$\Gamma, \Delta^\circ \vdash_{\mathsf{cf}} q^\circ : Q^\circ, \qquad \Gamma, \Delta^\circ, x' : S'^\circ \vdash_{\mathsf{cf}} m^\circ : M^\circ, \tag{11}$$

where $^\circ$ stands for $[y := x]$. Three cases are now distinguished:

[2] This theorem holds for every system satisfying correctness of contexts, the (var) rule, thinning, and strengthening.

1. $y' \neq y$ with $y' : \_ \in \Delta$. Therefore, $y' : \Pi t : Q^\circ.R^\circ \in \Gamma, \Delta^\circ$, and since we had $S'^\circ =_\beta R^\circ[t := q^\circ]$ and (11), by applying the (κ) rule we get

$$\Gamma, \Delta^\circ \vdash_{\mathsf{cf}} m^\circ[x' := y'\, q^\circ] : M^\circ[x' := y'\, q^\circ].$$

   However $x' \neq x$; then we can apply the substitution lemma of the untyped λ-calculus to get $Z^\circ[x' := y'\, q^\circ] \equiv Z[x' := y'\, q]^\circ$, which completes the proof (10).
2. $y' \neq y$ with $y' : \_ \in \Gamma$. The reasoning is similar.
3. $y' \equiv y$. In this case $A \equiv \Pi t : Q.R$, and $x : \Pi t : Q.R \in \Gamma, \Delta$, and the (κ) rule can be applied to get:

$$\Gamma, \Delta^\circ \vdash_{\mathsf{cf}} m^\circ[x' := x\, q^\circ] : M^\circ[x' := x\, q^\circ]. \tag{12}$$

   It suffices to note that (12) and (10) are identical substitutions. □

Corollary 15 *The cut-free sequent calculus $\mathcal{K}^{\mathsf{cf}}$ verifies the TCI structural rules of Gentzen/Kleene [10, p. 443].*

Remark 16 (Typing applications in sequent calculi for PTS**)** In the rest of this section, the choice of $(\mathcal{K})$ rule is justified.

Let $\vdash$ be the typing relation in a PTS. We will look for a sequent calculus $\vdash_{\mathsf{cf}}$ such that: (S) it is sound, and (N) it infers normalized objects, and is as close as possible to the original system. By (N) the (β) rule can be dispensed with. On the other hand, the application rule must be discarded since it removes Π from the premise. For CE to hold, a rule to type applications must be added. Every normalized application has the form $(x\, f_2 \dots f_n)[x := y\, f_1]$, and the following rule schema suffices to type it:

$$(\Pi\, left)\ \frac{\dots \qquad \Psi \vdash c : C}{\Psi' \vdash c[x := y\, a] : C[x := y\, a]}\,.$$

This rule is particularly useful when $x \in \mathsf{FV}(c)$; by applying (S) and Lemma $5(i)$ we get $x : S \in \Psi$. By the same reasoning, the variable y must occur in Ψ', whereas the variable x is no longer needed in Ψ'. By (S) and (N) and uniqueness of types for the variables (similar to Lemma $7(ii)$), $y : \Pi z : A.B \in \Psi'$ can be assumed. If y does not occur in the contexts of the premises of the rule schema above, then the $(\Pi\, left)$ rule must introduce the judgement $y : \Pi z : A.B$.

It may be argued that extending $\mathcal{K}^{\mathsf{cf}}$ with rules such as $(\Pi\, left)$ with proviso $y \notin \mathsf{Var}(\Psi)$ can make the proof of CE easier. However, the system so defined does not produce additional derivations. It should be noted that the Π connective could be introduced in $\mathcal{K}^{\mathsf{cf}}$ by the (var) and $(weak)$ rules.

To illustrate this, two generic rules removing the declaration $x : S$ and introducing $y : \Pi z : A.B$ in any position, either on the left or on the right of the removed declaration, are defined:

$$(\Pi\ell_1)\ \frac{\Gamma, \Gamma', x : S, \Delta \vdash c : C \qquad \Gamma \vdash a : A, \Pi z : A.B : s}{\Gamma, y : \Pi z : A.B, \Gamma', \Delta^\circ \vdash c^\circ : C^\circ} \quad \begin{cases} y \text{ fresh}, \\ S =_\beta B[z := a]. \end{cases}$$

$$(\Pi\ell_2)\ \frac{\Gamma, x : S, \Delta, \Delta' \vdash c : C \qquad \Gamma, \Delta \vdash a : A, \Pi z : A.B : s}{\Gamma, \Delta, y : \Pi z : A.B, \Delta'^{\circ} \vdash c^{\circ} : C^{\circ}} \quad \begin{cases} y \text{ fresh}, \\ x \notin \mathsf{FV}(\Delta), \\ S =_\beta B[z := a], \end{cases}$$

where $^{\circ} = [x := y\,a]$. Surprisingly, these rules are superfluous in $\mathcal{K}^{\mathsf{cf}}$:
1. The $(\Pi\ell_1)$ rule is admissible in $\mathcal{K}^{\mathsf{cf}}$, as shown in the following proof:

$$\dfrac{\Gamma \vdash a : A \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{cf}} \Pi : s}{\Gamma, y : \Pi \vdash_{\mathsf{cf}}}(var) \qquad \overbrace{\Gamma, \Gamma', x : S, \Delta}^{\Psi} \vdash_{\mathsf{cf}}}{\Gamma, y : \Pi, \Psi \vdash_{\mathsf{cf}}}\text{Lm 4} \qquad \Gamma, \Psi \vdash_{\mathsf{cf}} c : C}{\Gamma, y : \Pi, \Psi \vdash_{\mathsf{cf}} c : C}\,Thnng}{\Gamma, y : \Pi, \Gamma', \Delta^{\circ} \vdash_{\mathsf{cf}} c^{\circ} : C^{\circ}.}\,(\mathcal{K})$$

2. The $(\Pi\ell_2)$ rule is inferrable in $\mathcal{K}^{\mathsf{cf}}$, as shown in the following proof:

$$\dfrac{\Gamma, \Delta \vdash_{\mathsf{cf}} a : A \qquad \dfrac{\dfrac{(\Psi \equiv)\Gamma, \Delta \vdash_{\mathsf{cf}} \Pi : s}{\Psi, y : \Pi \vdash_{\mathsf{cf}}}(var) \qquad \dfrac{\Psi \vdash_{\mathsf{cf}} \qquad \Gamma, x : S, \Delta, \Delta' \vdash_{\mathsf{cf}} c : C}{\Psi, x : S, \Delta' \vdash_{\mathsf{cf}} c : C}\text{Lm 13}}{\Psi, y : \Pi, x : S, \Delta' \vdash_{\mathsf{cf}} c : C}\text{Lm 4}}{\Psi, y : \Pi, \Delta_2^{\circ} \vdash_{\mathsf{cf}} c^{\circ} : C^{\circ}.}\,(\mathcal{K})\ \square$$

A complete sequent calculus not satisfying cut elimination. Gentzen's $(\to L)$ rule must be carefully adapted. An immediate adaptation yields the rule

$$(\Pi\mathcal{L})\ \frac{\Gamma \vdash a : A \qquad \Gamma, x : S, \Delta \vdash c : C \qquad \Gamma \vdash \Pi z : A.B : s}{\Gamma, y : \Pi z : A.B, \Delta[x := y\,a] \vdash c[x := y\,a] : C[x := y\,a]} \quad \begin{cases} y \text{ fresh}, \\ S =_\beta B[z := a], \end{cases}$$

that is a particularization of the $(\Pi\ell_1)$ rule. $\mathcal{L}$ and $\mathcal{L}^{\mathsf{cf}}$ denotes the systems obtained by replacing the $(\mathcal{K})$ rule by $(\Pi\mathcal{L})$. By Remark 16, $\mathcal{L} \subseteq \mathcal{K} \equiv \mathcal{N}$. It must be noted that every instance of these system including the simple typed PTS $\lambda\!\to$ ([2, p. 215] does not satisfy cut elimination. The reason is that in $\lambda\!\to$ it is possible to infer:

$$A : *, q : A, y : A \to A \vdash y\,(y\,q) : A,$$

but $y\,(y\,q)$ cannot be typed in $\mathcal{L}^{\mathsf{cf}}$. In fact, the derivation $\Gamma \vdash_{\mathcal{L}^{\mathsf{cf}}} y\,(y\,q) : \_$ can only be obtained by the $(\Pi\mathcal{L})$ rule. However, if $c[x := y\,a] \equiv y\,(y\,q)$, only two cases have to be considered:
— $c \equiv x$, $a \equiv y\,q$. This can never hold, since $\Gamma \vdash_{\mathcal{L}^{\mathsf{cf}}} (a \equiv) y\,q : A$ with $y \notin \mathsf{Var}(\Gamma)$,
— $c \equiv y\,N$. This can never hold: $\Gamma, x : S, \Delta \vdash_{\mathcal{L}^{\mathsf{cf}}} y\,N : C$ with $y \notin \mathsf{Var}(\Gamma, x : S)$.
And therefore *cut elimination* does not hold. Curiously enough, adding the (cut) rule yields a system equivalent to the original one:

Lemma 17 $\mathcal{L} \equiv \mathcal{N}$.

Proof We need to prove $\mathcal{N} \subseteq \mathcal{L}$. It suffices to prove that the (apl') rule can be applied in $\mathcal{L}$. The $CrTyps$ property in $\mathcal{L}$ can be easily proved following a reasoning similar to that of Lemma 5. Now, let Π be $\Pi z : A.B$ and B_a be $B[z := a]$, then we follow the schema

$$\dfrac{\dfrac{\Gamma \vdash_{\mathcal{L}} a : A \qquad \dfrac{\Gamma \vdash_{\mathcal{L}} f : \Pi}{\Gamma \vdash_{\mathcal{L}} \Pi : s'}\, CrTyps \qquad \dfrac{\Gamma \vdash_{\mathcal{L}} B_a : s}{\Gamma, x : B_a \vdash_{\mathcal{L}} x : B_a}\,(var)}{\Gamma \vdash_{\mathcal{L}} f : \Pi \qquad \Gamma, y : \Pi \vdash_{\mathcal{L}} y\ a : B_a}\,(\Pi\mathcal{L})}{\Gamma \vdash_{\mathcal{L}} f\ a : B_a.\ \square}\,(cut)$$

We obtain similar results when we use $(\Pi\ell_1)$ or $(\Pi\ell_2)$ instead of $(\Pi\mathcal{L})$.

4 Related Works and Conclusions

Kleene stresses in [10, § 80] that the main advantage of the G_3 system is that the TCI structural rules are omitted. This makes it possible to describe an efficient decision algorithm for provability. Similarly, it is possible to design an efficient algorithm for type-checking (and proof-search) in cut-free PTS.

Over the last 70 years since Gerhard Gentzen's *Hauptsatz* [5], dozens of proofs of cut elimination for sequent calculi have appeared in the literature (see [15,16] for a survey). When lacking dependent types, there are a few previous proofs of cut elimination for sequent calculi with proof-terms that use a *left* rule similar to Gentzen's calculi. We can emphasize the proofs proposed in [15,6,17] for intersection and union types. As a remark, the proof by Yokouchi [17] considers sequential contexts, thus additional rules are needed to ensure contraction and permutation. In this paper we proved that an adaptation of the left rule used by Kleene provides TCI structural rules for managing contexts.

Gentzen's $(\to L)$ rule must be carefully adapted. We proved that an immediate adaptation yields the rule $(\Pi\mathcal{L})$, but every instance of $\mathcal{L}$ system including the PTS $\lambda\!\to$ does not satisfy cut elimination.

Though the proof of cut elimination for generic sequent calculi with proof-terms is known to be difficult, it is even a more intricate matter when some other features are considered, such as dependent types or sequential contexts. However, the problems stemming from the latter feature have been solved.

In [18] (an extension of this paper) it is proved that CE is equivalent to the admissibility of a rule to type applications of the form $y\,q$ in the system $\mathcal{K}^{\mathsf{cf}}$. As a result, CE is obtained in two families of systems characterized as follows. On the one hand, those PTS where in every rule $(s_1, s_2, s_3) \in \mathcal{R}$, the constant s_2 does not occur in the right hand side of an axiom. Thus, we obtain proofs of CE in the corners $\lambda\!\to$ and $\lambda 2$ of the λ-cube. In addition, since $\lambda 2 \approx PROP2$ [19, p. 151], thanks to the Howard-Curry-de Bruijn isomorphism, cut elimination for minimal implicational, second order sequent calculus is obtained, thus generalizing the result in [7].

On the other hand, some PTS are a subclass of order functional PTS [20]. These systems are weaker than the functional ones, and are characterized by

the possibility of extending for every term, a function (order) $\partial : \mathcal{S} \to \mathbb{Z}$ that enumerates every sort. When it is the case that $\partial s_1 = \partial s_2 = \partial s_3$ (for every rule (s_1, s_2, s_3)) these systems are particularly interesting as they are Π-independent: the well-typed dependent products $\Pi z : A.B$ satisfy $z \notin \mathsf{FV}(B)$. This result, together with strengthening in $\mathcal{K}$, yield CE as a simple corollary. The corners $\lambda\!\!\rightarrow$ and $\lambda\underline{\omega}$ of Barendregt's λ-cube are particular cases.

Also included in [18] there is an analysis of a Strong Cut Elimination (SCE) property, that can be defined taking a rule similar to ($\mathcal{K}$) but replacing $S \equiv B[z := a]$ in the proviso. The new systems $\mathcal{K}'$ and $\mathcal{K}^{\mathsf{cf}'}$ verifies the properties analyzed in this paper. A generation lemma for the $\mathcal{K}^{\mathsf{cf}'}$ system allows to refute SCE for the remaining systems in the λ-cube, as well as in other interesting systems: λHOL, λAUT_QE, $\lambda AUT-68$, and λPAL, all of them described in [2, p. 216].

As far as we know, there is no other proof of (weak) cut elimination in generic normalizing dependent type systems similar to PTS.

CE is an open problem for generic normalized systems. This is not surprising, and we have prove that CE is actually harder that the *Expansion Postponement* (EP) problem [21], posed by Henk Barendregt in August 1990. The relevance of EP stems from on its application to the correctness proof of certain type checking systems. Except for PTS with important restrictions, EP is thus far an open problem, even for normalizing PTS [22,23]. It is well-known that EP can be solved by the completeness of a certain system $\mathcal{N}_\beta$ that infers normal types only [21,13]. In [18] we have proved that CE ensures that $\mathcal{K}$ is correct with respect to $\mathcal{N}_\beta$, and therefore EP is easily obtained.

Acknowledgments

The authors are very grateful to Herman Geuvers for suggesting a typing rule inspiring our ($\mathcal{K}$) rule, Gilles Barthe for his valuable comments on this work, and Pablo López for helping in translating this paper.

References

1. H. Geuvers, M. Nederhof, Modular proof of Strong Normalization for the Calculus of Constructions, Journal of Functional Programming 1 (1991) 15–189.
2. H. P. Barendregt, Lambda Calculi with Types, in: S. Abramsky, D. Gabbay, T. S. Maibaum (Eds.), Handbook of Logic in Computer Science, Oxford University Press, 1992, Ch. 2.2, pp. 117–309.
3. F. Pfenning, Logical frameworks, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Vol. II, Elsevier Science, 2001, Ch. 17, pp. 1063–1147.
4. H. Barendregt, H. Geuvers, Proof-assistants using dependent type systems, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Vol. II, Elsevier Science, 2001, Ch. 18, pp. 1149–1238.
5. G. Gentzen, Untersuchungen über das Logische Schliessen, Math. Zeitschrift 39 (1935) 176,–210,405–431, translation in [24].

6. F. Barbanera, M. Dezani-Ciancaglini, U. de'Liguoro, Intersection and union types: Syntax and semantics, Information and Computation 119 (2) (1995) 202–230.
7. H. P. Barendregt, S. Ghilezan, Lambda terms for natural deduction, secuent calculus and cut elimination, Journal of Functional Programming 10 (1) (2000) 121–134.
8. M. Baaz, A. Leitsch, Comparing the complexity of cut-elimination methods, Lecture Notes in Computer Science 2183 (2001) 49–67.
9. D. Galmiche, D. J. Pym, Proof-search in type-theoretic languages: an introduction, Theoretical Computer Science 232 (1–2) (2000) 5–53.
10. S. C. Kleene, Introduction to Metamathematics, D. van Nostrand, Princeton, New Jersey, 1952.
11. D. Pym, A note on the proof theory of the $\lambda\Pi$–calculus, Studia Logica 54 (1995) 199–230.
12. L. van Benthem Jutting, Typing in Pure Type Systems, Information and Computation 105 (1) (1993) 30–41.
13. B. C. Ruiz, Sistemas de Tipos Puros con Universos, Ph.D. thesis, Universidad de Málaga (1999).
14. B. C. Ruiz, Condensing lemmas in Pure Type Systems with Universes, in: A. M. Haeberer (Ed.), 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98) Proceedings, Vol. 1548 of LNCS, Springer–Verlag, 1999, pp. 422–437.
15. J. Gallier, Constructive logics. I. A tutorial on proof systems and typed lambda -calculi, Theoretical Computer Science 110 (2) (1993) 249–339.
16. M. Baaz, A. Leitsch, Methods of cut elimination, Tec. rep., 11th European Summer School in Logic, Language and Information. Utrecht University (August 9-20 1999). URL `http://www.let.uu.nl/esslli/Courses/baaz-leitsch.html`
17. H. Yokouchi, Completeness of type assignment systems with intersection, union, and type quantifiers, Theoretical Computer Science 272 (2002) 341–398.
18. F. Gutiérrez, B. C. Ruiz, Sequent Calculi for Pure Type Systems, Tech. Report 06/02, Dept. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (Spain), `http://polaris.lcc.uma.es/~blas/ publicaciones/` (may 2002).
19. H. Geuvers, Logics and type systems, Ph.D. thesis, Computer Science Institute, Katholieke Universiteit Nijmegen (1993).
20. F. Gutiérrez, B. C. Ruiz, Order functional PTS, in: 11th International Workshop on Functional and Logic Programming (WFLP'2002), Vol. 76 of ENTCS, Elsevier, 2002, pp. 1–16, `http://www.elsevier.com/gej-ng/31/29/23/126/23/23/76012.pdf`.
21. E. Poll, Expansion Postponement for Normalising Pure Type Systems, Journal of Functional Programming 8 (1) (1998) 89–96.
22. B. C. Ruiz, The Expansion Postponement Problem for Pure Type Systems with Universes, in: 9th International Workshop on Functional and Logic Programming (WFLP'2000), Dpto. de Sistemas Informáticos y Computación, Technical University of Valencia (Tech. Rep.), 2000, pp. 210–224, september 28-30, Benicassim, Spain.
23. G. Barthe, B. Ruiz, Tipos Principales y Cierre Semi-completo para Sistemas de Tipos Puros Extendidos, in: 2001 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'01), Évora, Portugal, 2001, pp. 149–163.
24. G. Gentzen, Investigations into logical deductions, in: M. Szabo (Ed.), The Collected Papers of Gerhard Gentzen, North-Holland, 1969, pp. 68–131.

Constraint Solver Synthesis Using Tabled Resolution for Constraint Logic Programming

Slim Abdennadher[1] and Christophe Rigotti[2,⋆]

[1] Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de
[2] Laboratoire d'Ingénierie des Systèmes d'Information
Bâtiment 501, INSA Lyon, 69621 Villeurbanne Cedex, France
Christophe.Rigotti@insa-lyon.fr

Abstract. The goal of automated program synthesis is to bridge the gap between what is easy for people to describe and what is possible to execute on a computer. In this paper, we present a framework for synthesis of rule-based solvers for constraints given their logical specification. This approach takes advantage of the power of tabled resolution for constraint logic programming, in order to check the validity of the rules. Compared to previous work [8,19,2,5,3], where different methods for automatic generation of constraint solvers have been proposed, our approach enables the generation of more expressive rules (even recursive and splitting rules).

1 Introduction

Program synthesis research aims at maximally automating the passage from specifications to programs [9]. In the field of constraint solving, several methods have been proposed to automatically generate rule-based solvers for constraints given their logical specification [8,19,2,5,3]. It has also been shown in [4,5] that these rules generated automatically can lead to more efficient constraint reasoning than rules found by programmers.

In general, a rule-based constraint solver consists of rules to simplify constraints and rules to propagate new constraints. The first kind of rules, called *simplification rules*, are rules that rewrite constraints to simpler constraints while preserving logical equivalence. The rules of the second kind, called *propagation rules*, are used to add new constraints, which are logically redundant but which may cause further simplification.

Example 1. The following rule-based solver implements the constraint *min*, where $min(X, Y, Z)$ means that Z is the minimum of X and Y:

$$min(X, Y, Z),\ Y \leq X \ \Leftrightarrow\ Z = Y,\ Y \leq X.$$

⋆ The research reported in this paper has been supported by the Bavarian-French Hochschulzentrum.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 32–47, 2003.

$$min(X,Y,Z),\ X{\leq}Y \ \Leftrightarrow\ Z{=}X,\ X{\leq}Y.$$
$$min(X,Y,Z) \ \Rightarrow\ Z{\leq}X,\ Z{\leq}Y.$$

The first two rules are simplification rules, while the third one is a propagation rule. The first rule states that $min(X,Y,Z),\ Y{\leq}X$ can be simplified into $Z{=}Y,\ Y{\leq}X$. Analogously for the second rule. The third rule states that $min(X,Y,Z)$ implies $Z{\leq}X,\ Z{\leq}Y$. Operationally, we add these logical consequences as redundant constraints, the *min* constraint is kept. □

In this work, we propose a new method to automatically generate the propagation and simplification rules taking advantage from tabled resolution of constraint logic programming (CLP). Just like in logic programming (e.g., [21,22]), tabling is receiving increasing attention in the CLP community (e.g., [10,11]). A CLP system with tabling techniques can avoid redundant computations and thus many infinite loops.

The basic idea of our approach relies on the following observation: rules of the form $C \Rightarrow false$ are valid if the execution of the goal C finitely fails. These rules can be triggered to detect inconsistencies of a conjunction of constraints. However, rules can be much more interesting if they can be used earlier to propagate additional constraints. For example, it may be more advantageous when a goal $C, \neg(d)$ fails to generate the propagation rule $C \Rightarrow d$ than the rule $C, \neg(d) \Rightarrow false$. For the execution of these goals, we use a tabled resolution for CLP that terminates more often than execution based on SLD-like resolution.

In this paper, we present three algorithms that can be integrated to build an environment for generating rule-based constraint solvers. Two of the algorithms focus on how to generate propagation rules for constraints given their logical specification. The first algorithm generates only primitive propagation rules, i.e. rules with right hand side consisting of primitive constraints. Primitive constraints are those constraints for which solvers are already available. The second algorithm slightly modifies the first one to generate more general propagation rules with right hand side consisting of primitive and user-defined constraints. User-defined constraints are those defined by a constraint logic program. We also show that a slight extension of this algorithm allows us to generate the so-called splitting rules. The third algorithm focuses on transforming propagation rules into simplification rules to improve the time and space behavior of constraint solving. It takes advantage from the modification done in the second algorithm.

The generated rules can be used to suggest to a constraint solver designer interesting propagation and simplification over user-defined constraints, but it should be noticed that they can be also directly encoded in a rule-based programming language, e.g. Constraint Handling Rules (CHR) [12] to provide a running implementation.

Related Work

In [3], a method has been proposed to generate propagation rules from the intentional definition of the constraint predicates (eventually over infinite domains)

given by means of a constraint logic program. This method extended previous work [8,19,2,5] where different methods for automatic generation of propagation rules for constraints defined extensionally over finite domains have been proposed. The main idea of the method was to perform the execution of a possible left hand side of a rule by calling a CLP system. The right hand side of the rule consists of the least general generalization of the set of computed answers. Compared to this method, the approach presented in this paper has many advantages:

- It leads to a more expressive set of rules. For example, the rule for the well-known ternary *append* predicate for lists

$$append(X, Y, Z),\ Y{=}[]\ \Rightarrow\ X{=}Z.$$

  cannot be generated by the approach presented in [3] since the execution of the goal $append(X, Y, Z),\ Y{=}[]$ will lead to an infinite set of answers and thus compromises the computation of their least general generalization. The algorithm described in Section 2 is able to generate the above rule just by executing the goal $append(X, Y, Z),\ Y{=}[], X{\neq}Z$ with a tabled resolution for CLP.
  As by-product of the method presented here, rules representing symmetries can be automatically detected. For example, the rule

$$min(X, Y, Z)\ \Rightarrow\ min(Y, X, Z).$$

  expressing the symmetry of the *minimum* predicate with respect to the first and second arguments can be generated. In [3], it has been shown that these rules are very useful to reduce the size of a set of propagation rules since many rules become redundant when we know such symmetries. In [3], these rules cannot be generated automatically, but have to be provided by hand.
- It allows the generation of the so-called splitting rules. This is a new kind of rules that has not been considered in [8,19,2,5,3]. Splitting rules have been shown to be interesting in constraint solving, since they can be used to detect early alternative labeling cases or alternative solution sets. For example, the following rule handles the case when the third argument of *append* is a singleton list:

$$append(X, Y, Z),\ Z{=}[A]\ \Rightarrow\ X{=}[A]\ \vee\ Y{=}[A]$$

- It avoids the computation of the least general generalization (lgg) which is often a hard problem: To generate the rule

$$min(X, Y, Z)\ \Rightarrow\ Z{\leq}X,\ Z{\leq}Y.$$

  with the algorithm presented in [3], one has to compute the lgg of the answers to the goal $min(X, Y, Z)$. Since the lgg is mainly syntactical, the user has to guide the computation of the lgg by providing by hand the semantics of the constraints in the answers. With the method presented in this paper, the rule above can be generated just by calling the goals $min(X, Y, Z),\ Z{>}X$ and $min(X, Y, Z),\ Z{>}Y$ to check that their executions fail.

The method to transform some propagation rules into simplification rules presented in [4] is based on a confluence notion. This is a syntactical criterion that works when we have the whole set of rules defining the constraint, and thus it cannot be applied safely if only a part of the propagation rules have been generated. It also requires a termination test for rule-based programs consisting of propagation and simplification rules, and this test is in general undecidable. The new transformation method presented in this paper avoids these two restrictions.

The generation of rule-based constraint solvers is also related to the work done on *Generalized Constraint Propagation* [17], *Constructive Disjunction* [13,23], and *Inductive Logic Programming* [16]. These aspects have been briefly discussed in [3], and it should be pointed out that to our knowledge they still have not been used for the generation of constraint solvers.

Organization of the Paper

In Section 2, we present an algorithm to generate primitive propagation rules by using tabled constraint logic programming. In Section 3, we describe how to modify the algorithm to generate more general propagation rules. Section 4 presents a transformation method of propagation rules into simplification rules. Finally, we conclude with a summary and possibilities of further improvements.

2 Generation of Primitive Propagation Rules

We assume some familiarity with constraint logic programming [14,15]. There are two classes of distinguished constraints, primitive constraints and user-defined constraints. *Primitive constraints* are those constraints defined by a constraint theory CT and for which solvers are already available. *User-defined constraints* are those constraints defined by a constraint logic program P and for which we want to generate solvers. We assume that the set of primitive constraints is closed under negation, in the sense that the negation of each primitive constraint must be also a primitive constraint, e.g. $=$ and $\neq$ or $\leq$ and $>$. In the following, we denote the negation of a primitive constraint c by $not(c)$.

Definition 1. A *constraint logic program* is a set of clauses of the form

$$h \leftarrow b_1, \ldots, b_n, c_1, \ldots, c_m$$

where $h, b_1, \ldots, b_n$ are user-defined constraints and $c_1, \ldots, c_m$ are primitive constraints. A *goal* is a set of primitive and user-defined constraints. An *answer* is a set of primitive constraints. The logical semantics of a constraint logic program P is its Clark's completion and is denoted by P^*.

Definition 2. A *primitive propagation rule* is a rule of the form $C_1 \Rightarrow C_2$ or of the form $C_1 \Rightarrow false$, where C_1 is a set of primitive and user-defined constraints, while C_2 consists only of primitive constraints. C_1 is called the *left hand side* of the rule (*lhs*) and C_2 its *right hand side* (*rhs*). A rule of the form $C_1 \Rightarrow false$ is called *failure rule*.

In the following we use the notation $\exists_{-\mathcal{V}}(\phi)$ to denote the existential closure of ϕ except for the variables in the set $\mathcal{V}$.

Definition 3. A primitive propagation rule $\{d_1, \ldots, d_n\} \Rightarrow \{c_1, \ldots, c_m\}$ is *valid* with respect to the constraint theory CT and the program P if and only if $P^*, CT \models \bigwedge_i d_i \rightarrow \exists_{-\mathcal{V}}(\bigwedge_j c_j)$, where $\mathcal{V}$ is the set of variables appearing in $\{d_1, \ldots, d_n\}$. A failure rule $\{d_1, \ldots, d_n\} \Rightarrow \mathit{false}$ is *valid* with respect to CT and P if and only if $P^*, CT \models \neg\tilde{\exists}(\bigwedge_i d_i)$.

We now give an algorithm to generate such valid rules.

2.1 The PRIM-MINER Algorithm

The PRIM-MINER algorithm takes as input the program P defining the user-defined constraints. To specify the syntactic form of the rules, the algorithm needs also as input two sets of primitive and user-defined constraints denoted by $Base_{lhs}$ and $Cand_{lhs}$, and a set containing only primitive constraints denoted by $Cand_{rhs}$. The constraints occurring in $Base_{lhs}$ are the common part that must appear in the lhs of all rules, $Cand_{lhs}$ indicates candidate constraints used in conjunction with $Base_{lhs}$ to form the lhs, and $Cand_{rhs}$ are the candidate constraints that may appear in the rhs. Note that a syntactic analysis of P can suggest functors and constraint predicates to be used to form candidate constraints. The algorithm PRIM-MINER is presented in Figure 1 and generates a set of valid rules.

The basic idea of the algorithm relies on the following observation: to be able to generate a failure rule of the form $C \Rightarrow \mathit{false}$, we can simply check that the execution of the goal C finitely fails. Furthermore, while these rules are useful to detect inconsistencies, it is in general more interesting to propagate earlier some information that can be used for constraint solving, instead of waiting until a conjunction of constraints becomes inconsistent. Thus, for each possible lhs C (i.e., each subset of $Base_{lhs} \cup Cand_{lhs}$) the algorithm distinguishes two cases:

1. PRIM-MINER uses a CLP system to evaluate the goal C. If the goal finitely fails, then the failure rule $C \Rightarrow \mathit{false}$ is generated.
2. Otherwise the negation of each candidate constraint d from $Cand_{rhs}$ is added in turn to C and the goal $C \cup \{\mathit{not}(d)\}$) is evaluated. If the goal finitely fails, then the rule $C \Rightarrow \{d\}$ is generated.

In practice the evaluation of the goals is made using a bounded depth resolution procedure to avoid non-termination of the whole generation algorithm.

Following [2,5,3], the algorithm PRIM-MINER uses a basic ordering to prune the search space and to avoid the generation of many uninteresting rules. This pruning relies simply on the following observation. If $C_1 \Rightarrow \mathit{false}$ is valid, then rules of the form $C_2 \Rightarrow \mathit{false}$, where $C_1 \subset C_2$ are also valid but useless. So the algorithm considers first the smallest lhs with respect to set inclusion, and when it finds a valid failure rule $C_1 \Rightarrow \mathit{false}$ it discards from the lhs candidates any C_2 that is superset of C_1.

begin

$\mathcal{R}$ the resulting rule set is initialized to the empty set.
L is a list of all subsets of $Cand_{lhs}$,
in an order compatible with the subset partial ordering
(i.e., for all C_1 in L if C_2 is after C_1 in L then $C_2 \not\subset C_1$).

while L is not empty **do**
  Remove from L its first element denoted C_{lhs}.
  if the goal $(Base_{lhs} \cup C_{lhs})$ fails
    with respect to the constraint logic program P **then**
      add the failure rule $(Base_{lhs} \cup C_{lhs} \Rightarrow false)$ to $\mathcal{R}$
      and remove from L all supersets of C_{lhs}.
  else
    for all $d \in Cand_{rhs}$
      if the goal $(Base_{lhs} \cup C_{lhs} \cup \{not(d)\})$ fails
        with respect to the constraint logic program P **then**
          add the rule $(Base_{lhs} \cup C_{lhs} \Rightarrow \{d\})$ to $\mathcal{R}$.
      endif
    endfor
  endif
endwhile

output $\mathcal{R}$.

end

Fig. 1. The PRIM-MINER Algorithm

At first glance, the procedure used to evaluate the goals issued by the algorithm may be considered as a classical depth-first, left-to-right CLP resolution. However, we will show in Section 2.2 and Section 3 that a tabled CLP resolution extends greatly the class of rules that can be generated, by allowing termination of the evaluation in many interesting cases. Additionally, it should be noticed that the execution on the underlying CLP system is not required to enumerate all answers since PRIM-MINER only performs a fail/succeed test, and thus the CLP system can stop after a first answer has been found.

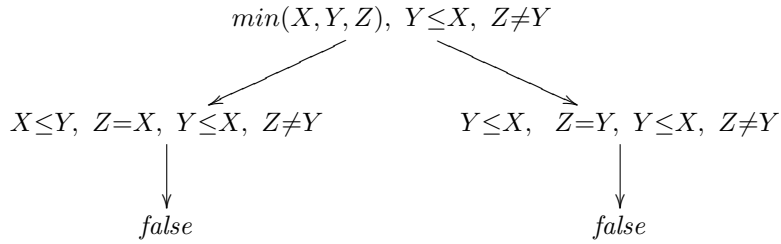
Example 2. Consider the following constraint logic program, where $min(X, Y, Z)$ means that Z is the minimum of X and Y and $\leq, =$ are primitive constraints:

$$min(X, Y, Z) \leftarrow X \leq Y, \quad Z = X.$$
$$min(X, Y, Z) \leftarrow Y \leq X, \quad Z = Y.$$

The algorithm with the appropriate input generates (among others) the rule

$$min(X, Y, Z),\ Y \leq X \Rightarrow Z = Y.$$

after having checked that the execution of the goal $min(X,Y,Z),\ Y \leq X,\ Z \neq Y$ fails by constructing the following derivation tree:



Note that we assume that the constraint solver for $\leq$ and $=$ is able to detect such inconsistencies. □

Soundness and Completeness. The PRIM-MINER algorithm attempts to extract all valid primitive propagation rules of the form $C_1 \Rightarrow \{d\}$ or $C_1 \Rightarrow \mathit{false}$ such that $Base_{lhs} \subseteq C_1$, $C_1 \setminus Base_{lhs} \subseteq Cand_{lhs}$, $d \in Cand_{rhs}$ and there is no other more general failure rule (i.e., no valid rule $C_2 \Rightarrow \mathit{false}$ where $C_2 \subset C_1$). In general, the algorithm cannot be complete, since the evaluation of some goals corresponding to valid rules may be non-terminating. In fact, this completeness can be achieved if more restricted classes of constraint logic programs are used to give the semantics of user-defined constraints and if the solver for the primitive constraints used by the underlying CLP system is satisfaction complete.

The soundness of the algorithm (i.e., only valid rules are generated) is guaranteed by the nice properties of standard CLP schemes [14] and tabled CLP schemes [10]. An important practical aspect, is that even a partial resolution procedure (e.g., bounded depth evaluation) or the use of an incomplete solver by the CLP system, does not compromise the validity of the rules generated.

2.2 Advantage of Tabled Resolution for Rule Generation

Termination of the evaluation of (constraint) logic programs has received a lot of attention. A very powerful and elegant approach based on tabled resolution has been developed, first for logic programming (e.g., [21,22]) and further extended in the context of CLP (e.g., [10,11]).

The intuitive basic principle of tabled resolution is very simple. Each new subgoal S is compared to the previous intermediate subgoals (not necessarily in the same branch of the resolution tree). If there is a previous subgoal I which is equivalent to S or more general than S, then no more unfolding is performed on S and answers for S are selected among the answers of I. This process is repeated for all subsequent computed answers that correspond to the subgoal I. In the case of tabled CLP resolution [10,11], the test to determine if a subgoal I is more general than a subgoal S is performed using an entailment test between the two conjunctions of constraints associated to these subgoals. For example, the subgoal $p(X,Y) \wedge X{<}Y \wedge Y{=}2$ will use answers selected among those of the subgoal $p(X,Y) \wedge X \leq Y$ since $X \leq Y$ is entailed by $X{<}Y \wedge Y{=}2$.

The use of such technique has not been widely accepted since if this leads to termination in many more cases than execution based on SLD-resolution, this should be paid by some execution overhead in general. When using the algorithm PRIM-MINER we can accept a slight decrease of performance (since the solver is constructed once) if this gives rise to an improvement of the termination capability which enables the generation of additional rules. Thus, tabled CLP can find very interesting applications in constraint solver synthesis. This will be illustrated in the following example.

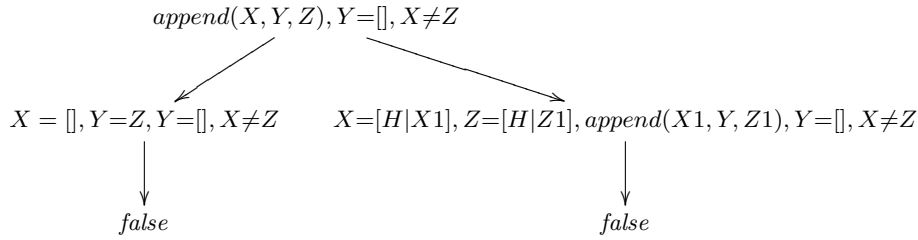
Example 3. Consider the well-known ternary *append* predicate for lists, which holds if its third argument is a concatenation of the first and the second argument.

$$append(X,Y,Z) \leftarrow X{=}[],\ Y{=}Z.$$
$$append(X,Y,Z) \leftarrow X{=}[H|X1],\ Z{=}[H|Z1],\ append(X1,Y,Z1).$$

In the following, we consider that equalities and disequalities over first order terms (e.g., $X{=}[H|X1]$) are defined as primitive constraints. The PRIM-MINER algorithm can generate (among others) the primitive propagation rule

$$append(X,Y,Z),\ Y{=}[]\ \Rightarrow\ X{=}Z$$

by executing the goal $append(X,Y,Z),\ Y{=}[],\ X{\neq}Z$ with a tabled CLP resolution. For a classical CLP scheme the resolution tree will be infinite, while in case of a tabled resolution it can be sketched as follows:



The initial goal $G_1 = (append(X,Y,Z),\ Y{=}[],\ X{\neq}Z)$ is more general than the subgoal $G_2 = (X{=}[H|X1],\ Z{=}[H|Z1],\ append(X1,Y,Z1),\ Y{=}[],\ X{\neq}Z)$, in the sense that $append(A,B,C),\ D{=}[E|A],\ F{=}[E|C],\ B{=}[],\ D{\neq}F$ entails $append(A,B,C),\ B{=}[],\ A{\neq}C$. So no unfolding is made on G_2, and the process waits for answers of G_1 to compute answers of G_2. Since G_1 has no further possibility of having answers, then G_2 fails and thus G_1 also fails. We refer the reader to [10,11] for a more detailed presentation of goal evaluation using a tabled CLP resolution.

Using this kind of resolution PRIM-MINER is also able to produce rules such as:

$$append(X,Y,Z),\ X{=}Z\ \Rightarrow\ Y{=}[].$$
$$append(X,Y,Z),\ Y{\neq}[]\ \Rightarrow\ X{\neq}Z.$$
$$append(X,Y,Z),\ X{\neq}[]\ \Rightarrow\ Z{\neq}[].$$

It should be pointed out that the previous method proposed in [3] to generate propagation rules for user-defined constraints is not able to obtain the four rules given in this example, even when using a tabled resolution. □

We run all examples presented in this paper, using our own implementation of tabled CLP resolution according to the description of [11].

2.3 Selection of Interesting Rules and Performance of Generation

Interesting Rules. If we use the algorithm PRIM-MINER as presented in Figure 1, we obtain a set of rules that is highly redundant and thus not suitable neither for human-reading nor for its direct use as an executable constraint solver. This problem has already been pointed out in [2,5], where techniques have been proposed to address it. The effectiveness of these techniques in a practical application has been shown in [4].

These techniques can be reused in a straightforward way to complete the algorithm PRIM-MINER. The overall principle is to simplify each new rule generated with respect to the set of rules already obtained. For example the rule $a, b \Rightarrow c$ can be removed if we have already generated the rule $a \Rightarrow c$. Such trivial case can be even incorporated in the generation algorithm to avoid the test of the validity of $a, b \Rightarrow c$ itself. However, in many cases redundancy is due to the fact that the application of a rule can be replaced by a sequence of applications of other rules. In practice such redundancy test can be done by checking the equivalence of two sets of propagation rules (one with the rule under consideration and one without) using for example the technique presented in [1].

Performance. The same important pruning of the search space proposed in [2,5] for constraints in $Cand_{lhs} \cap Cand_{rhs}$ can be still applied. This can be sketched as follows. If a rule $C_1 \Rightarrow d$ is generated and d is also in $Cand_{lhs}$ then there is no need to consider any C_2 such that C_2 is a superset of C_1 containing d to form the lhs of another rule. Furthermore an additional optimization specific to the new algorithm can be incorporated in the generation process for the constraints in $Cand_{lhs} \cap Cand_{rhs}$. It follows from the following observation. To obtain the rule $append(X, Y, Z),\ Y{=}[] \ \Rightarrow\ X{=}Z$ the algorithm issues the goal $append(X, Y, Z),\ Y{=}[],\ X{\neq}Z$ (which fails). Then if $X{\neq}Z \in Cand_{lhs}$ and $Y{\neq}[] \in Cand_{rhs}$ the same goal evaluation is used to produce the rule $append(X, Y, Z),\ X{\neq}Z \ \Rightarrow\ Y{\neq}[]$. Thus, the algorithm can be modified to handle in a specific way constraints in $Cand_{lhs} \cap Cand_{rhs}$ to avoid this kind of repeated evaluation of the same goal.

Finally, as in [2] and [3], if PRIM-MINER is implemented on a flexible platform (e.g., SICStus Prolog with CHR [12] support), then when a valid rule is extracted it can be immediately simplified with respect to the previous rules. Next, if it is not redundant it can be incorporated at runtime and thus be used actively to speed up further resolution and constraint solving steps called by the generation algorithm itself.

2.4 Generation of Primitive Splitting Rules

Splitting rules have been shown to be interesting in constraint solving [7], since they can be used to detect early alternative labeling cases or alternative solution sets. Such rules that can be generated by the extension proposed in this section are for example:

$$\begin{aligned} and(X,Y,Z),\ \ Z{=}0 \ &\Rightarrow\ X{=}0 \ \ \vee \ \ Y{=}0. \\ min(X,Y,Z) \ &\Rightarrow\ X{=}Z \ \ \vee \ \ Y{=}Z. \end{aligned}$$

where $and(X,Y,Z)$ means that Z is the Boolean conjunction of X and Y, defined by the facts $and(0,0,0), and(1,0,0), and(0,1,0),\ and(1,1,1)$; and where $min(X,Y,Z)$ is defined by the CLP program of Example 2.

In the following, we restrict ourself to the generation of *primitive splitting rules*.

Definition 4. A *primitive splitting rule* is a rule of the form $C \Rightarrow d_1 \vee d_2$, where d_1, d_2 are primitive constraints and C is a set of primitive and user-defined constraints. $\vee$ is interpreted like the standard disjunction, and primitive splitting rules have a straightforward associated semantics.

It should be noticed that this kind of rules can be directly encoded in existing rule-based languages (e.g., CHR$^\vee$ [6]).

The modification of the basic PRIM-MINER algorithm is as follows. For all C_{lhs}, it must also consider each different set $\{d_1, d_2\} \subseteq C_{rhs}$ with $d_1 \neq d_2$ and check if the goal $(Base_{lhs} \cup C_{lhs} \cup \{not(d_1), not(d_2)\})$ fails with respect to the constraint logic program P. If this is the case, it simply adds to $\mathcal{R}$ the primitive splitting rule $(Base_{lhs} \cup C_{lhs} \Rightarrow d_1 \vee d_2)$.

For example, the rule $min(X,Y,Z) \Rightarrow X{=}Z \ \vee \ Y{=}Z$ can be obtained by issuing the goal $min(X,Y,Z),\ X{\neq}Z,\ Y{\neq}Z$ and checking that its execution finitely fails.

Here again, the soundness of the generation relies on the properties of the underlying resolution used.

In practice, a huge number of splitting rules are not interesting because they are redundant with respect to some primitive propagation rules. So, to remove these uninteresting rules, the generation should preferably be done, by first using PRIM-MINER to obtain only primitive propagation rules, and then using the modified version of the algorithm to extract primitive spitting rules. Thus in this second step, redundant rules, with respect to the set resulting from the first step, can be discarded on-the-fly. For example the rule $a, b, c \ \Rightarrow \ d \vee e$ will be removed if we have already generated the rule $a, b \ \Rightarrow \ d$. Moreover, in such trivial redundancy cases, the test of the validity of the rule $a, b, c \ \Rightarrow \ d \vee e$ can itself be avoided, and more generally the test of any rule of the form $C \Rightarrow d \vee f$ and $C \Rightarrow f \vee d$, where $\{a, b\} \subseteq C$ and $f \in Cand_{rhs}$.

3 Generation of More General Propagation Rules

In this section, we modify the algorithm presented in Section 2 to handle a broader class of rules called *general propagation rules*, that encompasses the primitive propagation rules, and that can even represent recursive rules over user-defined constraints (i.e., rules where the same user-defined constraint predicate appears in the lhs and rhs).

Definition 5. A *general propagation rule* is a failure rule or a rule of the form $C_1 \Rightarrow C_2$, where C_1 and C_2 are sets of primitive and user-defined constraints.

The notion of validity defined for primitive propagation rules also applies to this kind of rules.

In the PRIM-MINER algorithm, the test of the validity of a propagation rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is performed by checking that the goal $Base_{lhs} \cup C_{lhs} \cup \{not(d)\}$ fails. For general propagation rules, d is no longer a primitive constraint but may be defined by a CLP program. In this case, the evaluation should be done using a more general resolution procedure to handle negated subgoals. However, to avoid the well known problems related to the presence of negation we can simply use a different validity test based on the following theorem.

Theorem 1. *Let $C_1 \Rightarrow C_2$ be a general propagation rule and $\mathcal{V}$ be the variables occurring in C_1. Let S_1 be the set of answers $\{a_1, \ldots, a_n\}$ to the goal C_1, and S_2 be the set of answers $\{b_1, \ldots, b_m\}$ to the goal $C_1 \cup C_2$. Then the rule $C_1 \Rightarrow C_2$ is valid if $P^*, CT \models \neg(\exists_{-\mathcal{V}}((a_1 \vee \ldots \vee a_n) \wedge \neg\exists_{-\mathcal{V}}(b_1 \vee \ldots \vee b_m)))$.*

This straightforward property comes from the completeness result of standard CLP schemes [14] which ensures that if a goal G has a finite computation tree, with answers $c_1, \ldots, c_n$ then $P^*, CT \models G \leftrightarrow \exists_{-\mathcal{V}_g}(c_1 \vee \ldots \vee c_n)$, where $\mathcal{V}_g$ is the set of variables appearing in G.

So, the modification proposed in this section consists simply in the replacement in algorithm PRIM-MINER of the call to the goal $Base_{lhs} \cup C_{lhs} \cup \{not(d)\}$ to check the validity of the rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$, by the following steps.

- First, collect the set of answers $\{a_1, \ldots, a_n\}$ to the goal $Base_{lhs} \cup C_{lhs}$.
- Then, collect the set of answers $\{b_1, \ldots, b_m\}$ to the goal $Base_{lhs} \cup C_{lhs} \cup \{d\}$.
- In each answer a_i (resp. b_i), rename with a fresh variable any variable that is not in $Base_{lhs} \cup C_{lhs}$ (resp. $Base_{lhs} \cup C_{lhs} \cup \{d\}$).
- Finally, perform a satisfiability test of $(a_1 \vee \ldots \vee a_n) \wedge \neg(b_1 \vee \ldots \vee b_m)$.
- If this test fails then the rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is valid.

Since answers only contain primitive constraints and since the set of primitive constraints is closed under negation, then we can perform the satisfiability test by rewriting $(a_1 \vee \ldots \vee a_n) \wedge \neg(b_1 \vee \ldots \vee b_m)$ into an equivalent disjunctive normal form, and then use the solver for primitive constraints on each sub-conjunctions.

It should be noticed that in cases where the evaluation of one of the two goals does not terminate before the bound of resolution depth is reached then $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is not considered as valid and the next rule is processed.

Example 4. Consider the following Boolean constraints: $neg(X, Y)$ imposing that Y is the Boolean complement of X and $xor(X, Y, Z)$ stating that Z is the result of the exclusive Boolean disjunction of X and Y. The modified algorithm presented above can generate the following rules:

$$xor(X, Y, Z),\ Z{=}1 \Rightarrow neg(X, Y).$$
$$xor(X, Y, Z),\ Y{=}1 \Rightarrow neg(X, Z).$$
$$xor(X, Y, Z),\ X{=}1 \Rightarrow neg(Y, Z).$$

For example, to test the validity of the first rule, the process is the following. First, the answers $A_1 := X{=}1 \wedge Y{=}0 \wedge Z{=}1$ and $A_2 := X{=}0 \wedge Y{=}1 \wedge Z{=}1$ to the goal $xor(X, Y, Z),\ Z{=}1$ are computed, then the same answers are collected for the goal $xor(X, Y, Z),\ Z{=}1,\ neg(X, Y)$ and finally the satisfiability test of $(A_1 \vee A_2) \wedge \neg(A_1 \vee A_2)$ fails and thus establishes the validity of the rule.

One other general rule that can be generated using the modified algorithm presented here and that cannot be obtained by previous approaches is for example:

$$and(X, Y, Z) \ \Rightarrow\ min(X, Y, Z).$$

Furthermore, rules representing symmetries can be automatically detected using the modified algorithm. For example, the rules

$$min(X, Y, Z) \ \Rightarrow\ min(Y, X, Z).$$
$$xor(X, Y, Z) \ \Rightarrow\ xor(Y, X, Z).$$

expressing the symmetry of the *min* and the *xor* constraints with respect to the first and second arguments can be generated. In [3], it has been shown that these rules are very useful to reduce the size of a set of propagation rules since many rules become redundant when we know such symmetries. In [3], these rules cannot be generated automatically, but have to be provided by the user. □

However, it must be pointed out that if the test presented in this section allows to handle a syntactically wider class of rules, it relies on different goal calls than the test of Section 2.1. So when testing the validity of a primitive propagation rule one of the techniques may lead to terminating evaluation while the other one may not. Thus in the case of primitive propagation rule it is preferable not to replace one test by the other, but to use both in a complementary way (run one of them, and if it reaches the bound of resolution depth then apply the other).

4 Generation of Simplification Rules

Since a propagation rule does not rewrite constraints but adds new ones, the constraint store may contain superfluous information. Constraints can be removed from the constraint store using *simplification rules.*

begin

$P' := P$
for each propagation rule of the form $C \Rightarrow D$ in P **do**
Find a proper subset E of C such that $Base_{lhs} \not\subseteq E$ and
$D \cup E \Rightarrow C$ is valid (using the validity test of Section 3)
If E exists **then**
$P' := (P' \backslash \{C \Rightarrow D\}) \cup \{C \Leftrightarrow D \cup E\}$
endif
endfor

output P'
end

Fig. 2. The Transformation Algorithm

Definition 6. A *simplification rule* is a rule of the form $C_1 \Leftrightarrow C_2$, where C_1 and C_2 are sets of primitive and user-defined constraints.

In [4], a method has been proposed to transform some propagation rules into simplification rules. It has been shown that this transformation is crucial to improve both the time and space behavior of constraint solving. The method is a kind of post-processing approach based on a confluence test which requires the termination of the set of rules. Termination is in general undecidable, this problem may jeopardize the practicability of this method. Furthermore, the confluence criterion works when we have the whole set of rules defining the constraint, and thus it cannot be applied safely if only a part of the propagation rules have been generated.

In this section, we show how the method presented in Section 3 can be used to transform some propagation rules into simplification rules avoiding these two restrictions. For a valid propagation rule of the form $C \Rightarrow D$, we try to find a proper subset E of C such that $D \cup E \Rightarrow C$ is valid too. If such E can be found, the propagation rule $C \Rightarrow D$ can be transformed into a simplification rule of the form $C \Leftrightarrow D \cup E$.

To simplify the presentation, we present the algorithm to transform (when possible) propagation rules into simplification rules independently from the algorithm presented in Section 3. Note that the algorithm for the generation of propagation rules can be slightly modified to incorporate this step and to directly generate simplification rules.

The algorithm is given in Figure 2 and takes as input the set of generated propagation rules and the common part that must appear in the lhs of all rules, i.e. $Base_{lhs}$.

To achieve a form of minimality based on the number of constraints, we generate simplification rules that will remove the greatest number of constraints.

So, when we try to transform a propagation rule into a simplification rule of the form $C \Leftrightarrow D \cup E$ we choose the smallest set E (with respect to the number of atomic constraints in E) for which the condition holds. If such a E is not unique, we choose any one among the smallest. The condition $Base_{lhs} \not\subseteq E$ is needed to be able to transform the propagation rules into simplification rules that rewrite constraints to *simpler* ones (primitive constraints if possible), as shown in the following example.

Example 5. Consider the following propagation rule R generated for the *append* constraint with $Base_{lhs} = \{append(X, Y, Z)\}$:

$$append(X, Y, Z),\ X{=}[]\ \Rightarrow\ Y{=}Z.$$

The algorithm tries the following transformations:

1. First, it checks if rule R can be transformed into the simplification rule $append(X, Y, Z),\ X{=}[] \Leftrightarrow Y{=}Z$. This is done by testing whether the rule $Y{=}Z \Rightarrow append(X, Y, Z),\ X{=}[]$ is valid. But, this is not the case and thus the transformation is not possible.
2. Next, the algorithm finds out that the rule $Y{=}Z,\ X{=}[] \Rightarrow append(X, Y, Z)$ is a valid rule and then the propagation rule R is transformed into the simplification rule $append(X, Y, Z),\ X{=}[] \Leftrightarrow X{=}[],\ Y{=}Z$.

Note that the propagation rule $Y{=}Z,\ append(X, Y, Z) \Rightarrow X{=}[]$ is also valid but transforming rule R into

$$append(X, Y, Z),\ X{=}[]\ \Leftrightarrow\ append(X, Y, Z),\ Y{=}Z.$$

will lead to a simplification rule which is uninteresting for constraint solving, i.e. using this rule the *append* constraint cannot be simplified and remains in the constraint store. The algorithm disables such transformation by checking the condition that all constraints of $Base_{lhs}$ cannot be shifted to the right hand side of the rule ($Base_{lhs} \not\subseteq E$). □

To test the validity of a rule, the transformation algorithm uses the techniques presented in Section 2 and Section 3 according to the form of the rule (i.e., primitive or general propagation rule). As for PRIM-MINER, the transformation algorithm can take advantage of tabled resolution for constraint logic programming to obtain terminating validity tests in more cases than when using standard CLP resolution schemes.

5 Conclusion and Future Work

In this paper, we have demonstrated how the power and the versatility of tabled constraint logic programming can be used for generating rule-based constraint solvers. We have presented three algorithms that can be integrated to build an environment for the synthesis of solvers. We have also shown that compared to

the algorithms described in [8,19,2,5,3] our approach is able to generate more expressive rules (including recursive and splitting rules).

One interesting direction for future work is to investigate the integration of constructive negation (e.g., [18,20]) in tabled resolution for CLP to generate constraint solvers, in order to check the validity of the propagation and simplification rules in more general cases. Another complementary aspect is the completeness of the solvers generated. It is clear that in general this property cannot be guaranteed, but in some cases it should be possible to check it, or at least to characterize the kind of consistency the solver can ensure.

References

1. S. Abdennadher and T. Frühwirth. Operational equivalence of CHR programs and constraints. In *5th International Conference on Principles and Practice of Constraint Programming, CP'99*, LNCS 1713. Springer-Verlag, 1999.
2. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP00*, LNCS 1894. Springer-Verlag, 2000.
3. S. Abdennadher and C. Rigotti. Towards inductive constraint solving. In *7th International Conference on Principles and Practice of Constraint Programming, CP'2001*, LNCS. Springer-Verlag, 2001.
4. S. Abdennadher and C. Rigotti. Using confluence to generate rule-based constraint solvers. In *Third International Conference on Principles and Practice of Declarative Programming*. ACM Press, 2001.
5. S. Abdennadher and C. Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *ACM Transactions on Computational Logic*, 2004. to appear.
6. S. Abdennadher and H. Schütz. CHR$^\vee$: A flexible query language. *Flexible Query Answering Systems*, LNAI 1495, 1998.
7. K. Apt. A proof theoretic view of constraint programming. *Special Issue of Fundamenta Informaticae*, 34(3), 1998.
8. K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *5th International Conference on Principles and Practice of Constraint Programming, CP'99*, LNCS 1713. Springer-Verlag, 1999.
9. A. Biermann. Automatic programming. *Encyclopedia of Artificial Intelligence, second, extended edition,*, 1992.
10. P. Codognet. A tabulation method for constraint logic programs. In *8th Symposium and Exhibition on Industrial Applications of Prolog*, 1995.
11. B. Cui and D. S. Warren. A system for tabled constraint logic programming. In *First International Conference on Computational Logic*, LNCS 1861. Springer-Verlag, 2000.
12. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
13. P. V. Hentenryck, V. Saraswat, and Y. Deville. Desing, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.
14. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.

15. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
16. S. Muggleton and L. De Raedt. Inductive Logic Programming : theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
17. T. L. Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–359, 1993.
18. T. Przymusinski. On constructive negation in logic programming. In *Proceedings of the North American Conference on Logic Programming*, 1990.
19. C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains via unification in finite algebra. In *New Trends in Constraints*. LNAI 1865, 2000.
20. P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.
21. H. Tamaki and T. Sato. OLD resolution with tabulation. In *3rd International Conference on Logic Programming*, LNCS 225. Springer-Verlag, 1986.
22. D. S. Warren. Memoing for logic programs. *Communications of ACM*, 35(4):93–11, 1992.
23. J. Würtz and T. Müller. Constructive disjunction revisited. In *Proc. of the 20th German Annual Conference on Artificial Intelligence*, LNAI 1137. Springer-Verlag, 1996.

Translating Datalog-Like Optimization Queries into ILOG Programs*

G. Greco, S. Greco, I. Trubitsyna, and E. Zumpano

DEIS, Univ. della Calabria, 87030 Rende, Italy
{ggreco,greco,irina,zumpano}@si.deis.unical.it

This paper presents a logic language, called $\mathcal{NP}\,Datalog$ suitable for expressing NP search and optimization problems. The 'search' language extends stratified Datalog with constraints and allows disjunction to define nondeterministically partitions of relations. It's well known that $\mathcal{NP}$ search problems can be formulated as unstratified DATALOG queries under nondeterministic stable model semantics so that each stable model corresponds to a possible solution. $\mathcal{NP}$ optimization problems are then formulated by adding a *max* (or *min*) construct to select the stable model (thus, the solution) which maximizes (resp., minimizes) the result of a polynomial function applied to the answer relation. The problem in using DATALOG$^{\neg}$ to express search and optimization problems is that the unrestricted negation in programs is often neither simple nor intuitive and, besides, it does not allow to discipline the expressive power. Thus, we consider restricted forms of negation which force user to write programs in a more disciplined way without loosing of expressive power. More specifically, we consider the language $\mathcal{NP}\,Datalog$ which extends DATALOG$^{\neg_s}$ with two simple forms of unstratified negation embedded into built-in constructs: head disjunction and constraints. Thus the core of our language is stratified Datalog extended with two constructs allowing nondeterministic selections and with query goals enforcing conditions to be satisfied by stable models.

As an example of the language, take the *Vertex Cover* problem:

Example 1. Given a graph $G = \langle N, E\rangle$, a subset of the vertex $V \subseteq N$ is a *vertex cover* of G if for each edge $(x, y) \in E$ either x or y is in V. The goal $\mathtt{min}|\mathtt{v}(\mathtt{X})|$ restricting the set of suitable stable models to those for which the subset of nodes v is minimum can be defined as follows:

$$\mathtt{v}(\mathtt{X},\mathtt{1}) \oplus \mathtt{v}(\mathtt{X},\mathtt{0}) \leftarrow \mathtt{node}(\mathtt{X})$$
$$\leftarrow \mathtt{edge}(\mathtt{X},\mathtt{Y}), \neg\mathtt{v}(\mathtt{X},\mathtt{1}), \neg\mathtt{v}(\mathtt{Y},\mathtt{1})$$

where the predicates *node* and *edge* define the vertices and the edges of the graph by means of a suitable number of facts. The first rule defines a partition of the relation node, i.e. $\oplus$ denotes exclusive disjunction; the second rule defines a constraints, i.e. it enforces that $\leftarrow$ is used to define a constraint rule which is satisfied only of the body is false. □

* Work partially supported by the Italian National Research Council (CNR) and by MURST (projects "Sistemi informatici integrati a supporto del bench-marking di progetti ed interventi ad innovazione tecnologica in campo agro-alimentare" and D2I).

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 48–49, 2003.

The $\mathcal{NP}$ *Datalog* program contains 1) *guess predicates*, defined by disjunctive rules 2) *view predicates*, which are deterministic predicates defined by standard rules not depending (transitively) on guess predicates; and 3) *derived predicates* defined by standard rules depending on guess predicates. Note that view predicates can be computed once (in terms of relational algebra for not recursive rules or by means of fixpoint algorithms for recursive rules), whereas derived predicates depend on the constraints and on the guess performed by partition rules; thus they are computed after the guess has been made.

The implementation of the language $\mathcal{NP}$ *Datalog* is carried out by means of a module which computes view predicates and translates the remaining part of the $\mathcal{NP}$ *Datalog* query into an ILOG program. ILOG programs are then computed by means of the ILOG OPL-studio system [3]. In particular queries can be evaluated by computing the $\texttt{DATALOG}^{\neg_s}$ component and translating the remaining part into an ILOG program. We are currently implementing a system prototype which executes $\mathcal{NP}$ *Datalog* queries by translating them into ILOG programs. In the following we informally show how $\mathcal{NP}$ *Datalog* queries are translated into an ILOG programs. The translation of the set of rules, of an $\mathcal{NP}$ *Datalog* program P, defining guess and derived predicates is performed by translating every standard rule defining a derived predicate into an ILOG statement stating how the predicate is computed. Constraint rules are similarly translated to rules defining derived predicates into ILOG constraints. Disjunctive rules are not explicitly translated as they are used to guess a partition of a relation. As regards to the query translation a search (resp. optimization) query $\langle P, v(X,t)\rangle$ (resp. $\langle P,\ min|v(X,t)|\rangle$) is translated into an ILOG fragment solving (minimizing the cardinality of $v(X,t)$ of) the problem defined by the translation of P.

Example 2. Min vertex cover. In order to show how the previous transformation rules work, we reconsider the *Min vertex Cover* problem modelled by means of an $\mathcal{NP}$ *Datalog* query in the Example 1. The corresponding ILOG program consists in the following instructions:

```
var bool v[d_v];
minimize
        sum (x in d_v) v[x]
subject to {
        forall (⟨x,y⟩ in d_r2)
                1 ⇒ (v[x] = 1 ∨ v[y] = 1); };
```

References

1. Greco, S., and Saccà, D., NP-Optimization Problems in Datalog. *Proc. Int. Logic Programming Symp.*, 1997, 181-195.
2. Kolaitis, P. G., and Thakur, M. N., Logical Definability of NP Optimization Problems. *Information and Computation*, No. 115, 1994, 321-353.
3. Van Hentenryck, P., *The OPL Optimization Programming Language* Mit Press, 1999.

Tabling Structures for Bottom-Up Logic Programming

Roger Clayton, John G. Cleary, Bernhard Pfahringer, and Mark Utting

Department of Computer Science, University of Waikato
Hamilton, New Zealand
{rjc4,jcleary,bernhard,marku}@cs.waikato.ac.nz

Generally logic programs are considered more abstract than programs written in other paradigms. Yet most logic programs are strongly dependent on programmer-defined data structures – such as lists in Prolog. Logic programs often become saturated with data structure manipulation code which obscures the original problem and requires large-scale rewriting to change data representations. To avoid this, the logic programming language Starlog provides a data-structure-free environment where data is stored relationally in tuples and programs are evaluated bottom-up [2].

To remain efficient the underlying data structures that hold relations must be carefully constructed with minimal programmer intervention. We store tuples in a global database that can be defined and optimised independent of a program's source. Indexing of tuples uses both their predicate names and arguments. The indexing order is specified by the programmer at compile time[1]. Arguments from different predicates with comparable arguments can be indexed together. The resulting database schema is static, thus minimizing runtime overheads.

Bottom-up evaluation of programs can be represented as a series of database operations (where the body of a rule is a database query and the head is an insert). Using this model we have developed the Starlog Data Structure Language (SDSL) to which Starlog programs can be translated. SDSL supports operations such as known-value lookups, unknown-value lookups and inserts on the global database. With programs represented in SDSL, code optimisations can remove repeated operations and can combine identical sections of code.

Using an SDSL version of a program further customisation of the global database is possible. Each argument index in the global database stores a set of values and therefore is implemented by a data structure. We have developed a library of data structures (that share a common interface) to implement the indexes in the database. Here the programmer can choose which data structures to use. However, to make an intelligent decision they must understand how their program uses data at runtime. Alternatively a generic data structure can be used but this will often be less efficient than choosing a single more specialized data structure.

Letting the compiler select the data structure itself can avoid such problems. In the paper we examine three different ways of doing this: Dynamic Estimation (DE); Static Estimation (SE); and Regression Estimation (RE).

[1] Work in progress indicates that efficient index orderings can be automatically inferred from mode information to minimise backtracking during searches.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 50–51, 2003.

From analysis of the SDSL code (together with other estimates of the expected data set) the compiler tries to find the most efficient data structure for each index. For example, when no known-value searches are performed on an index or when range information indicates that there are very few elements, an unsorted list would be the optimal choice. Alternatively, when searching for known values is frequent and many elements are expected then a hash-table is more efficient.

SE uses an explicit model of the variation of the execution time for each data structure. The model takes account of the number of entries in each data structure as well as how it is accessed. These values are inferred from the source code or by the programmer providing hints (for example that a large number of items will be stored). DE improves on this by collecting data on the data structure usage during a run of the program. In principle these more refined parameters should provide more accurate estimates of the performance of the different data structures.

Our preliminary results were taken from one example program whose database contains four argument indexes (for more detail about this and our results see technical report [1]). Neither the SE nor the DE estimates predict the absolute runtime or the relative ranking of the data structures accurately. In spite of this, the actual performance of the data structure combination with the lowest SE was ranked within the top 25% of all combinations. The data structure combination with the lowest DE was within the top 20%. The inaccuracy of both the DE and SE estimates appears to result from the difficulty of capturing data structure performance with a small number of parameters. For example, both the automatic garbage collection of Java and hardware caching can have a major effect on performance and are almost impossible to capture in a simple model.

The RE technique records the program's runtimes from multiple runs, using different combinations of data structures. RE then applies regression analysis assuming that the total execution time is the sum of the times used by each data structure. Only a fraction of all possible data structure combinations need be measured to infer the best combination. RE with a small numbers of combinations was checked against a complete run of all combinations. Timing only 4% of all combinations ensured the best estimate is in the top 1.3% of all runtimes.

Based on these preliminary results we suggest using SE to select the data structures during the early development and testing cycle. Its major advantage is that it seems to be able to avoid very bad selections. RE is indicated to optimise performance as part of final deployment. DE does not seem to be useful as it seldom improves significantly on the static analysis.

References

1. Roger Clayton, John G. Cleary, Bernhard Pfahringer, and Mark Utting. Optimising tabling structures for bottom-up logic programming. Technical Report, University of Waikato, Sept 2002.
2. John G. Cleary, Mark Utting, and Roger J. Clayton. Data structures considered harmful. In *Australasian Workshop on Computational Logic*, Feb 2000.

A General Framework for Variable Aliasing: Towards Optimal Operators for Sharing Properties

Gianluca Amato[1] and Francesca Scozzari[2]

[1] Dipartimento di Scienze, Università di Chieti-Pescara
amato@sci.unich.it
[2] Dipartimento di Informatica, Università di Pisa
scozzari@di.unipi.it

Abstract. We face the problem of devising optimal unification operators for sharing and linearity analysis of logic programs by abstract interpretation. We propose a new (infinite) domain $\mathtt{ShLin}^\omega$ which can be thought of as a general framework from which other domains can be easily derived by abstraction. The advantage is that $\mathtt{ShLin}^\omega$ is endowed with very elegant and optimal abstract operators for unification and matching, based on a new concept of *sharing graph* which plays the same role of alternating paths for pair sharing analysis. We also provide an alternative, purely algebraic description of sharing graphs. Starting from the results for $\mathtt{ShLin}^\omega$, we derive optimal abstract operators for two well-known domains which combine sharing and linearity: $\mathtt{ShLin}^2$ by Andy King and the classic $\mathtt{Sharing} \times \mathtt{Lin}$.

1 Introduction

In the field of static analysis of logic programs by abstract interpretation [7,8], the property of sharing has been the object of many works, both on the theoretical and practical point of view. The goal of (set) sharing analysis is to detect sets of variables which share a common variable in the answer substitutions. Typical applications of sharing analysis are in the fields of optimization of unification [21] and parallelization of logic programs [10].

It is now widely recognized that the original domain proposed for sharing analysis, namely `Sharing` [18,13] by Jacobs and Langen, is not very precise, so that it is often combined with other domains for treating freeness, linearity, groundness or structural information (see [3] for a comparative evaluation). In particular, adding some kind of linearity information seems to be very profitable, both for the gain in precision and speed which can be obtained, and for the fact that it can be easily and elegantly embedded inside the sharing groups (see [14]). However, optimal operators for combined analysis of sharing and linearity have never been devised, neither for the domain $\mathtt{ShLin}^2$ [14], nor for the more broadly adopted $\mathtt{Sharing} \times \mathtt{Lin}$ [9,20] or `ASub` [4]. The lack of optimal operators brings two kinds of disadvantages: first, the analysis obviously looses in precision when

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 52–70, 2003.

using sub-optimal abstract operators; second, computing approximated abstract objects can lead to a speed-down of the analysis. The latter is typical of sharing analysis, where abstract domains are usually defined in such a way that, the less information we have, the more abstract objects are complex. This is not the case for other kind of analyses, such as groundness analysis, where the complexity of abstract objects may grow accordingly to the amount of groundness information they encode. The role played by linearity in the unification process has never been fully clarified. The traditional domains which combine sharing and linearity information are too abstract to capture in a clean way the effect of repeated occurrences of a variable in a term and most of the effects of (non-)linearity are obscured by the abstraction process. In this paper, we investigate the interaction between sharing and linearity, and provide optimal abstract operators for two well-known domains which combine these properties. We start by introducing our concrete goal-dependent framework which is based on [6], with the improvements introduced in [1] concerning backward unification. We propose a slightly modified domain of substitutions, which are quotiented modulo an appropriate renaming w.r.t. the variables which are not of interest (see Jacobs and Langen' domain *ESubst* [13]). We define two operations of unification and matching which are used as an intermediate step toward the semantics functions for forward and backward unifications.

Inspired by $\mathtt{ShLin}^2$, we propose an abstract domain which is able to encode the *amount* of non-linearity, i.e., which keeps track of the exact number of occurrences of the same variable in a term. The domain we obtain is very simple and elegant, but cannot be directly used for static analysis, at least without resorting to widening operators, since it contains infinite ascending chains. However, in this domain the role played by (non-)linearity is manifest, and the optimal abstract operators for unification and matching [9,16,1] assume a very clean form. The cornerstone of the abstract unification is the concept of *sharing graph* which plays the same role of alternating paths [21,15] for pair sharing. A sharing graph is a graph theoretic notion to figure out sharing groups which are combined during the unification process to obtain a new sharing group. The use of sharing graphs offers a new perspective to look at single variables in the process of unification, and simplify the proofs of correctness and optimality of the abstract operators. We also provide a purely algebraic characterization of the results, which can help in implementing the domain by making use of widening operators and in devising abstract operators for further abstractions of $\mathtt{ShLin}^\omega$. We show that the domains $\mathtt{ShLin}^2$ and $\mathtt{Sharing} \times \mathtt{Lin}$ can be immediately obtained as abstractions of $\mathtt{ShLin}^\omega$ and we provide the optimal operators for unification and matching. We also provide a simplified version of the operators for the domain $\mathtt{Sharing} \times \mathtt{Lin}$ which is still correct, but which is optimal for one-binding substitutions only. We show that unification between an abstract object and a substitution cannot be computed one binding at a time while remaining optimal. Finally, we conclude with some open questions for future work.

2 Notation

Let $\mathbb{N}^+$ be the set of natural numbers without zero. A *(finite) multiset* is a map $X : \mathcal{X} \to \mathbb{N}^+$ where $\mathcal{X}$ is a finite set called the *support* of X and denoted by $\lfloor\!\lfloor X \rfloor\!\rfloor$. We often denote a multiset as $\{\!\{v_1, \ldots, v_n\}\!\}$ where $v_1, \ldots, v_n$ is a sequence of elements with repetitions. We also use the polynomial notation $X = v_1^{i_1}, \ldots, v_n^{i_n}$ to denote a multiset with support $\{v_1, \ldots, v_n\}$ such that $X(v_k) = i_k$. We extend the functional notation for multiset by writing $X(v) = 0$ if $v \notin \lfloor\!\lfloor X \rfloor\!\rfloor$.

If S is a set, a *multiset over* S is a multiset whose support is a subset of S. We denote by $\wp_m(S)$ the set of multisets over S and we write $X \subseteq_m S$ as an alternative for $X \in \wp_m(S)$. Any set S used as an argument to a multiset operator stands for the multiset with support S and such that $S(x) = 1$ for all $x \in S$. We denote by $X|_S$ the multiset defined as $X(v)$ if $v \in S$, 0 otherwise. $|X|$ will denote the number of elements in X including repeated elements, i.e., $\sum_{x \in \lfloor\!\lfloor X \rfloor\!\rfloor} X(x)$ when $X \neq \emptyset$, 0 otherwise. If E is any expression involving a variable x, we write $\sum_{x \in X} E(x)$ as a short form for $\sum_{x \in \lfloor\!\lfloor X \rfloor\!\rfloor} X(x) \cdot E(x)$. We use either $\{\!\{\}\!\}$ or $\emptyset$ for the multiset whose support is empty, while union and intersection of multisets are denoted by $\uplus$ and $\Cap$. If $X = \{\!\{X_1, \ldots, X_m\}\!\}$ is a multiset of multisets, then $\uplus X = X_1 \uplus \cdots \uplus X_m$. We also use the notation $\uplus_{i \in \{1,\ldots,m\}} X_i$ with the same meaning.

Let Atoms, Clauses, Body and Progs be the syntactic categories for atoms, clauses, bodies and programs respectively, where $\lambda \in$ Body stands for the empty body. We denote by *Subst* and *ISubst* the sets of substitutions and idempotent substitutions respectively, by ϵ the empty substitution and by *Ren* the set of renamings (i.e., invertible substitutions). Given a substitution θ, $\mathrm{dom}(\theta)$ and $\mathrm{rng}(\theta)$ denote the domain and range of θ. Let $\mathcal{V}$ be a denumerable set of variables. Given $v \in \mathcal{V}$ and a term t, we denote by $vars(t)$ the set of variables occurring in t and by $occ(v, t)$ the number of occurrences of v in t.

3 The Concrete Semantics

Our analysis is based on a (collecting) goal-dependent semantics for logic programs. We look for a concrete domain where substitutions explicitly show their variables of interest, which are "independent" from the other variables. This choice is motivated from the fact that, in practice, one needs to keep track of the current variables of interest during the analysis. Several semantics in the literature present these characteristics, e.g. the semantics in [19] based on exequations, [6] using idempotent substitutions and that in [13] based on the domain *ESubst* of existential substitutions. The latter semantics is probably the most appropriate to our goal, but it is based on a non-standard definition of substitution and the relation between the unification operator on *ESubst* and the standard one, although clear, is not well stated. Moreover, most of the research in combining sharing and linearity information has been done on the domain defined in [6], so that founding our work on a different framework would make difficult the comparison of our results to the literature. Therefore, we work with

[6] and show that with an appropriate equivalence relation on substitutions, one can obtain an equivalent semantics over existential substitutions.

3.1 Concrete Domain and Operators

The concrete domain is $\texttt{Rsub} = (\wp(\mathit{ISubst}) \times \wp_f(\mathcal{V})) \cup \{\bot_{\mathrm{Rs}}, \top_{\mathrm{Rs}}\}$ (see [6] for a detailed introduction). Rsub is partially ordered as follows: $\bot_{\mathrm{Rs}}$ is the bottom element, $\top_{\mathrm{Rs}}$ the top and $[\Theta_1, U_1] \leq_{\mathrm{Rs}} [\Theta_2, U_2]$ if and only if $U_1 = U_2$ and $\Theta_1 \subseteq \Theta_2$. Rsub is a complete lattice w.r.t. $\leq_{\mathrm{Rs}}$. The least upper bound of Rsub is denoted by $\sqcup_{\mathrm{Rs}}$.

We briefly recall the concrete operations proposed in [6] and refined in [1] with the introduction of two different operators for forward and backward unification. The concrete projection $\pi_{\mathrm{Rs}} : \texttt{Rsub} \times \texttt{Rsub} \to \texttt{Rsub}$ is defined as:

$$\begin{aligned}
&\pi_{\mathrm{Rs}}(\bot_{\mathrm{Rs}}, A) = \pi_{\mathrm{Rs}}(A, \bot_{\mathrm{Rs}}) = \bot_{\mathrm{Rs}} \\
&\pi_{\mathrm{Rs}}(\top_{\mathrm{Rs}}, A) = \pi_{\mathrm{Rs}}(A, \top_{\mathrm{Rs}}) = \top_{\mathrm{Rs}} \qquad\qquad \text{when } A \neq \bot_{\mathrm{Rs}} \\
&\pi_{\mathrm{Rs}}([\Theta_1, U_1], [\Theta_2, U_2]) = [\Theta_1, U_1 \cap U_2]
\end{aligned}$$

In the following, for the sake of conciseness, we define the behavior of the concrete and abstract operators only in case all the arguments are different from $\bot_{\mathrm{Rs}}$ and $\top_{\mathrm{Rs}}$. We implicitly assume that the result is $\bot_{\mathrm{Rs}}$ if any of the argument is $\bot_{\mathrm{Rs}}$, $\top_{\mathrm{Rs}}$ when any of the argument is $\top_{\mathrm{Rs}}$ and no argument is $\bot_{\mathrm{Rs}}$. The concrete forward unification is $\mathbf{U}^f_{\mathrm{Rs}} : \texttt{Rsub} \times \wp_f(\mathcal{V}) \times \mathsf{Atoms} \times \mathsf{Atoms} \to \texttt{Rsub}$ such that:

$$\begin{aligned}
\mathbf{U}^f_{\mathrm{Rs}}([\Theta, U_1], U_2, A_1, A_2) = [\{\mathrm{mgu}(\rho_1(\theta), \delta) \mid \theta \in \Theta, \\
\delta = \mathrm{mgu}(\rho_1(A_1) = A_2)\}, \rho_1(U_1) \cup U_2]
\end{aligned}$$

where $(\rho_1, \rho_2) = \mathit{Apart}(U_2)$, provided $\mathit{vars}(A_1) \subseteq U_1$ and $\mathit{vars}(A_2) \subseteq U_2$, $\bot_{\mathrm{Rs}}$ in all the other cases. We still need to define *Apart*. Given $U_2 \in \wp_f(\mathcal{V})$, take a partition $\{V_1, V_2\}$ of $\mathcal{V}$ such that V_1 and V_2 are infinite and $U_2 \subseteq V_2$. Then $\mathit{Apart}(U_2) = (\rho_1, \rho_2)$ where $\rho_1 : \mathcal{V} \to V_1$ and $\rho_2 : \mathcal{V} \to V_2$ are bijections such that, for each $x \in U_2$, $\rho_2(x) = x$. We apply such bijections to syntactic objects as if they were substitutions: in other words, if ρ is one of such bijections and $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, then $\rho(\theta) = \{\rho(x_1)/\rho(t_1), \ldots, \rho(x_n)/\rho(t_n)\}$.

The backward unification is defined by exploiting the relation $\preceq_U \subseteq \mathit{Subst} \times \mathit{Subst}$, which formalizes the fact that a substitution is an instance of another one w.r.t. a fixed set of variables of interest $U \in \wp_f(\mathcal{V})$:

$$\sigma \preceq_U \sigma' \iff \exists \delta \in \mathit{Subst}. \forall x \in U. \sigma(x) = \delta(\sigma'(x)) \ . \tag{1}$$

For instance $\{x/a, y/u\} \preceq_{\{x,y\}} \{y/v\}$, since we may choose $\delta = \{x/a, v/u\}$, although this is not the case with the standard instance ordering on substitutions. Note that $\{x/a, y/u\} \not\preceq_{\{x,y,v\}} \{y/v\}$.

The concrete backward unification $\mathbf{U}^b_{\mathrm{Rs}} : \texttt{Rsub} \times \texttt{Rsub} \times \mathsf{Atoms} \times \mathsf{Atoms} \to \texttt{Rsub}$ is given by:

$$\begin{aligned}
\mathbf{U}^b_{\mathrm{Rs}}([\Theta_1, U_1], [\Theta_2, U_2], A_1, A_2) = [\{\mathrm{mgu}(\rho_1(\sigma_1), \rho_2(\sigma_2), \delta) \mid \sigma_1 \in \Theta_1, \sigma_2 \in \Theta_2, \\
\delta = \mathrm{mgu}(\rho_1(A_1), A_2), \rho_1(\sigma_1) \preceq_{\rho_1(U_1)} \mathrm{mgu}(\rho_2(\sigma_2), \delta)\}, \rho_1(U_1) \cup U_2] \ .
\end{aligned}$$

where $(\rho_1, \rho_2) = Apart(U_2)$, provided $vars(A_1) \subseteq U_1$, $vars(A_2) \subseteq U_2$, $\bot_{\mathrm{Rs}}$ in all the other cases. Here we improve over the standard unification by requiring that $\rho_1(\sigma_1)$ (the exit substitution) is an instance of $\mathrm{mgu}(\rho_2(\sigma_2), \delta)$ (the entry substitution) w.r.t. the variables of the calling atom [9]. By using the previously defined operators, we provide a goal-dependent, bottom-up semantics for logic programs. A denotation is an element in the set of monotonic maps $\mathcal{D}en = \mathsf{Atoms} \to \mathtt{Rsub} \to \mathtt{Rsub}$ and the semantic functions $\mathcal{P} : \mathsf{Progs} \to \mathcal{D}en$, $\mathcal{C} : \mathsf{Clauses} \to \mathcal{D}en \to \mathcal{D}en$ and $\mathcal{B} : \mathsf{Body} \to \mathcal{D}en \to \mathtt{Rsub} \to \mathtt{Rsub}$ are defined according to [1] as follows.

$$\mathcal{P}[\![P]\!] = lfp \lambda d. \left(\bigsqcup_{cl \in P} \mathcal{C}[\![cl]\!]d \right)$$

$$\mathcal{C}[\![H \leftarrow B]\!]dAx = \pi_{\mathrm{Rs}}(\mathbf{U}^b_{\mathrm{Rs}}(x', x, H, A), x)$$

$$\text{where } x' = \mathcal{B}[\![B]\!]d(\pi_{\mathrm{Rs}}(\mathbf{U}^f_{\mathrm{Rs}}(x, vars(H \leftarrow B), A, H), [\emptyset, vars(H \leftarrow B))]$$

$$\mathcal{B}[\![\lambda]\!]dx = x$$

$$\mathcal{B}[\![A, B]\!]dx = \mathcal{B}[\![B]\!]d(dAx)$$

Given a program P and an atom A, the set of computed answers for A in P is given by $\mathcal{P}[\![P]\!]A([\{\epsilon\}, vars(A)])$.

3.2 A Different Concrete Domain

It turns out that the domain of idempotent substitutions is too concrete for the above semantics of logic programs. Given a goal $\mathtt{p(x,y)}$ in a program P, we do not really want to distinguish between the answers $\{x/y\}$, $\{y/x\}$ and $\{x/u, y/u\}$. Note that while $\{x/y\}$ and $\{y/x\}$ can be obtained from each other by renaming, the same does not hold for $\{x/y\}$ and $\{x/u, y/u\}$. Actually, in the literature we find several alternatives to solve this problem, like ex-equations [19], Herbrand constraints and the domain of existential substitutions *ESubst* [13]. The common viewpoint is that substitutions are viewed as constraints and that variables which are not of interest (like u in the previous example) are existentially quantified. We show that such domains naturally arise as appropriate equivalence classes of substitutions. Given two substitutions θ and θ' and a set of variables U, we define the equivalence relation:

$$\theta \sim_U \theta' \iff \exists \rho \in Ren. \forall v \in U.\ \theta(v) = \rho(\theta'(v)) \tag{2}$$

which is the equivalence relation induced by the preorder $\preceq_U$. By exploiting this relation, we can define a new domain $ISubst_\sim$ of *existential substitutions* as the disjoint union of all the $ISubst_{\sim_U}$, for $U \in \wp_f(\mathcal{V})$:

$$ISubst_\sim = \biguplus_{U \in \wp_f(\mathcal{V})} ISubst_{\sim_U} \ .$$

In the following we write $[\theta]_U$ for the equivalence class of θ w.r.t. $\sim_U$. For example, if $U = \{x, y\}$, then $[\{x/y\}]_U = [\{y/x\}]_U = [\{x/u, y/u\}]_U$. However, if

$U' = \{x, y, u\}$, so that u is included in the variables of interest, we have that $[\{x/y\}]_{U'} = [\{y/x\}]_{U'} \neq [\{x/u, y/u\}]_{U'}$.

Given $U, V \in \wp_f(\mathcal{V})$, $[\theta_1]_U, [\theta_2]_V \in \mathit{ISubst}_\sim$, we define:

$$\mathrm{mgu}([\theta_1]_U, [\theta_2]_V) = [\mathrm{mgu}(\theta_1', \theta_2')]_{U \cup V}$$

where $\theta_1' \sim_U \theta_1$, $\theta_2' \sim_U \theta_2$, $\mathrm{dom}(\theta_1') = U$, $\mathrm{dom}(\theta_2') = V$ and $\mathrm{rng}(\theta_1') \cap \mathrm{rng}(\theta_2') = \emptyset$. It can be proved that the definition does not depend from the choice of representatives. It is worth noting that the resultant domain $\mathit{ISubst}_\sim$ is isomorphic to the domain *ESubst* by Jacobs and Langen [13], which is based on a non standard definition of substitution.

By exploiting the domain $\mathit{ISubst}_\sim$ we can define a domain which is a complete abstraction of `Rsub`. We lift the equivalence $\sim_U$ to `Rsub`.

$$[\Theta_1, U_1] \sim [\Theta_2, U_2] \iff U_1 = U_2 \text{ and } \forall \theta \in \Theta_1 \exists \theta' \in \Theta_2 . \theta \sim_{U_1} \theta' \text{ and vice versa} \tag{3}$$

As shown in [1], $\sim$ is a congruence w.r.t. to the operations in `Rsub`. Since $[\{\sigma\}, U] \sim [\{\sigma'\}, U]$ iff $\sigma \sim_U \sigma'$ and moreover $[\Theta, U] = \bigsqcup_{\mathrm{Rs}} \{[\{\sigma\}, U] \mid \sigma \in \Theta\}$, it turns out that $\texttt{Rsub}_\sim$ (the quotient set of `Rsub` w.r.t. $\sim$) is isomorphic to the following domain:

$$\texttt{Psub} = \{[\Sigma, U] \mid \Sigma \subseteq \mathit{ISubst}_{\sim_U}, U \in \wp_f(\mathcal{V})\} \cup \{\bot_{\mathrm{Ps}}, \top_{\mathrm{Ps}}\}.$$

In the following, given $[\Sigma, U] \in \texttt{Psub}$, we will abuse the notation and write $\theta \in \Sigma$ for $[\theta]_U \in \Sigma$. The operators and semantic functions over `Rsub` induce corresponding operators on `Psub` by means of the isomorphism between `Psub` and $\texttt{Rsub}_\sim$. First of all, let us define the auxiliary operations of unification and matching. Forward and backward unification will be built starting from them. The concrete unification $\mathsf{unif}_{\mathrm{Ps}} : \texttt{Psub} \times \mathit{ISubst} \times \wp_f(\mathcal{V}) \to \texttt{Psub}$ is given by:

$$\mathsf{unif}_{\mathrm{Ps}}([\Sigma, U], \delta, V) = [\{\mathrm{mgu}([\sigma]_U, [\delta]_V) \mid [\sigma]_U \in \Sigma\}, U \cup V]$$

provided $\mathit{vars}(\delta) \subseteq V$. It is worth noting that when $V = U$, this is the standard unification which is usually considered in the literature on sharing. It is possible to define $\mathsf{unif}_{\mathrm{Ps}}$ to take an argument in $\mathit{ISubst}_\sim$ instead of a substitution and a set of variables. However, this would make the operation more general, since not all the elements of $\mathit{ISubst}_\sim$ admits a representative $[\theta]_U$ with $\mathit{vars}(\theta) \subseteq U$. Using this definition of $\mathsf{unif}_{\mathrm{Ps}}$, we are actually restricting our attention to this case, which simplifies the presentation of the abstract operators.

We then define the matching operation $\mathsf{match}_{\mathrm{Ps}} : \texttt{Psub} \times \texttt{Psub} \to \texttt{Psub}$ as:

$$\mathsf{match}_{\mathrm{Ps}}([\Theta_1, U_1], [\Theta_2, U_2]) = [\{\mathrm{mgu}([\theta_1]_{U_1}, [\theta_2]_{U_2}) \mid \theta_1 \preceq_{U_1} \theta_2, [\theta_1]_{U_1} \in \Theta_1, [\theta_2]_{U_2} \in \Theta_2\}, U_2]$$

provided $U_1 \subseteq U_2$. These can be used to define the forward and backward unification over `Psub` as follows:

$$\mathbf{U}^f_{\mathrm{Ps}}([\Sigma, U_1], U_2, A_1, A_2) = \mathsf{unif}_{\mathrm{Ps}}(\rho_1([\Sigma, U_1]), \mathrm{mgu}(\rho_1(A_1) = A_2), U_2)$$

$$\mathbf{U}^b_{\mathrm{Ps}}([\Sigma_1, U_1], [\Sigma_2, U_2], A_1, A_2) = \mathsf{match}_{\mathrm{Ps}}(\rho_1([\Sigma_1, U_1]), \mathsf{unif}_{\mathrm{Ps}}([\Sigma_2, U_2], \mathrm{mgu}(\rho_1(A_1) = A_2), \rho_1(U_1)))$$

where $(\rho_1, \rho_2) = \mathit{Apart}(U_2)$. These can be easily proved to correspond to the ones for $\mathtt{Rsub}_\sim$ with simple algebraic manipulations.

4 The Abstract Domain $\mathtt{ShLin}^\omega$

In this section we define a new abstract domain $\mathtt{ShLin}^\omega$. Since it is infinite and contains infinite ascending chains, it cannot be directly used for the analysis. It should be thought of as a general framework from which other domains can be easily derived by abstraction. The idea underlying the construction of $\mathtt{ShLin}^\omega$ is to count the exact number of occurrences of the same variable in a term. In this way, it extends the standard domain `Sharing` by recording, for each $v \in \mathcal{V}$ and $\theta \in \mathit{ISubst}$, not only the set $\{w \mid v \in \mathit{vars}(\theta(w))\}$ but the pairs $\{\langle w, \mathit{occ}(v, \theta(w))\rangle \mid v \in \mathit{vars}(\theta(w))\}$ with the use of multisets. We call *ω-sharing group* a multiset of variables and we build a domain which works on ω-sharing groups.

$$\mathtt{ShLin}^\omega = \{[S, U] \mid U \in \wp_f(\mathcal{V}), S \subseteq \wp_m(U), S \neq \emptyset \Rightarrow \emptyset \in S\} \cup \{\bot_\omega, \top_\omega\} \quad (4)$$

where $\bot_\omega$ is the least element, $\top_\omega$ is the greatest and $[S_1, U_1] \leq_\omega [S_2, U_2]$ iff $U_1 = U_2$ and $S_1 \subseteq S_2$. $\mathtt{ShLin}^\omega$ is a complete lattice and the l.u.b. is denoted by $\sqcup_\omega$.

Given a substitution θ and a variable $v \in \mathcal{V}$, we denote by $\theta^{-1}(v)$ the ω-sharing group B with support $\{w \mid v \in \mathit{vars}(\theta(w))\}$ and $B(w) = \mathit{occ}(v, \theta(w))$. Therefore $\theta^{-1}(v)$ maps each variable w to the number of occurrences of v in $\theta(w)$. We define the abstraction for a substitution θ w.r.t. the variables of interest in U:

$$\alpha_U(\theta) = \{\theta^{-1}(v)|_U \mid v \in \mathcal{V}\} \ . \quad (5)$$

Intuitively, each $B \in \alpha_U(\theta)$ corresponds to one or more variables which are shared by all the variables in B, each with the exact number of occurrences. For example, given $\theta = \{x/t(y, u, u), z/y, v/u\}$ and $U = \{w, x, y, z\}$, we have $\theta^{-1}(u) = x^2vu$, $\theta^{-1}(y) = xyz$, $\theta^{-1}(z) = \theta^{-1}(v) = \theta^{-1}(x) = \emptyset$ and $\theta^{-1}(s) = s$ for all the other variables (included w). Projecting over U we obtain $\alpha_U(\theta) = \{x^2, xyz, w, \emptyset\}$. Note that if $\theta_1 \sim_U \theta_2$ then $\alpha_U(\theta_1) = \alpha_U(\theta_2)$. Therefore, we can lift α_U to obtain a Galois insertion between `Psub` and $\mathtt{ShLin}^\omega$ as follows: $\alpha_\omega(\bot_{\mathrm{Rs}}) = \bot_\omega$, $\alpha_\omega(\top_{\mathrm{Rs}}) = \top_\omega$ and

$$\alpha_\omega([\Sigma, U]) = \left[\bigcup\{\alpha_U(\theta) \mid \theta \in \Sigma\}, U\right] \ . \quad (6)$$

In the following, we will omit to explicitly define the abstraction and concretization maps on the top and bottom elements of the domains. The projection operation is defined pointwise in the obvious way:

$$\pi_\omega([S_1, U_1], [S_2, U_2]) = [\{B|_{U_2} \mid B \in S_1\}, U_1 \cap U_2] \ . \quad (7)$$

4.1 Unification and Matching

The unification is much more complex and we prefer to characterize the operation of unification by means of graph theoretic notions. We first need to define the multiplicity of an ω-sharing group B in a term t as follows:

$$\chi(B,t) = \sum_{v \in [\![B]\!]} B(v) \cdot occ(v,t) \ . \tag{8}$$

For instance, $\chi(x^3yz^4, t(x,y,f(x,y,z))) = 3 \cdot 2 + 1 \cdot 2 + 4 \cdot 1 = 12$. If $B \in \alpha_U(\theta)$ represents the variable v (i.e., $B = \theta^{-1}(v) \cap U$) then $\chi(B,t)$ is the number of occurrences of v in $\theta(t)$.

A *sharing graph* is a directed multigraph whose nodes are labeled with sharing groups. In formulas, it is a tuple $\langle N, l, E\rangle$ where N is the finite set of nodes, $l : N \to \wp_m(\mathcal{V})$ is the labeling function and $E \in \wp_m(N \times N)$ is the multiset of edges. A *balanced* sharing graph for the equation $t_1 = t_2$ and a set of ω-sharing groups S is a sharing graph $G = \langle N, l, E\rangle$ such that:

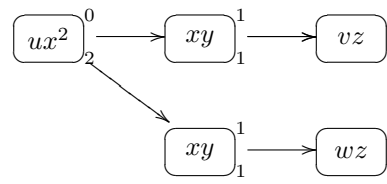
1. G is connected;
2. for each node $s \in N$, $l(s) \in S$;
3. for each node $s \in N$, the out-degree of s is equal to $\chi(l(s), t_1)$ and the in-degree of s is equal to $\chi(l(s), t_2)$.

Given a balanced sharing graph $G = \langle N, l, E\rangle$, we define the *resultant ω-sharing group* of G as $res(G) = \uplus_{s \in N} l(s)$. The set of resultants ω-sharing groups for $t_1 = t_2$ given a set S of sharing groups is denoted by:

$$\mathrm{mgu}(S, t_1 = t_2) = \{res(G) \mid G \text{ is a balanced sharing graph for } S \text{ and } t_1 = t_2\} \ .$$

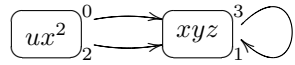
A sharing graph represents a possible way to merge together several sharing groups by unifying them with a given binding. Assume, for $j \in \{1,2\}$, $B_j \in \alpha_U(\theta)$, i.e., there exist $v_j \in \mathcal{V}$ such that $B_j = \theta^{-1}(v_j) \cap U$. When unifying θ with the binding $t_1 = t_2$, we know that $\mathrm{mgu}(\theta, t_1 = t_2) = \mathrm{mgu}(\theta(t_1) = \theta(t_2)) \circ \theta$ and that $\theta(t_i)$ contains $\chi(B_j, t_i)$ instances of v_j. An arrow from the sharing group B_1 to B_2 represents the fact that, in $\mathrm{mgu}(\theta(t_1) = \theta(t_2))$, one of the copies of v_1 is aliased with one of the copies of v_2. The third condition for balanced sharing graphs implies that each copy of v_j is aliased with some other variable. Therefore, we are considering the case when $\theta(t_1)$ and $\theta(t_2)$ only differs for the variables occurring in them. Although this is restrictive in general, it is enough to reach optimality when equations are reduced to solved normal form.

Example 1. Let $S = \{ux^2, xy, vz, wz, xyz\}$. The following is a balanced sharing graph for $t(x) = r(y,z)$ and S:



where pedices and apices on a sharing group B are respectively the value of $\chi(B, t(x))$ and $\chi(B, r(y, z))$. Therefore $uvwx^4y^2z^2 \in \mathrm{mgu}(S, t(x) = r(y, z))$.

Example 2. Let $S = \{ux^2, xy, vz, wz, xyz\}$ and $U = \{u, v, w, x, y, z\}$. The following is a balanced sharing graph for $x = r(y, y, z)$ and S:



where pedices and apices on a sharing group B are respectively the value of $\chi(B, x)$ and $\chi(B, r(y, y, z))$. Therefore $ux^3yz \in \mathrm{mgu}(S, x = r(y, y, z))$. Note that this sharing group can actually be generated by the substitution $\theta = \{x/r(v_1, v_1, v_2), y/v_2, z/v_2, u/v_1, v/a, w/a\}$ where a is a ground term. It is the case that $\alpha_U(\theta) \subseteq S$ and $\mathrm{mgu}(\theta, x = r(y, y, z))$ performs exactly the unification depicted by the sharing graph. Note that $\theta(x) = r(v_1, v_1, v_2)$ and $\theta(r(y, y, z)) = r(v_2, v_2, v_2)$ only differ for the choice of variables. After their unification, the first occurrence of v_1 from $\theta(x)$ get aliased with the first occurrence of v_2 in $\theta(r(y, y, z))$. Correspondingly, there is an edge in the sharing graph from $\theta^{-1}(v_1)$ and $\theta^{-1}(v_2)$.

We define $\mathrm{mgu}(S, \theta)$ with $\theta \in \mathit{ISubst}$ by induction on the number of bindings:

$$\mathrm{mgu}(S, \epsilon) = \epsilon \qquad \mathrm{mgu}(S, \{x/t\} \uplus \theta) = \mathrm{mgu}(\mathrm{mgu}(S, x = t), \theta)$$

where $\uplus$ is the disjoint union of bindings in a substitution. Now, we are ready to define the abstract unification in $\mathtt{ShLin}^\omega$ as:

$$\mathsf{unif}_\omega([S, U_1], \delta, U_2) = [\mathrm{mgu}(S \cup \{\{\!\{v\}\!\} \mid v \in U_2 \setminus U_1\}, \delta), U_1 \cup U_2]$$

provided that $\mathit{vars}(\delta) \subseteq U_2$.

Theorem 1. *The operation* unif_ω *is optimal and correct w.r.t.* $\mathsf{unif}_{\mathrm{Ps}}$

Note that the operation unif_ω is designed by first extending the domain in order to include all the variables in U_2 and then performing the operation, and that this construction yields to an optimal abstraction of the concrete unification. This is not the case for other abstract domains, e.g. `Sharing`, as shown in [1]. The proof of correctness is by induction on the number of bindings in the substitution. The proof of optimality is more complex and it is based on a notion of parallel abstract unification of multigraphs. We show that parallel unification gives the same results of the iterated use of the single binding unification.

Concerning the matching operation, assume $\alpha_\omega([\Sigma_i, U_i]) = [S_i, U_i]$ for $i \in \{1, 2\}$. If we unify a substitution $\sigma_1 \in \Sigma_1$ with $\sigma_2 \in \Sigma_2$ such that $\sigma_1 \preceq_{U_1} \sigma_2$, then $\sigma_1 \preceq_{U_1} \mathrm{mgu}(\sigma_1, \sigma_2)$ and thus $\alpha_{U_1}(\mathrm{mgu}(\sigma_1, \sigma_2)) \in S_1$. Moreover, the sharing groups in S_2 which do not contain any variable in U_1 are not affected by the unification, since the corresponding existential variable does not appear in $\sigma_2(v)$

for any $v \in U_1$. We can now design an abstract matching operation which follows the above guidelines:

$$\mathsf{match}_\omega([S_1, U_1], [S_2, U_2]) = \left[S_2' \cup \left\{X \in (S_2'')^* \mid X_{|U_1} \in S_1\right\}, U_1\right]$$

where $S_2' = \{B \in S_2 \mid B_{|U_1} = \emptyset\}$, $S_2'' = S_2 \setminus S_2'$ and $U_1 \subseteq U_2$. Here we also use the auxiliary operation $(\_)^*$ defined as:

$$S^* = \{\uplus\mathcal{S} \mid \mathcal{S} \in \wp_m(S)\} \quad . \tag{9}$$

Note that match_ω is very similar to the analogous operation for `Sharing` defined in [1].

Theorem 2. *The operation* match_ω *is optimal and correct w.r.t.* $\mathsf{match}_{\mathrm{Ps}}$. *Furthermore, it is complete when the second argument of* $\mathsf{match}_{\mathrm{Ps}}$ *contains a single substitution.*

The forward and backward unification operators $\mathbf{U}_\omega^f$ and $\mathbf{U}_\omega^b$ for $\mathtt{ShLin}^\omega$ are obtained by the corresponding definitions $\mathbf{U}_{\mathrm{Ps}}^f$ and $\mathbf{U}_{\mathrm{Ps}}^b$ for `Psub`, by replacing the matching and unification operations with their abstract counterparts. By exploiting the above results, it is now an easy task to show the following corollary.

Corollary 1. *The operators* $\mathbf{U}_\omega^f$ *and* $\mathbf{U}_\omega^b$ *are correct and optimal w.r.t.* $\mathbf{U}_{\mathrm{Ps}}^f$ *and* $\mathbf{U}_{\mathrm{Ps}}^b$.

4.2 A Characterization for Resultant Sharing Groups

The concept of resultant ω-sharing group, while suggestive and very intuitive, does not help in practice in the implementation of the operations. Although $\mathtt{ShLin}^\omega$ has not been designed to be directly implemented, some of its abstractions could. Providing a simpler definition for the set of resultant ω-sharing groups could help in developing the abstract operators for its abstractions. We show that given a set S of ω-sharing groups and an equation $t_1 = t_2$, the set of resultant ω-sharing groups has an elegant algebraic characterization.

Theorem 3. *Let* S *be a set of* ω*-sharing groups and* t_1, t_2 *be terms. Then* $B \in \mathrm{mgu}(S, t_1 = t_2)$ *iff* $B = \uplus_{i \in I} B_i$ *where* I *is a finite set,* $B_i \in S$ *for all* $i \in I$, *and*

$$\sum_{i \in I} \chi(B_i, t_1) = \sum_{i \in I} \chi(B_i, t_2) \geq |I| - 1 \quad .$$

Intuitively, the above condition ensures us that the out-degree and in-degree of the corresponding sharing graph $\sum_{i \in I} \chi(B_i, t_1)$ corresponds to the From the above theorem, we can now give an algebraic characterization of the abstract unification operator as follows.

$$\mathrm{mgu}(S, t_1 = t_2) = \left\{ \uplus\mathcal{S} \mid \mathcal{S} \in \wp_m(S), \sum_{B \in \mathcal{S}} \chi(B, t_1) = \sum_{B \in \mathcal{S}} \chi(B, t_2) \geq |\mathcal{S}| - 1 \right\}.$$

Example 3. Consider $S = \{xa, xb, z^2, zc\}$ and the equation $x = z$. Then if we choose $X = \{\!\{xa, xb, z^2\}\!\}$, we have $\chi(X, x) = 2 = \chi(X, z) \geq |X| - 1$. Therefore $x^2z^2ab \in \mathrm{mgu}(S, x = z)$. If we take $X = \{\!\{xa, xb, zc, zc\}\!\}$, although $\chi(X, x) = 2 = \chi(X, z)$, we have $|X| - 1 = 3$. Actually, $z^2c^2x^2ab \notin \mathrm{mgu}(S, x = z)$.

5 Domains for Linearity and Aliasing

In this section we show that two domains for sharing analysis with linearity information, namely the domain proposed by King in [14] and the classic reduced product $\mathtt{Sharing} \times \mathtt{Lin}$, can be obtained as abstraction of $\mathtt{ShLin}^\omega$. This allows us to design optimal abstract operators for both domains, by exploiting the results for $\mathtt{ShLin}^\omega$.

5.1 King's Domain

We first consider the domain for combined analysis of sharing and linearity introduced by King in [14]. We call *2-sharing group* a map $o : \mathcal{V} \to \{0, 1, \infty\}$ such that its support $\|o\| = \{v \in \mathcal{V} \mid o(v) \neq 0\}$ is finite. We write $o_m(x)$ to denote $o(x)$ if $o(x) \leq 1$, 2 otherwise (where $n \leq \infty$ for each $n \in \mathbb{N}$). Intuitively, a 2-sharing group o represents the sets $\gamma_2(o)$ of ω-sharing group given by

$$\gamma_2(o) = \{B \in \wp_m(\mathcal{V}) \mid \|o\| = \|B\| \wedge \forall x \in \|o\|. o_m(x) \leq B(x) \leq o(x)\} \ . \quad (10)$$

We denote by $Sg^2(V)$ the set of 2-sharing groups whose support is a subset of V. We use a polynomial notation for 2-sharing groups as for ω-sharing groups: a group o such that $\|o\| = \{x, y, z\}$, $o(x) = o(y) = 1$ and $o(z) = \infty$ will be denoted by xyz^∞. We also use $\emptyset$ for the 2-sharing group with empty support.

The idea is to use 2-sharing groups to keep track of linearity: If $o(x) = \infty$, it means that the variable x is not linear in the sharing group o. In [14] the number 2 is used as an exponent instead of ∞, but we prefer this notation to be coherent with ω-sharing groups. In the rest of this subsection, we use the term "sharing group" as a short form of 2-sharing group.

We first need to define an order relation over sharing groups, as follows:

$$o \leq o' \iff \|o\| = \|o'\| \wedge \forall x \in \|o\|. \ o(x) \leq o'(x) \ . \quad (11)$$

Given a sharing group o, we also define the *delinearization* operator o^2 as the sharing group $o' \geq o$ such that $\forall x \in \|o\|. o'(x) = \infty$. The operator is extended pointwise to sets and multisets. The domain we are interested in is the following:

$$\mathtt{ShLin}^2 = \big\{[S, U] \mid S \in \wp_\downarrow(Sg^2(U)), U \in \wp_f(\mathcal{V}), S \neq \emptyset \Rightarrow \emptyset \in S\big\} \cup \{\top_2, \bot_2\} \quad (12)$$

where $\wp_\downarrow(Sg^2(U))$ is the powerset of downward closed 2-sharing groups according to $\leq$ and $[S_1, U_1] \leq_2 [S_2, U_2]$ iff $U_1 = U_2$ and $S_1 \subseteq S_2$. Since we only consider downward closed sets, we are not able to state that some variable is definitively

non-linear. We define an adjunction with $\mathtt{ShLin}^\omega$ via the following concretization map $\gamma_2 : \mathtt{ShLin}^2 \to \mathtt{ShLin}^\omega$ which lifts the one given in (10):

$$\gamma_2([S,U]) = \left[\bigcup\{\gamma_2(o) \mid o \in S\}, U\right] \ . \tag{13}$$

It is possible to compose $\langle\alpha_\omega, \gamma_\omega\rangle$ with $\langle\alpha_2, \gamma_2\rangle$ to obtain a Galois insertion between $\mathtt{Psub}$ and $\mathtt{ShLin}^2$.

Example 4. Let $\theta = \{x/t(u,u,v), y/t(u,v), z/v\}$ with $U = \{x,y,z\}$ and consider its abstraction on $\mathtt{ShLin}^\omega$. We obtain $\alpha_\omega([\{\theta\},U]) = [\{x^2y, xyz\}, U]$ while $\alpha_2([\{x^2y, xyz\}, U]) = [\{x^\infty y, xy, xyz\}, U]$, since xy is introduced by the downward closure.

The least upper bound of $\mathtt{ShLin}^2$ is given by the downward closure of the union of the first components. Projection is given by:

$$\pi_2([S_1,U_1],[S_2,U_2]) = [\{o_{|U_2} \mid o \in S_1\}, U_1 \cap U_2] \ , \tag{14}$$

where $o_{|X}(v)$ is $o(v)$ if $v \in X$, 0 otherwise. Given two sharing groups o and o' we define:

$$o \square o' = \lambda v \in \mathcal{V}.o(v) \oplus o'(v) \tag{15}$$

where $0 \oplus x = x \oplus 0 = x$ and $\infty \oplus x = x \oplus \infty = 1 \oplus 1 = \infty$. We will use $\square\{\!\{o_1, \ldots, o_n\}\!\}$ for $o_1 \square \cdots \square o_n$. Note that $o^2 = o \square o$. According to the corresponding definition for ω-sharing groups, we also need to define the following auxiliary function.

$$S^* = \{\square\mathcal{S} \mid \mathcal{S} \in \wp_m(S)\} \ . \tag{16}$$

The minimum and maximum multiplicity of o in t are defined as follows:

$$\chi_m(o,t) = \sum_{v \in [\![o]\!]} o_m(v) \cdot occ(v,t) \qquad \chi_M(o,t) = \sum_{v \in [\![o]\!]} o(v) \cdot occ(v,t) \tag{17}$$

If B is an ω-sharing group represented by o, i.e., $B \in \gamma_2(o)$, then $\chi_m(o,t) \leq \chi(B,t) \leq \chi_M(o,t)$. Actually, not all the values between $\chi_m(o,t)$ and $\chi_M(o,t)$ may be assumed by $\chi(B,t)$, but this will not affect the precision of the abstract operators. Note that the maximum multiplicity $\chi_M(o,t)$ either is equal to the minimum multiplicity $\chi_m(o,t)$ or it is infinite. According to the above definitions, we can now define the multiplicity of a multiset of sharing groups.

$$\chi(Y,t) = \left\{n \mid \sum_{o \in Y} \chi_m(o,t) \leq n \leq \sum_{o \in Y} \chi_M(o,t)\right\} \ . \tag{18}$$

Again, this is a superset of all the possible values which can be obtained by combining the multiplicities of all the sharing groups in Y. But, as we will show later, this definition is sufficiently accurate to allow us to design the optimal abstract unification operator. This can actually be defined as follows:

$$\mathrm{mgu}(S, x = t) = {\downarrow}\{\square Y \mid Y \subseteq_m S, \exists n \in \chi(Y,x) \cap \chi(Y,t).\ n \geq |Y| - 1\} \ , \tag{19}$$

where $\downarrow X$ is the downward closure of X w.r.t. $\leq$. The basic idea is to check, for each $Y \subseteq_m S$, if there exists an instance of the ω-sharing groups of Y which satisfies the condition in Theorem 3.

Example 5. Let $S = \downarrow\{x^\infty a, x^\infty b, x^\infty c, z^\infty\}$ and $Y = \{\!\{x^\infty a, x^\infty b, xc, z^\infty\}\!\}$. We have that $\chi(Y, x) = \{n \mid n \geq 5\}$ and $\chi(Y, t(z, z)) = \{n \mid n \geq 4\}$. Since $t(z, z)$ contains two occurrences of z, the "actual" multiplicity of the sharing group z^∞ in $t(z, z)$ should be a multiple of 2. But we do not need to check this condition and can safely approximate this set with $\{n \mid n \geq 4\}$. Intuitively, this works because we can always choose a multiple which is contained in both $\chi(Y, x)$ and $\chi(Y, t)$ and which is an "actual" multiplicity. For instance, we can take $n = 6 \in \chi(Y, x) \cap \chi(Y, t(z, z))$ and since we have $6 \geq 3 = |Y| - 1$, we get that the sharing group $\square Y = x^\infty abcz^\infty$ belongs to $\mathrm{mgu}(S, x = t(z, z))$.

The same computation may be realized on the concrete side with the substitution $\theta = \{x/t(t(a, a, c), t(b, b, c)), z/t(v, v, v)\}$ on the variables of interest $U = \{x, z, a, b, c\}$. Note that $\alpha_2 \circ \alpha_\omega([\{\theta\}, U]) = [S, U]$ and $\mathrm{mgu}(\theta, x = t(z, z)) = \theta' = \{x/t(t(v, v, v), t(v, v, v)), z/t(v, v, v), a/v, b/v, c/v\}$ such that $\alpha_2 \circ \alpha_\omega([\{\theta'\}, U]) = [S', U]$ with $x^\infty abcz^\infty \in S'$. This shows that the 2-sharing group computed with the abstract mgu can be obtained with a concrete unification.

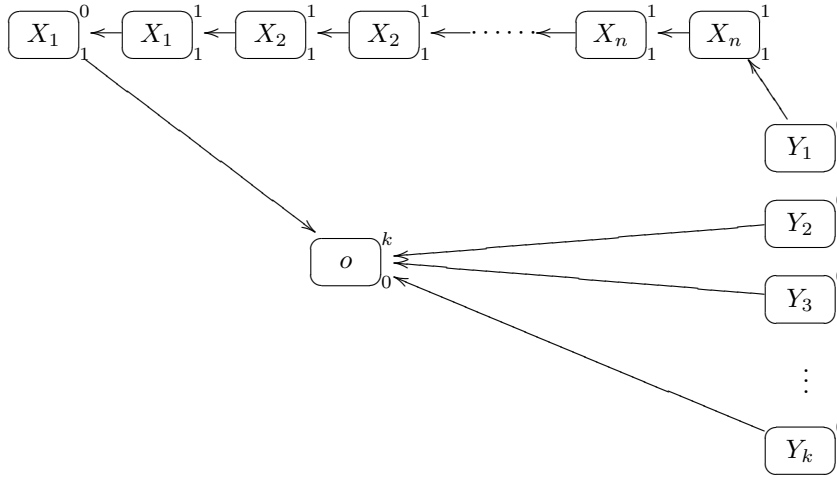
By exploiting the particular structure of 2-sharing groups, we can rewrite the mgu operator in a more constructive way, to be used in practice to implement the abstract operator. Given a set of sharing groups S and an equation $t_1 = t_2$, we define, for $i \in \{1, 2\}$ and $j \in \mathbb{N}$, $S_i^j = \{o \in S \mid \chi_M(o, t_i) = j\}$, $S_i = \{o \in S \mid \chi_M(o, t_i) \neq 0\}$, $P_i^j = S_i^j \setminus S_{3-i}$, $P_i = S_i \setminus S_{3-i}$ and $C^i = S_1^i \cap S_2^i$. We also write $S_i^{nl} = \{o \in S \mid \chi_M(o, t_i) > 1\}$. Note that, if t_1 is a variable, $S_1^{nl} = S_1^\infty$. Then, by considering x as t_1 and t as t_2, we can rewrite the mgu operator as follows.

$$
\begin{aligned}
&\mathrm{mgu}(S, x = t) = C^0 \cup \\
&\quad \downarrow\Big(\{\square X^2 \mid X \subseteq S_1 \cup S_2, X \cap S_1^{nl} \neq \emptyset, X \cap S_2^{nl} \neq \emptyset\} \cup \\
&\qquad \{\square X^2 \mid X \subseteq S_2^1, X \cap S_1^{nl} \neq \emptyset\} \cup \\
&\qquad \{o\square(\square X^2) \mid o \in P_1, X \subseteq S_2^1, X \cap S_1^{nl} \neq \emptyset \vee o \in P_1^\infty, X \cap P_2 \neq \emptyset\} \cup \\
&\qquad \{\square X^2 \mid X \subseteq S_1^1, X \cap S_2^{nl} \neq \emptyset\} \cup \\
&\qquad \{o\square(\square X^2) \mid o \in P_2, X \subseteq S_1^1, X \cap S_2^{nl} \neq \emptyset \vee o \in P_2^\infty, X \cap P_1 \neq \emptyset\} \cup \\
&\qquad \{\square X^2 \mid X \subseteq C^1\} \cup \\
&\qquad \{o\square Y\square(\square X^2) \mid o \in P_2, X \subseteq C^1, Y \subseteq_m P_1^1, |Y| = \chi_M(o, t) \in \mathbb{N}^+\}\Big) \ .
\end{aligned}
\tag{20}
$$

The initial C^0 is the set of sharing groups which are not related to any of the variables in the binding. The next seven cases correspond to different choices of

a multiset $\mathcal{S} \subseteq_m S_1 \cup S_2$ of sharing groups which we want to merge. In the first case, we require that there is at least a non-linear sharing group for x and t, while in the second and third case we only require the existence of a non-linear sharing group for x. The second line corresponds to the case when we do not have any element in P_1, while the third case is applied when there is exactly one element of P_1 in $\mathcal{S}$. The fourth and fifth case are symmetric to the second and third, when all the sharing groups are linear for x and non-linear for t. Note that in the third case we could replace S_1^{nl} with S_1^{∞}, since x is a variable, but the same is not allowed in the fifth case. The sixth and seventh case are applied when all the sharing groups are linear for x and t, but at most elements in P_2 which may have a finite maximum multiplicity. Note that Y in the seventh case is a multiset, while X is a set in all of its occurrences.

Below we depict a typical sharing graph for the seventh case in (20), assuming that $X = \{X_1, \ldots, X_n\}$, $Y = \{\!\{Y_1, \ldots, Y_k\}\!\}$ and $\chi_M(o, t) = k > 0$.



An interesting property of (20) is that it also works when S is not downward closed: if $\downarrow S = \downarrow R$ then $\mathrm{mgu}(S, x = t) = \mathrm{mgu}(R, x = t)$. This means that we do not have to compute and carry on the downward closure of a set but only its maximal elements. This simplifies the implementation of the abstract mgu. Moreover, the seven cases are obtained by disjoint choices of the multiset $\mathcal{S} \subseteq_m S_1 \cup S_2$ of sharing groups, to avoid as much as possible any duplication.

Example 6. Let $S = \{x^{\infty}y, y^{\infty}b, y, xa, z\}$ and consider the equation $x = y$. By the first case of (20) we obtain $x^{\infty}y^{\infty}b^{\infty}$ and $x^{\infty}y^{\infty}b^{\infty}a^{\infty}$. From the second and third case we obtain respectively $x^{\infty}y^{\infty}$ and $x^{\infty}y^{\infty}a$. The fourth and sixth case do not generate any sharing group, while from the fifth and seventh we have respectively $y^{\infty}x^{\infty}a^{\infty}b$ and xya, which are redundant. We also add the original sharing group z which is not related to either x nor y, which is therefore contained in C^0. The final result is

$$\mathrm{mgu}(S, x = y) = \downarrow\{x^{\infty}y^{\infty}b^{\infty}, x^{\infty}y^{\infty}b^{\infty}a^{\infty}, x^{\infty}y^{\infty}, x^{\infty}y^{\infty}a, z\} \ .$$

It is worth noting that $S \neq {\downarrow}S$ and that $\mathrm{mgu}(S, x = y) = \mathrm{mgu}({\downarrow}S, x = y)$.

We can now define the abstract unification on $\mathtt{ShLin}^2$ by enlarging the domain before computing the mgu:

$$\mathsf{unif}_2([S, U_1], \theta, U_2) = [\{\mathrm{mgu}(S \cup \{\{\!\{v\}\!\} \mid v \in U_2 \setminus U_1\}, \theta\}, U_2] \tag{21}$$

The abstract matching follows the same pattern as match_ω, and it is defined as:

$$\mathsf{match}_2([S_1, U_1], [S_2, U_2]) = \left[S_2' \cup {\downarrow}\left\{o \in (S_2'')^* \mid o_{|U_1} \in S_1\right\}, U_2\right] \tag{22}$$

where $S_2' = \{o \in S_2 \mid o_{|U_1} = \emptyset\}$, $S_2'' = S_2 \setminus S_2'$ and $U_1 \subseteq U_2$.

We can prove that both the operators are correct and optimal, and match_2 is complete w.r.t. single substitutions in its second argument. The forward and backward unification operators $\mathbf{U}_2^f$ and $\mathbf{U}_2^b$ for $\mathtt{ShLin}^2$ are obtained by the corresponding definitions $\mathbf{U}_{\mathrm{Ps}}^f$ and $\mathbf{U}_{\mathrm{Ps}}^b$ for $\mathtt{Psub}$, by replacing the matching and unification operations with their abstract counterparts.

Theorem 4. *The operators $\mathbf{U}_2^f$ and $\mathbf{U}_2^b$ are correct and optimal w.r.t. $\mathbf{U}_{\mathrm{Ps}}^f$ and $\mathbf{U}_{\mathrm{Ps}}^b$.*

5.2 The Domain $\mathtt{Sharing} \times \mathtt{Lin}$

In this section we deal with the reduced product $\mathtt{ShLin} = \mathtt{Sharing} \times \mathtt{Lin}$. We briefly recall the definition of the abstract domain and show the abstraction function from King's domain $\mathtt{ShLin}^2$ to $\mathtt{ShLin}$.

$$\mathtt{ShLin} = \{[S, L, U] \mid U \in \wp_f(\mathcal{V}), S \subseteq \wp(U), (S \neq \emptyset \Rightarrow \emptyset \in S), \\ U \setminus \mathit{vars}(S) \subseteq L \subseteq U\} \cup \{\bot_{sl}, \top_{sl}\} \ .$$

In an object $[S, L, U]$, S encodes the sharing information between the variables of interest U, while L is the set of linear variable. To be more precise, $[S, L, U]$ stands for the substitutions θ such that $\theta(x)$ is linear for all $x \in L$ and $\|\theta^{-1}(v)\| \cap U \in S$ for each $v \in \mathcal{V}$. Since ground terms are linear, L contains $U \setminus \mathit{vars}(S)$. Note that the last component U is redundant since it can be computed as $L \cup \mathit{vars}(S)$.

We define $[S, L, U] \leq_{sl} [S', L', U']$ iff $U = U'$, $S \subseteq S'$, $L \supseteq L'$. The abstraction map from $\mathtt{ShLin}^2$ to $\mathtt{ShLin}$ is defined in the obvious way.

$$\alpha([S, U]) = [\{\|o\| \mid o \in S\}, \{x \mid \forall o \in S.\ o(x) \leq 1\}, U] \ . \tag{23}$$

We call *sharing group* an element of $\wp_f(\mathcal{V})$. As for the previous domains, we use the polynomial notation to represent sharing groups.

Example 7. We keep on Example 4 and compute the abstraction from $\mathtt{ShLin}^2$ to $\mathtt{ShLin}$: $\alpha([\{x^\infty y, xy, xyz\}, U]) = [\{xy, xyz\}, \{y, z\}, U]$. Note that the variable x is nonlinear, and that the domain cannot encode the information that x is linear in the sharing group xyz while it is not in xy.

The abstract operator for projection is straightforward.

$$\pi_{sl}([S_1, L_1, U_1], [S_2, L_2, U_2]) = [\{B \cap U_2 \mid B \in S_1\}, L_1 \cap U_2, U_1 \cap U_2] \ . \quad (24)$$

As far as the abstract unification is concerning, we want to design an abstract operator over `ShLin` which is optimal for the unification of a single binding. Fixed a set L of linear variables, we define the maximum multiplicity of a sharing group B in a term t as follows:

$$\chi^L_M(B,t) = \begin{cases} \sum_{v \in B} occ(v,t) & \text{if } B \cap vars(t) \subseteq L \\ \infty & \text{otherwise} \end{cases} \quad (25)$$

We also define the maximum multiplicity of a term t in (S, L) as:

$$\chi(S, L, t) = \max_{B \in S} \chi^L_M(B, t) \ . \quad (26)$$

Then we define the abstract unification $\mathrm{mgu}(S, L, x = t)$ as the pair (S', L') where S' is computed as in (20) with χ_M and $\square$ replaced by χ^L_M and $\cup$ respectively (we can obviously ignore the delinearization operator $(\_)^2$ since $B \cup B = B$). The set L' is computed according to the following definition:

$$L' = (U \setminus vars(S')) \cup \begin{cases} L \setminus (vars(S_1) \cap vars(S_2)) & \text{if } x \in L \text{ and } \chi(S, L, t) \leq 1 \\ L \setminus vars(S_1) & \text{otherwise, if } x \in L \\ L \setminus vars(S_2) & \text{otherwise, if } \chi(S, L, t) \leq 1 \\ L \setminus (vars(S_1) \cup vars(S_2)) & \text{otherwise} \end{cases} \quad (27)$$

where $S_1 = \{B \in S \mid \chi^L_M(B, x) \neq 0\}$ and $S_2 = \{B \in S \mid \chi^L_M(B, t) \neq 0\}$.

Example 8. Let $S = \{xv, xy, zw\}, L = \{x, y, v, w\}$ and consider the binding $x = t(y, z)$. Then $\chi^L_M(xv, t) = 0$, since $xv \cap vars(t) = \emptyset$, $\chi^L_M(xy, t) = 1$ and $\chi^L_M(zw, t) = \infty$. As a result $\chi(S, L, t) = \infty$. In words, it means that the sharing group zw is not linear in t and that t itself is not linear. Note that all the other sharing groups are linear w.r.t. x since $x \in L$. Applying equation (20) as stated above, we obtain $S' = \{xy, xyzvw, xzvw\}$ and $L' = \{w\}$. This is more precise that the standard operators for `Sharing` $\times$ `Lin` [9]. Actually even with the optimizations proposed in [12,11] or [3], the standard operator is not able to infer that the sharing group xyv is not a possible result of the concrete unification. Note that it would be possible in a domain for rational trees, where the unification of $\{x/t(t(v, y), c), z/w\}$ with $x/t(y, z)$ succeeds with $\{x/t(t(v, y), c), z/c, w/c, y/t(v, y)\}$. This means that we are able to exploit the occur-check of the unification in finite trees. As a consequence, our abstract unification operator is not correct w.r.t. a concrete domain of rational substitutions [15]. However, our results improve over the abstract unification operators of the domains in the literature even in some cases which do not involve the occur-check. For example, if $S = \{xa, xb, xy, z\}$, $L = \{x, a, b, z\}$ and given the binding $x/t(y, z)$, we are able to state that $xzab$ is not a member of $\mathrm{mgu}(S, x = t(y, z))$, but the domains in [12,3,11] cannot.

Although the abstract operator for $\mathrm{mgu}(S, L, x = t)$ over $\mathtt{ShLin}$ is optimal for the unification with a single binding, the optimal operator $\mathrm{mgu}(S, L, \theta)$ for a generic substitution θ cannot be obtained by considering one binding at a time. Actually, let us take $\theta = \{x/t(y,z), s/t(a,b)\}$, $S = \{xs, y, z, a, b\}$, $L = \{x, z, s, a, b\}$. After the first binding we obtain $S' = \{xsy, xsz, a, b\}$ and $L' = \{z, a, b\}$. After the second binding $S'' = \{xsya, xsyb, xsyab, xsza, xszb, xszab\}$ and $L'' = \{z\}$. However, the sharing group $xszab$ cannot be obtained in practice since x is linear in the sharing group xsz, although $x \notin L'$. If we work in the domain $\mathtt{ShLin}^2$ from $Z = \{xs, y^\infty, y, z, a, b\}$, we obtain $Z' = {\downarrow}\{x^\infty s^\infty y^\infty, xsz, a, b\}$ and $Z'' = {\downarrow}\{x^\infty s^\infty y^\infty a^\infty b^\infty, x^\infty s^\infty y^\infty a^\infty, x^\infty s^\infty y^\infty b^\infty, xsza, xszb\}$. Note that Z'' does not contain $xszab$.

In order to obtain an optimal operator for a substitution θ, the obvious solution is to perform the computation over $\mathtt{ShLin}^2$ and to abstract the solution into $\mathtt{ShLin}$. We believe that the implementation of the abstract unification on $\mathtt{ShLin}^2$ could be optimized for this particular case. The same happens to the matching operation. Also in this case, we believe that the easiest approach is to compute over $\mathtt{ShLin}^2$, which is not particularly onerous since the abstract constraint does not increase in size when moving from $\mathtt{ShLin}$ to $\mathtt{ShLin}^2$. Corresponding definitions for unif_{sl}, match_{sl}, $\mathbf{U}^f_{sl}$ and $\mathbf{U}^b_{sl}$ are immediate.

6 Conclusion and Future Work

We summarize the main results of this paper.

- We clarify the relationship between domains of substitutions with existentially quantified variables, such as *ESubst* [13], and the standard domain of idempotent substitutions. To the best of our knowledge, this is the first time that a direct correspondence between *ESubst* and a quotient of idempotent substitutions has been showed.
- We propose a new domain $\mathtt{ShLin}^\omega$ as a general framework for investigating sharing and linearity properties. We introduce the notion of *(balanced) sharing graph* as a generalization of the concept of alternating path [21,15] used for pair sharing analysis and provide optimal abstract operators for $\mathtt{ShLin}^\omega$. By using sharing graphs instead of alternating paths, we also gain the ability to exploit the occur-check condition in order to avoid inexistent pair-sharing information (see Example 8).
- We show that $\mathtt{ShLin}^\omega$ is a useful starting point for studying further abstractions. We obtain the optimal operators for forward and backward unification in $\mathtt{Sharing} \times \mathtt{Lin}$ and King's domain $\mathtt{ShLin}^2$. This is the first paper which shows optimality results for a domain obtained by combining sharing and linearity information. Moreover, we propose an abstract unification algorithm which is strictly more precise than the other operators in the literature. Actually in [5] a variant of $\mathtt{Sharing} \times \mathtt{Lin}$ is proposed, based on *set logic programs*. However, despite the claim in the paper, the proposed operators are not optimal, as shown in [11]. Also the operators in [15] for $\mathtt{ASub}$ are not optimal when working over finite trees.

Recently, Lagoon and Stuckey proposed in [17] a new domain for encoding sharing and linearity information based on a notion of relation graph. Actually, relation graphs are used to represent the abstract objects and alternating paths to compute abstract unification. As a result, their abstract unification is not optimal on finite trees, since alternating paths cannot exploit the occur-check condition to avoid inexistent pair-sharing information. On the contrary, we use sharing graphs to compute abstract unification and multisets as abstract objects. Although the authors do not compare their domain to King's domain $\texttt{ShLin}^2$, we think that $\texttt{ShLin}^2$ is, at least, as precise as Lagoon and Stuckey's domain Ω_{Def} as far as pair-sharing information is concerning.

Several things remain to be explored: first of all, we plan to analyze the domain $\texttt{PSD} \times \texttt{Lin}$ [2] in our framework and, possibly, to devise a variant of $\texttt{ShLin}^2$ which enjoys a similar closure property for redundant sharing groups. This could be of great impact on the efficiency of the analysis. Moreover, we need to study the impact on the precision and performance by adopting the new optimal operators, possibly by implementing our operators in some well-known static analyzer.

In the recent years, many efforts has been made to study the behavior of logic programs in the domain of *rational trees* [15,22], since they formalize the standard implementations of logic languages. We have shown that our operators, which are optimal for finite trees, are not correct for rational trees , since they exploit the occur-check to reduce the sharing groups generated by the abstract unification (see Ex. 8). It would be interesting to adapt our framework to work with rational trees, in order to obtain optimal operators also for this case.

References

1. G. Amato and F. Scozzari. Optimality in goal-dependent analysis of sharing. Technical Report TR-02-06, Dipartimento di Informatica, Univ. di Pisa, May 2002.
2. R. Bagnara, P. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.
3. R. Bagnara, E. Zaffanella, and P. M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. In *Proc. of ACM Conf. PPDP*, pp. 103–114, 2000.
4. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *ICLP*, pp. 79–93, 1991.
5. M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In *Static Analysis Symposium*, pp. 68–82, 1997.
6. A. Cortesi, G. Filé, and W. W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, 1996.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL*, pp. 269–282, 1979.
8. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
9. W. Hans and S. Winkler. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report 92–27, Technical University of Aachen (RWTH Aachen), 1992.

10. M. V. Hermenegildo and F. Rossi. Strict and nonstrict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
11. P. M. Hill, E. Zaffanella, and R. Bagnara. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. Available at `http://www.cs.unipr.it/~bagnara/`.
12. J. Howe and A. King. Three Optimisations for Sharing. *Theory and Practice of Logic Programming*, 3(2):243–257, 2003.
13. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.
14. A. King. A synergistic analysis for sharing and groundness which traces linearity. In *ESOP*, vol. 788 of *LNCS*, pp. 363–378, 1994.
15. A. King. Pair-sharing over rational trees. *JLP*, 46(1-2):139–155, Nov. 2000.
16. A. King and M. Longley. Abstract matching can improve on abstract unification. Technical Report 4-95*, Computing Laboratory, Univ. of Kent, Canterbury, 1995.
17. V. Lagoon and P.J. Stuckey. Precise Pair-Sharing Analysis of Logic Programs. In *Proc. of PPDP*, 99–108, 2002.
18. A. Langen. *Static Analysis for Independent And-parallelism in Logic Programs*. PhD thesis, University of Southern California, Los Angeles, California, 1990.
19. K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM TOPLAS*, 16(3):607–648, 1994.
20. K. Muthukumar and M. V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *JLP*, 13(2&3):315–347, 1992.
21. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In *Proc. ESOP 86*, vol. 213 of LNCS, pp. 327–338, 1986.
22. E. Zaffanella. *Correctness, Precision and Efficiency in the Sharing Analysis of Real Logic Languages*. PhD thesis, School of Computing, University of Leeds, Leeds, U.K., 2001. Available at `http://www.cs.unipr.it/~zaffanella/`.

Two Variables per Linear Inequality as an Abstract Domain

Axel Simon[1], Andy King[1], and Jacob M. Howe[2]

[1] Computing Laboratory, University of Kent, Canterbury, UK
{a.m.king,a.simon}@ukc.ac.uk
[2] Department of Computing, City University, London, UK
jacob@soi.city.ac.uk

Abstract. This paper explores the spatial domain of sets of inequalities where each inequality contains at most two variables – a domain that is richer than intervals and more tractable than general polyhedra. We present a complete suite of efficient domain operations for linear systems with two variables per inequality with unrestricted coefficients. We exploit a tactic in which a system of inequalities with at most two variables per inequality is decomposed into a series of projections – one for each two dimensional plane. The decomposition enables all domain operations required for abstract interpretation to be expressed in terms of the two dimensional case. The resulting operations are efficient and include a novel planar convex hull algorithm. Empirical evidence suggests that widening can be applied effectively, ensuring tractability.

1 Introduction

The value of spatial domains such as intervals [13], affine spaces [19] and polyhedra [8] has been recognized since the early days of program analysis. One reoccurring theme in program analysis is the trade-off between precision of the domain and the tractability of the domain operations. In this regard, the polyhedral sub-domain that consists of sets of linear inequalities where each inequality contains at most two variables has recently attracted attention [26,27,33,35]. In fact, because of its tractability, this class of linear inequalities has recently been proposed for constraint logic programming [15,18]. This paper adapts this work to the requirements of program optimization and program development by equipping this domain with the operations needed for abstract interpretation. Two variable inequality domains have already proven useful in areas as diverse as program verification [29,34], model checking of timed automata [22,28], parallelization [2], locating security vulnerabilities [36], detecting memory leaks [33] and verifying program termination in logic programming [24]. Thus the applicability of the domain extends beyond logic programming [4,17] to other analysis problems in verification and program development.

The work of Miné [26] represents the state-of-the-art for program analysis with domains of inequalities restricted to two variables. He uses the so-called Octagon domain [26] where inequalities have unit coefficients of -1, 0 or +1. A

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 71–89, 2003.

difference-bound matrix (DBM) representation is employed that uses a $2d \times 2d$ matrix to encode a system of inequalities, S say, over d variables (the dimension). One key idea in this work is that of closure. Closure strengthens the inequalities of S (represented as a DBM) to obtain a new system S' (also represented as a DBM). For example, if $x+y \leq c' \in S'$, then $c' \leq c$ whenever S implies $x+y \leq c$. Thus applying closure maximally tightens each inequality, possibly introducing new inequalities. Projection, entailment and join apply closure as a preprocessing step both to preserve precision and simplify the domain operations themselves. For example, the join of two inequalities with identical coefficients, say $x-y \leq c_1$ and $x-y \leq c_2$, is simply $x-y \leq \max(c_1, c_2)$. Closure enables this simple join to be lifted point-wise to systems of inequalities. Since most domain operations require one or both of their arguments to be closed, these operations inherit the $O(d^3)$ complexity of the DBM closure operation. In this paper, we show how closure is also the key concept to tackle the two variable per inequality domain with unrestricted coefficients. Henceforth, our closure operator is referred to as completion to distinguish it from topological closure.

This paper draws together a number of strands from the verification, analysis and constraints literature to make the following novel contributions:

- We show that a polynomial completion algorithm which makes explicit all the two-dimensional projections of a system of (unrestricted) two variable inequalities enables each domain operation to be computed in polynomial time. Incredibly, such a completion operator already exists and is embedded into the satisfiability algorithm of Nelson [29].
- We explain how classic $O(m \log m)$ convex hull algorithms for sets of m planar points, such as [11], can be adapted to compute the join efficiently. The crucial point is that completion enables join to be computed point-wise on each two-dimensional projection which necessarily describes a planar object. Surprisingly little literature addresses how to efficiently compute convex hull of planar polyhedra (without the full complexity of the standard d-dimensional algorithm [6,23]) and as far as we are aware, our convex hull algorithm is unique (see [32] for a recent survey). Projection and entailment operators are also detailed.
- We also address scalability and present empirical evidence that the number of inequalities in each two-dimensional projection is small. This suggests a natural widening: limit the number of inequalities in each projection by a constant. This trivial widening obtains an $O(d^2)$ representation, like DBMs, without enforcing the requirement that coefficients are $-1, 0$ or $+1$. Note that in contrast to DBMs, our representation is dense – space is only required for those inequalities actually occurring in the system. The widening also causes completion to collapse to an $O(d^3(\log d)^2)$ operation which is competitive with the $O(d^3)$ DBM approach, taking into consideration the extra expressiveness.
- We also argue that the domain operations themselves are conceptually simple, straightforward to code and therefore more likely to be implemented correctly.

To summarize, we remove a serious limitation of the Octagon domain – that the coefficients must be unitary – without compromising tractability. Applications that employ the Octagon domain or related weaker domains [22,28,33] will therefore directly benefit from this work.

The paper is structured as follows. Section 2 presents the abstract domain. Section 3 explains how Nelson's satisfiability algorithm [29] can be adapted to complete a system. The next three sections explain how completion provides the basis for the domain operations. Section 7 presents empirical evidence for the practicality of the domain. The future and related work sections conclude.

2 Abstract Domain

To specify the domain algorithms and argue their correctness, we start the exposition by detailing some theoretical properties of polyhedral domains.

2.1 Convex Hull and Closure

An ϵ-ball around $\boldsymbol{y} \in \mathbb{R}^n$ is defined as $B_\epsilon(\boldsymbol{y}) = \{\boldsymbol{x} \in \mathbb{R}^n \mid \sum_{i=1}^{n}(x_i - y_i)^2 < \epsilon\}$. A set $S \subseteq \mathbb{R}^n$ is open if, given any $y \in S$, there exists $\epsilon > 0$ such that $B_\epsilon(y) \subseteq S$. A set $S \subseteq \mathbb{R}^n$ is closed iff $\mathbb{R}^n \setminus S$ is open. Note that if $S_i \subseteq \mathbb{R}^n$ is closed for each member of an index set $i \in I$ then $\cap\{S_i \mid i \in I\}$ is also closed. The (topological) closure of $S \in \mathbb{R}^n$ is defined $cl(S) = \cap\{S' \subseteq \mathbb{R}^n \mid S \subseteq S' \wedge S' \text{ is closed}\}$. The convex hull of $S \in \mathbb{R}^n$ is defined $conv(S) = \{\lambda\boldsymbol{x}+(1-\lambda)\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{y} \in S \wedge 0 \leq \lambda \leq 1\}$.

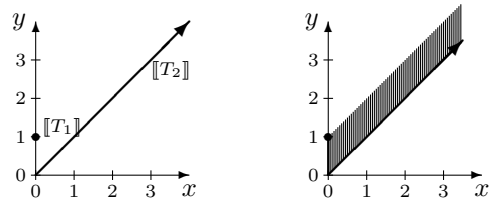
2.2 Two-Variables per Inequality Domain

Let X denote the finite set of variables $\{x_1, \ldots, x_n\}$ so that X is ordered lexicographically by $x_i \prec x_j$ iff $i < j$. Let Lin_X denote the set of (possibly rearranged) linear inequalities of the form $ax_i + bx_j \leq c$ where $a, b, c \in \mathbb{R}$. Let Two_X denote the set of all finite subsets of Lin_X. Note that although each set $T \in Two_X$ is finite, Two_X is not finite. Syntactic sugar of the form $x \leq y$ is used instead of $(+1)x + (-1)y \leq 0 \in Lin_X$ as well as $by + ax \leq c$ instead of $ax + by \leq c$.

Definition 1. The mapping $[\![.]\!] : Lin_X \to \mathbb{R}^n$ is defined: $[\![ax_i + bx_j \leq c]\!] = \{\langle y_1, \ldots, y_n\rangle \in \mathbb{R}^n \mid ay_i + by_j \leq c\}$ and the mapping $[\![.]\!] : Two_X \to \mathbb{R}^n$ is defined $[\![T]\!] = \cap\{[\![t]\!] \mid t \in T\}$.

For brevity, let $t^=$ represent the boundary of a given half-space, that is, define $t^= = \{ax_i + bx_j \leq c, -ax_i - bx_j \leq -c\}$ when $t \equiv ax_i + bx_j \leq c$. Two_X is ordered by entailment, that is, $T_1 \models T_2$ iff $[\![T_1]\!] \subseteq [\![T_2]\!]$. Equivalence on Two_X is defined $T_1 \equiv T_2$ iff $T_1 \models T_2$ and $T_2 \models T_1$. Moreover $T \models t$ iff $T \models \{t\}$ and $t_1 \equiv t_2$ iff $\{t_1\} \equiv \{t_2\}$. Let $Two_X^{\equiv\equiv} = Two_X/\equiv$. $Two_X^{\equiv\equiv}$ inherits entailment $\models$ from Two_X. In fact $\langle Two_X^{\equiv\equiv}, \models, \sqcap, \sqcup\rangle$ is a lattice (rather than a complete lattice) with $[T_1]_\equiv \sqcap [T_2]_\equiv = [T_1 \cup T_2]_\equiv$ and $[T_1]_\equiv \sqcup [T_2]_\equiv = [T]_\equiv$ where $[\![T]\!] = cl(conv([\![T_1]\!] \cup [\![T_2]\!]))$. Note that in general $conv([\![T_1]\!] \cup [\![T_2]\!])$ is not closed and therefore cannot be described by a system of non-strict linear inequalities as is illustrated below.

Example 1. Let $X = \{x, y\}$, $T_1 = \{x \leq 0, -x \leq 0, y \leq 1, -y \leq -1\}$ and $T_2 = \{-x \leq 0, x - y \leq 0, y - x \leq 0\}$ so that $[\![T_1]\!] = \{\langle 0, 1\rangle\}$ and $[\![T_2]\!] = \{\langle x, y\rangle \mid 0 \leq x \wedge x = y\}$. Then $conv([\![T_1]\!] \cup [\![T_2]\!])$ includes the point $\langle 0, 1\rangle$ but not the ray $\{\langle x, y\rangle \mid 0 \leq x \wedge x + 1 = y\}$ and hence is not closed.



The domain Two_X is a generic abstract domain that is not limited to a specific application. No concretization map is defined in this paper since such a map is specific to an application. However, if an application used the concretization map $\gamma(T) = [\![T]\!]$ then no abstraction map $\alpha : \wp(\mathbb{R}^n) \to Two_X$ would exist since there is no best abstraction e.g. for the set $\{\langle x, y\rangle \mid x^2 + y^2 \leq 1\}$. The problem stems from the fact that Two_X can contain an arbitrarily large number of inequalities. This contrasts with the Octagon domain where each planar object will be described by at most eight inequalities.

We will augment $\langle Two_X^{\equiv\equiv}, \models, \sqcap, \sqcup\rangle$ with projection $\exists$ and widening to accommodate the needs of abstract interpretation.

Definition 2. Projection operator $\exists_{x_i} : Two_X^{\equiv\equiv} \to Two_X^{\equiv\equiv}$ is defined $\exists_{x_i}([T_1]_{\equiv}) = [T_2]_{\equiv}$ where $[\![T_2]\!] = \{\langle y_1, \ldots, y_{i-1}, y, y_{i+1}, \ldots, y_n\rangle \mid y \in \mathbb{R} \wedge \langle y_1, \ldots, y_n\rangle \in [\![T_1]\!]\}$.

Projection can be calculated using Fourier-Motzkin variable elimination and from this it follows that $T_2 \in Two_X$ if $T_1 \in Two_X$.

2.3 Complete Form for the Two-Variables per Inequality Domain

The complete form for the two-variables per inequality domain is defined in terms of those variables that occur in a set of inequalities.

Definition 3. The mapping $var : Lin_X \to \wp(X)$ is defined:

$$var(ax + by \leq c) = \begin{cases} \emptyset & \text{if } a = b = 0 \\ \{y\} & \text{if } a = 0 \\ \{x\} & \text{if } b = 0 \\ \{x, y\} & \text{otherwise} \end{cases}$$

The mapping var captures those variables with non-zero coefficients. Observe that $var(t_1) = var(t_2)$ if $t_1 \equiv t_2$. In contrast, note that $var(0u + 0v \leq 1) = \emptyset = var(0x + 0y \leq -1)$. If $T \in Two_X$ then let $var(T) = \cup\{var(t) \mid t \in T\}$.

Definition 4. Let $Y \subseteq X$. The restriction operator π_Y is defined:

$$\pi_Y(T) = \{t \in T \mid var(t) \subseteq Y\}$$

Definition 5. The set of *complete* finite subsets of Lin_X is defined:

$$Two'_X = \{T \in Two_X \mid \forall t \in Lin_X \,.\, T \models t \;\Rightarrow\; \pi_{var(t)}(T) \models t\}$$

Proposition 1. Suppose $T \in Two_X$. Then there exists $T' \in Two'_X$ such that $T \subseteq T'$ and $T \equiv T'$.

Proof. Define $[T_{x,y}]_{\equiv} = \exists_{X \setminus \{x,y\}}([T]_{\equiv})$ for all $x, y \in X$ and $T' = T \cup \bigcup_{x,y \in X} T_{x,y}$. Since each $T_{x,y}$ is finite, T' is finite, hence $T' \in Two'_X$. By the definition of $\exists$, $T \models T_{x,y}$, hence $T \cup T_{x,y} \equiv T$ for all $x, y \in X$, thus $T \equiv T'$. Moreover $T \subseteq T'$. ∎

Corollary 1. $Two^{\equiv}_X = Two'_X/\equiv$.

2.4 Ordering the Two-Variables per Inequality Domain

Let $Y = \{x, y\} \subseteq X$ such that $x \prec y$ and consider $T = \{t_1, \ldots, t_n\} \in Two_Y$. Each t_i defines a half-space in the Y plane and therefore T can be ordered by the orientation of the half-spaces as follows:

Definition 6. The (partial) mapping $\theta : Lin_Y \to [0, 2\pi)$ is defined such that $\theta(ax + by \leq c) = \psi$ where $\cos(\psi) = -b/\sqrt{a^2 + b^2}$ and $\sin(\psi) = a/\sqrt{a^2 + b^2}$.

The mapping θ actually returns the anti-clockwise angle which the half-space $\{\langle x, y\rangle \mid y \geq 0\}$ has to be turned through to coincide with $\{\langle x, y\rangle \mid ax + by \leq 0\}$.

2.5 Entailment between Three Inequalities

This section demonstrates how entailment checks of the form $\{t_1\} \models t$ and $\{t_1, t_2\} \models t$ can be computed in constant time. The following proposition explains how this check reduces to applying the Cramer rule for the three inequality case and simple scaling for the two inequality case.

Proposition 2. Let $t_i \equiv a_i x + b_i y \leq c_i$ for $i = 1, 2$ and $t \equiv ax + by \leq c$. Then

$$\{t_1\} \models t \iff \begin{cases} \mathit{false} & \text{if } a_1 b - a b_1 \neq 0 \\ \mathit{false} & \text{else if } a_1 a < 0 \vee b_1 b < 0 \\ (a/a_1)c_1 \leq c & \text{else if } a_1 \neq 0 \\ (b/b_1)c_1 \leq c & \text{else if } b_1 \neq 0 \\ c_1 < 0 \vee (c \geq 0 \wedge a = 0 \wedge b = 0) & \text{otherwise} \end{cases}$$

$$\{t_1, t_2\} \models t \iff \begin{cases} \{t_1\} \models t \vee \{t_2\} \models t & \text{if } d = a_1 b_2 - a_2 b_1 = 0 \\ \mathit{false} & \text{else if } \lambda_1 = (a b_2 - a_2 b)/d < 0 \\ \mathit{false} & \text{else if } \lambda_2 = (a_1 b - a b_1)/d < 0 \\ \lambda_1 c_1 + \lambda_2 c_2 \leq c & \text{otherwise.} \end{cases}$$

If the inequalities t_1 and t differ in slope, then the determinant of their coefficients is non-zero and they cannot entail each other. Suppose now that the determinant is zero. Observe that the two inequalities have opposing feasible spaces whenever a_1 and a or b_1 and b have opposite signs. In this case t_1 cannot entail t. If t_1 has a non-zero coefficient, then entailment reduces to a simple comparison between the constants of the inequalities, suitably scaled. The fifth case matches the pathological situation of tautologous and unsatisfiable inequalities.

The entailment between three inequalities reduces to the former case if t_1 and t_2 have equal slope (the determinant is zero). Otherwise an inequality is constructed which has the same slope as t and which passes through the intersection point $[\![t_1^=]\!] \cap [\![t_2^=]\!]$ using the Cramer rule. Again, a comparison of the constants determines the entailment relationship. If either λ_1 or λ_2 is negative, the feasible space of the combination of t_1 and t_2 will oppose that of t, thus $\{t_1, t_2\}$ cannot entail t.

3 Completion: A Variant of Nelson's Satisfiability Algorithm

In this section we show how to complete a system of inequalities. This operation corresponds to the closure operation of Miné. We follow the approach that Nelson used for checking satisfiability [29]. One key concept in his algorithm is the notion of a filter that is formalized below.

Definition 7. Let $Y = \{x, y\} \subseteq X$. The mapping $\mathit{filter}_Y : \mathit{Two}_Y \to \mathit{Two}_Y$ is defined such that:

1. $\mathit{filter}_Y(T) \subseteq T$
2. $\mathit{filter}_Y(T) \equiv T$
3. for all $T' \subseteq T$ and $T' \equiv T$, $|\mathit{filter}_Y(T)| \leq |T'|$.

The role of filter_Y is to remove redundant elements from a set of inequalities over the variables Y. If the inequalities are ordered by angle, redundancy removal can be done surprisingly efficiently as illustrated in Fig. 1. The function *filter* returns a single contradictory inequality if the completed system S is unsatisfiable, and otherwise removes tautologies before sorting the inequalities. The loop then iterates over the inequalities once in an anti-clockwise fashion. It terminates when no more redundant inequalities can be found, that is, when (1) the whole set of inequalities has been traversed once (flag f is true) and (2) the inequalities with the largest and smallest angle are both non-redundant. Since the entailment check between three inequalities can be performed in constant time, the algorithm is linear. Note that different subsets of the input can be minimal. This occurs, for example, when the system is unsatisfiable. Then filter_Y returns one of these subsets.

The map filter_Y lifts to arbitrary systems of two-variable inequalities as follows:

Definition 8. The mapping $\mathit{filter} : \mathit{Two}_X \to \mathit{Two}_X$ is defined:

```
function filter_{x,y}(S ∈ Two_X) begin
    if ∃s ∈ S . s ≡ 0x + 0y ≤ −1 then return {s};
    T := {s ∈ S | s ≢ 0x + 0y ≤ 1};
    let T = {t_1, ..., t_m} such that θ(t_1) ≤ θ(t_2) ≤ ... ≤ θ(t_m);
    f := false;
    loop
        let {t_c, t_n, ..., t_l} = T; if |T| > 1 ∧ {t_n, t_l} ⊨ t_c then T := {t_n, ..., t_l}; else begin
            if θ(t_c) ≤ θ(t_l) ∧ f then return T;
            if θ(t_c) ≤ θ(t_l) then f := true;
            T := {t_l, t_c, t_n, ...};
        end;
    end;
end
```

Fig. 1. Algorithm for redundancy removal

$$\mathit{filter}(T) = \bigcup\{\mathit{filter}_Y(\pi_Y(T)) \mid Y \subseteq X \wedge |Y| = 2\}$$

The second key idea of Nelson is the *result* map that makes explicit those inequalities that are indirectly expressed by the system. The basic step is to generate all possible combinations of pairs of inequalities by eliminating their common variable.

Definition 9. The resultants map $\mathit{result} : \mathit{Two}_X \to \mathit{Two}_X$ is defined by:

$$\mathit{result}(T) = \left\{ aez - dby \leq af - dc \;\middle|\; \begin{array}{l} t_1, t_2 \in T \quad \wedge \\ t_1 \equiv ax + by \leq c \wedge \\ t_2 \equiv dx + ez \leq f \wedge \\ a > 0 \wedge d < 0 \end{array} \right\}$$

The following example demonstrates how *result* works on a chain of dependent variables:

Example 2. Let $T_0 = \{x_0 \leq x_1, x_1 \leq x_2, x_2 \leq x_3, x_3 \leq x_4\}$. We calculate $T_1 = \mathit{result}(T_0)$ and $T_2 = \mathit{result}(T_0 \cup T_1)$.

$$\mathit{result}(T_0) = \{x_0 \leq x_2, x_1 \leq x_3, x_2 \leq x_4\}$$
$$\mathit{result}(T_0 \cup T_1) = T_1 \cup \{x_0 \leq x_3, x_0 \leq x_4, x_1 \leq x_4\}$$

Note that $T_3 = \bigcup_{i=0}^{2} T_i$ is a fixpoint in $T_3 = \mathit{result}(T_3)$.

An important property of $T \cup \mathit{result}(T)$ is the way it halves the number of variables required to entail a given inequality t. Specifically, suppose $T \models t$. Then there exists $T' \subseteq T \cup \mathit{result}(T)$ such that $T' \models t$ and T' contains no more than half the variables of T. Lemma 1 formalizes this and is basically a reformulation of Lemma 1b of [29].

Lemma 1. Let $T \in \mathit{Two}_X$ and $t \in \mathit{Lin}_X$ such that $T \models t$. Then there exists $Y \subseteq X$ such that $|Y| \leq \lfloor |\mathit{var}(T)|/2 \rfloor + 1$ and $\pi_Y(T \cup \mathit{result}(T)) \models t$.

Lemma 1 suggests the following iterative algorithm for calculating completion that takes (approximately) $\log_2(|var(T)|)$ steps. Theorem 1 asserts its correctness.

Definition 10. The mapping $complete : Two_X \to Two_X$ is defined:

$$complete(T_0) = T_{\lceil \log_2(|var(T_0)|-1)\rceil} \text{ where } T_{i+1} = filter(T_i \cup result(T_i))$$

Theorem 1. $complete(T) \equiv T$ and $complete(T) \in Two'_X$ for all $T \in Two_X$.

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ where $f(n) = \lfloor n/2 \rfloor + 1$. The following table details $m \in \mathbb{N}$ for which $f^m(n) \leq 2$. Observe that $f^{\lceil \log_2(n-1)\rceil}(n) \leq 2$.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
m	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	...

Observe that $T \equiv T \cup result(T) \equiv filter(T \cup result(T))$ and by induction $T \equiv complete(T)$. Let $t \in Lin_X$ such that $complete(T) \models t$. Then $T \models t$. Let $T_0 = T$ and $T_{i+1} = filter(T_i \cup result(T_i))$. By induction and by Lemma 1, there exists $Y_i \subseteq var(T)$ such that $\pi_{Y_i}(T_i) \models t$ and $|Y_i| \leq f^i(|var(T)|)$. Therefore $|Y_{\lceil \log_2(|var(T)|-1)\rceil}| \leq 2$, hence $\pi_{var(t)}(complete(T)) \models t$ as required. ∎

Note that applying an additional completion step makes explicit all inequalities over one variable. Furthermore, applying it once more creates tautologous and contradictory inequalities. Applying these two additional completion steps enables filter to detect unsatisfability without employing any extra machinery.

Example 3. To illustrate how unsatisfiability is detected consider the system $T_0 = \{-x + y \leq -1, -2x - 3y \leq -6, 4x - 2y \leq -4\}$. The system is complete but two more completion steps are necessary to detect unsatisfiability. The calculation $T_1 = filter(T_0 \cup result(T_0)) = T_0 \cup \{-y \leq -2, -5x \leq -9, x \leq -3\}$ makes all inequalities over one variable explicit. Unsatisfiability becomes explicit when calculating $0 \leq -24 \in result(T_1)$. Finally $filter(result(T_1)) = \{0 \leq -24\}$ collapses the system to a single unsatisfiable constraint.

3.1 Complexity of the *complete* Operation

Nelson shows that his satisfiability algorithm is polynomial in the number of input inequalities [29]. For comparison with the DBM approach, consider the complexity of $filter(T_i \cup result(T_i))$ where $d = |var(T_i)|$ and $k = \max\{|\pi_Y(T_i)| \mid i \in [0, \lceil \log_2(|var(T)| - 1)\rceil] \wedge Y = \{x, y\} \subseteq var(T_i)\}$. Since each T_i may have $d(d-1)/2$ restrictions, a linear pass over $O(kd^2)$ inequalities is sufficient to partition the set of inequalities into d sets, one for each variable. Each set has at most $O(kd)$ elements, so calculating the resultants for each set is $O(k^2d^2)$, hence calculating all the resultants is $O(k^2d^3)$. The complexity of applying the linear *filter* is in $O(kd^2 + k^2d^3) = O(k^2d^3)$ which with sorting requires $O(k^2d^3 \log(k^2d^3)) = O(k^2d^3(\log(k)+\log(d)))$ time. The *complete* operation runs *result* $O(\log d)$ times which leads to an overall running time of $O(k^2d^3 \log(d)(\log(k) + \log(d)))$. In Section 7 we show that k is typically small and therefore can be limited by a constant with hardly any loss of expressiveness. This collapses the bound to $O(d^3(\log(d))^2)$ which is only slightly worse than the $O(d^3)$ closure of Miné [26].

```
function extreme(T ∈ Two_{x,y}) begin
    let T = {t_0, ..., t_{n-1}} such that θ(t_0) < θ(t_1) < ... < θ(t_{n-1});
    V := R := ∅;
    for i ∈ [0, n − 1] do let t_i ≡ ax + by ≤ c in begin
        // are the intersection points of this inequality degenerated?
        d_pre := (θ(t_i) − θ(t_{i−1 mod n})) mod 2π ≥ π ∨ n = 1;
        d_post := (θ(t_{i+1 mod n}) − θ(t_i)) mod 2π ≥ π ∨ n = 1;
        if d_pre then R := R ∪ {⟨b/√(a² + b²), −a/√(a² + b²)⟩};
        if d_post then R := R ∪ {⟨−b/√(a² + b²), a/√(a² + b²)⟩};
                  else V := V ∪ {v} where v ∈ [[t_i^=]] ∩ [[t^=_{(i+1) mod n}]];
        if d_pre ∧ d_post then begin
            if n = 1 then R := R ∪ {⟨−a/√(a² + b²), −b/√(a² + b²)⟩};
            V := V ∪ {v} where v ∈ [[t_i^=]]
        end
    end
    return ⟨V, R⟩
end
```

Fig. 2. Calculating the points and rays of a planar polyhedron

3.2 Satisfiability and the *complete* Operation

Nelson [29] originally devised this completion operation in order to construct a polynomial test for satisfiability. The following proposition explains how non-satisfiability can be observed after (and even during) the completion calculation. Specifically, the proposition asserts that non-satisfiability always manifests itself in the existence of at least one contradictory inequality.

Proposition 3. Let $T' \in Two'_X$. Then $[\![T']\!] = \emptyset$ iff $[\![\pi_\emptyset(T')]\!] = \emptyset$.

Proof. Let $T' \in Two'_X$. Suppose $[\![T']\!] = \emptyset$. Then $T' \models 0x + 0y \leq -1$. Since $var(0x+0y \leq -1) = \emptyset$, hence $\pi_\emptyset(T') \models 0x+0y \leq -1$ and therefore $[\![\pi_\emptyset(T')]\!] = \emptyset$. Since $\pi_\emptyset(T') \subseteq T'$ the converse follows.

4 Join: Planar Convex Hull on Each Projection

Computing the join corresponds to calculating the convex hull for polyhedra which is surprisingly subtle. The standard approach for arbitrary d-dimensional polyhedra involves applying the Chernikova [6] algorithm (or a variant [23]) to construct a vertices and rays representation which is potentially exponential [20]. By way of contrast, we show that convex hull for systems of two variables per inequality can be computed by a short polynomial algorithm.

The construction starts by reformulating the convex hull piece-wise in terms of each of its planar projections. Proposition 4 shows that this operation results in a complete system whenever its inputs are complete; equivalence with the fully dimensional convex hull operation is stated in Proposition 5.

Definition 11. The piece-wise convex hull $\curlyvee : {Two_X}^2 \to Two_X$ is defined $T_1 \curlyvee T_2 = \cup\{T_{x,y} \in Two_{\{x,y\}} \mid x, y \in X\}$ where $[\![T_{x,y}]\!] = cl(conv([\![\pi_{\{x,y\}}(T_1)]\!] \cup [\![\pi_{\{x,y\}}(T_2)]\!]))$.

Proposition 4. $T'_1 \curlyvee T'_2 \in Two'_X$ if $T'_1, T'_2 \in Two'_X$.

Proof. Let $t \in Lin_X$ such that $T'_1 \curlyvee T'_2 \models t$. Let $x, y \in X$ and let $[\![T_{x,y}]\!] = cl(conv([\![\pi_{\{x,y\}}(T'_1)]\!] \cup [\![\pi_{\{x,y\}}(T'_2)]\!]))$. Observe $\pi_{\{x,y\}}(T'_1) \models T_{x,y}$, therefore $T'_1 \models T'_1 \curlyvee T'_2$. Likewise $T'_2 \models T'_1 \curlyvee T'_2$, hence it follows that $T'_1 \models t$ and $T'_2 \models t$. Since $T'_1, T'_2 \in Two'_X$, $\pi_{var(t)}(T'_1) \models t$ and $\pi_{var(t)}(T'_2) \models t$, thus $[\![\pi_{var(t)}(T'_1)]\!] \subseteq [\![t]\!]$ and $[\![\pi_{var(t)}(T'_2)]\!] \subseteq [\![t]\!]$, hence $[\![\pi_{var(t)}(T'_2)]\!] \cup [\![\pi_{var(t)}(T'_2)]\!] \subseteq [\![t]\!]$. Therefore $[\![\pi_{var(t)}(T'_1 \curlyvee T'_2)]\!] = cl(conv([\![\pi_{var(t)}(T'_1)]\!] \cup [\![\pi_{var(t)}(T'_2)]\!])) \subseteq cl(conv([\![t]\!])) = [\![t]\!]$. Therefore $\pi_{var(t)}(T'_1 \curlyvee T'_2) \models t$ as required. ∎

Proposition 5. $[\![T'_1 \curlyvee T'_2]\!] = cl(conv([\![T'_1]\!] \cup [\![T'_2]\!]))$ if $T'_1, T'_2 \in Two'_X$.

Proof. Since $T'_1 \models T'_1 \curlyvee T'_2$ and $T'_2 \models T'_1 \curlyvee T'_2$, it follows that $cl(conv([\![T'_1]\!] \cup [\![T'_2]\!])) \subseteq [\![T'_1 \curlyvee T'_2]\!]$. Suppose there exists $\langle c_1, \ldots, c_n\rangle \in [\![T'_1 \curlyvee T'_2]\!]$ such that $\langle c_1, \ldots, c_n\rangle \notin [\![T']\!]$ where $[\![T']\!] = cl(conv([\![T'_1]\!] \cup [\![T'_2]\!]))$. Thus $\bigcup_{i=1}^{n} \{x_i \le c_i, c_i \le x_i\} \not\models T'$, hence there exists $ax_j + bx_k \le c \equiv t \in T'$ with $\bigcup_{i=1}^{n} \{x_i \le c_i, c_i \le x_i\} \not\models ax_j + bx_k \le c$. But $T'_1 \models T' \models t$ and $T'_2 \models T' \models t$. Since $T'_1 \in Two'_X$ and $T'_2 \in Two'_X$, it follows that $\pi_{\{x_j,x_k\}}(T'_1) \models t$ and $\pi_{\{x_j,x_k\}}(T'_2) \models t$. Hence $T'_1 \curlyvee T'_2 \models t$, thus $\bigcup_{i=1}^{n} \{x_i \le c_i, c_i \le x_i\} \models T'_1 \curlyvee T'_2$ but $\langle c_1, \ldots, c_n\rangle \notin [\![T'_1 \curlyvee T'_2]\!]$ which is a contradiction. ∎

Calculating the convex hull for a set of points in the plane has been studied extensively [32]. The convex hull of polytopes can be reduced to this problem by converting the polytopes into their vertex representation, calculating the convex hull of all vertices and converting back into the inequality representation. Although the generalization to planar polyhedra follows this three-step process, it is much more subtle and little literature has been written on this fundamental problem. Given a set of non-redundant inequalities, ordered by their orientation θ, the auxiliary function *extreme* in Figure 2 calculates a set of vertices and rays that represent the polyhedron. Rays are created when the angle between the current inequality t_i and the previous inequality is greater or equal to π (d_{pre} is true) and similarly for the next inequality (d_{post} is true). If both flags are true, we create an arbitrary point on the boundary of the halfspace of t_i to fix its representing rays in space. A pathological case arises when the polyhedron consists of a single halfspace ($n = 1$). In this case a third ray is created to indicate on which side the feasible space lies. Note that the maximum number of rays for each polyhedron is four, which occurs when T defines two facing halfspaces.

The main function *join* in Figure 3 uses *extreme* to compute the vertices and rays of each input polyhedron and catches the simple case of when both polyhedra consist of the same single point. Otherwise we calculate a square whose sides have length $2m$ which is centered on the origin and that contains all vertices in $V_1 \cup V_2$. For each ray $r \in R$ we translate each vertex in $V_1 \cup V_2$ in the direction

```
function join(T1 ∈ Two_X, T2 ∈ Two_X) begin
    if ∃t ∈ T1 . t ≡ 0x + 0y ≤ −1 then return T2;
    if ∃t ∈ T2 . t ≡ 0x + 0y ≤ −1 then return T1;
    // note: each Ti is non-redundant
    ⟨V1, R1⟩ := extreme(T1);
    ⟨V2, R2⟩ := extreme(T2);
    V := V1 ∪ V2; R := R1 ∪ R2; // Note: |R| ≤ 8
    if V = {⟨x1, y1⟩} ∧ R = ∅ then
        return {x ≤ x1, −x ≤ −x1, y ≤ y1, −y ≤ −y1};
    m := max{|x|, |y| | ⟨x, y⟩ ∈ V} + 1;
    //add a point along the ray, goes through x, y and is outside the box
    for ⟨x, y, a, b⟩ ∈ V1 ∪ V2 × R do V := V ∪ {⟨x + 2√2ma, y + 2√2mb⟩};
    {v0, ..., vn−1} := graham(V) such that v0, ..., vn−1 are ordered anti-clockwise
                                  and points on the boundary are not removed
    Tres := ∅; t_last := connect(vn−1, v0);
    for i ∈ [0, n − 1] do begin
        let ⟨x1, y1⟩ = vi, ⟨x2, y2⟩ = v(i+1) mod n, t = connect(vi, v(i+1) mod n)
        if (|x1|<m ∧ |y1|<m) ∨ (|x2|<m ∧ |y2|<m) ∧ θ(t) ≠ θ(t_last) then begin
            if (θ(t) − θ(t_last)) mod 2π = π ∧ |x1|<m ∧ |y1|<m then
                if y1 = y2 then Tres := Tres ∪ {sgn(x1 − x2)x ≤ sgn(x1 − x2)x1}
                           else Tres := Tres ∪ {sgn(y1 − y2)y ≤ sgn(y1 − y2)y1}
            Tres := Tres ∪ {t}; t_last := t;
        end
    end
    return Tres
end

function connect(⟨x1, y1⟩, ⟨x2, y2⟩)
    return (y2 − y1)x + (x1 − x2)y ≤ (y2 − y1)x1 + (x1 − x2)y1
```

Fig. 3. Convex hull algorithm for planar polyhedra

of the ray r. Note that the normalization of the rays and the translation by $2\sqrt{2}m$ ensures that the translated vertices are outside the square. We now apply the Graham convex hull algorithm [11], modified so that it removes all (strictly) interior vertices but retains points which lie on the boundary of the hull. What follows is a round-trip around this hull, translating two adjacent vertices into an inequality by calling *connect* if the following conditions are met: the inequality must have a different slope than the previously generated inequality and at least one of the two vertices must lie within the box. The two innermost if-statements deal with the pathological case of when V contains only colinear points and additional inequalities are needed to restrict the two opposing inequalities so that an (unbounded) line is not inadvertently generated.

The running time of this algorithm is dominated by the call to the convex hull algorithm of Graham [11] which takes $O(n \log n)$ time where $n = |V||R|$. However, $|R|$ is at most eight (and usually between zero and four). Since $O(|V|) = O(|T|)$ it follows that the overall running time is $O((|T_1| + |T_2|) \log(|T_1| + |T_2|))$.

5 Projection

Projection returns the most precise system which does not depend on a given variable. We provide a constructive definition of projection for (complete) systems. Proposition 6 states that this coincides with the spatial definition of projection. Furthermore we prove that this operation preserves completion.

Definition 12. The operator $\exists_x : \mathit{Two}_X \to \mathit{Two}_{X\setminus\{x\}}$ is defined $\exists_x(T) = \cup\{\pi_Y(T) \mid Y = \{y, z\} \subseteq X \setminus \{x\}\}$.

Proposition 6. $\exists_x([T']_\equiv) = [\exists_x(T')]_\equiv$ and $\exists_x(T') \in \mathit{Two}'_X$ for all $T' \in \mathit{Two}'_X$.

Proof. By Fourier-Motzkin $\exists_x([T']_\equiv) = [T]_\equiv$ where $T = \{t \in T' \cup \mathit{result}(T') \mid x \notin \mathit{var}(t)\}$. Observe that $T \models \exists_x(T')$. Now suppose $r \in T' \cup \mathit{result}(T')$ such that $x \notin \mathit{var}(r)$. Then $T' \models r$, hence $\pi_{\mathit{var}(r)}(T') \models r$ and therefore $\exists_x(T') \models r$, and thus $\exists_x(T') \models T$, hence $\exists_x(T') \equiv T$ as required.

Now let $t \in \mathit{Lin}_X$ such that $\exists_x(T') \models t$. Moreover $T' \models \exists_x(T') \models t$, hence $\pi_{\mathit{var}(t)}(T') \models t$. Since $x \notin \mathit{var}(t)$, $\pi_{\mathit{var}(t)}(\exists_x(T')) \models t$ as required. ∎

Consider a complete system that includes $y - x \leq 0$ and $x - z \leq 0$. Projecting out x will preserve the inequality $y - z \leq 0$ which completion has made explicit.

6 Entailment

Entailment checking between systems of inequalities can be reduced to checking entailment on their two dimensional projections. Moreover, entailment checking for a planar polyhedron can be further reduced to checking entailment between three single inequalities. We start by detailing the entailment relationship between systems of inequalities and their two dimensional projections.

Proposition 7. Let $T' \in \mathit{Two}'_X$ and $T \in \mathit{Two}_X$. Then $T' \models T$ iff $\pi_Y(T') \models \pi_Y(T)$ for all $Y = \{x, y\} \subseteq X$.

Proof. Suppose $T' \models T$. Let $t \in \pi_Y(T)$. Then $T' \models T \models t$. Hence $\pi_{\mathit{var}(t)}(T') \models t$. Since $\mathit{var}(t) \subseteq Y$, $\pi_Y(T') \models t$ and therefore $\pi_Y(T') \models \pi_Y(T)$.

Now suppose $\pi_Y(T') \models \pi_Y(T)$ for all $Y = \{x, y\} \subseteq X$. Let $t \in T$. Then $t \in \pi_{\mathit{var}(t)}(T)$, hence $T' \models \pi_{\mathit{var}(t)}(T') \models \pi_{\mathit{var}(t)}(T) \models t$. ∎

Note that the proposition does not require both systems of inequalities to be complete. Due to Proposition 7 it suffices to check that entailment holds for all planar projections. Therefore consider checking entailment between two non-redundant planar systems $T_1, T_2 \in \mathit{Two}_{\{x,y\}}$. To test $T_1 \models T_2$ it is sufficient to show that $T_1 \models t$ for all $t \in T_2$. This reduces to finding $t_i, t_{i+1} \in T_1$ such that $\theta(t_i) \leq \theta(t) < \theta(t_{i+1})$ (modulo 2π). If any of the tests $\{t_i, t_{i+1}\} \models t$ fail, *false* can be returned immediately. If the inequalities are ordered by angle, planar entailment checking is linear time as shown in Fig. 4.

```
function entails(T1 ∈ Two'X, T2 ∈ TwoX) begin
    if ∃t ∈ T1 . t ≡ 0x + 0y ≤ −1 then return true;
    if ∃t ∈ T2 . t ≡ 0x + 0y ≤ −1 then return false;
    let {t1, ..., tn} = T1 such that θ(t1) ≤ θ(t2) ≤ ... ≤ θ(tn);
    let {t'1, ..., t'm} = T2 such that θ(t'1) ≤ θ(t'2) ≤ ... ≤ θ(t'm);
    u := 1; l := n;
    for i ∈ [1, m] do begin
        while θ(tu) < θ(t'i) ∧ u ≤ n do begin
            l := u;
            u := u + 1;
        end
        if {tl, t(u mod n)} ⊭ t'i then return false;
    end;
    return true;
end;
```

Fig. 4. Algorithm for checking entailment of planar polyhedra

7 Widening

For domains that do not satisfy the ascending chain property, widening is necessary to enforce termination of fixpoint calculations [7] (for example in loops). Widening can also be used to improve space and time behavior. In the following sections we elaborate on both.

7.1 Widening for Termination

Any widening [7,8] for polyhedra can be applied to planar polyhedra and then lifted to systems of two variables per inequality. Since the domain is structured in terms of projections, one tactic for delaying widening, and thereby improving precision, is to only apply widening when the number of projections has stabilized and the dimension of each of the projections is also stable. One subtlety is that applying completion after widening can compromise termination by reintroducing inequalities that were removed during widening.

7.2 Widening for Tractability

To assess the tractability of the domain, we implemented a naïve completion operation and measured the growth both in the number of projections and inequalities. Our test data is obtained by generating random planar polytopes over different pairs of variables. Each polytope was constructed by computing the convex hull of a random set of points distributed across a square in $\mathbb{R}^2$. We set up three different scenarios called varying, constant and sparse. In the varying scenario, we created polytopes which had between 3 and 13 inequalities each until we reached 147 inequalities in total. To make the results comparable, we then applied completion to those systems which had exactly 100 non-redundant

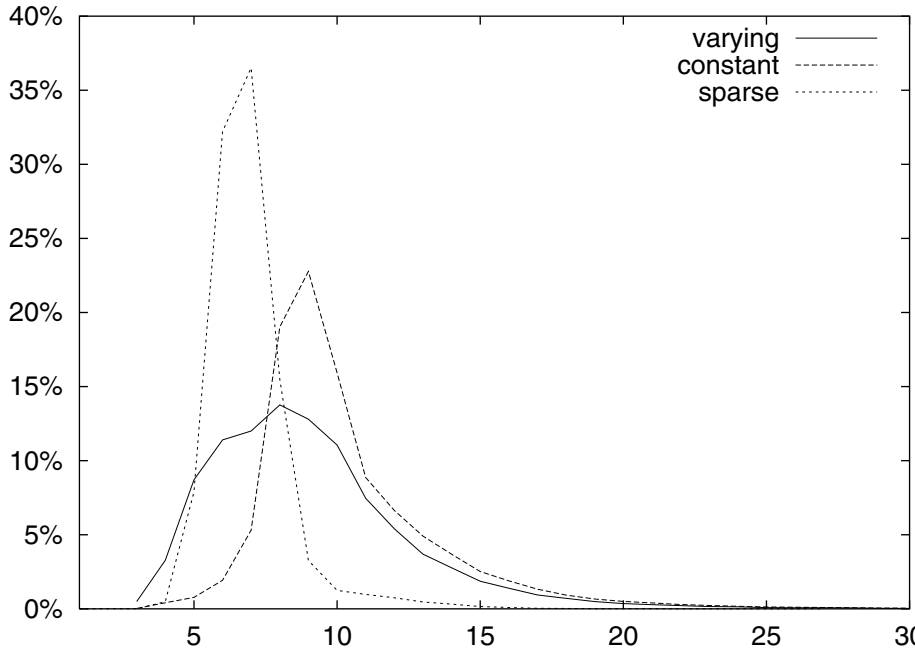


Fig. 5. The number of inequalities seems to be restricted in practice

inequalities. Redundancies can occur in the original system since two polytopes may share a common variable and a bound on this variable may propagate from one sub-system to the other, rendering inequalities superfluous. The constant scenario creates 10 inequalities for each pair of variables. Since fewer non-empty projections were initially generated (on average 143/10), the growth in the number of projections is larger – on average it increased to 32 projections. The last case, sparse, corresponds to a system where inequalities are weakly coupled, that is, few inequalities share variables. As expected the number of extra projections generated by completion is marginal. The results are summarized in Figure 6. Since randomly generated data offers no particular advantage to our completion algorithm over real data, it appears the completion will remain tractable in practice. In particular, the worst case quadratic growth in the number of projections is unlikely to arise.

An interesting observation is that the number of inequalities is not proportional to the number of points n over which the convex hull is calculated. This squares with probabilistic theory [5,31]. Specifically, the convex hull of a set of n points randomly distributed over a square is expected to have $O(\log n)$ extreme points [5], while a random set of n points restricted to a circle is expected to have $O(n^{\frac{1}{3}})$ extreme points [31]. In our experiments, less than 1% of all projections had more than 30 inequalities (see Fig. 5 for the distribution). This suggests that pruning the number of inequalities down to a constant bound will have little overall effect on precision, yet obtains an attractive $O(d^3(\log d)^2)$ perfor-

scenario	*varying*	*constant*	*sparse*
dimension	10	10	100
inequalities generated	147	143	139
inequalities per polyhedron	3–13	10	10
after redundancy removal			
remaining inequalities	100	100	100
avg. no of ineq. per polyhedron	5.3	7.0	7.1
after completion			
avg. resultant inequalities	210	189	106
increase in no of projections	56%	123%	9%
projections > 30 inequalities	0.22%	0.18%	0.00%

Fig. 6. The impact of calculating completion

mance guarantee. One way to systematically drop inequalities is to remove those that contribute least to the shape, that is, remove the inequality that contributes the shortest edge to the polyhedron.

8 Future Work

Using union-find an arbitrary $T \in Two_X$ can be partitioned in near-linear time into a system $\{T_1, \ldots, T_p\}$ such that $var(T_i) \cap var(T_j) = \emptyset$ whenever $i \neq j$. This decomposition enables the complexity of completion to be reduced to $O(d^3(\log d)^2)$ where $d = \max\{|var(T_1)|, \ldots, |var(T_p)|\}$. This tactic, which is applicable to any polyhedral domain, will be useful if the coupling between variables is low.

The completion of a system T is currently computed iteratively in approximately $\log_2(|var(T)|)$ steps. The completion operation could benefit from applying a strategy such as semi-naïve iteration [3] that would factor out some of the repeated work.

9 Related Work

The Octagon domain [26] represents inequalities of the form $ax_i + bx_j \leq c$ where $a, b \in \{1, 0, -1\}$ and $x_i, x_j \in X$. The main novelty of [26] is to simultaneously work with a set of positive variables x_i^+ and negative variables x_i^- and consider a DBM over $\{x_1^+, x_1^-, \ldots, x_d^+, x_d^-\}$ where $d = |X|$. Then $x_i - x_j \leq c$, $x_i + x_j \leq c$ and $x_i \leq c$ can be encoded respectively as $x_i^+ - x_j^+ \leq c$, $x_i^+ - x_j^- \leq c$ and $x_i^+ - x_i^- \leq 2c$. Thus an $2d \times 2d$ square DBM matrix is sufficient for this domain. Note that this DBM representation contains entries of the form $x_i^+ - x_j^+ \leq \infty$ whenever $x_i - x_j$ is not constrained (and likewise for $x_i + x_j \leq c$ and $x_i \leq c$). Closure is computed with an all-pairs Floyd-Warshall shortest-path algorithm that is $O(d^3)$ and echos ideas in the early work of Pratt [30]. Other earlier work on this theme considered the domain of inequalities of the form $x_i - x_j \leq c$

[25,33], though the connection between bounded differences [9] and abstract interpretation dates back (at least) to Bagnara [1]. Very recently, Miné [27] has generalized DBMs to a class of domains that represent invariants of the form $x - y \in C$ where C is a non-relational domain that represents, for example, a congruence class [12]. This work is also formulated in terms of shortest-path closure and illustrates the widespread applicability of the closure concept.

Another thread of work is that of Su and Wagner [35] who propose a polynomial algorithm for calculating integer ranges as solutions to two variable per inequality systems, despite the intractability of some of these problems [21]. However, efficient integer hull algorithms do exist for the planar case [10,14]. Combined with our completion technique, this suggests a new tractable way of calculating the integer convex hulls for two variable systems that promises to be useful in program analysis.

It is well-known that the linear programming problem – the problem of maximizing a linear function subject to linear inequalities – is polynomial time (Turing) equivalent to the problem of deciding whether a linear system is satisfiable. Moreover, the problem of deciding whether a linear system is satisfiable can be transformed into an equivalent problem where each inequality contains at most three variables (with at most a polynomial increase in the number of variables and inequalities). Thus an efficient algorithm for solving this problem is also an efficient algorithm for solving the linear programming problem and vice versa. This equivalence, and negative results such as [20], explains the interest in checking the satisfiability of systems of linear inequalities where each inequality contains at most two variables that dates back to [29,30,34]. Of all the proposals for checking the satisfiability of a system T, the algorithm of [16] is most in tune with the requirements of abstract interpretation due to its succinctness and its $O(|T||var(T)|^2 \log(|T|))$ running time which is guaranteed without widening. This result (and related results) provide fast entailment checking algorithms which may be useful for efficient fixpoint detection.

The trade-off between expressiveness and tractability is also an important consideration in constraint solving and in this context the class of two variables per inequality has also received attention [15,18]. Jaffar *et al* [18] extend the closure algorithm of Shostak [34] for checking satisfiability over the reals to the integers by alternating closure with a tightening operation. However, this procedure is not guaranteed to either terminate nor detect satisfiability. Jaffar *et al* [18] go onto show that two-variables per inequality constraints with unit coefficients can be solved in polynomial time and that this domain supports efficient entailment checking and projection. More recently, Harvey and Stuckey [15] have shown how to reformulate this solver to formally argue completeness.

10 Conclusion

We proposed a new abstract domain of linear inequalities where each of the inequalities has at most two variables and the coefficients are unrestricted. We have shown how a (polynomial) completion operation leads to efficient and sim-

ple domain operations. Empirical evidence was presented that suggests that the domain is both tractable and well suited to widening.

Acknowledgments

We thank Roberto Bagnara, Les Hatton, Peter Linnington, Mike Norrish, Antoine Miné, Justin Pearson and Warwick Harvey for interesting discussions on polyhedra libraries and linear inequalities. We also thank the anonymous referees for their comments.

References

1. R. Bagnara. *Data-Flow Analysis for Constrant Logic-Based Languages*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1997.
2. V. Balasundaram and K. Kennedy. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. In *Programming Language Design and Implementation*, pages 41–53. ACM Press, 1989.
3. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *International Conference on Management of Data*, pages 16–52. ACM Press, 1986.
4. F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathbb{R}$). In *Logic Program Synthesis and Transformation (Selected Papers)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1997.
5. J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors. *Journal of the ACM*, 25:536–543, 1978.
6. N. V. Chernikova. Algorithm for Discovering the Set of All Solutions of a Linear Programming Problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.
7. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.
8. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
9. E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3):281–331, 1987.
10. S. D. Feit. A Fast Algorithm for the Two-Variable Integer Programming Problem. *Journal of the ACM*, 31(1):99–113, 1984.
11. R. L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972.
12. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *International Joint Conference on the Theory and Practice of Software Development*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer-Verlag, 1991.
13. W. H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3), 1977.

14. W. Harvey. Computing Two-Dimensional Integer Hulls. *SIAM Journal on Computing*, 28(6):2285–2299, 1999.
15. W. Harvey and P. J. Stuckey. A Unit Two Variable per Inequality Integer Constraint Solver for Constraint Logic Programming. *Australian Computer Science Communications*, 19(1):102–111, 1997.
16. D. S. Hochbaum and J. Naor. Simple and Fast Algorithms for Linear and Integer Programs with Two Variables per Inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.
17. J. M. Howe and A. King. Specialising Finite Domain Programs using Polyhedra. In *Logic Programming, Synthesis and Transformation (Selected Papers)*, volume 1817 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1999.
18. J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond Finite Domains. In *International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94. Springer-Verlag, 1994.
19. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
20. V. Klee and G. J. Minty. How Good is the Simplex Algorithm? In *Inequalities – III*. Academic Press, New York and London, 1972.
21. J. C. Lagarias. The Computational Complexity of Simultaneous Diophantine Approximation Problems. *SIAM Journal on Computing*, 14(1):196–209, 1985.
22. K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.
23. H. Le Verge. A Note on Chernikova's Algorithm. Technical Report 1662, Institut de Recherche en Informatique, Campus Universitaire de Beaulieu, France, 1992.
24. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 63–77. MIT Press, 1997.
25. A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.
26. A. Miné. The Octagon Abstract Domain. In *Eighth Working Conference on Reverse Engineering*, pages 310–319. IEEE Computer Society, 2001.
27. A. Miné. A Few Graph-Based Relational Numerical Abstract Domains. In *Ninth International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 117–132. Springer-Verlag, 2002.
28. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Conference of the European Association for Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1999.
29. C. G. Nelson. An $n^{\log(n)}$ Algorithm for the Two-Variable-Per-Constraint Linear Programming Satisfiability Problem. Technical Report STAN-CS-78-689, Stanford University, Department of Computer Science, 1978.
30. V. R. Pratt. Two Easy Theories Whose Combination is Hard, September 1977. http://boole.stanford.edu/pub/sefnp.pdf.
31. H. Raynaud. Sur L'enveloppe Convexe des Nuages de Points Aléatoires dans $\mathbb{R}^n$. *Journal of Applied Probability*, 7(1):35–48, 1970.
32. R. Seidel. Convex Hull Computations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 361–376. CRC Press, 1997.

33. R. Shaham, H. Kolodner, and M. Sagiv. Automatic Removal of Array Memory Leaks in Java. In *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2000.
34. R. Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the ACM*, 28(4):769–779, 1981.
35. Z. Su and D. Wagner. Efficient Algorithms for General Classes of Integer Range Constraints, July 2001. http://www.cs.berkeley.edu/~zhendong/.
36. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*. Internet Society, 2000.

Convex Hull Abstractions in Specialization of CLP Programs

Julio C. Peralta[1,⋆] and John P. Gallagher[2,⋆⋆]

[1] Instituto de Investigación en Matemáticas Aplicadas y en Sistemas
Circuito Escolar s/n, Ciudad Universitaria, México, D.F.
jperalta@leibniz.iimas.unam.mx, jpg@ruc.dk
[2] Dept. of Computer Science, Building 42.1, University of Roskilde
P.O. Box 260, DK-4000 Roskilde, Denmark
jpg@ruc.dk

Abstract. We introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls). This domain is used in an algorithm for specialization of constraint logic programs. The algorithm incorporates in a single phase both top-down goal directed propagation and bottom-up answer propagation, and uses a widening on the convex hull domain to ensure termination. We give examples to show the precision gained by this approach over other methods in the literature for specializing constraint logic programs. The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. Assignments, inequalities and equalities with arithmetic expressions can be interpreted as constraints during specialization, thus increasing the amount of specialization that can be achieved.

1 Introduction

Program specialization is sometimes regarded as being achieved in three phases: *pre-processing of the program, analysis* and *program generation*. During preprocessing the input program may be subject to some minor syntactic analyses or changes, ready for the analysis phase. The analysis computes some data-flow and control-flow information from the program and the specialization query. Finally, at program generation time the result of the analysis is used to produce a residual program reflecting the result of the analysis. In off-line specialization the three phases are consecutive, whereas in on-line specialization and driving the analysis and program generation phases are merged or interleaved.

The use of abstract interpretation techniques to assist program specialization is well-established [9,8,16,17,24] and goes back to the invention of binding time analysis to compute static-dynamic annotations [15]. More complex and expressive abstract domains have been used such as regular tree structures [22,10,12,18].

⋆ Supported in part by CONACYT Project I39201-A
⋆⋆ Partially supported by European Framework 5 Project ASAP (IST-2001-38059)

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 90–108, 2003.

In this paper we focus on an abstract domain based on arithmetic constraints. Abstract interpretations based on arithmetic constraints have already been developed [4,2,26]. We show how the analysis phase of a specialization algorithm can benefit from advances made in that field. We introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls [4]). The operations on this domain are developed from a standard constraint solver.

We then employ this domain within a generic algorithm for specialization of (constraint) logic programs [12]. The algorithm combines analysis over an abstract domain with partial evaluation. Its distinguishing feature is the analysis of the success constraints (or answer constraints) as well as the call constraints in a computation. This allows us to go beyond the capability of another recent approach to use a linear constraint domain in constraint logic program specialization [6].

The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. We can reason in constraint terms about the arithmetic expressions that occur in logic programs, treating them as constraints (for instance `X is Expr` is treated as {`X = Expr`}). In addition, the algorithm provides a contribution to the growing field of using specialization for model checking infinite state systems [19].

In this paper a constraint domain based on linear arithmetic equalities and inequalities is reviewed (Section 2). The structure of the specialization algorithm is presented (Section 3), along with examples illustrating key aspects. Next, in Section 4 more examples of specialization using the domain of linear arithmetic constraints are given. Then, comparisons with related work are provided in Section 5. Finally, in the last section (Section 6) some final remarks and pointers for future work are considered.

2 A Constraint Domain

Approximation in program analysis is ubiquitous, and so is the concept of a domain of properties. The analysis phase of program specialization is no exception.

2.1 Linear Arithmetic Constraints

Our constraint domain will be based on linear arithmetic constraints, that is, conjunctions of equalities and inequalities between linear arithmetic expressions. The special constraints true and false are also included. This domain has been used in the design of analysers and for model checking infinite state systems. Here we use it for specialization of (constraint) logic programs.

Let Lin be the theory of linear constraints over the real numbers. Let C and D be two linear constraints. We write $C \sqsubseteq D$ iff $\mathsf{Lin} \models \forall(C \rightarrow D)$. C and D are *equivalent*, written $C \equiv D$, iff $C \sqsubseteq D$ and $D \sqsubseteq C$. Let C be a constraint and V be a set of variables. Then $\mathsf{project}_V(C)$ is the projection of constraint C onto the variables V; the defining property of projection is that

$\mathsf{Lin} \models \forall V(\exists V'.C \leftrightarrow \mathsf{project}_V(C))$, where $V' = \mathsf{vars}(C) \setminus V$. Given an expression e let us denote $\mathsf{vars}(e)$ as the set of variables occurring in e. If $\mathsf{vars}(e) = V$, we sometimes refer to $\mathsf{project}_e(C)$ rather than $\mathsf{project}_V(C)$ when speaking of the projection of C onto the variables of e.

Arithmetic constraints can be presented in their simplified form, removing redundant constraints. Constraint simplification serves as a satisfiability check: the result of simplifying a constraint is false if and only if the constraint is unsatisfiable. If a constraint C is satisfiable, we write $\mathsf{sat}(C)$. Because we used the CLP facilities of SICStus Prolog all these operations (projection, simplification and checking for equivalence) are provided for the domain of linear constraints over rationals and reals. We refer the interested reader to a survey on CLP [14] for a thorough discussion on the subject.

Intuitively, a constraint represents a convex polyhedron in cartesian space, namely the set of points that satisfy the constraint. Let S be a set of linear arithmetic constraints. The *convex hull* of S, written $\mathsf{convhull}(S)$, is the least constraint (with respect to the $\sqsubseteq$ ordering on constraints) such that $\forall S_i \in S.S_i \sqsubseteq \mathsf{convhull}(S)$. So $\mathsf{convhull}(S)$ is the smallest polyhedron that encloses all members of S. Further details and algorithms for computing the convex hull can be found in the literature [4].

2.2 Constrained Atoms and Conjunctions

Now we must define our abstract domain. It consists of equivalence classes of *c-atoms*, which are constrained atoms. Each c-atom is composed of two parts, an atom and a linear arithmetic constraint.

Definition 1 (c-atoms and c-conjunctions). *A c-conjunction is a pair $\langle B, C\rangle$; B denotes a conjunction of atomic formulas (atoms) and C a conjunction of arithmetic constraints, where $\mathsf{vars}(C) \subseteq \mathsf{vars}(B)$. If B consists of a single atom the pair is called a c-atom.*

(Note that c-conjunctions are defined as well as c-atoms, since they occur in our algorithm. However, the domain is constructed only of c-atoms).

Given any arithmetic constraint C and atom A, we can form a c-atom $\langle A, C'\rangle$, where $C' = \mathsf{project}_A(C)$. Any atom A can be converted to a c-atom $\langle A', C\rangle$ by replacing each non-variable arithmetic expression occurring in A by a fresh variable[1], obtaining A'. Those expressions which were replaced together with the variables that replace them are added as equality constraints to the constraint part C of the c-atom. For example, the c-atom obtained from $p(f(3), Y + 1)$ is $\langle p(f(X_1), X_2), (X_1 = 3, X_2 = Y + 1)\rangle$.

A c-atom represents a set of concrete atoms. We define the *concretization* function γ as follows.

[1] By parsing the arguments the desired terms can be selected.

Definition 2 (γ). *Let $\mathcal{A} = \langle A, C\rangle$ be a c-atom. Define the concretization function γ as follows.*

$$\gamma(\mathcal{A}) = \left\{ A\theta \;\middle|\; \begin{array}{l}\theta \text{ is a substitution } \wedge \\ \forall\varphi.\mathsf{sat}(C\theta\varphi)\end{array} \right\}$$

γ is extended to sets of c-atoms: $\gamma(S) = \bigcup\{\gamma(\mathcal{A}) \mid \mathcal{A} \in S\}$.

There is a partial order on c-atoms defined by $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if and only if $\gamma(\mathcal{A}_1) \subseteq \gamma(\mathcal{A}_2)$. Two c-atoms $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent, written $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ and $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$. Equivalence can also be checked using syntactic comparison of the atoms combined with constraint solving, using the following lemma.

Lemma 1. *Let $\mathcal{A}_1 = \langle A_1, C_1\rangle$ and $\mathcal{A}_2 = \langle A_2, C_2\rangle$ be two c-atoms. Let $\langle \bar{A}_1, \bar{C}_1\rangle$ and $\langle \bar{A}_2, \bar{C}_2\rangle$ be the c-atoms obtained by removing repeated variables from A_1 and A_2 and adding constraints to C_1 and C_2 in the following manner. If a variable X occurs more than once in the atom, then one occurrence is replaced by a fresh variable W and the constraint $X = W$ is added to the corresponding constraint part.*

Then $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if there is a renaming substitution θ such that $\bar{A}_1\theta = \bar{A}_2$ and $\bar{C}_1\theta \equiv \bar{C}_2$.

Now we are in a position to define the domain and the operations on the elements of our domain. The relation $\equiv$ on c-atoms is an equivalence relation. The abstract domain consists of equivalence classes of c-atoms. For practical purposes we consider the domain as consisting of *canonical* constrained atoms, which are standard representative c-atoms, one for each equivalence class. These are obtained by renaming variables using a fixed set of variables, and representing the constraint part in a standard form. Hence we speak of the domain operations as being on c-atoms, whereas technically they are operations on equivalence classes of c-atoms.

Next we define the upper bound of c-atoms which combines the *most specific generalization operator (msg)* [25] on terms and the *convex hull* [4] on arithmetic constraints. The idea is to compute the *msg* of the atoms, and then to rename the constraint parts suitably, relating the variables in the original constraints to those in the *msg*, before applying the convex hull operation.

The following notation is used in the definition. Let θ be a substitution whose range only contains variables; the domain and range of θ are $\mathsf{dom}(\theta)$ and $\mathsf{ran}(\theta)$ respectively. $\mathsf{alias}(\theta)$ is the conjunction of equalities $X = Y$ such that there exist bindings X/Z and Y/Z in θ, for some variables X, Y and Z. Let $\bar{\theta}$ be any substitution such that $\mathsf{dom}(\bar{\theta}) = \mathsf{ran}(\theta)$ and $X\bar{\theta}\theta = X$ for all $X \in \mathsf{ran}(\theta)$. (That is, $\bar{\theta} = \varphi^{-1}$ where φ is some bijective subset of θ with the same range as θ).

The following definition is a reformulation of the corresponding definition given previously [26].

Definition 3 (Upper bound of c-atoms, $\sqcup$). *Let $\mathcal{A}_1 = \langle A_1, C_1\rangle$ and $\mathcal{A}_2 = \langle A_2, C_2\rangle$ be c-atoms. Their upper bound $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is c-atom $\mathcal{A}_3 = \langle A_3, C_3\rangle$ defined as follows.*

1. $A_3 = \mathsf{msg}(A_1, A_2)$, *where* $\mathsf{vars}(A_3)$ *is disjoint from* $\mathsf{vars}(A_1) \cup \mathsf{vars}(A_2)$.
2. *Let* $\theta_i = \{X/U \mid X/U \in \mathsf{mgu}(A_i, A_3), U \text{ is a variable}\}$, *for* $i = 1, 2$. *Then* $C_3 = \mathsf{project}_{A_3}(\mathsf{convhull}(\{\mathsf{alias}(\theta_i) \cup C_i\bar{\theta}_i \mid i = 1, 2\}))$.

$\sqcup$ *is commutative and associative, and we can thus denote by* $\sqcup(S)$ *the upper bound of the elements of a set of c-atoms* S.

Example 1. Let $\mathcal{A}_1 = \langle p(X, X), X > 0\rangle$ and $A_2 = \langle p(U, V), -U = V\rangle$. Then $\mathcal{A}_1 \sqcup \mathcal{A}_2 = \langle p(Z_1, Z_2), Z_2 \geq -Z_1\rangle$. Here, $\mathsf{mgu}(p(X, X), p(U, V)) = p(Z_1, Z_2)$, $\theta_1 = \{Z_1/X, Z_2/X\}$, $\theta_2 = \{Z_1/U, Z_2/V\}$, $\mathsf{alias}(\theta_1) = \{Z_1 = Z_2\}$, $\mathsf{alias}(\theta_2) = \emptyset$, $\bar{\theta}_1 = \{X/Z_1\}$ (or $\{X/Z_2\}$) and $\bar{\theta}_2 = \{U/Z_1, V/Z_2\}$. Hence we compute the convex hull of the set $\{(Z_1 = Z_2, Z_1 > 0), (-Z_1 = Z_2)\}$, which is $Z_2 \geq -Z_1$.

Like most analysis algorithms, our approach computes a monotonically increasing sequence of abstract descriptions, terminating when the sequence stabilizes at a fixed point. Because infinite ascending chains may arise during specialization it is not enough to have an upper bound operator, in order to reach a fixpoint. An operator called *widening* may be interleaved with the upper bound to accelerate the convergence to a fixpoint and ensure termination of an analysis based on this domain. When widening we assume that the c-atoms can be renamed so that their atomic parts are identical, and the widening is defined solely in terms of widening of arithmetic constraints, ∇_c [4]. This is justified since there are no infinite ascending chains of atoms with strictly increasing generality. Hence the atom part of the c-atoms does not require widening.

Definition 4 (Widening of c-atoms, ∇). *Given two c-atoms* $\mathcal{A}_1 = \langle A_1, C_1\rangle$ *and* $\mathcal{A}_2 = \langle A_2, C_2\rangle$, *where* A_1 *and* A_2 *are variants, say* $A_2 = A_1\rho$. *The widening of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, *denoted as* $\mathcal{A}_1 \nabla \mathcal{A}_2$ *is c-atom* $\mathcal{A}_3 = \langle A_2, C_3\rangle$ *where* $C_3 = C_1\rho \ \nabla_c \ C_2$.

For instance, the widening of $\langle p(X), X \geq 0, X \leq 1\rangle$ and $\langle p(Y), Y \geq 0, Y \leq 2\rangle$ is $\langle p(Y), Y \geq 0\rangle$.

3 An Algorithm for Specialization with Constraints

In this section we describe an algorithm for specialization, incorporating operations on the domain of convex hulls. The algorithm is based on one presented previously [12], where we used a domain of regular trees in place of convex hulls, and the operations named ω, calls and answers are taken from there. The operations ω and aunf^* (which is used in the definition of calls) were taken from Leuschel's top-down abstract specialization framework [17]. The answer propagation aspects of our algorithm are different from Leuschel's answer propagation method, though. There is no counterpart of the answers operation in Leuschel's framework. The differences between the approaches were discussed in our previous work [12].

The structure of the algorithm given in Figure 1 is independent of any particular domain of descriptions such as regular types or convex hulls. The operations concerning convex hulls appear only within the domain-specific operations calls, ω, ∇ and answers.

INPUT: a program P and a c-atom $\mathcal{A}$
OUTPUT: two sets of c-atoms (calls and answers)

begin
  $S_0 := \{\mathcal{A}\}$
  $T_0 := \{\}$
  $i := 0$
  repeat
    $S_{i+1} := \omega(\mathsf{calls}(S_i, T_i), S_i)$
    $T_{i+1} := T_i \nabla \mathsf{answers}(S_i, T_i)$
    $i := i + 1$
  until $S_i = S_{i-1}$ and $T_i = T_{i-1}$
end

Fig. 1. Partial Evaluation Algorithm with Answer Propagation

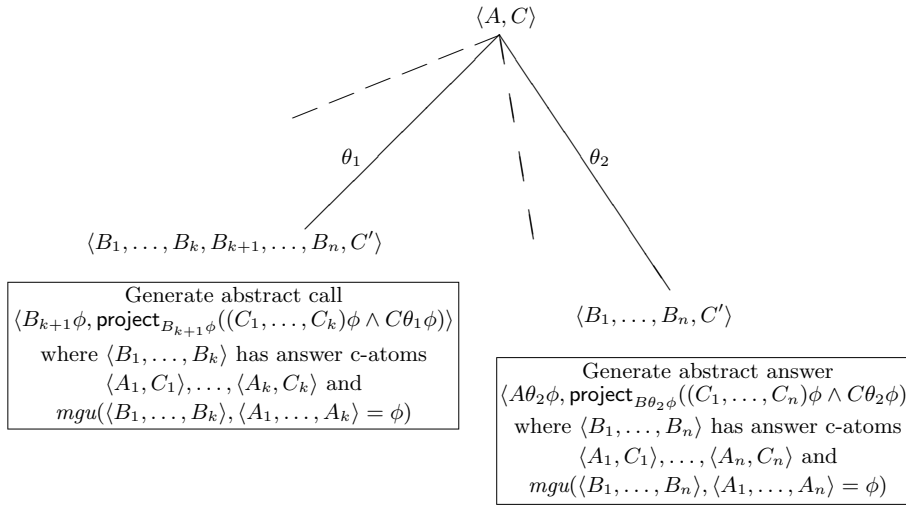


Fig. 2. The generation of calls and answers

3.1 Generation of Calls and Answers

The idea of the algorithm is to accumulate two sets of c-atoms. One set represents the set of *calls* that arise during the computation of the given initial c-atom $\mathcal{A}$. The other set represents the set of *answers* for calls.

At the start, the set of calls S_0 contains only the initial goal c-atom, and the set of answers T_0 is empty. Each iteration of the algorithm extends the current sets S_i and T_i of calls and answers. The diagram in Figure 2 illustrates the process of extending the sets. All existing calls $\mathcal{A} = \langle A, C\rangle \in S_i$ are unfolded according to some unfolding rule. This yields a number of *resultants* of the form $(A, C)\theta \leftarrow B_1, \ldots, B_l, C'$, where $A\theta \leftarrow B_1, \ldots, B_l$ is a result of unfolding A

and C' is the accumulated constraint; that is, C' is the conjunction of $C\theta$ and the other constraints introduced during unfolding. If $\mathsf{sat}(C')$ is false then the resultant is discarded. The unfolding process is performed in the algorithm by the operation aunf^*, defined as follows.

Definition 5 ($\mathsf{aunf}, \mathsf{aunf}^*$). *Let P be a definite constraint program and $\mathcal{A} = \langle A, C\rangle$ a c-atom. Let $\{A\theta_1 \leftarrow L_1, C_1, \ldots, A\theta_n \leftarrow L_n, C_n\}$ be some partial evaluation [20] of A in P, where $C_i, L_i (1 \leq i \leq n)$ are the constraint and non-constraint parts respectively of each resultant body. Then define*

$$\mathsf{aunf}(\mathcal{A}) = \left\{\ A\theta_i \leftarrow L_i, (C_i \wedge C\theta_i) \,\middle|\, 1 \leq i \leq n, \mathsf{sat}(C_i \wedge C\theta_i)\ \right\}.$$

Let S be a set of c-atoms. We define $\mathsf{aunf}^*(S)$ as:

$$\mathsf{aunf}^*(S) = \left\{\ (L, project_L(C')) \,\middle|\, \begin{array}{l}\langle A, C\rangle \in S \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A})\end{array}\ \right\}$$

In the following examples, assume that the unfolding rule selects the leftmost atom provided that it matches at most one clause (after discarding matches that result in an unsatisfiable constraint), otherwise selects no atom.

Example 2. Consider the following simple program P.

```
s(X,Y,Z) <-
  p(X,Y,Z), q(X,Y,Z)
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,Y1,Z1)
```

```
q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

Let S be $\{\langle \mathtt{s(X,Y,Z)}, \mathtt{X} > 2\rangle\}$. Then $\mathsf{aunf}^*(S) = \{(\mathtt{p(X1,Y,Z1)}, \mathtt{q(X,Y,Z)}, (\mathtt{X} > 2, \mathtt{X} = \mathtt{X1} + 3, \mathtt{Z} = \mathtt{Z1} + 3)\}$. The unfolding rule results in four steps: the unfolding of the atom `s(X,Y,Z)` followed by three unfoldings of `p`, since the initial constraint $\mathtt{X} > 2$ implies that the base case `p(0,0,0)` cannot be matched so long as the first argument of `p` is greater than zero.

Note that the range of the function aunf^* is the set of c-conjunctions. The current answers from T_i are then applied, from left to right, to the c-conjunctions generated by aunf^*. If there is some prefix $B_1 \ldots, B_k$ $(k < l)$ in a c-conjunction, having a solution in T_i, then a call to an instance of B_{k+1} is generated. More precisely, we define a function calls as follows. We first define the notion of a "solution" of a conjunction with respect to a set of c-atoms.

Definition 6 (solution of a conjunction). *Let $(B_1, \ldots, B_l)$ be a conjunction of atoms and T be a set of c-atoms. Then $\langle \varphi, \bar{C}\rangle$ is a solution for $(B_1, \ldots, B_l)$ in T if there is a sequence of c-atoms $\langle \mathcal{A}_1, \ldots, \mathcal{A}_l\rangle$ where $\mathcal{A}_j = \langle A_j, C_j\rangle \in T$, $1 \leq j \leq l$, such that $\mathsf{mgu}((B_1, \ldots, B_l), (A_1, \ldots, A_l)) = \varphi$, and $\mathsf{sat}(\bar{C})$ (where $\bar{C} = (C_1 \wedge \cdots \wedge C_l)\varphi$).*

Definition 7 (calls). *Let S_i be a set of call c-atoms and T_i be a set of answer c-atoms. Define $\mathsf{calls}(S_i, T_i)$ to be the set of c-atoms $\langle B_{k+1}\varphi, \mathsf{project}_{B_{k+1}\varphi}(\bar{C} \wedge C'\varphi)\rangle$ where*

1. $\langle B_1, \ldots, B_l, C' \rangle \in \mathsf{aunf}^*(S_i)$, *and*
2. *there is a conjunction* $(B_1, \ldots, B_k)$ $(k < l)$ *which has a solution* $\langle \varphi, \bar{C} \rangle$ *in* T_i, *and* $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 3. Let P be the program from Example 2 and let S be $\{\langle \mathtt{s(X,Y,Z)}, \mathtt{X} > \mathtt{2} \rangle\}$. Let $T = \{\langle \mathtt{p(X1,Y1,Z1)}, \mathtt{X1} = \mathtt{0}, \mathtt{Y1} = \mathtt{0}, \mathtt{Z1} = \mathtt{0} \rangle\}$. Then $\mathsf{calls}(S,T) = \{\langle \mathtt{p(X1,Y,Z1)}, \mathtt{true} \rangle, \langle \mathtt{q(X,Y,Z)}, \mathtt{X} = \mathtt{3}, \mathtt{Y} = \mathtt{3}, \mathtt{Z} = \mathtt{3} \rangle\}$. Note that the call to `q` arises from applying the solution for `p` and simplifying the accumulated constraints.

An answer is derived by finding a resultant $A\theta \leftarrow B_1, \ldots, B_k, C'$ whose body has a solution in the current set of answers. The function answers is defined as follows.

Definition 8 (answers). *Let* S_i *be a set of call c-atoms and* T_i *be a set of c-atoms. Define* $\mathsf{answers}(S_i, T_i)$ *to be the set of answer c-atoms* $\langle A\theta\varphi, \mathsf{project}_{A\theta\varphi}(\bar{C} \wedge C'\varphi) \rangle$ *where*

1. $\mathcal{A} = \langle A, C \rangle \in S_i$, *and*
2. $A\theta \leftarrow B_1, \ldots, B_l, C' \in \mathsf{aunf}(\mathcal{A})$, *and*
3. $(B_1, \ldots, B_l)$ *has a solution* $\langle \varphi, \bar{C} \rangle$ *in* T_i, *and* $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 4. Let P be the program from Example 2 and let S be $\{\langle \mathtt{p(X,Y,Z)}\ \mathtt{true} \rangle\}$. Let $T = \{\langle \mathtt{p(X1,Y1,Z1)}, \mathtt{X1} = \mathtt{0}, \mathtt{Y1} = \mathtt{0}, \mathtt{Z1} = \mathtt{0} \rangle\}$. Then $\mathsf{answers}(S,T) = \{\langle \mathtt{p(X,Y,Z)}, \mathtt{X} = \mathtt{1}, \mathtt{Y} = \mathtt{1}, \mathtt{Z} = \mathtt{1} \rangle\}$.

An important feature of the algorithm is that no call to a body atom is generated until the conjunction of atoms to its left has an answer. One effect of this is to increase specialization because the propagation of answers for some atom restricts the calls to its right. Secondly, answers can only be generated for called atoms, and no answer to an atom is generated until there is an answer to the whole body of some resultant for that atom. There can exist abstract calls that have no corresponding answers; these represent concrete calls that either fail or loop. In fact, infinitely failed computations are not distinguished from finitely failed computations, with the result that programs that produce infinitely failing computations can be specialized to ones that fail finitely. The examples later in this section illustrate this point.

3.2 Approximation Using Convex Hulls and Widening

Call and answer c-atoms derived using the calls and answers functions are added to the sets S_i and T_i respectively. There is usually an infinite number of c-atoms that can be generated in this way. The purpose of the ω and ∇ functions in the algorithm is to force termination. The ω function computes a safe approximation of the calls and answers, using the *convex hull* and *widening* operations, both of which are standard in analyses based on linear arithmetic constraints.

On each iteration, the sets of call c-atoms are partitioned into sets of "similar" c-atoms. The notion of "similar" is heuristic: the only requirements are that the

definition of similarity should yield a finite partition, and that all c-atoms in one subset should have the same predicate name. In our implementation we partitioned based on the *trace terms* or "unfolding patterns" of the c-atoms [11]. We assume a function that partitions a set S of c-atoms into a finite set $\{S_1, \ldots, S_m\}$ of disjoint subsets of S, and computes the upper bound of each subset. The function $\mathsf{partition}(S)$ is defined as $\mathsf{partition}(S) = \{\sqcup(S_1), \ldots, \sqcup(S_m)\}$. It is desirable though not essential that $\sqcup(S)$ belongs to the same set as S.

Even if the partition is finite, a widening is required to enforce termination. The widening step is defined between the sets of c-atoms on two successive iterations of the algorithm. Let S, S' be two sets of c-atoms, where we assume that both S and S' are the result of a $\mathsf{partition}$ operation. Define $S'\nabla S$ to be

$$\left\{ \mathcal{A}'\nabla\mathcal{A} \;\middle|\; \begin{array}{l} \mathcal{A}' \in S', \mathcal{A} \in S, \\ \mathcal{A}', \mathcal{A} \text{ are in the same set} \end{array} \right\}$$

$$\bigcup$$

$$\left\{ \mathcal{A} \;\middle|\; \begin{array}{l} \mathcal{A} \in S, \\ \nexists \mathcal{A}' \in S' \text{ in the same set as } \mathcal{A} \end{array} \right\}$$

Finally the operation ω can be defined as $\omega(S, S') = S'\nabla\mathsf{partition}(S)$. This ensures termination if the number of sets returned by $\mathsf{partition}$ is bounded. The definition states that each element $\mathcal{A}$ of S is replaced by the result of widening $\mathcal{A}$ with the element from S' from the same set, if such an element exists.

3.3 Generation of the Specialized Program

After termination of the algorithm, the specialized program is produced from the final sets of calls and answers S and T respectively. It consists of the following set of clauses.

$$\left\{ \mathsf{rename}(A\theta\varphi \leftarrow L\varphi, C'\varphi) \;\middle|\; \begin{array}{l} \mathcal{A} = \langle A, C\rangle \in S, \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A}), \\ L \text{ has solution } \langle \varphi, \bar{C}\rangle \text{ in } T, \\ \mathsf{sat}(\bar{C} \wedge C'\varphi) \end{array} \right\}$$

That is, each of the calls is unfolded, and the answers are applied to the bodies of the resultants. Note that we do not add the solution constraints $\bar{C}$ to the generated clause, so as not to introduce redundant constraints. The rename function is a standard renaming to ensure independence of different specialized versions of the same predicate, as used in most logic program specialization systems (see for example [8] for a description of the technique).

Example 5. Consider again the example from Example 2. We specialize this program with respect to the c-atom $\langle \mathtt{s(X,Y,Z)}, \mathsf{true}\rangle$ assuming the usual left-to-right computation rule. Note that the concrete goal `s(X,Y,Z)` does not have any solutions, although with the standard computation rule the computation is infinite.

After the first few iterations of the algorithm the answer for `p(X,Y,Z)` is computed, after widening the successive answers `p(0,0,0)`, `p(1,1,1)`, `p(2,2,2)`, ... This in turn generates a call to `q(X,Y,Z)`. The c-atom describing the answers for `p(X,Y,Z)` is $\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0 \rangle$ and thus the call $\langle \mathtt{q}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0 \rangle$ generated. Further iterations of the algorithm show that this call to `q` has no answers. Concretely, the call would generate an infinite failed computation.

When the algorithm terminates, the complete set of calls obtained is

$$\{\langle \mathtt{s}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true} \rangle, \langle \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true} \rangle, \langle \mathtt{q}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0 \rangle\}.$$

The set of answers is $\{\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0 \rangle\}$. Thus we can see that there are some calls (namely, to `q` and `s`) that have no answers.

To generate the specialized program from this set of calls and answers, we generate resultants for the calls, and apply the answers to the bodies. Since there is no answer for `q(X,Y,Z)` in the resultant for `s(X,Y,Z)`, `s(X,Y,Z)` fails and the specialized program is empty. The specialized program thus consists only of the resultants `p(0,0,0)` and `p(X,X,X) <- {X = Y+1}, p(Y,Y,Y)`. The failure of the original goal is immediately apparent since there are no clauses for predicate `s`.

Example 6. More insight into the nature of the approximation can be gained by considering the same program as in the previous example, except that the body goals are reversed in the clause for `s`. In this case `q(X,Y,Z)` is called first, and the answers for `q` constrain the calls to `p`. The call $\langle \mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true} \rangle$ results in the abstract answer c-atom $\langle \mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y} \rangle$. Again, widening is essential to derive this answer. Note that the solution `q(0,0,0)` is included as a result of the convex hull approximation, even though this is not a concrete solution.

This answer is then propagated to the call to `p`, hence there is a call c-atom $\langle \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y} \rangle$. Specialization of this call to `p` gives the abstract answer $\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0 \rangle$.

The specialized program corresponding to this set of calls and answers is the following.

```
s(0,0,0) <-
  q(0,0,0), p(0,0,0).
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,X1,X1)

q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

The instance of the clause for `s` is obtained by conjoining the answers for the body goals `q(X,Y,Z), p(X,Y,Z)`, that is, $\mathtt{X} \geq 0, \mathtt{X} = \mathtt{Y}, \mathtt{X} = \mathtt{Z}, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y}$, which simplifies to the constraint $\mathtt{X} = 0, \mathtt{Y} = 0, \mathtt{Z} = 0$. The above program does not make the failure of `s(X,Y,Z)` explicit; a non-trivial post-processing such as another run of the specialization algorithm would be needed to discover the failure of the call `q(0,0,0)`. The general point here is that the convex hull approximation loses the information that `q(0,0,0)` is not a solution for `q(X,Y,Z)`.

The two examples taken together show that the direction of propagation of answers affects precision. It would be possible to design an algorithm incorporating more sophisticated propagation, but post-processing or re-specialization is a practical alternative for experimental studies.

Note that the above presentation of the algorithm is naive in the sense that the sets of calls and answers need not be totally recomputed on each iteration. We use standard techniques to optimize the algorithm, focusing on the "new" calls and answers on each iteration. We can also use the recursive structure of the target program to optimize the iterative structure of the algorithm. Instead of one global fixpoint computation, we compute a series of fixpoints, one for each group of mutually recursive predicates.

3.4 Correctness of the Specialization

A program that has been specialized with respect to a c-atom $\mathcal{A} = \langle A, C\rangle$ produces the same answers as the original program for any terminating computation for any query in $\gamma(\mathcal{A})$. Note that the proposition below states nothing about the preservation of looping computations in the original program. A goal that loops in the original program can finitely fail in the specialized program.

Proposition 1. *Let P be a definite CLP program and $\mathcal{A}$ a c-atom. Let P' be the specialized program derived by the algorithm described above, with initial c-atom $\mathcal{A}$. Let S and T be the sets of call and answer c-atoms returned by the algorithm. Then for any goal $G =\leftarrow B_1, \ldots, B_k$ such that $i = 1 \cdots k$ and $B_i \in \gamma(\mathcal{A}')$ for some $\mathcal{A}' \in S$, $P \cup \{G\}$ has an answer ρ if and only if $P' \cup \{G\}$ has an answer ρ. Also, if $P \cup \{G\}$ fails finitely then $P' \cup \{G\}$ fails finitely.*

Proof. Suppose there is a terminating (possibly failed) derivation of $P' \cup \{G\}$. We argue by induction on the length of the derivation. If the derivation has length 0, then G fails immediately. We know that there is some $\mathcal{A}' = \langle A', C'\rangle$ in S such that $B_1 \in \gamma(\mathcal{A}')$, since the first call c-atom is $\langle B_1, \mathsf{true}\rangle$, and so S should contain an element $\mathcal{A}'$ such that $\langle B_1, \mathsf{true}\rangle \sqsubseteq \mathcal{A}'$. So a failure means that (i) there are no resultants for A', or (ii) that no resultant body has an answer, or (iii) that there is a resultant $A'\theta \leftarrow L$ with an answer φ for L given by the set of answer c-atoms, but B_1 does not unify with $A'\theta\varphi$. In the case of (i) there is a finitely failed computation tree of $P \cup \{G\}$. In the case of (ii) or (iii) there is either a finitely failed computation tree of $P \cup \{G\}$, or the computation tree for $P \cup \{G\}$ is infinitely failed.

If the derivation has length 1, with answer substitution ρ, then $G =\leftarrow B_1$ and there is some unit clause in P' whose head unifies with B_1 with substitution ρ. Now, unit clauses in P' may come from two sources: either they are already in P or they are the result of successfully unfolding the body of a non-unit clause, also in P. Hence by definition of the residual program construction the mgus are equivalent modulo variable renaming.

If all derivations of length at most m in P' have a corresponding derivation in P, then we show that all derivations of length $m + 1$ in P' do as well. Suppose the first clause used in the derivation is $A'\theta \leftarrow L$, $\mathsf{mgu}(B_1, A'\theta) = \varphi$ and

```
sat(_,true) <-
sat(_,false) <- fail
sat(E,P) <- prop(E,P)
sat(E,and(F,G)) <-
        sat(E,F),
        sat(E,G)
sat(E,or(_F,G)) <-
        sat(E,G)
sat(E,or(F,_G)) <-
        sat(E,F)
sat(E,not(F)) <-
        not(sat(E,F))
sat(E,en(F)) <-
        trans(_Act,E,Ei),
        sat(Ei,F)
sat(E,an(F)) <-
        not(sat(E,en(not(F))))
sat(E,eu(F,G)) <-
        sat_eu(E,F,G)
sat(E,au(F,G)) <-
      sat(E,not(eu(not(G),
                   and(not(F),not(G))))),
      sat_noteg(E,not(G))
sat(E,ef(F)) <-
      sat(E,eu(true,F))
sat(E,af(F)) <-
      sat_noteg(E,not(F))
sat(E,eg(F)) <-
      not(sat_noteg(E,F))
sat(E,ag(F)) <-
      sat(E,not(ef(not(F))))
sat_eu(E,_F,G) <-
      sat(E,G)
sat_eu(E,F,G) <-
      sat(E,F),
      trans(_Act,E,Ei),
      sat_eu(Ei,F,G)
sat_noteg(E,F) <-
      sat(E,not(F))
sat_noteg(E,F) <-
      not((trans(_Act,E,Ei),
           not(sat_noteg(Ei,F))))
```

Fig. 3. CTL metainterpreter

$(L, B_2, \ldots, B_k)\varphi$ has a derivation in P' of length at most m. By the induction hypothesis there is a corresponding derivation for $(L, B_2, \ldots, B_k)\varphi$ in P. Then clearly there is a derivation in P corresponding to the $m+1$ step derivation in P', obtained by concatenating the steps corresponding to the clause $A'\theta \leftarrow L$.

The above argument establishes soundness. For completeness, a sketch of a proof is provided. For each terminating derivation of $P \cup \{G\}$ we can construct a terminating derivation in $P' \cup \{G\}$. The clauses in P' that are needed to construct such a derivation exist by virtue of the closedness of the sets of calls and answers. That is $S = \omega(\mathsf{calls}(S,T), S)$ and $T = T \nabla \mathsf{answers}(S,T)$. Furthermore, the answers produced by successful derivations in P can be reproduced by derivations in P' by virtue of the correctness of the unfolding function aunf, and the procedure for computing the solution of a conjunction with respect to a set of answer c-atoms.

4 Examples

We implemented the algorithm described in the previous section, using the SICStus Prolog linear arithmetic constraint solver. Next we present some examples where on-line specialization as presented here is used for verifying some formulas in CTL [3].

Specialization can be seen as an approach to model-checking infinite systems [19,7] and in this context our more powerful specialization techniques are highly

relevant. We used the CTL metainterpreter shown in Fig. 3 (also used by M. Leuschel et al. [19]).

The set of transitions[2](predicate `trans/3` in the figure) of the system to be verified in the form of a (C)LP program is appended to this metainterpreter. Also, the property (predicate `prop/2` in the CTL metainterpreter) with respect to which verification is to be carried out should be specified. Finally, the specialization query provides the initial state and the CTL formula which is to be verified for the given system and initial state.

4.1 Specialization Strategy

Before applying the convex hull specialization, we performed a trivial top-down specialization with respect to the given goal. The main effect of this stage was to unfold the calls to the transition relation `trans/3`. In principle, this unfolding could be performed during the execution of the main specialization algorithm. However, the overall process is faster and easier to control when doing the specialization in two stages.

Example 7. Consider for instance the following transition system, where `trans(t,[X,Y],[Z,W])` holds iff state `[Z,W]` may be obtained from state `[X,Y]` using transition `t`.

```
trans(t1,[P1,P2],[X,P3]) <-
        X is 0,
        P1>=1,
        P2>=0,
        P3>=0,
        P3 is P2+1

trans(t2,[P1,P3],[P4,P2]) <-
        P1>=0,
        P2>=0,
        P4 is P1+2,
        P3 is P2+1
```

The encoding of an unsafe state property `[X,Y]` with `X>=3` is added as another clause in the CTL metainterpreter.

```
prop([X,Y],p(unsafe)) <- X>=3
```

The specialization query from initial state `[X,Y]` with `X=1,Y=0` for CTL formula `ef(p(unsafe))`[3] is `<- sat([1,0],ef(p(unsafe)))`. As a result of specializing the CTL metainterpreter with a description (transition system) of the system and a state property with respect to the query above, we obtained the empty program. This is equivalent to saying that there is no residual program in which state `[1,0]` may reach state `[X,Y]` with `X>=3`. Had we obtained a residual program we would have interpreted the residual program as the set of traces which lead from the initial state to the unsafe state, as above.

This behaviour may be regarded as that of a model checker, hence we argue that our specializer may be used as a model checker for some infinite state systems. The only requirement is that those systems may be expressed as definite (constraint) logic programs and the CTL formulas does not use negation.

[2] A transition system may be that of a Kripke structure or a Petri Net, for instance.
[3] Meaning that there exists a state in the future such that state property `unsafe` holds.

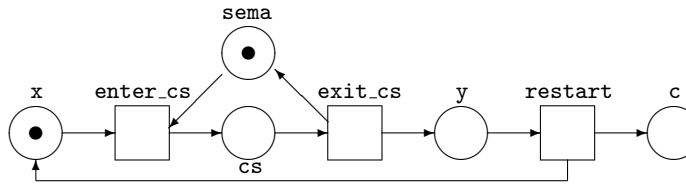


Fig. 4. Petri Net with one semaphore

Example 8. Figure 4 depicts a Petri net modeling one process with its critical section (`cs`) and a semaphore (`sema`) controlling access to it. The definition of predicate `trans/3` corresponding to the transition relation of the Petri net above, follows.

```
trans(enter_cs,[X,Sema,Cs,Y,C],[X1,Sema1,Cs1,Y,C]) <-
        X>=1, X1 is X-1,
        Sema>=1, Sema1 is Sema-1,
        Cs>=0, Cs1 is Cs+1
trans(exit_cs,[X,Sema,Cs,Y,C],[X,Sema1,Cs1,Y1,C]) <-
        Sema>=0, Sema1 is Sema+1,
        Cs>=1, Cs1 is Cs-1, Y>=0, Y1 is Y+1
trans(restart,[X,Sema,Cs,Y,C],[X1,Sema,Cs,Y1,C1]) <-
        X>=0, X1 is X+1,
        Y>=1, Y1 is Y-1,
        C>=0, C1 is C+1
```

Next, we may specify with the following clause the unsafe property of more than two processes being in their critical section (`cs`) at the same time:

```
prop([_X,_Sema,Cs,_Y,_C],p(unsafe)) <-  Cs>=2.
```

Now, for the specialization query, with constraint[4] `X>=1`:

```
<- sat([X,1,0,0,0],ef(p(unsafe)))
```

we obtained the empty program, thus denoting that there is no path from the initial state (`[X,1,0,0,0]`), with `X>=1` leading to a state where property `p(unsafe)` holds.

Example 9. Another way of specifying concurrent systems was proposed by U. A. Shankar [28]. Delzanno and Podelski [5], in turn, propose a systematic method to translate such specifications into CLP programs. Our translation is similar to theirs, differing only in the form of the clauses produced, mainly due to the meaning of the predicate employed.

Figure 5 below contains a specification of the bakery algorithm for two processes using the technique above cited.

[4] For every token in the place with name `X` we associate a process, thus the constraint `X>=1`.

Control variables $p_1, p_2 : \{think,\ wait,\ use\}$
Data variables $turn_1, turn_2$: int
Initial condition $p_1 = think \wedge p_2 = think \wedge turn_1 = turn_2 = 0$

Events	**cond** $p_1 = think$	**action** $p_1' = wait \wedge turn_1' = turn_2 + 1$
	cond $p_1 = wait \wedge turn_1 < turn_2$	**action** $p_1' = use$
	cond $p_1 = wait \wedge turn_2 = 0$	**action** $p_1' = use$
	cond $p_1 = use$	**action** $p_1' = think \wedge turn_1' = 0$
	... symmetrically for Process 2	

Fig. 5. The bakery algorithm

Such a specification may be readily translated into the following definition of the `trans` predicate:

```
trans(f,[think,A,P2,B],[wait,A1,P2,B]) <-  A>=0, A1 is B+1
trans(f,[P1,A,think,B],[P1,A,wait,B1]) <-  B>=0, B1 is A+1
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  A>=0, A<B
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  B>=0, B<A
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  B=0
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  A=0
trans(t,[use,A,P2,B],[think,A1,P2,B]) <-  A>=0, A1=0
trans(t,[P1,A,use,B],[P1,A,think,B1]) <-  B>=0, B1=0
```

Consequently, an unsafe property for the previous system would be a state where the two processes are in their critical section (denoted as `use`) at the same time. This property is denoted as the clause:

```
prop([use,A,use,B],p(unsafe)) <-
```

Furthermore, verifying that there is no state of the above mentioned system where such an unsafe state holds amounts to obtaining an empty program for the following query:

```
<-sat([think,0,think,0],ef(p(unsafe)))
```

where the variables denoting the turn of each process, namely `A,B`, are initially constrained by `A=B=0`. As a result of the specialization we obtained the empty program thus verifying that there is no unsafe state in any path beginning from the initial state described in figure 5.

In a similar way we verified some correctness property [19] of the producers and consumers algorithm [1] for one producer, one consumer and a buffer of size one. The authors [19] could not successfully specialize this last example.

4.2 Assessment

Here we have shown some applications of our specialization strategy to infinite state model checking. Compared to other approaches using specialization for the same purpose, we believe our approach sheds some insight into the field. The

example of the bakery protocol was also verified by Fioravanti et al. [7]. As opposed to their approach we show the actual specialization strategy and its use in other related examples. We depart from a general CTL metainterpreter whereas Fioravanti et al. present a somehow specialized version of a CTL metainterpreter.

For the other examples of this section M. Leuschel et al. [19] have a four stage model checker, as opposed to ours which is just one specialization step. That is, M. Leuschel et al. first pass through an off-line specializer, then one or more specialization passes of their on-line specializer and finally one pass through a most-specific-version analyser.

Admittedly examples 8 and 9 in this section do not propagate answers, and require a simple unfolding prior to specialization with answers. By contrast, example 7 and the producer-consumer of [19] do not need any prior unfolding and have some limited answer propagation. That is, specialization with answers could be applied directly to the metainterpreter (together with the transition definition and the property), to yield the expected verification results. The running example of Section 3 does indeed need and use answer propagation.

5 Related Work

Despite the fact that unfold-fold approaches to program transformation and program specialization based on a fixpoint calculation are not directly comparable, there are some unfold-fold methods related to our techniques.

In [21] the authors propose the use of convex-hull analysis to enable optimization/specialization of CLP programs. Their removal, refinement and reordering may be rendered as transformation rules. The fairness of comparing our technique with theirs is dubious because theirs is used for compilation and ours for specialization, and potentially the former is a special case of the latter one. A weak form of their method was later dubbed by Fioravanti et al. [6] as contextual specialization.

Peralta and Gallagher [23] use arithmetic constraints (convex hulls) to specialize CLP programs, especially an interpreter for imperative programs.

Their specialization abstract domain is the same as that one used here, but the specializer only propagates information top-down and cannot achieve the effects of answer propagation.

Fioravanti et al. [6] (without reference to [13,23]) argue an automatic specialization method based on folding and unfolding among other transformation techniques. They use a domain of atomic formulas constrained by arithmetic expressions with upper bound based on widening alone, rather than the combination of convex hull and widening which is known to give better approximations. The aspects of their method concerned with specialization resemble a top-down on-line specializer with a subsequent "contextual specialization", and thus does not in general achieve the effects of answer propagation.

Another application of specialization using abstract interpretation over polyhedral descriptions followed by a contextual specialization was given by Howe *et al.* [13]. This approach is similar in being based on abstract interpretation over

a domain of polyhedra. Its bottom-up analysis of answers is not as powerful as ours, which combines top-down and bottom-up propagation.

Conjunctive Partial Deduction (CPD) [27] aims to solve the answer propagation problem in a different way. The approach is to preserve shared information between subgoals by specializing conjunctions rather than atoms. It is not yet clear whether CPD or answer propagation via atoms, or some combination of both, will be most effective. In the extreme case of CPD, no resolvent is ever split, and no answer propagation is needed. However in general resolvents can be of unbounded size, some splitting is therefore needed, and answer propagation is required to preserve shared information between conjunctions.

6 Final Remarks

We have presented an algorithm for specialization of definite (C)LP programs. Its main novelty is the propagation of calls and answers described by atoms whose arguments are described by convex hulls. The use of answer propagation with an expressive domain like convex hulls gives increased specialization. By interpreting Prolog arithmetic as constraints we can also apply the algorithm to "non-constraint" programs.

6.1 Future Work

At the moment we can only specialize definite (constraint) logic programs. Because negation in CTL is interpreted as negation in (constraint) logic programs, this restricts us to model checking of safety properties, as opposed to liveness properties. Extending the presented techniques to include negation is the focus of our current research.

Scalability of our specialization method is one avenue into which we plan to extend the current proposal. Thus making our specialization techniques applicable to larger systems.

Also, in order to improve precision of our specialization with answers more sophisticated domains are sought.

References

1. Krysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.
2. F. Benoy and A. King. Inferring argument size relations in CLP($\mathcal{R}$). In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, pages 204–223, Sweden, 1996. Springer-Verlag, LNCS 1207.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 2000.
4. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978. Also in "http://www.di.ens.fr/~cousot/COUSOTpapers/POPL78.shtml".

5. G. Delzanno and A. Podelski. Model Checking in CLP. In W. R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–239. Springer-Verlag, LNCS 1579, 1999.
6. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Automated strategies for specialising constraint logic programs. In Kung-Kiu Lau, editor, *10th International Workshop on Logic-based Program Synthesis and Transformation*, pages 125–146. Springer-Verlag, LNCS 2042, 2000.
7. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. Technical Report DSSE-TR-2001-3, Department of Electronics and Computer Science, University of Southampton, 2001. Proceedings of the Second International Workshop on Verification and Computational Logic (VCL'01).
8. 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.
9. J. Gallagher, M. Codish, and E.Y. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.
10. J. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.
11. John P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemman, editors, *Partial Evaluation*, pages 115–136. Springer-Verlag, LNCS 1110, 1996.
12. John P. Gallagher and Julio C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3):143–172, 2001.
13. J.M. Howe and A. King. Specialising finite domain programs using polyhedra. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *Springer-Verlag Lecture Notes in Computer Science*, pages 118–135, April 2000.
14. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19(20):503–581, 1994.
15. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.
16. Neil D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pages 396–405, 1997.
17. M. Leuschel. Program specialisation and abstract interpretation reconciled. In Joxan Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press.
18. M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In A. Pettorossi, editor, *(Pre)Proceedings of LOPSTR-2001 11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR-2001), Paphos, Cyprus*, December 2001.
19. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 62–81. Springer Verlag, LNCS 1817, 1999.
20. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

21. K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *Proceedings of the Twentieth Symposium on Principles of Programming Languages*, pages 334–344, Charleston, South Carolina, 1993. ACM Press.
22. Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.
23. Julio C. Peralta and John P. Gallagher. Imperative program specialisation: An approach using CLP. In Annalisa Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 102–117. Springer Verlag, LNCS 1817, 1999.
24. G. Puebla, M. Hermenegildo, and J. P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 75–84, San Antonio, Texas, January 1999.
25. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, pages 135–151, 1970.
26. Hüseyin Sağlam and John P. Gallagher. Constrained regular approximations of logic programs. In N. Fuchs, editor, *LOPSTR'97*, pages 282–299. Springer-Verlag, LNCS 1463, 1997.
27. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2-3):231–277, 1999. Erratum appeared in JLP 43(3): 265(2000).
28. U. A. Shankar. An Introduction to Assertional Reasoning for Concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.

Collecting Potential Optimisations

Nancy Mazur, Gerda Janssens, and Wim Vanhoof

Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B–3001 Heverlee, Belgium
{nancy,gerda,wimvh}@cs.kuleuven.ac.be

Introduction and Motivation

Most program analyses and optimisations face the problem of controlling polyvariance, i.e. deciding in how many ways a single program entity is analysed and/or annotated for optimisation. In the context of logic programming, an analysis creates a number of versions for each original predicate, where each such version corresponds with a particular use or optimisation of the predicate. There is a trade-off between the number of versions (the size of the program) and the optimality of the resulting program.

The problem can be tackled in different ways. The classic approach is to analyse the program in a top-down fashion, starting from an initial call, and to create a different version of a predicate for each description of a call that occurs during the analysis. To counter the creation of too many versions, a pruning phase can be introduced, like in [3]. A disadvantage of such a call-dependent approach is that the program typically needs to be reanalysed for each newly encountered use of a predicate (as the underlying analysis usually only guarantees that the optimisations are safe for a restricted number of uses of the predicate). Moreover, if the analysis limits the number of versions that it creates, it mainly achieves this by widening or abstraction of the set of encountered call descriptions without taking into account the optimisations induced by these call descriptions. Another typical approach to version control is the one we followed in the context of compile-time garbage collection, where we generate at most two versions of each predicate: a non-optimised version that is always safe to call, and a fully optimised version that can only be used if the caller of the predicate meets the (often harsh) conditions that guarantee that *all* the optimisations can safely be performed. Although there is no risk for code explosion, a lot of intermediate opportunities for optimisations are missed.

The underlying problem is that most of the approaches lack an adequate mechanism for collecting and comparing the possible optimisations *before* the actual versions are created. In this work we propose an *optimisation derivation system* that derives the explicit relations that exist between a particular optimisation opportunity within a predicate and the requirements on the call that allow the optimisation to be safely performed. This information allows to (automatically) reason about the versions that are interesting to generate (e.g. how many and which optimisations are induced by a particular call description); it can be interesting feedback to the programmer (e.g. why can some specific atom

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 109–110, 2003.

not be optimised?); and finally, such a system can be seen as a step towards conceptually separating the analysis of a program from the generation of the versions, opening the field for better theoretical insights into the latter problem.

Optimisation Derivation Framework

We assume that the user program comes pre-annotated with goal-independent information at each *atom of interest*. This information is expressed in terms of an abstract domain $\mathcal{A}$ upon which the intended optimisations for these atoms can be decided. We require that $\mathcal{A}$ has a (monotone) combination operator $\otimes : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ which is used to safely approximate goal-dependent information given a description of a specific call and a goal-independent annotation.

The atoms of interest consist of *base atoms* – atoms for which the optimisation criteria are predefined, and *call atoms* – predicate calls through which optimisation criteria are propagated. Let o_a be a function that, when applied to a goal-dependent abstract description for the atom a, returns true if the intended optimisation for a is safe for that description, and false otherwise.

Consider the predicate definition: $p \leftarrow \ldots a, \ldots$. Let a be an atom of interest, and let ι_a be the goal-independent annotation of a. Let δ_p be a description of the call to p, then we can verify the optimisation of a by computing $o_a(\iota_a \otimes \delta_p)$. Our idea is to use the above formula to find such a "most general" δ_p. This can be done by computing the *generalised pseudo-complement* of ι_a w.r.t. $\otimes$ (much like the pseudo-complement is used in the backwards propagation of call patterns [1]). Using the pseudo-complement, a new function o_p can be generated with which the optimisation of a within calls to p can be checked by checking the goal-dependent abstract descriptions of calls to p. If each predicate contains only one atom of interest, then the propagation of these functions up the call graph is straightforward. If p contains several atoms of interest, their optimisation functions need to be combined in order to propagate them up the call graph. There are a number of ways to combine these functions: e.g. by using the least upper bound or the greatest lower bound, or by collecting them as sets. The exact choice has an effect on the information resulting from the analysis: either the system derives necessary conditions or sufficient conditions. Further details are discussed in [2].

References

1. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4-5):517–547, July-September 2002.
2. N. Mazur, G. Janssens, and W. Vanhoof. Collecting Potential Optimisations. Report CW 357, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2003.
3. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

An Operational Approach to Program Extraction in the Calculus of Constructions

Maribel Fernández[1] and Paula Severi[2,⋆]

[1] DI-LIENS, École Normale Supérieure/CNRS UMR 8548,
45 rue d'Ulm, 75005 Paris, France
maribel@di.ens.fr
[2] Departimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
severi@di.unito.it

Abstract. The Theory of Specifications is an extension of the Calculus of Constructions where the specification of a problem, the derivation of a program, and its correctness proof, can all be done within the same formalism. An operational semantics describes the process of extracting a program from a proof of its specification. This has several advantages: from the user's point of view, it simplifies the task of developing correct programs, since it is sufficient to know just one system in order to be able to specify, develop and prove the correction of a program; from the implementation point of view, the fact that the extraction procedure is part of the system allows to control in a finer way its interactions with the rest of the system. In this paper we continue the study of the Theory of Specifications and propose a solution to restore subject reduction and strong normalization. Counterexamples for subject reduction and strong normalization for this theory have been shown in [RS02].

1 Introduction

A specification of a program, such as *for every finite list of natural numbers there is a sorted permutation*, is in general of the form $\forall x.\exists y.P(x,y)$. In type theory, this is expressed as a type $\Pi x{:}A.\Sigma y{:}B.P(x,y)$. The idea of program extraction is to extract from an inhabitant t of such a type a function $f : A \to B$ with the property that $P(x, f(x))$ holds for all x. The problem with standard extraction methods is that in general the extracted program contains superfluous information from its correctness proof.

In the last two decades several approaches to program extraction in type theory have been studied. In the Theory of Specifications introduced in [SS01,SS02], the process of extracting a program from a proof of its specification is modelled by means of a reduction relation $\to_\sigma$ which erases all superfluous information from the program. In other words, the operational semantics describes explicitly

⋆ This work has been supported by the ECOS-Sud program of cooperation between France and Uruguay. The second author is also partially supported by IST-2001-322222 MIKADO; IST-2001-33477 DART.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 111–125, 2003.

the extraction procedure. This has several advantages: on the one hand, the extracted program, its specification and its correctness proof are all expressed in the same language; on the other hand, by making the extraction procedure explicit, we are able to control it in a finer way (this is analogous to what happens in explicit substitution calculi, where by making the substitution operation part of the system we are able to control its application).
The Theory of Specifications can also be used to model the method of program development by stepwise refinement (see for instance [Luo93] for a presentation of data refinement using type theory). The powerful type system of the Theory of Specifications allows us to define functions between specifications, and the σ-reduction can be used to compute the correctness condition of a refinement step.
The formulation of the Theory of Specifications given in [SS01] is based on Martin Löf's type theory and the formulation in [SS02] is based on the Calculus of Constructions. In this paper we focus on the second formulation. We add a pair constructor and a σ-conversion rule to the Calculus of Constructions. In our notation, we use the same pair constructor for types and objects. A Σ-type is abbreviated as a pair, i.e. $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,, \lambda x{:}A.P\rangle$. Due to the addition of the σ-reduction rules, the pair constructor becomes stronger than the strong-Σ [Luo89]: It has at least the expressive power of strong-Σ since it is possible to code the first and second projections and hence it is not necessary to add them as primitives [SS02]. Moreover, it goes beyond strong-Σ since it has an additional property which makes program extraction always possible. This additional property corresponds to the internalization of the notion of realizability: any specification is computationally equal to a basic specification $\Sigma x{:}A.(Px)$ and any proof of this specification is computationally equal to a pair $\langle a\,, p\rangle$ with $a{:}A$ and $p{:}(Pa)$. Then, program extraction from an inhabitant of a specification consists in just taking the first component of the pair. For instance, an inhabitant of a specification $\Pi x{:}A.\Sigma y{:}B.(P\,x\,y)$ reduces to a pair $\langle f\,, q\rangle$ where f is the extracted program of type $A \to B$ and q is the proof of its correctness $\Pi x{:}A.(P\,x\,(f\,x))$.
The correctness of the program extraction process is proved in [SS01,SS02], by showing that the σ-reduction relation is normalizing and confluent, and that normal forms of specifications are pairs consisting of a program and its correctness proof. The components of a pair are typable in a subsystem of the Theory of Specifications called Verification Calculus, which excludes the formation of data types depending on propositions.
In a recent paper [RS02] it is shown that subject reduction and strong normalization in the Theory of Specifications of [SS02] do not hold. In this paper, we give a new formulation of the Theory of Specifications that enjoys the properties of subject reduction and strong normalization. These are important properties from the point of view of the implementation of the theory since they ensure that any evaluation strategy will eventually reach a normal form and there is no need for dynamic type checking.

Related Work. In Coq until version 6.3 [Bar99] the extraction procedure has been performed by means of an external function based on realizability interpretations [PM89b,PM89a]. The normal form of our σ-reduction corresponds to the functions $\mathcal{E}$ and $\mathcal{R}$ of [PM89b,PM89a] which compute the extracted program and the proof of its correctness respectively. The σ-reduction extends $\mathcal{E}$ and $\mathcal{R}$ to type systems beyond $F\omega$ and CC [SS02], and it allows us to study the intermediate steps in the process of program extraction since an inhabitant s of a specification may need several steps of σ-reduction to reach the normal form $\langle \mathcal{E}(s), \mathcal{R}(s) \rangle$. The extraction procedure in Coq version 7 [Tea] is also based on a reduction relation, however, there is an important difference: in all versions of Coq so far the extracted program is given in ML whereas the Theory of Specifications unifies the programming language and the logic (they are the same language). The σ-reduction is integrated into the system as part of its evaluation mechanism.

The Theory of Specifications is also related to other programming logics that allow the derivation of implementations as pairs program-proof developed in parallel. Among these systems, we can mention for instance the Deliverables [BM90] and the programming logic $\lambda\omega_L$ [Pol94].

In the Theory of Deliverables, a specification is a pair consisting of a data type and a predicate over it, and the definition of deliverable corresponds in a certain way to our notion of functions between specifications. Σ-types are used to put together both the components of specifications and the functions between them. This makes it problematic to obtain a good definition of function in a direct way. To overcome this difficulty second-order deliverables have to be defined using a global specification as a parameter. In the Theory of Specifications, first and second order deliverables can be uniformly expressed using the Π-constructor as functions between specifications.

Poll's $\lambda\omega_L$ is a subsystem of the Verification Calculus. However, the notions of specification and implementation are not completely formalized in this language. Our notion of pair formalizes Poll's idea of couple-derivation rules which belongs to the meta-language in [Pol94].

In [Luo93] a different approach to program derivation is presented using an extended Calculus of Constructions. The idea is to specify a data type and then refine it until an implementation is obtained. This approach seems closer in spirit to the formal derivation of a program from a specification using the B method [Abr96]. The refinement function, which is defined externally in the theory of [Luo93] (it is not part of the reduction relation), can also be internalized in the Theory of Specifications, using functions between specifications.

Overview. We define an extended Calculus of Constructions and the Verification Calculus as Pure Type Systems in Section 2. In Section 3 we present a new version of the Theory of Specifications and give an example of program extraction. In Section 4 we prove several properties of the Theory of Specifications, in particular strong normalization and subject reduction, which are the main contributions of the paper. We conclude in Section 5.

2 Background

In this section we recall the definition of the Calculus of Constructions extended with an infinite type hierarchy [Luo89,Bar99] and introduce the Verification Calculus. We assume the reader to be familiar with the notion of Pure Type System [Bar92], and recall just the typing rules.
Let $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be the specification of a Pure Type System, i.e. a set $\mathcal{S}$ of sorts, a set $\mathcal{A}$ of axioms and a set $\mathcal{R}$ of rules. A Pure type System is defined by the rules shown in Figure 1.

Axiom
$\vdash k_1{:}k_2 \quad (k_1, k_2) \in \mathcal{A}$

Start
$$\frac{\Gamma \vdash U{:}k}{\Gamma, x{:}U \vdash x{:}U}\ x\ \Gamma\text{-fresh}$$

Weakening
$$\frac{\Gamma \vdash U{:}k \quad \Gamma \vdash v{:}V}{\Gamma, x{:}U \vdash v{:}V}\ x\ \Gamma\text{-fresh}$$

β-Conversion
$$\frac{\Gamma \vdash u{:}U \quad \Gamma \vdash U'{:}k}{\Gamma \vdash u{:}U'}\ U =_\beta U'$$

Product $(k_1, k_2, k_3) \in \mathcal{R}$
$$\frac{\Gamma \vdash U{:}k_1 \quad \Gamma, x{:}U \vdash V{:}k_2}{\Gamma \vdash \Pi x{:}U.V{:}k_3}$$

Abstraction
$$\frac{\Gamma, x{:}U \vdash v{:}V \quad \Gamma \vdash \Pi x{:}U.V{:}k}{\Gamma \vdash \lambda x{:}U.v{:}\Pi x{:}U.V}$$

Application
$$\frac{\Gamma \vdash v{:}\Pi x{:}U.V \quad \Gamma \vdash u{:}U}{\Gamma \vdash v\,u{:}V[u/x]}$$

Fig. 1. Typing rules for Pure Type Systems

The specification C for the extended calculus of constructions. The extended Calculus of Constructions [Luo89] is obtained as a Pure Type System with the following specification C.

1. Sorts. $Type^n$ for all $n \in \mathsf{Nat}$.
2. Axioms. $Type^n : Type^{n+1}$ for all $n \in \mathsf{Nat}$.
3. Rules. $(Type^n, Type^0, Type^0)$ $(Type^n, Type^m, Type^m)$ for $n \leq m$.

Duplicating C for the Verification Calculus. The Verification Calculus, used to prove the soundness of the Theory of Specifications, can be defined as a PTS with specification VC consisting of two copies of C, one for data types and the other for propositions.

1. Sorts. The sorts are either data-sorts $dType^i$ or prop-sorts $pType^i$ for $i \in \mathsf{Nat}$. Note that the sorts in VC are obtained by concatenating the letter d or p to the sorts of C. We write dk for the concatenation of the letter d with the sort k and pk for the concatenation of the letter p with the sort k.
2. Axioms. For each axiom $k : k'$ in C we add two axioms in the Verification Calculus:
$$dk : dk' \qquad pk : pk'.$$
3. Rules. For each rule (k_1, k_2, k_3) in C, we add three rules in the Verification Calculus:
$$(dk_1, dk_2, dk_3) \qquad (dk_1, pk_2, pk_3) \qquad (pk_1, pk_2, pk_3)$$

3 The Theory of Specifications

In this section we define the Theory of Specifications as a variant of [SS02] and give an example of an application. The reduction relation in this theory is the combination of β and σ. Counterexamples to subject reduction and strong normalization for β-reduction in the Theory of Specifications of [SS02] are shown in [RS02]. We restore subject reduction and strong normalization by restricting β-reduction to σ-normal forms.

3.1 Triplicating C for the Theory of Specifications

The Theory of Specifications is a Pure Type System extended with pairs. Its specification TS (sorts, axioms and product rules) is defined as follows:

1. Sorts. The sorts in the Theory of Specifications are either data-sorts $dType^i$, prop-sorts $pType^i$ or spec-sorts $sType^i$ for $i \in \mathsf{Nat}$. Note that for each sort k in C, the data-sorts are obtained as the concatenation of d with k (written dk), similarly we obtain the prop-sorts and the spec-sorts as pk and sk.
2. Axioms. For each axiom $k : k'$ in C we add three axioms in the Theory of Specifications:

$$dk : dk' \qquad pk : pk' \qquad sk : sk'$$

3. Rules. For each rule (k_1, k_2, k_3) in C, we add the following rules:

$$(uk_1, vk_2, vk_3)$$

where u and v are either d, p or s.

3.2 Splitting the Syntax

The set of pseudoterms is extended with expressions of the form $\langle u\,, v\rangle$ for pairs. We assume the set of variables is split in three pairwise disjoint sets: data-variables, prop-variables and spec-variables. We denote data-variables by xd, yd, prop-variables by xp, yp and spec-variables by xs, ys.
The *heart* of a pseudoterm is defined as follows: $\mathsf{heart}(u) = u$ if u is either a variable, a sort or a pair, $\mathsf{heart}(\Pi x{:}U.V) = \mathsf{heart}(\lambda x{:}U.V) = \mathsf{heart}(V\,U) = \mathsf{heart}(V)$.
The set of pseudoterms is split in three sets:
A *data-pseudoterm* is a pseudoterm whose heart is a data-variable or a data-sort. We denote data-pseudoterms by A, B, $a, b \ldots$.
A *prop-pseudoterm* is a pseudoterm whose heart is a prop-variable or a prop-sort. We denote prop-pseudoterms by P, Q, $p, q, \ldots$
A *spec-pseudoterm* is a pseudoterm whose heart is a spec-variable, a spec-sort or a pair. We denote spec-pseudoterms by S, T, $s, t, \ldots$.
We use the metavariables U, V, u, v ... for any pseudoterm.

We make the following conventions[1]:

1. In $\lambda x{:}U.V$ or $\Pi x{:}U.V$ we have that both x and U are data-pseudoterms, both are prop-pseudoterms or both are spec-pseudoterms. The same restriction applies to any typing context Γ: if $x{:}U \in \Gamma$ then both x and U are data-pseudoterms, both are prop-pseudoterms or both are spec-pseudoterms.
2. For all pseudoterms $\langle u\,, v\rangle$, we assume that u is a data-pseudoterm and v is a prop-pseudoterm.

3.3 Operational Semantics for Program Extraction

We define the reductions $\to_\sigma$ and $\to_\beta$ on pseudoterms. The first reduction, defined in [SS02], gives an operational semantics to program extraction. The essential property of the notion of specification is that it always computes to a pair. The reduction relation $\to_\sigma$ performs the task of splitting spec-pseudoterms into pairs. For the latter to hold, a spec-variable should also reduce to a pair. For this, from now on we assume that there exists an injective function Ψ such that $\Psi(xs) = \langle xd\,, xp\rangle$ for all spec-variables xs. The definition of $\to_\sigma$ is shown in Figure 2.

Splitting

$xs \to_\sigma \langle xd\,, xp\rangle$

$sType^n \to_\sigma \langle dType^n\,, \lambda xd{:}dType^n.(xd \to pType^n)\rangle$

Eliminating proofs from programs

$\Pi xp{:}P.A \to_\sigma A$ if $xp, xs \notin FV(A)$

$\lambda xp{:}P.a \to_\sigma a$ if $xp, xs \notin FV(a)$

$a\,p \to_\sigma a$

Currryfication

$\Pi xs{:}\langle A\,, P\rangle.U \to_\sigma \Pi xd{:}A.\Pi xp{:}(P\,xd).U$

$\lambda xs{:}\langle A\,, P\rangle.u \to_\sigma \lambda xd{:}A.\lambda xp{:}(P\,xd).u$

$u\,\langle a\,, p\rangle \to_\sigma u\,a\,p$

Distributivity

$\Pi x{:}U.\langle A\,, P\rangle \to_\sigma \langle \Pi x{:}U.A\,, \lambda f{:}\Pi x{:}U.A.\Pi x{:}U.(P\,(f\,x))\rangle$

$\lambda x{:}U.\langle a\,, p\rangle \to_\sigma \langle \lambda x{:}U.a\,, \lambda x{:}U.p\rangle$

$\langle a\,, p\rangle\,u \to_\sigma \langle a\,u\,, p\,u\rangle$

Fig. 2. Definition of σ-reduction

We denote by $\twoheadrightarrow_\sigma$ the reflexive transitive closure of $\to_\sigma$, and by $=_\sigma$ its reflexive, symmetric and transitive closure.
To avoid the problems mentioned in [RS02] we restrict β-reduction to σ-normal forms.

[1] See [SS02] where these restrictions are imposed by a grammar.

We define the reduction $\rightarrow_\beta$ as the least relation on σ-normal forms that contains the following rules and is closed under the compatibility rules:

$$(\lambda xd{:}A.u)\, a \rightarrow_\beta u[a/xd]$$
$$(\lambda xp{:}P.u)\, p \rightarrow_\beta u[p/xp]$$

The reduction relation $\rightarrow_{\beta\sigma}$ on pseudoterms is defined as $\rightarrow_\sigma \cup \rightarrow_\beta$ and $\twoheadrightarrow_{\beta\sigma}$ denotes the transtive closure of $\rightarrow_{\beta\sigma}$.

Remark 1. The rule $xs \rightarrow_\sigma \langle xd\,, xp\rangle$ deserves more explanations: the left-hand side is a variable of the Theory of Specifications (a spec-pseudoterm) which we rewrite to the pair of variables $\Psi(xs) = \langle xd\,, xp\rangle$. It is therefore a rule scheme, representing an infinite number of rules, one for each xs. The variables xs of the Theory of Specifications are seen as constants in the signature of the rewrite system.

The following notion of *completeness* is needed to restrict substitution (though we restrict β-reduction to σ-normal forms, substitution is performed in the Application rule). We also need a modified definition of *freshness* with respect to a typing context.

- The variable xd (resp. xp) is *complete* for a term u if $xs \notin FV(u)$. The variable xd (resp. xp) is *fresh* for a context Γ (Γ-fresh for short) if xs and xd (resp. xp) do not occur in Γ.
- The variable xs is *complete* for a term u if $xd, xp \notin FV(u)$. It is *fresh* for a context Γ if xs, xd and xp do not occur in Γ.

Whenever a substitution is performed $u[v/x]$ we require that the variable x is complete for u.

3.4 Typing Pairs

The Theory of Specifications is inductively defined by adding the rules in Figure 3 to the rules of Pure Type Systems when the specification is TS (see [SS02]).

The Theory of Specifications enjoys the following basic properties.

Lemma 3.1. (Classification)
If $\Gamma \vdash u{:}U$ then u, U are both data-pseudoterms, or both prop-pseudoterms, or both spec-pseudoterms.

Lemma 3.2. (Correctness)
If $\Gamma \vdash u{:}U$ then $\Gamma \vdash U{:}k$.

Theorem 3.3. (Interchange)
$\Gamma, xs{:}\langle A\,, P\rangle, \Delta \vdash u{:}U$ if and only if $\Gamma, xd{:}A, xp{:}(P\, xd), \Delta \vdash u{:}U$.

Pair Type
$$\frac{\Gamma \vdash A{:}dType^n \quad \Gamma \vdash P{:}A \to pType^n}{\Gamma \vdash \langle A\,, P\rangle{:}sType^n}$$

Pair Object
$$\frac{\Gamma \vdash a{:}A \quad \Gamma \vdash p{:}P\,a \quad \Gamma \vdash \langle A\,, P\rangle{:}sType^n}{\Gamma \vdash \langle a\,, p\rangle{:}\langle A\,, P\rangle}$$

Spec-variable
$$\frac{\Gamma \vdash xd{:}A \quad \Gamma \vdash xp{:}(P\,xd) \quad \Gamma \vdash \langle A\,, P\rangle{:}sType^n}{\Gamma \vdash xs{:}\langle A\,, P\rangle}\ xs \text{ is not in } \Gamma$$

Data-variable
$$\frac{\Gamma \vdash xs{:}\langle A\,, P\rangle}{\Gamma \vdash xd{:}A}\ xd \text{ is not in } \Gamma$$

Prop-variable
$$\frac{\Gamma \vdash xs{:}\langle A\,, P\rangle}{\Gamma \vdash xp{:}P\,xd}\ xp \text{ is not in } \Gamma$$

σ-conversion
$$\frac{\Gamma \vdash u{:}U \quad \Gamma \vdash U'{:}k}{\Gamma \vdash u{:}U'}\quad U{=}_\sigma U'$$

Fig. 3. Typing rules for pairs

This is proved by induction on the derivation. The only interesting case is the start rule, where we apply one of the rules for spec-variables, data-variables or prop-variables.

Theorem 3.4. (Inversion)

1. If $\Gamma \vdash \langle a\,, p\rangle{:}U$ then $\exists A, P$ such that $\Gamma \vdash a{:}A$, $\Gamma \vdash p{:}Pa$ and $V =_\sigma \langle A\,, P\rangle$, where $\Gamma \vdash A{:}dType^n$, $\Gamma \vdash P{:}A \to pType^n$, $\Gamma \vdash \langle A\,, P\rangle{:}sType^n$.
2. If $\Gamma \vdash \Pi x{:}U.V{:}k$ then $\Gamma, x{:}U \vdash V{:}k$ and $\Gamma \vdash U{:}k'$, where $(k', k, k) \in \mathcal{R}$.

The proof is by induction on the derivation.

3.5 An Example of Program Extraction

In order to develop an example of program extraction, we have first to illustrate how σ-reduction is extended to deal with natural numbers. This extension is presented in [SS02] in a general setting for the Calculus of Inductive Constructions. We recall an example.

The inductive data type Nat with constructors zero and suc has the following elimination rule:

$$\text{(natrec)}\ \frac{\begin{array}{l}\Gamma \vdash n : \mathsf{Nat}\\ \Gamma, x{:}\mathsf{Nat} \vdash U : k,\ k \in \{dType^0, pType^0, sType^0\}\\ \Gamma \vdash u_1 : U[n/x]\\ \Gamma \vdash u_2 : \Pi m{:}\mathsf{Nat}.(U[m/x] \to U[\mathsf{suc}\,m/x])\end{array}}{\Gamma \vdash (\mathsf{natrec}\ u_1\ u_2\ n) : U[n/x]}$$

For the sake of exposition we omitted the first argument of natrec that corresponds to the type U. The reduction for natrec is defined as follows:

$$\begin{array}{ll}(\mathsf{natrec}\ u_1\ u_2\ 0) & \to_\iota u_1\\ (\mathsf{natrec}\ u_1\ u_2\ \mathsf{suc}\ m) & \to_\iota (u_2\,m\,(\mathsf{natrec}\ u_1\ u_2\ m))\end{array}$$

In order to obtain the correct reduction for this operator when U is a specification, we must extend the definition of σ-reduction to include the following distributivity rule. In that rule we will use the following abbreviations:

$\hat{P} = \lambda n{:}Nat.(P\, n\, (\mathsf{natrec}\ A\ a_1\ a_2)n)$,
$\hat{p_2} = \lambda n{:}Nat.\lambda q{:}(\hat{P}\, n).(p_2\, n\, (\mathsf{natrec}\ a_1\ a_2\ n)\, q)$

(Distributivity of natrec over pairs)
$(\mathsf{natrec}\ \langle a_1\, , p_1\rangle\ \langle a_2\, , p_2\rangle\ n) \to_\sigma \langle (\mathsf{natrec}\ a_1\ a_2\ n)\, , (\mathsf{natrec}\ p_1\ \hat{p_2}\ n)\rangle$

Note that the predicate $\hat{P}$ is in fact the predicate P applied to the first component of the pair.
We are now ready to give an example of program extraction in the Theory of Specifications. We consider the specification stating that every natural number not equal to zero has a predecessor. Using the definition $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\, , P\rangle : sType^0$, this specification is written as follows:

$$S = \Pi n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$$

In first-order logic S would be written as $\forall n.\exists m.(n > 0) \to (n = \mathsf{suc}\, m)$.
We use the following abbreviations and assumptions:

- $A = \lambda n{:}\mathsf{Nat}.\mathsf{Nat}$,
- $P = \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$,
- $U = \lambda n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.(P\, n\, m)$,
- there are terms p_0, p_m such that $\vdash p_0 : P\, 0\, 0$ and $m : \mathsf{Nat} \vdash p_m : P\, (\mathsf{suc}\, m)\, m$,
- $q = \lambda m{:}\mathsf{Nat}.\lambda x{:}(U\ m).p_m$.

The following term is an inhabitant of the type S:

$$\begin{aligned} s = \lambda n : \mathsf{Nat}.\, (\mathsf{natrec}\ & \langle 0\, , p_0\rangle \\ & (\lambda m{:}\mathsf{Nat}.\lambda x{:}(U\ m).\langle m\, , p_m\rangle) \\ & n) \end{aligned}$$

Using σ-reduction, s is reduced to a pair. The first component is the extracted program, that is, the predecessor function, and the second component is the proof of its correctness.

$$s \twoheadrightarrow_\sigma \langle \underbrace{\lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ 0\, (\lambda m{:}\mathsf{Nat}.\lambda x d{:}\mathsf{Nat}.m)\, n}_{\mathsf{predecessor}}, \lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ p_0\, q\, n\rangle$$

4 Properties

In this section we prove some important meta-theoretical properties of the Theory of Specifications. Confluence has already been proved in [SS01]. Our main contribution is to prove that the formulation of the Theory of Specifications presented in this paper satisfies subject reduction and strong normalization.

Theorem 4.1. (Strong Normalization for σ)
The reduction relation $\rightarrow_\sigma$ is strongly normalizing.

This is proved by giving an interpretation function h that decreases with the reduction $\rightarrow_\sigma$, i.e. if $u \rightarrow_\sigma u'$ then $\mathsf{h}(u) > \mathsf{h}(u')$. See Definition A.1 in the appendix. We now give a proof of confluence of σ-reduction which is slightly different from the one presented in [SS01,SS02]. We use the following characterization of σ-normal forms.

Theorem 4.2. (Characterization of σ-normal forms)
Let a be a data-pseudoterm, p be a prop-pseudoterm and s be a spec-pseudoterm. Then a, p and s are in σ-normal form if and only if they can be defined by the following grammar:

$$A := xd \mid dk \mid \lambda xd{:}A.A \mid A\,A \mid \Pi xd{:}A.A$$
$$P := xp \mid pk \mid \lambda xd{:}A.P \mid \lambda xp{:}P.P \mid P\,A \mid P\,P \mid \Pi xd{:}A.P \mid \Pi xp{:}P.P$$
$$S := \langle A\,, P\rangle$$

Corollary 4.3.
The σ-normal forms are closed under β-reduction.

Theorem 4.4. (Confluence of σ and $\beta\sigma$)

1. $\rightarrow_\sigma$ is locally confluent.
2. $\rightarrow_\sigma$ is confluent.
3. $\rightarrow_{\beta\sigma}$ is confluent.

Proof: The first part is proved by inspecting the three critical pairs generated by the rules in the definition of σ-reduction. The second part follows by the previous lemma and strong normalization of $\rightarrow_\sigma$, using Newman's Lemma [New42]. The third part is a consequence of the confluence of σ (previous part) and β [SS01] using Corollary 4.3. Due to our restriction of β-reduction, in a $\beta\sigma$-reduction sequence the σ-steps are always performed before the β-steps. $\diamond$

The unique σ-normal form of a term u is denoted by $[\![u]\!]$. Note that the σ-normal form of s is always of the form $\langle a\,, p\rangle$ where a is a data-pseudoterm that does not contain prop-pseudoterms and p is a prop-pseudoterm.
The following theorem says that $[\![\ ]\!]$ is a "mapping" from the Theory of Specifications to the Verification Calculus. Derivation in the Verification Calculus is denoted by $\vdash_{\mathsf{VC}}$.

Theorem 4.5. (Soundness) [SS02]
Let $\Gamma \vdash u{:}U$.

1. If u is a data or prop-pseudoterm then $[\![\Gamma]\!] \vdash_{\mathsf{VC}} [\![u]\!]{:}[\![U]\!]$.

2. If u is a spec-pseudoterm then $[\![U]\!] = \langle A\,, P\rangle$, $[\![u]\!] = \langle a\,, p\rangle$, $[\![\Gamma]\!] \vdash_{\mathsf{VC}} a{:}A$ and $[\![\Gamma]\!] \vdash_{\mathsf{VC}} p{:}P\,a$.

The proof is by induction on the derivation.

Since $[\![\ ]\!]$ computes the normal form of $\rightarrow_\sigma$, and the Verification Calculus is β-normalizing, we have that the Theory of Specifications is $\rightarrow_{\beta\sigma}$-normalizing [SS02].
Since the function $[\![\ ]\!]$ is the identity for terms of the Verification Calculus, we have that the Theory of Specifications is conservative with respect to the Verification Calculus [SS02].
Using soundness and unicity of types in the Verification Calculus it is easy to prove unicity of types in the Theory of Specifications.

Lemma 4.6. (Unicity of Types) Let u be a term such that $\Gamma \vdash u{:}U$ and $\Gamma \vdash u{:}U'$.

1. If u is a data- or a prop-pseudoterm, then $U =_{\beta\sigma} U'$.
2. If u is a spec-pseudoterm, then $u =_{\beta\sigma} \langle a\,, p\rangle$, $U =_{\beta\sigma} \langle A\,, P\rangle$, and $U' =_{\beta\sigma} \langle A\,, P'\rangle$ such that $Pa =_{\beta\sigma} P'a$.

Corollary 4.7. (Unicity of Sorts) If $\Gamma \vdash U{:}k$ and $\Gamma \vdash U{:}k'$ then $k = k'$.

Theorem 4.8. (Strong Normalization for $\beta\sigma$)
The Theory of Specifications is $\beta\sigma$-strongly normalizing.

Proof: Suppose u is typable in the Theory of Specifications and there exists an infinite $\beta\sigma$-reduction sequence starting from u:

$$u = u_0 \rightarrow_{\beta\sigma} u_1 \rightarrow_{\beta\sigma} u_2 \ldots$$

Since σ is strongly normalizing, this reduction sequence should contain infinite β-steps. We consider the first term u_n in the sequence from which we perform a β-step. This step is performed on a σ-normal form due to our restriction of β-reduction. All the consecutive terms to u_n are in σ-normal forms because β-reduction does not create σ-redexes (Corollary 4.3). Hence we have an infinite β-reduction sequence starting from u_n. By soundness, either u_n is typable in the Verification Calculus or it is a pair whose components are typable in the Verification Calculus. This contradicts the fact that the Verification Calculus is β-strongly normalizing [Luo89]. ◇

The following lemma is needed in the proof of subject reduction for β.

Lemma 4.9. (Expansion of contexts)
Let Γ' be a valid context (i.e. there are v, V such that $\Gamma' \vdash v{:}V$).
If $\Gamma \vdash u{:}U$ and $\Gamma' \twoheadrightarrow_{\beta\sigma} \Gamma$ then $\Gamma' \vdash u{:}U$.

We prove the statement for only one step of β or σ-expansion by induction on the derivation.

Theorem 4.10. (Subject Reduction for β)
If $u \to_\beta u'$ and $\Gamma \vdash u{:}U$ then $\Gamma \vdash u'{:}U$.

Proof: The term u is in σ-normal form. By soundness and subject reduction for β in the Verification Calculus, we have that

1. If u is a data or prop-pseudoterm then $[\![\Gamma]\!] \vdash_{\mathsf{VC}} [\![u']\!]{:}[\![U]\!]$.
2. If u is a spec-pseudoterm then $[\![U]\!] = \langle A\,, P\rangle$, $[\![u]\!] = \langle a\,, p\rangle$, $[\![u']\!] = \langle a'\,, p'\rangle$, $[\![\Gamma]\!] \vdash_{\mathsf{VC}} a'{:}A$ and $[\![\Gamma]\!] \vdash_{\mathsf{VC}} p'{:}P\,a$, and therefore $[\![\Gamma]\!] \vdash_{\mathsf{VC}} u'{:}[\![U]\!]$.

Since the context Γ and the type U are correct, we have that $\Gamma \vdash u'{:}[\![U]\!]$ (by Lemma 4.9). By applying σ-conversion rule, we have $\Gamma \vdash u'{:}U$. ◇

From the equations $P \to A =_\sigma A =_\sigma (P \to Q) \to A$ we can type the term $(\lambda xp{:}P.xd\,(yp\,xp))\,yp$ in the context $yp : (P \to Q)$. This term is not in σ-normal form and it cannot be β-reduced[2].
In order to prove that the rules that eliminate proofs from programs preserve the typing, we need to prove that the following rule is admissible:

$$(\mathsf{Strengthening})\ \frac{\Gamma, xp{:}P, \Delta \vdash u{:}U}{\Gamma, \Delta \vdash u{:}U} \quad xp, xs \notin FV(\Delta) \cup FV(u) \cup FV(U)$$

The following statement implies the admissibility of the Strengthening rule.

Lemma 4.11. (Strengthening)

1. Let U be in σ-normal form. If $\Gamma, xp{:}P, \Delta \vdash U{:}k$ and $xp, xs \notin FV(\Delta) \cup FV(U)$ then $\Gamma, \Delta \vdash U{:}k$.
2. If $\Gamma, xp{:}P, \Delta \vdash u{:}U$ and $xp, xs \notin FV(\Delta) \cup FV(u)$ then $\Gamma, \Delta \vdash u{:}\mathsf{nf}_{\beta\sigma}(U)$.

The $\mathsf{nf}_{\beta\sigma}U$ denotes the $\beta\sigma$-normal form of U.

Proof: The first part is proved using soundness, strengthening in the Verification Calculus and expansion of contexts (Lemma 4.9). The second part is proved by induction on the derivation. The interesting cases are the abstraction rule and the application rule. In both cases we use the previous part, unicity of sorts, and inversion. ◇

Theorem 4.12. (Subject Reduction for σ)
If $u \to_\sigma u'$ and $\Gamma \vdash u{:}U$ then $\Gamma \vdash u'{:}U$.

[2] Otherwise it would β-reduce to a term containing the self-application $xd\,(yp\,yp)$ which is not typable (see [RS02] for a counterexample of strong normalization).

Proof: This is proved by induction on the structure of u. The only interesting case is when u is itself a σ-redex. We distinguish cases according to the σ-rule applied. We show some cases. In each case we proceed by induction on the type derivation for u, distinguishing cases according to the last rule applied.

1. $u \equiv xs{\to_\sigma}\langle xd\,, xp\rangle \equiv u'$.
   Start Rule. We derive the same type for u' using Data-variable, Prop-variable and Pair Object.
   Spec-variable. We use Pair Object.
   All the other cases follow directly by induction.
2. $u \equiv sType^n{\to_\sigma}\langle dType^n\,, \lambda xd{:}dType^n.(xd \to pType^n)\rangle \equiv u'$
   The only interesting case is the Axiom, all the other cases follow directly by induction.
   Axiom. Using Pair Type, it is sufficient to prove that $\Gamma \vdash dType^n{:}dType^{n+1}$ and $\Gamma \vdash \lambda xd{:}dType^n.(xd \to pType^n){:}dType^n \to pType^{n+1}$. The first is an axiom, and the second follows easily using the Abstraction rule.
3. $u \equiv \Pi xp{:}P.A{\to_\sigma}A \equiv u'$, if $xp, xs \notin A$.
   The only interesting case is Product, the others follow directly by induction.
   Product. The bound variable is a prop-variable, therefore we can eliminate the binder and use Strengthening.
4. $u \equiv \lambda xp{:}P.a{\to_\sigma}a \equiv u'$, if $xp, xs \notin A$.
   The only interesting case is Abstraction, the others follow directly by induction.
   Abstraction. We derive $\Gamma \vdash \lambda xp{:}P.a{:}\Pi xp{:}P.V \equiv U$ using $\Gamma, xp{:}P \vdash a{:}V$ and $\Gamma \vdash \Pi xp{:}P.V{:}k$. By soundness and σ-Conversion, $\Gamma, xp{:}P \vdash a{:}[\![V]\!]$. Using the characterization of σ-normal forms (see Theorem 4.2 in the appendix) and Strengthening we obtain $\Gamma \vdash a{:}[\![V]\!]$, and by σ-Conversion we derive $\Gamma \vdash a{:}U$.
5. $u \equiv a\,p{\to_\sigma}a \equiv u'$.
   The only interesting case is Application, the others follow directly by induction. The proof is similar to the previous case.
6. $u \equiv \Pi xs{:}\langle A\,, P\rangle.V{\to_\sigma}\Pi xd{:}A.\Pi xp{:}(P\,xd).V \equiv u'$.
   The only interesting case is Product, the others follow directly by induction.
   Product. We deduce $\Gamma \vdash \Pi xs{:}\langle A\,, P\rangle.V{:}U \equiv k_2$ from $\Gamma \vdash \langle A\,, P\rangle{:}k_1$ and $\Gamma, xs{:}\langle A\,, P\rangle \vdash V{:}k_2$. Using the Classification Lemma, $k_1 \equiv sType^n$, and we can derive $\Gamma \vdash A{:}dType^n$ and $\Gamma \vdash P{:}A \to pType^n$. We conclude using Product twice.
7. $u \equiv \lambda xs{:}\langle A\,, P\rangle.u{\to_\sigma}\lambda xd{:}A.\lambda xp{:}(P\,xd).u \equiv u'$.
   The only interesting case is Abstraction, the others follow directly by induction. We use the Interchange Theorem , and the previous part.
8. $u \equiv u\,\langle a\,, p\rangle{\to_\sigma}u\,a\,p \equiv u'$.
   The only interesting case is Application, the others follow directly by induction. Again, we use the Interchange Theorem and the following property: $V{\twoheadrightarrow_\sigma}V[\langle xd\,, xp\rangle/xs]$ using the σ-rule for spec-variables, for which we have already proved preservation of types. We conclude by using σ-Conversion and Application twice.
9. The Distributivity rules are proved by induction on the type derivation, using the Classification Lemma, Soundness, and Inversion.

$\diamond$

5 Further Work

A natural extension of the Theory of Specifications would be the introduction of explicit substitutions and explicit control of ressources (copying and erasing). We plan to define a version of the Theory of Specifications with explicit substitutions and ressource management, and study the interactions between β, σ and substitution in this framework. We hope that this would allow us to define an efficient strategy of reduction for the Theory of Specifications.
It is necessary to define a syntax directed set of rules for the Theory of Specifications for the type inference algorithm. We should remove rules like $\beta\sigma$-conversion which introduce ambiguities in the derivation. Proving the equivalence between the syntax-directed set of rules and the Theory of Specifications is not an easy task.

References

Abr96. J-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

Bar92. H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.

Bar99. Barras et al. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 1999.

BM90. R. Burstall and J. McKinna. Deliverables: An approach to program development in the calculus of constructions. In *Proceedings of the First Workshop on Logical Frameworks*, pages 113–121, 1990.

Luo89. Z. Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of LICS '89*, IEEE, pages 386–395. IEEE Computer Society Press, 1989.

Luo93. Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3:333–363, 1993.

New42. M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

PM89a. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Prog ramming Languages*, Austin, January 1989. ACM.

PM89b. C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, 1989.

Pol94. E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1994.

RS02. F. van Raamsdonk and P. Severi. Eliminating proofs from programs. In *Proceedings of Third International Workshop on Logical Frameworks and Meta-Languages*, volume 70.2 of *Electronic Notes in Theoretical Computer Science*, 2002.

SS01. P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1):61–87, 2001.

SS02. P. Severi and N. Szasz. Internal Program Extraction in the Calculus of Inductive Constructions. In *6th Argentinian Workshop in Theoretical Computer Science (WAIT'02), 31st JAIIO*, 2002.

Tea. Coq Development Team. The Coq proof assistant reference manual version 7.1 2001. URL: `http://pauillac.inria.fr/coq/doc/main.html`.

A Appendix

Definition A.1. (Interpretation function)
The function h from the set of pseudoterms into Nat is defined by induction as follows.

Variables.
$\mathsf{h}(xd) = 0$
$\mathsf{h}(xp) = 0$
$\mathsf{h}(xs) = 1$

Sorts.
$\mathsf{h}(dType^i) = 0$
$\mathsf{h}(pType^i) = 0$
$\mathsf{h}(sType^i) = 1$

Pairs.
$\mathsf{h}(\langle a\,, p\rangle) = \mathsf{h}(a) + \mathsf{h}(p)$

Products.
$\mathsf{h}(\Pi xd{:}A.B) = \mathsf{h}(A) + \mathsf{h}(B)$
$\mathsf{h}(\Pi xp{:}P.B) = \mathsf{h}(P) + \mathsf{h}(B) + 1$
$\mathsf{h}(\Pi xs{:}S.B) = \mathsf{h}(S) + \mathsf{h}(B) + 2$
$\mathsf{h}(\Pi xd{:}A.Q) = \mathsf{h}(A) + \mathsf{h}(Q)$
$\mathsf{h}(\Pi xp{:}P.Q) = \mathsf{h}(P) + \mathsf{h}(Q)$
$\mathsf{h}(\Pi xs{:}S.Q) = \mathsf{h}(S) + \mathsf{h}(Q) + 1$
$\mathsf{h}(\Pi xd{:}A.T) = 3 * \mathsf{h}(A) + 2 * \mathsf{h}(T) + 1$
$\mathsf{h}(\Pi xp{:}P.T) = 3 * \mathsf{h}(P) + 2 * \mathsf{h}(T) + 4$
$\mathsf{h}(\Pi xs{:}S.T) = 6 * \mathsf{h}(S) + 4 * \mathsf{h}(T) + 10$

Abstractions.
$\mathsf{h}(\lambda xd{:}A.a) = \mathsf{h}(A) + \mathsf{h}(a)$
$\mathsf{h}(\lambda xp{:}P.a) = \mathsf{h}(P) + \mathsf{h}(a) + 1$
$\mathsf{h}(\lambda xs{:}S.a) = \mathsf{h}(S) + \mathsf{h}(a) + 2$
$\mathsf{h}(\lambda xd{:}A.p) = \mathsf{h}(A) + \mathsf{h}(p)$
$\mathsf{h}(\lambda xp{:}P.p) = \mathsf{h}(P) + \mathsf{h}(p)$
$\mathsf{h}(\lambda xs{:}S.p) = \mathsf{h}(S) + \mathsf{h}(p) + 1$
$\mathsf{h}(\lambda xd{:}A.s) = 2 * \mathsf{h}(A) + \mathsf{h}(s) + 1$
$\mathsf{h}(\lambda xp{:}P.s) = 2 * \mathsf{h}(P) + \mathsf{h}(s) + 2$
$\mathsf{h}(\lambda xs{:}S.s) = 2 * \mathsf{h}(S) + \mathsf{h}(s) + 4$

Applications.
$\mathsf{h}(a\,b) = \mathsf{h}(a) + \mathsf{h}(b)$
$\mathsf{h}(a\,p) = \mathsf{h}(a) + \mathsf{h}(p) + 1$
$\mathsf{h}(a\,s) = \mathsf{h}(a) + \mathsf{h}(s) + 2$
$\mathsf{h}(p\,a) = \mathsf{h}(p) + \mathsf{h}(a)$
$\mathsf{h}(p\,q) = \mathsf{h}(p) + \mathsf{h}(q)$
$\mathsf{h}(p\,s) = \mathsf{h}(p) + \mathsf{h}(s) + 1$
$\mathsf{h}(s\,a) = 2 * \mathsf{h}(a) + \mathsf{h}(s) + 1$
$\mathsf{h}(s\,p) = 2 * \mathsf{h}(p) + \mathsf{h}(s) + 2$
$\mathsf{h}(s\,t) = 2 * \mathsf{h}(t) + \mathsf{h}(s) + 4$

Refinement of Higher-Order Logic Programs

Robert Colvin[1], Ian Hayes[2], David Hemer[1], and Paul Strooper[2]

[1] Software Verification Research Centre
{robert,hemer}@itee.uq.edu.au
[2] School of Information Technology and Electrical Engineering
University of Queensland, Brisbane, Australia, 4072
{ianh,pstroop}@itee.uq.edu.au

Abstract. A refinement calculus provides a method for transforming specifications to executable code, maintaining the correctness of the code with respect to its specification. In this paper we extend the refinement calculus for logic programs to include higher-order programming capabilities in specifications and programs, such as procedures as terms and lambda abstraction. We use a higher-order type and term system to describe programs, and provide a semantics for the higher-order language and refinement. The calculus is illustrated by refinement examples.

1 Introduction

The logic programming refinement calculus [5] provides a method for systematically deriving logic programs from formal specifications. It is based on: a *wide-spectrum language* [12] that can express both specifications and executable programs; a *refinement relation* that models the notion of correct implementation; and a collection of *refinement laws* providing the means to refine specifications to code in a stepwise fashion.

The wide-spectrum language includes assumptions and specification constructs, as well as a subset that corresponds to Horn clauses (code). The refinement relation is defined so that an implementation must produce the same set of solutions as the specification it refines, but it need do so only when the assumptions hold. There are refinement laws for manipulating assumptions and specifications, and for introducing code constructs. The decision of which refinement law to apply at each step is determined by the developer. The calculus could be used as the basis for a program synthesis system [3] that would allow the developer or refiner to obtain some degree of automation, depending on the problem and the synthesis scheme chosen.

In this paper we extend the refinement calculus for logic programs so that variables may range over procedures, i.e., procedures become terms in our language. We can then make procedure calls on variables that represent procedures, and pass procedures as parameters. We may also construct procedures anonymously (lambda abstraction). We achieve this by embedding the specification language in a type system developed by Nadathur and Miller [10] for λProlog [9]. The semantics of the language and refinement are then extended to include typed variables, and procedures become special types of functions in the term language.

The paper is structured as follows. In Sect. 2 we give an overview of the type and term system presented in [10]. In Sect. 3 we extend the type system with the type *Cmd*

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 126–143, 2003.

representing programs, and present our wide-spectrum language. We also give the intuition behind the refinement relation (the notion of implementation), and some example programs. In Sect. 4 we present some examples of refining higher-order procedures. In Sections 5 - 8 we present the semantics of refinement and our higher-order wide-spectrum language. In Sect. 9 we discuss problems with allowing equality on procedure-valued terms in the context of refinement and higher-order logic programing.

2 Type System

In this section we present a general introduction to the type system described by Nadathur and Miller [10] for λProlog. In Sect. 3 we discuss how the type system is used to represent commands in our wide-spectrum language. The system is based on Church's Simple Theory of Types. It includes base types as well as functional types, and allows abstraction and application as terms.

2.1 Types

We use the following definitions.

Name	Represents	Examples
$\mathcal{S}$	base types	$\mathbb{Z}, \mathbb{B}$
$\mathcal{C}$	type constructors	$list/1, set/1, \rightarrow /2$
Types	all allowed types	$\mathbb{Z}, list(\mathbb{Z}), \mathbb{Z} \rightarrow \mathbb{B}$

The set $\mathcal{S}$ includes all the base types, such as integers ($\mathbb{Z}$) and booleans ($\mathbb{B}$). The set $\mathcal{C}$ is the set of all type constructors, that is, functors which take types as arguments and return a type. For example, *list* is a constructor which takes a type, e.g., $\mathbb{Z}$, and returns a type representing a list of elements of $\mathbb{Z}$. Typically we only need to be concerned with the set *Types*. This includes every base type and every possible application of the type constructors. In particular, it includes every possible function from a type to a type, for instance $\mathbb{Z} \rightarrow list(\mathbb{Z})$, and $list(\mathbb{Z}) \rightarrow list(list(\mathbb{Z} \rightarrow \mathbb{Z})) \rightarrow \mathbb{B}$. The function type constructor '$\rightarrow$' associates to the right.

We assume a set of variables, *Var*, which have an associated type, and a set of constants (including functions) of given types. There is at least one variable and constant of each type.

We assume a base type *Pred*, representing first-order predicates. The operators for the type *Pred* include conjunction, disjunction, etc., and the existential and universal quantifiers. We assume a rich set of mathematical operators, including equality and arithmetic operators, is available in our predicate language. The semantics of predicates is discussed in more detail in Sect. 5.2.

2.2 Terms

We define a *term* in this language as follows.

1. a constant or variable of type σ is a term of type σ.

2. $(\lambda X{:}\,\sigma \bullet E)$, where X is a variable of type σ and E is a term of type τ, is a term of type $\sigma \to \tau$.
3. $F(E)$, where F is a term of type $\sigma \to \tau$ and E is a term of type σ, is a term of type τ.

Thus a term can be a variable or a constant, a lambda abstraction, or an application. For example, given the integers as constants of type $\mathbb{Z}$, the (constant) empty list of integers $[\,]_{\mathbb{Z}}$, and the constructor $[.\mid .]_{\mathbb{Z}}$ for lists of integers, we can write terms such as $[1 \mid [2 \mid [\,]_{\mathbb{Z}}]_{\mathbb{Z}}]_{\mathbb{Z}}$. In the rest of the paper we drop the subscript on the list constants and use the usual list notation for lists of any type, though in practice there are different constructors for each distinct list type. Note that badly typed function applications are not terms.

3 Wide-Spectrum Language

In our wide-spectrum language we can write both specifications as well as executable programs. This has the benefit of allowing stepwise refinement within a single notational framework. To make the use of higher-order constructs easier, we have embedded our language in the type system described in Sect. 2. We introduce a new base type, *Cmd*, representing all possible commands that may be constructed in our language. We describe the basic constructors of the language in the next section, then discuss how procedures are treated in the framework, allowing reasoning about higher-order constructs. Our base language is similar to that presented in [5], except that in the new type system, quantified variables and parameters to procedures must be typed.

3.1 Basic Constructs

A summary of the basic constructs of the language is shown in Fig. 1.

Cmd	Type		Example
$\{\,\}$	$\mathit{Pred} \to \mathit{Cmd}$	assumption	$\{X \neq 0\}$
$\langle\,\rangle$	$\mathit{Pred} \to \mathit{Cmd}$	specification	$\langle Y = 2 * X\rangle$
$\vee$	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	disjunction	$S \vee T$
$\wedge$	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	parallel conj.	$S \wedge T$
,	$\mathit{Cmd} \to \mathit{Cmd} \to \mathit{Cmd}$	sequential conj.	S, T
$\exists_\sigma$	$(\sigma \to \mathit{Cmd}) \to \mathit{Cmd}$	existential quant.	$(\exists X{:}\,\mathbb{Z} \bullet S)$
$\forall_\sigma$	$(\sigma \to \mathit{Cmd}) \to \mathit{Cmd}$	universal quant.	$(\forall X{:}\,\mathbb{Z} \bullet S)$

Fig. 1. Summary of wide-spectrum language

Specifications and Assumptions. A specification $\langle P\rangle$, where P is a predicate, represents a set of instantiations of the free variables of the program that satisfy P. For example, the specification $\langle X = 5 \vee X = 6\rangle$ represents the set of instantiations $\{5, 6\}$ for X.

An assumption $\{A\}$, where A is a predicate, allows us to state formally what a program fragment assumes about the context in which it is used. For example, some programs may require that an integer parameter be non-zero, expressed as $\{X \neq 0\}$.

Program Operators. The disjunction of two programs $(S \vee T)$ computes the union of the results of the two programs. There are two forms of conjunction: a parallel version $(S \wedge T)$, where S and T are evaluated independently and the intersection of their respective results is formed on completion; and a sequential form (S, T), where S is evaluated before T. In the sequential case, T may assume the context established by S.

Quantifiers. For brevity, the existential quantifier $(\exists_\sigma(\lambda X{:}\sigma \bullet S))$ will be written in the usual way, i.e., $(\exists X{:}\sigma \bullet S)$. It generalises disjunction, computing the union of the results of S for all possible values of X of type σ. Similarly, the universal quantifier $(\forall X{:}\sigma \bullet S)$ computes the intersection of the results of S for all possible values of X of type σ. Note that there are an infinite number of quantifiers, as there is an $\exists_\sigma$ and $\forall_\sigma$ for each type σ.

The following *Cmd* is an example of a program that can be constructed in our language.

$$\{X, Z \in \mathbb{Z} \wedge X \neq 0\}, \langle Y = Z \operatorname{div} X \rangle$$

The program assumes that the variables X and Z are bound to integers, and that X is non-zero, then establishes the relation that Y is the whole number division of Z by X.

3.2 Procedures

A procedure in the wide-spectrum language is a function whose result type is *Cmd*, and whose argument types are *not* of type *Cmd* or *Pred* (i.e., the constructors in Fig. 1 are not procedures). In addition, a procedure must be a closed term, that is, contain no free variables.

A summary of some relevant procedure-related constructs is given in Fig. 2. We discuss each below, and describe the method we use for procedure definition.

Syntax		Type
$pc(T)$ or $P(T)$	procedure call	*Cmd*
$(\lambda V{:}\mathit{list}(\mathbb{Z}) \bullet S)$	non-recursive proc.	$\mathit{list}(\mathbb{Z}) \rightarrow \mathit{Cmd}$
$\mu P \bullet (\lambda V{:}\mathbb{Z} \bullet \ldots P(X) \ldots)$	recursive proc.	$\mathbb{Z} \rightarrow \mathit{Cmd}$
$id \mathrel{\widehat{=}} proc$	procedure definition	

Fig. 2. Procedure-based constructs

Procedure Call. A procedure call is the application of a procedure to parameters, and is a term of type *Cmd*. Note that we allow application of procedure variables; i.e., if P is a procedure variable, $P(T)$ is a *Cmd* (variables are disallowed as functors in some versions of Prolog).

Non-recursive Procedures. A non-recursive procedure is a term of the form $(\lambda X{:}\sigma \bullet S)$, where S is a wide-spectrum program and X is a parameter to the procedure of type σ (a procedure may have multiple parameters, expressed in the usual way).

Recursive Procedures. We use the least fix-point operator μ to define the meaning of a recursive procedure. We use the following notation:

$$\mu P \bullet (\lambda X{:}\,\sigma \bullet \mathcal{C}(P))$$

The body of the procedure, $\mathcal{C}$, encodes zero or more recursive calls to P.

Procedure Definition. A procedure definition has the form $p \mathrel{\widehat{=}} proc$, where p is some name and *proc* is a procedure. For example, we may define a procedure *double* that doubles an integer:

$$double \mathrel{\widehat{=}} (\lambda N{:}\,\mathbb{Z}, N'{:}\,\mathbb{Z} \bullet \{N \in \mathbb{Z}\}, \langle N' = N * 2\rangle)$$

Note that the syntax for a procedure definition ($\widehat{=}$) just introduces a shorthand for the procedure itself; the names of the procedures are not semantic entities in our system. The type of *double* is $\mathbb{Z} \to (\mathbb{Z} \to Cmd)$.

We do not allow terms representing commands to appear inside predicates. This does not remove the ability to do any of the things we would normally like in our higher-order programming language – it is our wide-spectrum programming language that we are making higher-order, not the predicate language (allowing higher-order predicates, that is, predicate variables to range over predicates, is different from allowing predicate variables to range over commands). Allowing the wide-spectrum language constructs to appear inside predicates would unnecessarily complicate the predicate language and its semantics.

3.3 Refinement

To model the notion of implementation, we define the refinement operator, '$\sqsubseteq$'. We say a program S refines to a program T, written $S \sqsubseteq T$, if T terminates more often that S (w.r.t. its *assumptions*), and if T preserves the same effect on its free variables. A formal definition of refinement is presented in Sect. 7.

We present a number of derived refinement laws below. Each law represents a refinement (synthesis/transformation) that may be made. Where a law is divided into two parts divided by a horizontal line, the part above the line is the proof obligation that must be satisfied for the refinement below the line to be applied.

Law 1 *Equivalent specifications*

$$\frac{P \equiv Q}{\langle P\rangle \sqsubseteq \langle Q\rangle}$$

We can refine a specification by transforming its predicate under logical equivalence.

Law 2 *Weaken assumption*

$$\frac{A \Rrightarrow B}{\{A\} \sqsubseteq \{B\}}$$

We may weaken an assumption by transforming its predicate under implication.

Law 3 *List case analysis*

$$\begin{array}{l} \{L \in list(\sigma)\}, S \\ \sqsubseteq \{L \in list(\sigma)\}, ((\langle L = [\,]\rangle \wedge S) \vee \\ \qquad (\exists H{:}\,\sigma, T{:}\,list(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge S)) \end{array}$$

A program S which has an associated assumption that variable L is a list may be split into the cases where L is empty and non-empty. This law has no proof obligations.

Law 4 *Recursion introduction*
Where $(\sigma, \prec)$ *is a well founded set,*

$$\frac{\forall X{:}\,\sigma \bullet \big(\forall Y{:}\,\sigma \bullet \{Y \prec X\}, pc(Y) \sqsubseteq id(Y)\big) \Rightarrow pc(X) \sqsubseteq \mathcal{C}(id)}{pc \sqsubseteq (\mu\, id \bullet (\lambda X{:}\,\sigma \bullet C(id)))}$$

This law is similar to the one presented and proved in [5], and follows from the principle of well-founded induction.

In order to apply Law 4, we must prove that the predicate above the line holds. This is done by showing that $pc(X) \sqsubseteq \mathcal{C}(id)$, assuming that

$$(\forall Y{:}\,\sigma \bullet \{Y \prec X\}, pc(Y) \sqsubseteq id(Y)) \tag{1}$$

We call (1) the inductive hypothesis. It permits refinements to recursive calls as long as the parameter Y is less than X according to some well-founded relation $\prec$ on σ, ensuring that the recursion will terminate. The program $\mathcal{C}(id)$ is just some program that may involve calls on the procedure id. Note that if $pc \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet S)$ then $pc(X)$ is equivalent to the procedure's body, S.

The law is used in the following steps:

1. Focus on the body of pc, S.
2. Refine S, possibly using the inductive hypothesis to introduce a call to id.
3. Call this refined program $\mathcal{C}(id)$.
4. Then, the proof obligation for the law has been proved (by instantiating $\mathcal{C}$), and the original non-recursive procedure pc has been refined to the recursive procedure $(\mu\, id \bullet (\lambda X{:}\,\sigma \bullet C(id)))$.

The second step will in general be complex, involving user direction as with most program derivations. The other steps in the process are trivial, including the construction of the inductive hypothesis, which is determined by the syntactic form of the specification.

3.4 Example Specification and Implementation

Consider the following specification of the standard higher-order procedure *map*, that applies a procedure P to all the elements in a list L, returning the list L'.

Definition 1 *Map*

$$\begin{array}{l} map \mathrel{\widehat{=}} \lambda P{:}\,\sigma \to \tau \to Cmd, L{:}\,list(\sigma), L'{:}\,list(\tau) \bullet \\ \quad \{L \in list(\sigma)\}, \\ \quad \langle \#L = \#L' \rangle \wedge (\forall i{:}\,1..\#L \bullet P(L(i), L'(i))) \end{array}$$

Note the assumption that L is a list. The type of L is given by its type declaration in the parameter list, but to guarantee that L is instantiated (i.e., not unbound) the assumption must be included. This allows us to implement the procedure using recursion. For the well-founded ordering we use $<$ on the length of the list parameter L. The assumption $\{L \in \mathit{list}(\sigma)\}$ ensures L is bound; without the assumption, L could be unbound, and the recursion would not terminate.

We briefly outline the refinement of *map* below. We wish to refine it to a recursive procedure, and therefore use Law 4. Thus we must discharge the proof obligation, which involves refining the body of *map* assuming the following inductive hypothesis. We use m as the name of the recursive call.

$$\begin{array}{l}(\forall T: \mathit{list}(\sigma), T': \mathit{list}(\sigma), P: \sigma \to \tau \to \mathit{Cmd} \bullet \\ \quad \{\#T < \#L\}, \\ \quad \{T \in \mathit{list}(\sigma)\}, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#T \bullet P(T(i), T'(i))) \\ \quad \sqsubseteq m(P, T, T'))\end{array} \tag{2}$$

First we refine the body of *map* (Definition 1) by splitting into the cases where L is empty and non-empty (Law 3) and simplifying.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#L \bullet P(L(i), L'(i))))\end{array}$$

Now we split the universal quantification over i in the range $1..\#L$ into the cases where $i = 1$ and i is in the range $2..\#L$. We note that $L(1) = H$ and $L'(1) = H'$, and that indexing L in the range $2..\#L$ is equivalent to indexing its tail T in the range $1..\#T$ (and similarly for L' and T'). We also add some assumptions about T, which we do by noting that $L \in \mathit{list}(\sigma)$ and $L = [H \mid T]$.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad P(H, H') \land \\ \quad \{\#T < \#L\}, \\ \quad \{T \in \mathit{list}(\sigma)\}, \\ \quad \langle \#T = \#T' \rangle \land (\forall i: 1..\#T \bullet P(T(i), T'(i))))\end{array}$$

Note that the bottom three lines match the left side of the inductive hypothesis (2), and therefore we can use it to introduce a call to m.

$$\begin{array}{l}\{L \in \mathit{list}(\sigma)\}, \\ \langle L = [\,] \land L' = [\,] \rangle \lor \\ (\exists H: \sigma, H': \tau, T: \mathit{list}(\sigma), T': \mathit{list}(\tau) \bullet \\ \quad \langle L = [H \mid T] \land L' = [H' \mid T'] \rangle, \\ \quad P(H, H') \land m(P, T, T')\end{array}$$

The above refinement steps comprise the proof obligation for Law 4, and thus we may refine the original procedure *map* to:

$$\begin{array}{l}\mu\, m \bullet \lambda P{:}\,\sigma \to \tau \to \mathit{Cmd}, L{:}\,\mathit{list}(\sigma), L'{:}\,\mathit{list}(\tau) \bullet \\ \quad \langle L = [\,] \land L' = [\,]\rangle \lor \\ \quad (\exists H{:}\,\sigma, H'{:}\,\tau, T{:}\,\mathit{list}(\sigma), T'{:}\,\mathit{list}(\tau) \bullet \\ \qquad \langle L = [H \mid T] \land L' = [H' \mid T']\rangle, \\ \qquad P(H, H') \land m(P, T, T'))\end{array}$$

Now that we have an implementation for *map*, we want to be able to refine programs like

$$\{X \in \mathit{list}(\mathbb{Z})\}, \langle \#X = \#X'\rangle \land (\forall i{:}\,1..\#X \bullet \mathit{double}(X(i), X'(i)))$$

to $\mathit{map}(\mathit{double}, X, X')$. This refinement is trivial by folding, i.e., pattern matching with the specification of *map* (though in general, higher-order matching is non-trivial [2]).

More generally, we have the following refinement law.

Law 5 *Parameter application.* Given $pc \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet S)$ then

$$S[\tfrac{Y}{X}] \sqsubseteq pc(Y)$$

4 Higher-Order Refinement

In this section we provide a more complex example, in which we use some general algebraic properties to match a specification with a recursive procedure definition. Consider a procedure *foldR*, where a call $\mathit{foldR}(P, \mathit{Base})(L, \mathit{Result})$ applies procedure P, representing a binary operator, right-associatively to the list L, starting with base element *Base* (typically the identity of the binary operator represented by P), producing the answer *Result*. For example, assuming *plus* implements binary addition, i.e.,

$$\mathit{plus} \mathrel{\widehat{=}} (\lambda X, Y, Z{:}\,\mathbb{Z} \bullet \{X, Y \in \mathbb{Z}\}, \langle Z = X + Y\rangle)$$

$\mathit{foldR}(\mathit{plus}, 0)([1, 2, 3], X)$ would bind X to 6 (the result of 1+(2+(3+0))).

As a second example, given the definition

$$\mathit{snoc} \mathrel{\widehat{=}} (\lambda A{:}\,\sigma, B{:}\,\mathit{list}(\sigma), C{:}\,\mathit{list}(\sigma) \bullet \langle C = B \frown [A]\rangle)$$

a list may be reversed (inefficiently) using *foldR*:

$$\mathit{reverse}(R, R') \sqsubseteq \mathit{foldR}(\mathit{snoc}, [\,])(R, R')$$

or more succinctly,

$$\mathit{reverse} \sqsubseteq \mathit{foldR}(\mathit{snoc}, [\,])$$

This may be transformed to a more efficient version using a similar higher-order procedure *foldL*, that captures left-associativity, as shown in [14].

We define *foldR* as a procedure that takes a procedure and a base value and returns a recursive procedure.

$$\begin{array}{l} foldR \mathrel{\widehat{=}} (\lambda P{:}\,\sigma \to \tau \to \tau \to Cmd, Base{:}\,\tau \bullet \\ \quad (\mu fr \bullet \\ \quad\quad (\lambda L{:}\,list(\sigma), Result{:}\,\tau \bullet \\ \quad\quad\quad \{L \in list(\sigma)\}, \\ \quad\quad\quad (\langle L = [\,] \wedge Result = Base\rangle \vee \\ \quad\quad\quad (\exists H{:}\,\sigma, T{:}\,list(\sigma) \bullet \langle L = [H \mid T]\rangle, \\ \quad\quad\quad\quad (\exists R{:}\,\tau \bullet fr(T,R) \wedge P(H,R,Result))))))) \end{array}$$

We prove a general theorem for using *foldR* to implement relations that satisfy certain properties.

Theorem 1.
Given constants Q: $list(\sigma) \to \tau \to Pred$, B: τ, *and* P: $\sigma \to \tau \to \tau \to Pred$, *that satisfy:*

$$Q([\,], B) \tag{3}$$

$$Q([H \mid T], N) \Leftrightarrow (\exists R{:}\,\tau \bullet Q(T,R) \wedge P(H,R,N)) \tag{4}$$

and a procedure op that implements P, i.e.,

$$op \mathrel{\widehat{=}} (\lambda H{:}\,\sigma, R{:}\,\tau, N{:}\,\tau \bullet \langle P(H,R,N)\rangle) \tag{5}$$

then the following refinement holds:

$$(\lambda L{:}\,list(\sigma), N{:}\,\tau \bullet \{L \in list(\sigma)\}, \langle Q(L,R)\rangle) \sqsubseteq foldR(op, B)$$

Note that right-associativity is encoded into properties (3) and (4). In the summation example above, Q is the relation between a list of numbers and its sum, B is 0, and P is binary addition. In the reverse example, Q is the reverse relation between lists, B is [ ], and P is the predicate

$$(\lambda A{:}\,\sigma, B{:}\,list(\sigma), C{:}\,list(\sigma) \bullet C = B \frown [A])$$

which is implemented by the procedure *snoc*.

Proof. We use Law 4, and therefore assume the following inductive hypothesis.

$$(\forall T{:}\,list(\sigma), N'{:}\,\tau \bullet \{\#T < \#L\}, \{T \in list(\sigma)\}, \langle Q(T,N')\rangle \sqsubseteq fr(T,N')) \tag{6}$$

We begin the refinement of the body of the procedure on the left-hand side of the refinement.

$$\begin{array}{ll} & \{L \in list(\sigma)\}, \langle Q(L,N)\rangle \\ \sqsubseteq & \text{case analysis on } L \text{ using Law 3 and simplifying } Q([\,],N) \text{ using (3)} \\ & \{L \in list(\sigma)\}, \\ & (\langle L = [\,]\rangle \wedge \langle N = B\rangle \vee \\ & (\exists H{:}\,\sigma, T{:}\,list(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \langle Q(L,N)\rangle)) \\ \sqsubseteq & \text{focus on second disjunct} \end{array}$$

$$
\begin{array}{l}
1 \bullet\ (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \langle Q(L,N)\rangle) \\
\quad \sqsubseteq \text{ expand } Q \text{ using (4)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \\
\qquad\quad \langle(\exists R{:}\,\tau \bullet Q(T,R) \wedge P(H,R,N))\rangle) \\
\quad \sqsubseteq \text{ Lift quantification and conjunction} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle \wedge \\
\qquad\quad (\exists R{:}\,\tau \bullet \langle Q(T,R)\rangle \wedge \langle P(H,R,N)\rangle)) \\
\quad \sqsubseteq \text{ implement using } \mathit{op} \text{ from (5)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet \langle Q(T,R)\rangle \wedge \mathit{op}(H,R,N))) \\
\quad \sqsubseteq \text{ include assumptions about } T \text{ (from } L \in \mathit{list}(\sigma) \text{ and } L = [H \mid T]) \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet (\{\#T < \#L\}, \{T \in \mathit{list}(\sigma)\}, \langle Q(T,R)\rangle) \\
\qquad\qquad \wedge \mathit{op}(H,R,N))) \\
\quad \sqsubseteq \text{ introduce recursive call from (6)} \\
\qquad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad\quad (\exists R{:}\,\tau \bullet \mathit{fr}(T,R) \wedge \mathit{op}(H,R,N)))
\end{array}
$$

The above refinement steps complete the proof obligation for Law 4, resulting in the procedure:

$$
\begin{array}{l}
(\mu \mathit{fr} \bullet \lambda L{:}\,\mathit{list}(\sigma), N{:}\,\tau \bullet \\
\quad \{L \in \mathit{list}(\sigma)\}, \\
\quad (\langle L = [\,]\rangle \wedge \langle N = B\rangle \vee \\
\quad (\exists H{:}\,\sigma, T{:}\,\mathit{list}(\sigma) \bullet \langle L = [H \mid T]\rangle, \\
\qquad (\exists R{:}\,\tau \bullet \mathit{fr}(T,R) \wedge \mathit{op}(H,R,N)))))
\end{array}
$$

This is equivalent to $\mathit{foldR}(\mathit{op}, B)$. □

To apply this theorem to the summation example we need the definitional properties of *sum* corresponding to (3) and (4) respectively:

$$
\begin{array}{l}
\mathit{sum}([\,], 0) \\
\mathit{sum}([H \mid T], N) \Leftrightarrow (\exists R{:}\,\mathbb{Z} \bullet \mathit{sum}(T,R) \wedge N = H + R)
\end{array}
$$

and a procedure that implements binary addition

$$
\mathit{plus} \mathrel{\widehat{=}} (\lambda X, Y, Z{:}\,\mathbb{Z} \bullet \{X \in \mathbb{Z} \wedge Y \in \mathbb{Z}\}, \langle Z = X + Y\rangle)
$$

With Q, B and P as *sum*, 0 and binary addition respectively, we can apply Theorem 1 to deduce

$$
(\lambda L{:}\,\mathit{list}(\mathbb{Z}), N{:}\,\mathbb{Z} \bullet \{L \in \mathit{list}(\mathbb{Z})\}, \langle \mathit{sum}(L,R)\rangle) \sqsubseteq \mathit{foldR}(\mathit{plus}, 0)
$$

For the example above $\mathit{op}(H,R,N)$ represented a binary function from H and R to N. Now we consider an example where *op* represents a relation where there may be more than one value of N for a given pair of values for H and R. The permutation relation on lists may be formulated to correspond with the properties (3) and (4).

$$
\begin{array}{l}
\mathit{permutation}([\,], [\,]) \\
\mathit{permutation}([H \mid T], P) \Leftrightarrow \\
\quad (\exists R{:}\,\mathit{list}(\sigma) \bullet \mathit{permutation}(T,R) \wedge \mathit{interleave}(H,R,P))
\end{array}
$$

where

$$interleave(H, R, P) \Leftrightarrow \\ (\exists F, B: list(\sigma) \bullet R = F \frown B \wedge P = F \frown [H] \frown B)$$

Given some implementation *interleaveOp* of *interleave*, we can apply Theorem 1 to deduce

$$(\lambda L: list(\sigma), P: list(\sigma) \bullet \{L \in list(\sigma)\}, \langle permutation(L, P)\rangle) \\ \sqsubseteq foldR(interleaveOp, [\,])$$

and hence for appropriately typed L and P,

$$\{L \in list(\sigma)\}, \langle permutation(L, P)\rangle \sqsubseteq foldR(interleaveOp, [\,])(L, P)$$

5 Semantics

In this section we ascribe a meaning to terms of type *Cmd*, and define formally the refinement relation. The semantics we use are a modified version of earlier work [5,4]. We begin our formal treatment of the semantics by defining the domains over which our semantics of programs are given. We present an abridged version, which we modify to cope with changes for higher-order features. The main changes occur at the lowest level, where we define the set of values in our universe, and the evaluation of a term. Once these changes are in place, the semantics for the language constructs and refinement are similar to the semantics described in [5].

5.1 Variables and Values

For each type σ (from Sect. 2.1) we assume an associated set of values Val_σ. We assume that no value is a member of two different value sets, i.e.,

$$(\forall \sigma, \tau: Types \bullet \sigma \neq \tau \Rightarrow Val_\sigma \cap Val_\tau = \{\})$$

We define *Val* as the union of all sets Val_σ.

$$Val = \bigcup \{\sigma: Types \bullet Val_\sigma\}$$

Similarly we assume a unique set of variables Var_σ for each type σ, and define *Var* as the union of these sets.

5.2 Bindings, States and Predicates

A *binding* is a total function that maps every variable to a value of the correct type.

$$Bnd == \{b: Var \rightarrow Val \mid (\forall \sigma: Types \bullet (\forall V: Var_\sigma \bullet b(V) \in Val_\sigma))\}$$

Each binding corresponds to a single ground answer to a Prolog-like query. The mechanism for representing "unbound" variables is described below.

A *state* is a set of bindings:

$$State == \mathbb{P}\, Bnd$$

A state corresponds to our usual notion of a predicate with some free variables, which is true or false once provided with a binding for those variables, i.e., for a binding in the state. Given a predicate P, we write pred P to denote the set of bindings satisfying P. A completely unbound variable of type σ is represented by a (possibly infinite) state that has one binding to each element of Val_σ.

5.3 Term Evaluation

A term has a value relative to some binding. For a term T and binding b,

1. if T is a variable V, $\mathsf{eval}\ b\ T$ is simply the value for V in b, i.e., $b(V)$;
2. if T is a function application $F(X)$, $\mathsf{eval}\ b\ T$ is the evaluation of F applied to the evaluation of X, i.e., $(\mathsf{eval}\ b\ F)(\mathsf{eval}\ b\ X)$;
3. if T is a function $(\lambda X{:}\,\sigma \bullet e)$ where e is of type τ, $\mathsf{eval}\ b\ T$ is a function from elements of Val_σ to elements of Val_τ, defined as follows.

   $$(\lambda V{:}\, Val_\sigma \bullet eval\ (b \oplus \{X \mapsto V\})\ e)$$

   The binding $b \oplus \{X \mapsto V\}$ is the binding that maps all variables as b maps them, but with X mapped to V. In general the function $f \oplus g$, where f and g are functions, behaves as g for all elements in the domain of g, and as f for elements in the domain of f that are not in the domain of g.

6 Program Execution

In this section we model programs as functions from state to state. The notation $\{X{:}\,T \mid P \bullet E\}$ describes the set of values of the expression E, for each X of type T for which predicate P holds (when P is *true* we may omit it, i.e., $\{X{:}\,T \bullet E\}$). The notation $\{X{:}\,T \mid P\}$ filters the elements of T to leave only those that satisfy P.

6.1 Executions

We define the semantics of our language in terms of *executions*, which are mappings from initial states to final states. The mapping is partial because the program is only well-defined for those initial states that guarantee satisfaction of all the program's assumptions. Executions satisfy three healthiness properties, which restrict executions to model pure logic programs.

$$\begin{aligned} Exec == \{e{:}\, State \nrightarrow State \mid \\ \quad \operatorname{dom} e = \mathbb{P}\{b{:}\ Bnd \mid \{b\} \in \operatorname{dom} e\} \wedge & \quad (7) \\ \quad (\forall s{:}\operatorname{dom} e \bullet e(s) \subseteq s) \wedge & \quad (8) \\ \quad (\forall s{:}\operatorname{dom} e \bullet e(s) = \{b{:}\, s \mid e(\{b\}) = \{b\}\})\} & \quad (9) \end{aligned}$$

The notation '$\nrightarrow$' denotes a partial function. An execution, e, maps a state, s (a set of bindings or possible answers), to a new state, $e(s)$. Because execution of a command always constrains the set of possible answers, $e(s)$ must be a subset of s (property (8)). For a pure logic program, the new state can be constructed by considering whether each binding b in s is kept or not (property (9)); note that because of property (8), $e(\{b\})$ is either $\{b\}$ or $\{\}$. For a pure logic program, the domain of execution e can be determined by considering for each binding b whether or not $\{b\}$ is in the domain of e. The states in the domain of e are then all possible subsets of the set of all such bindings (property (7)). The healthiness properties are discussed in more detail in [5].

6.2 Semantic Function for Commands

We define the semantics of the commands in our language via a function that takes a command and returns the corresponding execution.

$$\mid \mathsf{exec}: \mathit{Cmd} \to \mathit{Exec}$$

The semantics of the basic commands (excluding recursion, which is treated in Section 8) is shown in Figure 3. In the remainder of this section, we explain the definitions. In [4], we show that all executions constructed using the definitions satisfy the healthiness properties of executions.

$$\begin{aligned}
&\mathsf{exec}(\langle P \rangle) = (\lambda\, s: \mathit{State} \bullet s \cap \overline{P}) \\
&\mathsf{exec}(\{A\}) = (\lambda\, s: \mathbb{P}\,\overline{A} \bullet s) \\
&\mathsf{exec}(c_1 \vee c_2) = \mathsf{exec}\, c_1 \cup \mathsf{exec}\, c_2 \\
&\mathsf{exec}(c_1 \wedge c_2) = \mathsf{exec}\, c_1 \cap \mathsf{exec}\, c_2 \\
&\mathsf{exec}(c_1 \,,\, c_2) = \mathsf{exec}\, c_1 \fatsemi \mathsf{exec}\, c_2 \\
&\mathsf{exec}(\exists\, V: \sigma \bullet c) = \mathsf{exists}_\sigma\, V (\mathsf{exec}\, c) \\
&\mathsf{exec}(\forall\, V: \sigma \bullet c) = \mathsf{forall}_\sigma\, V (\mathsf{exec}\, c)
\end{aligned}$$

Fig. 3. Execution semantics of basic commands

Specifications and Assumptions. The result of executing specification $\langle P \rangle$ consists of those bindings in s that satisfy P.

An assumption $\{A\}$ is defined for all states s such that A holds for all bindings in s; the result of executing assumption $\{A\}$ has no effect (the set of bindings remains unchanged).

Propositional Operators. Disjunction and parallel conjunction are defined as point-wise union and intersection of the corresponding executions. We present the definitions as Z axiomatic definitions [16]; the signatures are given above the line, and the definitions in the form of predicates are given below the line.

$$\_ \cap \_ : \mathit{Exec} \times \mathit{Exec} \to \mathit{Exec}$$
$$\_ \cup \_ : \mathit{Exec} \times \mathit{Exec} \to \mathit{Exec}$$
$$(e_1 \cap e_2) = (\lambda\, s : \mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 \bullet (e_1\ s) \cap (e_2\ s))$$
$$(e_1 \cup e_2) = (\lambda\, s : \mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 \bullet (e_1\ s) \cup (e_2\ s))$$

For a conjunction $(c_1 \wedge c_2)$, if a state s is mapped to s' by $\mathsf{exec}\, c_1$ and s is mapped to s'' by $\mathsf{exec}\, c_2$, then $\mathsf{exec}(c_1 \wedge c_2)$ maps s to $s' \cap s''$. Disjunction is similar, but gives the union of the resulting states instead of intersection.

Sequential conjunction (c_1 , c_2) is defined as function composition of the corresponding executions.

Quantifiers. For a type σ, variable V, and a state s, we define the state '$\mathsf{unbind}_\sigma\ V\ s$' as one whose bindings match those of s in every place except V, which is mapped to all values of type σ.

$$\mathsf{unbind}_\sigma : \mathit{Var}_\sigma \to \mathit{State} \to \mathit{State}$$
$$\mathsf{unbind}_\sigma\ V\ s = \{b : s;\ x : \mathit{Val}_\sigma \bullet b \oplus \{V \mapsto x\}\}$$

Execution of an existentially quantified command $(\exists\, V : \sigma \bullet c)$ from an initial state s is defined if executing c is defined in the state s', which is the same as s except that V is unbound. Since executions either keep or discard individual bindings (property (8)), the execution of $(\exists\, V : \sigma \bullet c)$ keeps a binding b if there exists some value x such that b, with V mapped to x, would be kept by the execution of c. A binding b is kept, therefore, if $e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)$, where e is the execution of c. We thus make the following definition of the existential quantifier for executions.

$$\mathsf{exists}_\sigma : \mathit{Var}_\sigma \to \mathit{Exec} \to \mathit{Exec}$$
$$\mathsf{exists}_\sigma\ V\ e = (\lambda\, s : \mathit{State} \mid \mathsf{unbind}_\sigma\ V\ s \in \mathrm{dom}\, e \bullet \{b : s \mid (\exists\, x : \mathit{Val}_\sigma \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})$$

Universal quantification behaves in a similar fashion, except that to retain a binding b, execution of e must retain $b \oplus \{V \mapsto x\}$ for all values x of type σ.

$$\mathsf{forall}\sigma : \mathit{Var}_\sigma \to \mathit{Exec} \to \mathit{Exec}$$
$$\mathsf{forall}\sigma\ V\ e = (\lambda\, s : \mathit{State} \mid \mathsf{unbind}_\sigma\ V\ s \in \mathrm{dom}\, e \bullet \{b : s \mid (\forall\, x : \mathit{Val}_\sigma \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})$$

7 Refinement

An execution e_1 is refined by an execution e_2 if and only if e_2 is defined wherever e_1 is and they agree on their outputs whenever both are defined. This is the usual "definedness" order on partial functions, as used, for example, by Manna [7]: it is simply defined by the subset relation of functions viewed as sets of pairs, i.e.,

$$e_1 \sqsubseteq_{\mathit{Exec}} e_2 \Leftrightarrow e_1 \subseteq e_2$$

Thus, if (s_1, s_2) is in e_1, then it must also be in e_2. Since both e_1 and e_2 are functions, there can be no other state associated with initial state s_1. This ensures that the set of answers is preserved by refinement, when the assumptions associated with e_1 hold. For some state $s' \notin \mathrm{dom}\, e_1$, (s', s'') may be in e_2 for any s''; in this case, the assumptions for e_1 do not hold (in s'), and thus e_2 may choose any answer (as long as the properties for executions are maintained).

Refinement is a pre-order — a reflexive and transitive relation — because subset is a pre-order on sets. Refinement for commands is defined in terms of refinement of executions.

$$c_1 \sqsubseteq c_2 \Leftrightarrow \mathsf{exec}(c_1) \sqsubseteq_{Exec} \mathsf{exec}(c_2)$$

Refinement equivalence ($\sqsupseteq\sqsubseteq$) is defined for *Cmd* and *Exec* as refinement in both directions.

8 Recursion

In this section we discuss the semantics of recursion. The treatment is different to our earlier approaches, as we do not have a separate environment that maps procedure names to procedures, because this may now be represented as part of the state. In addition, the move to higher-order programs means that some programs we can construct are not continuous, and thus to construct the fix point of a recursive procedure we need to go past the first infinite ordinal.

Consider the following function which takes as its argument a procedure and returns a procedure.

$$Ctx \mathrel{\widehat{=}} (\lambda P{:}\,\sigma \to Cmd \bullet (\lambda X{:}\,\sigma \bullet \mathcal{C}(P)))$$

$\mathcal{C}(P)$ is some *Cmd* in our language involving calls the procedure P.

We define $\mathbf{abort}_\sigma$ to be the least defined procedure in our language, i.e., it always aborts for any input of type σ.

$$\mathbf{abort}_\sigma \mathrel{\widehat{=}} (\lambda X{:}\,\sigma \bullet \{false\})$$

Now, as all contexts definable in our language are montonic [4], the least fix-point $\mu\, Ctx$ of Ctx exists [1], and furthermore, there exists an ordinal γ such that

$$\mu\, .Ctx = Ctx^{\gamma}(\mathbf{abort}_\sigma)$$

Given the definition of Ctx above, we write $\mu\, .Ctx$ as

$$\mu P \bullet (\lambda X{:}\,\sigma \bullet \mathcal{C}(P))$$

9 Procedures and Equality

While a logic programming language can provide a primitive equality relation for terms composed from basic types, extending equality to procedures is problematic. Many versions of Prolog represent procedures as terms, but implement equality as syntactic equality on the terms. However two syntactically different procedures may be semantically

equivalent, and hence such an implementation does not reflect the desired semantics. Unfortunately determining whether two syntactically different procedures are semantically equivalent is an undecidable problem, and hence equality cannot be implemented on procedures.

One special case that avoids this problem is an equality of the form $P = proc$, where the variable P is unbound. P can be bound to $proc$ and no comparison of procedures is required. However, such an equality is still problematic in the context of the refinement calculus, because if $proc$ is refined by $proc'$, one would like to be able to replace $proc$ by $proc'$ in any context. However $proc'$ is not equivalent to $proc$, it is a refinement.

This issue also complicates the notion of monotonicity of the language when procedure equality is included. For instance, the program

$$P = proc \wedge P = proc \tag{10}$$

should refine to just $P = proc$. However, if we have $proc \sqsubseteq proc'$ and $proc \sqsubseteq proc''$ for some $proc'$ and $proc''$ which do not refine each other, the following is also a valid refinement.

$$P = proc' \wedge P = proc'' \tag{11}$$

The two bindings for P are both semantically and syntactically different, and thus the program should fail – yet a program that fails would not typically be regarded as a valid refinement of (10).

The problems with such a construct prompted us to disallow the use of procedure equality. The only mechanism we allow to bind a procedure value to a variable is via parameter passing. In this case the formal parameter variable is guaranteed to be unbound, and gets bound once at the time the procedure is called.

Note that a procedure passed as a parameter can be replaced by a refinement. For example, a call of the form $q(proc, X)$, where $proc$ is a procedure parameter, can be replaced by $q(proc', X)$ if $proc \sqsubseteq proc'$. This is guaranteed by the monotonicity under refinement of our language.

By disallowing procedure equality, we limit the programs we can describe. We cannot have a procedure that accepts an unbound procedure variable as an input and on completion binds the variable to some procedure. For example we cannot write a procedure that succeeds and binds P to $test$ if its other parameter satisfies $test$.

$$testBind \mathrel{\widehat{=}} (\lambda P{:}\sigma \to Cmd, X{:}\sigma \bullet \{X \in \sigma\}, test(X) \wedge P = test)$$

However, we may still achieve the same effect on non-procedure variables without the procedure equality command. For instance, for a program $\mathcal{C}(P, Y)$ containing calls to procedure variable P and references to some set of non-procedure variables Y (possibly containing X), a program

$$testBind(P, X), \mathcal{C}(P, Y)$$

may be rewritten as

$$test(X), \mathcal{C}(test, Y)$$

In both cases the bindings for the variables in Y will be the same, but the latter program does not constrain P in any way.

10 Conclusions

In this paper we have presented a semantics for refinement of higher-order logic programs. We have used Nadathur and Miller's semantics for higher-order logic programs [10] for the basis of our semantics. They describe an implementation of the semantics, involving hereditary Harrop formulas, in the logic programming language λProlog [9]. The calculus we describe here would be suitable for developing programs for higher-order languages such as λProlog and Mercury [15].

In general, higher-order programming is more developed in the functional programming community. Some examples of development of functional programs include a refinement calculus for nondeterministic expressions [17] and development of extended ML programs [13]. Lacey, Richardson, and Smaill [6] use higher-order techniques to synthesise both first- and higher-order logic programs. They develop an automatic synthesiser in λClam, which is a proof system written in λProlog. They adopt a proof-planning approach to the problem. The refinement calculus approach we take is similar to Morgan's approach for imperative programs [8], though he does not explicitly mention higher-order programming. Naumann [11] uses a predicate transformer semantics to give a semantics for a higher-order imperative programming language, including a procedure binding construct.

In this paper we have extended earlier work [5] by including a type system in our wide-spectrum language. The semantics of dealing with procedures has also been simplified, by eliminating the need for a mapping from procedure identifiers to procedures (an environment). We presented some examples of using higher-order features in refinement, and discussed some of the problems associated with a mechanism for binding a procedure to a variable in a logic program development framework. We have distinguished higher-order programming, where variables may take the value of procedures, from meta-programming, where variables may take the value of any command in our language, e.g., a procedure call rather than a procedure itself. Meta-programming by this definition is not dealt with in this paper – this is an avenue for future work.

References

1. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
2. Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.
3. Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19,20:321–350, 1994. Special Issue: Ten Years of Logic Programming.
4. I. Hayes, R. Nickson, P. Strooper, and R. Colvin. A declarative semantics for logic program refinement. Technical Report 00-30, Software Verification Research Centre, The University of Queensland, 2000.
5. I. J. Hayes, R. Colvin, D. Hemer, R. Nickson, and P. A. Strooper. A refinement calculus for logic programs. *Theory and Practice of Logic Programming*, 2(4–5):425–460, July–September 2002.
6. David Lacey, Julian Richardson, and Alan Smaill. Logic program synthesis in a higher-order setting. In John W. Lloyd et al., editor, *Computational Logic 2000*, volume 1861 of *LNAI*, pages 87–100. Springer-Verlag, 2000.

7. Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
8. Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
9. G. Nadathur and D. Miller. An overview of Lambda-PROLOG. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, 1988.
10. G. Nadathur and D. Miller. Higher-order logic programming. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Clarendon Press, Oxford, 1998.
11. D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, September 2001.
12. H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
13. D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Springer Workshops in Computing, pages 99–130. Springer, 1990.
14. S. Seres and M. Spivey. Higher-order transformation of logic programs. In K.-K. Lau, editor, *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 2000)*, volume 2042 of *LNCS*, pages 57–68. Springer-Verlag, 2000.
15. Z. Somogyi, F.J. Henderson, and T.C. Conway. Mercury, an efficient purely declarative logic programming language. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 499–512, Glenelg, South Australia, 1995. Australian Computer Science Communications.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
17. Nigel Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, Department of Computer Science, University of Queensland, 1994.

A Generic Program for Minimal Subsets with Applications

Rudolf Berghammer

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel, Germany

Abstract. We formally develop a generic program which computes a minimal subset satisfying a certain property of a given set. To improve the efficiency of instantiations, refinements are investigated. Finally, instantiations are presented which correspond to the solution of well-known graph-theoretic problems and some further applications are sketched.

1 Introduction

Besides the computation of minimum resp. maximum subsets (with respect to size) of a given universe U the computation of minimal resp. maximal subsets (with respect to inclusion) of U is an important optimization problem, too. On the one hand, a given problem often directly demands a minimal resp. maximal subset. Examples are spanning forests and vertex bases of graphs. In the first case maximal cycle-free (or, equivalently, minimal connected) sets of edges are searched for and in the second case the task is to compute a minimal set of vertices from which all other vertices can be reached. Furthermore, minimal resp. maximal subsets are frequently used as starting points for the computation of minimum resp. maximum subsets in order to accelerate the latter. Matchings (these are sets M of edges of graphs such that for all vertices x at most one edge of M is incident on x) are a good example. To compute a maximum matching, normally a maximal matching is first computed in a very simple and efficient way. Only then the relatively large-scaled process of the iterative search for so-called augmenting alternating chains to receive a maximum matching (introduced in [5]) is started. Finally, minimal resp. maximal subsets also are used as approximations if an efficient computation of the respective minimum and maximum subsets seems to be impossible because of complexity-theoretic reasons. This might be the case for transitive reductions of graphs g, which are minimal subsets of the edges that maintain all reachability relations between vertices of g. The original interest is laid on smallest subsets of the edges which lead to the same reflexive-transitive closure as g. The subgraphs of g induced by such subsets are called minimum equivalent digraphs. Obviously, the interest is frequently limited to transitive reductions only because of the NP-completeness of the minimum equivalent digraph problem.

Since minimal and maximal subsets can be treated in a dual way, in this article we focus upon the first case. Similar to [17], first we formally develop in

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 144–157, 2003.

Section 2 through the invariant technique a generic program which computes for a given finite set U and a specific predicate $\mathcal{P}$ on the powerset 2^U a minimal subset of U for which $\mathcal{P}$ holds. In the same section we also have a look at refinements of this program, which allow in many cases to improve the efficiency of a concrete instantiation. Then Section 3 demonstrates such instantiations to solve two graph-theoretic problems, viz. the approximation of a minimum vertex cover and the computation of a transitive reduction resp. the approximation of a minimum equivalent digraph. We have implemented these algorithms since we believe that for a final assessment of their practical use there is a definite need to supplement the theoretical worst-case analysis with experiments. Some experimental results are presented in Section 3. They also contain a comparison with two algorithms known from the literature which, finally, leads to their improvement in practice. Section 4 contains some concluding remarks.

2 Generic Computation of Minimal Subsets

Let U be a finite set and $\mathcal{P}$ a predicate on its powerset 2^U. We assume that $\mathcal{P}$ is *upwards-closed*, which means that for all sets $X, Y \in 2^U$

$$X \subseteq Y \wedge \mathcal{P}(X) \implies \mathcal{P}(Y)\,. \tag{1}$$

Note that (1) implies the negation $\neg\mathcal{P}$ of $\mathcal{P}$ to be *downwards-closed*, i.e., for all sets $X, Y \in 2^U$ if $Y \supseteq X$ and $\mathcal{P}(Y)$ is false, then also $\mathcal{P}(X)$ is false.

2.1 The Program Development

Assuming property (1), in the following we shall systematically derive a generic imperative program that computes – for U as its input – a minimal subset of U that satisfies the predicate $\mathcal{P}$. If we use the variable A of type 2^U as the program's output, then a formal specification of the requirements on A is given by the postcondition

$$\mathsf{post}(A) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge \forall\, X \in 2^A : \mathcal{P}(X) \to X = A\,.$$

For the derivation of the program we combine the invariant technique with set-theoretic and logical calculations. Doing so, for a subset $X \subseteq U$ and an element $x \in U$ we write $X \ominus x$ instead of $X \setminus \{x\}$ and $X \oplus x$ instead of $X \cup \{x\}$ to enhance readability. Heading towards a program with an initialization followed by a while-loop, the formal derivation is carried out in three stages.

First we have to develop a loop invariant. Here we follow the most commonly used technique of generalizing the postcondition (for details and many examples we refer to the textbooks [4, 6]). The corresponding calculation introduces a new variable B of type 2^U and looks as follows:

$$\begin{array}{ll}
 & \mathsf{post}(A) \\
\Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, X \in 2^A : \mathcal{P}(X) \to X = A \\
\Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, X \in 2^A : X \neq A \to \neg\mathcal{P}(X) \\
\Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, x \in A : \neg\mathcal{P}(A \ominus x) \\
\Longleftarrow & \mathcal{P}(A) \wedge B = \emptyset \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\,.
\end{array}$$

Only the direction "$\Longleftarrow$" of the third step is non-trivial and needs an explanation: If $X \in 2^A$ and $X \neq A$, then there exists $x \in A$ such that $X \subseteq A \ominus x$. Hence $\neg\mathcal{P}(A \ominus x)$ implies $\neg\mathcal{P}(X)$ since $\neg\mathcal{P}$ is downwards-closed due to (1). Guided by the above implication, we now define

$$\mathsf{inv}(A,B) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)$$

as loop invariant and choose $B = \emptyset$ as exit condition of the while-loop. In Pascal-like program notation, hence we have the following outline:

$$\begin{array}{l} \ldots \\ \{\ \mathsf{inv}(A,B)\ \} \\ \textbf{while}\ B \neq \emptyset\ \textbf{do}\ \cdots\ \textbf{od} \\ \{\ \mathsf{post}(A)\ \}\,. \end{array}$$

In the second stage of the program development we now have to consider the initialization of the variables A and B. We do this in combination with the choice of a suitable precondition, i.e., a requirement on the program's input U which is sufficiently general and implies additionally that the initialization establishes the loop invariant. Since we want to compute a subset of U that satisfies $\mathcal{P}$, it seems to be reasonable to demand that there exists such a subset. Due to assumption (1) this property is equivalent to $\mathcal{P}(U)$ being true. Hence we choose

$$\mathsf{pre}(U) \;:\Longleftrightarrow\; \mathcal{P}(U)$$

as precondition. This works and yields an initialization which assigns U to both variables A and B. Here is the formal justification:

$$\begin{array}{ll} & \mathsf{pre}(U) \\ \Longleftrightarrow & \mathcal{P}(U) \wedge \forall\, x \in \emptyset : \neg\mathcal{P}(U \ominus x) \\ \Longleftrightarrow & \mathcal{P}(U) \wedge U \subseteq U \wedge \forall\, x \in U \setminus U : \neg\mathcal{P}(U \ominus x) \\ \Longleftrightarrow & \mathsf{inv}(U,U)\,. \end{array}$$

Having achieved an initialization, in the last stage of the program derivation we must elaborate a loop body which maintains the invariant and ensures termination of the loop. To this end we suppose that the value of B is non-empty and b is an arbitrary element contained in it. We consider two cases. First we assume $\mathcal{P}(A \ominus b)$ to be true. Then

$$\begin{array}{ll} & \mathsf{inv}(A,B) \\ \Longleftrightarrow & \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \Longrightarrow & \mathcal{P}(A) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \Longrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in (A \setminus B) \ominus b : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longleftrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in (A \ominus b) \setminus (B \ominus b) : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longleftrightarrow & \mathsf{inv}(A \ominus b, B \ominus b)\,. \end{array}$$

This calculation uses in the third step that $\mathcal{P}(A \ominus b)$ is true and the predicate $\neg\mathcal{P}$ is downwards-closed. The remaining steps apply only basic set theory and

logic and the definition of the invariant. Now we deal with the case of $\mathcal{P}(A \ominus b)$ being false. Here we obtain

$$\begin{array}{ll} & \mathsf{inv}(A, B) \\ \iff & \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \implies & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in (A \setminus B) \oplus b : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in A \setminus (B \ominus b) : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathsf{inv}(A, B \ominus b)\,, \end{array}$$

where the assumption on $\mathcal{P}(A \ominus b)$ is used again in the third step and basic set theory and the definition of the invariant are applied otherwise.

In view of the just derived implications the invariant is maintained if we change in each run through the while-loop the value of B into $B \ominus b$ and, provided $\mathcal{P}(A \ominus b)$ holds, the value of A into $A \ominus b$. This can be easily achieved by a conditional and leads to the following greedy-like completion of the above program outline, where the statement $b :\in B$, known from the refinement calculus (cf. [15]), non-deterministically assigns some member of the value of B to b:

$$\begin{array}{l} \{\ \mathsf{pre}(U)\ \} \\ A, B := U, U; \\ \{\ \mathsf{inv}(A, B)\ \} \\ \textbf{while } B \neq \emptyset \textbf{ do} \\ \quad b :\in B; \\ \quad \textbf{if } \mathcal{P}(A \ominus b) \textbf{ then } A, B := A \ominus b, B \ominus b \\ \qquad\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\ \{\ \mathsf{post}(A)\ \}\,. \end{array} \qquad (\mathrm{MIN}_1)$$

Termination of this program is obvious since its input U is finite and, therefore, the initial value of B is finite, too. The size of the value of B, however, is strictly decreased by every run through the while-loop.

2.2 Refinements

The running time of an instantiation of the generic program (MIN_1) mainly depends on two facts, viz. the costs for the evaluation of the predicate $\mathcal{P}$ and the number of runs through the while-loop. Guided by these simple observations we now refine the program in such a way that in many cases the efficiency is – partly considerably – improved.

First we consider the evaluation of $\mathcal{P}$. From analyzing many instances of (MIN_1), e.g., those mentioned in the introduction, the additional assumption on $\mathcal{P}$ to be *decremental* arose. This property means that there exists a predicate $\mathcal{Q}$ on $2^U \times U$ such that for all non-empty sets $X \in 2^U$ and elements $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge \mathcal{Q}(X, x)\,. \qquad (2)$$

Since $\mathcal{P}(A)$ is part of the loop invariant $\mathsf{inv}(A, B)$, from (2) we get that the condition $\mathcal{P}(A \ominus b)$ of the conditional statement of (MIN_1) can be replaced by $\mathcal{Q}(A, b)$. Doing so, we obtain the following refinement:

$$
\begin{array}{l}
\{\ \mathsf{pre}(U)\ \} \\
A, B := U, U; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(A, b) \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ \mathsf{post}(A)\ \}\,.
\end{array}
\qquad (\mathrm{MIN}_2)
$$

This program is more efficient than (MIN_1) if the evaluation of $\mathcal{Q}(X, x)$ is less expensive than that of $\mathcal{P}(X \ominus x)$, which proved to be true in all instances we have investigated.

With regard to the second observation it is obvious that the number of runs through the while-loop of (MIN_2) equals the size of the input set U. Heading towards increasing efficiency, it therefore seems to be a good idea to refine additionally the program's initialization by an efficient precomputation phase yielding a subset of U for which $\mathcal{P}$ holds and whose size is much smaller than that of U. This prompts us to introduce a new variable S of type 2^U to hold the value of the precomputation. Obviously $\mathcal{P}(S)$ and $\mathsf{inv}(S, S)$ are equivalent. Hence we can weaken the previous precondition to *true* and demand $\mathcal{P}(S)$ as postcondition of the precomputation. Altogether, we obtain the following correct refinement of the generic program (MIN_2) in which the pseudo code $\gg$ precomputation $\ll$ denotes the precomputation phase:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
\gg \text{precomputation} \ll; \\
\{\ \mathcal{P}(S)\ \} \\
A, B := S, S; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(A, b) \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ \mathsf{post}(A)\ \}\,.
\end{array}
\qquad (\mathrm{MIN}_3)
$$

If we compare this program to the first solution[1] of [17], then, besides the fact that we compute minimal subsets and [17] maximal ones, there are two main differences. First in [17] no precomputation is used which, however, is very profitable in many applications. Second in [17] an execution of the body of the while-loop enlarges the present solution but makes a second set – the candidates for an enlargement – smaller. This leads to an equality test on sets as condition of the program's while-loop which is more costly than our simple emptiness test (or "being the universe" test in the dual program for computing maximal subsets satisfying a downwards-closed predicate).

[1] In [17] data refinement is used to develop from it a further generic program, called second solution. This refinement assumes very specific properties of the used predicate and is tailored to the two applications the article deals with.

3 Applications

In the following we instantiate the generic programs (MIN_2) and (MIN_3) to solve two graph-theoretic problems and sketch some further applications. We assume the reader to be familiar with the basic notations of graph theory; otherwise see, for instance, [3].

3.1 Vertex Cover

Let an undirected graph $g = (V, E)$ with finite sets V of vertices and E of edges be given, where each edge is a set $\{x, y\}$ of distinct vertices x and y. We suppose that in our programming language the two operations

$$\mathit{neigh}(x) = \{y \in V : \{x, y\} \in E\} \qquad \mathit{inc}(e) = \{f \in E : e \cap f \neq \emptyset\}$$

are implemented. They compute for $x \in V$ the set $\mathit{neigh}(x)$ of all neighbours resp. for $e \in E$ the set $\mathit{inc}(e)$ of all edges incident on it.

A subset C of V is called a *vertex cover* of the graph g if $C \cap e \neq \emptyset$ for all $e \in E$, i.e., if each edge of g is incident on at least one vertex of C. It is obvious that the predicate $\mathcal{P}$ on the powerset 2^V, defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; \forall\, e \in E : X \cap e \neq \emptyset\,, \tag{3}$$

is upwards-closed and $\mathcal{P}(V)$ holds. Furthermore, from (3) we immediately obtain for all non-empty sets $X \in 2^V$ of vertices and vertices $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge \mathit{neigh}(x) \subseteq X\,. \tag{4}$$

Now we choose the inclusion $\mathit{neigh}(x) \subseteq X$ of (4) as predicate $\mathcal{Q}(X, x)$. Then $\mathcal{P}$ is decremental in the sense of (2) and we arrive at the following instantiation of (MIN_2) for computing a minimal vertex cover of g:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := V, V; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathit{neigh}(b) \subseteq A \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ minimal vertex cover of } g\ \}\,.
\end{array} \tag{VC_1}$$

During the execution of the program (VC_1) the inclusion $\mathit{neigh}(b) \subseteq A$ is tested exactly once for each vertex of g. Hence the total running time is $O(|E|)$ if g is represented by an adjacency list (see e.g., [3]) and a set is represented by a Boolean array and its size.

If we want to compute a minimum vertex cover instead of a minimal one, then we are concerned with an NP-complete problem. See, for instance, [3] for a

proof of this fact. In this textbook also a simple $O(|E|)$ approximation algorithm for the minimum vertex cover problem is presented and attributed to Gavril and Yannakakis. If this algorithm is expressed as an annotated while-program, we obtain the following code:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
F, S := E, \emptyset; \\
\{\ S \text{ vertex cover of } g_F = (V, E \setminus F) \wedge \mathsf{inv}'(F, S)\ \} \\
\textbf{while } F \neq \emptyset \textbf{ do} \qquad\qquad (\mathrm{VC}_2) \\
\quad e :\in F; \\
\quad F, S := F \setminus \mathit{inc}(e), S \cup e \ \ \textbf{od} \\
\{\ S \text{ vertex cover of } g \wedge |S| \leq 2c_{opt}\ \}\,.
\end{array}
$$

Here c_{opt} denotes the *size of a minimum vertex cover* of g. The second conjunct $\mathsf{inv}'(F, S)$ of the loop invariant of (VC_2) is defined as

$$
\mathsf{inv}'(F, S) \ :\Longleftrightarrow\ \begin{cases} \exists\, M \in 2^E :\ M \text{ matching of } g \ \wedge \\ \qquad\qquad\quad |S| \leq 2|M| \ \wedge \\ \qquad\qquad\quad \forall\, f \in F, m \in M : f \cap m = \emptyset \end{cases}
$$

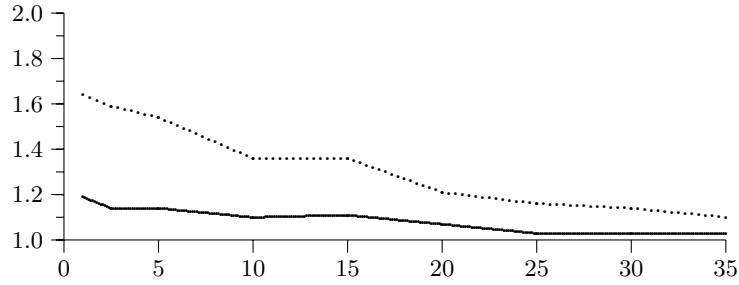
and formalizes the idea that the set of all edges that were picked by the statement $e :\in F$ during the execution of program (VC_2) forms a matching M of g such that $|S| \leq 2|M|$ and edges of F and M have no vertex in common. It is not hard to see that the loop invariant of (VC_2) is established by the initialization and maintained by the body of the while-loop. (In doing so, the third part of the body of $\mathsf{inv}'(F, S)$ is necessary to prove that for a matching M of g and an edge $e \in F$ also $M \oplus e$ is a matching of g.) Hence if the program terminates, then the first part of its postcondition trivially holds. The remaining part $|S| \leq 2c_{opt}$ follows from the fact that each minimum vertex cover C^* must include at least one vertex of any edge of any matching. This implies $|M| \leq |C^*| = c_{opt}$ from which we get the desired result $|S| \leq 2c_{opt}$ due to $|S| \leq 2|M|$.

If we compare the two programs (VC_1) and (VC_2) wrt. the usual worst case approximation bound (see, for example, [8] for details), then program (VC_2) is superior. For any input graph $g = (V, E)$ it computes a vertex cover whose size is guaranteed to be no greater than twice the minimum size c_{opt} of a vertex cover of g. In contrast with this, for a "star-like" input graph g the result of program (VC_1) may contain $(|V|-1)c_{opt}$ vertices if the non-deterministic statement $b :\in B$ unfortunately never selects the star's centre.

But if we compare the quality of the results produced for randomly generated inputs, which is a well-established mode to assess algorithms in practice, then the situation changes and program (VC_1) becomes superior. We have formulated both programs in the programming language of the relation-algebraic prototyping tool RelView (see e.g., [1, 2]) and performed numerous experiments. In each experiment we fixed a number n of vertices, generated random graphs with n vertices (more exactly: their symmetric and irreflexive adjacency relations), where we varied the number of edges from 1% to 35%, and compared the results of (VC_1) and (VC_2) with the sizes of the minimum vertex covers. Due to the

very efficient ROBDD-implementation of relations (see [12]), RelView allows treating arbitrary graphs with up to 150 vertices. For dense graphs even larger vertex sets are possible. E.g., on a Sun Fire-280R workstation with 750 MHz and 4 GByte main memory running Solaris 8 the enumeration of all minimum vertex covers for randomly generated graphs with 250 vertices takes approx. 3000 sec. (200 sec. resp. 50 sec.) in the case of 65% (75% resp. 85%) of all edges.

The following figure shows one result for $n = 100$. On its x-axis the graphs' density is listed and the ratio of the results' size and c_{opt} is listed on the y-axis. The dotted curve belongs to (VC_2) and the other one to (VC_1). Both curves have been obtained by considering 180 graphs and computing the ratios' arithmetic mean values.



All other experiments we have performed showed the same tendency. Based on these results, hence it seems to be reasonable to combine both programs, i.e., to instantiate the generic program (MIN_3) instead of (MIN_2) and to choose Gavril and Yannakakis' algorithm (VC_2) as precomputation phase. Doing so, we arrive at the following result:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
F, S := E, \emptyset; \\
\{\ S \text{ vertex cover of } g_F = (V, E \setminus F) \wedge \mathsf{inv}'(F, S)\ \} \\
\textbf{while } F \neq \emptyset \textbf{ do} \\
\quad e :\in F; \\
\quad F, S := F \setminus \mathit{inc}(e), S \cup e \ \ \textbf{od}; \\
\{\ S \text{ vertex cover of } g \wedge |S| \leq 2c_{opt}\ \} \\
A, B := S, S; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathit{neigh}(b) \subseteq A \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\qquad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ minimal vertex cover of } g \wedge |S| \leq 2c_{opt}\ \}\,.
\end{array}
\qquad (\mathrm{VC}_3)
$$

This program has also $O(|E|)$ as running time (practical experiments have shown that it is only a little bit slower than each of the above single programs), the theoretical worst case approximation bound 2 of (VC_2), but also the good practical behaviour of (VC_1).

3.2 Transitive Reduction

For computing a transitive reduction we assume a *strongly connected*[2] directed graph $g = (V, R)$ with a finite set V of vertices and a relation R on V containing the graph's edges. (Now, each edge is a directed pair of vertices.) Strongly connectedness of g can be described by the equation $R^* = \mathsf{L}$, where the star denotes the reflexive-transitive-closure operator and L denotes the universal relation $V \times V$. In this section we shall also use the transposition operator on relations which we denote as $^\mathsf{T}$.

As already mentioned in the introduction, a transitive reduction T of g is a minimal subrelation of R such that $T^* = R^*$. Due to the strong connectedness of g and the monotonicity of the star-operator wrt. relation inclusion, the predicate $\mathcal{P}$ on the powerset 2^R, defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; X^* = \mathsf{L}\,, \tag{5}$$

is upwards-closed and $\mathcal{P}(R)$ is valid. Furthermore, a little reflection shows that for all non-empty relations $X \in 2^R$ and edges $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge x \in (X \ominus x)^*\,. \tag{6}$$

Hence if we choose the part $x \in (X \ominus x)^*$ of (6) as predicate $\mathcal{Q}(X, x)$, then the predicate $\mathcal{P}$ of (5) is decremental in the sense of (2) and we can instantiate the generic program (MIN_2) accordingly. This yields:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := R, R; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } b \in (A \ominus b)^* \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ transitive reduction of } g\ \}\,.
\end{array} \tag{TR_1}$$

In this program the condition $b \in (A \ominus b)^*$ describes a simple reachability problem: Is the sink of b reachable from its source via edges from $A \ominus b$? This can be tested in time $O(|V| + |A|)$ using an adjacence list representation of A. Hence the total running time of (TR_1) is $O(|R|^2)$.

To get a more efficient program for the transitive reduction of g, we now instantiate the generic program (MIN_3) and compute the result S of the precomputation as follows: Starting from a common root, first we compute a spanning arborescence $t_1 = (V, T_1)$ of g and a spanning arborescence $t_2 = (V, T_2)$ of the transposed graph $g^\mathsf{T} = (V, R^\mathsf{T})$. Then we define S as $T_1 \cup T_2^\mathsf{T}$ and obtain by that a strongly connected subgraph $t = (V, S)$ of g such that $|S| \leq 2|V| - 2$.

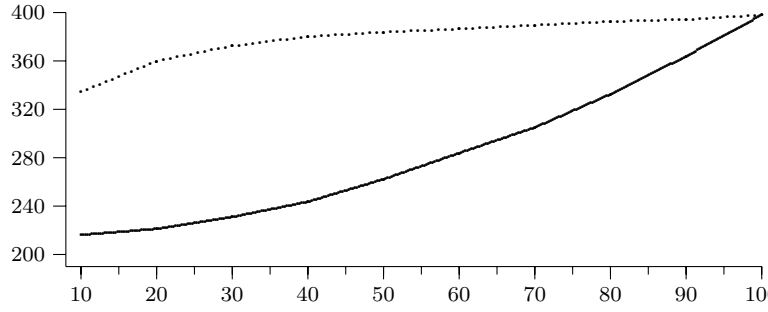
[2] The general case easily can be reduced to the case of strongly connected graphs. See, for instance, [16].

Depth-first search (abbreviated as DFS) is a first possibility to compute spanning arborescences of directed graphs in edge-linear time. If we suppose a corresponding function *Dfs*, then we arrive at the following refinement of program (TR_1) the running time of which now is $O(|V|^2)$:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
r :\in V; \\
S := \mathit{Dfs}(R, r) \cup \mathit{Dfs}(R^{\mathsf{T}}, r)^{\mathsf{T}}; \\
A, B := S, S; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg \mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } b \in (A \ominus b)^* \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ transitive reduction of } g\ \}\,.
\end{array}
\qquad (\mathrm{TR}_2)
$$

Of course, one is particularly interested in transitive reductions with few edges, i.e., in approximations of minimum equivalent digraphs. Concentrating on this aspect and using implementations in the RelView system and the C programming language, we have experimented with the program (TR_2) and with a variant of it which applies breadth-first search (shortly: BFS) instead of DFS for computing the spanning arborescences of the precomputation. The program using DFS proved to be superior. But all our experiments also showed the same negative tendency, viz. the denser the input graph gets the larger gets the number of edges of the result. This is in contrast to the expectation that the number of edges of a good approximation of a minimum equivalent digraph should decrease if the density of the underlying graph increases.

As an example, in the following figure the result of one experiment with a fixed number of 200 vertices and 200 randomly generated input graphs is depicted. (Of course, also much larger graphs have been condidered; see [10].)

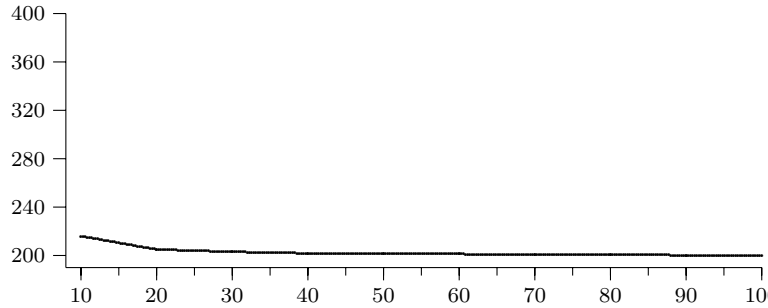


As in the case of vertex covers the graphs' density is listed on the x-axis. On the y-axis the arithmetic mean-values of the sizes of the computed transitive reductions are shown, where the lower curve belongs to the DFS-precomputation and the upper one to the BFS-precomputation.

The reason for the unsatisfactory behaviour of the previous programs for computing small transitive reductions gets clear if one depicts their computed

results on the graph window of RelView. They consist of a lot of short cycles and their shortness even increases if the inputs get denser. But obviously, a small transitive reduction should be composed of a few long cycles only. To obtain this goal, we have used a variant of (TR_2) in which the first arborescence $t_1 = (V, T_1)$ is computed using DFS but the second one $t_2 = (V, T_2)$ using BFS. This specific precomputation was motivated by the following idea: If DFS is applied to get t_1, then this arborescence contains long paths from its root r to every other vertex of g. Due to the use of BFS, however, the transpose of the arborescence t_2 consists of shortest paths from every vertex of g except r back to r. Hence the union of t_1 and t_2^T leads to a starting point of the minimalization process which is composed of long cycles only and obviously this property descends upon the process' result.

Again we have tested this approach and arrived, for example, in the case of 200 randomly generated input graphs, each with the same number of 200 vertices, at the following diagram:



The curves of all other experiments we have performed look rather similar. See [10] for details.

In the course of the diploma thesis [10] the DFS/BFS-approach to transitive reductions has been compared with the best approximation algorithm for minimum equivalent digraphs of [11]. The latter algorithm computes an approximation in nearly linear time with approximation bound between $\frac{\pi^2}{6} \approx 1.64$ and $\frac{\pi^2+1}{6} \approx 1.81$. Although the approximation bound of our program is 2, in all practical experiments it proved to be superior: If n is the number of vertices of the input graph, then the algorithm of [11] leads to results which consist in the average of $\frac{3n}{2}$ edges (i.e., $\approx$300 edges if $n = 200$ instead of the $\approx$210 edges our algorithm yields) and, furthermore, are no transitive reductions. The latter means that they contain superfluous edges. Hence it seems to be reasonable to combine it with our approach. As in the case of vertex covers, this leads to an algorithm with the better theoretical approximation bound and good practical results.

3.3 Some Further Applications

Having presented two graph-theoretic applications in great detail, now we want to sketch some further applications of our generic minimalization programs and their maximalization variants.

First, we have applied the programs to all problems mentioned in the introduction. Furthermore, we have solved many other problems with their means including the computation of kernels of undirected graphs (which are maximal independent sets of vertices) and of minimal transversals of hypergraphs. We have used the programs also for some filter problems on sets. The latter applications are based on the fact that $\{x \in U : Q(x)\}$ is the unique maximal subset of U that satisfies the downwards-closed predicate $\mathcal{P}$ defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; \forall\, x \in X : Q(x) \tag{7}$$

and, furthermore, $\mathcal{P}(\emptyset)$ holds and $\mathcal{P}$ is *incremental* in the sense that for all sets $X \in 2^U$ with $X \neq U$ and all elements $x \in U \setminus X$

$$\mathcal{P}(X \oplus x) \iff \mathcal{P}(X) \wedge \mathcal{Q}(x)\,. \tag{8}$$

As a consequence of (7) and (8), we can instantiate the maximalization variant of (MIN_2) which immediately leads to the following generic program:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := \emptyset, U; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(b) \textbf{ then } A, B := A \oplus b, B \ominus b \\
\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A = \{x \in U : Q(x)\ \}\,.
\end{array} \tag{FIL}$$

The invariant $\mathsf{inv}(A, B)$ of the program (FIL) is given by

$$\mathsf{inv}(A, B) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge A \subseteq U \setminus B \wedge \forall\, x \in (U \setminus B) \setminus A : \neg\mathcal{P}(A \oplus x)\,.$$

A concrete instantiation of our approach is also part of RelView. This system offers some commands for the nice drawing of graphs, one of them also for so-called orthogonal grid drawing. Given a (directed or undirected) graph in which each vertex has at most four neighboars, the underlying algorithm follows the general approach: In a first step it computes a planar subgraph with as much as possible edges. Then the subgraph is drawn on a grid (using Tamassia and Tollis' method [18] in our case). Finally, the remaining edges are inserted into this drawing to obtain an orthogonal drawing of the original graph. Finding a maximum planar subgraph is an NP-complete problem. Hence existing algorithms use heuristics. We have decided to approximate maximum planar subgraphs by maximal ones and used the maximalization variant of our approach (to be planar is a downwards-closed property and empty graphs are pranar) in combination with the well-known Hopcroft-Tarjan planarity test (see [9]) for this task. The experiences we gained so far show that this approach is fast and achieves good drawings in practice.

4 Concluding Remarks

In this article, we have first formally developed a generic program for computing a minimal subset satisfying an upwards-closed predicate $\mathcal{P}$ from a logical specification by combining the invariant technique with set-theoretic and logical calculations. To improve the efficiency of concrete instantiations, we investigated then two refinements. The first refinement assumes $\mathcal{P}$ to be decremental and the second one applies in addition a precomputation phase which yields a good starting point for the minimalization process. Next, we have instantiated the refinements to solve two graph-theoretic problems, viz. the approximation of a minimum vertex cover and the computation of a transitive reduction as approximation of a minimum equivalent digraph. Doing so, we also have presented some experimental results and compared resp. combined our programs with two algorithms known from the literature. This leads to improvements of the practical behaviour of the latter ones. Finally, we have sketched some further applications of the programs and their variants which compute a maximal subset satisfying a downwards-closed predicate.

If we put our work in the context of other work on deriving a family of algorithms, then we see that it follows one of the key idea of generic programming, viz. the lifting of concrete algorithms to as general a level as possible without loosing efficiency. Of course, this is not the only development method. Another very prominent approach uses transformational programming and develops it step-wise from a common specification.

Presently we investigate further instantiations of our generic programs and modifications of them. In particular we seek for ways to formalize the precomputation phase of (MIN_3) in specific cases and to integrate objective functions. Using formal methods we hope that in the latter case the role of the different axioms of the set structures underlying the usual approaches (like independence structures, matroids and greedoids; see [7] for details) become more clear. We also investigate at present how to use signatures, structures, and functors in the functional language Standard ML to implement our generic programs and concrete instantiations in an easy and flexible way and we plan to do the same for an object-oriented language in the future, too. Finally we are interested in the general application of formal specification and development methods in the context of generic programs. We believe that their manifold reusability absolutely requires a well-defined functionality and a rigor mathematical verification that guarantees the specified behaviour. All this is done in combination with so-called algorithm engineering techniques (see e.g., [13, 14]) to get, besides the advantages of formal methods, also a feeling for the programs' actual efficiency and quality and to close the gap between theory and practice.

References

1. R. Behnke et al. Applications of the RelView system. In: R. Berghammer, Y. Lakhnech (eds.): Tool support for system specification, development and verification. Springer, pages 33-47, 1999.

2. R. Berghammer, B. von Karger, and C. Ulke. Relation-algebraic analysis of Petri nets with RelView. In: T. Margaria, B. Steffen (eds.): Proc. 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems, LNCS 1055, Springer, pages 49-69, 1996.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. The MIT Press, 1990.
4. E.W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.
5. J. Edmonds. Paths, trees, and flowers. Canadian Journal of Mathematics 17:449-467, 1965.
6. D. Gries. *The science of computer programming*. Springer, 1981.
7. P. Helman, B.M.E. Moret, H.D. Shapiro. An exact characterization of greedy structures. SIAM Journal Disc. Math. 6:274-283, 1993.
8. D. Hochbaum (ed.). *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1995.
9. J. Hopcroft and R. Tarjan. Efficient planarity testing. Journal of the ACM 21:549-568, 1974.
10. C. Kasper. *Investigating algorithms for transitive reductions and minimum equivalent digraphs (in German)*. Diploma thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.
11. S. Khuller, B. Raghavachari, and N. Young. Approximating the minimum equivalent digraph. SIAM Journal of Computing 24:859-972, 1995.
12. B. Leoniuk. *ROBDD-based implementation of relations and relational operations with applications (in German)*. Ph.D. thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.
13. B.M.E. Moret and H.D. Shapiro. Algorithms and experiments: The new (and old) methodology. Journal of Universal Computer Science 7:434-446, 2001.
14. B.M.E. Moret. Towards a discipline of experimental algorithmics. DIMACS Series in Discrete Mathematics and Theoretical Computer Science (to appear).
15. C. Morgan and T. Vickers (eds.). *On the refinement calculus*. Formal Approaches to Computing and Information Technology, Springer, 1992.
16. H. Noltemeier. Reduktion von Präzedenzstrukturen. Zeitschrift für Operations Research 20:151-159, 1976.
17. J. Ravelo. Two graph algorithms derived. Acta Informatica 36:489-510, 1999.
18. R. Tamassia and T.G. Tollis. Planar grid embedding in linear time. IEEE Transactions on Circuits and Systems 36:1230-1234, 1989.

Justification Based on Program Transformation*
(Extended Abstract)

Hai-Feng Guo[1], C.R. Ramakrishnan[2], and I.V. Ramakrishnan[2]

[1] Computer Science Department
University of Nebraska at Omaha
Omaha, NE 68182-0116
haifengguo@mail.unomaha.edu
[2] State University of New York at Stony Brook
Stony Brook, NY 11794, USA
{cram,ram}@cs.sunysb.edu

Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing evidence, in terms of a proof, for this truth. Justification plays a fundamental role in automatic verification, especially model checking [1]. For instance it can be used for efficient generation of parse trees, synthesis controllers for embedded systems [7], etc.

In an earlier work, we had given algorithms [8,2] for justification using a tabled logic programming system. The naturalness of using a tabled logic programming system for justification is that the answer tables created during query evaluation also serve as the witnesses supporting the result. Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing concise evidence, in terms of a proof, for this truth. Toward that end we presented algorithms for justifying such logic programs by post-processing the memo tables created during query evaluation. Justification in this post-processing fashion is "non-intrusive" in the sense that it is completely decoupled from query evaluation process and is done only after the evaluation is completed.

Justification is done in a post-evaluation phase, by meta-interpreting the memo tables and clauses of the program. This is a major source of inefficiency because meta-interpretation can be significantly slower than the original query evaluation. Moreover, to avoid cyclic explanations we have to maintain a history of the literals that have been used on the proof path. Prior to adding another literal to the proof we have to check if it already appears in the history, a procedure that suffers from quadratic time complexity.

In the full paper [3] we present a general justification technique based on program transformation [5]. First consider justifying true literals. For each literal of the form $p(\bar{t})$ the transformation generates a literal $p_t(\bar{t}, Y)$ such that whenever $p(\bar{t})\theta$ succeeds for some substitution θ, $p_t(\bar{t}, Y)\theta$ succeeds, and in addition, $Y\theta$ represents a valid proof for $p(\bar{t})\theta$. This extra-argument scheme, however, cannot be directly used for tabled programs. Consider a tabled logic program with cyclically defined clauses. A single answer can have infinite number of valid proof paths, and the transformed program will attempt to capture each of these

* Research partially supported by NSF awards EIA-9705998, CCR-9876242, IIS-0072927, EIA-9901602 and CCR-0205376.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 158–159, 2003.

paths. To avoid this problem, instead of using the extra argument, we store the first proof of a tabled literal in an associated (asserted) database. Proofs for tabled literals are accumulated without the need for cycle checking.

The above scheme cannot capture the justification of a false literal, i.e., the reasons for *failure* to prove a literal. For a false literal, we generate the justification based on the notion of completed definition [4] of general logic programs using two-valued logic. To justify a false literal, we generate the dual definition for each predicate defined in the program. We generate a transformed (dual) program such that a literal is false in the original program if and only if its dual in the transformed program is true. Justifying a false literal in the original program thus amounts to justifying its dual is true in the transformed program. Note that justifying a false literal hence involves evaluation of a new (dual) program.

To the best of our knowledge justification via program transformation is a new approach. Our justification scheme has been successfully applied for generation of witnesses and counterexamples in our μ-calculus model checker XMC [6]. Preliminary performance evaluation indicates that the transformation-based justifier for the XMC system adds very little overhead to the XMC model checker. When the model checker deems that a property is true, the overhead for justification to collect that proof is less than 8%. When a property is false, generating the evidence for the absence of a proof takes about 140% of the original evaluation time. In contrast, the meta-interpreter-based justifier originally implemented in the XMC system had an overhead of 4 to 10 *times* the original evaluation time, regardless of the result of the model checking run.

References

1. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 424–435, 1995.
2. Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Speculative beats conservative justification. In *International Conference on Logic Programming*, pages 150–165, 2001.
3. Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Justification based on program transformation. Technical Report, 2002. Available from `http://www.faculty.ist.unomaha.edu/hguo/papers/lopstr02/ext.pdf`.
4. J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, 1987.
5. A. Pettorossi and M. Proietti. Automatic derivation of logic programs by transformation. In *Lecture Notes for the 2000 European Summer School on Logic, Language, and Information*, 2000.
6. C.R Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. Xmc: A logic programming based verification toolset. In *Computer Aided Verification*, 2000.
7. Parthasarathi Roop. Forced simulation: a formal approach to component based development of embedded systems, 2000. PhD thesis.
8. Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Principles and Practice of Declarative Programming*, pages 178–189, 2000.

Combining Logic Programs and Monadic Second Order Logics by Program Transformation

Fabio Fioravanti[1], Alberto Pettorossi[2], and Maurizio Proietti[1]

[1] IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
{fioravanti,proietti}@iasi.rm.cnr.it
[2] DISP, University of Roma Tor Vergata, I-00133 Roma, Italy
adp@iasi.rm.cnr.it

Abstract. We present a program synthesis method based on unfold/fold transformation rules which can be used for deriving terminating definite logic programs from formulas of the Weak Monadic Second Order theory of one successor (WS1S). This synthesis method can also be used as a proof method which is a decision procedure for closed formulas of WS1S. We apply our synthesis method for translating CLP(WS1S) programs into logic programs and we use it also as a proof method for verifying safety properties of infinite state systems.

1 Introduction

The Weak Monadic Second Order theories of k successors (WSkS) are theories of the second order predicate logic which express properties of finite sets of finite strings over a k-symbol alphabet (see [24] for a survey). Their importance relies on the fact that they are among the most expressive theories of predicate logic which are decidable. These decidability results were proved in the 1960's [4,22], but they were considered as purely theoretical results, due to the very high complexity of the automata-based decision procedures.

In recent years, however, it has been shown that some Monadic Second Order theories can, in fact, be decided by using ad-hoc, efficient techniques, such as BDD's and algorithms for finite state automata. In particular, the MONA system implements these techniques for the WS1S and WS2S theories [9].

The MONA system has been used for the verification of several non-trivial finite state systems [3,11]. However, the Monadic Second Order theories alone are not expressive enough to deal with properties of infinite state systems and, thus, for the verification of such systems alternative techniques have been used, such as those based on the embedding of the Monadic Second Order theories into more powerful logical frameworks (see, for instance, [2]).

In a previous paper of ours [7] we proposed a verification method for infinite state systems based on CLP(WSkS), which is a constraint logic programming language resulting from the embedding of WSkS into logic programs. In order to perform proofs of properties of infinite state systems in an automatic way according to the approach we have proposed, we need a system for constraint logic programming which uses a solver for WSkS formulas and, unfortunately, no such system is available yet.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 160–181, 2003.

In order to overcome this difficulty, in this paper we propose a method for translating CLP(WS1S) programs into logic programs. This translation is performed by a two step program synthesis method which produces terminating definite logic programs from WS1S formulas. Step 1 of our synthesis method consists in deriving a normal logic program from a WS1S formula, and it is based on a variant of the Lloyd-Topor transformation [14]. Step 2 consists in applying an unfold/fold transformation strategy to the normal logic program derived at the end of Step 1, thereby deriving a terminating definite logic program. Our synthesis method follows the general approach presented in [16,17]. We believe that, by following this approach, we can translate CLP(WSkS) programs into definite logic programs also for $k > 1$. We leave this task for future research.

The specific contributions of this paper are the following ones.

(1) We provide a synthesis strategy which is guaranteed to terminate for any given WS1S formula.

(2) We prove that, when we start from a closed WS1S formula φ, our synthesis strategy produces a program which is either (i) a unit clause of the form $f \leftarrow$, where f is a nullary predicate equivalent to the formula φ, or (ii) the empty program. Since in case (i) φ is true and in case (ii) φ is false, our strategy is also a decision procedure for WS1S formulas.

(3) We show through a non-trivial example, that our verification method based on CLP(WS1S) programs is useful for verifying properties of infinite state transition systems. In particular, we prove the safety property of a mutual exclusion protocol for a set of processes whose cardinality may change over time. Our verification method requires: (i) the encoding into WS1S formulas of both the transition relation and the elementary properties of the states of a transition system, and (ii) the encoding into a CLP(WS1S) program of the safety property under consideration. Here we perform our verification task by translating the CLP(WS1S) program into a definite logic program, thereby avoiding the use of a solver for WS1S formulas. The verification of the safety property has been performed by using a prototype tool built on top of the MAP transformation system [23].

2 The Weak Monadic Second Order Theory of One Successor

We will consider a *first order* presentation of the Weak Monadic Second Order theory of one successor (WS1S). This first order presentation consists in writing formulas of the form $n \in S$, where $\in$ is a first order predicate symbol (to be interpreted as membership of a natural number to a finite set of natural numbers), instead of formulas of the form $S(n)$, where S is a *predicate variable* (to be interpreted as ranging over finite sets of natural numbers).

We use a *typed* first order language, with the following two types: *nat*, denoting the set of natural numbers, and *set*, denoting the set of the finite sets of natural numbers (for a brief presentation of the typed first order logic the reader may look at [14]). The alphabet of WS1S consists of: (i) a set *Ivars* of *individual*

variables $N, N_1, N_2, \ldots$ of type *nat*, (ii) a set *Svars* of *set variables* $S, S_1, S_2, \ldots$ of type *set*, (iii) the nullary function symbol 0 (*zero*) of type *nat*, and the unary function symbol s (*successor*) of type $nat \to nat$, and (iv) the binary predicate symbols $\leq$ of type $nat \times nat$, and $\in$ of type $nat \times set$. $Ivars \cup Svars$ is ranged over by $X, X_1, X_2, \ldots$ The syntax of WS1S is defined by the following grammar:

$$\begin{array}{lll} \textit{Individual terms}: & n & ::= 0 \mid N \mid s(n) \\ \textit{Atomic formulas}: & A & ::= n_1 \leq n_2 \mid n \in S \\ \textit{Formulas}: & \varphi & ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists N\, \varphi \mid \exists S\, \varphi \end{array}$$

When writing formulas we feel free to use also the connectives $\vee$, $\rightarrow$, $\leftrightarrow$ and the universal quantifier $\forall$, as shorthands of the corresponding formulas with $\neg$, $\wedge$, and $\exists$. Given any two individual terms n_1 and n_2, we will write the formulas $n_1 = n_2$, $n_1 \neq n_2$, and $n_1 < n_2$ as shorthands of the corresponding formulas using $\leq$. For reasons of simplicity, we have assumed that the symbol $\leq$ is primitive, although it is also possible to define it in terms of $\in$ [24].

An example of a WS1S formula is the following formula μ, with free variables N and S, which expresses that N is the maximum number in a finite set S:

$$\mu \ : \ N \in S \wedge \neg\exists N_1 (N_1 \in S \wedge \neg N_1 \leq N)$$

The semantics of WS1S formulas is defined by considering the following *typed interpretation* $\mathcal{N}$:
(i) the domain of the type *nat* is the set *Nat* of the natural numbers and the domain of the type *set* is the set $P_{fin}(Nat)$ of all finite subsets of *Nat*;
(ii) the constant symbol 0 is interpreted as the natural number 0 and the function symbol s is interpreted as the successor function from *Nat* to *Nat*;
(iii) the predicate symbol $\leq$ is interpreted as the less-or-equal relation on natural numbers, and the predicate symbol $\in$ is interpreted as the membership of a natural number to a finite set of natural numbers.

The notion of a *variable assignment* σ *over a typed interpretation* is analogous to the untyped case, except that σ assigns to a variable an element of the domain of the type of the variable. The definition of the *satisfaction relation* $I \models_\sigma \varphi$, where I is a typed interpretation and σ is a variable assignment is also analogous to the untyped case, with the only difference that when we interpret an existentially quantified formula we assume that the quantified variable ranges over the domain of its type. We say that a formula φ is *true* in an interpretation I, written as $I \models \varphi$, iff $I \models_\sigma \varphi$ for all variable assignments σ. The problem of checking whether or not a WS1S formula is true in the interpretation $\mathcal{N}$ is decidable [4].

3 Translating WS1S Formulas into Normal Logic Programs

In this section we illustrate Step 1 of our method for synthesizing definite programs from WS1S formulas. In this step, starting from a WS1S formula, we derive a *stratified* normal logic program [1] (simply called *stratified programs*) by applying a variant of the Lloyd-Topor transformation, called *typed Lloyd-Topor*

transformation. Given a stratified program P, we denote by $M(P)$ its *perfect model* (which is equal to its *least Herbrand model* if P is a definite program) [1].

Before presenting the typed Lloyd-Topor transformation, we need to introduce a definite program, called *NatSet*, which axiomatizes: (i) the natural numbers, (ii) the finite sets of natural numbers, (iii) the ordering on natural numbers ($\leq$), and (iv) the membership of a natural number to a finite set of natural numbers ($\in$). We represent: (i) a natural number $k\,(\geq 0)$ as a ground term of the form $s^k(0)$, and (ii) a set of natural numbers as a finite, ground list $[b_0, b_1, \ldots, b_m]$ where, for $i = 0, \ldots, m$, we have that b_i is either `t` (short for `true`) or `f` (short for `false`). A number k belongs to the set represented by $[b_0, b_1, \ldots, b_m]$ iff $0 \leq k \leq m$ and $b_k = \mathtt{t}$. Thus, the finite, ground lists $[b_0, b_1, \ldots, b_m]$ and $[b_0, b_1, \ldots, b_m, \mathtt{f}, \ldots, \mathtt{f}]$ represent the same set. In particular, the empty set is represented by any list of the form $[\mathtt{f}, \ldots, \mathtt{f}]$. The program *NatSet* consists of the following clauses (we adopt infix notation for $\leq$ and $\in$):

NatSet:

$$\begin{array}{ll} nat(0) \leftarrow & 0 \leq N \leftarrow \\ nat(s(N)) \leftarrow nat(N) & s(N_1) \leq s(N_2) \leftarrow N_1 \leq N_2 \\ set([\,]) \leftarrow & 0 \in [\mathtt{t}|S] \leftarrow \\ set([\mathtt{t}|S]) \leftarrow set(S) & s(N) \in [B|S] \leftarrow N \in S \\ set([\mathtt{f}|S]) \leftarrow set(S) & \end{array}$$

Atoms of the form $nat(N)$ and $set(S)$ are called *type atoms*. Now we will establish a correspondence between the set of WS1S formulas which are true in the typed interpretation $\mathcal{N}$ and the set of the so-called *explicitly typed* WS1S formulas which are true in the least Herbrand model $M(NatSet)$ (see Theorem 1 below).

Given a WS1S formula φ, the *explicitly typed* WS1S formula corresponding to φ is the formula φ_τ constructed as follows. We first replace the subformulas of the form $\exists N\, \psi$ by $\exists N\, (nat(N) \wedge \psi)$ and the subformulas of the form $\exists S\, \psi$ by $\exists S\, (set(S) \wedge \psi)$, thereby getting a new formula φ_η where every bound (individual or set) variable occurs in a type atom. Then, we get:

$$\varphi_\tau \;:\; nat(N_1) \wedge \ldots \wedge nat(N_h) \wedge set(S_1) \wedge \ldots \wedge set(S_k) \wedge \varphi_\eta$$

where $N_1, \ldots, N_h, S_1, \ldots, S_k$ are the variables which occur free in φ.

For instance, let us consider again the formula μ which expresses that N is the maximum number in a set S. The explicitly typed formula corresponding to μ is the following formula:

$$\mu_\tau \;:\; nat(N) \wedge set(S) \wedge N \in S \wedge \neg \exists N_1 (nat(N_1) \wedge N_1 \in S \wedge \neg N_1 \leq N)$$

For reasons of simplicity, in the following Theorem 1 we identify: (i) a natural number $k\,(\geq 0)$ in *Nat* with the ground term $s^k(0)$ representing that number, and (ii) a finite set of natural numbers in $P_{fin}(Nat)$ with any finite, ground list representing that set. By using these identifications, we can view any variable assignment over the typed interpretation $\mathcal{N}$ also as a variable assignment over the untyped interpretation $M(NatSet)$ (but not vice versa).

Theorem 1. Let φ be a WS1S formula and let φ_τ be the explicitly typed formula corresponding to φ. For every variable assignment σ over $\mathcal{N}$, $\mathcal{N} \models_\sigma \varphi$ iff $M(NatSet) \models_\sigma \varphi_\tau$.

Proof. The proof proceeds by induction on the structure of the formula φ.

(i) Suppose that φ is of the form $n_1 \leq n_2$. By the definition of the satisfaction relation, $\mathcal{N} \models_\sigma n_1 \leq n_2$ iff the natural number $\sigma(n_1)$ is less or equal than the natural number $\sigma(n_2)$. By the definition of least Herbrand model and by using the clauses in *NatSet* which define $\leq$, $\sigma(n_1)$ is less or equal than $\sigma(n_2)$ iff $M(NatSet) \models \sigma(n_1) \leq \sigma(n_2)$ (here we identify every natural number n with the ground term $s^n(0)$). It can be shown that $M(NatSet) \models nat(\sigma(n_1))$ and $M(NatSet) \models nat(\sigma(n_2))$. Thus, $M(NatSet) \models \sigma(n_1) \leq \sigma(n_2)$ iff $M(NatSet) \models_\sigma nat(n_1) \wedge nat(n_2) \wedge n_1 \leq n_2$. Now, the term n_1 is either of the form $s^{m1}(0)$ or of the form $s^{m1}(N_1)$, where $m1$ is a natural number. Similarly, the term n_2 is either of the form $s^{m2}(0)$ or of the form $s^{m2}(N)$, where $m2$ is a natural number. We consider the case where n_1 is $s^{m1}(N_1)$ and n_2 is $s^{m2}(N_2)$. The other cases are similar and we omit them. It can be shown that, for all natural numbers m, $M(NatSet) \models_\sigma nat(s^m(N))$ iff $M(NatSet) \models_\sigma nat(N)$. Thus, $M(NatSet) \models_\sigma nat(s^{m1}(N_1)) \wedge nat(s^{m2}(N_2)) \wedge s^{m1}(N_1) \leq s^{m2}(N_2)$ iff $M(NatSet) \models_\sigma nat(N_1) \wedge nat(N_2) \wedge s^{m1}(N_1) \leq s^{m2}(N_2)$, that is, $M(NatSet) \models_\sigma (n_1 \leq n_2)_\tau$.
(ii) The case where φ is of the form $n \in S$ is similar to Case (i).
(iii) Suppose that φ is of the form $\neg\psi$. By the definition of the satisfaction relation and the induction hypothesis, $\mathcal{N} \models_\sigma \neg\psi$ iff $M(NatSet) \models_\sigma \neg(\psi_\tau)$. Since ψ_τ is of the form $a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \psi_\eta$, where $X_1, \ldots, X_k$ are the free variables in ψ and $a_1(X_1), \ldots, a_k(X_k)$ are type atoms, by logical equivalence, we get: $M(NatSet) \models_\sigma \neg(\psi_\tau)$ iff $M(NatSet) \models_\sigma (a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \neg(\psi_\eta)) \vee \neg(a_1(X_1) \wedge \ldots \wedge a_k(X_k))$. Finally, since for all variable assignments σ, $M(NatSet) \models_\sigma a_1(X_1) \wedge \ldots \wedge a_k(X_k)$, we have that $M(NatSet) \models_\sigma \neg(\psi_\tau)$ iff $M(NatSet) \models_\sigma (a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \neg(\psi_\eta))$, that is, $M(NatSet) \models_\sigma (\neg\psi)_\tau$ (to see this, note that $\neg(\psi_\eta)$ is equal to $(\neg\psi)_\eta$).
(iv) The case where φ is of the form $\psi_1 \wedge \psi_2$ is similar to Case (iii).
(v) Suppose that φ is of the form $\exists N_1\, \psi$. By the definition of the satisfaction relation and by the induction hypothesis, $\mathcal{N} \models_\sigma \exists N_1\, \psi$ iff there exists n_1 in *Nat* such that $M(NatSet) \models_{\sigma[N_1 \mapsto n_1]} \psi_\tau$. Since ψ_τ is of the form $nat(N_1) \wedge \ldots \wedge nat(N_h) \wedge set(S_1) \wedge \ldots \wedge set(S_k) \wedge \psi_\eta$, where $N_1, \ldots, N_h, S_1, \ldots, S_k$ are the free variables in ψ, we have that:
there exists n_1 in *Nat* such that $M(NatSet) \models_{\sigma[N_1 \mapsto n_1]} \psi_\tau$
iff $M(NatSet) \models_\sigma \exists N_1\, (nat(N_1) \wedge \ldots \wedge nat(N_h) \wedge set(S_1) \wedge \ldots \wedge set(S_k) \wedge \psi_\eta)$
iff (by logical equivalence) $M(NatSet) \models_\sigma nat(N_2) \wedge \ldots \wedge nat(N_h) \wedge set(S_1) \wedge \ldots \wedge set(S_k) \wedge (\exists N_1\, nat(N_1) \wedge \psi_\eta)$
iff (by definition of explicitly typed formula) $M(NatSet) \models_\sigma (\exists N_1\, \psi)_\tau$.
(vi) The case where φ is of the form $\exists S\, \psi$ is similar to Case (v). □

As a straightforward consequence of Theorem 1, we have the following result.

Corollary 1. For every closed WS1S formula φ, $\mathcal{N} \models \varphi$ iff $M(NatSet) \models \varphi_\tau$.

Notice that the introduction of type atoms is indeed necessary, because there are WS1S formulas φ such that $\mathcal{N} \models \varphi$ and $M(NatSet) \not\models \varphi$.

For instance, $\mathcal{N} \models \forall N_1 \exists N_2\, N_1 \leq N_2$ and $M(NatSet) \not\models \forall N_1 \exists N_2\, N_1 \leq N_2$. Indeed, for a variable assignment σ over $M(NatSet)$ which assigns $[\,]$ to N_1, we have $M(NatSet) \not\models_\sigma \exists N_2\, N_1 \leq N_2$. (Notice that σ is not a variable assignment over $\mathcal{N}$ because $[\,]$ is not a natural number.)

Now we present a variant of the method proposed by Lloyd and Topor [14], called *typed Lloyd-Topor transformation*, which we use for deriving a stratified program from a given WS1S formula φ. We need to consider a class of formulas of the form: $A \leftarrow \beta$, called *statements*, where A is an atom, called the *head* of the statement, and β is a formula of the first order predicate calculus, called the *body* of the statement. In what follows we write $C[\gamma]$ to denote a formula where the subformula γ occurs as an *outermost conjunct*, that is, $C[\gamma] = \psi_1 \wedge \gamma \wedge \psi_2$ for some subformulas ψ_1 and ψ_2.

The Typed Lloyd-Topor Transformation.
We are given in input a set of statements, where: (i) we assume without loss of generality, that the only connectives and quantifiers occurring in the body of the statements are $\neg, \wedge$, and $\exists$, and (ii) $X, X_1, X_2, \ldots$ denote either individual or set variables.

(A) We repeatedly apply the following rules A.1–A.4 until a set of clauses is generated:

(A.1) $A \leftarrow C[\neg\neg\gamma]$ is replaced by $A \leftarrow C[\gamma]$.

(A.2) $A \leftarrow C[\neg(\gamma \wedge \delta)]$ is replaced by $A \leftarrow C[\neg newp(X_1, \ldots, X_k)]$
$newp(X_1, \ldots, X_k) \leftarrow \gamma \wedge \delta$
where *newp* is a new predicate and $X_1, \ldots, X_k$ are the variables which occur free in $\gamma \wedge \delta$.

(A.3) $A \leftarrow C[\neg\exists X\, \gamma]$ is replaced by $A \leftarrow C[\neg newp(X_1, \ldots, X_k)]$
$newp(X_1, \ldots, X_k) \leftarrow \gamma$
where *newp* is a new predicate and $X_1, \ldots, X_k$ are the variables which occur free in $\exists X\, \gamma$.

(A.4) $A \leftarrow C[\exists X\, \gamma]$ is replaced by $A \leftarrow C[\gamma\{X/X_1\}]$
where X_1 is a new variable.

(B) Every clause $A \leftarrow G$ is replaced by $A \leftarrow G_\tau$.

Definition 1. Given a WS1S formula φ with free variables $X_1, \ldots, X_n$, we denote by $Cls(f, \varphi_\tau)$ the set of clauses derived by applying the typed Lloyd-Topor transformation starting from the singleton $\{f(X_1, \ldots, X_n) \leftarrow \varphi\}$, where f is a new n-ary predicate symbol.

By construction, we have that $NatSet \cup Cls(f, \varphi_\tau)$ is a stratified program. We have the following theorem whose proof is similar to those presented in [14,16].

Theorem 2. Let φ be a WS1S formula with free variables $X_1, \ldots, X_n$ and let φ_τ be the explicitly typed formula corresponding to φ. For all ground terms $t_1, \ldots, t_n$, $M(NatSet) \models \varphi_\tau\{X_1/t_1, \ldots, X_n/t_n\}$ iff $M(NatSet \cup Cls(f, \varphi_\tau)) \models f(t_1, \ldots, t_n)$.

From Theorems 1 and 2 we have the following corollary.

Corollary 2. For every WS1S formula φ with free variables $X_1, \ldots, X_n$, and for every variable assignment σ over the typed interpretation $\mathcal{N}$, $\mathcal{N} \models_\sigma \varphi$ iff $M(NatSet \cup Cls(f, \varphi_\tau)) \models f(\sigma(X_1), \ldots, \sigma(X_n))$. In particular, for every closed WS1S formula φ, $\mathcal{N} \models \varphi$ iff $M(NatSet \cup Cls(f, \varphi_\tau)) \models f$.

Let us consider again the formula μ expressing that N is the maximum number in set S. By applying the typed Lloyd-Topor transformation starting from $\{max(S,N) \leftarrow \mu\}$ we get the following set $Cls(max, \mu_\tau)$ of clauses:

$max(S,N) \leftarrow nat(N) \wedge set(S) \wedge N \in S \wedge \neg newp(S,N)$
$newp(S,N) \leftarrow nat(N) \wedge nat(N_1) \wedge set(S) \wedge N_1 \in S \wedge \neg N_1 \leq N$

Unfortunately, the stratified program $NatSet \cup Cls(f, \varphi_\tau)$ derived by applying the typed Lloyd-Topor transformation starting from $\{f(X_1, \ldots, X_n) \leftarrow \varphi\}$, is not always satisfactory from a computational point of view, because it may not terminate when evaluating the query $f(X_1, \ldots, X_n)$ by using SLDNF resolution. (Actually, the above program $Cls(max, \mu_\tau)$ that computes the maximum number of a set, terminates for all ground queries, but in Section 5 we will give an example where the program derived by applying the typed Lloyd-Topor transformation does not terminate.) Similar termination problems may occur if one uses *tabled resolution* [5], instead of SLDNF resolution.

To overcome this problem, we apply to the program $NatSet \cup Cls(f, \varphi_\tau)$ the unfold/fold transformation strategy which we will describe in Section 5. In particular, by applying this strategy we derive definite programs which terminate for all ground queries by using LD resolution (that is, SLD resolution with the leftmost selection rule).

4 The Transformation Rules

In this section we describe the transformation rules which we use for transforming stratified programs. These rules are a subset of those presented in [16,17], and they are those required for the unfold/fold transformation strategy presented in Section 5.

For presenting our rules we need the following notions. A variable in the body of a clause C is said to be *existential* iff it does not occur in the head of C. The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the set of the clauses of P whose head predicate is p. The *extended definition* of a predicate p in a program P, denoted by $Def^*(p, P)$, is the union of the definition of p and the definitions of all predicates in P on which p depends. (See [1]for the definition of the *depends on* relation.) A program is *propositional* iff every predicate occurring in the program is nullary. Obviously, if P is a propositional program then, for every predicate p, $M(P) \models p$ is decidable.

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of programs, where for $0 \leq k \leq n-1$, program P_{k+1} is derived from program P_k by the application of one of the transformation rules R1–R4 listed below. For $0 \leq k \leq n$, we consider the set $Defs_k$ of the clauses introduced by the following rule R1 during the construction of the transformation sequence $P_0, \ldots, P_k$.

When considering clauses of programs, we will feel free to apply the following transformations which preserve the perfect model semantics: (1) renaming of variables, (2) rearrangement of the order of the literals in the body of a clause, and (3) replacement of a conjunction of literals the form $L \wedge L$ in the body of a clause by the literal L.

Rule R1. Definition. We get the new program P_{k+1} by adding to program P_k a clause of the form $newp(X_1, \ldots, X_r) \leftarrow L_1 \wedge \ldots \wedge L_m$, where: (i) the predicate $newp$ does not occur in the sequence $P_0, \ldots, P_k$, and (ii) $X_1, \ldots, X_r$ are distinct (individual or set) variables occurring in $L_1 \wedge \ldots \wedge L_m$.

Rule R2. Unfolding. Let C be a renamed apart clause in P_k of the form: $H \leftarrow G_1 \wedge L \wedge G_2$, where L is either the atom A or the negated atom $\neg A$. Let $H_1 \leftarrow B_1, \ldots, H_m \leftarrow B_m$, with $m \geq 0$, be all clauses of program P_k whose head is unifiable with A and, for $j = 1, \ldots, m$, let ϑ_j the most general unifier of A and H_j. We consider the following two cases.

Case 1: L is A. By *unfolding* clause C *w.r.t.* A we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{(H \leftarrow G_1 \wedge B_1 \wedge G_2)\vartheta_1, \ldots, (H \leftarrow G_1 \wedge B_m \wedge G_2)\vartheta_m\}$. In particular, if $m = 0$, that is, if we unfold C w.r.t. an atom which is not unifiable with the head of any clause in P_k, then we derive the program P_{k+1} by deleting clause C.

Case 2: L is $\neg A$. Assume that: (i) $A = H_1\vartheta_1 = \cdots = H_m\vartheta_m$, that is, for $j = 1, \ldots, m$, A is an instance of H_j, (ii) for $j = 1, \ldots, m$, $H_j \leftarrow B_j$ has no existential variables, and (iii) $Q_1 \vee \ldots \vee Q_r$, with $r \geq 0$, is the disjunctive normal form of $G_1 \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_2$. By *unfolding* clause C *w.r.t.* $\neg A$ we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{C_1, \ldots, C_m\}$, where for $j = 1, \ldots, r$, C_j is the clause $H \leftarrow Q_j$.
In particular: (i) if $m = 0$, that is, A is not unifiable with the head of any clause in P_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause C, and (ii) if for some $j \in \{1, \ldots, m\}$, B_j is the empty conjunction, that is, A is an instance of the head of a unit clause in P_k, then we derive P_{k+1} by deleting clause C from P_k.

Rule R3. Folding. Let $C : H \leftarrow G_1 \wedge B\vartheta \wedge G_2$ be a renamed apart clause in P_k and $D : Newp \leftarrow B$ be a clause in $Defs_k$. Suppose that for every existential variable X of D, we have that $X\vartheta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $Y\vartheta$, for any variable Y occurring in B and different from X. By *folding* clause C using clause D we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{H \leftarrow G_1 \wedge Newp\,\vartheta \wedge G_2\}$.

Rule R4. Propositional Simplification. Let p be a predicate such that $Def^*(p, P_k)$ is propositional. If $M(Def^*(p, P_k)) \models p$ then we derive $P_{k+1} = (P_k - Def(p, P_k)) \cup \{p \leftarrow\}$. If $M(Def^*(p, P_k)) \models \neg p$ then we derive $P_{k+1} = (P_k - Def(p, P_k))$.

We have the following correctness result, similar to [16].

Theorem 3. [Correctness of the Unfold/Fold Transformation Rules] Let us assume that during the construction of a transformation sequence $P_0, \ldots, P_n$, each clause of $Defs_n$ which is used for folding, is unfolded (before or after its use for folding) w.r.t. an atom whose predicate symbol occurs in P_0. Then, $M(P_0 \cup Defs_n) = M(P_n)$.

Notice that we cannot replace 'atom' by 'literal' in the statement of Theorem 3. Indeed, let us consider the program P_0: $\{p \leftarrow \neg q(X), \quad q(X) \leftarrow q(X),$

$q(X) \leftarrow r\}$. By unfolding the clause $p \leftarrow \neg q(X)$ w.r.t. the literal $\neg q(X)$ and then folding, we get the following program P_1: $\{p \leftarrow p \wedge \neg r,\ \ q(X) \leftarrow q(X),\ q(X) \leftarrow r\}$. We have that $p \in M(P_0)$ and $p \notin M(P_1)$.

5 The Unfold/Fold Synthesis Method

In this section we present our program synthesis method, called *unfold/fold synthesis method*, which derives a definite program from any given WS1S formula. We show that the synthesis method terminates for all given formulas and also the derived programs terminate according to the following notion of program termination: a program P *terminates for a query* Q iff every SLD-derivation of $P \cup \{\leftarrow Q\}$ via any computation rule is finite.

The following is an outline of our unfold/fold synthesis method.

The Unfold/Fold Synthesis Method.
Let φ be a WS1S formula with free variables $X_1, \ldots, X_n$ and let φ_τ be the explicitly typed formula corresponding to φ.

Step 1. We apply the *typed Lloyd-Topor transformation* and we derive a set $\mathit{Cls}(f, \varphi_\tau)$ of clauses such that: (i) f is a new n-ary predicate symbol, (ii) $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ is a stratified program, and (iii) for all ground terms $t_1, \ldots, t_n$, $M(\mathit{NatSet}) \models \varphi_\tau\{X_1/t_1, \ldots, X_n/t_n\}$ iff $M(\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)) \models f(t_1, \ldots, t_n)$.

Step 2. We apply the *unfold/fold transformation strategy* (see below) and from the program $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ we derive a definite program *TransfP* such that, for all ground terms $t_1, \ldots, t_n$,
(i) $M(\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)) \models f(t_1, \ldots, t_n)$ iff $M(\mathit{TransfP}) \models f(t_1, \ldots, t_n)$, and
(ii) *TransfP* terminates for the query $f(t_1, \ldots, t_n)$.

In order to present the unfold/fold transformation strategy which we use for realizing Step 2 of our synthesis method, we introduce the following notions of *regular natset-typed clauses* and *regular natset-typed definitions*.

We say that a literal is *linear* iff each variable occurs at most once in it. The syntax of regular natset-typed clauses is defined by the following grammar (recall that by N we denote individual variables, by S we denote set variables, and by $X, X_1, X_2, \ldots$ we denote either individual or set variables):

Head terms: $h ::= 0 \mid s(N) \mid [\,] \mid [\mathtt{t}|S] \mid [\mathtt{f}|S]$
Natset-typed clauses: $C ::= p(h_1, \ldots, h_k) \leftarrow \ \mid p_1(h_1, \ldots, h_k) \leftarrow p_2(X_1, \ldots, X_m)$

where for every natset-typed clause C, (i) both $\mathit{hd}(C)$ and $\mathit{bd}(C)$ are linear atoms, and (ii) $\{X_1, \ldots, X_m\} \subseteq \mathit{vars}(h_1, \ldots, h_k)$ (that is, C has no existential variables). A *regular natset-typed program* is a set of regular natset-typed clauses which includes the clauses defining the predicates *nat* and *set* (see Section 3). The reader may check that the program *NatSet* presented in Section 3 is indeed a regular natset-typed program.

Lemma 1. Let P be a regular natset-typed program.
(i) If p is a nullary predicate, then $\mathit{Def}^*(p, P)$ is propositional.

(ii) Suppose that, for each nullary predicate p, $Def^*(p, TransfP)$ is either the empty set or the singleton $\{p \leftarrow\}$, then P terminates for every ground query $q(t_1, \ldots, t_n)$ with $n \geq 0$.

Proof. Point (i) follows from the fact that a regular natset-typed clause has no existential variables and no ground argument can occur in its body. For proving Point (ii) let us consider an SLD-derivation $\leftarrow q(t_1, \ldots, t_n), \ldots, \leftarrow q_i(s_1, \ldots, s_k), \leftarrow q_j(u_1, \ldots, u_m), \ldots$ Since $\leftarrow q(t_1, \ldots, t_n)$ is a ground goal and the clauses of P have no existential variables, every goal in the sequence is ground. If $m = 0$ then $Def^*(q_j, TransfP)$ is either the empty set or the singleton $\{q_j \leftarrow\}$ and the SLD-derivation is finite. If $m > 0$, by the definition of regular natset-typed clauses, each term among $u_1, \ldots, u_m$ is a proper subterm of a term among $s_1, \ldots, s_k$. Thus, also in this case the SLD-derivation is finite. □

The syntax of natset-typed definitions is given by the following grammar:

Individual terms:	n	$::= 0 \mid N \mid s(n)$
Terms:	t	$::= n \mid S$
Type atoms:	T	$::= nat(N) \mid set(S)$
Literals:	L	$::= p(t_1, \ldots, t_k) \mid \neg p(t_1, \ldots, t_k)$
Natset-typed definitions:	D	$::= p(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$

where for every natset-typed definition D, $vars(D) \subseteq vars(T_1 \wedge \ldots \wedge T_r)$.

A sequence $D_1, \ldots, D_s$ of natset-typed definitions is said to be a *hierarchy* iff for $i = 1, \ldots, s$ the predicate appearing in $hd(D_i)$ does not occur in $D_1, \ldots, D_{i-1}, bd(D_i)$. Notice that in a hierarchy of natset-typed definitions, any predicate occurs in the head of at most one clause.

One can show that given a WS1S formula φ the set $Cls(f, \varphi_\tau)$ of clauses derived by applying the typed Lloyd-Topor transformation is a hierarchy $D_1, \ldots, D_s$ of natset-typed definitions and the last clause D_s is the one defining f.

The Unfold/Fold Transformation Strategy.

Input: (i) A regular natset-typed program P where for each nullary predicate p, $Def^*(p, TransfP)$ is either the empty set or the singleton $\{p \leftarrow\}$, and (ii) a hierarchy $D_1, \ldots, D_s$ of natset-typed definitions such that no predicate occurring in P occurs also in the head of a clause in $D_1, \ldots, D_s$.
Output: A regular natset-typed program *TransfP* such that, for all ground terms $t_1, \ldots, t_n$,
(i) $M(P \cup \{D_1, \ldots, D_s\}) \models f(t_1, \ldots, t_n)$ iff $M(TransfP) \models f(t_1, \ldots, t_n)$;
(ii) *TransfP* terminates for the query $f(t_1, \ldots, t_n)$.

TransfP := P; *Defs* := $\emptyset$;
FOR $i = 1, \ldots, s$ DO
Defs := *Defs* $\cup \{D_i\}$; *InDefs* := $\{D_i\}$;
By the definition rule we derive the program *TransfP* $\cup$ *InDefs*.
WHILE *InDefs* $\neq \emptyset$ DO
(1) *Unfolding.* From program *TransfP* $\cup$ *InDefs* we derive *TransfP* $\cup U$ by: (i) applying the unfolding rule w.r.t. each atom occurring positively in the body of a

clause in *InDefs*, thereby deriving $TransfP \cup U_1$, then (ii) applying the unfolding rule w.r.t. each negative literal occurring in the body of a clause in U_1, thereby deriving $TransfP \cup U_2$, and, finally, (iii) applying the unfolding rule w.r.t. ground literals until we derive a program $TransfP \cup U$ such that no ground literal occurs in the body of a clause of U.

(2) *Definition-Folding.* From program $TransfP \cup U$ we derive $TransfP \cup F \cup NewDefs$ as follows. Initially, *NewDefs* is the empty set. For each non-unit clause C: $H \leftarrow B$ in U,
(i) we apply the definition rule and we add to *NewDefs* a clause of the form $newp(X_1, \ldots, X_k) \leftarrow B$, where $X_1, \ldots, X_k$ are the non-existential variables occurring in B, unless a variant clause already occurs in *Defs*, modulo the head predicate symbol and the order and multiplicity of the literals in the body, and (ii) we replace C by the clause derived by folding C w.r.t. B. The folded clause is an element of F.
No transformation rule is applied to the unit clauses occurring in U and, therefore, also these unit clauses are elements of F.

(3) $TransfP := TransfP \cup F$; $\quad Defs := Defs \cup NewDefs$; $\quad InDefs := NewDefs$

END WHILE;

Propositional Simplification. For each predicate p such that $Def^*(p, TransfP)$ is propositional, we apply the propositional simplification rule and
if $M(TransfP) \models p$ then $TransfP := (TransfP - Def(p, TransfP)) \cup \{p \leftarrow\}$
else $TransfP := (TransfP - Def(p, TransfP))$

END FOR

The reader may verify that if we apply the unfold/fold transformation strategy starting from the program *NatSet* together with the clauses $Cls(max, \mu_\tau)$ which we have derived above by applying the typed Lloyd-Topor transformation, we get the following program:

$max([\mathtt{t}|S], 0) \leftarrow new1(S)$
$max([\mathtt{t}|S], s(N)) \leftarrow max(S, N)$
$max([\mathtt{f}|S], s(N)) \leftarrow max(S, N)$
$new1([\,]) \leftarrow$
$new1([\mathtt{f}|S]) \leftarrow new1(S)$

To understand the first clause, recall that the empty set is represented by any list of the form $[\mathtt{f}, \ldots, \mathtt{f}]$. A detailed example of application of the unfold/fold transformation strategy will be given later.

In order to prove the correctness and the termination of our unfold/fold transformation strategy we need the following two lemmas whose proofs are mutually dependent.

Lemma 2. During the application of the unfold/fold transformation strategy, *TransfP* is a regular natset-typed program.

Proof. Initially, *TransfP* is the regular natset-typed program P. Now we assume that *TransfP* is a regular natset-typed program and we show that after an execution of the body of the FOR statement, *TransfP* is a regular natset-typed program.

In order to prove that after the execution of the WHILE statement, *TransfP* is a regular natset-typed program, we show that every new clause E which is added to *TransfP* at Point (3) of the strategy is a regular natset-typed clause. Indeed, clause E is derived from a clause D of *InDefs* by unfolding (according to the Unfolding phase) and by folding (according to the Definition-Folding phase). By Lemma 3, D is a natset-typed definition of the form $p(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$. By the inductive hypothesis *TransfP* is a regular natset-typed program and, therefore, by unfolding we get clauses of the form D' : $p(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_{m1}$, where: (a) $h_1, \ldots, h_k$ are head terms, and (b) $p(h_1, \ldots, h_k)$ is a linear atom. Hence, *either* E is a unit clause of the form $p(h_1, \ldots, h_k) \leftarrow$ *or* E is a clause derived by folding according to the Definition-Folding phase and it is of the form $p(h_1, \ldots, h_k) \leftarrow newp(X_1, \ldots, X_m)$, where $\{X_1, \ldots, X_m\} \subseteq vars(h_1, \ldots, h_k)$. Thus, E is a regular natset-typed clause.

We complete the proof by observing that by applying the propositional simplification rule to a regular natset-typed program we derive a regular natset-typed program. □

Lemma 3. During the application of the unfold/fold transformation strategy, *InDefs* is a set of natset-typed definitions.

Proof. Let us consider the i-th execution of the body of the FOR statement. Initially, *InDefs* is the singleton set $\{D_i\}$ of natset-typed definitions. Now we assume that *InDefs* is a set of natset-typed definitions and we prove that, after an execution of the WHILE statement, *InDefs* is a set of natset-typed definitions. It is enough to show that every new clause E which is added to *InDefs* at Point (3) of the strategy, is a natset-typed definition. By the Folding phase of the strategy, E is a clause of the form $newp(X_1, \ldots, X_k) \leftarrow B$, where B is the body of a clause derived from a clause D of *InDefs* by unfolding. By the inductive hypothesis, D is a natset-typed definition of the form $p(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$. Since, by Lemma 2, *TransfP* is a regular natset-typed program, by unfolding we get clauses of the form D' : $p(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_{m1}$, where: (a) $T'_1, \ldots, T'_{r1}$are type atoms, (b) $L'_1, \ldots, L'_m$ are literals as specified by the grammar defining natset-typed definitions, and (c) $vars(D') \subseteq vars(T'_1 \wedge \ldots \wedge T'_{r1})$. Thus, E is a natset-typed definition of the form $newp(X_1, \ldots, X_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_{m1}$ with $vars(E) \subseteq vars(T'_1 \wedge \ldots \wedge T'_{r1})$. □

Theorem 4. Let P and $D_1, \ldots, D_s$ be the input program and the input hierarchy, respectively, of the unfold/fold transformation strategy and let *TransfP* be the output of the strategy. Then: (1) *TransfP* is a natset-typed program, (2) for every nullary predicate p, $Def^*(p, TransfP)$ is either $\emptyset$ or $\{p \leftarrow\}$, and (3) for all ground terms $t_1, \ldots, t_n$,

(3.1) $M(P \cup \{D_1, \ldots, D_s\}) \models f(t_1, \ldots, t_n)$ iff $M(TransfP) \models f(t_1, \ldots, t_n)$;

(3.2) *TransfP* terminates for the query $f(t_1, \ldots, t_n)$.

Proof. Point (1) is a straightforward consequence of Lemma 2.

Point (2) follows from Lemma 2, Lemma 1, and the fact that in the last step of the unfold/fold transformation strategy we apply the propositional simplification rule for each predicate having an extended definition which is propositional.

In order to prove Point (3.1) we first notice that the unfold/fold transformation strategy generates a transformation sequence (see Section 4), where: the initial program is P, the final program is the final value of *TransfP*, and the set of clauses introduced by the definition rule R1 is the final value of *Defs*. To see that our strategy indeed generates a transformation sequence, let us observe the following facts (A) and (B).
(A) By the hypotheses on the input sequence $D_1, \ldots, D_s$, the addition of *InDefs* to *TransfP* at the beginning of each execution of the body of the FOR statement is an application of the definition rule.
(B) When unfolding a clause in U_1 w.r.t. a negative literal $\neg p(t_1, \ldots, t_k)$, we have that:
(B.1) Condition (i) of Case (2) of the unfolding rule (see Section 4) is satisfied. Indeed: (a) for $i = 1, \ldots, k$, t_i is a term of one of the following forms: 0, $s(n)$, $[\,]$, $[\mathtt{t}|S]$, $[\mathtt{f}|S]$ (because, by Lemma 3, every clause in *InDefs* is a natset-typed definition and the clauses in U_1 are derived by unfolding a clause in *InDefs* w.r.t. all atoms occurring positively in its body and, in particular, w.r.t. all type atoms), and (b) the head $p(h_1, \ldots, h_k)$ of any clause in *TransfP* is a linear atom where $h_1, \ldots, h_k$ are head terms (by Lemma 2 and the definition of regular natset-typed clauses).
(B.2) Condition (ii) of Case (2) of the unfolding rule is satisfied because *TransfP* is a regular natset-typed program (see Lemma 2) and thus, no clause in *TransfP* has existential variables.

Now, the transformation sequence constructed by the unfold/fold transformation strategy satisfies the hypothesis of Theorem 3. Indeed, every clause D in *InDefs* is a natset-typed definition with at least one type atom in its body and during an application of the unfold/fold transformation strategy D is unfolded w.r.t. all type atoms (see Point (i) of the Unfolding phase). Thus, by Theorem 3, we have that $M(P \cup \mathit{Defs}) = M(\mathit{TransfP})$. Observe that $\mathit{Def}^*(f, P \cup \mathit{Defs}) = \mathit{Def}^*(f, P \cup \{D_1, \ldots, D_s\})$ and, therefore, $M(P \cup \{D_1, \ldots, D_s\}) \models f(t_1, \ldots, t_n)$ iff $M(P \cup \mathit{Defs}) \models f(t_1, \ldots, t_n)$ iff $M(\mathit{TransfP}) \models f(t_1, \ldots, t_n)$.

Finally, let us prove Point (3.2). If $n = 0$, by Point (2) of this theorem, $\mathit{Def}^*(f, \mathit{TransfP})$ is either $\emptyset$ or $\{f \leftarrow\}$. Thus, *TransfP* terminates for the query f. If $n > 0$, by Point (1) of this theorem, *TransfP* is a natset-typed program and thus, by Lemma 1, *TransfP* terminates for the ground query $f(t_1, \ldots, t_n)$. □

Theorem 5. The unfold/fold transformation strategy terminates.

Proof. We have to show that the WHILE statement in the body of the FOR statement terminates.

Each execution of the Unfolding phase terminates. Indeed, (a) the number of applications of the unfolding rule at Points (i) and (ii) is finite, because *InDefs* is a finite set of clauses and the body of each clause has a finite number of literals, and (b) at Point (iii) only a finite number of unfolding steps can be applied w.r.t. ground literals, because the value of *TransfP* during the Unfolding phase is a program which terminates for every ground query (by Lemma 1).

Each execution of the Definition-Folding phase terminates because a finite number of clauses are introduced by definition and a finite number of clauses are folded.

Thus, in order to show that the strategy terminates, it is enough to show that after a finite number of executions of the body of the WHILE statement, we get $InDefs = \emptyset$. Let $Defs_j$ and $InDefs_j$ be the values of $Defs$ and $InDefs$, respectively, at the end of the j-th execution of the body of the WHILE statement. If the WHILE statement terminates after z executions of its body, then, for all $j > z$, we define $Defs_j$ to be $Defs_z$ and $InDefs_j$ to be $\emptyset$. We have that, for any $j \geq 1$, $InDefs_j = \emptyset$ iff $Defs_{j-1} = Defs_j$. Since for all $j \geq 1$, $Defs_{j-1} \subseteq Defs_j$, the termination of the strategy is a consequence of the following property:

$$\text{there exists } K > 0 \text{ such that, for all } j \geq 1, |Defs_j| \leq K \qquad (\dagger)$$

Let $TransfP_0$, $Defs_0$, and $InDefs_0$ ($\subseteq Defs_0$) be the values of $TransfP$, $Defs$, and $InDefs$, respectively, at the beginning of the execution of the WHILE statement. Property (†) is a consequence of the fact that, for all $D \in Defs_j$, the following properties hold:

(a) every predicate occurring in $bd(D)$ also occurs in $TransfP_0 \cup InDefs_0$;

(b) for every literal L occurring in $bd(D)$,

$height(L) \leq \max\{height(M) \mid M \text{ is a literal in the body of a clause in } Defs_0\}$

where the *height* of a literal is defined to be the length of the maximal path from the root to a leaf of the literal considered as a tree;

(c) $|vars(D)| \leq \max\{vars(D') \mid D' \text{ is a clause in } Defs_0\}$;

(d) no two clauses in $Defs_j$ can be made equal by one or more applications of the following transformations: renaming of variables, renaming of head predicates, rearrangement of the order of the literals in the body, and deletion of duplicate literals.

Properties (a)–(d) can be proved by using the following facts: (i) $bd(D)$ is equal to $bd(E')$ where E' is derived by unfolding (according to the Unfolding phase of the strategy) a clause E in $TransfP_0 \cup InDefs_j$ and E belongs to $InDefs_j$, (ii) by Lemma 2, $TransfP_0$ is a regular natset-typed program, and (iii) by Lemma 3, for all $j \geq 1$, $InDefs_j$ is a set of natset-typed definitions. □

6 Deciding WS1S via the Unfold/Fold Proof Method

In this section we show that if we start from a *closed* WS1S formula φ, our synthesis method can be used for checking whether or not $\mathcal{N} \models \varphi$ holds and, thus, our synthesis method works also as a proof method which is a decision procedure for closed WS1S formulas.

If φ is a *closed* WS1S formula then the predicate f introduced when constructing the set $Cls(f, \varphi_\tau)$, is a nullary predicate. Let $TransfP$ be the program derived by the unfold/fold transformation strategy starting from the program $NatSet \cup Cls(f, \varphi_\tau)$. As already known from Point (2) of Theorem 4, we have that $Def^*(f, TransfP)$ is either the empty set or the singleton $\{f \leftarrow\}$. Thus, we can decide whether or not $\mathcal{N} \models \varphi$ holds by checking whether or not $f \leftarrow$ belongs to $TransfP$. Since the unfold/fold transformation strategy always terminates, we

have that our unfold/fold synthesis method is indeed a decision procedure for closed WS1S formulas. We summarize our proof method as follows.

The Unfold/Fold Proof Method.
Let φ be a closed WS1S formula.
Step 1. We apply the typed Lloyd-Topor transformation and we derive the set $\mathit{Cls}(f, \varphi_\tau)$ of clauses.
Step 2. We apply the unfold/fold transformation strategy and from the program $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ we derive a definite program *TransfP*.
If the unit clause $f \leftarrow$ belongs to *TransfP* then $\mathcal{N} \models \varphi$ else $\mathcal{N} \models \neg\varphi$.

Now we present a simple example of application of our unfold/fold proof method.

Example 1. (*Existence of a Larger Number.*) Let us consider the closed WS1S formula $\varphi : \forall X\, \exists Y\, X \leq Y$. By applying the typed Lloyd-Topor transformation starting from the statement $f \leftarrow \varphi$, we get the following set of clauses $\mathit{Cls}(f, \varphi_\tau)$:

1. $h(X) \leftarrow \mathit{nat}(X) \wedge \mathit{nat}(Y) \wedge X \leq Y$
2. $g \leftarrow \mathit{nat}(X) \wedge \neg h(X)$
3. $f \leftarrow \neg g$

Now we apply the unfold/fold transformation strategy to the program *NatSet* and the following hierarchy of natset-typed definitions: clause 1, clause 2, clause 3. Initially, the program *TransfP* is *NatSet*. The transformation strategy proceeds left-to-right over that hierarchy.

(1) *Defs* and *InDefs* are both set to {clause 1}.
(1.1) *Unfolding.* By unfolding, from clause 1 we get:

4. $h(0) \leftarrow$
5. $h(0) \leftarrow \mathit{nat}(Y)$
6. $h(s(X)) \leftarrow \mathit{nat}(X) \wedge \mathit{nat}(Y) \wedge X \leq Y$

(1.2) *Definition-Folding.* In order to fold the body of clause 5 we introduce the following new clause:

7. $\mathit{new}1 \leftarrow \mathit{nat}(Y)$

Clause 6 can be folded by using clause 1. By folding clauses 5 and 6 we get:

8. $h(0) \leftarrow \mathit{new}1$
9. $h(s(X)) \leftarrow h(X)$

(1.3) At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{\text{clause 4, clause 8, clause 9}\}$, $\mathit{Defs} = \{\text{clause 1, clause 7}\}$, and $\mathit{InDefs} = \{\text{clause 7}\}$.
(1.4) By first unfolding clause 7 and then folding using clause 7 itself, we get:

10. $\mathit{new}1 \leftarrow$
11. $\mathit{new}1 \leftarrow \mathit{new}1$

No new clause is introduced (i.e., $\mathit{NewDefs} = \emptyset$). At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{\text{clause 4, clause 8, clause 9, clause 10, clause 11}\}$, $\mathit{Defs} = \{\text{clause 3, clause 7}\}$, and $\mathit{InDefs} = \emptyset$. Thus, the WHILE statement terminates.
Since $\mathit{Def}^*(\mathit{new}1, \mathit{TransfP})$ is propositional and $M(\mathit{TransfP}) \models \mathit{new}1$, by the propositional simplification rule we have:

$\mathit{TransfP} = \mathit{NatSet} \cup \{\text{clause 4, clause 8, clause 9, clause 10}\}$.

(2) *Defs* is set to {clause 1, clause 2, clause 7} and *InDefs* is set to {clause 2}.
(2.1) *Unfolding.* By unfolding, from clause 2 we get:
12. $g \leftarrow nat(X) \wedge \neg h(X)$
(Notice that, by unfolding, clause $g \leftarrow \neg h(0)$ is deleted.)
(2.2) *Definition-Folding.* Clause 12 can be folded by using clause 2 which occurs in *Defs*. Thus, no new clause is introduced (i.e., $NewDefs = \emptyset$) and by folding we get:
13. $g \leftarrow g$
(2.3) At this point $TransfP = NatSet \cup$ {clause 4, clause 8, clause 9, clause 10, clause 13}, $Defs =$ {clause 1, clause 2, clause 7}, and $InDefs = \emptyset$. Thus, the WHILE statement terminates.
Since $Def^*(g, TransfP)$ is propositional and $M(TransfP) \models \neg g$, by the propositional simplification rule we delete clause 13 from *TransfP* and we have:
$TransfP = NatSet \cup$ {clause 4, clause 8, clause 9, clause 10}.
(3) *Defs* is set to {clause 1, clause 2, clause 3, clause 7} and *InDefs* is set to {clause 3}.
(3.1) *Unfolding.* By unfolding clause 3 we get:
14. $f \leftarrow$
(Recall that, there is no clause in *TransfP* with head g.)
(3.2) *Definition-Folding.* No transformation steps are performed on clause 14 because it is a unit clause.
(3.3) At this point $TransfP = NatSet \cup$ {clause 4, clause 8, clause 9, clause 10, clause 14}, $Defs =$ {clause 1, clause 2, clause 3, clause 7}, and $InDefs = \emptyset$.
The transformation strategy terminates and, since the final program *TransfP* includes the unit clause $f \leftarrow$, we have proved that $\mathcal{N} \models \forall X\, \exists Y\, X \leq Y$.

We would like to notice that neither SLDNF nor Tabled Resolution (as implemented in the XSB system [21]) are able to construct a refutation of $NatSet \cup Cls(f, \varphi_\tau) \cup \{\leftarrow f\}$ (and thus construct a proof of φ), where φ is the WS1S formula $\forall X\, \exists Y\, X \leq Y$. Indeed, from the goal $\leftarrow f$ the new goal $\leftarrow \neg g$ is generated, and neither SLDNF nor Tabled Resolution are able to infer that $\leftarrow \neg g$ succeeds by detecting that $\leftarrow g$ generates an infinite set of failed derivations. □

We would like to mention that some other transformations could be applied for enhancing our unfold/fold transformation strategy. In particular, during the strategy we may apply the subsumption rule to shorten the transformation process by deleting some useless clauses. For instance, in Example 1 we can delete clause 5 which is subsumed by clause 4, thereby avoiding the introduction of the new predicate *new*1. In some other cases we can drop unnecessary type atoms. For instance, in Example 1 in clause 1 the type atom $nat(X)$ can be dropped because it is implied by the atom $X \leq Y$. The program derived at the end of the execution of the WHILE statement of the unfold/fold transformation strategy are nondeterministic, in the sense that an atom with non-variable arguments may be unifiable with the head of several clauses. We can apply the technique presented in [18] for deriving deterministic programs thereby reducing the number of heads which are unifiable at each unfolding step.

When the unfold/fold transformation strategy is used for program synthesis, it is often the case that the above mentioned transformations also improve the efficiency of the derived programs.

Finally, we would like to notice that the unfold/fold transformation strategy can be applied starting from a program $P \cup \mathit{Cls}(f, \varphi_\tau)$ (instead of $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$) where: (i) P is the output of a previous application of the strategy, and (ii) φ is a formula built like a WS1S formula, except that it uses predicates occurring in P (besides $\leq$ and $\in$). Thus, we can synthesize programs (or construct proofs) in a *compositional* way, by first synthesizing programs for subformulas. We will follow this compositional methodology in the example of the following Section 7.

7 An Application to the Verification of Infinite State Systems: The Dynamic Bakery Protocol

In this section we present an example of verification of a safety property of an infinite state system by considering CLP(WS1S) programs [10]. As already mentioned, by applying our unfold/fold synthesis method we will then translate CLP(WS1S) programs into logic programs.

The syntax of CLP(WS1S) programs is defined as follows. We consider a set of *user-defined* predicate symbols. A CLP(WS1S) clause is of the form $A \leftarrow \varphi \wedge G$, where A is an atom, φ is a formula of WS1S, G is a conjunction of literals, and the predicates occurring in A or in G are all user-defined. A CLP(WS1S) *program* is a set of CLP(WS1S) clauses. We assume that CLP(WS1S) programs are stratified. Given a CLP(WS1S) program P, we define the semantics of P to be its perfect model, denoted $M(P)$ (here we extend to CLP(WS1S) programs the definitions which are given for normal logic programs in [1]).

Our example concerns the Dynamic Bakery protocol, called *DBakery* for short, and we prove that it ensures mutual exclusion in a system of processes which share a common resource, even if the number of processes in the system changes during a protocol run in a dynamic way. The *DBakery* protocol is a variant of the N-process Bakery protocol [12].

In order to give the formal specifications of the *DBakery* protocol and its mutual exclusion property, we will use CLP(WS1S) as we now indicate. The transition relation between pairs of system states, the initial system state, and the system states which are *unsafe* (that is, the system states where more than one process uses the shared resource) are specified by WS1S formulas. However, in order to specify the mutual exclusion property we cannot use WS1S formulas only. Indeed, mutual exclusion is a reachability property which is undecidable in the case of infinite state systems. The approach we follow in this example is to specify reachability (and, thus, mutual exclusion) as a CLP(WS1S) program (see the program $P_{DBakery}$ below).

Let us first describe the *DBakery* protocol. We assume that every process is associated with a natural number, called a *counter*, and two distinct processes have distinct counters. At each instant in time, the system of processes is represented by a pair $\langle W, U \rangle$, called a *system state*, where W is the set of the counters

of the processes *waiting* for the resource, and U is the set of the counters of the processes *using* the resource.

A system state $\langle W, U\rangle$ is *initial* iff $W \cup U$ is the empty set. The transition relation from a system state $\langle W, U\rangle$ to a new system state $\langle W', U'\rangle$ is the union of the following three relations:

(T1: *creation of a process*)
if $W \cup U$ is empty *then* $\langle W', U'\rangle = \langle\{0\}, \emptyset\rangle$ *else* $\langle W', U'\rangle = \langle W \cup \{m{+}1\}, U\rangle$, where m is the maximum counter in $W \cup U$,

(T2: *use of the resource*)
if there exists a counter n in W which is the minimum counter in $W \cup U$ *then* $\langle W', U'\rangle = \langle W - \{n\}, U \cup \{n\}\rangle$,

(T3: *release of the resource*)
if there exists a counter n in U *then* $\langle W', U'\rangle = \langle W, U - \{n\}\rangle$.

The mutual exclusion property holds iff from the initial system state it is not possible to reach a system state $\langle W, U\rangle$ which is *unsafe*, that is, such that U is a set of at least two counters.

Let us now give the formal specification of the *DBakery* protocol and its mutual exclusion property. We first introduce the following WS1S formulas (between parentheses we indicate their meaning):

$e(X) \equiv \neg\exists x\ x{\in}X$ (the set X is empty)
$max(X,m) \equiv m{\in}X \wedge \forall x\ (x{\in}X \rightarrow x{\leq}m)$ (m is the maximum in the set X)
$min(X,m) \equiv m{\in}X \wedge \forall x\ (x{\in}X \rightarrow m{\leq}x)$ (m is the minimum in the set X)

(Here and in what follows, for reasons of readability, we allow ourselves to use lower case letters for individual variables of WS1S formulas.)
A system state $\langle W, U\rangle$ is *initial* iff $\mathcal{N} \models init(\langle W, U\rangle)$, where:
$(\alpha)\ \ init(\langle W, U\rangle) \equiv e(W) \wedge e(U)$
The transition relation R between system states is defined as follows:
$\langle\langle W, U\rangle, \langle W', U'\rangle\rangle \in R$ iff
$\mathcal{N} \models cre(\langle W, U\rangle, \langle W', U'\rangle) \vee use(\langle W, U\rangle, \langle W', U'\rangle) \vee rel(\langle W, U\rangle, \langle W', U'\rangle)$
where the predicates *cre*, *use*, and *rel* define the transition relations T1, T2, and T3, respectively. We have that:

$(\tau 1)\ \ cre(\langle W, U\rangle, \langle W', U'\rangle) \equiv U'{=}U \wedge \exists Z(Z{=}W{\cup}U \wedge ((e(Z) \wedge W'{=}\{0\}) \vee (\neg e(Z) \wedge \exists m(max(Z,m) \wedge W'{=}W{\cup}\{s(m)\}))))$

$(\tau 2)\ \ use(\langle W, U\rangle, \langle W', U'\rangle) \equiv \exists n\,(n \in W \wedge \exists Z\,(Z{=}W \cup U \wedge min(Z,n)) \wedge W'{=}W{-}\{n\} \wedge U'{=}U{\cup}\{n\})$

$(\tau 3)\ \ rel(\langle W, U\rangle, \langle W', U'\rangle) \equiv W'{=}W \wedge \exists n\,(n \in U \wedge U'{=}U{-}\{n\})$

where the subformulas involving the set union ($\cup$), set difference ($-$), and set equality ($=$) operators can be expressed as WS1S formulas.

Mutual exclusion holds in a system state $\langle W, U\rangle$ iff $\mathcal{N} \models \neg unsafe(\langle W, U\rangle)$, where $unsafe(\langle W, U\rangle) \equiv \exists n_1\,\exists n_2\,(n_1{\in}U \wedge n_2{\in}U \wedge \neg(n_1{=}n_2))$, i.e., a system state $\langle W, U\rangle$ is unsafe iff there exist at least two distinct counters in U.

Now we will specify the system states reached from a given initial system state by introducing the following CLP(WS1S):

$$
\begin{array}{ll}
P_{DBakery}: & reach(S) \leftarrow init(S) \\
& reach(S1) \leftarrow cre(S, S1) \wedge reach(S) \\
& reach(S1) \leftarrow use(S, S1) \wedge reach(S) \\
& reach(S1) \leftarrow rel(S, S1) \wedge reach(S)
\end{array}
$$

where $init(S)$, $cre(S, S1)$, $use(S, S1)$, and $rel(S, S1)$ are the WS1S formulas $\alpha, \tau 1, \tau 2$, and $\tau 3$ listed above.

From $P_{DBakery}$ we derive a definite program $P'_{DBakery}$ by replacing the WS1S formulas occurring in $P_{DBakery}$ by the corresponding atoms $init(S)$, $cre(S, S1)$, $use(S, S1)$, and $rel(S, S1)$, and by adding to the program the clauses (not listed here) defining these four atoms. These clauses can be derived from the corresponding WS1S formulas $\alpha, \tau 1, \tau 2$, and $\tau 3$, by applying the unfold/fold synthesis method (see Section 5). Let us call these clauses *Init*, *Cre*, *Use*, and *Rel*, respectively.

In order to verify that the *DBakery* protocol ensures mutual exclusion for every system of processes whose number dynamically changes over time, we have to prove that for every ground term s denoting a finite set of counters, $ur(s) \notin M(P'_{DBakery} \cup \{\text{clause } 1\})$, where clause 1 is the following clause which we introduce by the definition rule:

1. $ur(S) \leftarrow unsafe(S) \wedge reach(S)$

and $unsafe(S)$ is defined by a set, called *Unsafe*, of clauses which can be derived from the corresponding WS1S formula by using the unfold/fold synthesis method.

In order to verify the mutual exclusion property for the *DBakery* protocol it is enough to show that $P'_{DBakery} \cup \{\text{clause } 1\}$ can be transformed into a new definite program without clauses for $ur(S)$. This transformation can be done, as we now illustrate, by a straightforward adaptation of the proof technique presented for Constraint Logic Programs in [7]. In particular, before performing folding steps, we will add suitable atoms in the bodies of the clauses to be folded.

We start off this verification by unfolding clause 1 w.r.t. the atom *reach*. We obtain the following clauses:

2. $ur(S) \leftarrow unsafe(S) \wedge init(S)$
3. $ur(S1) \leftarrow unsafe(S1) \wedge cre(S, S1) \wedge reach(S)$
4. $ur(S1) \leftarrow unsafe(S1) \wedge use(S, S1) \wedge reach(S)$
5. $ur(S1) \leftarrow unsafe(S1) \wedge rel(S, S1) \wedge reach(S)$

Now we can remove clause 2 because $M(Unsafe \cup Init) \models \neg \exists S\, (unsafe(S) \wedge init(S))$. The proof of this facts and the proofs of the other facts we state below, are performed by applying the unfold/fold proof method of Section 5. Then, we fold clauses 3 and 5 by using the definition clause 1 and we obtain:

6. $ur(S1) \leftarrow unsafe(S1) \wedge cre(S, S1) \wedge ur(S)$
7. $ur(S1) \leftarrow unsafe(S1) \wedge rel(S, S1) \wedge ur(S)$

Notice that this application of the folding rule is justified by the following two facts:

$M(Unsafe \cup Cre) \models \forall S \forall S1\ (unsafe(S1) \wedge cre(S, S1) \rightarrow unsafe(S))$
$M(Unsafe \cup Rel) \models \forall S \forall S1\ (unsafe(S1) \wedge rel(S, S1) \rightarrow unsafe(S))$

so that, before folding, we can add the atom $unsafe(S)$ to the bodies of clauses 3 and 5. Now, since $M(Unsafe \cup Use) \models \neg\forall S\,\forall S1\,(unsafe(S1) \wedge use(S, S1) \rightarrow unsafe(S))$, clause 4 *cannot* be folded using the definition clause 1. Thus, we introduce the new definition clause:

8. $p1(S) \leftarrow c(S) \wedge reach(S)$

where $c(\langle W, U\rangle) \equiv \exists n\,(n \in W \wedge \exists Z\,(Z = W \cup U \wedge min(Z, n))) \wedge \neg e(U)$ which means that: in the system state $\langle W, U\rangle$ there is at least one process which uses the resource and there exists a process waiting for the resource with counter n which is the minimum counter in $W \cup U$.

Notice that, by applying the unfold/fold synthesis method, we may derive a set, called *Busy* (not listed here), of definite clauses which define $c(S)$.

By using clause 8 we fold clause 4, and we obtain:

9. $ur(S1) \leftarrow unsafe(S1) \wedge use(S, S1) \wedge p1(S)$

We proceed by applying the unfolding rule to the newly introduced clause 8, thereby obtaining:

10. $p1(S) \leftarrow c(S) \wedge init(S)$
11. $p1(S1) \leftarrow c(S1) \wedge cre(S, S1) \wedge reach(S)$
12. $p1(S1) \leftarrow c(S1) \wedge use(S, S1) \wedge reach(S)$
13. $p1(S1) \leftarrow c(S1) \wedge rel(S, S1) \wedge reach(S)$

Clauses 10 and 12 are removed, because

$M(Busy \cup Init) \models \neg\exists S\,(c(S) \wedge init(S))$
$M(Busy \cup Use) \models \neg\exists S\,\exists S1\,(c(S1) \wedge use(S, S1))$

We fold clauses 11 and 13 by using the definition clauses 8 and 1, respectively, thereby obtaining:

14. $p1(S1) \leftarrow c(S1) \wedge \ cre(S, S1) \wedge p1(S)$
15. $p1(S1) \leftarrow c(S1) \wedge \ rel(S, S1) \wedge ur(S)$

Notice that this application of the folding rule is justified by the following two facts:

$M(Busy \cup Cre) \models \forall S\,\forall S1\ ((c(S1) \wedge cre(S, S1)) \rightarrow c(S))$
$M(Busy \cup Rel) \models \forall S\,\forall S1\ ((c(S1) \wedge rel(S, S1)) \rightarrow unsafe(S))$

Thus, starting from program $P'_{DBakery} \cup \{\text{clause 1}\}$ we have derived a new program Q consisting of clauses 6, 7, 14, and 15. Since all clauses in $Def^*(ur, Q)$ are recursive, we have that for every ground term s denoting a finite set of counters, $ur(s) \notin M(Q)$ and by the correctness of the transformation rules [17], we conclude that mutual exclusion holds for the *DBakery* protocol.

8 Related Work and Conclusions

We have proposed an automatic synthesis method based on unfold/fold program transformations for translating CLP(WS1S) programs into normal logic programs. This method can be used for avoiding the use of ad-hoc solvers for WS1S constraints when constructing proofs of properties of infinite state multi-process systems.

Our synthesis method follows the general approach presented in [17] and it terminates for any given WS1S formula. No such termination result was given in [17]. In this paper we have also shown that, when we start from a closed WS1S

formula φ, our synthesis strategy produces a program which is either (i) a unit clause of the form $f \leftarrow$, where f is a nullary predicate equivalent to the formula φ, or (ii) the empty program. Since in Case (i) φ is true and in Case (ii) φ is false, our strategy is also a decision procedure for closed WS1S formulas. This result extends [16] which presents a decision procedure based on the unfold/fold proof method for the *clausal fragment* of the WSkS theory, i.e., the fragment dealing with universally quantified disjunctions of conjunctions of literals.

Some related methods based on program transformation have been recently proposed for the verification of infinite state systems [13,20]. However, as it is shown by the example of Section 7, an important feature of our verification method is that the number of processes involved in the protocol may change over time and other methods find it problematic to deal with such dynamic changes. In particular, the techniques presented in [20] for verifying safety properties of *parametrized systems* deal with reactive systems where the number of processes is a parameter which does not change over time.

Our method is also related to a number of other methods which use logic programming and, more generally, constraint logic programming for the verification of reactive systems (see, for instance, [6,8,15,19]). The main novelty of our approach w.r.t. these methods is that it combines logic programming and monadic second order logic, thereby modelling in a very direct way systems with an unbounded (and possibly variable) number of processes.

Our unfold/fold synthesis method and our unfold/fold proof method have been implemented by using the MAP transformation system [23]. Our implementation is reasonably efficient for WS1S formulas of small size (see the example formulas of Section 7). However, our main concern in the implementation was not efficiency and the performance of our system should not be compared with ad-hoc, well-established theorem provers for WS1S formulas based on automata theory, like the MONA system [9]. Nevertheless, we believe that our technique has its novelty and deserves to be developed because, being based on unfold/fold transformation rules, it can easily be combined with other techniques for program derivation, specialization, synthesis, and verification, which are also based on unfold/fold transformation rules.

Acknowledgements

We thank the participants of LOPSTR'02 and the referees for helpful comments.

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
2. D. Basin and S. Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke (eds.), *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pp. 39–56. Research Studies Press/Wiley, 2000.
3. D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3):255–288, 1998.

4. J. R. Büchi. Weak second order arithmetic and and finite automata. *Z. Maath Logik Grundlagen Math*, 6:66–92, 1960.
5. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
6. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland (ed.), *Proc. TACAS'99*, LNCS 1579, pp. 223–239. Springer, 1999.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In A. Pettorossi (ed.), *Proc. LOPSTR 2001*, LNCS 2372, pp. 111–128. Springer, 2002.
8. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proc. CONCUR '97*, LNCS 1243, pp. 96–107. Springer, 1997.
9. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma et al. (eds.), *Proc. TACAS '95*, LNCS 1019, pp. 89–110. Springer, 1996.
10. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
11. N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pp. 101–110. ACM, 1996.
12. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, 1974.
13. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi (ed.), *Proc. LOPSTR '99*, LNCS 1817, pp. 63–82. Springer, 1999.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987.
15. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd et al. (eds.), *Proc. CL'2000*, LNAI 1861, pp. 384–398, 2000.
16. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd et al. (eds.), *Proc. CL'2000*, LNAI 1861, pp. 613–628. Springer, 2000.
17. A. Pettorossi and M. Proietti. Program Derivation = Rules + Strategies. In A. Kakas and F. Sadri (eds.), *Computational Logic: Logic Programming and Beyond* (*in honour of Bob Kowalski, Part I*), LNCS 2407, pp. 273–309. Springer, 2002.
18. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th POPL*, pp. 414–427. ACM Press, 1997.
19. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proc. CAV '97*, LNCS 1254, pp. 143–154. Springer, 1997.
20. A. Roychoudhury and I.V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proc. CAV 2001*, pp. 25–37, 2001.
21. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB system, version 2.2., 2000.
22. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.
23. The MAP group. The MAP transformation system. Available from `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2002.
24. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa (eds.), *Handbook of Formal Languages*, volume 3, pp. 389–455. Springer, 1997.

Verification in ACL2 of a Generic Framework to Synthesize SAT–Provers*

F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina

Departamento de Ciencias de la Computación e Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
http://www.cs.us.es/{~fmartin, ~jalonso,~mjoseh,~jruiz}

Abstract. We present in this paper an application of the ACL2 system to reason about propositional satisfiability provers. For that purpose, we present a framework where we define a generic transformation based SAT–prover, and we show how this generic framework can be formalized in the ACL2 logic, making a formal proof of its termination, soundness and completeness. This generic framework can be instantiated to obtain a number of verified and executable SAT–provers in ACL2, and this can be done in an automatized way. Three case studies are considered: semantic tableaux, sequent and Davis–Putnam methods.

1 Introduction

ACL2 [8] is a programming language, a logic for reasoning about programs in the language, and a theorem prover supporting formal reasoning in the logic. These components make ACL2 a particularly suitable system for reasoning about decision procedures, since proving and computing tasks can be done in the same system. Efficiency is one of the design goals of the system. Usually, it is obtained by building specific procedures to solve concrete problems. On the other hand, system characteristics make possible the development of generic procedures based on logic specifications. These generic procedures can be instantiated to obtain concrete ones [9], and this instantiation can be done maintaining efficiency to some extent.

In this paper, we describe an application of the ACL2 system to reason formally about a family of propositional satisfiability decision procedures. The common pattern of these procedures is that they can be described as rule based transformation systems. For that purpose, we develop a generic framework into which these SAT–provers can be placed. A generic SAT–prover is formalized in ACL2 and its main properties are proved; using functional instantiation, concrete instances of the generic framework can be defined to obtain formally verified and Common Lisp executable SAT–provers. We will also describe how this instantiation process can be automatized. Three case studies are considered: semantic tableaux, sequent calculus and the Davis–Putnam method.

* This work has been supported by project TIC2000-1368-C03-02 (Ministry of Science and Technology, Spain)

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 182–198, 2003.

This paper is organized as follows. In section 2 we define a generic framework in order to build a generic transformation based SAT–prover, and we sketch a proof of its termination, soundness and completeness properties. We also describe how three well–known SAT–provers methods (tableaux, sequent calculus and Davis–Putnam method) can be placed into the generic framework. In section 3 we show how this framework can be formalized in ACL2 and how its main properties can be established. In section 4 we describe how these generic definitions and theorems can be instantiated in an automatized way, to obtain verified and executable Common Lisp definitions of tableaux based, sequent based and Davis–Putnam SAT–provers. Finally, in section 5 we draw some conclusions and discuss future work.

2 A Generic Framework to Develop Propositional SAT–Provers

Analyzing some well–known methods of proving propositional satisfiability (such as sequents, tableaux or Davis–Putnam), we can observe a common behavior. They do not work directly on formulas but on objects built from formulas. The objects are repeatedly modified using expansion rules, reducing their complexity in such a way that their meaning is preserved. Eventually, from some kind of simple objects, a *distinguished valuation* proving satisfiability of the original formula can be obtained. If no such object is found, then unsatisfiability of the original formula is proved.

We can see this behavior in the semantic tableaux method with an example (figure 1–left). From the formula $(p \to q) \wedge p$ a tree with a single node is built. In a first step the formula is expanded obtaining one extension with two formulas $p \to q$ and p. In a second step the formula $p \to q$ is expanded obtaining two extensions, the first with the formula $\neg p$ and the second with the formula q. The left branch becomes closed (with complementary literals) and the right one provides a model σ. Thus, the tableaux method can be seen as the application of a set of expansion rules acting on branches of trees (the objects) until a branch without complementary literals is obtained. For this branch, a distinguished valuation (making that branch true) is easily obtained. Otherwise, all branches are closed and unsatisfiability is proved. In figure 1–right we can see how the sequent method behaves in a similar way, where objects are now sequents.

So our goal in this section is to describe a generic framework where these methods can be placed. First we introduce some notation. We consider an infinite set of symbols Σ and a set of truth values, $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$, where $\mathbf{t}$ denotes *true* and $\mathbf{f}$ denotes *false*. $\mathbb{P}(\Sigma)$ denotes the set of propositional formulas on Σ (the truth values are not considered as formulas), where the basic connectives are $\neg$, $\wedge$, $\vee$, $\to$ and $\leftrightarrow$. $\overline{F}$ denotes the complementary of a formula F. A literal is a formula p or $\neg p$, where $p \in \Sigma$. A clause is a finite sequence of literals. A valuation is a function $\sigma : \Sigma \longrightarrow \mathbb{B}$, we denote $\mathbb{V}_\Sigma$ the set of all valuations defined on Σ. The valuations are extended to $\mathbb{P}(\Sigma)$ in the usual way. We denote $\sigma \models F$ when $\sigma(F) = \mathbf{t}$, and we say that σ is a model of F. A valuation σ is a model

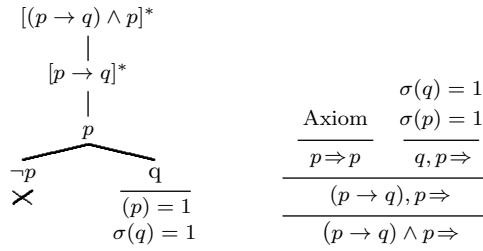


Fig. 1. An example of tableaux and sequents methods

of a clause C, if it is a model of some literal in C. The capital Greek letters Γ and Δ (possibly with subscripts) denote sequences of formulas. We will use the notation $\langle e_1, ..., e_k \rangle$ to represent a finite sequence, and O^* to denote the set of sequences of elements from the set O. We write $\langle \Gamma_1, F, \Gamma_2 \rangle$ or Γ_1, F, Γ_2, to distinguish the formula F in a sequence of formulas. Finally, $\mathcal{O}rd$ denotes the class of all ordinals.

Definition 1. *A* Propositional Transformation System *(for short, PTS) is a triple $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, where $\mathcal{O}$, $\mathcal{R}$ and $\mathcal{V}$ are sets such that $\mathcal{R} \subseteq \mathcal{O} \times (\mathcal{O}^* \cup \{\mathbf{t}\})$ and $\mathcal{V} \subseteq \mathcal{O} \times \mathbb{V}_\Sigma$.*

We will call $\mathcal{O}$ the set of *propositional objects* (or simply *objects*) and $\mathcal{R}$ the set of *expansion rules*. An element $(O, L) \in \mathcal{R}$ will be denoted as $O \rightsquigarrow_\mathcal{G} L$. Note that we allow rules of the form $O \rightsquigarrow_\mathcal{G} \langle\rangle$ and rules of the form $O \rightsquigarrow_\mathcal{G} \mathbf{t}$. When $(O, \sigma) \in \mathcal{V}$ we say that σ is a *distinguished valuation* for O, denoted as $\sigma \models_\mathcal{G} O$.

Definition 2. *Given a PTS $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$:*

1. *A* computation rule *is a function $r : \mathcal{O} \longrightarrow \mathcal{O}^* \cup \{\mathbf{t}\}$ such that $r \subseteq \mathcal{R}$.*
2. *A* representation function *is a function $i : \mathbb{P}(\Sigma) \longrightarrow \mathcal{O}$.*
3. *A* measure function *is a function $\mu : \mathcal{O} \longrightarrow \mathcal{O}rd$.*
4. *A* model function *is a function $\sigma : \mathcal{O}_\mathbf{t} \longrightarrow \mathbb{V}_\Sigma$, where $\mathcal{O}_\mathbf{t} = \{O \in \mathcal{O} : O \rightsquigarrow_\mathcal{G} \mathbf{t}\}$.*

Given a PTS $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, a computation rule r and a representation function i, we define the following algorithm $SAT_\mathcal{G}$ for proving satisfiability of a propositional formula.

Algorithm 3 ($SAT_\mathcal{G}$) *The input to this algorithm is a propositional formula F and it proceeds as follows:*

1. *Initially the list of objects $\langle i(F) \rangle$ is considered.*
2. *Given a list of objects $\langle O_1, ..., O_n \rangle$, an element O_j is selected.*
   (a) *If $r(O_j) = \mathbf{t}$, then the algorithm stops returning $\langle O_j \rangle$.*
   (b) *If $r(O_j) = \langle O'_1, ..., O'_m \rangle$, then the algorithm returns to point 2 with the list $\langle O'_1, ..., O'_m, O_1, ..., O_{j-1}, O_{j+1}, ..., O_n \rangle$.*
3. *If the list of objects becomes empty, the algorithm stops returning $\mathbf{f}$.*

The intuitive idea is simple: given F, we start with the initial object $i(F)$ and repeatedly apply the rules of $\mathcal{R}$ until $\mathbf{t}$ is obtained or until there are no more objects left (termination of this process will be guaranteed by a measure function μ). In the first case, from an object O_j such that $r(O_j) = \mathbf{t}$ we can obtain a distinguished valuation using a model function, which turns out to be a model of the original formula. In the second case, the original formula is unsatisfiable. Let us now establish the main properties of this generic algorithm.

Definition 4. *We say that* $SAT_{\mathcal{G}}$ *is* complete *if for all* $F \in \mathbb{P}(\Sigma)$ *such that* $\exists \sigma : \sigma \models F$, *then* $SAT_{\mathcal{G}}(F) \neq \mathbf{f}$. *We say that it is* sound *if for all* $F \in \mathbb{P}(\Sigma)$ *such that* $SAT_{\mathcal{G}}(F) \neq \mathbf{f}$, *then* $\exists \sigma : \sigma \models F$.

Theorem 1. *Let* $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$ *be a PTS,* r *a computation rule,* i *a representation function,* μ *a measure function and* σ *a model function, such that the following properties holds:*

$\mathcal{P}_1$*:* $O_i \in r(O) \Longrightarrow \mu(O_i) < \mu(O)$
$\mathcal{P}_2$*:* $F \in \mathbb{P}(\Sigma) \Longrightarrow (\sigma \models F \Longleftrightarrow \sigma \models_{\mathcal{G}} i(F))$
$\mathcal{P}_3$*:* $O \in \mathcal{O} \wedge r(O) \neq \mathbf{t} \Longrightarrow (\sigma \models_{\mathcal{G}} O \Longleftrightarrow \exists O_i \in r(O), \sigma \models_{\mathcal{G}} O_i)$
$\mathcal{P}_4$*:* $O \in \mathcal{O} \wedge r(O) = \mathbf{t} \Longrightarrow \sigma(O) \models_{\mathcal{G}} O$

then the algorithm $SAT_{\mathcal{G}}$ *terminates for any formula and is complete and sound. Furthermore, if* $SAT_{\mathcal{G}}(F) = \langle O \rangle$ *then* $\sigma(O) \models F$.

Termination Proof: To prove termination of $SAT_{\mathcal{G}}$ we must prove that point 2 is a finite loop. Assume that the list of objects in point 2 is $\langle O_1, ..., O_n \rangle$, the selected element is O_j and $r(O_j) = \langle O'_1, ..., O'_m \rangle$.

We consider the relation $<_\mu$ in $\mathcal{O}$ defined as follows $O_1 <_\mu O_2$ if and only if $\mu(O_1) < \mu(O_2)$. Obviously, $<_\mu$ is a well founded relation on $\mathcal{O}$. Then, for every k, $O'_k <_\mu O_j$ by $\mathcal{P}_1$. Therefore, the multiset $\{O'_1, ..., O'_m, O_1, ..., O_{j-1}, O_{j+1}, ..., O_n\}$ is smaller than $\{O_1, ..., O_n\}$ with respect to the multiset extension of $<_\mu$ (which is also well-founded as proved in [5]). This proves termination of $SAT_{\mathcal{G}}$.

Completeness Proof: First of all note that, by $\mathcal{P}_3$, if the algorithm reaches point 2–(b), σ is a distinguished valuation of some object in the list considered in point 2 if and only if it is a distinguished valuation of some object in the new list built in point 2–(b).

If $\sigma \models F$ then, by $\mathcal{P}_2$, $\sigma \models_{\mathcal{G}} i(F)$. Then, by the above observation, in every list considered in point 2 exists O such that $\sigma \models_{\mathcal{G}} O$. Therefore the list in point 2 cannot become empty and, as the algorithm terminates, in some step an object O' such that $r(O') = \mathbf{t}$ will be considered. Then $SAT_{\mathcal{G}}(F) = \langle O' \rangle \neq \mathbf{f}$.

Soundness Proof: If $SAT_{\mathcal{G}}(F) = \langle O \rangle$ then $r(O) = \mathbf{t}$ and, by $\mathcal{P}_4$, $\sigma(O) \models_{\mathcal{G}} O$. Then, by the property noted in the completeness proof, in every list considered in point 2 exists O such that $\sigma(O) \models_{\mathcal{G}} O$. Therefore, this holds for the initial list considered $\langle i(F) \rangle$, i.e., $\sigma(O) \models_{\mathcal{G}} i(F)$, and, by $\mathcal{P}_2$, $\sigma(O) \models F$. □

2.1 Semantic Tableaux

We consider the semantic tableaux method as it is described in [6]. The tableaux expansion rules are concisely presented using the uniform notation [14]. Using

this notation, non–literal formulas are classified as doubly negated, α–formulas (equivalent to the conjunction of two components α_1 and α_2) or β–formulas (equivalent the a disjunction of two components, β_1 and β_2)[1].

We now describe the PTS $\mathcal{T} = \langle \mathcal{O}_\tau, \mathcal{R}_\tau, \mathcal{V}_\tau \rangle$ associated with the semantic tableaux method. In this PTS, $\mathcal{O}_\tau$ is the set of tableau branches (represented as lists of formulas), $\mathcal{V}_\tau$ is the set of pairs (θ, σ) such that σ makes true every formula in θ, and $\mathcal{R}_\tau$ the set of rules given by the following rule schemata:

$$
\begin{array}{|l|}
\hline
\mathcal{RT}_1 : \langle \Gamma_1, G, \Gamma_2, \neg G, \Gamma_3 \rangle \rightsquigarrow_\tau \langle\rangle \\
\mathcal{RT}_2 : \langle \Gamma_1, \neg\neg G, \Gamma_2 \rangle \rightsquigarrow_\tau \langle \langle \Gamma_1, G, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_3 : \langle \Gamma_1, \alpha, \Gamma_2 \rangle \rightsquigarrow_\tau \langle \langle \Gamma_1, \alpha_1, \alpha_2, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_4 : \langle \Gamma_1, \beta, \Gamma_2 \rangle \rightsquigarrow_\tau \langle \langle \Gamma_1, \beta_1, \Gamma_2 \rangle, \langle \Gamma_1, \beta_2, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_5 : \Gamma \rightsquigarrow_\tau \mathbf{t} \text{ if } \Gamma \text{ does not have non–literal} \\
\qquad\qquad \text{nor complementary formulas} \\
\hline
\end{array}
$$

We define the representation function i_τ such that for every $F \in \mathbb{P}(\Sigma)$, $i_\tau(F) = \langle F \rangle$. Obviously, $\sigma \models F \iff \sigma \models_\tau i(F)$ (property $\mathcal{P}_2$).

We consider any computation rule, r_τ, such that, for every branch θ, $r_\tau(\theta)$ is a rule from the above table that can be applied to θ.

We define the uniform measure, denoted as u, as follows: $u(F) = 5 * \delta_\leftrightarrow(F) + 2 * (\delta_\wedge(F) + \delta_\vee(F) + \delta_\rightarrow(F)) + \delta_\neg(F)$, where $\delta_\circ(F)$ computes the number of occurrences of the connective $\circ$ in F. This measure has the following properties: $u(\alpha_1) + u(\alpha_2) < u(\alpha)$, $u(\beta_1) < u(\beta)$, $u(\beta_2) < u(\beta)$ and $u(F) < u(\neg\neg F)$. We define the measure function, μ_τ, as the sum of the uniform measure of the formulas in a branch. By the properties of u, the expansion rules reduce the measure of a branch; therefore $\theta_i \in r_\tau(\theta) \Longrightarrow \mu_\tau(\theta_i) < \mu_\tau(\theta)$ (property $\mathcal{P}_1$).

The uniform notation ensures that an α (β) formula is logically equivalent to the conjunction (disjunction) of its components and a doubly negated formula $\neg\neg F$ is also logically equivalent to F. Hence, if $\theta \rightsquigarrow_\tau L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_\tau \theta \iff \exists \theta_i \in L, \sigma \models_\tau \theta_i$. According to the definition of computation rule, this trivially implies property $\mathcal{P}_3$.

Finally, we define the model function σ_τ such that for every branch θ without non–literal nor complementary formulas, $\sigma_\tau(\theta) \models p$ if and only if p is a positive literal occurring in θ. Obviously if $r_\tau(\theta) = \mathbf{t}$ then $\sigma_\tau(\theta) \models_\tau \theta$ (property $\mathcal{P}_4$).

Then, by theorem 1, the algorithm SAT_τ terminates for any formula and is complete and sound. The algorithm applied to the example of figure 1–left performs the following steps (represented as $\longmapsto_{SAT_\tau}$):

$$
\begin{aligned}
\langle\langle (p \rightarrow q) \wedge p \rangle\rangle &\longmapsto_{SAT_\tau} \langle\langle p \rightarrow q, p \rangle\rangle \longmapsto_{SAT_\tau} \\
&\longmapsto_{SAT_\tau} \langle\langle p, \neg p \rangle, \langle p, q \rangle\rangle \longmapsto_{SAT_\tau} \langle\langle p, q \rangle\rangle \longmapsto_{SAT_\tau} \langle\langle p, q \rangle\rangle
\end{aligned}
$$

2.2 Sequents and the Gentzen System

We denote *sequents* as $\Gamma \Rightarrow \Delta$, where Γ and Δ are lists of formulas. An *atomic sequent* is a sequent in which every formula is atomic. A valuation σ makes the

[1] We extend the uniform notation to include equivalence: $F \leftrightarrow G$ is considered a β–formula with components $\beta_1 = F \wedge G$, $\beta_2 = \neg F \wedge \neg G$, and $\neg(F \leftrightarrow G)$ is considered a β–formula with components $\beta_1 = F \wedge \neg G$, $\beta_2 = \neg F \wedge G$.

sequent $\Gamma \Rightarrow \Delta$ true if and only if $\exists X \in \Gamma, (\sigma \not\models X) \vee \exists Y \in \Delta, (\sigma \models Y)$. We will consider the axioms and rules of Gentzen System G' presented in [7], with two additional rules about equivalence:

$$\frac{\Gamma_1, F, G, \Gamma_2 \Rightarrow \Delta \qquad \Gamma_1, \Gamma_2 \Rightarrow F, G, \Delta}{\Gamma_1, F \leftrightarrow G, \Gamma_2 \Rightarrow \Delta} \ (\leftrightarrow\text{: left})$$

$$\frac{F, \Gamma \Rightarrow \Delta_1, G, \Delta_2 \qquad G, \Gamma \Rightarrow \Delta_1, F, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \leftrightarrow G, \Delta_2} \ (\leftrightarrow\text{: right})$$

In order to prove the satisfiability of a formula F, the rules are applied to the initial sequent $F \Rightarrow$ until atomic sequents are obtained. Some atomic sequents provide countermodels of the initial sequent, and hence, models of the original formula. See [7] for more background about the sequent method.

We now describe the PTS $\mathcal{S} = \langle \mathcal{O}_s, \mathcal{R}_s, \mathcal{V}_s \rangle$ associated with the sequents method. In this PTS, $\mathcal{O}_s$ is the set of sequents (represented as pairs of lists of formulas), $\mathcal{V}_s$ is the set of pairs (S, σ) such that σ makes S false and $\mathcal{R}_s$ the set of rules given by the following rule schemata:

$\mathcal{RS}_1 : \langle \Gamma_1, F, \Gamma_2 \rangle \Rightarrow \langle \Delta_1, F, \Delta_2 \rangle \rightsquigarrow_s \langle\rangle$
$\mathcal{RS}_2 : \langle \Gamma_1, \neg F, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle \rangle$
$\mathcal{RS}_3 : \langle \Gamma_1, F \wedge G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_4 : \langle \Gamma_1, F \vee G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_5 : \langle \Gamma_1, F \rightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_6 : \langle \Gamma_1, F \leftrightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, G, \Delta \rangle \rangle$
$\mathcal{RS}_7 : \Gamma \Rightarrow \langle \Delta_1, \neg F, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_8 : \Gamma \Rightarrow \langle \Delta_1, F \wedge G, \Delta_2 \rangle \rightsquigarrow_s \langle \Gamma \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle, \Gamma \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_9 : \Gamma \Rightarrow \langle \Delta_1, F \vee G, \Delta_2 \rangle \rightsquigarrow_s \langle \Gamma \Rightarrow \langle F, G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{10} : \Gamma \Rightarrow \langle \Delta_1, F \rightarrow G, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{11} : \Gamma \Rightarrow \langle \Delta_1, F \leftrightarrow G, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle, \langle G, \Gamma \rangle \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{12} : \Gamma \Rightarrow \Delta \rightsquigarrow_s \mathbf{t}$ if $\Gamma \Rightarrow \Delta$ is an atomic sequent and $\Gamma \cap \Delta = \emptyset$

The representation function i_s builds the sequent $F \Rightarrow$ for every $F \in \mathbb{P}(\Sigma)$. Thus, $\sigma \models F \iff \sigma \models_s i_s(F)$ (property $\mathcal{P}_2$).

We consider any computation rule, r_s, such that, for every sequent S, $r_s(S)$ is a rule from the above table that can be applied to S.

We define the measure function μ_s as the number of occurrences of propositional connectives in a sequent. The expansion rules reduce this number, therefore $S_i \in r_s(S) \Longrightarrow \mu_s(S_i) < \mu_s(S)$ (property $\mathcal{P}_1$).

Given an expansion rule $S \rightsquigarrow_s L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_s S \iff \exists S_i \in L, \sigma \models_s S_i$. By the definition of computation rule, this implies property $\mathcal{P}_3$.

Finally, we define the model function σ_s such that for every atomic non–axiom sequent S, $\sigma_s(S) \models p$ if and only if p occurs in the left part of S. Obviously, if $r_s(S) = \mathbf{t}$ then $\sigma_s(S) \models_s S$ (property $\mathcal{P}_4$).

Then, by theorem 1, the algorithm SAT_s terminates for any formula and is complete and sound. The algorithm applied to the example of figure 1–right performs the following steps (represented as $\longmapsto_{SAT_S}$):

$$\langle\langle(p \to q) \wedge p\rangle \Rightarrow\rangle \longmapsto_{SAT_S} \langle\langle p \to q, p\rangle \Rightarrow\rangle \longmapsto_{SAT_S} \langle p \Rightarrow p, \langle q, p\rangle \Rightarrow\rangle \longmapsto_{SAT_S} \langle\langle q, p\rangle \Rightarrow\rangle \longmapsto_{SAT_S} \langle\langle q, p\rangle \Rightarrow\rangle$$

2.3 Davis–Putnam Method

The Davis–Putnam method is a procedure to decide the satisfiability of a set of clauses. Then, if $\mathcal{FC}$ is a procedure to obtain a set of clauses logically equivalent to a formula F, we can use the Davis–Putnam method to decide the satisfiability of F, applying the method to $\mathcal{FC}(F)$. See [6] for more background about the Davis–Putnam method.

We now describe the PTS $\mathcal{D} = \langle \mathcal{O}_\mathcal{D}, \mathcal{R}_\mathcal{D}, \mathcal{V}_\mathcal{D}\rangle$ associated with the Davis–Putnam method. Now, $\mathcal{O}_\mathcal{D}$ is the set of pairs $\langle S, M\rangle$, where S is a set of clauses and M is a literal list without complementary literals and such that for every L in M, neither L nor $\overline{L}$ is in some clause in S. $\mathcal{V}_\mathcal{D}$ is the set of pairs $(\langle S, M\rangle, \sigma)$ such that σ is model of every clause in S and every literal in M. $\mathcal{R}_\mathcal{D}$ is the set of rules given by the following rule schemata:

$\mathcal{RD}_1 : \langle S, M\rangle \rightsquigarrow_\mathcal{D} \langle\rangle$ if the empty clause is in S
$\mathcal{RD}_2 : \langle S, \langle L_1, \ldots, L_n\rangle\rangle \rightsquigarrow_\mathcal{D} \langle\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle\rangle$
if the unitary clause $\{L\}$ is in S
$\mathcal{RD}_3 : \langle S, \langle L_1, \ldots, L_n\rangle\rangle \rightsquigarrow_\mathcal{D} \langle\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle, \langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n\rangle\rangle\rangle$
where L is a literal in a clause in S
$\mathcal{RD}_4 : \langle\langle\rangle, M\rangle \rightsquigarrow_\mathcal{D} \mathbf{t}$

where S is a set of clauses, $\langle S, M\rangle$ and $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$ are elements in $\mathcal{O}_\mathcal{D}$, and $S_L = \{C - \{\overline{L}\} : C \in S \text{ and } L \notin C\}$ and $S_{\overline{L}} = \{C - \{L\} : C \in S \text{ and } \overline{L} \notin C\}$.

The representation function $i_\mathcal{D}$ builds a pair $\langle \mathcal{FC}(F), \langle\rangle\rangle$, where $\mathcal{FC}$ is assumed to be a correct procedure to obtain a set of clauses logically equivalent to F, that is, $\sigma \models F \iff \sigma \models \mathcal{FC}(F) \iff \sigma \models_\mathcal{D} i_\mathcal{D}(F)$ (property $\mathcal{P}_2$).

We consider a computation rule, $r_\mathcal{D}$, that applies one of the expansion rules schemata in the following preference order $\mathcal{RD}_1$, $\mathcal{RD}_2$, $\mathcal{RD}_3$ and $\mathcal{RD}_4$.

We define the measure function $\mu_\mathcal{D}$, such that, for every pair $\langle S, M\rangle \in \mathcal{O}_\mathcal{D}$, $\mu_\mathcal{D}(\langle S, M\rangle)$ is the total number of literals of the clauses of S. The expansion rules reduce this value, therefore $\langle S_i, M_i\rangle \in r_\mathcal{D}(\langle S, M\rangle) \implies \mu_\mathcal{D}(\langle S_i, M_i\rangle) < \mu_\mathcal{D}(\langle S, D\rangle)$ (property $\mathcal{P}_1$).

Given an expansion rule $\langle S, M\rangle \rightsquigarrow_\mathcal{D} L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_\mathcal{D} \langle S, M\rangle \iff \exists \langle S_i, M_i\rangle \in L, \sigma \models_\mathcal{D} \langle S_i, M_i\rangle$. By the definition of computation rule, this implies property $\mathcal{P}_3$.

Finally, we define the model function $\sigma_\mathcal{D}$ such that for every pair $\langle\langle\rangle, M\rangle$ and $p \in \Sigma$, $\sigma_\mathcal{D}(\langle\langle\rangle, M\rangle) \models p$ if and only if $p \in M$. Obviously, if $r_\mathcal{D}(\langle S, M\rangle) = \mathbf{t}$ then $\sigma_\mathcal{D}(\langle S, M\rangle) \models_\mathcal{D} \langle S, M\rangle$ (property $\mathcal{P}_4$).

Then, by theorem 1, the algorithm $SAT_\mathcal{D}$ terminates for any formula and is complete and sound. The algorithm applied to the formula $(p \to q) \wedge p$ performs the following steps (represented as $\longmapsto_{SAT_\mathcal{D}}$):

$$\langle\langle\{\neg p, q\}, \{p\}\rangle, \langle\rangle\rangle \longmapsto_{SAT_\mathcal{D}} \langle\langle\{\{q\}\rangle, \langle p\rangle\rangle\rangle \longmapsto_{SAT_\mathcal{D}} \langle\langle\langle\rangle, \langle q, p\rangle\rangle\rangle \longmapsto_{SAT_\mathcal{D}} \langle\langle\langle\rangle, \langle q, p\rangle\rangle\rangle$$

3 Formalizing the Generic SAT–Prover in ACL2

Let us see in this section how we formalize the concepts and results of the previous section in the ACL2 logic.The ACL2 logic is a quantifier–free, first–order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp. The logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference include those for propositional calculus, equality, and instantiation, as well as the introduction of new total recursive functions by the *principle of definition* (using `defun`) and constrained functions (via `encapsulate`). The ACL2 theorem prover mechanizes that logic, being particularly well suited for obtaining automatized proofs based on simplification and induction. For a detailed description of ACL2, we refer the reader to the ACL2 book [8].

3.1 Definition of the Generic Algorithm

Before reasoning in ACL2 about the algorithm $SAT_{\mathcal{G}}$, we have to define in the ACL2 logic the functions introduced by the generic framework presented in section 2. These ACL2 functions and their intended meanings are shown in the following table:

`gen-object-p`(O)	$O \in \mathcal{O}$
`gen-repr`(F)	$i(F)$
`gen-comp-rule`(O)	$r(O)$
`gen-dist-val`(σ, O)	$\sigma \models_{\mathcal{G}} O$
`gen-model`(O)	$\sigma(O)$
`gen-measure`(O)	$\mu(O)$
`gen-select`(lst)	selects an element from a list lst

These functions are not introduced in the ACL2 logic using the principle of definition. Since they are generic, we define them by means of the `encapsulate` mechanism, which allows the user to introduce new function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms. In this case, the properties about the generic functions are the following[2]:

THEOREM: `gen-object-p-gen-repr`
`propositional-p`$(F) \rightarrow$ `gen-object-p`(`gen-repr`(F))

THEOREM: `gen-object-p-gen-comp-rule`
`gen-object-p`$(O_1) \wedge (O_2 \in$ `gen-comp-rule`$(O_1))$
$\rightarrow$ `gen-object-p`(O_2)

[2] The expressions provided to ACL2 are written in Common Lisp notation but, to improve their legibility, we present them using a infix notation.

THEOREM: e0-ordinalp-gen-measure
e0-ordinalp(gen-measure(O))

THEOREM: P1
$O_2 \in$ gen-comp-rule(O_1)
$\rightarrow$ gen-measure(O_2) $<$ gen-measure(O_1)

THEOREM: P2
propositional-p(F)
$\rightarrow$ (gen-dist-val(σ, gen-repr(F)) $\leftrightarrow$ models(σ, F))

DEFINITION:
gen-dist-val-lst(σ, O-lst) =
if endp(O-lst) **then nil**
else gen-dist-val(σ, car(O-lst))
$\vee$ gen-dist-val-lst(σ, cdr(O-lst))
fi

THEOREM: P3
gen-object-p(O) $\wedge$ (gen-comp-rule(O) $\neq$ **t**)
$\rightarrow$ (gen-dist-val-list(σ, gen-comp-rule(O))
$\leftrightarrow$ gen-dist-val(σ, O))

THEOREM: P4
gen-object-p(O) $\wedge$ (gen-comp-rule(O) = **t**)
$\rightarrow$ gen-dist-val(gen-model(O), O)

THEOREM: gen-select-member
consp(lst) $\rightarrow$ (gen-select(lst) $\in$ lst)

The first three properties state that the functions gen-repr, gen-comp-rule and gen-measure take values as expected, when acting on elements of their intended domains. The properties named P1, P2, P3 and P4 are the corresponding formalization of the properties $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ and $\mathcal{P}_4$, respectively, as defined in the hypothesis of theorem 1 [3]. The symbol $<$ denotes the "less than" relation between ordinals. Finally, note that we also introduce a function gen-select, that selects an element from any non–empty list. This function is needed in the definition of the generic SAT algorithm.

Once the functions of our generic framework have been (abstractly) defined, we define in ACL2 the function generic-sat, implementing the algorithm $SAT_{\mathcal{G}}$:

[3] The functions propositional-p and models are defined in a previous ACL2 formalization about the syntax and semantics of propositional logic; they define, respectively, the propositional formulas and models of formulas. The function gen-dist-val-lst can be seen as a generalized disjunction of the predicate gen-dist-val acting on the objects of a list.

DEFINITION:
`generic-sat-lst`(*O-lst*) =
if `endp`(*O-lst*) **then nil** (1)
else let* *O* **be** `gen-select`(*O-lst*), (2)
rest **be** `remove-one`(`gen-select`(*O-lst*), *O-lst*),
expansion **be** `gen-comp-rule`(*O*) (3)
in
if *expansion* = **t then** `list`(*O*) (4)
else `generic-sat-lst`(*expansion* @ *rest*)
fi
fi
MEASURE: `gen-measure-lst`(*O-lst*)
WELL FOUNDED RELATION: $<_{mul}$

DEFINITION:
`generic-sat`(*F*) = `generic-sat-lst`(`list`(`gen-repr`(*F*)))

where the symbol @ is the "append" operation between lists.

Note that the main function of this algorithm is given by the recursive function `generic-sat-lst`, acting on a list of objects to be expanded. When a rule of the form $\langle O, \mathbf{t} \rangle$ is applied to a selected object O, the algorithm returns a singleton list containing O (4). According to the property assumed about the function `gen-model`, this object has a distinguished valuation. Thus, returning the object will be useful to provide a model of the input formula. On the other hand, when there are no more objects to be expanded, the algorithm returns **f**, represented as the ACL2 symbol `nil` (1).

This algorithm is left unspecified in two aspects: first, no concrete computation rule is defined by the generic function `gen-comp-rule` (3); second, the object to which the expansion rule is applied, selected by the abstractly defined function `gen-select`, is not specified (2).

3.2 Termination

By the ACL2 principle of definition, new function definitions are admitted as axioms only if there exists a well–founded measure in which the arguments of each recursive call decrease, ensuring in this way that no inconsistencies are introduced by new definitions. In the case of the function `generic-sat-lst` the heuristics of the prover are not able to find a suitable termination argument, so we must explicitly provide a measure on its argument that decreases in every recursive call with respect to a well–founded relation.

In ACL2, the only primitive well–founded relation is `e0-ord-<`, the "less than" relation between ordinals up to ε_0, represented in terms of lists and natural numbers, given by the predicate `e0-ordinalp`. Nevertheless, the user can introduce a new well founded relation by providing the corresponding monotone ordinal function.

To show termination of `generic-sat-lst`, we follow the lines described in the informal proof given in section 2. The measure associated to its argument is given by a function `gen-measure-lst` that computes the list of the ordinal measures of the objects of a given list. This measure decreases with respect to the well founded relation `mul-e0-ord-<` (denoted as $<_{mul}$), defined as the multiset relation induced by `e0-ord-<`. Since `e0-ord-<` is well–founded, so is its induced multiset relation [5]. This result was formalized in the ACL2 logic in [12], where a macro named `defmul` was also developed. This macro automatically generates the definitions and theorems needed to define a well–founded multiset relation induced by a given well–founded relation.

The main termination property of `generic-sat-lst` is given by the following theorem, establishing that this measure decreases in every recursive call, and allowing the admission of the function `generic-sat-lst`.

THEOREM: generic-sat-lst-termination-property
let* *O* **be** `gen-select`(*O-lst*),
rest **be** `remove-one`(`gen-select`(*O-lst*), *O-lst*),
expansion **be** `gen-comp-rule`(*O*)
in
`consp`(*O-lst*) $\wedge$ (*expansion* $\neq$ **t**)
$\rightarrow$ `gen-measure-lst`(*expansion* @ *rest*) $<_{mul}$ `gen-measure-lst`(*O-lst*)

3.3 Soundness and Completeness

The following theorems state the formal properties of the function `generic-sat` (soundness and completeness):

THEOREM: soundness-generic-sat
`propositional-p`(F) $\wedge$ `generic-sat`(F) $\rightarrow$ `models`(`generic-mod`(F), F)

THEOREM: completeness-generic-sat
`propositional-p`(F) $\wedge$ `models`(σ, F) $\rightarrow$ `generic-sat`(F)

These two theorems formalize theorem 1 in ACL2. They are proved along the lines of the informal proof given in section 2, basically first proving by induction analogue properties about the function `generic-sat-lst`. Here the properties assumed about the generic functions showed in the subsection 3.1 plays a crucial role. The function `generic-mod`, provides a model of a satisfiable formula, its definition is the following:

DEFINITION:
`generic-mod`(F) = `gen-model`(`first`(`generic-sat`(F)))

4 Instantiating the Generic Framework

Concrete SAT–prover will be given by defining concrete counterparts of the abstractly defined functions given in subsection 3.1. With these concrete functions, one can define concrete version of the algorithm `generic-sat`.

A derived rule of inference in ACL2, *functional instantiation*, allows some kind of second–order reasoning: theorems about previously defined (or abstractly defined) functions can be instantiated with function symbols known to have the same properties. In this case, if the assumed properties about the generic functions are verified by the concrete functions, then by functional instantiation we can easily conclude termination, soundness and completeness of the concrete SAT–prover.

4.1 A Tableaux Based SAT–Prover

Along the lines of subsection 2.1, we can define in ACL2 a tableaux based instantiation of the generic framework. For that purpose we define a tableaux version of the generic functions given in subsection 3.1: `tableaux-object-p`, `tableaux-repr`, `tableaux-comp-rule`, `tableaux-dist-val`, `tableaux-model`, `tableaux-measure` and `tableaux-select`. These functions are defined as suggested in subsection 2.1. In this case, objects are lists of propositional formulas, representing branches in a tableau.

For example, the definition of the computation rule is the following:

DEFINITION:
`tableaux-comp-rule`(θ) =
if `closed-tableau`(θ) **then nil** $\mathcal{R}\tau_1$
else let F **be** `one-formula`(θ)
in
if `doubly-neg-p`(F)
then `list`(`add`(`neg-neg-component`(F), `remove-one`(F, θ))) $\mathcal{R}\tau_2$
elseif `alfa-formula-p`(F)
then `list`(`add`(`component-1`(F),
`add`(`component-2`(F), `remove-one`(F, θ)))) $\mathcal{R}\tau_3$
elseif `beta-formula-p`(F)
then `list`(`add`(`component-1`(F), `remove-one`(F, θ)),
`add`(`component-2`(F), `remove-one`(F, θ))) $\mathcal{R}\tau_4$
else t $\mathcal{R}\tau_5$
fi
fi

Here the function `closed-tableau` checks if a branch has complementary formulas. In this case, the branch is expanded to the empty list. Otherwise, a formula is selected using a function `one-formula`, and the branch is expanded according to the type of the formula, as described by $\mathcal{R}_\tau$.

Note that this computation rule implements a strategy for applying the tableaux expansion rules in a preference order, given by a function `one-formula`. Any other strategy could have been defined, provided that the properties assumed about the generic functions could be proved for the concrete counterparts. In this case, these properties are proved easily, except for `P3` and `P4`, which are somewhat more elaborated.

Once the assumed properties in the generic framework have been proved for the tableaux case, we can instantiate the generic SAT–prover algorithm, and prove analogue theorems of termination, soundness and completeness, but now using functional instantiation. The same procedure would have to be done for every concrete instantiation of the generic framework, so it makes sense to use a tool to mechanize this process to some extent.

In [9], we describe a user tool we developed to instantiate generic ACL2 theories. This tool turns out to be a valuable help in this context, where we have developed a generic theory about SAT–provers and we want to instantiate the theory to obtain concrete, formally verified and executable SAT–provers.

We defined a macro named `make-generic-theory`, which receives as argument a list of ACL2 events (definitions and theorems) that can be instantiated. When an ACL2 book[4] developing a generic theory is created, we include a call to this macro in its last line, for example, in the book that formalizes the generic framework for SAT–provers (as described in the previous section), we include the following last call:

```
(make-generic-theory *generic-sat*)
```

Here `*generic-sat*` is a constant containing the events corresponding to the generic definitions and theorems that can be instantiated by other ACL2 books. For example, the definition of `generic-sat` and the theorems establishing its properties. When this macro call is executed, it defines a new macro that receiving as input a functional substitution, generates the corresponding functional instantiation of the instantiable events.

For example, once defined the functions implementing the tableaux counterparts of the generic functions, when we include the book with the generic SAT–prover formalization, a macro `definstance-*generic-sat*` is automatically defined, and we can use this macro to automatically generate instantiated events for the tableux based SAT–prover, as follows:

```
(definstance-*generic-sat*
  ((gen-object-p       tableaux-object-p)
   (gen-repr           tableaux-repr)
   (gen-dist-val       tableaux-dist-val)
   (gen-dist-val-list  tableaux-dist-val-list)
   (gen-comp-rule      tableaux-comp-rule)
   (gen-select         tableaux-select)
   (gen-measure        tableaux-measure)
   (gen-model          tableaux-model))
   "-tableaux")
```

Note that this macro receives as input a functional substitution, associating every function of the generic framework with its tableaux counterpart. It also

[4] A collection of ACL2 definitions and proved theorems is usually stored in a certified file of *events* (a *book* in the ACL2 terminology), that can be included in other books.

receives a string, used to name the new events generated, by appending it to the name of the original event.

The result of this macro call is the *automatic* generation of the events needed to define and verify in ACL2 a tableaux based propositional SAT–prover. As a consequence, the definition of a function named `generic-sat-tableaux` is generated in an analogue way to `generic-sat` (using the tableaux auxiliary functions). And also the following theorems, establishing the soundness and completeness of `generic-sat-tableaux` are automatically generated and proved:

THEOREM: soundness-generic-sat-tableaux
`propositional-p`(F) $\wedge$ `generic-sat-tableaux`(F)
$\rightarrow$ `models`(`generic-mod-tableaux`(F), F)

THEOREM: completeness-generic-sat-tableaux
`propositional-p`(F) $\wedge$ `models`(σ, F)
$\rightarrow$ `generic-sat-tableaux`(F)

Note that, once proved that the tableaux counterparts of the generic functions verify the properties showed in subsection 3.1, no additional proof effort is needed to define and verify the tableaux–based SAT–prover.

4.2 Sequent and Davis–Putnam Based SAT–Provers

We can follow an analogous procedure to define and verify, sequent and Davis–Putnam instantiations of the generic SAT–prover. This is done by a macro call similar to that used in the tableaux case.

As with tableaux, the functional substitution used in the macro call relates the generic functions with their concrete counterparts. Of course, these concrete functions have to be previously defined, their properties proved and the book with the generic development included. These functions are defined as suggested in subsections 2.2, for the sequent based SAT–prover, and 2.3, for the Davis–Putnam based SAT–prover.

4.3 Execution Examples

The functions implementing the previous SAT–provers are executable in any compliant Common Lisp (with the appropriated files loaded). In the following table we present the results of applying the tableaux and sequents procedures to prove the satisfiability of a propositional version of the N-queens problem[5]. We also apply the Davis–Putnam procedure to the same problem. Note that the Davis–Putnam procedure works with propositional clauses and a previous translation of propositional formulas into clauses is needed in this case (we do not include the translation times).

[5] All results are in seconds of user CPU on a double 800MHz Pentium III.

N	Tableaux	Sequents	Davis-Putnam
2	0.010	0.000	0.000
3	0.060	0.020	0.010
4	0.530	0.180	0.040
5	2.370	0.820	0.140
6	212.070	72.600	0.250
7	750.540	255.640	0.570

The complete files with definitions and theorems about the generic framework, the instances and the examples, are available on the Web in
`http://www.cs.us.es/~fmartin/acl2-gen-sat/`

5 Conclusions and Further Work

We have presented an application of the ACL2 theorem prover to reason about SAT decision procedures. First, we considered a generic SAT–prover, having the essential properties of every transformation based SAT–prover. Second, we reasoned about the generic algorithm, establishing its main properties. And third, we obtained verified and executable SAT–provers using functional instantiation. This last process can be done in an automatized way.

The main effort in the formalization of the generic SAT–prover has been done in the termination proof of the generic algorithm. This proof is based on a previous work about multiset relations [12]. With respect to the instantiation procedure, the main effort has been done in the proof of the concrete version of properties `P3` and `P4` and the development of the instantiation tool. A detailed presentation of this tool can be found in [9]. The following table summarizes the number of definitions, theorems and hints needed to formalize and prove each section:

Section	Definitions	Theorems	Hints
Generic algorithm	15	37	13
Tableaux based SAT–prover	23	53	13
Sequent based SAT–prover	21	37	9
Davis–Putnam SAT–prover	23	47	9

There is some related work in mechanical verification of SAT–provers. A classical example is Boyer and Moore's propositional tautology checker [3], presented as an `IF-THEN-ELSE` normalization procedure and verified using Nqthm (the predecessor of ACL2). This example has been formalized in other systems as well. A more recent work is done by Caldwell [4] using Nuprl and program extraction to obtain a mechanically verified sequent proof system for propositional logic. See this reference for an additional account of related works.

The methodology we have followed turns out to be suitable for mechanical verification. Reasoning first about the generic algorithm allows us to concentrate on the essential aspects of the process, making verification tasks easier. Functional instantiation allows us to verify concrete instances of the algorithm,

without repeating the main proof effort and allowing some kind of mechanization of the process.

We have used this methodology to develop resolution based SAT–provers: we have defined a generic resolution procedure and we have proved its termination, soundness and completeness properties. The completeness proof is based on the developed by Bezem in [2]. This work can be found in [10].

An additional step in this methodology could be refinement. We could define more efficient functions and obtain their properties by proving equivalence theorems with the less efficient ones. In this line of work we have implemented a DPLL (Davis Putnam Logemann Loveland) procedure in ACL2, based on [15], which turns out to be much more efficient than the one presented here (It solves the 7-queens problem in 0.010 seconds and the 16-queens problem in 0.750 seconds). Using this technique of refinement, we could verify this more efficient version of the Davis–Putnam procedure

Another line of work is to apply the generic framework to other SAT methods (KE, TAS-D [1], etc). Finally, we also plan to use the same methodology to develop generic framework for non–classical and first–order logics. In the last case, we think that the development of a generic framework could be easy, combining the results presented here with a verified unification algorithm, such as the algorithm presented in [13] and [11].

References

1. G. Aguilera, I.P. de Guzman, M. Ojeda–Aciego and A. Valverde. *Reductions for non-clausal theorem proving.* Theoretical Computer Science 266, pages 81–112. Elsevier, 2001.
2. M. Bezem. *Completeness of resolution revisited.* Theoretical Computer Science 74, no. 2, pages 227–237, 1990.
3. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
4. J. Caldwell. *Decidability Extracted: Synthesizing "Correct-by-Construction" Decision Procedures from Constuctive Proofs.* PhD thesis, Cornell University, 1998
5. N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. In *Proceedings of the Sixth International Colloquium on Automata, Languages and Programming*, LNCS 71, pages 188–202. Springer–Verlag, 1979.
6. M.C. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer–Verlag, New York, 1990.
7. J.H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving.* Harper and Row Publishers, 1986.
8. M. Kaufmann, P. Manolios, and J S. Moore. *Computer–Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
9. F.J. Martin–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory, 2002.
10. F.J. Martin–Mateos. Teoria computacional (en ACL2) sobre calculos proposicionales. PhD thesis, University of Seville, 2002.
11. J.L. Ruiz–Reina. Una teoria computacional acerca de la logica ecuacional. PhD thesis, University of Seville, 2001.

12. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martin. Multiset Relations: a Tool for Proving Termination. In *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Departament, University of Texas, 2000.
13. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martin. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover. In *Proceedings AGP'99, Joint Conference on Declarative Programming*, L'Aquila (Italia), 1999.
14. R.M. Smullyan. *First–Order Logic*. Springer–Verlag: Heidelberg, Germany, 1968.
15. H. Zhang and M.E. Stickel. Implementing the Davis–Putnam method *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.

A Proof System for Information Flow Security*

Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari di Venezia
{bossi,focardi,piazza,srossi}@dsi.unive.it

Abstract. *Persistent_BNDC* (*P_BNDC*, for short) is an information-flow security property for processes in dynamic contexts, i.e., contexts that can be reconfigured at runtime. Intuitively, *P_BNDC* requires that *high level* interactions never interfere with the *low level* behavior of the system, in every possible state. *P_BNDC* is verified by checking whether the system interacting with a high level component is bisimilar or not to the system in isolation. In this work we contribute to the verification of information-flow security in two respects: (*i*) we give an *unwinding* condition that allows us to express *P_BNDC* in terms of a local property on high level actions and (*ii*) we exploit this local property in order to define a proof system which provides a very efficient technique for the development and the verification of *P_BNDC* processes.

1 Introduction

Systems are becoming more and more complex, and the security community has to face this by taking into account new threats and potentially dangerous situations. A significant example is the introduction of *process mobility* among different architectures and systems, where an application running in a "secure way" inside one environment could enter an "insecure state" while moving to a different environment. In this setting, security properties should correctly deal with such a dynamic nature of executions.

A number of formal definitions of security properties (see, for instance, [1,8,10,15,18,19,23,26,27,28]) has been proposed in the literature. *Persistent_BNDC* (*P_BNDC*, for short), proposed in [11], is a security property which is suitable to analyze processes in completely dynamic hostile environments, i.e., environments which can be dynamically reconfigured at run-time, changing in unpredictable ways. The notion of *P_BNDC* is based on the idea of Non-Interference [12,25,28] (formalized as *BNDC* [10]) and requires that every state which is reachable by the system still satisfies a basic Non-Interference property. If this holds, one is assured that even if the environment changes during the execution no malicious attacker will be able to compromise the system, as every possible reachable state is guaranteed to be secure. In [11] it has been proved that *P_BNDC* may be verified by checking whether the system interacting with a

* This work has been partially supported by the MURST project "Modelli formali per la sicurezza" and the EU Contract IST-2001-32617 "Models and Types for Security in Mobile Distributed Systems" (MyThS).

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 199–218, 2003.

high level component is *behaviorally equivalent* or not to the system in isolation, where behavioral equivalence is defined in terms of a suitable notion of weak bisimulation[1]. Moreover, in [3] it has been shown that *P_BNDC* may also be verified by checking whether the system is weakly bisimilar to a rectification of the system itself, which makes it *P_BNDC*. Both of these techniques can be fully automatized if the *labelled transition system*, i.e., the automata representing the operational behavior of the considered system, is composed of a finite number of states. In particular, there exist efficient algorithms for checking bisimulation equivalences (see, e.g., [4,14,22,7]) which are polynomial with respect to the number of states and transitions of the underlying transition system. However this kind of *behavioral verification* often suffers of the so-called state-explosion problem, i.e., the number of states increases exponentially with respect to the degree of parallelism inside the considered system. The reason is that every interleaving among parallel processes needs to be represented.

In this work we contribute to the verification of information-flow security in two respects: (*i*) we give an *unwinding* condition that allows us to express *P_BNDC* in terms of a local property of high level actions and (*ii*) we exploit this local property in order to define a proof system which provides a very efficient technique for the development and verification of *P_BNDC* processes.

The unwinding condition, similar to other already proposed in different settings (see, e.g., [16,18,20,24]), requires that every high level event is "simulable" by a sequence of internal moves, i.e, that every time a high level event is performed moving the system to a state E', a state E'' is also reachable (through internal computation) which is equivalent to E' from a low level point of view, written $E' \setminus H \approx E'' \setminus H$. Intuitively, if this holds no high level event h should be observable by a low level user, as there always exists a low-level equivalent state that the system may reach without performing h. We prove that this local property is a necessary and sufficient condition for *P_BNDC*.

As noticed in [16], unwinding conditions are useful for giving efficient proof techniques. Indeed, we use our local characterization to define a proof system which allows us to *statically prove* that a process is *P_BNDC*, i.e., by just inspecting its syntax. State-explosion is avoided by exploiting the compositionality of *P_BNDC* with respect to the parallel operator which is the source of the exponential growing of the number of states in a system. Moreover, the system offers a mean to build processes which are *P_BNDC* by construction in an incremental way. Our proof system extends the one given in [17] for finite processes, i.e., processes that may only perform finite sequences of actions. In particular, we are able to deal also with recursive processes which may perform unbounded sequences of actions. To illustrate the effectiveness of the new technique, we apply the proof system to the small, but non-trivial, example of an access monitor also considered in [10].

[1] In [10], it is shown that bisimulation-based properties are able to detect potential flows due to deadlocks caused by high level activity. Such flows are not revealed by simply observing traces, i.e., execution sequences.

The paper is organized as follows. In Section 2 we present some basic notions on the *SPA* language. In Section 3 we recall the *P_BNDC* property and we give the new *unwinding* condition. In Sections 4 and 5 we introduce our new proof system and in 6 we illustrate the usefulness of it on a simple example. Finally, in Section 7 we draw some conclusions. All proofs are collected in the Appendix.

2 Basic Notions: The SPA Language

In this section we report from [10] the syntax and semantics of the *Security Process Algebra*. The *Security Process Algebra* (SPA, for short) [10] is a variation of Milner's CCS [21], where the set of visible actions is partitioned into high level actions and low level ones in order to specify multilevel systems. SPA syntax is based on the same elements as CCS that is: a set $\mathcal{L}$ of *visible* actions such that $\mathcal{L} = I \cup O$ where $I = \{a, b, \ldots\}$ is a set of *input* actions and $O = \{\bar{a}, \bar{b}, \ldots\}$ is a set of *output* actions; a special action τ which models internal computations, i.e., not visible outside the system; a complementation function $\bar{\cdot} : \mathcal{L} \to \mathcal{L}$, such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$, and $\bar{\tau} = \tau$; $Act = \mathcal{L} \cup \{\tau\}$ is the set of all *actions*. The set of visible actions is partitioned into two sets, H and L, of high and low actions such that $\overline{H} = H$ and $\overline{L} = L$. The syntax of SPA *processes* is defined by

$$E ::= \mathbf{0} \mid a.E \mid E + E \mid E|E \mid E \setminus v \mid E[f] \mid Z$$

where $a \in Act$, $v \subseteq \mathcal{L}$, $f : Act \to Act$ is such that $f(L) \subseteq L \cup \{\tau\}$, $f(H) \subseteq H \cup \{\tau\}$, $f(\bar{\alpha}) = \overline{f(\alpha)}$ and $f(\tau) = \tau$, and Z is a constant which must be associated to a definition $Z \stackrel{\text{def}}{=} E$. Constants are useful to define recursive systems.

Intuitively, $\mathbf{0}$ is the empty process that does nothing; $a.E$ is a process that can perform an action a and then behaves as E; $E_1 + E_2$ represents the nondeterministic choice between the two processes E_1 and E_2; $E_1|E_2$ is the parallel composition of E_1 and E_2, where executions are interleaved, possibly synchronized on complementary input/output actions, producing an internal action τ; $E \setminus v$ is a process E prevented from performing actions in v; $E[f]$ is the process E whose actions are renamed *via* the relabelling function f.

Given a fixed language $\mathcal{L}$ we denote by $\mathcal{E}$ the set of all SPA processes, by $\mathcal{E}_H$ the set of all high level processes, i.e., those constructed over $H \cup \{\tau\}$, and by $\mathcal{E}_L$ the set of all low level processes, i.e., those constructed over $L \cup \{\tau\}$,

The operational semantics of SPA processes is given in terms of *Labelled Transition Systems* (LTS). A LTS is a triple $(S, A, \to)$ where S is a set of states, A is a set of labels (actions), $\to \subseteq S \times A \times S$ is a set of labelled transitions. The notation $(S_1, a, S_2) \in \to$ (or equivalently $S_1 \stackrel{a}{\to} S_2$) means that the system can move from the state S_1 to the state S_2 through the action a. The operational semantics of SPA is the LTS $(\mathcal{E}, Act, \to)$, where the states are the terms of the algebra and the transition relation $\to \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is defined by structural induction as the least relation generated by the inference rules reported in Figure 1. The operational semantics for a process E is the subpart of the SPA LTS reachable from the initial state and we refer to it as $LTS(E) = (S_E, Act, \to)$.

$$\text{Prefix}\ \frac{}{a.E \xrightarrow{a} E}$$

$$\text{Sum}\ \frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\text{Parallel}\ \frac{E_1 \xrightarrow{a} E_1'}{E_1|E_2 \xrightarrow{a} E_1'|E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1|E_2 \xrightarrow{a} E_1|E_2'} \qquad \frac{E_1 \xrightarrow{a} E_1' \;\; E_2 \xrightarrow{\bar{a}} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'} \quad a \in \mathcal{L}$$

$$\text{Restriction}\ \frac{E \xrightarrow{a} E'}{E \setminus v \xrightarrow{a} E' \setminus v} \ \text{if } a \notin v$$

$$\text{Relabelling}\ \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

$$\text{Definition}\ \frac{E \xrightarrow{a} E'}{Z \xrightarrow{a} E'} \ \text{if } Z \stackrel{\text{def}}{=} E$$

Fig. 1. The operational rules for SPA

In the paper we use the following notations. If $t = a_1 \cdots a_n \in Act^*$ and $E \xrightarrow{a_1} \cdots \xrightarrow{a_n} E'$, then we say that E' is reachable from E and write $E \xrightarrow{t} E'$, or simply $E \rightsquigarrow E'$. We also write $E \stackrel{t}{\Longrightarrow} E'$ if $E(\xrightarrow{\tau})^* \xrightarrow{a_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{a_n} (\xrightarrow{\tau})^* E'$ where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions. If $t \in Act^*$, then $\hat{t} \in \mathcal{L}^*$ is the sequence gained by deleting all occurrences of τ from t. As a consequence, $E \stackrel{\hat{a}}{\Longrightarrow} E'$ stands for $E \stackrel{a}{\Longrightarrow} E'$ if $a \in \mathcal{L}$, and for $E(\xrightarrow{\tau})^* E'$ if $a = \tau$ (note that $\stackrel{\tau}{\Longrightarrow}$ requires at least one τ labelled transition while $\stackrel{\hat{\tau}}{\Longrightarrow}$ means zero or more τ labelled transitions). Moreover, we say that a process E is *closed* if it does not contain constants. Given two processes E, F we write $E \equiv F$ when E and F are syntactically equal.

The concept of *observation equivalence* between two processes is based on the idea that two systems have the same semantics if and only if they cannot be distinguished by an external observer. This is obtained by defining an equivalence relation over $\mathcal{E}$. We report here the definition of two observational equivalences: *strong bisimulation* and *weak bisimulation* [21]. Intuitively, strong bisimulation equates two processes if they mutually simulate their behavior step by step.

Definition 1 (Strong Bisimulation). *A binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* strong bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$,*

- *if $E \xrightarrow{a} E'$, then there exists F' such that $F \xrightarrow{a} F'$ and $(E', F') \in \mathcal{R}$;*
- *if $F \xrightarrow{a} F'$, then there exists E' such that $E \xrightarrow{a} E'$ and $(E', F') \in \mathcal{R}$.*

Two processes $E, F \in \mathcal{E}$ are strong bisimilar*, denoted by $E \sim F$, if there exists a strong bisimulation $\mathcal{R}$ containing the pair (E, F).*

Relation $\sim$ is the largest strong bisimulation and is an equivalence relation [21].

Weak bisimulation is similar to strong bisimulation but it does not care about internal τ actions. So, when P simulates an action of Q, it can also execute some τ actions before or after that action.

Definition 2 (Weak Bisimulation). *A binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* weak bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$,*

- *if $E \xrightarrow{a} E'$, then there exists F' such that $F \stackrel{\hat{a}}{\Longrightarrow} F'$ and $(P', F') \in \mathcal{R}$;*
- *if $F \xrightarrow{a} F'$, then there exists E' such that $E \stackrel{\hat{a}}{\Longrightarrow} E'$ and $(E', F') \in \mathcal{R}$.*

Two processes $E, F \in \mathcal{E}$ are weakly bisimilar*, denoted by $E \approx F$, if there exists a weak bisimulation $\mathcal{R}$ containing the pair (E, F).*

Relation $\approx$ is the largest weak bisimulation and is an equivalence relation. Moreover, $\sim \subseteq \approx$ [21].

We use the notation $E[Y := X]$ to denote the process obtained by replacing in the process E the constant Y with the constant X. The following lemma provides us a syntactic way to determine when two constants are strong bisimilar.

Lemma 1. *Let X, Y be two constants defined by $X \stackrel{\text{def}}{=} E$ and $Y \stackrel{\text{def}}{=} F$. If $E[Y := X] \equiv F[Y := X]$ then $E \sim F$.*

3 The *P_BNDC* Security Property

We first recall from [11] the *Persistent Bisimulation-based Non Deducibility on Compositions* (*P_BNDC*, for short) security property and its characterization in terms of weak bisimulation up to high level actions. We start by recalling the definition of *Bisimulation-based Non Deducibility on Compositions* (*BNDC*, for short) [10]. The *BNDC* security property aims at guaranteeing that no information flow from the high to the low level is possible, even in the presence of malicious processes. The main motivation is to protect a system also from internal attacks, which could be performed by the so called *Trojan Horse* programs, i.e., programs that are apparently honest but hide inside some malicious code.

Property *BNDC* is based on the idea of checking a system against all high level potential interactions, representing all possible high malicious programs. A system E is *BNDC* if for every high process Π a low user cannot distinguish E from $(E|\Pi)$, i.e., if Π cannot interfere [12] with the low level execution of E.

Definition 3 (BNDC). *Let $E \in \mathcal{E}$ be a process.*

$$E \in BNDC \;\; \textit{iff} \;\; \forall\, \Pi \in \mathcal{E}_H,\; E \setminus H \approx (E|\Pi) \setminus H.$$

In [11] it is shown that *BNDC* is not strong enough for systems in dynamic execution environments. To deal with these situations, the property *P_BNDC* is introduced. Intuitively, a system E is *P_BNDC* if it never reaches insecure states.

Definition 4 (Persistent_BNDC). *Let $E \in \mathcal{E}$ be a process.*

$$E \in P\_BNDC \;\; \textit{iff} \;\; E \rightsquigarrow E' \textit{ implies } E' \in BNDC.$$

Example 1. Consider the process $E_1 \equiv l.h.j.\mathbf{0} + l.(\tau.j.\mathbf{0} + \tau.\mathbf{0})$ where $l, j \in L$ and $h \in H$. E_1 can be proved to be $BNDC$. Indeed, the causality between h and j in the first branch of the process is "hidden" by the second branch $l.(\tau.j.\mathbf{0} + \tau.\mathbf{0})$, which may simulate all the possible interactions with a high level process. Suppose now that E_1 is moved in the middle of a computation. This might happen when it find itself in the state $h.j.\mathbf{0}$ (after the first l is executed). Now it is clear that this process is not secure, as a direct causality between h and j is present. In particular $h.j.\mathbf{0}$ is not $BNDC$ and this gives evidence that E_1 is not P_BNDC. The process may be "repaired" as follows: $E_2 \equiv l.(h.j.\mathbf{0}+\tau.j.\mathbf{0}+\tau.\mathbf{0})+l.(\tau.j.\mathbf{0}+\tau.\mathbf{0})$. It may be proved that E_2 is P_BNDC. Note that, from this example it follows that $P\_BNDC \subset BNDC$.

In [11] it has been proven that the property P_BNDC is equivalent to the security property $SBSNNI$ [9,10] which is automatically checkable over finite-state processes. However, this property still requires a universal quantification over all the possible reachable states from the initial process E. In [11] it has been shown that this can be avoided, by including the idea of "being secure in every state" inside the bisimulation equivalence notion. This is done by defining an equivalence notion which just focus on observable actions not belonging to H.

In the following we propose another characterization of P_BNDC processes which allows us to express P_BNDC in terms of a local property of high level actions. This characterization recalls the unwinding conditions proposed in other settings (e.g., [16,18,20,24]). In [16] it is shown how unwinding conditions can be used for the verification of security properties. Here we use our characterization to prove the correctness of the proof system defined in the next sections.

Theorem 1. *Let $E \in \mathcal{E}$ be a process.*

$$E \in P\_BNDC$$
$$\textit{iff}$$
$$\textit{if } E \rightsquigarrow E_i \xrightarrow{h} E_j, \textit{ then } E_i \stackrel{\hat{\tau}}{\Longrightarrow} E_k \textit{ and } E_j \setminus H \approx E_k \setminus H.$$

The class of P_BNDC processes enjoys the compositional properties below.

Lemma 2 (Compositionality). *The following properties hold:*

1. *if E is a closed process in $\mathcal{E}_L$, then $E \in P\_BNDC$;*
2. *if E is a closed process in $\mathcal{E}_H$, then $E \in P\_BNDC$;*
3. *if $E \in P\_BNDC$, then $E \setminus v \in P\_BNDC$;*
4. *if $E \in P\_BNDC$, then $E[f] \in P\_BNDC$;*
5. *if $E, F \in P\_BNDC$, then $E|F \in P\_BNDC$;*
6. *if $E_i, F_j \in P\_BNDC$, $i \in I$ and $j \in J$, then $\sum_{i\in I} a_i.E_i + \sum_{j\in J}(h_j.F_j + \tau.F_j) \in P\_BNDC$, where $a_i \in L$ and $h_j \in H$;*
7. *if $E \in P\_BNDC$ and $X \stackrel{\text{def}}{=} E$, then $X \in P\_BNDC$.*

4 Hypothetical *P_BNDC* Processes

In this section we develop a proof system which allows us to build *P_BNDC* processes in an incremental way. It is composed by a set of rules whose conclusion is in the form $E \in \mathcal{HP}[A]$, where A is a set of constants. The intended meaning of the judgment is that E is a *P_BNDC* process provided that all the constants in A are *P_BNDC*. The set A plays the role of a set of assumptions: if it is empty then E is *P_BNDC* otherwise we are still working on our construction under open hypothesis.

Definition 5 ($\mathcal{HP}[A]$). *Let A be a set of constants and E be a SPA process where some of the constants in A may occur. We say that E is P_BNDC* under the hypothesis in *A, denoted by $E \in \mathcal{HP}[A]$, if $E \in$ P_BNDC provided that all the constants in A are P_BNDC.*

Example 2. Let $a, b \in L$ and let $E \equiv a.X + b.Y$. It holds that $E \in \mathcal{HP}[\{X, Y\}]$, since if X and Y are *P_BNDC*, then so is $a.X + b.Y$.

The rules in our proof system are suggested by the compositional properties of *P_BNDC* (see Lemma 2).

Definition 6 (Core). Core *is the proof system containing the following rules.*

$$\frac{}{P \in \mathcal{HP}[\emptyset]} \quad P \in \mathcal{E}_L,\ P \text{ is closed} \quad (Low)$$

$$\frac{}{P \in \mathcal{HP}[\emptyset]} \quad P \in \mathcal{E}_H,\ P \text{ is closed} \quad (High)$$

$$\frac{}{X \in \mathcal{HP}[\{X\}]} \quad X \text{ is a constant} \quad (Const)$$

$$\frac{E \in \mathcal{HP}[A]}{E \setminus v \in \mathcal{HP}[A]} \quad (Rest)$$

$$\frac{E \in \mathcal{HP}[A]}{E[f] \in \mathcal{HP}[A]} \quad (Label)$$

$$\frac{E \in \mathcal{HP}[A] \quad F \in \mathcal{HP}[B]}{E|F \in \mathcal{HP}[A \cup B]} \quad (Par)$$

$$\frac{E_i \in \mathcal{HP}[A_i] \quad F_j \in \mathcal{HP}[B_j]}{\sum_{i \in I} a_i.E_i + \sum_{j \in J}(h_j.F_j + \tau.F_j) \in \mathcal{HP}[\cup_{i \in I} A_i \cup \cup_{j \in J} B_j]} \quad a_i \in L \cup \{\tau\}, h_j \in H \quad (Choice)$$

$$\frac{E \in \mathcal{HP}[A]}{X \in \mathcal{HP}[A]} \quad X \stackrel{\text{def}}{=} E \quad (Def)$$

Theorem 2 (Correctness). *The system Core is correct, i.e., if there exists a proof in Core which ends with $E \in \mathcal{HP}[A]$, then E is* P_BNDC *provided that all the constants in A are* P_BNDC.

Corollary 1. *If there exists a proof of $E \in \mathcal{HP}[\emptyset]$ in Core, then E is P_BNDC.*

Notice that the system Core is not complete. One reason is that the rule (*Choice*) treats only some specific situations suggested by our characterization of Theorem 1 which can be detrmined by simple syntactic tests: for instance, the case that $E \xrightarrow{\tau} F_j$ holds whenever $E \xrightarrow{h} F_j$ holds. We could strengthen the rule by adding more complex tests based on bisimulation, but our purpose is to have a proof system whose rules are completely syntactic. Note that this is not so restrictive in the synthesis of *P_BNDC* processes, while in the case of verification it is not difficult to perform ad hoc modifications of rule (*Choice*). A second source of incompleteness comes from the lack of rules for systems of definitions which are necessary to define recursive processes. We will treat this case in the next section.

In order to derive that a process is *P_BNDC* by using Core we have to use processes for which we are able to prove that they are in $\mathcal{HP}[A]$ and then provide *P_BNDC* definitions for the constants in A.

Example 3. Let $a, b \in L$ and $h \in H$. The following derivation in Core

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b.\mathbf{0} \in \mathcal{HP}[\emptyset]}\,(Low)}{h.b.\mathbf{0} + \tau.b.\mathbf{0} \in \mathcal{HP}[\emptyset]}\,(Choice)}{a.(h.b.\mathbf{0} + \tau.b.\mathbf{0}) \in \mathcal{HP}[\emptyset]}\,(Choice) \qquad \dfrac{}{a.\mathbf{0} \in \mathcal{HP}[\emptyset]}\,(Low)}{a.(h.b.\mathbf{0} + \tau.b.\mathbf{0}) | a.\mathbf{0} \in \mathcal{HP}[\emptyset]}\,(Par)$$

proves that $a.(h.b.\mathbf{0} + \tau.b.\mathbf{0})|a.\mathbf{0}$ is *P_BNDC*. While the derivation below

$$\dfrac{\dfrac{\dfrac{}{X \in \mathcal{HP}[\{X\}]}\,(Var)}{a.X \in \mathcal{HP}[\{X\}]}\,(Choice) \qquad \dfrac{}{b.\mathbf{0} \in \mathcal{HP}[\emptyset]}\,(Low)}{a.X | b.\mathbf{0} \in \mathcal{HP}[\{X\}]}\,(Par)$$

proves that $E \equiv a.X|b.\mathbf{0}$ is in $\mathcal{HP}[\{X\}]$, which means that whenever we provide a proof of the fact that X is *P_BNDC* we obtain that E is *P_BNDC*.

In Core there is no way to eliminate the hypothesis in the recursive definitions. If X is a constant which has a definition $X \stackrel{\text{def}}{=} E$, and X occurs in E, then we are only able to prove that $X \in \mathcal{HP}[X]$, i.e., X is *P_BNDC* if X is *P_BNDC*. We will provide a more powerful system in the next section.

5 Systems of Definitions

It is possible to associate to a constant X a definition $X \stackrel{\text{def}}{=} E$ where E may possibly contain X as well as other constants. When we have a set of definitions

$$\{Z_k \stackrel{\text{def}}{=} E_k \mid k \in K\}$$

which mutually depend on each other we call this set *system of definitions*. We consider only systems of definitions in which there is at most one definition for each constant occurring in the system. A system of definitions is *weakly guarded* if all the constants Z_k, $k \in K$, occur only within some subexpression of the form $a.F$. As an example, $Z = Z$ is not weakly guarded. In this paper we restrict to this class of systems of definitions since a weakly guarded system of definitions uniquely defines, up to strong bisimulation, a process (see [21]). Given a system of definitions $S = \{Z_k \stackrel{\text{def}}{=} E_k\}_{k \in K}$ we denote by $Const(S)$ the set $\{Z_k \mid k \in K\}$.

We have to pay attention to the transformations we apply to a system of definitions, in order to avoid indesiderable effects. For instance, if we substitute a subexpression of E_k with a weakly bisimilar one we may not obtain a weakly bisimilar constant. Consider the system $\{X \stackrel{\text{def}}{=} a.X + \tau.Y; Y \stackrel{\text{def}}{=} b.Y + c.Y\}$ and replace $\tau.Y$ with Y obtaining the transformed system $\{X \stackrel{\text{def}}{=} a.X + Y; Y \stackrel{\text{def}}{=} b.Y + c.Y\}$. In the first system X can reach, by a τ move, a state which does not allows a moves. This cannot be simulated (even weakly) by the constant X in the second system. Hence the constants defined by the two systems are not bisimilar. Nevertheless, there are transformations which preserve weak bisimulation.

Lemma 3. *Let $X \stackrel{\text{def}}{=} \sum_{i \in I} a_i.E_i$ be a definition and $F \approx E_j$, for some $j \in I$. Let $Y \stackrel{\text{def}}{=} \sum_{i \in I} a_i.E'_i$ be a new definition where $E'_i \equiv E_i$ for all $i \in I, i \neq j$ and $E'_i \equiv F$ for $i = j$. Then $X \approx Y$.*

Next we introduce a transformation on processes which is at the basis of the syntactic conditions in the rule we are going to define on systems of definitions. In practice given a process E our transformation maps E into $E_{\backslash v}$, which is a sort of canonical form of $E \setminus v$, i.e., a process strong bisimilar to $E \setminus v$.

Definition 7 ($E_{\backslash v}$). *Let $v \subseteq \mathcal{L}$ and $E \in \mathcal{E}$. We define the process $E_{\backslash v}$ by induction on the structure of E.*

- $E \equiv X$: $E_{\backslash v} \equiv X \setminus v$;
- $E \equiv a.E'$: *If* $a \in v$ *then* $E_{\backslash v} \equiv \mathbf{0}$ *else* $E_{\backslash v} \equiv a.E'_{\backslash v}$;
- $E \equiv E' + E''$: *If* $E'_{\backslash v} \equiv \mathbf{0}$ *then* $E_{\backslash v} \equiv E''_{\backslash v}$ *else if* $E''_{\backslash v} \equiv \mathbf{0}$ *then* $E_{\backslash v} \equiv E'_{\backslash v}$ *else* $E_{\backslash v} \equiv E'_{\backslash v} + E''_{\backslash v}$;
- $E \equiv E'|E''$: $E_{\backslash v} \equiv (E'|E'') \setminus v$;
- $E \equiv E' \setminus w$: $E_{\backslash v} \equiv (E'_{\backslash v}) \setminus w$;
- $E \equiv E'[f]$: $E_{\backslash v} \equiv (E'[f]) \setminus v$.

Lemma 4. *Let $v \subseteq \mathcal{L}$ and $E \in \mathcal{E}$. It holds that $E_{\backslash v} \sim E \setminus v$.*

Example 4. Consider the expression $E \equiv a.X + h.b.Y + \tau.Y$. Let v be such that $a, b \notin v$ and $h \in v$. We obtain $E_{\backslash v} \equiv a.(X \setminus v) + \tau.(Y \setminus v)$.

Example 5. Consider the system

$$\begin{cases} X \stackrel{\text{def}}{=} h.(h.X + \tau.X) + a.Y \\ Y \stackrel{\text{def}}{=} h.Y + \tau.Y \end{cases}$$

where $H = \{h\}$. The constant X reaches with a high action $E \equiv h.X + \tau.X$. Moreover $E_{\backslash H} \equiv \tau.(X \setminus H)$, which implies $E \setminus H \approx \tau.X \approx X$. Since X reaches X with zero τ moves, the high transition which leads to X cannot cause problems, hence we would like to prove that this system defines *P_BNDC* processes.

The following lemma allows us to syntactically determine when two constants are such that $X \setminus H \approx Y \setminus H$. In this case, if X reaches Y with a high transition, we do not have security problems since X reaches with zero τ transition X.

Lemma 5. *Let X, Y be two constants defined by $X \stackrel{\text{def}}{=} E$ and $Y \stackrel{\text{def}}{=} F$. If $(E_{\backslash v})[Y := X] \equiv (F_{\backslash v})[Y := X]$ then $X \backslash v \sim Y \backslash v$.*

The novel characterization of *P_BNDC* stated in Theorem 1 together with Lemma 5 indicate to us some cases in which a high level transition out-coming from a variable X does not compromise the security of the system. This cases are captured by the notion of $safe(X, S)$ introduced in the definition below. The intuitive meaning of the set $safe(X, S)$ is that if $F \in safe(X, S)$, then F can be safely reached by X with a high transition.

Definition 8 ($safe(Z_k, S)$). *Let $S = \{Z_k \stackrel{\text{def}}{=} E_k\}_{k \in K}$ be a system of definitions. For each $k \in K$ we define the set $safe(Z_k, S) = \cup_{i=1}^{4} safe_i(Z_k, S)$ where*

$$\begin{aligned} safe_1(Z_k, S) &= \{F \mid Z_k \stackrel{\hat{\tau}}{\Longrightarrow} F\} \\ safe_2(Z_k, S) &= \{F \mid F_{\backslash H} \equiv Z_{k \backslash H}\} \\ safe_3(Z_k, S) &= \{F \mid F_{\backslash H} \equiv \tau.Z_{k \backslash H}\} \\ safe_4(Z_k, S) &= \{Z_j \mid E_{j \backslash H}[Z_k := Z_j] \equiv E_{k \backslash H}[Z_k := Z_j]\}. \end{aligned}$$

Example 6. Let $a, b \in L$ and $h \in H$. Consider the system S:

$$\begin{cases} X = h.Y + \tau.Z \\ Y = a.Y \\ Z = \tau.Y \\ W = a.(h.b.X + \tau.b.X) + b.Y + h.W \\ V = a.V + h.Y \end{cases}$$

We have that $Y \in safe_1(X, S)$, $W \in safe_2(W, S)$ (and also $W \in safe_1(W, S)$), and $Y \in safe_4(V, S)$.

Consider now the system S' of Example 5, in this case we have that $(h.X + \tau.X) \in safe_3(X, S')$.

The following lemma is useful to prove the main result of this section, i.e., to characterize *syntactically safe* systems.

Lemma 6. *Let $E \in \mathcal{HP}[\emptyset]$ be derived in Core. If $E \rightsquigarrow E' \xrightarrow{h} E''$, then $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''$ and $E'' \setminus H \approx E''' \setminus H$.*

Let $E \in \mathcal{HP}[A]$ be derived in Core without applying the rule (Par). If $E \rightsquigarrow E' \xrightarrow{h} E''$ without using the definitions of the constants in A, then $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''$ and $E'' \setminus H \approx E''' \setminus H$.

Definition 9 (Safe system). *Let $S = \{Z_k \stackrel{\text{def}}{=} E_k\}_{k \in K}$ be a system of definitions of the form*

$$E_k \equiv \sum_{i_k \in I_k} a_{i_k}.E_{i_k} + \sum_{j_k \in J_k} h_{j_k}.F_{j_k}.$$

The system S is said to be safe *if and only if for each $j_k \in J_k$ it holds that $F_{j_k} \in safe(Z_k, S)$ and for each G in $SubEx(S) = \cup_{k \in K}(\cup_{i_k \in I_k}\{E_{i_k}\} \cup \cup_{j_k \in J_k}\{F_{j_k}\})$ one of the following properties holds:*

- *Core proves $G \in \mathcal{HP}[\emptyset]$, or*
- *Core proves $G \in \mathcal{HP}[A_G]$ without applying the rule (Par), for some set A_G.*

We call the set $A = \cup_{G \in SubEx(S)} A_G$ safety set *of S (notation: $Safety(S)$).*

Example 7. The system S

$$\begin{cases} X \stackrel{\text{def}}{=} a.X + \tau.(\tau.Y + a.Y) + h.Y \\ Y \stackrel{\text{def}}{=} h.Y + \tau.Z + a.(\tau.Z + h.Z) \end{cases}$$

is safe and its safety set is $\{X, Y, Z\}$. In fact:

- $X \xrightarrow{h} Y$ and $X \stackrel{\hat{\tau}}{\Longrightarrow} Y$, hence $Y \in safe(X, S)$;
- $Y \xrightarrow{h} Y$ and $Y \in safe(Y, S)$;
- X and $(\tau.Y + a.Y)$ and Y can be derived to be $\mathcal{HP}[\{X\}]$ and $\mathcal{HP}[\{Y\}]$ respectively without applying (Par);
- Z and $(\tau.Z + h.Z)$ can be derived to be $\mathcal{HP}[\{Z\}]$ without applying (Par).

Theorem 3. *Let $S = \{Z_k \stackrel{\text{def}}{=} E_k\}_{k \in K}$ be a safe system of definitions with safety set A. Then for all $k \in K$ the constant Z_k is in $\mathcal{HP}[A \setminus \{Z_{k'} \mid k' \in K\}]$.*

Example 8. Consider again the system of Example 7. By Theorem 3, both X and Y belongs to $\mathcal{HP}[\{Z\}]$.

Suppose now that we want to extend the system of definitions of example above by adding the definition $Z \stackrel{\text{def}}{=} \tau.X + a.Y + b.Z$. Since we already discarded the assumptions X and Y, we would like to be able to deduce $Z \in \mathcal{HP}[\emptyset]$. We can do it if we extend the notion of safe system by relaxing the request that all the proofs are performed in Core and allow them to be carried on in any correct proof system for the judgenment $E \in \mathcal{HP}[A]$ which extends Core and satisfies Lemma 6.

Definition 10 (SafeSys). *Let SafeSys be the system of rules obtained by adding to Core the following rule (Sys):*

$$\frac{G_1 \in \mathcal{HP}[A_{G_1}] \cdots G_n \in \mathcal{HP}[A_{G_n}]}{Z \in \mathcal{HP}[Safety(S) \setminus Const(S)]} \; S \text{ safe}, Z \in Const(S), \{G_1 \ldots G_n\} = SubEx(S)$$

where a system S is safe *if and only if it satisfies all the conditions of Definition 9 with Core replaced by SafeSys in the two items.*

Theorem 4. *If there exists a proof in SafeSys which ends with $E \in \mathcal{HP}[A]$, then E is P_BNDC provided that the constants in A are P_BNDC.*

Example 9. In this example we illustrate a simple derivation in the full system SafeSys. Let $a, b \in L$ and $h \in H$. Consider the systems

$$S_X = \{X \stackrel{\text{def}}{=} a.X\}$$
$$S_Y = \{Y \stackrel{\text{def}}{=} h.Y\}$$
$$S_Z = \{Z \stackrel{\text{def}}{=} h.Z + b.(X|Y)\}$$

In order to prove that Z is P_BNDC we have to use three times the rule (Sys).

$$\dfrac{\dfrac{\dfrac{}{Z \in \mathcal{HP}[\{Z\}]}\,(Const) \qquad \dfrac{\dfrac{\dfrac{}{X \in \mathcal{HP}[\{X\}]}\,(Const)}{X \in \mathcal{HP}[\emptyset]}\,S_X(Sys) \qquad \dfrac{\dfrac{}{Y \in \mathcal{HP}[\{Y\}]}\,(Const)}{Y \in \mathcal{HP}[\emptyset]}\,S_Y(Sys)}{X|Y \in \mathcal{HP}[\emptyset]}\,(Par)}{Z \in \mathcal{HP}[\emptyset]}\,S_Z(Sys)$$

6 Example: A Process Monitor

We consider the process *Access_Monitor* which has been widely discussed in [10]. It is defined as a process which handles read and write requests from high and low level users on two binary objects: a high level variable and a low level one. To avoid information flows from high to low, two access control rules are imposed: (*i*) *no read up*: low level users cannot read from high level object; (*ii*) *no write down*: high level users cannot write into low level object. As a consequence, low level users are allowed to write into both objects and read only from the low one; conversely, high level users can read from both objects and write only into the high one. As the objects are binary, there are only two values to read or write: 0 and 1. When an object receives a read request it returns its actual value and resets itself in the same state; when it processes a write request it moves into the corresponding state.

In [10], the authors develop different definitions for the process Monitor. The aim is finding a process which is *BNDC* and for which this property is easy to check. Here we show how it is easy to synthesize a *P_BNDC* Monitor in SafeSys.

Let $access\_read(u, x)$ and $access\_write(u, x, y)$ be the access requests of the user u ($u = 0$ low, $u = 1$ high) for the object x ($x = 0$ low, $x = 1$ high) and the value y, and $\overline{val(u, y)}$ defines the values returned to the user u, where $y \in \{0, 1, err\}$. All the actions that involve high level users, i.e., the ones with $u = 1$ are considered high level ones.

In order to develop *Access_Monitor* we associate to each object x a private monitor $Monitor(x)$ which handles the requests to the object x in a secure way. As it is shown in [10], if we are able to build two *P_BNDC* processes realizing the two private monitors, we can then easily construct (by (Par) and $(Rest)$ rules) a *P_BNDC* process realizing *Access_Monitor*.

Since each object has two possible values we have to define four processes: *M00* and *M01* defining $Monitor(0)$, *M10* and *M11* defining $Monitor(1)$. For sake of simplicity we indicate them by *Mxy*. To develop their (recursive) definitions, we first assume that all of them are *P_BNDC* and then we construct a safe system of definitions whose safety set contains exactly these assumptions.

We start by considering $Monitor(0)$ which handles the accesses to the low level object. For both of its components, there are six different possible requests, two *access_read*: $access\_read(u, 0)$, $u \in \{0, 1\}$, and four *access_write*: $access\_write(u, 0, y)$, $u, y \in \{0, 1\}$.

First we consider the requests from the low level users ($u = 0$). Since both read and write on the same level are allowed, the reaction of *M0y* are the natural ones: on a read request it returns the correct value (y) and on a write request it moves into the right state. In Core there are the derivations:

$$\frac{}{Mxy \in \mathcal{HP}[\{Mxy\}]}\ (Const) \qquad \frac{\dfrac{}{M0y \in \mathcal{HP}[\{M0y\}]}\ (Const)}{\overline{val(0, y)}.M0y \in \mathcal{HP}[\{M0y\}]}\ (Choice)$$

The requests from the high level user (high actions), need more care. Since high users cannot write down, the only possible reaction to the high requests $access\_write(1, 0, z)$, $z \in \{0, 1\}$ is a reset of the actual state. As regards the request $access\_read(1, 0)$, a problem arises since the action $(\overline{val(1, y)})$ returning the value y to the high level user is a high action and we cannot derive in Core the judgement $access\_read(1, 0).\overline{val(1, y)}.M0y \in \mathcal{HP}[\{M0y\}]$, $y \in \{0, 1\}$. Note that process $access\_read(1, 0).\overline{val(1, y)}.M0y$ is potentially dangerous as a deadlock could be caused since no high level user is accepting the output action $\overline{val(1, y)}$ (see [10] for more detail on how this could be exploited to obtain an information flow from high to low). A possible solution is suggested in [3] where a lossy channel is introduced. Intuitively, the low level object sends the right value but its answer might be lost. This is represented by process $\overline{val(1, y)}.M0y + \tau.M0y$. Note that now no deadlock may be caused by high activity as it is always possible to reach *M0y* through an internal action. Now, in Core we can derive:

$$\frac{\dfrac{}{M0y \in \mathcal{HP}[\{M0y\}]}\ (Const)}{\overline{(val(1, y)}.M0y + \tau.M0y) \in \mathcal{HP}[\{M0y\}]}\ (Choice)$$

Summing up, to define $Monitor(0)$ we can introduce the system of definitions:

$$\begin{aligned} M00 \stackrel{\text{def}}{=}\ & access\_read(0,0).\overline{val(0,0)}.M00 \\ & + access\_read(1,0).(\overline{val(1,0)}.M00 + \tau.M00) \\ & + access\_write(0,0,0).M00 \\ & + access\_write(0,0,1).M01 \\ & + access\_write(1,0,0).M00 \\ & + access\_write(1,0,1).M00 \end{aligned}$$

$$\begin{aligned} M01 \stackrel{\text{def}}{=}\ & access\_read(0,0).\overline{val(0,1)}.M01 \\ & + access\_read(1,0).(\overline{val(1,1)}.M01 + \tau.M01) \\ & + access\_write(0,0,0).M00 \\ & + access\_write(0,0,1).M01 \\ & + access\_write(1,0,0).M01 \\ & + access\_write(1,0,1).M01 \end{aligned}$$

where each $G \in SubEx(Monitor(0))$ is derivable in Core without using (Par).

In order to apply the rule (Sys) we have to prove also that the system $Monitor(0)$ is safe. To this aim, we have to prove that $safe(M0y, Monitor(0))$ contains $M0y$ and $(\overline{val(1,0)}.M0y + \tau.M0y)$. Both statements holds since $M0y \stackrel{\hat{\tau}}{\Longrightarrow} M0y$ $(safe_1)$ and $(\overline{val(1,0)}.M0y + \tau.M0y)_{\setminus H} \equiv \tau.M0y$ $(safe_3)$. Hence, we can apply the rule (Sys) to derive that both $M00$ and $M01$ are P_BNDC.

The construction of the monitor for the high level object is similar to the one used to derive the system of definitions for $Monitor(0)$. It is easy to see that each subexpression in the right sides of the following system defining $Monitor(1)$ is derivable in Core without using (Par).

$$\begin{aligned} M10 \stackrel{\text{def}}{=}\ & access\_read(1,1).(\overline{val(1,0)}.M10 + \tau.M10) \\ & + access\_read(0,1).\overline{val(0,err)}.M10 \\ & + access\_write(0,1,0).M10 \\ & + access\_write(0,1,1).M11 \\ & + access\_write(1,1,0).M10 \\ & + access\_write(1,1,1).M11 \end{aligned}$$

$$\begin{aligned} M11 \stackrel{\text{def}}{=}\ & access\_read(1,1).(\overline{val(1,1)}.M11 + \tau.M11) \\ & + access\_read(0,1).\overline{val(0,err)}.M11 \\ & + access\_write(0,1,0).M10 \\ & + access\_write(0,1,1).M11 \\ & + access\_write(1,1,0).M10 \\ & + access\_write(1,1,1).M11 \end{aligned}$$

As in the previous case we have to prove that the system is safe. To this aim we have to prove that: $safe(M1y, Monitor(1))$ contains $(\overline{val(1,0)}.M1y + \tau.M1y)$, $M1y$ and $M1z$, where $z = 1 - y$. The first two conditions can be treated exactly as in the previous case. To prove the third one we need to observe that if we

substitute *M11* by *M10* in both right sides of the two definitions and apply the $\backslash_H$ transformation we obtain in both sides the same term:

$$\begin{aligned} &access\_read(0,1).\overline{val(0,err)}.M10 \\ &+access\_write(0,1,0).M10 \\ &+access\_write(0,1,1).M10. \end{aligned}$$

Hence $M1z \in safe_4(M1y, Monitor(1))$, thus by (Sys) we can derive that both *M10* and *M11* are *P_BNDC*.

7 Related Works and Conclusion

In this paper we have proposed a new local characterization of *P_BNDC* and a proof system that allows us to efficiently construct and verify *P_BNDC* processes. We have shown the effectiveness of the new technique through the example of the Access Monitor.

It is worthwhile noticing that there are many other approaches to the verification of information flow properties. For instance, there are verification techniques for information flow security which are based on types (see, e.g., [28,25,13,5]) and control flow analysis (see, e.g., [2,6]). However, most of them are concerned with different models, e.g., trace semantics [15,16,18,19].

In this paper we follow the approach of Focardi and Gorrieri [10] and focus on bisimulation based information flow properties. To the best of our knowledge, there is only another example of a proof system for security proposed by Martinelli in [17]. However, Martinelli's system deals only with finite processes. Our proof system extends [17] to the case of recursively defined processes. We avoid the state explosion problem by exploiting the compositionality results of *P_BNDC*. Indeed, if a property is preserved when secure systems are composed, then the analysis may be performed on subsystems and, in case of success, the system as a whole can be proved to be secure (see also [8,9,19]).

References

1. M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, 1999.
2. C. Bodei, P. Degano, F. Nielson, and H. Nielson. Static analysis for the pi-calculus with applications to security. *Information and Computation*, 168(1):68–92, 2001.
3. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Transforming processes to ensure and check information flow security. In H. Kirchner and C. Ringeissen, editors, *Int. Conference on Algebraic Methodology and Software Technology (AMAST'02)*, volume 2422 of *LNCS*, pages 271–286. Springer, 2002.
4. A. Bouali and R. de Simone. Symbolic Bisimulation Minimization. In *Proc. of Computer Aided Verification*, volume 663 of *LNCS*, pages 96–108. Springer, 1992.
5. G. Boudol and I. Castellani. Non-Interference for Concurrent Programs. In *Proc. of Int. Colloquium on Automata, Languages and Programming*, volume 2076 of *LNCS*, pages 382–395. Springer, 2001.

6. C. Braghin, A. Cortesi, and R. Focardi. Control Flow Analysis of Mobile Ambients with Security Boundaries. In *Proc. of IFIPM Int. Conf. on Formal Methods for Open Object-Based Distributed Systems*, pages 197–212. Kluwer, 2002.
7. A. Dovier, C. Piazza, and A. Policriti. A Fast Bisimulation Algorithm. In *Proc. of Computer Aided Verification*, volume 2102 of *LNCS*, pages 79–90. Springer, 2001.
8. N. A. Durgin, J. C. Mitchell, and D. Pavlovic. A Compositional Logic for Protocol Correctness. In *Proc. of Computer Security Foundations Workshop*. IEEE, 2001.
9. R. Focardi and R. Gorrieri. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.
10. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer, 2001.
11. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. In *Proc. of 15th Computer Security Foundations Workshop*, pages 307–319. IEEE, 2002.
12. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 11–20. IEEE, 1982.
13. M. Hennessy and J. Riely. Information Flow vs. Resource Access in the Asynchronous Pi-Calculus. In *Proc. of Int. Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *LNCS*, pages 415–427. Springer, 2000.
14. D. Lee and M. Yannakakis. Online Minimization of Transition Systems. In *Proc. of 24th Symp. on Theory of Computing*, pages 264–274. ACM, 1992.
15. H. Mantel. Possibilistic Definitions of Security - An Assebly Kit -. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 185–199. IEEE, 2000.
16. H. Mantel. Unwinding Possibilistic Security Properties. In *Proc. of European Symp. on Research in Computer Security*, volume 2895 of *LNCS*. Springer, 2000.
17. F. Martinelli. Partial Model Checking and Theorem Proving for Ensuring Security Properties. In *Proc. Computer Security Foundations Workshop*. IEEE, 1998.
18. D. McCullough. A Hookup Theorem for Multilevel Security. *IEEE Transactions on Software Engineering*, pages 563–568, June 1990.
19. J. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proc. Symp. on Security and Privacy*, pages 79–93, 1994.
20. J. K. Millen. Unwinding Forward Correctability. In *Proc. of 7th Computer Security Foundations Workshop*, pages 2–10. IEEE, 1994.
21. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
22. R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
23. L. C. Paulson. Proving Properties of Security Protocols by Induction. In *Proc. of 10th Computer Security Foundations Workshop*, pages 70–83. IEEE, 1997.
24. J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report Technical Report CSL-92-02, SRI International, December 1992.
25. A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proc. of Computer Security Foundations Workshop*. IEEE, 2000.
26. S. Schneider. Verifying Authentication Protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9), 1998.
27. V. Shmatikov and J. C. Mitchell. Analysis of a Fair Exchange Protocol. In *Proc. of 7th Annual Symp. on Network and Distributed System Security*, pages 119–128. Internet Society, 2000.
28. G. Smith and D. M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proc. of 25th Symp. on Principles of Programming Languages*, pages 355–364. ACM, 1998.

Appendix

This Appendix contains all proofs of the results presented in the paper.

Proof of Lemma 1. The proof of this lemma is done by transition induction, i.e., by induction on the length of the derivation of $E \xrightarrow{a} E'$ using the rules in Figure 1 (see [21]). Let

$$\mathcal{S} = \{(E', F') \mid E'[Y := X] \equiv F'[Y := X], \\ \text{where } X \stackrel{\text{def}}{=} E, Y \stackrel{\text{def}}{=} F \text{ and } E[Y := X] \equiv F[Y := X]\}.$$

We prove that $\mathcal{S}$ is a strong bisimulation.

Let $(E', F') \in \mathcal{S}$ and $E' \xrightarrow{a} E''$ we have to prove that there exists F'' such that $F' \xrightarrow{a} F''$ and $(E'', F'') \in \mathcal{S}$. The proof follows by transition induction on the inference $E' \xrightarrow{a} E''$.

- $E' \equiv a.E''$. Since $a.E''[Y := X] \equiv F'[Y := X]$, also F' admits an a transition $F' \xrightarrow{a} F''$ where $E''[Y := X] \equiv F''[Y := X]$. Then, $(E'', F'') \in \mathcal{S}$.
- $E' \equiv E'_1 + E'_2$. Assume $E'_1 \xrightarrow{a} E''$. Since $(E'_1 + E'_2)[Y := X] \equiv F'[Y := X]$, there exists F'_1 and F'_2 such that $F' = F'_1 + F'_2$ and $E'_1[Y := X] \equiv F'_1[Y := X]$. Then, by inductive hypothesis, there exists F'' such that $F'_1 \xrightarrow{a} F''$ and $(E'', F'') \in \mathcal{S}$.
- $E' \equiv E'_1 | E'_2$. As in the previous case, there exist F'_1 and F'_2 such that $F' = F'_1 | F'_2$, $E'_1[Y := X] \equiv F'_1[Y := X]$ and $E'_2[Y := X] \equiv F'_2[Y := X]$. We consider the case of synchronization, the other cases are similar and simpler. Assume $a = \tau$, $E'_1 \xrightarrow{b} E''_1$, $E'_2 \xrightarrow{\bar{b}} E''_2$ and $E'' \equiv E''_1 | E''_2$,. Then, by inductive hypothesis, there exist F''_1, F''_2 such that $F'_1 \xrightarrow{b} F''_1$, $F'_2 \xrightarrow{\bar{b}} F''_2$, $(E''_1, F''_1) \in \mathcal{S}$, $(E''_2, F''_2) \in \mathcal{S}$. Hence there exists $F'' \equiv F''_1 | F''_2$, such that $F' \xrightarrow{\tau} F''$ and $(E'', F'') \in \mathcal{S}$.
- $E' \equiv E_1 \backslash v$. Similar to the previous cases.
- $E' \equiv E_1[f]$. Similar to the previous cases.
- $E' \equiv Z$ where Z is a constant. There are only two symmetric non trivial cases: $E' \equiv Z \equiv X$ and $F' \equiv Y$ or $E' \equiv Z \equiv Y$ and $F' \equiv X$. In both cases, $E \xrightarrow{a} E''$ by a shorter inference, and since $E[Y := X] \equiv F[Y := X]$, by inductive hypothesis, there exists F'' such that $F \xrightarrow{a} F''$ and $(E'', F'') \in \mathcal{S}$.

□

Proof of Theorem 1. $\Leftarrow$) Let E be a process such that for all E_1 reachable from E, if $E_1 \xrightarrow{h} E_2$ then $E_1 \stackrel{\hat{\tau}}{\Longrightarrow} E_3$ and $E_2 \setminus H \approx E_3 \setminus H$. Let

$$\mathcal{S} = \{(E_i \setminus H, (E_i | \Pi) \setminus H) \mid \Pi \in \mathcal{E}_H \text{ is a process and } E \rightsquigarrow E_i\}.$$

We prove that $\mathcal{S}$ is a weak bisimulation up to $\approx$. We have to consider the following cases:

- $(E_i | \Pi) \setminus H \xrightarrow{\tau} (E_i | \Pi_1) \setminus H$. Since $E_i \setminus H \stackrel{\hat{\tau}}{\Longrightarrow} E_i \setminus H$, by definition of $\mathcal{S}$ we have $(E_i \setminus H, (E_i | \Pi_1) \setminus H) \in \mathcal{S}$.

– $(E_i|\Pi) \setminus H \xrightarrow{l} (E_j|\Pi) \setminus H$, with $l \in L \cup \{\tau\}$. Hence $E_i \setminus H \xrightarrow{l} E_j \setminus H$ and, by definition of $\mathcal{S}$, $(E_j \setminus H, (E_j|\Pi) \setminus H) \in \mathcal{S}$.
– $(E_i|\Pi) \setminus H \xrightarrow{\tau} (E_j|\Pi_1) \setminus H$ where $E_i \xrightarrow{h} E_j$. By hypothesis $E_i \setminus H \stackrel{\hat{\tau}}{\Longrightarrow} E_k \setminus H$ and $E_j \setminus H \approx E_k \setminus H$. Hence, $E_k \setminus H \approx E_j \setminus H \; \mathcal{S} \; (E_j|\Pi_1) \setminus H)$.
– $E_i \setminus H \xrightarrow{a} E_j \setminus H$. Then, $(E_i|\Pi) \setminus H \xrightarrow{a} (E_j|\Pi) \setminus H$ and $(E_j \setminus H, (E_j|\Pi) \setminus H) \in \mathcal{S}$.

$\Rightarrow$) Let E be P_BNDC. Then, for all E_i reachable from E, $E_i \in P\_BNDC$. In particular, for all E_i reachable from E and for all $\Pi \in \mathcal{E}_H$, $E_i \setminus H \approx (E_i|\Pi) \setminus H$. Suppose that $E_i \xrightarrow{h} E_j$. Let $\Pi \equiv \bar{h}$. Then $(E_i|\Pi) \setminus H \xrightarrow{\tau} E_j \setminus H$. Since $E_i \setminus H \approx (E_i|\Pi) \setminus H$, $E_i \setminus H \stackrel{\hat{\tau}}{\Longrightarrow} E_k \setminus H$ and $E_j \setminus H \approx E_k \setminus H$. □

Proof of Lemma 2. (1) is an immediate consequence of Theorem 1.

(2) follows from Theorem 1, since if $E \rightsquigarrow E'$ and $E' \xrightarrow{h} E''$, then $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'$ and $E' \setminus H \approx \mathbf{0} \approx E'' \setminus H$.

In order to prove (3) and (4) it is sufficient to observe that if E is P_BNDC, then so are $E \setminus v$ and $E[f]$, since the first operation does not add high level transitions, while the second does not exchange low and high actions.

As far as (5) is concerned, it is known that if $E, F \in P\_BNDC$, then $E|F$ is P_BNDC (see [10]).

We now prove (6) by using Theorem 1. Let $E_i, F_j \in P\_BNDC$, with $i \in I, j \in J$. Consider $R \equiv (\sum_{i \in I} a_i.E_i + \sum_{j \in J}(h_j.F_j + \tau.F_j))$. If R reaches R' with at least one transition, then either there exists $i \in I$ such that E_i reaches R' or there exists $j \in J$ such that F_j reaches R', hence R' is P_BNDC. If R reaches E' with no transitions, then $R' \equiv R$, hence if $R' \xrightarrow{h} R''$, then there exists $j \in J$ such that $R'' \equiv F_j$, and $R' \xrightarrow{\tau} F_j$, so we have the thesis.

(7) immediately follows from the operational semantics of SPA terms. □

Proof of Theorem 2. We prove that all the rules in Core are correct.

The correctness of rules (*Low*) and (*High*) directly follows from Lemma 2.

Rule (*Const*) is trivially correct.

From Lemma 2 we have the correctness of rules (*Rest*), (*Label*), (*Par*), (*Choice*), and (*Def*) in the case in which $A = \emptyset$. The general case follows immediately by the definition of $\mathcal{HP}[A]$. □

Proof of Lemma 3. Immediate since if $X \xrightarrow{a_i} E_i$ with $i \neq j$, then $Y \xrightarrow{a_i} E_i$ and if $X \xrightarrow{a_j} E_j$ then $Y \xrightarrow{a_j} F$ with $F \approx E_j$. □

Proof of Lemma 4. By induction on the structure of E.
– $E \equiv X$ is a constant. It is immediate, since by definition $E_{\setminus v}$ is $X \setminus v$.
– $E \equiv a.E'$. By inductive hypothesis on E' we have the thesis.
– $E \equiv E' + E''$. We have $E \setminus v \sim E' \setminus v + E'' \setminus v \sim E'_{\setminus v} + E''_{\setminus v}$. If $E'_{\setminus v} \equiv \mathbf{0}$, then $E'_{\setminus v} + E''_{\setminus v} \sim E''_{\setminus v}$, hence we have the thesis. Similarly we obtain the thesis if $E''_{\setminus v} \sim \mathbf{0}$. In the third case we already have the thesis.
– $E \equiv E'|E''$. It is trivial.
– $E \equiv E' \setminus w$. We have $E \setminus v \sim E' \setminus w \setminus v \sim E' \setminus v \setminus w \sim E'_{\setminus v} \setminus w$.

– $E \equiv E'[f]$. It is trivial.

□

Proof of Lemma 5.

$$\begin{array}{ll} X\backslash v \sim E\backslash v & \text{since } X \stackrel{\text{def}}{=} E; \\ \sim E_{\backslash v} & \text{by Lemma 4;} \\ \sim F_{\backslash v} & \text{by Lemma 1;} \\ \sim F\backslash v & \text{by Lemma 4;} \\ \sim Y\backslash v & \text{since } Y \stackrel{\text{def}}{=} F; \end{array}$$

□

Proof of Lemma 6. The first part of the lemma immediately follows from Theorem 1, since if E has been proved to be $\mathcal{HP}[\emptyset]$ in Core, then it is P_BNDC.

The second part follows by induction on the length of the proof $E \in \mathcal{HP}[A]$ in Core.

If $E \equiv P$ and P is a closed process and $P \in \mathcal{E}_L$ then, since P is P_BNDC, by Theorem 1 we have the thesis.

If $E \equiv P$ and P is a closed process and $P \in \mathcal{E}_H$ then, since P is P_BNDC, by Theorem 1 we have the thesis.

If $E \equiv X$ and $X \in A$, then we immediately get the thesis, since X reaches only X and X does not perform high actions without using its definition.

If $E \equiv X$ and $X \notin A$, then $X \stackrel{\text{def}}{=} E_1$ and Core proves that $E_1 \in \mathcal{HP}[A]$, with a shorter proof. Since $X \rightsquigarrow E' \stackrel{h}{\rightarrow} E''$ if and only if $E_1 \rightsquigarrow E' \stackrel{h}{\rightarrow} E''$ by inductive hypothesis on E_1 we have the thesis.

If $E \equiv E_1 \setminus v$, then if $E_1 \setminus v \rightsquigarrow E' \setminus v \stackrel{h}{\rightarrow} E'' \setminus v$ by inductive hypothesis $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''$ with $E''\backslash H \approx E'''\backslash H$, hence $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''\backslash v$ with $E''\backslash v\backslash H \approx E'''\backslash v\backslash H$

If $E \equiv E_1[f]$, as in the previous case we obtain the thesis by inductive hypothesis.

If $E \equiv \sum_{i\in I} a_i.E_i + \sum_{j\in J}(h_j.F_j + \tau.F_j)$, then if $E' \equiv E$ we immediately get the thesis, otherwise we obtain it by inductive hypothesis. □

Proof of Theorem 3. By using Theorem 1 we have to prove that if $Z_k \rightsquigarrow P' \stackrel{h}{\rightarrow} P''$ without applying the definitions of the constants in $A \setminus \{Z_{k'} \mid k' \in K\}$, then $P' \stackrel{\hat{\tau}}{\Longrightarrow} P'''$ without applying the definitions of the constants in $A\backslash\{Z_{k'} \mid k' \in K\}$, with $P'' \setminus H \approx P''' \setminus H$.

We proceed by induction of the number of applications of the (Definition) rule in the semantic derivation of $Z_k \rightsquigarrow P'$.

If the rule has never been applied, then $P' \equiv Z_k$. If $Z_k \stackrel{h}{\rightarrow} P''$, then there exists j_k such that $P'' \equiv F_{j_k}$ and $F_{j_k} \in \mathit{safe}(Z_k, S)$. Hence four cases are possible: (1) $Z_k \stackrel{\tau}{\Longrightarrow} F_{j_k}$; (2) $F_{j_k\backslash H} \equiv Z_{k\backslash H}$; (3) $F_{j_k\backslash H} \equiv \tau.Z_{k\backslash H}$; (4) $F_{j_k} \equiv Z_j$ and $E_{j\backslash H}[Z_k := Z_j] \equiv E_{k\backslash H}[Z_k := Z_j]$. In case (1) we obtain the thesis since $F_{j_k} \setminus H \approx F_{j_k} \setminus H$ no matter which is the definition for the constants which occur in it. In case (2) we have that Z_k reaches with zero τ actions Z_k and $Z_k \setminus H \approx$

$Z_{k\setminus H} \equiv F_{j_k\setminus H} \approx F_{j_k} \setminus H$. Similarly we obtain the thesis in case (3). In case (4) we obtain the thesis since Z_k reaches with zero τ actions Z_k and from Lemma 5 $Z_k \setminus H \approx Z_j \setminus H$.

If the rule has been applied exactly once, then the rule has been applied to Z_k in the first step, i.e. $Z_k \stackrel{a_{i_k}}{\to} E_{i_k} \rightsquigarrow P' \stackrel{h}{\to} P''$ (or $Z_k \stackrel{h_{j_k}}{\to} F_{j_k} \ldots$) and in the derivation of $E_{i_k} \rightsquigarrow P' \stackrel{h}{\to} P''$ the (Definition) rule has never been applied. This means that $E_{i_k} \rightsquigarrow P' \stackrel{h}{\to} P''$ without using the definitions of the Z's. From Lemma 6 we have that $P' \stackrel{\hat{\tau}}{\Longrightarrow} P'''$ without using the definition of the Z's and $P''' \setminus H \approx P'' \setminus H$. This implies that $P' \stackrel{\hat{\tau}}{\Longrightarrow} P'''$ with $P''' \setminus H \approx P'' \setminus H$ no matter which is the definition of the Z's.

Let us assume that we have proved that for each $k \in K$ if $Z_k \rightsquigarrow P' \stackrel{h}{\to} P''$ with n applications of the (Definition) rule, then $P' \stackrel{\hat{\tau}}{\Longrightarrow} P'''$ with $P'' \setminus H \approx P''' \setminus H$. Let $Z_k \rightsquigarrow P' \stackrel{h}{\to} P''$ with $n+1$ applications of the (Definition) rule. This means that $Z_k \stackrel{a_{i_k}}{\to} E_{i_k} \rightsquigarrow Z_r \rightsquigarrow P' \stackrel{h}{\to} P''$ or $Z_k \stackrel{h_{j_k}}{\to} F_{j_k} \rightsquigarrow Z_r \rightsquigarrow P' \stackrel{h}{\to} P''$, and since the (Definition) rule has been applied once in the first step we have that in $Z_r \rightsquigarrow P' \stackrel{h}{\to} P''$ the (Definition) rule is applied at most n times. Hence by inductive hypothesis we have the thesis. □

Proof of Theorem 4. By Theorem 3 we have that if the rule (Sys) is applied once, then the proof is correct.

If the rule (Sys) is applied more than once, then we obtain the thesis since Lemma 6 holds also if G has been proved to be in $\mathcal{HP}[A]$ by applying the rule (Sys). This last can be proved by induction on the number of application of the rule (Sys). □

Forward Slicing of Multi-paradigm Declarative Programs Based on Partial Evaluation*

Germán Vidal

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Program slicing has been mainly studied in the context of imperative languages, where it has been applied to many software engineering tasks, like program understanding, maintenance, debugging, testing, code reuse, etc. This paper introduces the first forward slicing technique for multi-paradigm declarative programs. In particular, we show how program slicing can be defined in terms of online partial evaluation. Our approach clarifies the relation between both methodologies and provides a simple way to develop program slicing tools from existing partial evaluators.

1 Introduction

Essentially, program slicing [34] is a method for decomposing programs by analyzing their data and control flow. It has many applications in the field of software engineering (e.g., program understanding, maintenance, debugging, merging, testing, code reuse, etc). This concept was originally introduced by Weiser [33] in the context of imperative programs. Surprisingly, there are very few approaches to program slicing in the context of declarative programming (some notable exceptions are, e.g., [13,21,24,25,29]). Roughly speaking, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [10,19] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A complete survey on program slicing can be found, e.g., in [30].

The main purpose of partial evaluation techniques is to specialize a given program w.r.t. part of its input data—hence it is also known as *program specialization*. The partially evaluated program will be (hopefully) executed more

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana under grant CTIDIA/2002/205, and by the MCYT under grants HA2001-0059, HU2001-0019 and HI2000-0161.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 219–237, 2003.

efficiently since those computations that depend only on the known data are performed—at partial evaluation time—once and for all. Many *online* partial evaluation schemes follow a common pattern: given a program and a partial call, the partial evaluator builds a *finite* representation—generally a graph—of the possible executions of the partial call and, then, systematically extracts a *residual* program (the partially evaluated program) from this graph. This view of partial evaluation clearly shows the similarities with program slicing: both techniques should construct a finite representation of some program execution, usually with part (or none) of the input data.

In this paper, we present a forward slicing method based on (online) partial evaluation. While the construction of a graph representing some program execution is quite similar in both techniques, the extraction of the final program is rather different. Partial evaluation usually achieves its effects by compressing paths in the graph and by renaming expressions in order to remove unnecessary function symbols. Hence, partial evaluation constructs a *new*, residual program. In contrast, program slicing should preserve the structure of the original program: statements can be (totally or partially) deleted but new statements cannot be introduced. Following [12], partial evaluators can be classified into the following categories:

- *monovariant*: each function of the original program gives rise to (at most) one residual function,
- *polyvariant*: each function of the original program may give rise to one or more residual functions,
- *monogenetic*: each residual function stems from one function of the original program, and
- *polygenetic*: each residual function may stem from one or more functions of the original program.

According to this classification, forward slicing can be seen as a particular form of *monovariant* and *monogenetic* partial evaluation (in order to preserve a one-to-one relation between the functions of the original and residual programs).

Unfortunately, monovariant/monogenetic partial evaluation could be rather imprecise, thus resulting in an unnecessarily large residual program (i.e., slice). To overcome this problem, we introduce an extended operational semantics to perform partial evaluations, which helps us to preserve as much information as possible while maintaining the monovariant/monogenetic nature of the process. In order to center the discussion, we present our developments in the context of a multi-paradigm declarative language which integrates features from functional and logic programming (like, e.g., Curry [17] or Toy [22]).

The main contributions of this work are the following: (1) We define the first forward slicing technique for functional logic programs. Moreover, the application of our developments to pure (lazy) functional programs would be straightforward, since either the syntax and the underlying (online) partial evaluators (e.g., positive supercompilation [28]) share many similarities. (2) Our method is defined in terms of an existing partial evaluation scheme; therefore, it is easy to implement by adapting current partial evaluators. Furthermore, we do not need

to consider separately static/dynamic slicing, since partial evaluation naturally accepts partially instantiated calls. (3) Our approach helps us to clarify the relation between (forward) slicing and (online) partial evaluation. Also, we discuss several possibilities to perform backward slicing by extending our developments.

This paper is organized as follows. In the next section we introduce some foundations for understanding the subsequent developments. Section 3 introduces the forward slicing technique. First, we recall the narrowing-driven approach to partial evaluation (Sect. 3.1); then we introduce an algorithm for computing program dependences by partial evaluation (Sect. 3.2); finally, we present our method to extract program slices (Sect. 3.3). Section 4 discusses two possible extensions of the scheme in order to perform backward slicing. Finally, we report in Sect. 5 an implementation of our program slicing tool and conclude in Sect. 6 with a comparison to related work. More details and proofs of all technical results can be found in [32].

2 Foundations

Recent proposals for multi-paradigm declarative programming (including features from the functional, logic and concurrent paradigms) consider inductively sequential rewrite systems [6] as *source programs* and a combination of needed narrowing [7] (the counterpart of call-by-name evaluation) and residuation as operational semantics [14]. In actual implementations, e.g., the PAKCS environment [15] for Curry [17], programs may also include a number of additional features: calls to external (built-in) functions, concurrent constraints, higher-order functions, overlapping left-hand sides, guarded expressions, etc. In order to ease the compilation of programs as well as to provide a common interface for connecting different tools working on source programs, a *flat representation* for programs has recently been introduced. This representation is based on the formulation of [16] to express pattern-matching by case expressions. The complete flat representation is called FlatCurry [15,17] and is used as an intermediate language during the compilation of source programs.

In order to simplify the presentation, we will only consider the *core* of the flat representation. Extending the developments in this work to the remaining features is not difficult and, indeed, the implementation reported in Sect. 5 covers most of the additional features. The syntax of flat programs is summarized in Fig. 1, where $\overline{o_n}$ stands for the *sequence* of objects $o_1, \ldots, o_n$. We consider the following domains:

x, y, z	$\in \mathcal{X}$	(variables)	a, b, c	$\in \mathcal{C}$	(constructor symbols)
f, g, h	$\in \mathcal{F}$	(defined functions)	$e_1, e_2, \ldots$	$\in \mathcal{E}$	(expressions)
$t_1, t_2, \ldots$	$\in \mathcal{T}$	(terms)	$v_1, v_2, \ldots$	$\in \mathcal{V}$	(values)

The only difference between *terms* and *expressions* is that the latter may contain case expressions. We say that a term is *operation-rooted* (resp. *constructor-rooted*) if it has a defined function symbol (resp. a constructor symbol) at the outermost position. *Values* are terms in head normal form, i.e., variables or

$\mathcal{R} ::= D_1 \ldots D_m$	(program)	$t ::= x$	(variable)
$D ::= f(\overline{x_n}) = e$	(rule)	$\mid c(\overline{t_n})$	(constructor call)
$e ::= t$	(term)	$\mid f(\overline{t_n})$	(function call)
$\mid$ *case* x *of* $\{\overline{p_m \to e_m}\}$	(rigid case)	$p ::= c(\overline{x_n})$	(flat pattern)
$\mid$ *fcase* x *of* $\{\overline{p_m \to e_m}\}$	(flexible case)		

Fig. 1. Syntax of Flat Programs

constructor-rooted terms. A program $\mathcal{R}$ consists of a sequence of function definitions D such that each function is defined by a single rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression e composed by variables, constructors, function calls, and case expressions for pattern-matching. The general form of a case expression is[1]:

$$(f)case\ x\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_m(\overline{x_{n_m}}) \to e_m\}$$

where x is a variable, $c_1, \ldots, c_m$ are different constructors of the type of x, and $e_1, \ldots, e_m$ are expressions (possibly containing nested (f)*case*'s). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i. The difference between *case* and *fcase* only shows up when the argument, x, is a free variable (within a particular computation): *case* suspends—which corresponds to *residuation*, i.e., pure functional reduction—whereas *fcase* nondeterministically binds this variable to a pattern in a branch of the case expression—which corresponds to either narrowing [7] and driving [31]. Note that our functional logic language mainly differs from typical (lazy) functional languages in the presence of flexible case expressions.

Example 1. Consider the well-known function `app` to concatenate two lists. It can be defined in the flat representation as follows[2]:

```
app x y = case x of { [ ]      → y ;
                      (z : zs) → z : app zs y }
```

where `[ ]` denotes the empty list and `x : xs` a list with first element `x` and tail `xs`.

An automatic transformation from source (inductively sequential [6]) programs to flat programs is introduced in [16]. Translated programs always fulfill the following restrictions: case expressions in the right-hand sides of program rules appear always in the outermost positions (i.e., there is no case expression inside a function or constructor call) and all case arguments are variables, thus the syntax of Fig. 1 is general enough for our purposes. We shall assume these restrictions on flat programs in the following.

[1] We write (f)*case* for either *fcase* or *case*.

[2] Although we consider in this work a first-order representation, we use a curried notation in concrete examples (as it is common practice in functional languages).

3 Forward Slicing

This section presents a forward slicing technique based on (online) partial evaluation. In our context, any program expression may play the role of *slicing criterion*. Therefore, we do not need to distinguish between dynamic and static slicing, it only depends on the degree of instantiation of the slicing criterion. Given a program and a (possibly incomplete) call—the slicing criterion—, we return a *program slice* containing those parts of the original program which are reachable from the slicing criterion, i.e., which are needed to evaluate the slicing criterion. Functions in the slice should belong to the original program, although we accept the removal of alternatives in case expressions and the elimination of (unnecessary) function calls.

Example 2. Let us consider the following simple program:

```
foo x y z = fst (len (app x y), snd z)
len x     = case x of { [] → Z; (y:ys) → Succ (len ys) }
app x y   = case x of { [] → y; (z:zs) → z:app zs y }
fst (x,y) = x
snd (x,y) = y
```

where natural numbers are build from constructors `Z` and `Succ`. Function "`foo`" computes the length of the list resulting from the concatenation of its two first arguments. The unnatural definition of this function will be useful to illustrate the effects of program slicing. Standard functions "`len`", "`app`", "`fst`", and "`snd`" return, respectively, the length of a list, the concatenation of two lists, the first element of a tuple, and the second element of a tuple. Let us consider that the slicing criterion is the expression "`foo [] y z`". Then the computed slice should be as follows:

```
foo x y z = fst (len (app x y), ⊤)
len x     = case x of { [] → Z; (y:ys) → Succ (len ys) }
app x y   = case x of { [] → y }
fst (x,y) = x
```

Here, the second alternative of the case expression in the right-hand side of function "`app`" has been removed, since it is not needed to execute the slicing criterion. Also, the evaluation of the call to "`snd`" in the right-hand side of function "`foo`" is not needed—the outermost function, "`fst`", only demands the evaluation of the first component of the tuple—and, thus, it has been replaced by $\top$, a special symbol denoting that some subterm is missing due to the slicing process. Since function "`snd`" is no longer necessary, its definition has been completely deleted. Note that this slice could not be constructed by using a simple graph of functional dependencies (e.g., function "`snd`" depends on function "`foo`" but it does not appear in the computed slice).

Usually, slicing criteria are program calls whose execution traces we are interested in (e.g., to correct a bug or to extract a program fragment which we want to reuse in another context).

As discussed in the introduction, our developments rely on the fact that forward slicing can be regarded as a form of monovariant/monogenetic partial evaluation. This requirement is necessary in order to ensure that there is a one-to-one relation between the functions of the original and residual programs, which is crucial to produce a fragment of the original functions rather than a specialized version. Basically, the following two points should be taken into account:

- Since monovariant/monogenetic partial evaluation propagates information poorly, we need a carefully designed operational mechanism which avoids the loss of information (i.e., program dependences) as much as possible.
- The extraction of residual rules should be modified in order to ensure that the residual program is always a *fragment* of the original one.

The rest of this section introduces our program slicing technique based on an appropriate extension of a monovariant/monogenetic partial evaluator. We proceed in a stepwise manner: first, we recall an online partial evaluation scheme; then we modify the kernel of this algorithm in order to compute program dependences; finally, we present a method to construct the desired program slice from the computed program dependences.

3.1 Monovariant/Monogenetic Partial Evaluation

In this section, we recall the main algorithm of narrowing-driven partial evaluation [3,4]. Essentially, it proceeds by iteratively unfolding a set of function calls, testing the *closedness* of the unfolded expressions, and adding to the current set those calls (in the derived expressions) which are not closed. This process is repeated until all the unfolded expressions are closed, which guarantees the correctness of the transformation process [5], i.e., that the resulting set of expressions covers all the possible computations for the initial call. This iterative style of performing partial evaluation was first described by Gallagher [11] for the partial evaluation of logic programs. The computation of a closed set of expressions can be regarded as the construction of a graph with all the program points which are reachable from the initial call. Intuitively, an expression is *closed* whenever its maximal operation-rooted subterms (function calls) are instances of the already partially evaluated terms. Formally, the closedness condition is defined as follows:

Definition 1. *Let E be a finite set of expressions. We say that an expression e is closed w.r.t. E (or E-closed) iff one of the following conditions hold:*

- *e is a variable;*
- *$e = c(e_1, \ldots, e_n)$ is a constructor call and $e_1, \ldots, e_n$ are recursively E-closed;*
- *$e = (f)case\ e'\ of\ \{\overline{p_k \to e_k}\}$ is a case expression and $e', e_1, \ldots, e_k$ are recursively E-closed;*
- *e is operation-rooted, there is an expression $e' \in E$, a matching substitution σ, with $e = \sigma(e')$, and for all $x \mapsto e'' \in \sigma$, e'' is recursively E-closed.*

Input: a program $\mathcal{R}$ and a term t
Output: a residual program $\mathcal{R}'$
Initialization: $i := 0;\ E_0 := \{t\}$
Repeat
$E' := \mathit{unfold}(E_i, \mathcal{R})$;
$E_{i+1} := \mathit{abstract}(E_i, E')$;
$i := i + 1$;
Until $E_i = E_{i-1}$ (modulo renaming)
Return:
$\mathcal{R}' := \mathit{build\_residual\_program}(E_i, \mathcal{R})$

Fig. 2. Narrowing-Driven Partial Evaluation Procedure

The basic partial evaluation procedure is shown in Fig. 2. The operator *unfold* takes a program and a set of expressions, computes a *finite* set of (possibly incomplete) finite derivations, and returns the set of derived expressions. Function *abstract* is used to properly add the new expressions to the current set of (to be) partially evaluated expressions. The main loop of the algorithm can be seen as a *pre-processing* stage whose aim is to find a closed set of expressions. Note that no residual rules are actually constructed during this phase. Only when a closed set of expressions is eventually found, residual rules are built (usually, by applying one more time the unfolding operator, followed by a post-processing of renaming and some post-unfolding transformations).

Basically, a monovariant/monogenetic partial evaluation algorithm can be designed from the procedure in Fig. 2 by imposing several restrictions:

1. Expressions in the current set should be operation-rooted terms without nested function calls (i.e., of the form $f(\overline{t_n})$, where f is a defined function symbol and $t_1, \ldots, t_n$ are constructor terms). This is necessary to ensure that partial evaluation is monogenetic.
2. The unfolding operator should perform only a one-step evaluation of each call. This condition is required to guarantee that no reachable function is hidden by the unfolding process.
3. Finally, the abstraction operator should ensure that the current set of terms contains at most one term for each function symbol. In this way, we enforce the monovariant nature of the partial evaluation process.

A trivial partial evaluator fulfilling the above restrictions could proceed by *flattening* all terms containing nested function symbols and by replacing those terms rooted by the same function symbol with some appropriate generalization (e.g., their *most specific generalization*). For instance, a term of the form "`len` (`app` [ ] [ ])" would be replaced by the terms "`len y`" and "`app` [ ] [ ]". However, this naive treatment would imply a serious loss of precision, e.g., the fact that `len` is only called with the result of "`app` [ ] [ ]" (an empty list).

To avoid this loss of precision, we drop the first restriction above, i.e., we consider arbitrary operation-rooted terms (possibly) containing nested function

calls. Also, we extend the partial evaluation mechanism in order to work on states rather than on expressions. States have the form $\langle e, S\rangle$, where e is the expression to be evaluated and S is the *stack* (a list) which represents the current "evaluation context"[3]. The empty stack is denoted by $[\,]$. An important property of our flat language is that it evaluates function calls lazily: an expression containing nested function calls is evaluated by first unfolding the outermost function; inner function calls are only evaluated *on demand*, i.e., when they appear as the argument of some case expression. For instance, "`len` (`app` [ ] [ ])" is unfolded to "`case` (`app` [] []) `of` {...}"; then the evaluation of function "`len`" cannot continue until the inner call to "`app`" is reduced to a value. Unfortunately, this interleaved evaluation is problematic in our context since it would give rise to a polygenetic partial evaluation. In contrast, we should perform a *complete one-step unfolding* of each function call separately (i.e., a function unfolding followed by the reduction of all the case structures in the unfolded expression). Here, the stack becomes relevant in order to store outer function calls until a complete one-step unfolding is possible. The next section introduces an extended semantics which is appropriate to deal with states.

3.2 Computing Program Dependences

In our context, the computation of program dependences boils down to computing the set of functions which are *reachable* from the slicing criterion, i.e., the functions which are *needed* to evaluate the slicing criterion. For this purpose, we introduce the extended operational semantics of Fig. 3. Let us briefly explain the rules of this operational semantics. Rule select is used to select the appropriate branch and continue with the evaluation of this branch. When the argument of a case expression is a free variable, rule guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; thus, the resulting calculus becomes non-deterministic as well. Rule flatten is used to avoid the unfolding of those (operation-rooted) terms whose unfolding would demand the evaluation of some inner call. In this case, we delay the function unfolding and continue evaluating the demanded inner call. Auxiliary function *flat* is used to flatten these states. Here, we use subscripts in the arrows to indicate the application of some concrete rule(s). Function *flat* proceeds as follows: When the expression in the input state can be reduced by using rules select and guess to a case expression with a function call in the argument position (which is thus *demanded*), function *flat* returns a new state whose first component is the demanded call, $g(\overline{t'_m})$, and whose stack is augmented by adding a new pair $(f(\overline{t_n})[g(\overline{t'_m})/x], x)$. Here, $f(\overline{t_n})[g(\overline{t'_m})/x]$ denotes the term obtained from $f(\overline{t_n})$ by replacing the selected occurrence of the inner call, $g(\overline{t'_m})$, with a fresh variable x. This pair contains all the necessary information to reconstruct the original expression once the inner call is evaluated to a value (in rule replace). Rule fun performs a simple function unfolding when rule flatten does not apply, i.e., when

[3] Similar operational semantics which make use of a stack can be found in [2,27].

$$
\begin{array}{lll}
\textsf{(select)} & \langle\, (f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to e_k}\},\ S\,\rangle & \Longrightarrow\ \langle\, \rho(e_i),\ S\,\rangle \\
& \text{if } p_i = c(\overline{x_n}) \text{ and } \rho = \{\overline{x_n \mapsto t_n}\} \text{ for some } i \in \{1,\ldots,k\} & \\
\textsf{(guess)} & \langle\, (f)case\ x\ of\ \{\overline{p_k \to e_k}\},\ S\,\rangle & \Longrightarrow\ \langle\, \rho(e_i),\ S\,\rangle \\
& \text{if } \rho = \{x \mapsto p_i\} \text{ for all } i = 1,\ldots,k & \\
\textsf{(flatten)} & \langle\, f(\overline{t_n}),\ S\,\rangle & \Longrightarrow\ \langle\, g(\overline{t'_m}),\ S^f\,\rangle \\
& \text{if } \mathit{flat}(f(\overline{t_n}), S) = \langle g(\overline{t'_m}), S^f\rangle & \\
\textsf{(fun)} & \langle\, f(\overline{t_n}),\ S\,\rangle & \Longrightarrow\ \langle\, \rho(e),\ S\,\rangle \\
& \text{if } \mathit{flat}(f(\overline{t_n}), S) = \bot,\ f(\overline{x_n}) = e \in \mathcal{R},\ \overline{x_n} \text{ are fresh, and } \rho = \{\overline{x_n \to t_n}\} & \\
\textsf{(replace)} & \langle\, v,\ (f(\overline{t_n}), x) : S\,\rangle & \Longrightarrow\ \langle\, \rho(f(\overline{t_n})),\ S\,\rangle \\
& \text{if } v \text{ is a value and } \rho = \{x \mapsto v\} &
\end{array}
$$

$$
\begin{array}{lll}
\textsf{where}\ \mathit{flat}(\langle f(\overline{t_n}), S\rangle) & = & \textsf{if}\ \rho(e) \Longrightarrow^*_{\textsf{select/guess}} (f)case\ g(\overline{t'_m})\ of\ \{\ \ldots\ \} \\
& & \quad \textsf{then}\ \langle g(\overline{t'_m}), (f(\overline{t_n})[g(\overline{t'_m})/x], x) : S\rangle \\
& & \quad \textsf{else}\ \bot \\
& & \textsf{with}\ f(\overline{x_n}) = e \in \mathcal{R},\ \overline{x_n}\ \textsf{fresh, and}\ \rho = \{\overline{x_n \to t_n}\}
\end{array}
$$

Fig. 3. Extended Operational Semantics

function *flat* returns $\bot$. Finally, rule replace allows us to retake the evaluation of some delayed function call once the demanded inner call is reduced to a value.

The extended operational semantics behaves almost identically to the standard semantics for flat programs of [16]. There are, though, two slight differences:

- In the standard semantics, rigid case expressions with a free variable in the argument position *suspends*. In our case, rule guess proceeds with their evaluation as if they were flexible. This is motivated by the fact that we may have *incomplete* information; hence, in order to be on the safe side (and do not miss any reachable function), we should consider all the alternatives of rigid case expressions.
- The order of evaluation is changed. In our extended semantics, we delay those function unfoldings which cannot be followed by the reduction of all the case expressions in the corresponding right-hand side.

In spite of these differences, both calculi trivially produce the same results for input expressions involving no suspension. The relevance of the extended semantics stems from the fact that computations can now be split into a number of consecutive sequences of steps of the form:

$$
\Longrightarrow^*_{\textsf{flatten}} \underbrace{\Longrightarrow_{\textsf{fun}} \Longrightarrow^*_{\textsf{select/guess}}}_{seq\_1} \Longrightarrow^*_{\textsf{replace}} \Longrightarrow^*_{\textsf{flatten}} \underbrace{\Longrightarrow_{\textsf{fun}} \Longrightarrow^*_{\textsf{select/guess}}}_{seq\_2} \Longrightarrow^*_{\textsf{replace}} \cdots
$$

where each subsequence, *seq_i*, represents a complete one-step evaluation of some function call. From these sequences, a monogenetic/monovariant partial evaluation scheme can easily be defined (and, thus, our program slicing technique). Let

Input: a program $\mathcal{R}$ and an operation-rooted term t
Output: a set of states $\mathcal{S}$
Initialization: $i := 0$; $\mathcal{S}_0 := \{\langle t', S\rangle\}$, where $\langle t, [\,]\rangle \Longrightarrow^*_{\mathsf{flatten}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{flatten}}$
Repeat
$\mathcal{S}' := \mathit{unfold}(\mathcal{S}_i, \mathcal{R})$;
$\mathcal{S}_{i+1} := \mathit{abstract}(\mathcal{S}_i, \mathcal{S}')$;
$i := i + 1$;
Until $\mathcal{S}_i = \mathcal{S}_{i-1}$ (modulo renaming)
Return: $\mathcal{S} := \mathcal{S}_i$

Fig. 4. Computation of Reachable Program Points

us note that our operational semantics could also be a good candidate to develop alternative approaches for computing program dependences (e.g., to develop a dependency graph analysis based on abstract interpretation).

The algorithm of Fig. 2 is now slightly modified in order to work with states. The new algorithm (Fig. 4) does not compute a residual program but the set of states which are reachable from the initial call; note that they are equivalent to the final set of *closed* terms computed by the algorithm of Fig. 2 (except for the fact that terms are represented by states). The new algorithm starts by flattening the initial term in order to ensure that a complete one-step unfolding can be performed. We now tackle the definition of appropriate unfolding and abstraction operators. Our one-step unfolding operator is defined as follows:

$$\mathit{unfold}(\mathcal{S}) = \bigcup_{s \in \mathcal{S}} \mathit{unf}(s)$$

where

$$\mathit{unf}(\langle t, S\rangle) = \{\langle t', S\rangle \mid \langle t, S\rangle \Longrightarrow_{\mathsf{fun}} \langle t'', S\rangle \Longrightarrow^*_{\mathsf{select/guess}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{select/guess}}\}$$

This unfolding operator always performs a complete one-step unfolding of each input expression. The associated stack S remains unchanged since only rules flatten and replace can modify the current stack. Function *unf* returns a *set* of derived states because of the non-determinism of the underlying operational semantics.

Before defining our abstraction operator, we need the following auxiliary notion. States returned by the unfolding operator and, then, reduced by rules replace and flatten are called *flattened* states. Formally, let s be a state returned by operator *unfold* with $s \Longrightarrow^*_{\mathsf{replace/flatten}} s' \not\Longrightarrow$. Then s' is called a *flattened* state. Flattened states have a particular form, as stated by the following lemma:

Lemma 1. *Let s be a flattened state. Then s has the form $\langle v, [\,]\rangle$, where v is a value, or $\langle f(\overline{t_n}), S\rangle$, where $f(\overline{t_n})$ is an operation-rooted term.*

In order to add new states to the current set of states, our abstraction operator proceeds as follows:

$$\mathit{abstract}(\mathcal{S}, \{s_1, \ldots, s_n\}) = \mathit{abs}(\mathit{abs}(\ldots \mathit{abs}(\mathcal{S}, s'_1) \ldots, s'_{n-1}), s'_n)$$

where:

$$s_i \Longrightarrow^*_{\mathsf{replace/flatten}} s'_i \not\Longrightarrow_{\mathsf{replace/flatten}} \quad (\text{for all } i = 1, \ldots, n)$$

Basically, function *abstract* starts by flattening the input states by applying (zero or one steps of) rule replace, followed by (zero or more steps of) rule flatten. Function *abs* is defined inductively on the structure of flattened states (according to Lemma 1):

$abs(\mathcal{S}, \langle x, [\,] \rangle) = \mathcal{S}$
$abs(\mathcal{S}, \langle c(\overline{t_n}), [\,] \rangle) = abstract(\mathcal{S}, \mathcal{S}')$
if $\overline{t'_m}$ are the maximal operation-rooted subterms of $c(\overline{t_n})$ and $\mathcal{S}' = \{\overline{\langle t'_m, [\,] \rangle}\}$
$abs(\mathcal{S}, \langle f(\overline{t_n}), S \rangle) =$

$$\begin{cases} \mathcal{S} \cup \{\langle f(\overline{t_n}), S \rangle\} & \text{if } \nexists \langle f(\overline{t'_n}), S' \rangle \in \mathcal{S} \\ \mathcal{S} & \text{else if } \langle f(\overline{t_n}), S \rangle \text{ is } \mathcal{S}\text{-closed} \\ abstract(\mathcal{S}^*, \mathcal{S}'') & \text{otherwise, where } \langle f(\overline{t'_n}), S' \rangle \in \mathcal{S}, \\ & msg(\langle f(\overline{t'_n}), S' \rangle, \langle f(\overline{t_n}), S \rangle) = (\langle f(\overline{t''_n}), S' \rangle, \mathcal{S}''), \\ & \text{and } \mathcal{S}^* = (\mathcal{S} \setminus \{\langle f(\overline{t'_n}), S' \rangle\}) \cup \{\langle f(\overline{t''_n}), S' \rangle\} \end{cases}$$

Informally speaking, function *abs* determines the corresponding action depending on the first component of the new state. If it is a variable, we discard the state. If it is constructor-rooted, we try to (recursively) add the maximal operation-rooted subterms. If it is a function call, then we have three possibilities:

- If there is no call to the same function in the current set, the new state is added to the current set of states.
- If there is a call to the same function in the current set, but the new call is *closed* w.r.t. this set, it is discarded.
- Otherwise, we generalize the new state and the existing state with the same outermost function—which is trivially unique by definition of *abstract*—and, then, we try to (recursively) add the states computed by function *msg*.

The notion of closedness is easily extended from expressions to states: a state $\langle t, S \rangle$ is *closed* w.r.t. a set of states $\mathcal{S}$ iff $S[t]$ is T-closed (according to Def. 1), with $T = \{S'[t'] \mid \langle t', S' \rangle \in \mathcal{S}\}$. Here, $S[t]$ denotes the term represented by $\langle t, S \rangle$, i.e., inner calls are moved back to their positions in the outer calls of the stack. For instance, given the state $\langle t, S \rangle = \langle \mathtt{y}, [(\mathtt{len\ x_2},\ \mathtt{x_2}), (\mathtt{fst\ (x_1,\ snd\ z)},\ \mathtt{x_1})] \rangle$, we have $S[t] = \mathtt{fst\ (len\ y,\ snd\ z)}$.

The operator *msg* on states is defined as follows. First, we recall the standard notion of *msg* on terms: a term t is a *generalization* of terms t_1 and t_2 if both t_1 and t_2 are instances of t; furthermore, term t is the *msg* of t_1 and t_2 if t is a generalization of t_1 and t_2 and, for any other generalization t' of t_1 and t_2, t is an instance of t'. Now, the *msg* of two states is defined by

$$msg(\langle t_1, S_1 \rangle, \langle t_2, S_2 \rangle) \;=\; (\langle t, S_1 \rangle, calls(\sigma_1) \cup calls(\sigma_2) \cup calls(S_2))$$

where $msg(t_1, t_2) = t$, and σ_1 and σ_2 are the matching substitutions, i.e., $\sigma_1(t) = t_1$ and $\sigma_2(t) = t_2$. The auxiliary function *calls* returns a set of states of the form $\langle t, [\,] \rangle$ for each maximal operation-rooted term t in (the range of) a substitution or in a stack. The abstraction operator is safe in the following sense:

Lemma 2. *Let $\mathcal{S}$ be a set of flattened states and $\mathcal{S}'$ a set of unfolded states (as returned by unfold). Then the states in $\mathcal{S} \cup \mathcal{S}'$ are closed w.r.t. abstract$(\mathcal{S}, \mathcal{S}')$.*

The above lemma is the key result to ensure the correctness of our approach. Indeed, it will allow us to prove that the generated slice is executable and that it contains all the functions which are needed to execute the slicing criterion.

Example 3. Let us consider again the program of Example 2. Given the slicing criterion "`foo [] y z`", the initial set of states is $\mathcal{S}_0 = \{\langle$`foo [] y z`$, [\,]\rangle\}$. Now, we show the sequence of iterations performed by the algorithm in Fig. 4:

$\mathcal{S}'_0 = \{\langle$`fst (len (app [] y), snd z)`$, [\,]\rangle\}$
$\mathcal{S}_1 = \mathcal{S}_0 \cup \{\langle$`app [] y`$, [($`len x2, x2`$), ($`fst (x1, snd z), x1`$)]\rangle\}$

$\mathcal{S}'_1 = \mathcal{S}'_0 \cup \{\langle$`y`$, [($`len x2, x2`$), ($`fst (x1, snd z), x1`$)]\rangle\}$
$\mathcal{S}_2 = \mathcal{S}_1 \cup \{\langle$`len y`$, [($`fst (x1, snd z), x1`$)]\rangle\}$

$\mathcal{S}'_2 = \mathcal{S}'_1 \cup \{\langle$`Z`$, [($`fst (x1, snd z), x1`$)]\rangle, \langle$`Succ (len ys)`$, [($`fst (x1, snd z), x1`$)]\rangle\}$
$\mathcal{S}_3 = \mathcal{S}_2 \cup \{\langle$`fst (x3, snd z)`$, [\,]\rangle\}$

$\mathcal{S}'_3 = \mathcal{S}'_2 \cup \{\langle$`x3`$, [\,]\rangle\}$ and $\mathcal{S}_4 = \mathcal{S}_3$

where $\mathcal{S}'_i = \mathit{unfold}(\mathcal{S}_i, \mathcal{R})$ and $\mathcal{S}_{i+1} = \mathit{abstract}(\mathcal{S}_i, \mathcal{S}'_i)$, for $i = 0, \ldots, 3$. Therefore, the algorithm returns the following set of states:

$$\begin{array}{l}\mathcal{S} = \{\ \langle\texttt{foo [] y z}, [\,]\rangle,\ \langle\texttt{app [] y}, [(\texttt{len x2, x2}), (\texttt{fst (x1, snd z), x1})]\rangle,\\ \qquad \langle\texttt{len y}, [(\texttt{fst (x1, snd z), x1})]\rangle,\ \langle\texttt{fst (x3, snd z)}, [\,]\rangle \qquad\qquad \}\end{array}$$

The total correctness of the algorithm in Fig. 4 is stated in the following theorem:

Theorem 1. *Given a flat program $\mathcal{R}$ and an initial term t, the algorithm in Fig. 4 terminates computing a set of states $\mathcal{S}$ such that $\langle t, [\,]\rangle$ is $\mathcal{S}$-closed.*

3.3 Extraction of the Slice

The final step of the process consists in the construction of the residual program, i.e., the program slice. Let us recall that it must be a fragment of the original program—thus no instantiation of variables is allowed—and produce the same outputs for the slicing criterion as the original program. Therefore, we can only remove certain parts of the program: case alternatives which are not reachable from the slicing criterion as well as (inner) function calls which are not needed to evaluate the slicing criterion. While case alternatives can simply be discarded, inner calls which are not needed are replaced by a distinguished symbol $\top$ (in practice, any constant value may play the role of $\top$ since the evaluation of such a subterm will not be required). The interest in producing *executable* slices comes from the fact that it facilitates program reuse and, more importantly, it allows us to apply a number of existing techniques to the computed slice (e.g., debugging, program analysis, verification, program transformation, etc).

Let us assume that, given a program $\mathcal{R}$ and a term t, the algorithm of Fig. 4 returns the set of states $\mathcal{S}$. In principle, we could construct a residual program following the standard narrowing-driven specialization method as follows: for each

state $\langle t, S\rangle \in \mathcal{S}$, we produce a residual rule $S[t] = S[t']$ (we ignore here the renaming of expressions), where $\langle t, S\rangle \Longrightarrow_{\mathsf{fun}} \langle t'', S\rangle \Longrightarrow^*_{\mathsf{select/guess}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{select/guess}}$. The residual program constructed in this way would be correct in the sense that any computation for t in $\mathcal{R}$ could also be performed in the residual program. In general, however, the residual program would not be a fragment of the original one since the left-hand sides of residual rules may contain non-variable arguments (even nested function calls). Therefore, we proceed as follows:

- Firstly, residual rules are constructed only for the first components of the states in $\mathcal{S}$. This is safe in our approach because the "context" (recorded in the stack) is not used when performing a complete one-step evaluation; moreover, if the evaluation of some call in the stack is required, it will appear in the first component of some other state.
- Secondly, we slightly change the rules of Fig. 3 so that only some case alternatives can be removed (if they are not reachable) but the case structure remains.
- Finally, in order to avoid the instantiation of variables, the new rules maintain separately the program expression being reduced and the bindings for the program variables.

The slice built in this way is appropriate *when the extended operational semantics of Fig. 3 is considered*. Unfortunately, this is no longer true under the standard operational semantics when the program contains *non-terminating* functions. Consider, for instance, the following program rules:

```
g x = g x                f x = case x of { [] -> [] }
```

Given the initial term "`f (g x)`", whose flattening is $\langle$`g x, [(f y, y)]`$\rangle$, the algorithm of Fig. 4 returns the final set $\{\langle$`g x, [(f y, y)]`$\rangle\}$ since the evaluation of "`f y`" is not required. Therefore, the computed slice would be the rule `g x = g x`. Under the extended semantics, the initial state $\langle$`f (g x), []`$\rangle$ performs exactly the same steps in both the original program and the computed slice. However, under the standard semantics (where outer function calls are not delayed), the initial term "`f (g x)`" is reduced to "`case (g x) of {[] → []}`" (and then enters into an infinite loop) which is not possible in the slice. In order to have a complete equivalence w.r.t. the standard semantics, we also need to add residual definitions for those functions in the stacks which are not closed w.r.t. the set of first components of the states in $\mathcal{S}$. The following auxiliary function returns all the relevant terms:

$$\mathit{residual\_calls}(\mathcal{S}) = T_{\mathcal{S}} \cup \{t' \mid \langle t, S\rangle \in \mathcal{S},\ t' \in \mathit{calls}(S),\ \mathsf{and}\ t'\ \mathsf{is\ not}\ T_{\mathcal{S}}\text{-}\mathsf{closed}\}$$

where $\mathcal{S}$ is the set of states returned by the algorithm of Fig. 4 and

$$T_{\mathcal{S}} = \{t \mid \langle t, S\rangle \in \mathcal{S}\}$$

The program slice is then computed by using the following function:

$$\begin{aligned}\mathit{build\_slice}(T) = \mathsf{if}\ T = \{\,\}\ \mathsf{then}\ \{\,\}\ \ \mathsf{else}\ \{f(\overline{x_n}) = e'\} \cup \mathit{build\_slice}(T')\\ \mathsf{where}\ \ T = \{f(\overline{t_n})\} \cup T',\ \ f(\overline{x_n}) = e \in \mathcal{R},\ \ \overline{y_n}\ \mathsf{are\ fresh},\\ \rho = \{\overline{y_n \mapsto t_n}\},\ \mathsf{and}\ [\![e]\!]\rho \longrightarrow^* e' \not\longrightarrow\end{aligned}$$

Rule	*Expr*	$\longrightarrow$	*Expr*
var	$[\![x]\!]\rho$	$\longrightarrow$	x
cons	$[\![c(\overline{t_n})]\!]\rho$	$\longrightarrow$	$c([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho)$
select	$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]\rho$	$\longrightarrow$	$(f)case\ x\ of\ \{p_i \to [\![e_i]\!]\rho'\}$
guess	$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]\rho$	$\longrightarrow$	$(f)case\ x\ of\ \{\overline{p_k \to [\![e_k]\!]\rho_k}\}$
fun	$[\![f(\overline{t_n})]\!]\rho$	$\longrightarrow$	$f([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho)$
remove	$[\![f(\overline{t_n})]\!]\rho$	$\longrightarrow$	$\top$

where in select: $\rho(x) = c(\overline{t_n}),\ p_i = c(\overline{x_n}),\ \rho' = \{\overline{x_n \mapsto t_n}\} \circ \rho,$ and $i \in \{1, \ldots, k\}$
guess: $\rho(x) \in \mathcal{X},\ p_i = \{x \mapsto p_i\} \circ \rho,$ and $i \in \{1, \ldots, k\}$
fun: $\rho(f(\overline{t_n}))$ is closed w.r.t. $residual\_calls(\mathcal{S})$
remove: $\rho(f(\overline{t_n}))$ is not closed w.r.t. $residual\_calls(\mathcal{S})$

Fig. 5. Simplified Unfolding Rules

In order to extract the program slice, function *build_slice* is called with the result of function *residual_calls*, i.e., the initial call is as follows:

$$build\_slice(residual\_calls(\mathcal{S}))$$

where the set $\mathcal{S}$ is the output of the algorithm in Fig. 4. The new calculus which is used to construct the rules of the slice is depicted in Fig. 5. First, note that the symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]\rho$ are purely syntactical, i.e., they are only used to mark subexpressions where the inference rules may be applied. The substitution ρ is used to store the bindings for the program variables. Let us briefly explain the rules of the new calculus. Rule var simply returns a variable unchanged. Rule cons applies to constructor-rooted terms; it leaves the constructor symbol and, then, it is (recursively) applied to inspect the arguments. Rules select and guess proceed similarly to their counterpart in Fig. 3 but leave the case structure; the substitution ρ is used to check the current value of the case argument. We only deal with *variable* case arguments since the considered expression is the right-hand side of some program rule (see Fig. 1). Note that rule guess is now deterministic (and, thus, the entire calculus). Finally, rules fun and remove are used to reduce function calls: when the function call is closed w.r.t. $residual\_calls(\mathcal{S})$, we proceed as in rule cons; otherwise, we return $\top$ (which means that the evaluation of this function call is not needed).

The following example illustrates the computation of a program slice.

Example 4. Let us consider the set of states computed in Example 3. From this set, function *residual_calls* returns the set of terms:

$$\{\texttt{foo [ ] y z},\ \texttt{app [ ] y},\ \texttt{len y},\ \texttt{fst (x, snd z)}\}$$

Now, we construct a residual rule for each term of the set. For "`foo [ ] y z`", the residual rule is: "`foo x y z = fst (len (app x y))`, $\top$)", since the following derivation is possible (with $\rho = \{x \mapsto [\,]\}$):

$$\begin{array}{rll} [\![\texttt{fst (len (app x y), snd z)}]\!]\rho & \longrightarrow^*_{\mathsf{fun}} & \texttt{fst (len (app } [\![\texttt{x}]\!]\rho\ [\![\texttt{y}]\!]\rho), [\![\texttt{snd z}]\!]\rho) \\ & \longrightarrow^*_{\mathsf{var}} & \texttt{fst (len (app x y)}, [\![\texttt{snd z}]\!]\rho) \\ & \longrightarrow_{\mathsf{remove}} & \texttt{fst (len (app x y)}, \top) \end{array}$$

By constructing a residual rule associated to each of the remaining terms, the computed slice coincides with the one which is shown in Example 2.

The computed slice is executable and contains all the functions which are needed to evaluate the slicing criterion. This property is inherited by the correctness of the underlying partial evaluation process.

Theorem 2. *Let $\mathcal{R}$ be a flat program and t a term. Let $\mathcal{S}$ be a set of states computed by the algorithm in Fig. 4 from $\mathcal{R}$ and t. Then t computes the same values in $\mathcal{R}$ and in build_slice(residual_calls($\mathcal{S}$)).*

This section has shown the definition of a practical forward slicing technique based on a partial evaluation scheme. In the next section, we discuss several possibilities to design a backward slicing method following a similar style.

4 Backward Slicing

In this section, we informally explain how the scheme presented so far could be extended in order to perform (static/dynamic) backward slicing. In principle, backward slicing can be seen from two different perspectives (in either case, we consider that the program contains a distinguished function, *main*, which is used to start the execution of the program):

Type I: A naive approach to backward slicing implies extracting those statements of the original program that may reach the slicing criterion, when the program is executed starting at function *main*.

Type II: A different notion of backward slicing has been introduced in [24] to perform program slicing of functional programs. In this work, the slicing criterion is some *part* of the output of function *main* (described by means of a projection). Then the method extracts those program statements which are needed to compute the desired fragment of the output.

A naive approach to Type I backward slicing can easily be defined from the technique presented so far. Currently, the algorithm in Fig. 4 only computes the set of reachable function calls, starting at the slicing criterion. Now, we should start the algorithm with a call to function *main*. However, in contrast to the approach of the previous section, we should explicitly compute the graph of dependences, i.e., it is not sufficient to have the set of reachable nodes but we also need the relationships among them. By traversing this graph backwards (from the slicing criterion), we could easily select the desired nodes and, then, construct its associated slice (by using the same program extraction method of Sect. 3.3).

Type II backward slicing is more useful but also more complex. Nevertheless, we could still adapt the previous developments in order to cope with this form of slicing. Functional logic languages naturally support some form of constraint solving; indeed, they accept the evaluation of either function calls $f(\overline{t_n})$ and equational constraints $f(\overline{t_n})$ =:= t. In this context, the slicing criterion can be

defined by providing an equational constraint $main(\overline{t_n})$ `=:=` t, where t is a term which contains free variables for those parts of the output which are not relevant for the slice, and some special values for those parts of the output whose computation is required. Then the algorithm in Fig. 4 could be adapted to work with equational constraints and only evaluate those function calls which are needed to produce the outputs determined by t. The resulting method would share many similarities with the technique for backward slicing of [24].

These approaches are subject of ongoing work. It would be also interesting to relate backward slicing with recent approaches to function inversion [1,26].

5 Implementation

In order to check the practicality of the ideas presented so far, a prototype implementation of the program slicing tool has been developed[4]. In particular, it has been implemented by adapting an existing partial evaluator for Curry programs [3]. The resulting tool covers not only the flat programs of Sect. 2, but source Curry programs (which are automatically translated to the flat syntax). We accept higher-order functions, overlapping left-hand sides, several predefined functions, etc; all these features were also available in the underlying partial evaluator. It required a small implementation effort, i.e., only the underlying meta-interpreter needed significant changes.

Our slicing tool is able to compute the slice of Example 2, thus it is strictly more powerful than naive approaches based on graphs of functional dependences. Preliminary results are quite encouraging. In fact, in contrast to the original partial evaluator, it can deal with larger programs efficiently; this is mainly due to the monovariant/monogenetic nature of the basic algorithm, which simplifies the computation of a closed set of terms.

6 Conclusions and Related Work

This work presents the first approach to forward slicing of multi-paradigm (functional logic) programs. Our developments rely on adapting an online partial evaluation scheme for such programs. Thus, the implementation of the resulting slicing tool can easily be undertaken by adapting existing partial evaluation tools. Moreover, our approach helps to clarify the relation between program slicing and partial evaluation in a functional logic context. The application of our developments to pure (lazy) functional programs would be straightforward, since the considered language is a conservative extension of a pure lazy functional language and the (online) partial evaluation techniques are similar (e.g., positive supercompilation [28]).

As mentioned in the introduction, we found very few approaches to program slicing in the context of declarative programming. Some exceptions are [13,21,24,25,29]). Among them, the closest to our work is [24], the only of them

[4] It is publicly available from `http://www.dsic.upv.es/~gvidal`.

which considers a functional syntax for programs. In contrast to our approach, [24] defines a *backward* slicing technique. As in our case, some of his developments are inspired by previous techniques coming from partial evaluation (like, e.g., [18,23]). In the context of imperative programs, [8] has shown how forward slicing can be used to carry out binding-times analyses for imperative programs. Our approach can be seen as complementary: we show how to use (online) partial evaluation to perform forward slicing in a declarative context.

Future work includes the definition of appropriate algorithms to perform backward slicing (as discussed in Sect. 4). It would be also interesting to investigate alternative approaches to program slicing based on abstract interpretation (e.g., by approximating the operational semantics presented in Sect. 3.2). A different line of research involves the definition of a forward slicing technique for logic programs by exploiting the similarities between narrowing-driven specialization and conjunctive partial deduction [9]. In this context, precision could be improved by considering refined frameworks for partial deduction like, e.g., *abstract* partial deduction [20]. This topic is subject of ongoing work.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2002 for many useful comments and suggestions.

References

1. S.M. Abramov and R. Glück. The Universal Resolving Algorithm: Inverse Computation in a Functional Language. In *Mathematics of Program Construction. Proceedings*, pages 187–212. Springer LNCS 1837, 2000.
2. E. Albert, M. Hanus, F. Huch, J. Olvier, and G. Vidal. Operational Semantics for Functional Logic Languages. In *Proc. of the Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'02)*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
3. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
4. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.
6. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
8. M. Das, T. Reps, and P. Van Hentenryck. Semantic Foundations of Binding-Time Analysis for Imperative Programs. In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pages 100–110, 1995.

9. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
10. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.
12. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, 1996.
13. V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of SAS'98*, pages 115–133, 1998.
14. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.
15. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.
16. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
18. J. Hughes. Backwards Analysis of Functional Programs. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, Amsterdam, 1988.
19. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.
20. M. Leuschel. Program Specialization and Abstract Interpretation Reconciled. In *Proc. of the Joint Int'l Conf. and Symp. on Logic Programming (JICSLP'98)*, pages 220–234. MIT Press, 1998.
21. M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc. of LOPSTR'96*, pages 83–103. LNCS 1207 83–103, 1996.
22. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
23. T. Mogensen. Separating Binding-Times in Language Specifications. In *Proc. of 4th Int'l Conf. on Functional Programming and Computer Architecture (FPCA'89)*, pages 12–25. ACM, New York, 1989.
24. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.
25. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of SAS'96*, pages 317–331. Springer LNCS 1145, 1996.
26. J.P. Secher and M.H. Sørensen. From Checking to Inference via Driving and Dag Grammars. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 41–51. ACM, New York, 2002.
27. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

28. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
29. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
30. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
31. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
32. G. Vidal. Forward Slicing by Partial Evaluation. Technical report, DSIC, Technical University of Valencia, 2003. Available from `http://www.dsic.upv.es/~gvidal`.
33. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.
34. M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

A Fixed Point Semantics for Logic Programs Extended with Cuts

Wim Vanhoof, Remko Tronçon, and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium
{wimvh,remko,maurice}@cs.kuleuven.ac.be

Abstract. In this paper, we develop a bottom-up fixed point semantics for pure Prolog programs extended with !/0 that allows to reconstruct the operational semantics of a particular goal. Our semantics captures both the order in which solutions are computed by SLD-resolution and their multiplicity.

1 Introduction

Semantics of logic programming is an active research area within logic programming, and several approaches have been developed within the field to define the "meaning" of a program. The ability to define the meaning of a program in a formal way has a number of clear advantages. It cannot only help understanding the program, but it also provides a formal way to prove correctness of program transformations (like e.g. partial evaluation [14]). Furthermore, abstracting a suitable semantics is an appealing way to derive correct program analyses or transformations. One of the most attractive features of logic programming is the equivalence between the so-called *declarative* (or fixed point) semantics of a logic program and its *operational* semantics. In its most basic form, the declarative semantics of a logic program P is defined as its minimal Herbrand model, which can be shown equivalent [26] to the success set of the program, when the latter is defined as the set of ground atoms that can be refuted in P.

The s-semantics approach [12,5] to logic programming extends this basic equivalence result by allowing variables in the Herbrand universe. The resulting fixed point semantics computes a denotation of a program P that, when combined with an atomic query, characterises exactly the set of computed answer substitutions of that query in P. By abstracting the fixed-point semantics, one can obtain a program analysis that computes goal-independent analysis results that correctly model the computed answer substitutions created when constructing refutations. This makes the s-semantics an appealing semantics for program analysis and transformation and it has been used as a foundation of several abstract interpretation frameworks [3,9].

Appealing as the s-semantics approach may be for logic program analysis and transformation, it does not model any control information that is associated with evaluating a Prolog program. While the s-semantics correctly models the computed answer substitutions that are computed by SLD-resolution, it does

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 238–257, 2003.

not model the order in which these answers are computed, nor their multiplicity. Consequently, the s-semantics is not a sufficient basis for the analysis or transformation of Prolog programs when order and multiplicity of the computed answers need to be preserved. These control aspects are important when analysing Prolog programs, in particular in the presence of cuts.

In this work, we develop an alternative fixed point semantics. The bottom-up derivations in the resulting denotation incorporate the presence of cuts in the program and contain the necessary control information such that a simple combination operator suffices to retrieve a sequence of answers for a particular atomic query. The resulting sequence is equivalent with the sequence of computed answers as returned by SLD-resolution using the left-to-right selection rule of Prolog. Our semantics hence captures the order in which solutions are computed, their multiplicity and deals with programs that contain cut constructs. The main motivation for our work is to construct a clean and simple fixed point semantics that can easily be abstracted to obtain goal-independent program analyses or transformations that preserve the operational semantics of the program under SLD-resolution. In particular, we feel our semantics can serve as a semantic basis for bottom-up partial evaluation.

The remainder of this paper is organised as follows. In Section 2 we introduce some basic concepts and notation, and define the operational semantics of a program as a sequence of computed answers. In Section 3 we introduce a fixed point semantics for pure Prolog programs and prove the equivalence between this semantics and the operational semantics defined in Section 2. We extend these results towards programs that contain the cut operator in Section 4. We conclude the paper in Section 5, where we discuss related work and give a possible application of our semantics.

2 Preliminaries

In what follows, we assume the reader to be familiar with the basic logic programming concepts as they are found, for example, in [1,18]. In this work, we restrict ourselves to definite programs, and assume the Prolog variant of logic programming. In particular, we consider a program to be defined as a sequence of program clauses, where each program clause is labelled by a natural number identifying its position in the program. We will denote the i-th program clause as $H \overset{i}{\leftarrow} B_1, \ldots, B_n$, although the label may be dropped when irrelevant. As usual, a substitution is defined as a finite mapping from distinct variables to terms. The set of all substitutions is denoted by $Subst$. As usual, we denote substitutions as $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where each $X_i \neq t_i$. Given substitutions θ and σ, $\theta\sigma$ denotes there composition and if V is a set of variables, then $\theta_{|_V}$ denotes θ restricted to V. The substitution ϵ denotes the empty substitution. If E is an expression (be it a term, atom or clause) and θ a substitution, then $E\theta$ denotes the result of applying θ to E and is defined as the expression obtained from E by simultaneously replacing the variables from the domain of θ that occur in E by their corresponding term. We call $E\theta$ an instance of E. If E is an expression

and F is an instance of E, then E is said to be more general than F, denoted $E \preceq F$, or simply a generalisation of F. If E and F are expressions such that E is an instance of F and F is an instance of E, then E and F are called variants, denoted by $E \approx F$. A substitution θ such that $E\theta \approx E$ is called a renaming substitution for E. Given a set of expressions I and a syntactic object C, we denote by $A_1, \ldots, A_n \ll_C I$ that $A_1, \ldots, A_n$ are variants of expressions in I renamed apart from C and from each other. If $E \preceq F$ and $E \not\approx F$, we say that E is strictly more general than F, denoted with $E \prec F$. A substitution θ is a unifier for atoms A and B if $A\theta = B\theta$; it is a most general unifier for A and B if for each unifier σ of A and B there exists a substitution γ such that $\sigma = \theta\gamma$. As usual, we denote the most general unifier for atoms A and B (modulo variable renaming) by $mgu(A, B)$. The notion of mgu is easily extended to conjunctions of atoms. As usual, we assume a goal (or query) to be a clause of the form $\leftarrow A_1, \ldots, A_n$. In the rest of the paper, we will often denote such a goal by the symbol Q.

The operational behavior of a logic program is characterised by the notion of SLD-derivation [1,18]. As we have introduced labeled clauses, we can use labels instead of clauses when formalising the notion of SLD-derivation. Hence, an SLD-derivation of a goal Q_0 in a program P consists of a possibly infinite sequence of goals, denoted $Q_0 \overset{i_1,\theta_1}{\rightarrow} Q_1 \overset{i_2,\theta_2}{\rightarrow} Q_2 \ldots$ such that each Q_j is derived from Q_{j-1} and a freshly renamed copy of the i_j-th program clause using the most general unifier θ_j. Since we consider the Prolog execution mechanism, we assume that SLD-derivations are constructed using the left-to-right selection rule, that is: a goal Q_{j+1} is derived from Q_j and a clause C using θ if and only if Q_{j+1} is the goal $\leftarrow (B_1, \ldots, B_n, A_2, \ldots, A_m)\theta$ where $Q_j = \leftarrow A_1, \ldots, A_m$, C is the clause $H \leftarrow B_1, \ldots, B_n$ and $\theta = mgu(H, A_1)$. An SLD-derivation can be finite or infinite. A finite SLD-derivation that ends in an empty goal (i.e. a goal without any atom) is called a successful derivation, or an SLD-refutation. We denote an SLD-refutation of Q in P by $Q \rightsquigarrow_P \square$. A finite derivation that ends in a goal of which the selected atom does not unify with the head of any program clause is called a failed SLD-derivation. A substitution θ is a computed answer substitution (abbreviated as *cas*) for Q in P if there exists an SLD-refutation $Q \rightsquigarrow_P \square$ such that σ is the composition of the most general unifiers associated to the derivation and θ is the restriction of σ to the variables of Q. We denote this fact by $Q \overset{\theta}{\rightsquigarrow}_P \square$. In what follows we also use partial derivations and write $Q \overset{\theta}{\rightsquigarrow}_P R$ with θ the restriction to the variables of Q of the composition of the mgus (θ is called the cas of the partial derivation).

Since we assume that SLD-derivations are constructed using the left-to-right selection rule, such a derivation and its associated computed answer substitution are completely characterised (modulo variable renaming) by the sequence of labels identifying the clauses that were used in the derivation.

Definition 1 (Characteristic derivation). *The* characteristic derivation *of the SLD-derivation*

$$Q_0 \overset{i_1,\theta_1}{\rightarrow} Q_1 \overset{i_2,\theta_2}{\rightarrow} \ldots$$

is the sequence $\langle i_1, i_2, \ldots \rangle$ *of natural numbers.*

```
(1) path(X,Y):- edge(X,Y).                    (3) edge(a,b).
(2) path(X,Y):- edge(X,Z), path(Z,Y).         (4) edge(a,c).
                                              (5) edge(c,b).
```

Fig. 1.

Being sequences of natural numbers, we can impose the lexicographic order relation on characteristic derivations: $\langle i_1, i_2, \ldots\rangle > \langle j_1, j_2, \ldots\rangle$ if and only if either $i_1 > i_2$ or $i_1 = j_2$ and $\langle i_2, \ldots\rangle > \langle j_2, \ldots\rangle$. Note that this order relation induces an order relation over SLD-refutations. In what follows, we will often write $Q_1 \rightsquigarrow \Box > Q_2 \rightsquigarrow \Box$ if the characteristic derivation of $Q_1 \rightsquigarrow \Box$ is larger then the characteristic derivation of $Q_2 \rightsquigarrow \Box$. Note also that the order between characteristic refutations for a goal Q reflects precisely the order in which the refutations are constructed and answers are returned by a system that uses a left-to-right selection rule in combination with lexicographic ordering, as Prolog systems do.

Example 1. Consider as a working example the program in Fig. 1, consisting of the definition of the `path/2` predicate and a database of facts `edge/2`. The clauses are labelled for later reference. One can construct the following refutations of $\leftarrow path(a, X)$.

$$
\begin{aligned}
&(\leftarrow path(a,X)) \overset{1}{\rightarrow} (\leftarrow edge(a,X)) \overset{3,X=b}{\rightarrow} \Box\\
&(\leftarrow path(a,X)) \overset{1}{\rightarrow} (\leftarrow edge(a,X)) \overset{4,X=c}{\rightarrow} \Box\\
&(\leftarrow path(a,X)) \overset{2}{\rightarrow} (\leftarrow edge(a,Z), path(Z,X)) \overset{4,Z=c}{\rightarrow}\\
&\qquad\qquad (\leftarrow path(c,X)) \overset{1}{\rightarrow} (\leftarrow edge(c,X)) \overset{5,X=b}{\rightarrow} \Box
\end{aligned}
$$

Consequently, the characteristic derivations of these refutations are respectively $(1,3)$, $(1,4)$ and $(2,4,1,5)$ and we have that $(2,4,1,5) > (1,4) > (1,3)$.

The lexicographic order relation allows to define the semantics behavior of a Prolog program as follows:

Definition 2 (Operational semantics). *Let P be a program and Q a goal. The* operational behavior *of Q w.r.t. P is defined as $\mathcal{O}(P,Q) = \langle\theta_1, \theta_2, \theta_3, \ldots\rangle$ where*

- $\theta_i \in \mathcal{O}(P,Q)$ *if and only if* $Q \overset{\theta_i}{\rightsquigarrow}_P \Box$.
- $\forall\theta_i, \theta_j \in \mathcal{O}(P,Q)$*: if* $j > i$ *then* $Q \overset{\theta_j}{\rightsquigarrow}_P \Box > Q \overset{\theta_i}{\rightsquigarrow}_P \Box$

Modeling the operational behavior of a logic program as a sequence of answers rather than as a set (as e.g. in [12,5]) explicitly takes into account the order in which solutions are found, and allows for multiple occurrences of the same answers (as e.g. in [4,22,11,8]).

Example 2. If P represents the program in Fig. 1 and Q the goal $\leftarrow path(a, X)$, then we have that $\mathcal{O}(P,Q) = (\{X/b\}, \{X/c\}, \{X/b\})$.

Note that $\mathcal{O}(P,Q)$ may be an infinite sequence, modeling the case where the Prolog evaluation mechanism constructs an infinite number of derivations and returns an infinite number of answers. Also note that an infinite derivation does not contribute to $\mathcal{O}(P,Q)$. Hence the fact that $\theta_i \in \mathcal{O}(P,Q)$ does not imply that Prolog computes it, as Prolog can get lost in an infinite derivation before arriving at the derivation leading to θ_i. However, if θ_i *is* computed, so are $\theta_1, \ldots, \theta_{i-1}$.

In what follows, we define a bottom-up characterisation of the operational semantics. We first define the semantics for pure Prolog programs without cut, and extend it later on to deal with the cut operator.

3 A Bottom-Up Semantics for Pure Prolog

The s-semantics approach [12,5] defines a fixed point semantics that captures to some extent the procedural behavior of logic programs. The usual Herbrand base [18,1] is extended to the set of all the possibly non-ground atoms modulo variance and an immediate consequences operator, say T_P^s, is defined. Given a set of atoms I, $T_P^s(I)$ comprises a new set of atoms that can be derived by unifying atoms from I with the body atoms of a clause in P. The fixed point semantics of a program then equals the fixed point of the T_P^s-operator on an initially empty set of atoms, denoted by $\mathrm{lfp}(T_P^s)$. These notions are due to [12,5,6]. This semantics captures the operational behavior of a program to the extent that $p(X_1, \ldots, X_n)\theta \in \mathrm{lfp}(T_P^s)$ if and only if $p(X_1, \ldots, X_n) \overset{\theta}{\leadsto}_P \square$ [12,5]. As said before, it does not capture the order in which solutions are computed by SLD-resolution, nor their multiplicity. If we want to model the operational behavior of a program as defined above by a bottom-up, fixed point semantics, the atoms in $\mathrm{lfp}(T_P^s)$ need to be related to each other in the same way that the corresponding SLD-derivations are related. In what follows, we introduce an operator that employs the labels that are associated to the program clauses in order to trace *how* an atom was derived by bottom-up derivation. Given this trace, it becomes possible to reconstruct the characteristic derivation of an SLD-refutation for the atom, as such providing a suitable mechanism for relating the atoms derived bottom-up.

In order to trace how an atom is derived by an immediate consequences operator, we associate each such atom with an AND-tree whose nodes are labelled with the labels identifying the clauses that were used for resolution. That is, if a clause $H \overset{l}{\leftarrow} B_1, \ldots, B_n$ is used to derive some atom A from atoms $A_1, \ldots, A_n$ then the AND-tree associated with A has root l and has as children the AND-trees τ_i associated with the derivations of the atoms A_i. In what follows, we use the notation $l(\tau_1, \ldots, \tau_n)$ for an AND-tree with label l and children $\tau_1, \ldots, \tau_n$. We call the combination of an atom with such an AND-tree a *bottom-up derivation*. If we denote the set of all bottom-up derivations by $\mathcal{BU}$, we can define our immediate consequences operator T_P^d as follows:

Definition 3 (T_P^d-operator). *Let P be a definite program and I a set of bottom-up derivations. We define $T_P^d : \wp(\mathcal{BU}) \mapsto \wp(\mathcal{BU})$ as follows:*

$$T_P^d(I) = \left\{ (H\theta, \tau) \middle| \begin{array}{l} \textit{Let } C \textit{ be the clause } H \overset{l}{\leftarrow} B_1, \ldots, B_n \in P, \\ (A_1, \tau_1), \ldots, (A_n, \tau_n) \ll_C I, \\ \theta = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n)), \\ \tau = l(\tau_1, \ldots, \tau_n) \end{array} \right\}$$

T_P^d is monotonic on the complete lattice $(\wp(\mathcal{BU}), \subseteq)$ and hence the existence of its least fixed point, $\mathrm{lfp}(T_P^d)$ is immediate [18]. The fixed point semantics of a program P is then defined as $\mathcal{F}(P) = \mathrm{lfp}(T_P^d)$. For an atom A, we define $\mathcal{F}_A(P)$ as the set of bottom-up derivations from $\mathrm{lfp}(T_P^d)$ of which the atom unifies with A, formally:

Definition 4. *Let P be a definite program and A an atom. We define*

$$\mathcal{F}_A(P) = \{(A', \tau) \mid (A', \tau) \in \mathit{lfp}(T_P^d) \textit{ and } mgu(A, A') \textit{ exists.}\}.$$

Note that it is possible for a single atom A' to occur multiple times in $\mathcal{F}_A(P)$, each time with a different associated bottom-up derivation. This corresponds with $\mathcal{O}(P, \leftarrow A)$ in the fact that a single answer can be returned multiple times, each time with a different computation. This is in contrast with the classic T_P operator, where the lack of the explicit derivations prohibits multiple occurrences of the same atom in $\mathrm{lfp}(T_P)$.

Example 3. Let P denote the program in Fig. 1. We have that

$$\mathcal{F}_{path(X,Y)}(P) = \left\{ \begin{array}{ll} (path(a,b), 1(3)) & (path(a,c), 1(4)) \\ (path(c,b), 1(5)) & (path(a,b), 2(4, 1(5))) \end{array} \right\}$$

By a standard technique, we get that $(A, \tau) \in \mathrm{lfp}(T_P^d)$ if and only if $\leftarrow A \overset{\epsilon}{\rightsquigarrow}_P \square$ with ϵ the empty computed answer substitution. Being selection-rule independent, a bottom-up derivation characterises a number of SLD-refutations; one for each combination of selected atoms in successive SLD-derivation steps. Recall that we want to relate our fixed point semantics with the operational semantics that is defined in terms of SLD-derivations using the left to right selection rule. Hence, we are in particular interested in relating a bottom-up derivation (A, τ) with an SLD-refutation for A that is constructed using the left-to-right selection rule. We show that by collecting the labels of the AND-tree in a bottom-up derivation (A, τ) in a depth-first, left-to-right manner, one obtains the characteristic derivation of an SLD-refutation for A.

Definition 5. *Let τ denote a bottom-up derivation. The* depth-first, left-to-right traversal *of τ, denoted by $\overline{\tau}$ is the sequence over $\mathbb{N}$ recursively defined as follows:*

$$\overline{l(\tau_1, \ldots, \tau_n)} = l \bullet \overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$$

where $\bullet$ denotes concatenation of sequences.

For the bottom-up derivations of Example 1, we have that $\overline{1(3)} = (1, 3)$ and $\overline{2(4, 1(5))} = (2, 4, 1, 5)$. Note that the order over sequences induces an order over AND-trees. In what follows, we write $\tau_1 < \tau_2$ when $\overline{\tau_1} < \overline{\tau_2}$. Now, the following property holds:

Proposition 1. *Let P be a program. If $(A, \tau) \in lfp(T_P^d)$ and $\overline{\tau}$ denotes the depth-first, left-to-right traversal of τ, then $\overline{\tau}$ is the characteristic derivation of an SLD-refutation of $\leftarrow A$ in P with empty computed answer substitution, constructed by the left-to-right selection rule.*

Proof. The proof is by induction on the number of iterations of T_P^d. First, assume $(A, \tau) \in T_P^d \uparrow 1$. Since $(A, \tau) \in T_P^d(\emptyset)$, τ must be of the form $l()$ (having no children) which means that $A \overset{l}{\leftarrow}$ is a clause in P. Consequently, there exists an SLD-refutation of $\leftarrow A$ in P with cas ϵ whose characteristic derivation is l.

Now, assume that it holds for bottom-up derivations obtained in k iterations (induction hypothesis). Assume that $(A, \tau) \in T_P^d \uparrow k+1$. By construction, τ is of the form $l(\tau_1, \ldots, \tau_n)$ and $\overline{\tau} = l \bullet \overline{\tau_1} \bullet \ldots \bullet \overline{\tau_n}$. By definition, $A = H\theta$ with $H \overset{l}{\leftarrow} B_1, \ldots, B_n \in P$, $\theta = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n))$ and $(A_1, \tau_1), \ldots, (A_n, \tau_n) \in T_P^d \uparrow k$. Since $A = H\theta$, we have that $mgu(H, A) = \theta_{|_H}$ and hence there exists a partial SLD-derivation $\leftarrow A \overset{\epsilon}{\leadsto}_P (\leftarrow B_1, \ldots, B_n)\theta_{|_H}$ with the sequence l as its characteristic derivation. Note that the cas is ϵ as A is an instance of H. Now, remains to prove that $(\leftarrow B_1, \ldots, B_n)\theta_{|_H} \overset{\sigma}{\leadsto}_P \square$ with $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$ as the characteristic derivation and σ restricted to the variables of $A(= H\theta_{|_H})$ the empty substitution, i.e. $dom(\sigma) \cap vars(H\theta_{|_H}) = \emptyset$.

We can write θ as $\theta_{|_H} \cup \theta_B$ with $dom(\theta_B) \cap vars(H\theta_{|_H}) = \emptyset$. By induction hypothesis, for each i, $\leftarrow A_i$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_i$. Hence $\leftarrow A_1, \ldots, A_n$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$. Hence also $(\leftarrow A_1, \ldots, A_n)\theta$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$. But $(\leftarrow A_1, \ldots, A_n)\theta = (\leftarrow B_1, \ldots, B_n)\theta = (\leftarrow B_1, \ldots, B_n)(\theta_{|_H} \cup \theta_B) = (\leftarrow B_1, \ldots, B_n)\theta_{|_H}\theta_B$. Let μ be the substitution accumulated in the refutation of $(B_1, \ldots, B_n)\theta_{|_H}\theta_B$. Note that $dom(\mu) \cap varsH\theta = \emptyset$ because $(dom(\theta_B) \cup dom(\mu)) \cap vars(H\theta) = \emptyset$ and hence also $dom(\theta_B\mu) \cap varsH\theta = \emptyset$. According to the lifting lemma [18], there exists a refutation for $(\leftarrow B_1, \ldots, B_n)\theta_{|_H}$ with accumulated substitution μ' such that $\theta_B\mu = \mu'\gamma$ for some γ. We have $dom(\mu') \subseteq dom(\theta_B\mu)$ hence $dom(\mu') \cap vars(H\theta_{|_H}) = \emptyset$; this holds also for the cas σ of the refutation with accumulated substitution μ'. $\square$

The above property relates the bottom-up derivation (A, τ) with an SLD-refutation (constructed using the left-to-right selection rule) for $\leftarrow A$ and provides as such the basis to relate the fixed point semantics with the operational semantics. Observe, however that the reverse of Proposition 1 does not necessarily hold. An atom A might have a refutation in P with an empty answer substitution, $\leftarrow A \overset{\epsilon}{\leadsto}_P \square$ while $\text{lfp}(T_P^d)$ does not contain A but only a generalisation of A. However, if we restrict our attention to atomic goals of the form $p(\overline{X})$, i.e. all arguments variables, the following does hold[1].

[1] Note that this restriction does not diminish the generality of the technique but eases the formulation of the results.

Lemma 1. *Let P be a definite program. If $\leftarrow p(\overline{X}) \overset{\theta}{\rightsquigarrow}_P \square$ with the characteristic derivation δ, then there exists a substitution σ and a labeled* AND*-tree τ such that $(p(\overline{X})\sigma, \tau) \in lfp(T_P^d)$, $\overline{\tau} = \delta$ and $p(\overline{X})\theta \approx p(\overline{X})\sigma$.*

Proof. We prove this by induction on the length of the SLD-refutation. First, consider a refutation $\leftarrow p(\overline{X}) \overset{\theta}{\rightsquigarrow}_P \square$ of length 1, with characteristic derivation the single label l. This means that there exist a renamed clause $p(\overline{X})\theta) \leftarrow\in P$, and consequently we have that a renaming of $(p(\overline{X})\theta, l) \in \mathrm{lfp}(T_P^d)$, i.e. $(p(\overline{X})\sigma, l) \in \mathrm{lfp}(T_P^d)$ with $p(\overline{X})\sigma \approx p(\overline{X})\theta$.

Next, assume that it holds for SLD-refutations of length k. Consider an SLD-refutation $\leftarrow p(\overline{X}) \overset{\theta}{\rightsquigarrow}_P \square$ of length $k+1$, with characteristic derivation $l, l_1, \ldots, l_k$. If the clause labelled l is of the form $H \overset{l}{\leftarrow} B_1, \ldots, B_n$, then the first step in the refutation is the derivation $\leftarrow p(\overline{X}) \overset{\sigma}{\rightsquigarrow}_P \leftarrow B_1, \ldots, B_n$ with $H = p(\overline{X})\sigma$. To ease the formulation of the proof, let us – without loosing generality – assume that $n = 2$; i.e. $H \overset{l}{\leftarrow} B_1, B_2$. Now, the remainder of the SLD-refutation of $p(\overline{X})$ can be split into the refutations $\leftarrow B_1 \overset{\theta_1}{\rightsquigarrow}_P \square$ and $\leftarrow B_2\theta_1 \overset{\theta_2}{\rightsquigarrow}_P \square$ with characteristic derivations $l_1, \ldots, l_s$ and $l_{s+1}, \ldots, l_k$ respectively. If we assume that $B_1 = q_1(\overline{Y})\gamma_1$, there also exists an SLD-refutation $\leftarrow q_1(\overline{Y}) \overset{\sigma_1}{\rightsquigarrow}_P \square$ such that $\theta_1 = mgu(B_1, q_1(\overline{Y})\sigma_1)$. Likewise, if we assume that $B_2 = q_2(\overline{Z})\gamma_2$, there also exists an SLD-refutation $\leftarrow q_2(\overline{Z}) \overset{\sigma_2}{\rightsquigarrow}_P \square$ such that $\theta_2 = mgu(B_2\theta_1, q_2(\overline{Z})\sigma_2)$. Although B_1 and B_2 may have variables in common, $dom(\theta_1) \cap dom(\theta_2) = \emptyset$: no variable can be bound by both θ_1 and θ_2 since $q_1(\overline{Y})\sigma_1$ and $q_2(\overline{Z})\sigma_2$ have no variables in common and if θ_1 binds a variable, every occurrence of that variable is bound in $B_2\theta_1$ and it cannot be in $dom(\theta_2)$. Consequently, we have that $\theta_1\theta_2 = mgu((B_1, B_2), (q_1(\overline{Y})\sigma_1, q_2(\overline{Z})\sigma_2))$ (1).

Now, by induction hypothesis, a renaming of $(q_1(\overline{Y})\sigma_1, \tau_1) \in \mathrm{lfp}(T_P^d)$ and $(q_2(\overline{Z})\sigma_2), \tau_2) \in \mathrm{lfp}(T_P^d)$ with τ_1 and τ_2 such that $\overline{\tau}_1 = l_1, \ldots, l_s$ and $\overline{\tau}_2 = l_{s+1}, \ldots, l_k$. By (1), it follows then that $(H\theta_1\theta_2, l(\tau_1, \tau_2)) \in \mathrm{lfp}(T_P^d)$. We have that $H\theta_1\theta_2 = p(\overline{X})\sigma\theta_1\theta_2 = p(\overline{X})\theta$. Moreover, $\overline{l(\tau_1, \tau_2)} = l, l_1, \ldots, l_s, l_{s+1}, \ldots, l_k$ which concludes the proof. □

Together, Proposition 1 and Lemma 1 extend a classic result (found e.g. in [12,5]) which essentially states the equivalence between the *set* of computed answers Θ for a goal $\leftarrow p(X_1, \ldots, X_n)$ and the atoms $\{p(X_1, \ldots, X_n)\theta \mid \theta \in \Theta\}$ from the least fixed point of the immediate consequences operator. The following corollary extends this result to regular atoms:

Corollary 1. *Let P be a definite program and A an atom. If $A \overset{\theta}{\rightsquigarrow}_P \square$ with characteristic derivation δ, then there exists an atom A' and a labelled* AND*-tree τ such that $(A', \tau) \in lfp(T_P^d)$, $\overline{\tau} = \delta$ and $A\theta = A'\gamma$ with $\gamma = mgu(A', A)$.*

The equivalence of the depth-first, left-to-right traversal of the AND-tree τ with the characteristic derivation of the atom allows to order the atoms in $\mathcal{F}_{p(\overline{X})}(P)$ lexicographically, such that the order between the atoms $p(\overline{t_1})$, $p(\overline{t_2})$,... reflects the order in which the corresponding computed answers $\{\overline{X}/\overline{t_1}\}$,

$\overline{X}/\overline{t_2}\}$, ... are returned for the goal $\leftarrow p(\overline{X})$. In what follows, we consider $(A, \tau_A) < (B, \tau_B)$ if and only if $\tau_A < \tau_B$. Note that, by construction, $<$ is a total order relation on the set $\text{lfp}(T_P^d)$. Given $\mathcal{F}_A(P)$, we denote with $\mathcal{F}_A^{\langle\rangle}(P)$ the sequence containing the bottom-up derivations from $\mathcal{F}_A(P)$ sorted using $<$.

Definition 6. *Let P be a program and A an atom. Then we define*

$$\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$$

where each $(A_i, \tau_i) \in \mathcal{F}_A(P)$ and for each (A_i, τ_i), $(A_j, \tau_j) \in \mathcal{F}_A(P)$: $j > i$ if and only if $(A_j, \tau_j) > (A_i, \tau_i)$.

The following theorem states that the operational semantics of a goal of the form $\leftarrow p(\overline{X})$ is modeled precisely by $\mathcal{F}^{\langle\rangle}_{p(\overline{X})}(P)$, obtained from the program's fixed point semantics $\text{lfp}(T_P^d)$.

Theorem 1. *Let P be a definite program and A an atom of the form $p(\overline{X})$. Assume that $\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots \rangle$. Then we have that for all i $A_i \approx A\sigma_i$.*

Again, we can generalise this result to regular atoms:

Corollary 2. *Let P be a definite program and A an atom. Assume that $\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots \rangle$. Then we have that for all i $A_i\theta_i \approx A\sigma_i$ where $\theta_i = mgu(A, A_i)$.*

4 A Bottom-Up Semantics for Pure Prolog Extended with Cut

In what follows, we extend the fixed point semantics defined in the previous section in order to deal with pure Prolog programs that possibly contain !/0 constructs. Programs that use the !/0 construct manipulate the SLD search process by literally cutting away part of the search space, i.e. avoiding the construction of a number of SLD-derivations. If we want to retain the equivalence between the operational and fixed point semantics for programs that contain cuts, we must ensure that $\mathcal{F}_A(P)$ does not contain any answers that would have been cut away during the SLD search process for $\leftarrow A$ in P. Let us first recall the semantics of !/0. Assume that during the construction of an SLD-derivation an atom A is unified with a clause $H \leftarrow B_1, \ldots, !, \ldots, B_n$ containing a !/0. The atom A is called the *parent atom* of the !/0 and the meaning of the !/0 is that it succeeds and discards all alternative derivations that may originate between (and including) the parent atom and the !/0 [18].

Example 4. Reconsider a slight variant of the `path` program where a !/0 has been added so that some solutions are prevented (cut away) as depicted in Fig. 2.

If we regard the operational semantics $\mathcal{O}(P, Q)$ as a set of refutations for Q, ordered by the lexicographic order relation $<$, dealing with programs containing

```
(1) path(X,Y):- edge(X,Y),!.
(2) path(X,Y):- edge(X,Z), path(Z,Y).
```

```
(3) edge(a,b).
(4) edge(a,c).
(5) edge(c,b).
```

Fig. 2.

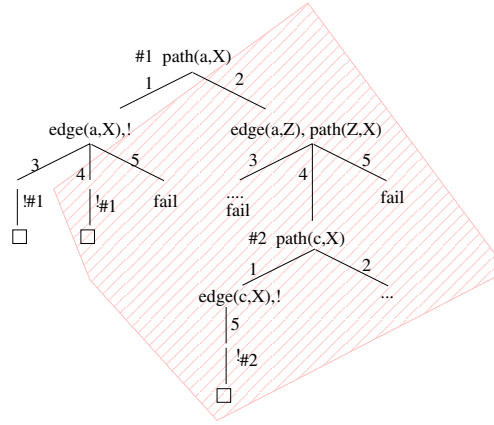


Fig. 3.

a cut resorts to removing from $\mathcal{O}(P, Q)$ those refutations that are "covered" by a cut operation. Informally, we can say that a refutation ρ is covered by a cut operation in a set of SLD-derivations S if:

- there is a derivation $\rho' \in S$ containing a !/0 such that $\rho' < \rho$, and
- ρ and ρ' are identical upto (and including) the parent atom, but differ somewhere between the parent atom and the selection of the !/0 operation in ρ'.

Formalising the above is non-trivial and requires keeping track of the precise relation between each selected atom and its "parent" atom in a single derivation, and between derivations that have a common subderivation. Indeed, dealing with programs containing a cut operation in the operational semantics requires dealing with the SLD-*tree* for a goal Q in P, rather than with the individual refutations for Q. For the example program of Fig. 2, an SLD-tree for the goal $\leftarrow path(a, X)$, constructed using the left-to-right selection rule, is depicted in Fig. 3. The shaded part represents the alternatives that are cut away by the !/0 in the leftmost branch.

Incorporating this behavior in the fixed point semantics boils down to removing those bottom-up derivations from $\mathcal{F}_A(P)$ that correspond with the answers that are removed when evaluating $\leftarrow A$ by SLD-resolution. In order to trace which bottom-up derivations must be removed from $\mathcal{F}_A(P)$, we first need to incorporate the presence of a !/0 construct into the bottom-up derivations. We treat an occurrence of !/0 in the body of a clause as an atom that is defined by a single clause, namely $! \stackrel{!}{\leftarrow} true$ that is labelled by the special label '!'. Consequently, occurrences of !/0 in the body of a clause resolve to true and the computed bottom-up derivations contain the term '!' as the label of a

(unit) clause. Note that when converting a bottom-up derivation into a characteristic derivation, the relation between an occurrence of !/0 and the label of its parent atom is immediate. Indeed, the cut in an AND-node of the form $l(\tau_1, \ldots, \tau_{k-1}, !, \tau_{k+1}, \ldots, \tau_n)$ refers to the choicepoint that is created by unifying the parent atom with the clause labelled l. We will see further how this observation enables an elegant formulation of the notion of coveredness that was introduced informally earlier.

Example 5. Let P denote the program in Figure 2. The bottom-up derivations constructed by T_P^d are similar to those of Example 3, except for the presence of cuts in the AND-trees:

$$\left\{ \begin{array}{ll} (path(a,b), 1(3,!)) & (path(a,c), 1(4,!)) \\ (path(c,b), 1(5,!)) & (path(a,b), 2(4, 1(5,!))) \end{array} \right\}$$

There is a catch, though. Neither the operational semantics, nor the fixed point semantics introduced before deal with finitely failing derivations. Still, failing derivations must be considered when dealing with programs containing !/0. Consider the following example:

Example 6. Let P be the following program

```
(1) r(X):- p(X), !, q(X).
(2) p(a).
(3) p(b).
(4) q(b).
```

Note that $\mathrm{lfp}(T_P^d) \ni \{(r(b), 1(3,!,4)\}$. However, $\mathcal{O}(P, \leftarrow r(X)) = \langle\rangle$ since $\leftarrow r(X)$ has a lexicographically smaller derivation that finitely fails but that contains a cut operation that removes the succeeding derivation with answer X/b.

In order to deal with finitely failing derivations in a satisfactory way, it suffices to make failure explicit in the program under consideration. Therefore, instead of computing $\mathrm{lfp}(T_P^d)$, we compute $\mathrm{lfp}(T_{P'}^d)$, where P' is the program that is obtained from P by adding an extra, last clause to every predicate that always succeeds and is labelled by the special label 'fail'.

Example 7. Let P be the program from Example 6, then P' is the following program:

```
(1) r(X):- p(X), !, q(X).          (fail) r(X).
(2) p(a).                          (fail) p(X).
(3) p(b).                          (fail) q(X).
(4) q(b).
```

Note that the introduction of the extra clauses does not alter the successful derivations of the program, where a successful derivation is a derivation that does not contain the label 'fail'[2]. Also note that the introduction of the two special labels – 'fail' and '!'– does not alter the definition of T_P^d, nor the definitions of a

[2] Note that the number of failing derivations that effectively needs to be constructed for the semantics can substantially be reduced by pruning during the computation of T_P^d.

bottom-up derivation and its depth-first, left to right traversal. Their presence does make, however, the lexicographic order relation no longer a complete order on the domain of AND-trees (and hence also on $\mathrm{lfp}(T_P^d)$) since the order between '!' or 'fail' and a natural number is undefined. For our purposes, it suffices to consider lexicographic ordering as a partial order, in which $fail > l$ holds for any label $l \neq fail$[3].

Example 8. Let P be the program of Example 7 and consider the goal $\leftarrow r(X)$. We have that

$$\mathcal{F}_{r(X)}(P) = \left\{ \begin{array}{ll} (r(a), 1(2, !, fail)) & (r(b), 1(fail, !, 4)) \\ (r(b), 1(3, !, 4)) & (r(A), 1(fail, !, fail)) \\ (r(b), 1(3, !, fail)) & (r(A), fail) \end{array} \right\}$$

In what follows, we need two important operations on an AND-tree. First, we need to be able to retrieve the i-th label that is encountered when traversing an AND-tree in depth-first, left to right order. Being the i-th element in the depth-first, left to right traversal of the AND-tree (denoted by $\overline{\tau}$), we denote this element by $\overline{\tau}^i$. Next, we need to be able to retrieve the subtree that is found in the i-th node of an AND-tree, when the tree is traversed in a depth-first, from left to right manner. We denote this subtree by $s_i(\tau)$. Formally:

Definition 7 (subtree). *Let τ be an* AND*-tree. We denote with $\#(\tau)$ the number of* AND*-nodes in τ. For any k such that $1 \leq k \leq \#(\tau)$, define s_k recursively as follows:*

$$s_k(l(\tau_1, \ldots, \tau_n)) = \left\{ \begin{array}{ll} l(\tau_1, \ldots, \tau_n) & \text{if } k = 1 \\ s_{k-1-\sum_{j:1}^{i-1} \#(\tau_j)}(\tau_i) & \text{if } k > 1 \end{array} \right.$$

where $i \in \mathbb{N}$ is such that $\sum_{j:1}^{i-1} \#(\tau_j) < k - 1 \leq \sum_{j:1}^{i} \#(\tau_j)$.

Example 9. Consider the AND-tree $\tau = 2(4, 1(5, !))$ taken from Example 5. We have that $s_1(\tau) = \tau$, $s_2(\tau) = 4$, $s_3(\tau) = 1(5, !)$, $s_4(\tau) = 5$, $s_5(\tau) =!$.

It remains to define what bottom-up derivations in $\mathcal{F}_A(P)$ are cut away by a cut contained in some other bottom-up derivation in $\mathcal{F}_A(P)$.

Definition 8 (coveredness). *Let S be a set of* AND*-trees. We say that an* AND*-tree τ is* covered *in S if and only if there exists $\tau' \in S$ and $k > 0$ such that*

- *$\forall i, 0 < i < k$, we have that $\overline{\tau}^i = \overline{\tau'}^i$, and*
- *$s_k(\tau') = l'(\tau'_1, \ldots, \tau'_n)$ with $\tau'_m =!$ for some m in the range $1, \ldots, n$, fail does not occur in $\tau'_1, \ldots, \tau'_{m-1}$ and either*
  - *$s_k(\tau) = l'(\tau_1, \ldots, \tau_n)$ and $\exists j : 1 \leq j < m$ such that $\tau'_j < \tau_j$ and $\tau_1 = \tau'_1, \ldots, \tau_{j-1} = \tau'_{j-1}$, or*

[3] The label '!' does not require special treatment: due to the construction of the order relation, a label '!' will only be compared against another label '!'.

- $s_k(\tau) = l(\tau_1, \ldots, \tau_q)$ *with* $l' < l$.

An AND-tree τ is covered in a set of AND-trees if the set contains a lexicographically smaller tree τ' containing a cut such that the depth-first left to right traversals of τ and τ' differ between the cut and the node it refers to. Note that formulating coveredness is easier on the level of the AND-trees than on the level of their depth-first left to right traversals, due to the explicit relation between a cut and its parent node in the notion of an AND-node.

Example 10. Let S denote the set of bottom up derivations depicted in Example 5. We have that $(path(a,c), 1(4,!))$ (denoted by τ_1) is covered in S due to the bottom-up derivations $\tau' = (path(a,b), 1(3,!))$ in S. Indeed, we have $s_1(\tau_1) = 1(4,!)$, $s_1(\tau') = 1(3,!)$ and $3 < 4$ (first case in Definition 8). Likewise, we have that $\tau_2 = (path(a,b), 2(4, 1(5,!)))$ is covered in S due to the same τ'. Indeed, we have that $s_1(\tau_2) = 2(4, 1(5,!)))$, $s_1(\tau') = 1(3,!)$ and $1 < 2$ (second case in Definition 8).

For an atom A, we define $\mathcal{C}_A$ as the subset of $\mathcal{F}_A(P)$ containing the derivations whose associated AND-tree is covered by an AND-tree in $\mathcal{F}_A(P)$. Formally:

Definition 9. *Let P be a definite program and A an atom. We define*

$$\mathcal{C}_A = \{(B,\tau) \mid (B,\tau) \in \mathcal{F}_A(P) \text{ and } \tau \text{ is covered in } \mathcal{T}\}$$

where $\mathcal{T} = \{\tau \mid (B,\tau) \in \mathcal{F}_A(P)\}$.

At last, we can refine the definition of the bottom-up semantics in order to incorporate programs that contain cut:

Definition 10. *Let P be a program and A an atom. Then we define*

$$\mathcal{F}^{\langle\rangle}_A(P) = \langle (A_1,\tau_1), (A_2,\tau_2), \ldots \rangle$$

where each $(A_i,\tau_i) \in (\mathcal{F}_A(P) \setminus \mathcal{C}_A(P))$ *and for each* (A_i,τ_i), $(A_j,\tau_j) \in \mathcal{F}_A(P)$: $j > i$ *if and only if* $(A_j,\tau_j) > (A_i,\tau_i)$.

Example 11. Let P denote the program depicted in Figure 2 extended with the clauses labelled 'fail'. We have that

$$\mathcal{F}_{path(X,Y)}(P) = \left\{ \begin{array}{ll} (path(a,b), 1(3,!)) & (path(X,Y), 1(fail,!)) \\ (path(a,c), 1(4,!)) & (path(a,Y), 2(3, fail)) \\ (path(c,b), 1(5,!)) & (path(a,Y), 2(4, fail)) \\ (path(a,b), 2(4, 1(5,!))) & (path(a,Y), 2(3, 1(fail,!))) \\ & \ldots \end{array} \right\}$$

and $\mathcal{C}_{path(X,Y)}$ contains all derivations from $\mathcal{F}_{path(X,Y)}(P)$ but $(path(a,b), 1(3,!))$. Consequently we have that

$$\mathcal{F}^{\langle\rangle}_{path(X,Y)}(P) = \langle (path(a,b), 1(3,!)) \rangle.$$

The following example shows the importance of constructing the finitely failing derivations:

Example 12. Let P be the program of Example 7 and consider the goal $\leftarrow r(X)$. $\mathcal{F}_{r(X)}(P)$ is depicted in Example 8. The only non-failing derivation in $\mathcal{F}_{r(X)}(P)$, $(r(b), 1(3, !, 4))$, is covered in $\mathcal{F}_{r(X)}(P)$ by the failing derivation $(r(a), 1(2, !, fail))$. Hence, we have that $\mathcal{F}^{\langle\rangle}_{r(X)}(P) = \langle\rangle$, corresponding with $\mathcal{O}(P, \leftarrow r(X))$.

Now, we can extend the result of Theorem 1 to programs that possibly contain cuts as follows.

Theorem 2. *Let P be a definite program that possibly contains cuts and A an atom of the form $p(\overline{X})$. Assume that $\mathcal{F}^{\langle\rangle}_{A}(P) = \langle(A_1, \tau_1), (A_2, \tau_2), \ldots\rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle\sigma_1, \sigma_2, \ldots\rangle$. Then we have that for all i $A_i \approx A\sigma_i$.*

Again, the theorem states the equivalence of the operational semantics and the fixed point semantics in the presence of cut operations and is extended to regular atoms by the following corollary.

Corollary 3. *Let P be a definite program that possibly contains cuts and let A be an atom. Assume that $\mathcal{F}^{\langle\rangle}_{A}(P) = \langle(A_1, \tau_1), (A_2, \tau_2), \ldots\rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle\sigma_1, \sigma_2, \ldots\rangle$. Then we have that for all i $A_i\theta_i \approx A\sigma_i$ where $\theta_i = mgu(A, A_i)$.*

5 Discussion

In this work, we have developed a fixed point semantics for pure Prolog programs with cut. We believe that our semantics is simple, elegant and can be the main building block for bottom-up program analyses and transformations that preserve the operational semantics of the transformed program. Our semantics captures the order in which answers are computed, their multiplicity, and deals with pure prolog programs extended with cuts. In the remainder of this section, we first discuss a possible application of our semantics and conclude by discussing related work.

5.1 An Application of the Semantics

A possible application of the semantics introduced in this paper is the construction of a program transformation that propagates information upwards in a program. Bottom-up propagation of information and the computation of success information has been considered before (e.g. in [19,15]) and has been advocated as a suitable technique for augmenting top-down partial deduction techniques (e.g. [17]). In previous work [28,27] we have developed a complete partial deduction scheme based on an abstraction of the immediate consequence operator. The resulting technique specialises a so-called "open" logic program (i.e. a program that calls a number of predicates that are left undefined) with respect to a definition of the missing predicates.

Such a technique has great potential for speeding up *inductive logic programming* (ILP) techniques in the fields of machine learning and data mining. An ILP problem comprises a background theory, represented by an open logic program, and a database of (positive and negative) examples where each such example is again represented by a logic program that consists of, among others, a particular definition of the predicates that are left undefined in the background knowledge. The task of the ILP system then is to constructed a hypothesis that "explains" the positive examples and rejects the negative ones. The main reason why an ILP application can profit from program specialisation is due to the background theory. Its role is to define a number of additional relations over the problem domain that provide, when combined with a particular example, extra "knowledge" within the context of the example[4]. However, the generality of the background theory can introduce a performance penalty since the search for the best hypothesis involves the evaluation of many queries over each example extended with the background theory. Hence, it makes sense to (partially) specialise the background theory and to use this specialisation for the many query evaluations. A concrete motivating example in the ILP context is the mutagenesis data set [25], where the goal is to learn a rule to predict the mutagenicity[5] of molecules. For this problem, some preliminary experiments that we conducted show that learning from a dataset that was obtained by fully specialising (using hand-crafted techniques) the background knowledge with respect to some examples results in an improvement in speed of a factor 40, while only introducing 15% extra code. Although the ILP setting seems a "traditional" setting suited for program specialisation, it has a number of characteristics that make it hard to apply standard out-of-the-box partial evaluation techniques on it. Unlike the usual approach to partial evaluation, in which the evaluation of a program is restricted to (instances of) a particular query, the first objective of specialising the background knowledge with respect to a particular example is essentially *query-independent*. Rather than propagating data provided in a query downwards in a program – performing as much as the unifications as possible underway – one aims at partially precomputing the relations in the background theory by propagating the information that resides in the predicate definitions that constitute a particular example upwards in the background theory.

A limitation of the bottom-up techniques from [28,27] is that they do not preserve multiplicity of answers, nor the order in which solutions are computed. In addition, they are not capable of dealing with programs that contain cuts. This is a major disadvantage if these (or other) techniques are to be applied to ILP problems that usually rely strongly on these operational characteristics of Prolog. In the remainder of this section, we sketch how the semantics that is presented in this work could be abstracted into a transformation that performs bottom-up partial deduction. A full formal development of the transformation is outside the scope of this paper.

[4] In fact, the ability to express background knowledge is one of the main advantages in expressivity that ILP offers over other, less expressive formalisms for data mining.

[5] Mutagenicity is the ability to mutate DNA, which is a possible cause of cancer.

In general, $\text{lfp}(T_P^d)$ will not be finite and one should endorse T_P^d with abstraction in order to compute a finite set of *partial* bottom-up derivations rather than an infinite set of complete bottom-up derivations. In previous work [28,27] we have shown that this can be achieved by redefining the immediate consequence operator over a domain of clauses, rather than atoms (like e.g. in [5,6]) and combining it with an abstraction function that replaces a clause $A \leftarrow \overline{B}$ that represents a partial derivation under construction with a tautology $H \leftarrow H$, where H is a generalisation of A. In this work, we take a slightly different approach. First, we extend the notion of a labelled *and*-tree such that its leaf nodes are either labelled AND-nodes with no children, denoted by $l()$ (as before) or atoms, denoted by (H). The idea is that, like a labelled AND-tree in a bottom-up derivation (A, τ) represents the computation that was performed to derive A, an atom in a leaf of τ represents a placeholder for some other computation that is required to compute A. We will use the notation $\mathit{leaves}(\tau)$ to denote the sequence of atoms that is obtained by traversing a labelled AND-tree depth first, from left to right. As such, the bottom-up derivation becomes a partial derivation and the answers computed by (A, τ) become dependent on the answers computed by the atoms in the leaves of τ. If no control information was to be exploited, the partial derivation could as well be represented by a clause $A \leftarrow \mathit{leaves}(\tau)$. The reason for keeping the atoms in τ rather than directly deriving a clause is in particular due to the handling of cuts, which we treat later on.

In what follows, we will denote the combination of an atom with such an extended AND-tree as a *clausal bottom-up derivation* and use $\mathcal{CBU}$ to denote the set of all such clausal bottom-up derivations. Likewise, we introduce $T_P^{cl} : \wp(\mathcal{CBU}) \mapsto \wp(\mathcal{CBU})$, whose definition equals the definition of T_P^d; the only difference being the domain $\mathcal{CBU}$ rather than $\mathcal{BU}$.

To convert the computation of a possibly infinite set of bottom-up derivations into the computation of a *finite* set of *partial* bottom-up derivations, we introduce the notion of an abstraction function. Such an abstraction function maps a set of clausal bottom-up derivations onto a set of clausal bottom-up derivations in the following way:

Definition 11. *An idempotent function* $f : \wp(\mathcal{CBU}) \mapsto \wp(\mathcal{CBU})$ *is an* abstraction function *if* $\forall S \in \wp(\mathcal{CBU})$ *it holds that if* $(H, \tau) \in S$*, then either* $(H, \tau) \in f(S)$ *or* $\exists (H', (H')) \in f(S)$ *such that* $H' \preceq H$*.*

Note that in Definition 11, $(H', (H'))$ represents a special bottom-up derivation consisting of an atom H' with an associated AND-tree that consists of a single (leaf) node labelled by the atom itself. The central idea behind abstracting a set of (clausal) bottom-up derivations is that a derivation (H, τ) can be dropped from the set if there exists a derivation $(H', (H'))$ in the set with H' a more general atom than H. When abstraction is combined with T_P^{cl}, this reflects the fact that the derivation (H, τ) is no longer to be extended from H, but a new bottom-up derivation is started from a more general atom H'.

The computation of $\text{lfp}(T_P^d)$ can then be abstracted by the computation of a finite sequence of sets of clausal bottom-up derivations $S_0, S_1, \ldots, S_n$ such that

$S_0 = \emptyset$ and for all i,

$$S_i = f(T_P^{cl}(S_{i-1}))$$

and $S_n = S_{n-1}$ with f being an abstraction function. It can be proven – under the appropriate conditions [28,27] – that, when the clauses from S_n are recombined with the clauses that were dropped during the subsequent applications of f in the computation of the sequence $S_0, S_1, \ldots, S_n$, one obtains a program P' for which the *declarative* semantics equals the declarative semantics of the original program, that is $\text{lfp}(T_{P'}) = \text{lfp}(T_P)$. If that is the case, we say that S_n is a *suitable* abstraction of $\mathcal{F}(P)$. In general, we will use $\mathcal{F}^{cl}(P)$ to denote a suitable abstraction of $\mathcal{F}(P)$.

The question that remains is how to convert the clausal bottom-up derivations in $\mathcal{F}^{cl}(P)$ into a residual program P' such that for a particular goal Q, $\mathcal{O}(P,Q) = \mathcal{O}(P',Q)$. First, note that the partial order relation defined over bottom-up derivations remains defined over the domain of clausal bottom-up derivations. Hence, for an atom A, we can define $\mathcal{F}_A^{cl}(P)$ in a similar way as $\mathcal{F}_A^{\langle\rangle}(P)$. Note however that, unlike $\mathcal{F}_A^{\langle\rangle}(P)$, $\mathcal{F}_A^{cl}(P)$ is not unique, in the sense that it depends on a particular abstraction function. Definition 6 can be refined into:

Definition 12. *Let P be a program, A an atom of the form $\leftarrow p(\overline{X})$ and $\mathcal{F}^{cl}(P)$ a suitable abstraction of $\mathcal{F}(P)$. We define*

$$\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), \ldots, (A_n, \tau_n) \rangle$$

where each $(A_i, \tau_i) \in \mathcal{F}^{cl}(P)$ and for each $(A_i, \tau_i), (A_j, \tau_j) \in \mathcal{F}^{cl}(P) : j > i$ if and only if $(A_j, \tau_j) > (A_i, \tau_i)$.

Now, if no abstraction was employed during the computation of $\mathcal{F}^{cl}(P)$, none of the leaves of the bottom-up derivations are atoms and each bottom-up derivation $(A', \tau) \in \mathcal{F}_A^{\langle\rangle}(P)$ effectively constitutes an answer for A. Consequently, one can generate a specialised program for A by removing those derivations that are covered by a cut as before and generating a unit clause for each remaining bottom-up derivation. If abstraction was employed, on the other hand, some of the bottom-up derivations contain atoms as their leaves, effectively representing a placeholder for a computation and one should generate plain clauses rather than unit clauses in the residual program. This, however, requires a different treatment of the cuts that are possibly present in the bottom-up derivations. Indeed, the presence of an atom in a bottom-up derivation can make the effect of a cut undecidable at transformation-time since the atom represents a placeholder for (a number of) computations of which the effect is unknown. Consequently, if the atom is in the scope of a cut, both the computation represented by the atom and the cut must be residualised in the transformed program such that the cut exhibits the desired behavior – depending on the residualised computations – in the transformed program.

5.2 Related Work

Operational and fixed point semantics of logic programs is a well-studied research field. In [4], an operational semantics is presented that models the meaning of a goal by a sequence of computed answers. The semantics handles programs with cut and contains a bottom element that represents infinite looping. The semantics characterises a program by functional equations and is mainly developed towards proving termination of logic programs. Also [22] defines an operational semantics by means of a sequence of answers and uses it to show that a number of transformation rules (unfold/fold) preserve this semantics. It does not deal with programs containing cuts. In [2], Prolog control is explicitly modeled in a constraint logic language and a declarative and operational semantics is presented. It encodes the Prolog search rule, but does not deal with programs containing cuts.

In [10], a semantics expressed in term of SLD-derivations is defined. The semantics of a program is goal-independent and various properties of SLD-derivations are studied. The main focus is on compositionality of the semantics, which does not deal with programs containing cuts. A semantics that is perhaps more related to our work is [11]. It defines a denotational semantics for Prolog programs that contain cuts. The meaning of a program is defined by a number of semantic functions on the syntactic constructs of a program that can be used to compute a goal-independent denotation of the program. The work is motivated mainly by the need to verify a number of transformations. The main difference with our work, is that we obtain a denotation of the program by applying a bottom-up immediate consequences operator, which seems to be more appropriate in our context since the main motivation of our work lies in abstracting the semantics to obtain a bottom-up partial evaluation scheme. Also closely related is [24]. It defines a general framework for goal-independent abstract semantics of Prolog that also models the depth-first search rule and the cut. The semantics consists of a sequence of pairs, of which one part denotes a computed answers while the other part – the so called "observability constraints" – give information about the cut executions. One of the main distinctions between our approach and the one of [24] is that in the latter work, the denotation consists of AND-OR trees, rather than AND-trees alone. Although this makes the technique interesting and applicable in a broad number of situations, the semantics also relies on (substantially complicated) tree operations to combine elements from the computed denotation. We believe that for our particular application – the creation of a 2-phase partial evaluator for Prolog – a semantics that is particularly targeted towards this goal might be preferable.

In [8], the authors present a framework for the abstract interpretation of Prolog that handles the depth-first search rule and cut. It models program execution by sequences of substitutions that represent the sequences of computed answer substitutions returned by a goal $\leftarrow p(\overline{X})$. Contrary to our approach, however, the framework is essentially an adaptation of top-down frameworks for abstract interpretation. Finally, the framework of [13] defines an operational abstract se-

mantics. It decorates abstract OLDT trees with extra control information, that is later on used by the !/0 operations to prune the OLDT tree.

Acknowledgements

We thank Mike Codish, Alberto Pettorossi and Marco Comini for stimulating discussions, and anonymous referees for their extensive and constructive comments on a draft of the current paper.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. Roberto Barbuti, Michael Codish, Roberto Giacobazzi, and Giorgio Levi. Modelling Prolog control. *Journal of Logic and Computation*, 3(6):579–603, December 1993.
3. Roberto Barbuti, Roberto Giacobazzi, and Giorgio Levi. A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. *ACM TOPLAS*, 15(1):133–181, January 1993.
4. Marianne Baudinet. Proving termination of Prolog programs: A semantic approach. *Journal of Logic Programming*, 14(1 & 2):1–29, 1992.
5. 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.
6. 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.
7. M. Bugliesi and F. Russo. Partial evaluation in Prolog: Some improvements about cut. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 645–660, Cleveland, Ohio, USA, 1989.
8. Baudouin Le Charlier, Sabina Rossi, and Pascal Van Hentenryck. Sequence-based abstract interpretation of Prolog. *Theory and Practice of Logic Programming*, 2(1):25–84, 2002.
9. M. Codish, D. Dams, and E. Yardeni. Bottom–Up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125, February 1994.
10. Marco Comini and Maria Chiara Meo. Compositionality properties of SLD-derivations. *Theoretical Computer Science*, 211(1 & 2):275–309, January 1999.
11. S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.
12. 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.
13. G. Filè and S. Rossi. Static analysis of Prolog with cut. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programmng and Automated Reasoning*, pages 134–145, 1993. Springer-Verlag.
14. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

15. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming H Languages, Implementations, Logics and Programs (PLILP'96)*, pages 137–151, Aachen, Germany, 1996. Springer-Verlag. LNCS 1140.
16. Michael Leuschel. Partial evaluation of the "real thing". In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'94*, Pisa, Italy, 1994.
17. Michael Leuschel. Program specialisation and abstract interpretation reconciled. In Joxan Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 220 – 234, Manchester, UK, 1998. MIT Press.
18. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
19. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. ALP, IEEE, The MIT Press.
20. R. A. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Proceedings of the International Symposium on Logic Programming*, pages 68–72. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.
21. S. Prestwich. An unfold rule for full Prolog. In Kung-Kiu Lau and Tim Clement, editors, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing, pages 199–213, London, July 2–3 1993. Springer Verlag.
22. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, 1991.
23. D. Sahlin. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing*, 12(1):7–51, 1993.
24. F. Spoto. Operational and goal-independent denotational semantics for Prolog with cut. *Journal of Logic Programming*, 42(1):1–46, 2000.
25. Ashwin Srinivasan, Stephen Muggleton, Michael J. E. Sternberg, and Ross D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.
26. M. H. van Emden and R. Kowalski. The Semantics of Predicate Logic as Programming Language. *Journal of ACM*, 23(4):733–743, 1976.
27. W. Vanhoof, D. De Schreye, and B. Martens. A framework for bottom up specialisation of logic programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the Joint International Symposia PLILP/ALP 1998*, volume 1490 of *Lecture Notes In Computer Science*, pages 54–72. Springer-Verlag, 1998.
28. W. Vanhoof, D. De Schreye, and B. Martens. Bottom-up partial deduction of logic programs. *The Journal of Functional and Logic Programming*, 1999:1–33, 1999.

Abstract Partial Deduction Challenged
(Summary)

Stefan Gruner

Declarative Systems and Software Engineering Group
Department of Electronics and Computer Science
University of Southampton - SO17 1BJ **(GB)**
sg@ecs.soton.ac.uk

"Every day try to falsify your favourite theory" (Karl Popper)

This short paper (poster) summarises my presentation [6] at this workshop [1] in September 2002[1]. It also reflects some of the discussions at (and some new insights since) the workshop, before this summary went into print.

After several years of Abstract Partial Deduction (APD) —which is, for reasons of simplicity, here just regarded as a joint technique combining *Partial Deduction* and *Abstract Interpretation*— at least two APD tool prototypes are available: These are SP [4] and ECCE [8] which both operate in the abstract domain of Regular Unary Logic (RUL) [13]. Both are implemented in PROLOG, and both transform PROLOG input to PROLOG output during an abstract partial deduction run. These two are the tools under consideration here. (The system CPE [12] is not considered because it doesn't support the language PROLOG.) Due to cooperation between the authors of [8][4], essential parts of the RUL processing source code of SP are reused and integrated into the source code of the ECCE software system. Despite of certain problems which these tools may still run into on particular input programs (especially nontermination? — see discussion in [10]), it had been conjectured in [9] that RUL-APD could produce interesting results in *Infinite State Systems Verification* (ISSV), especially when applied to systems described in terms of a process algebra (e.g. CSP). My experiments referred to in this summary were mainly motivated by that conjecture [9] (but also by the insights of [11]). Further I was interested into a direct performance comparison between the above mentioned RUL-APD tools in their currently available versions [8][4]. Finally, by choosing the well-known Bakery protocol [7] as an example which has already been successfully used as a test case for another logic-based approach to ISSV [2][3], the RUL-APD approach explored here can be roughly compared to that other (and partially successful) approach[2].

[1] I have safely kept my program source codes for those who wish to repeat or extend my experiments. Please contact me to obtain the according files which, due to lack of space, cannot be printed out into an appendix to this summary.

[2] Note though that [2] did not specify Lamport's original protocol. They analysed an over-simplified variant of it which is flawed by circularity: mutual exclusion is there achieved by means of mutual exclusion because their specification does not allow more than one involved process at the same time to read the registers of the other processes.

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 258–259, 2003.

In the context sketched above, my experiments have shown that in two variations of the Bakery system with parameters (variables) for the number of processes and the number of system steps, neither ECCE nor SP were able to detect the implicit system safety property by means of APD. In a parameterless CSP-wrapped variant with two "hard-wired" processes, however, the safety property could indeed be APD-detected[3]. Sufficient explanations of the reasons of those phenomena are still missing, but I regard this epistemical gap as a reasonable motivation for further research[4]. Latest work on APD with an abstract domain different from RUL already seems to indicate a promising direction [5].

References

1. F. Bueno & M. Leuschel (Eds.): *LOPSTR'02 PreProceedings of the International Workshop on Logic Based Program Development and Transformation* (This Workshop). Technical Report, Facultad de Informática, Universidad Politécnica de Madrid (E), September 2002.
2. F. Fioravanti & A. Pettorossi & M. Proietti: *Verification of Sets of Infinite State Processes using Program Transformation.* Proc. of LOPSTR'2001, LNCS 2372, Springer, 2002.
3. F. Fioravanti & A. Pettorossi & M. Proietti: *Combining Logic Programs and Monadic Second Order Logic by Program Transformation.* This Workshop [1], pp.166-181.
4. J. Gallagher: *SP System.* `http://www.cs.bris.ac.uk/~john/software.html`
5. J. Gallagher & J. Peralta: *Convex Hull Abstractions in Specialisation of CLP Programs.* This Workshop [1], pp.104-114.
6. S. Gruner: *Abstract Partial Deduction Challenged — Extended Abstract of Ongoing Work.* This Workshop [1], pp.251-258.
7. L. Lamport: *A New Solution to Dijkstra's Concurrent Programming Problem.* Communications of the ACM 17/8, pp.453-455, 1974.
8. M. Leuschel: *ECCE.* `http://www.ecs.soton.ac.uk/~mal/systems/ecce.html`
9. M. Leuschel & S. Gruner: *Abstract Conjunctive Partial Deduction using Regular Types and its Application to Model Checking.* Proc. of LOPSTR'2001, LNCS 2372, Springer, 2002.
10. P. Mildner: *Type Domains for Abstract Interpretation — A Critical Study.* Uppsala Theses in Comp. Sc. 31, ISBN 91-506-1345-6, Univ. Uppsala (S), 1999.
11. G. Snelting: *Paul Feyerabend und die Softwaretechnologie.* Informatik Spektrum 21/5, pp.273-276, Springer 1998.
12. G. Vidal: *Curry Partial Evaluator.* `http://www.dsic.upv.es/users/elp/peval/peval.html`
13. E. Yardeni & E. Shapiro: *A Type System for Logic Programs.* Journal of Logic Programming 10/2, pp.125-153, Elsevier, 1991.

[3] A significant difference between SP and ECCE could not be found in these experiments.

[4] In a later experiment (after this workshop) I found out that in the context of the test specification the tools could not detect the uniqueness of the minimum operation, thus the validity of $((S = S') \longrightarrow (min(S) = min(S'))$, which is a crucial preliminary of Lamport's original safety property proof.

Towards Correct Object-Oriented Design Frameworks in Computational Logic
(Extended Abstract)

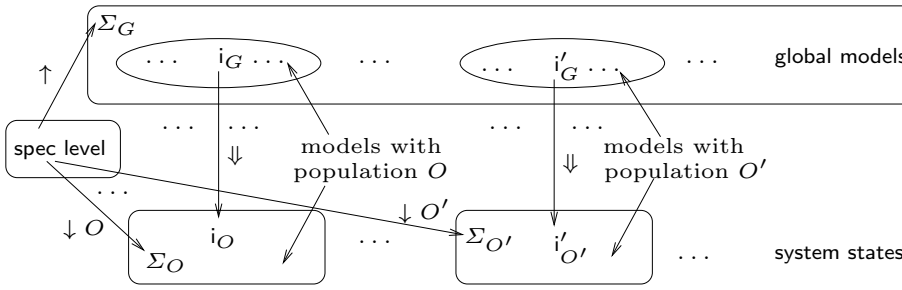
Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk
[2] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Object-Oriented Design (OOD) Frameworks are groups of interacting objects; they are increasingly recognised as better units of design and reuse than single objects (or classes), and are the focus of much attention in the evolution of current OO technology into Component-based Software Development (CBD) (see e.g. [1]).

For CBD, a framework should have a good component *interface* and should incorporate a notion of *a priori correctness*, namely *pre*-proved correctness of any given component operation with respect to its own specifications. Here, we will discuss a priori correctness of OOD frameworks, following on from [2] where we considered the correctness of static frameworks, i.e. ones without state transitions in its objects.

The formalisation of static OO frameworks, which we call *OO systems*, is based on correctness of non-OO frameworks which we call *specification frameworks*, which is in turn based on *steadfastness*, namely our a priori correctness for logic programs [3]. We define an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ as a set $C_1, \ldots, C_n$ of classes based on a specification framework $\mathcal{D}$, which codifies the problem domain and the necessary data types. To reason about OO systems, we introduce a three-level architecture, containing an *object*, a *specification* and a *global level*:



OO Systems are defined at the *specification level*, which defines the classes of the system, together with their *class attributes, constraints, specifications*

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 260–261, 2003.

and *methods*. The *object level* models distributed populations of collaborating objects. Each population O underlies a signature Σ_O and, by the $\downarrow O$ map, class constraints, specifications and methods map into Σ_O, and collectively give rise to a *distributed specification framework* $\mathcal{S}_O$. The models i_O of $\mathcal{S}_O$ represent the *system states* with population O. Finally, the *global level* considers a system as a whole; it has a global signature Σ_G and, by the $\uparrow$ map, class constraints, specifications and methods collectively map into a *global specification framework* $\mathcal{S}_G$, containing one *global program* P_G. The link between the three levels is given by the following *reflection property*:

$$\mathsf{i}_G \models F\uparrow \;\text{ iff }\; \mathsf{i}_G\Downarrow \models F \downarrow \mathsf{i}_G$$

By the $\Downarrow$-projection, each global model i_G represents a system state $\mathsf{i}_G\Downarrow$ with a population O, F is a formula of the specification level, $F\uparrow$ is its global representation, and $F \downarrow \mathsf{i}_G = F \downarrow O$ represents its meaning in the system state with the population O of i_G. As shown in [2], the global level works as a *meta-level* to reason about a priori correctness of the object distributed level, by reasoning about the steadfastness of P_G.

The multi-level approach can be extended to dynamic OO systems, by introducing time and actions. At the *object level*, we model observable behaviours as *observation sequences* $(S_0, a_0, S_1, a_1, S_2, \ldots)$, where, for $j \geq 0$, S_j is a (static) *system state* and S_j, a_j, S_{j+1} is a state-transition by the *action occurrence* a_j. At the *specification level*, we have a language where we can mention time, to specify class actions and properties of the observable behaviours. The *global level* works as a meta-level, and the reflection property holds, where now a global model i_G represents an observation sequence $\mathsf{i}_G\Downarrow$ of the object level, $F\uparrow$ represents the class formula F at the global level and $F \downarrow \mathsf{i}_G$ is a sequence of formulas representing the meaning of F over the observation sequence $\mathsf{i}_G\Downarrow$. The global level contains one global program P_G, open in the global predicate $occ(i, a)$ (meaning: action a has been observed at time i) and a global model i_G contains an interpretation j of occ. By reflection, we can reason about the properties of object level as corresponding properties of the j-models [3] of the open global program P_G.

The formalism is rather general and we believe that it is promising. It has been exploited for atomic actions, which cannot be interrupted, and our aim is to further develop and test it with some case studies.

References

1. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
2. K.-K. Lau and M. Ornaghi. Correct object-oriented systems in computational logic. In A. Pettorossi, editor, *Proc. LOPSTR 01, Lecture Notes in Computer Science 2372*, pages 168–190. Springer-Verlag, 2002.
3. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.

Mapping Modular SOS to Rewriting Logic

Christiano de O. Braga[1], E. Hermann Hæusler[2],
José Meseguer[3], and Peter D. Mosses[4]

[1] Departamento de Ciência da Computação, Universidade Federal Fluminense
[2] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro
[3] Computer Science Department, University of Illinois at Urbana-Champaign
[4] BRICS & Department of Computer Science, University of Aarhus

Abstract. Modular SOS ($MSOS$) is a framework created to improve the modularity of structural operational semantics specifications, a formalism frequently used in the fields of programming languages semantics and process algebras. With the objective of defining formal tools to support the execution and verification of $MSOS$ specifications, we have defined a mapping, named $\mathcal{M}to\mathcal{R}$, from $MSOS$ to rewriting logic (RWL), a logic which has been proposed as a logical and semantic framework. We have proven the correctness of $\mathcal{M}to\mathcal{R}$ and implemented it as a prototype, the MSOS-SL Interpreter, in the Maude system, a high-performance implementation of RWL. In this paper we characterize the $\mathcal{M}to\mathcal{R}$ mapping and the MSOS-SL Interpreter.
The reader is assumed to have some basic knowledge of structural operational semantics and object-oriented concepts.

1 Introduction

In this paper we present a formal mapping from modular structural operational semantics ($MSOS$) to rewriting logic (RWL), named $\mathcal{M}to\mathcal{R}$, its correctness proof, and its prototype implementation, the MSOS-SL Interpreter.

The intuitive idea underlying the $\mathcal{M}to\mathcal{R}$ mapping is that there is a very close connection between SOS transition rules and rewrite rules in rewriting logic: each SOS transition rule becomes a *conditional rewrite rule*, with the premises of the SOS transition rule translated into *conditions* of the rewrite rule, and the conclusion of the SOS transition rule translated into the rewrite rule itself. Furthermore, modularity in $MSOS$ transition rules is preserved by translating it as *inheritance* in object-oriented rewrite theories [8].

This work makes these intuitions precise by defining formally the $\mathcal{M}to\mathcal{R}$ map and proving its correctness. Using reflection the $\mathcal{M}to\mathcal{R}$ map can then be defined in an executable way in the Maude RWL interpreter, thus giving rise to the MSOS-SL Interpreter.

This paper is organized as follows. Section 2 briefly presents the $MSOS$ and RWL frameworks. Section 3 formally describes the $\mathcal{M}to\mathcal{R}$ mapping, showing the syntactical transformation from $MSOS$ specifications to RWL rewrite theories and the semantic mapping from arrow-labeled transition systems ($ALTS$), the models of $MSOS$ specifications, to $\mathcal{R}$-systems, the models of rewrite theories. Section 4 presents the proof

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 262–277, 2003.

of correctness of $\mathcal{M}to\mathcal{R}$, showing that computations are preserved in the model of the generated rewrite theory. Section 5 briefly describes the MSOS-SL Interpreter, our prototype implementation of $\mathcal{M}to\mathcal{R}$. We conclude this paper in Section 6 with some final remarks and future work.

2 Modular SOS and Rewriting Logic

This section formally presents the $MSOS$ and RWL frameworks. Section 2.1 begins with a motivation for $MSOS$ giving a small example for the modularity problem in SOS and then formalizes the concepts of a $MSOS$ specification and an arrow-labeled transition system, the model for a $MSOS$ specification. Section 2.2 gives very concise definitions for rewrite theories in rewriting logic and their models, viewed as transition systems. These two definitions are used in the formal characterization of the $\mathcal{M}to\mathcal{R}$ mapping in Section 3 and in the correctness proof of $\mathcal{M}to\mathcal{R}$ in Section 4, respectively.

2.1 Modular Structural Operational Semantics

Modularity in SOS specifications was left as an open problem by Plotkin in [14]. To illustrate the SOS modularity problem, let us consider the following SOS rules for the gradual evaluation of expressions e to their values v, where an expression is an addition of expressions or a value, and values are natural numbers.

Example 1 (Addition of natural numbers in SOS).

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \qquad \frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1 + e_2 \rightarrow v_1 + e_2'} \tag{1}$$

If one now considers a functional extension, the existing rules would have to be modified in order to support the notion of environments, besides the addition of the rule for function application. In the original formulation of the rules, the transition relation only involved expressions as configurations. Now it involves both expressions and an environment, so all the rules have to be reformulated.

A further imperative extension would cause more modifications, adding the notion of stores. The transition relation now involves the syntax, the environment, and the store.

Extensions with (interactive) input-output would require the use of labeled transitions, and entail yet another reformulation of the above rules. We refer to [5, 12] for the complete presentation of Example 1 and a second example describing an extension to the ML language with concurrency primitives, respectively.

To improve the modularity of SOS specifications, and in particular to avoid the need for reformulations, $MSOS$ restricts configurations to (abstract) syntax, and requires semantic structures (such as environments and stores) to be incorporated in labels on transitions.

In categorical terms, the labels can be seen as the arrows of a category where the semantic states are the objects. For this reason such a transition system is called an arrow-labeled transition system ($ALTS$).

Definition 1 (Arrow-labeled transition systems). *An arrow-labeled transition system or $ALTS$ is a labeled transition system where its set of labels is the set of the* arrows *of a category $\mathbb{A}$, represented as $Morph(\mathbb{A})$. The objects of $\mathbb{A}$, written $Obj(\mathbb{A})$, are the states of the $ALTS$. The source $\mathbf{pre}(\alpha)$ of a label α on a transition, together with the configuration "before" the transition, represents the state of the computation before the transition, and its target $\mathbf{post}(\alpha)$, together with the configuration "after" the transition, represents the state afterward. The composition of arrows α_1 and α_2 is defined iff $\mathbf{post}(\alpha_1) = \mathbf{pre}(\alpha_2)$. The identity arrows of $\mathbb{A}$ are written $\mathbb{I}^{\mathbb{A}}$, or simply $\mathbb{I}$ when $\mathbb{A}$ is evident; they are called* silent *or* unobservable *arrows and are used to label transitions that merely reduce configurations. For two transitions $\gamma \xrightarrow{\alpha_1} \gamma'$, $\gamma'' \xrightarrow{\alpha_2} \gamma'''$ to be adjacent, two conditions must be met, namely $\gamma' = \gamma''$, and $\mathbf{post}(\alpha_1) = \mathbf{pre}(\alpha_2)$. Terminating or infinite sequences of adjacent transitions then define the* computations *of the $ALTS$.*

In the context of the $MSOS$ specification of a programming language semantics, a label represents the "semantic" state of a program, that is, the information associated, for example, with the store or with the environment. A label captures the semantic state both before and after a transition, possibly with some more information associated with the transition itself (such as the usual synchronization signals of CCS [10]).

Label categories are built using the so called *label transformers*. Label transformers assume that the label category $\mathbb{A}$ comes equipped with additional structure, such that the arrows in $\mathbb{A}$ have components. Each component is accessed by using a particular index in a set $Index$ of indices, which is mapped to a value in the set $Univ$ of information structures. That is, it is assumed that there are sets $Index$, of indices, and $Univ$, of information structures, and functions $\mathbf{set} : Morph(\mathbb{A}) \times Index \times Univ \to Morph(\mathbb{A})$ and $\mathbf{get} : Morph(\mathbb{A}) \times Index \to Univ$, where the intuitive meaning is that $\mathbf{set}(\alpha, i, u)$ changes the component of $\alpha \in Morph(\mathbb{A})$ indexed by i to u, leaving the rest unchanged; and $\mathbf{get}(\alpha, i)$ yields the component of α indexed by i. Thus, labels might be seen as *mappings* from indices to $Univ$.

A label transformer T involves adding a fresh new index, say i, to the original set $Index$, and transforms the original category of labels $\mathbb{A}$ to the product category $T(\mathbb{A}) = \mathbb{A} \times \mathbb{C}_T$. The original $\mathbf{set}$ and $\mathbf{get}$ functions are extended in the obvious way, that is, by treating the arrows $\beta \in \mathbb{C}_T$ as the information stored in the new index i.

There are three such label transformers that can be used as basic building blocks to extend the label categories of $MSOS$ specifications. They correspond to the following three basic choices of $\mathbb{C}_T$:

- A discrete category E, typically understood as a set of environments, giving rise to the label transformer ***ContextInfo***(i, E).
- A cartesian product S^2, understood as a binary relation, and viewed as a category with set of objects S and arrow composition given by relational composition. S is typically understood as a set of stores. This gives rise to the label transformer ***MutableInfo***(i, S). The functions $\mathbf{get\_pre}$ and $\mathbf{set\_post}$ are defined to access the first, respectively change the second, component of the pair.
- A monoid (A, f, τ), viewed as a one-object category, and is typically understood as a set of actions. The ***EmittedInfo***(i, A, f, τ) label transformer arises with this choice of $\mathbb{C}_T$.

Definition 2 (*MSOS* **specifications).** *A specification in modular structural operational semantics (MSOS) is a structure of the form* $\mathcal{M} = \langle \Sigma, Lc, Tr \rangle$. *The* Σ *component is the* signature *of* $\mathcal{M}$*, defined as a set of operation signatures characterizing the language defined by* $\mathcal{M}$. *The* Lc *component is a* label category declaration*, named* Lc*, and is defined as an application of a composition of any combination of the three label transformers to a label category. The component* Tr *is the set of* transition rules *of* $\mathcal{M}$*, represented as inference rules*[1].

Example 2 (MSOS for the language in Example 1). The labeled transition rules of the *MSOS* specification for the language in Example 1 considering the functional and imperative extensions that we have described are as follows:

$$\frac{n = n_1 + n_2}{n_1 \ + \ n_2 \xrightarrow{\ \iota\ } n} \qquad \frac{e_1 \xrightarrow{\ \alpha\ } e_1'}{e_1 \ + \ e_2 \xrightarrow{\ \alpha\ } e_1' \ + \ e_2} \qquad \frac{e_2 \xrightarrow{\ \alpha\ } e_2'}{v_1 \ + \ e_2 \xrightarrow{\ \alpha\ } v_1 \ + \ e_2'} \tag{2}$$

$$\frac{e_1 \xrightarrow{\ \alpha\ } e_1'}{e_1 \ e_2 \xrightarrow{\ \alpha\ } e_1' \ e_2} \qquad \frac{e_2 \xrightarrow{\ \alpha\ } e_2'}{v_1 \ e_2 \xrightarrow{\ \alpha\ } v_1 \ e_2'} \tag{3}$$

$$\frac{\alpha' = \mathbf{set}(\alpha, env, \mathbf{get}(\alpha, env)[x \mapsto v]) \quad e \xrightarrow{\ \alpha'\ } e'}{\lambda x \, (e) \ v \xrightarrow{\ \alpha\ } e'} \tag{4}$$

$$\frac{\mathbf{get}(\iota, env)(x) = v}{x \xrightarrow{\ \iota\ } v} \tag{5}$$

$$\frac{\alpha = \mathbf{set\_post}(\iota, sto, \mathbf{get\_pre}(\iota, sto)[l \mapsto v])}{l \ := v \xrightarrow{\ \alpha\ } ()} \tag{6}$$

$$\frac{\mathbf{get\_pre}(\iota, sto)(l) = v}{l \xrightarrow{\ \iota\ } v} \tag{7}$$

The label category may be defined by the following nested application of label transformers:

$$\textbf{\textit{MutableInfo}}(sto, Store)(\textbf{\textit{ContextInfo}}(env, Env)(\mathbb{1})). \tag{8}$$

That is, $Index = \{env, sto\}$ corresponding to adding two new components to the trivial category $\mathbb{1}$ (consisting of just one object and one arrow), namely the arrows and the objects of the categories formed from the sets *Env* and *Store* by the label applied transformers.

Note that the above rules did not require any change to accommodate the functional and imperative extensions, and that they would not have to be changed at all in a further extension. To allow concurrency and message-passing, for instance, as in primitives [12], one would add a new semantic structure to the labels. The new rules would **set** and **get** data to and from the new index involved in the extension, but the previous rules wouldn't have to be changed.

[1] We refer to [4, Definition 4, page 6] for the grammar definition of the transition rules in an *MSOS* specification.

2.2 Rewriting Logic

Let us now present the rewriting logic framework by defining the concepts of rewrite theories (specifications in RWL) and $\mathcal{R}$-systems, the models for rewrite theories. We refer to [9] for a complete presentation of the rewriting logic framework.

Definition 3 (Rewrite theory). *A rewrite theory $\mathcal{R}$ is a 4-tuple $\mathcal{R} = \langle \Sigma, E, L, R \rangle$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of labels, and R is a set of pairs $R \subseteq L \times (T_{\Sigma,E}(X)^2)^+$ whose first component is a label and whose second component is a nonempty sequence of pairs of the E-equivalence class of terms with $X = \{x_1, \ldots x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called* rewrite rules.

Due to space limitations, let us briefly explain the syntax for object-oriented rewrite theories in rewriting logic necessary to follow the examples. A *class* is declared in a rewrite theory using the syntax *class* ⟨class-name⟩ | ⟨attribute-name⟩ : ⟨attribute-type⟩, ... ⟨attribute-name ⟩ : ⟨attribute-type⟩. where the non-terminals class-name, attribute-name, and attribute-type are strings representing what their names imply. In a object-oriented rewrite theory an *object* is a term of sort *Object* structured according to the class declaration of the object's class-name. Object-oriented theories are actually "syntactic-sugar" for rewrite theories. Roughly speaking class declarations are mapped to sort declarations and inheritance is represented via the subsorting relation. We refer to [6] for a detailed presentation of the translation from object-oriented theories to rewrite theories.

A (conditional) rewrite rule is declared with the concrete syntax crl : ⟨left-hand-side-pattern⟩ ⇒ ⟨right-hand-side-pattern⟩ *if* ⟨condition-set⟩. The non-terminals left-hand-side-pattern and right-hand-side-pattern are nonempty sequences of the E-equivalence class of terms with variables. Roughly speaking, the condition set is a conjunction of predicates or rewrites.

Operations are defined using the keyword *op* and equations with the *eq* operator. Variables are declared using the keywords *var* or *vars*.

The paper [9] gives a precise definition for $\mathcal{R}$-systems in categorical terms. Instead, in Section 4 we use the computational view of $\mathcal{R}$-systems as transition systems to prove the correctness of $\mathcal{M}to\mathcal{R}$, the mapping from $MSOS$ to RWL. Under the computational view the objects of a $\mathcal{R}$-systems $\mathcal{S}$ are the states of the transition system $\mathcal{TS}$ that models a rewrite theory $\mathcal{R}$ and the arrows of $\mathcal{S}$ are the transitions of $\mathcal{TS}$.

A very important characteristic its logic is *reflective* [1,3], in a precise mathematical way. There is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that it can be represented in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, which is a *meta* representation of the rewrite theory $\mathcal{R}$; any terms t and t' in $\mathcal{R}$ as terms $\overline{t}$, $\overline{t'}$, which are the meta representations of terms t and t' respectively; and any pair $(\mathcal{R}, t)$ as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that the following equivalence holds

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle. \tag{9}$$

Since $\mathcal{U}$ is representable in itself, a "reflective tower" can be achieved with an arbitrary number of levels of reflection, since the following equivalences hold.

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

Reflection in rewriting logic gives the formal basis for the transformation process in $\mathcal{M}to\mathcal{R}$, since the transformation process is defined as a meta-function in rewriting logic, as explained in Section 3. This then allows a very direct reflective implementation of the MSOS-SL Interpreter in Maude.

3 $\mathcal{M}to\mathcal{R}$: Mapping $MSOS$ to RWL

This section formally presents the mapping from modular structural operational semantics ($MSOS$) to rewriting logic (RWL).

Section 3.1 gives a formal definition for the *syntactical* transformation from $MSOS$ specifications to RWL rewrite theories, represented by the function $\mathcal{M}to\mathcal{R} : \mathcal{M} \rightarrow \mathcal{R}$, and some auxiliary definitions. Section 3.2 formalizes the *semantic* mapping from arrow-labeled transition systems to $\mathcal{R}$-systems via the mapping from $ALTS$ to transition systems (represented by the mapping $\mathcal{A}to\mathcal{TS} : \mathcal{A} \rightarrow \mathcal{TS}$), and the correspondence between transition systems and $\mathcal{R}$-systems in [9].

3.1 From $MSOS$ Specifications to Rewrite Theories

In order to define a mapping from one language to another, one has to relate their syntaxes, and semantic models. In this section we define how $MSOS$ specifications are transformed into rewrite theories. Actually we use *object-oriented* rewrite theories [6, 8], which provide some auxiliary syntactic sugar to specify classes and objects.

The main idea of the mapping, which is reflected both at syntactic and semantic levels, is to expose the information encapsulated in the labels of the label category into the configurations of rules, at the syntactic level, and to the computations states, at the semantic level.

At the syntactic level, the transformation of an $MSOS$ specification $\mathcal{M}$ maps the label category declaration of $\mathcal{M}$ into a class declaration in a rewrite theory $\mathcal{R}$, and the transition rules of $\mathcal{M}$ into rewrite rules in $\mathcal{R}$, such that the label information in $MSOS$ transition rules is moved to the configurations of the RWL rewrite rules. The label variables and operations on labels on the premises of the transition rules of $\mathcal{M}$, are transformed into variables of sort *Object* and operations on objects, respectively, in $\mathcal{R}$.

The transformation for $MSOS$ specifications is given in Definition 4. Note that the function $\mathcal{M}to\mathcal{R}$ is overloaded for each component in an $MSOS$ specification, returning its counterpart in a rewrite theory.

Definition 4 (Transformation for $MSOS$ specifications). *The transformation function $\mathcal{M}to\mathcal{R} : \mathcal{M} \rightarrow \mathcal{R}$, transforms an $MSOS$ specification $\mathcal{M} = \langle \Sigma, Lc, Tr \rangle$ into a rewrite theory $\mathcal{R} = \langle \Sigma, R, E \rangle$[2] in the following way:*

- *The label category declaration Lc in $\mathcal{M}$ is transformed into E, the equational part of the rewrite theory $\mathcal{R}$ in the form of a class declaration named* `Lc`*, together with* $\mathbf{get}$-i *and* $\mathbf{set}$-i *operations on objects for each index i in Lc, with the expected*

[2] Labels in $\mathcal{R}$ are abstracted to simplify the presentation.

behavior. (The operation $\mathbf{set}$-i *updates the value mapped to* i *with a given value, keeping the remaining attributes unchanged, and the operation* $\mathbf{get}$-i *yields the value mapped to the attribute* i*.) Each index* i *in* Lc *become an attribute in* `Lc`*, typed according to the label transformer declaration that declared* i *in* Lc*, in the following way:*

- *For an index* i *declared in* Lc *via a* ***ContextInfo***(i, T) *declaration, the type of the values mapped to* i *in* `Lc` *is* T*.*
- *For an index* i *declared in* Lc *via a* ***MutableInfo***(i, T) *declaration, the type of the values mapped to* i *in* `Lc` *is* T*.*
- *For an index* i *declared in* $\mathcal{M}$ *via an* ***EmittedInfo***$(i, T^*, \cdot, e)$*, the type of the values mapped to* i *in* `Lc` *is* $List[T^*]$*.*[3]

– *Each transition rule* tr *in* $\mathcal{M}$*, is transformed into a rewrite rule* r *in* $\mathcal{R}$ *with left-hand side formed by a pair of the respective* tr *configuration and the pre label component. The right-hand side pattern of* r *is built analogously. We refer to [4, Sections 3.1 and 6.3.3] for a complete description of the transformation of the label operations. Allow* α *to be the label of the transition being specified, and* β_1 *to* β_n *label variables;* σ*,* σ'*,* γ_1 *to* γ_n*,* γ'_1 *to* γ'_n *to be non-empty sequences of* Σ*-terms with variables;* $\alpha_{\mathbf{pre}}$*,* $\alpha_{\mathbf{post}}$*,* ${\beta_1}_{\mathbf{pre}}$ *to* ${\beta_n}_{\mathbf{pre}}$ *and* ${\beta_1}_{\mathbf{post}}$ *to* ${\beta_n}_{\mathbf{post}}$ *to be variables of type* Object*.*

$$\mathcal{M}to\mathcal{R}\left(\frac{\gamma_1 \xrightarrow{\beta_1} \gamma'_1 \wedge \gamma_2 \xrightarrow{\beta_2} \gamma'_2 \ldots \wedge \gamma_n \xrightarrow{\beta_n} \gamma'_n}{\sigma \xrightarrow{\alpha} \sigma'}\right) = \tag{10}$$

$$crl : (\sigma, \alpha_{\mathbf{pre}}) \Rightarrow (\sigma', \alpha_{\mathbf{post}})$$
$$\textit{if } (\gamma_1, {\beta_1}_{\mathbf{pre}}) \Rightarrow (\gamma'_1, {\beta_1}_{\mathbf{post}}) \wedge \ \ldots \ \wedge (\gamma_n, {\beta_n}_{\mathbf{pre}}) \Rightarrow (\gamma'_n, {\beta_n}_{\mathbf{post}}) \ .$$

Premises that do not care about labels remain untouched. Identity-labeled transition rules are transformed into rewrite rules as described above together with the assertion that the associated $\mathbf{pre}$ *and* $\mathbf{post}$ *variables of sort* Object *have to be equal.*

Example 3 (The rewrite theory for Example 2).

The transformation of the label category yields a class declaration with two attributes, namely *sto* of type *Store* and *env* of type *Env*. In this example we will name the class *Example 2*, but in MSOS-SL (Section 5) a label declaration has an associated name which becomes the class name in the resulting rewrite theory.

$$\mathcal{M}to\mathcal{R}(\textbf{\textit{MutableInfo}}(sto, Store)(\textbf{\textit{ContextInfo}}(env, Env)(\mathbb{1}))) = \tag{11}$$
$$class\ Example2 \mid sto : Store, env : Env \ .$$

The transformation of the label category declaration also generates set and get operation declarations and equations to *env* and *sto* object attributes, instances of class *Example2* and the appropriate variable declarations.

[3] The reason why the type of the values mapped to ***EmittedInfo***-declared indices are mapped to lists in the rewrite theory is because it is necessary to keep the "history" of the emitted information in a computation, due to the mapping of the label category to a *preorder*. This is a semantic issue which will become clearer in Definition 5.

The declaration for the label variable α gives rise to the declarations of the (mathematical) variables of sort *Object* $\alpha_{\mathbf{pre}}$ and $\alpha_{\mathbf{post}}$.

The first rule in (2), which specifies a transition with an identity label, is transformed into a conditional rewrite rule as follows.

$$\mathcal{M}to\mathcal{R}\left(\frac{n = n_1 + n_2}{n_1 \ + \ n_2 \xrightarrow{\iota} n}\right) = \tag{12}$$
$$crl : (n_1 \ + \ n_2, \iota_{\mathbf{pre}}) \Rightarrow (n, \iota_{\mathbf{post}}) \ \mathit{if} \ n = n_1 + n_2 \ \wedge \ \iota_{\mathbf{pre}} = \iota_{\mathbf{post}} \ .$$

The second rule in (2) is transformed into a conditional rewrite rule as follows.

$$\mathcal{M}to\mathcal{R}\left(\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \ + \ e_2 \xrightarrow{\alpha} e_1' \ + \ e_2}\right) = \tag{13}$$
$$crl : (e_1 \ + \ e_2, \alpha_{\mathbf{pre}}) \Rightarrow (e_1' \ + \ e_2, \alpha_{\mathbf{post}}) \ \mathit{if} \ (e_1, \alpha_{\mathbf{pre}}) \Rightarrow (e_1', \alpha_{\mathbf{post}}) \ .$$

The third rule in (2) together with the rules in (3) have a similar transformation.

The transformation for the rule in (4), which specifies the application of a lambda abstraction, generates the following conditional rewrite rule.

$$\mathcal{M}to\mathcal{R}\left(\frac{\alpha' = \mathbf{set}(\alpha, env, \mathbf{get}(\alpha, env)[x \mapsto v]) \quad e \xrightarrow{\alpha'} e'}{\lambda x \, (e) \ v \xrightarrow{\alpha} e'}\right) = \tag{14}$$
$$\begin{array}{l} crl : (\lambda x \, (e) \ v, \alpha_{\mathbf{pre}}) \Rightarrow (e', \alpha_{\mathbf{post}}) \ \mathit{if} \\ \alpha'_{\mathbf{pre}} = \mathbf{set\text{-}env}(\alpha_{\mathbf{pre}}, \mathbf{get\text{-}env}(\alpha_{\mathbf{pre}})[x \mapsto v]) \ \wedge \\ (e, \alpha'_{\mathbf{pre}}) \Rightarrow (e', \alpha'_{\mathbf{post}}) \ \wedge \\ \alpha_{\mathbf{post}} = \mathbf{set\text{-}env}(\alpha'_{\mathbf{post}}, \mathbf{get\text{-}env}(\alpha_{\mathbf{pre}})[x \mapsto v]) \ . \end{array}$$

The rule in (5) specifies access to a value bound to a variable in the environment. It is transformed into the following conditional rewrite rule.

$$\mathcal{M}to\mathcal{R}\left(\frac{\mathbf{get}(\iota, env)(x) = v}{x \xrightarrow{\iota} v}\right) = \tag{15}$$
$$crl : (x, \iota_{\mathbf{pre}}) \Rightarrow (v, \iota_{\mathbf{post}}) \ \mathit{if} \ \iota_{\mathbf{pre}} = \iota_{\mathbf{post}} \ \wedge \ v = \mathbf{get\text{-}env}(\iota_{\mathbf{pre}})(x) \ .$$

The rule in (6) specifies assignment of a value to a cell mapped to a variable in the storage. It is transformed into the following rewrite rule.

$$\mathcal{M}to\mathcal{R}\left(\frac{\alpha = \mathbf{set\_post}(\iota, sto, \mathbf{get\_pre}(\iota, sto)[l \mapsto v])}{l \ := v \xrightarrow{\alpha} \ ()}\right) = \tag{16}$$
$$\begin{array}{l} crl : (l \ := v, \alpha_{\mathbf{pre}}) \Rightarrow (\,() \,, \alpha_{\mathbf{post}}) \ \mathit{if} \\ \iota_{\mathbf{pre}} = \alpha_{\mathbf{pre}} \ \wedge \ \iota_{\mathbf{pre}} = \iota_{\mathbf{post}} \ \wedge \\ \alpha_{\mathbf{post}} = \mathbf{set\text{-}sto}(\iota_{\mathbf{pre}}, \mathbf{get\text{-}sto}(\iota_{\mathbf{pre}})[l \mapsto v]) \ . \end{array}$$

Finally, the rule in (7) specifies how to access a value stored in a cell mapped to a variable in the storage. It is transformed into the following rewrite rule.

$$\mathcal{M}to\mathcal{R}\left(\frac{\mathbf{get\_pre}(\iota, sto)(l) = v}{l \xrightarrow{\iota} v}\right) = \tag{17}$$
$$crl : (l, \iota_{\mathbf{pre}}) \Rightarrow (v, \iota_{\mathbf{post}}) \; if \; \iota_{\mathbf{pre}} = \iota_{\mathbf{post}} \; \wedge \; v = \mathbf{get\text{-}sto}(\iota_{\mathbf{pre}})(l) \; .$$

3.2 The Mapping from $ALTS$ to $\mathcal{R}$-systems

The de-encapsulation of the information in a label in the mapping from $MSOS$ to RWL is represented at the semantic level by the mapping $\mathcal{A}to\mathcal{TS}$ from arrow-labeled transition systems to (unlabeled) transition systems. The label information is moved from the labels in the transitions of an $ALTS$ $\mathcal{A}$ to the configurations of the transitions of the transition system $\mathcal{TS}(\mathcal{A})$, mapped from $\mathcal{A}$. The mapping $\mathcal{A}to\mathcal{TS}$ accomplishes that by representing the label category $\mathbb{A}_{\mathcal{A}}$ as a *preorder*, $Pre(\mathbb{A})$, and then moving the elements of $Pre(\mathbb{A})$ into the configurations of $\mathcal{TS}(\mathcal{A})$, defined as pairs composed of elements of the set of configurations of $\mathcal{A}$ and $Pre(\mathbb{A})$. Note that the $Pre(\mathbb{A})$ have the same set $Index$ as $\mathbb{A}$. Moreover, each element in $Pre(\mathbb{A})$ has the same indices as its counterpart in $\mathbb{A}$.

To describe how a label category $\mathbb{A}$ is mapped into its preorder view $Pre(\mathbb{A})$ it is necessary to describe how values mapped to indices in a label $\alpha \in \mathbb{A}$ are mapped to values related to indices in elements of $Pre(\mathbb{A})$. The mapping of indices in the labels of $\mathbb{A}_{\mathcal{A}}$ built via the application of ***ContextInfo*** and ***MutableInfo*** label transformers is rather straightforward. For these two cases, the information mapped to an index i in a label α in $\mathbb{A}$ is already present in the two components given by the application of the $\mathbf{pre}$ and $\mathbf{post}$ operations on α. That is, label categories built via the application of these two label transformers already have a preorder structure, with the ordering relation given by identity and pairing relations, respectively.

The main issue in the process of viewing $\mathbb{A}$ as the preorder $Pre(\mathbb{A})$ relates to the treatment of indices built via the ***EmittedInfo*** label transformer. Such indices are mapped to free monoids. A free monoid is a special case of a preorder, when the prefix order of the free monoid is considered as the ordering relation. Nevertheless, in the context of arrow-labeled transition systems, to consider the prefix order of the monoid mapped to an ***EmittedInfo***-declared index it is necessary to consider a "history" of emitted elements throughout a computation. This history is automatically made available if we consider label-sequences of derivatives in the *derivation tree* [10, Page 48] view of the transitions of $\mathcal{A}$, that represents branches or *paths* in the derivation tree of $\mathcal{A}$. Note that these paths actually represent computations in the transitions of $\mathcal{A}$. The label-sequence of a derivative captures exactly the notion of a "history" of labels from the initial state up to a certain (target) configuration in a computation.

Using lists as the data structure to model the history of values mapped to indices declared via an ***EmittedInfo*** label transformer, and the $\mathbf{pre}$ and $\mathbf{post}$ projection functions on labels for indices built using ***ContextInfo*** and ***MutableInfo*** label transformers, we define two functions, namely $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$, that compute the configurations of $\mathcal{TS}(\mathcal{A})$ when applied to the derivatives of the derivation tree representation

of the transitions of $\mathcal{A}$. Note that the definitions for $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ capture the notion of the preorder label category, since the second projection of the elements they return is of type $Pre(\mathbb{A})$.

To summarize, the mapping $\mathcal{A}to\mathcal{TS}$ is defined as in the context of an $MSOS$ specification $\mathcal{M}$, generating a transition system $\mathcal{TS}(\mathcal{A})$ from an arrow-labeled transition system $\mathcal{A}$ induced by $\mathcal{M}$ such that the configurations of the transitions of $\mathcal{TS}(\mathcal{A})$ are given by the application of the functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ to the derivatives of the paths in the derivation tree representation of the transitions of $\mathcal{A}$. The forthcoming definitions formalize this explanation.

Syntactical convention The expression $\alpha.i$ is an abbreviation for $\mathbf{get}(\alpha, i)$.

Definition 5 (Preorder view of a Label Category). *A preorder view of a label category $\mathbb{A}$, in the context of a $MSOS$ specification $\mathcal{M}$, is represented as $Pre(\mathbb{A})$, such that for every label α in $\mathbb{A}$ there exist two elements in $Pre(\mathbb{A})$, representing the label information prior, during, and posterior to the transition involving α.*

The elements of $Pre(\mathbb{A})$ are tuples whose elements are the indices of the arrow structure of $Morph(\mathbb{A})$. The indices are mapped to values typed as follows.

- *For an index i declared in $Lc_{\mathcal{M}}$ via a **ContextInfo**(i, T) declaration, $Pre(\mathbb{A}) = \langle T, id_T \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is T with the ordering relation defined as the identity relation id_T.*
- *For an index i declared in $Lc_{\mathcal{M}}$ via a **MutableInfo**(i, T) declaration, $Pre(\mathbb{A}) = \langle T, \preceq \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is T, with the ordering relation $\preceq$ defined as follows. Given $a, b \in T$, $a \preceq b$ if $\langle a, b \rangle = \alpha.i$.*
- *For an index i declared in $Lc_{\mathcal{M}}$ via an **EmittedInfo**$(i, T^*, \cdot, e)$, $Pre(\mathbb{A}) = \langle T^*, \leq \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is $List[T^*]$, with the prefix ordering relation $\leq$ according to [10].*

Definition 6 (Path in a derivation tree). *A* path *in a derivation tree is a branch of the tree beginning at the initial state and is represented as the set of derivatives that characterize that branch. The set of paths in the derivation tree of an $ALTS$ $\mathcal{A}$ is represented as $\Pi(\mathcal{A})$.*

Definition 7 (Derivatives of an $ALTS$). *The set of derivatives of an arrow-labeled transition system $\mathcal{A}$ is defined as $\mathcal{D}(\mathcal{A}) = \{\ \delta \mid \pi \in \Pi(\mathcal{A}), \delta$ is a derivative in $\pi\}$. The set of derivatives of a path π is defined as $\mathcal{D}_{\pi}(\mathcal{A}) = \{\ \delta \mid \delta$ is a derivative in $\pi\}$.*

Definition 8 (Partial functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$). *Let us consider $\Pi(\mathcal{A})$, the set of paths of a derivation tree of an arrow labeled transition system $\mathcal{A}$ induced by an $MSOS$ specification $\mathcal{M}$; γ_0 to γ_n configurations in $\Gamma_{\mathcal{M}}$; ι an identity label in $\mathbb{I}_{\mathbb{A}}$; $\alpha_1 \ldots \alpha_n$ labels in $Morph(\mathbb{A}_{\mathcal{A}})$; ω and ω' elements of $Pre(\mathbb{A}_{\mathcal{A}})$; and δ a derivative in $\mathcal{D}(\mathcal{A})$, having the following form:*

$$\delta = \{\ (\iota, \gamma_0), (\alpha_1, \gamma_1), \ldots, (\alpha_1 \ldots \alpha_{n-1}, \gamma_{n-1}), (\alpha_1 \ldots \alpha_{n-1}\alpha_n, \gamma_n)\ \}$$

Functions τ_1 and τ_2 are the first and second projection functions, respectively, for the pair $(\Gamma \times Pre(\mathbb{A}_{\mathcal{A}}))$.

The partial functions $preState_{\mathcal{A}}$ *and* $postState_{\mathcal{A}}$ *are only defined for values* $\pi \in \Pi(\mathcal{A})$ *and* $\delta \in \mathcal{D}(\mathcal{A})$ *such that* δ *is a derivative in* π*. They have the following signatures.*

$$preState_{\mathcal{A}} : \Pi(\mathcal{A}) \times \mathcal{D}(\mathcal{A}) \rightarrow (\Gamma \times Pre(\mathbb{A}))$$
$$postState_{\mathcal{A}} : \Pi(\mathcal{A}) \times \mathcal{D}(\mathcal{A}) \rightarrow (\Gamma \times Pre(\mathbb{A}))$$

The function $preState_{\mathcal{A}}$ *is defined inductively on the length of the derivatives of a path. The base case of* $preState_{\mathcal{A}}$ *is applied to a path* π *and the first immediate derivative in* π*. It is defined by the following equation,*

$$preState_{\mathcal{A}}(\pi, (\alpha_1, \gamma_1)) = (\gamma_0, \omega) \tag{18}$$

where γ_0 *is the initial configuration of* $\mathcal{A}$ *and the indices of* ω *are associated to values in the following way:*

- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***ContextInfo***(i, T) *label transformer, its value in* ω *is the same as in the label* α_1*, that is,* $\omega.i = \alpha_1.i = \mathbf{pre}(\alpha_1).i$*.*
- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***MutableInfo***(i, T) *label transformer , with* v_1 *and* v_2 *elements of type* T*,* $\omega.i = v_1 = \mathbf{pre}(\alpha_1).i$ *with* $\alpha_1.i = (v_1, v_2)$*.*
- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via an* ***EmittedInfo***$(i, T^*, \cdot, e)$ *label transformer,* $\omega.i = emptyList$*.*

The inductive case of function $preState_{\mathcal{A}}$ *is applied to a path* π *and a derivative in* π*, and is defined by the following equation,*

$$preState_{\mathcal{A}}(\pi, (\alpha_1 \ldots \alpha_{n-1}\alpha_n, \gamma_n)) = (\gamma_{n-1}, \omega) \tag{19}$$

where γ_{n-1} *is such that* $(\alpha_1 \ldots \alpha_{n-1}, \gamma_{n-1})$ *is a derivative in* π*, and the indices of* ω *are associated to values in the following way:*

- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***ContextInfo***(i, T) *label transformer,* $\omega.i = \alpha_n.i = \mathbf{pre}(\alpha_n).i$*.*
- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***MutableInfo***(i, T) *label transformer, with* v_1 *and* v_2 *elements of type* T*,* $\omega.i = v_1 = \mathbf{pre}(\alpha_n).i$ *with* $\alpha_n.i = (v_1, v_2)$*.*
- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***EmittedInfo***$(i, T^*, \cdot, e)$ *label transformer,*

$$\omega.i = append(\alpha_{n-1}.i, (\tau_2(preState_{\mathcal{A}}(\pi, (\alpha_1 \ldots \alpha_{n-1}, \gamma_{n-1})))).i).$$

The function $postState_{\mathcal{A}}$ *applied to* π *and a derivative in* π*, is defined by the following equation,*

$$postState_{\mathcal{A}}(\pi, (\alpha_1 \ldots \alpha_{n-1}\alpha_n, \gamma_n)) = (\gamma_n, \omega) \tag{20}$$

where the indices of ω' *are associated to values in the following way:*

- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***ContextInfo***(i, T) *label transformer, its value in* ω *is the same as in the label* α_n*, that is,* $Pre(\mathbb{A})$*,* $\omega.i = \alpha_n.i = \mathbf{post}(\alpha_n).i$*.*
- *For an index* i *declared in* $Lc_{\mathcal{M}}$ *via a* ***MutableInfo***(i, T) *label transformer declaration, with* v_1 *and* v_2 *elements of type* T*,* $\omega.i = v_2 = \mathbf{post}(\alpha_n).i$ *with* $\alpha_n.i = (v_1, v_2)$*.*

– *For an index i declared in $Lc_{\mathcal{M}}$ via an* ***EmittedInfo***$(i, T^*, \cdot, e)$ *label transformer,*

$$\omega.i = append(\alpha_n.i, (\tau_2(preState_{\mathcal{A}}(\pi, (\alpha_1 \ldots \alpha_n, \gamma_n)))).i).$$

Definition 9 (The mapping from $ALTS$ to TS). *The mapping $\mathcal{A}to\mathcal{TS} : \mathcal{A} \rightarrow \mathcal{TS}$, from $ALTS$ to TS, when applied to an $ALTS$ of the form $\mathcal{A} = \langle \Gamma, Morph(\mathbb{A}), \longrightarrow \rangle$, with Γ the configurations of $\mathcal{A}$, $Morph(\mathbb{A})$ the arrows of a label category, and $\longrightarrow$ the transition relation declared as $\longrightarrow \subseteq \Gamma \times Morph(\mathbb{A}) \times \Gamma$, generates a transition system of the form $\mathcal{TS}(\mathcal{A}) = \langle (\Gamma \times Pre(\mathbb{A})), \longrightarrow_{\mathcal{TS}(\mathcal{A})} \rangle$, with the transition relation defined as*

$$\longrightarrow_{\mathcal{TS}(\mathcal{A})} \subseteq (\Gamma \times Pre(\mathbb{A})) \times (\Gamma \times Pre(\mathbb{A})) \tag{21}$$

and $Pre(\mathbb{A})$ a preorder view of the label category $\mathbb{A}$. The elements of $(\Gamma \times Pre(\mathbb{A}))$ and $\longrightarrow_{\mathcal{TS}(\mathcal{A})}$ are given by the following equations, where $\gamma \in \Gamma$, π is a path in $\Pi(\mathcal{A})$ and δ a derivative in $\mathcal{D}_\pi(\mathcal{A})$.[4]

$$\mathcal{TS}(\mathcal{A}) = \langle\ \Gamma_{\mathcal{TS}(\mathcal{A})}, \longrightarrow_{\mathcal{TS}(\mathcal{A})}\ \rangle \tag{22}$$

$$\Gamma_{\mathcal{TS}(\mathcal{A})} = \{\ preState_{\mathcal{A}}(\pi, \delta) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_\pi(\mathcal{A})\ \} \cup \{\ postState_{\mathcal{A}}(\pi, \delta) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_\pi(\mathcal{A})\ \} \tag{23}$$

$$\longrightarrow_{\mathcal{TS}(\mathcal{A})} = \{\ (\ preState_{\mathcal{A}}(\pi, \delta),\ postState_{\mathcal{A}}(\pi, \delta)\ ) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_\pi(\mathcal{A})\ \} \tag{24}$$

4 The Correctness Proof of $\mathcal{M}to\mathcal{R}$

The main idea behind the mapping from $MSOS$ to RWL is to "de-encapsulate" the information inside the labels in $MSOS$. According to Definition 4, this is accomplished syntactically by enriching the configurations of rewrite rules with label information. Semantically this means mapping the arrow-labeled transition system induced by an $MSOS$ specification to a(n) (unlabeled) transition system, as shown by Definition 9.

In order to prove that the mapping is correct, it is necessary to show that computations are preserved by the $\mathcal{R}$-system $\mathcal{T}_{\mathcal{R}}$, the *initial* model of the rewrite theory generated from an $MSOS$ specification. The proof is twofold. Firstly, it is necessary to show that the transition system obtained from an arrow-labeled transition system by viewing its label category as a preoder preserves the computations of the original arrow-labeled transition system. Secondly, it should be proven that this transition system is the model for the rewrite theory resulting from the application of $\mathcal{M}to\mathcal{R}$ to that $MSOS$ specification.

The proof of Theorem 1 establishes that there exists a transition system associated with an arrow-labeled transition system induced by an $MSOS$ specification $\mathcal{M}$. Using the equivalence between $\mathcal{R}$-systems and transition systems originally presented in [9, Section 3.3], and summarized in Section 2.2, it is proven that there exists a $\mathcal{R}$-system equivalent to an arrow-labeled transition system. The proof of Theorem 2 shows that this $\mathcal{R}$-system is a model for the rewrite theory generated by the application of $\mathcal{M}to\mathcal{R}$ to $\mathcal{M}$.

[4] Note that the partial functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ are applied to $\mathcal{D}_\pi(\mathcal{A})$ and not to $\mathcal{D}(\mathcal{A})$.

Theorem 1 (Preservation of computation in $\mathcal{A}to\mathcal{TS}$). *For every arrow-labeled transition system $\mathcal{A}$, in the context of specifications in $MSOS$, there exists a corresponding transition system $\mathcal{TS}(\mathcal{A})$ according to Definition 9 such that the states and transitions of $\mathcal{A}$ are in a one-to-one correspondence to states and transitions of $\mathcal{TS}(\mathcal{A})$ and that all transitions of finite length of $\mathcal{TS}(\mathcal{A})$ are in $\mathcal{A}$.*

Proof Sketch. Since $\Pi(\mathcal{A})$ is the spanning tree of the graph view of the transition relation set of $\mathcal{A}$, having the initial state of $\mathcal{A}$ as root, the elements of the transition relation set in $\mathcal{TS}(\mathcal{A})$ are in one-to-one correspondence to the elements of the transition relation set in $\mathcal{A}$, when the reflexive steps are removed and the transitive steps are expanded from $\mathcal{TS}(\mathcal{A})$.

The elements of the preorder $Pre(\mathbb{A})$ are tuples formed by values associated to the indices of the objects in the label category $\mathbb{A}$. Thus, the elements of the preorder $Pre(\mathbb{A})$ are in one-to-one correspondence to the indices of the objects in the label category $\mathbb{A}$. □

Corollary 1 ($ALTS$—$\mathcal{R}$-system correspondence). *For every arrow-labeled transition system $\mathcal{A}$ induced by a $MSOS$ specification $\mathcal{M}$, there exists a corresponding $\mathcal{R}$-system for a rewrite theory $\mathcal{R}$ that preserves the state information of $\mathcal{A}$, that preserves the computations of $\mathcal{A}$ and that has its finite length transitions represented in $\mathcal{A}$.*

Proof Sketch. Applying the mapping from arrow-labeled transition systems to transition systems, to any $ALTS$ $\mathcal{A}$, yields a corresponding transition system $\mathcal{TS}(\mathcal{A})$, according to Definition 9, that is a model $\mathcal{S}$ of a rewrite theory $\mathcal{R}$ via the equivalence between $\mathcal{R}$-systems and transition systems, defined in [9, Section 3.3]. □

Theorem 2 shows that the rewrite theory obtained by the application of $\mathcal{M}to\mathcal{R}$ to $\mathcal{M}$ is the very theory satisfied by $\mathcal{S}$ mentioned in the proof sketch of Theorem 1. Note that $\mathcal{S}$ is indeed the initial model of the rewrite theory $\mathcal{M}to\mathcal{R}(\mathcal{M})$.

Theorem 2 ($\mathcal{TS}(\mathcal{A})$ is the model for $\mathcal{R}$). *If $\mathcal{M}$ is an $MSOS$ specification, $\mathcal{A}$ is the arrow-labeled transition system induced by $\mathcal{M}$, $\mathcal{TS}(\mathcal{A})$ is the transition system equivalent to $\mathcal{A}$ and $\mathcal{R}$ is the rewrite theory produced by the application of the mapping to $\mathcal{M}$ then $\mathcal{TS}(\mathcal{A})$ is a model for $\mathcal{R}$.*

Proof Sketch. This proof is done by induction on the set of transition rules of $\mathcal{M}$ and case analysis on the structure of each transition rule. It is shown that for every possible $\mathcal{M}$ modeled by $\mathcal{A}$, $\mathcal{TS}(\mathcal{A})$ is a model for $\mathcal{R}$, and according to Corollary 1, $\mathcal{TS}(\mathcal{A})$ corresponds to a $\mathcal{R}$-system. □

Corollary 2 (Correctness of the mapping). *Every transition* t *in an arrow-labeled transition system $\mathcal{A}$ induced by a $MSOS$ specification $\mathcal{M}$ has a corresponding transition in $\mathcal{S}$, the $\mathcal{R}$-system that models the rewrite theory $\mathcal{R}$ generated via the application of the mapping to $\mathcal{M}$.*

Proof Sketch. Corollary 1 proves that $\mathcal{A}$ has an associated transition system $\mathcal{TS}(\mathcal{A})$, correspondent to a $\mathcal{R}$-system, that preserves its computations. Theorem 2 proves that $\mathcal{TS}(\mathcal{A})$ is a model for $\mathcal{R}$. □

5 The MSOS-SL Interpreter

The MSOS-SL Interpreter [4, Chapter 6] is a prototype implementation of the $\mathcal{M}to\mathcal{R}$ mapping in the Maude system, a high-performance implementation of RWL [2]. The MSOS-SL Interpreter is built on top of Maude+rc, an extension of the Maude system that we have developed, that supports the so called *rewriting conditions*, that is, rewritings in the conditions of conditional rewrite rules. This extension is mainly a term evaluation strategy that implements a Prolog-like search. We refer to [4] for a complete explanation of Maude+rc.[5]

Formally, the mapping from $MSOS$ to RWL transformation process is a mapping ϕ_{MSOS} : `MsosModule` $\rightarrow$ `EnStrOModule` $\rightarrow$ `Module`, where the sort `MsosModule` represents $MSOS$ specifications, the `EnStrOModule` represents object-oriented rewrite theories, and the sort `Module` represents rewrite theories.

The mapping from `EnStrOModule` $\rightarrow$ `Module` was developed by Duran in his Ph.D. thesis [6] and is implemented as part of the Maude system.

The mapping from $MSOS$ to RWL is implemented in Maude as a further extension to the module algebra of Maude, built on top of Maude+rc. The module algebra extended by Maude+rc is further extended with the sort `MsosModule`, which defines a language that we have named MSOS-SL, an acronym for $MSOS$ specification language. Terms in `MsosModule` are transformed into terms in `EnStrOModule` following the transformations formally defined in Section 3, that is, the label category declaration in a term in `MsosModule` is transformed into a class declaration in a term in `EnStrOModule`, and the transition rule set declaration in a term in `MsosModule` is transformed into a rewrite rule set with rewrite conditions in `EnStrOModule`. This implementation defines the MSOS-SL Interpreter.

In [5] the present authors have proposed the Maude Action Tool (MAT), an executable environment for Action Semantics [11], a formalism for specifying the operational semantics of programming languages in a modular and readable way, using the reflective capabilities of Maude. The work presented in [5] was a prototype of MAT, in which we were experimenting with the ideas of the transformation from $MSOS$ to RWL, which are now formalized in Section 3. The rewrite theory representing the $MSOS$ specification of action notation, which was manually written for the MAT first prototype, is now automatically generated by the MSOS-SL Interpreter when the MSOS-SL specification of action notation is entered in the MSOS-SL Interpreter. Of course, the $MSOS$ specification of action notation is now one among a very wide variety of $MSOS$ specifications that the MSOS-SL Interpreter can execute. Due to space limitations, we can not present all the details of the relationship between MAT and the MSOS-SL Interpreter. For a complete presentation, we refer to [4, Chapter 6].

6 Conclusion

We have presented $\mathcal{M}to\mathcal{R}$, a proven correct mapping from modular structural operational semantics ($MSOS$) to rewriting logic (RWL). Such mapping allows the defini-

[5] The *rcrl* rule extension to the Maude system that we have described in [5] evolved into Maude+rc.

tion of formal tools for $MSOS$ such as the MSOS-SL Interpreter, our implementation for $\mathcal{M}to\mathcal{R}$ as a prototype in the Maude system. The MSOS-SL Interpreter allows the execution of $MSOS$ specifications when written using the MSOS-SL specification language: a Maude-like language with support for the declaration of label categories and (arrow-)labeled transition rules.

Another possible mapping from $MSOS$ to RWL would consider labeled transitions as terms in the generated rewrite theory. This is the so-called sequent-based mapping [7]. This approach leads indeed to a simpler translation scheme. However our mapping derives a direct interpreter for $MSOS$. The former mapping would require existential queries in order to execute a $MSOS$ specification in a prolog-like way. From an efficiency point of view and in the context of a Maude implementation both approaches are comparable assuming that rewriting conditions are implemented in the Maude virtual machine. In [13] a $MSOS$ prolog-based executable environment following the sequent-based approach is presented. The efficiency of MSOS-SL Interpreter will only be reasonable and thus comparable to such implementations or usable by third parties when rewriting conditions are native implemented in the Maude virtual machine. This feature will be available in the next version of the Maude system, Maude 2.0. Another interesting issue regarding MSOS-SL Interpreter when compared to a prolog-based implementation is the possibility of incorporating the model-checking facilities that the Maude 2.0 will provide.

Our future work will focus on two main subjects discussed above: improving the efficiency of the MSOS-SL Interpreter and adding model checking capabilities to the MSOS-SL Interpreter, evolving the interpreter into a tool set. At the implementation level, these two tasks are closely coupled with the next release of the Maude 2.0 system, since rewriting conditions are scheduled to be supported and a model checker for linear time temporal logic is already present in the most recent alpha releases of the Maude system.

References

1. M. Clavel. *Reflection in rewriting logic: metalogical foundations and metaprogramming applications*. CSLI Publications, 2000.
2. M. Clavel, F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, `http://maude.csl.sri.com`, January 1999.
3. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, August 2002.
4. C. de O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`.
5. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST'00, Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology, Iowa City, IA, USA*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer, May 2000.
6. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Mlaga, Escuela Tcnica Superior de Ingeniera Informtica, 1999.

7. N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. `http://maude.csl.sri.com/papers`.
8. J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *Proceedings ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM Press, 1990.
9. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1989.
11. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
12. P. D. Mosses. A Modular SOS for ML concurrency primitives. Technical Report Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
13. P. D. Mosses. Pragmatics of Modular SOS. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.
14. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, 1981.

Program Synthesis Based on the Equivalent Transformation Computation Model

Kiyoshi Akama[1], Ekawit Nantajeewarawat[2], and Hidekatsu Koike[1]

[1] CIMS, Hokkaido Univ., North 11, West 5, Sapporo, Hokkaido, 060-0811, Japan
{akama,koke}@cims.hokudai.ac.jp
[2] IT, Sirindhorn Intl. Inst. of Tech., Thammasat Univ., Pathumthani 12121, Thailand
ekawit@siit.tu.ac.th

Abstract. Effective generation of efficient and correct programs from specifications is the underlying design motivation of the Equivalent Transformation (ET) computation model. This concise paper explains how the ET model satisfies major required features of a program synthesis framework, and outlines a three-phase program synthesis method.

Motivation and Introduction. In declarative paradigms, a declarative description plays the role of a precise specification, and, at the same time, operates as a program. A number of works on amalgamation and generalization of declarative languages have been proposed. Most of them have been driven mainly by computation-oriented requirements, e.g., enhancement of operational semantics and integration of computation models; other important related concepts such as program synthesis and program transformation are investigated only afterwards and not inherent in their designs. In comparison, the *Equivalent Transformation (ET)* paradigm, originally proposed in [2], takes a program-synthesis-oriented approach. As such, its primary philosophical principle is to consider essential features of a framework for program synthesis first and to develop a computation model accordingly. Three major required features of a program synthesis framework are identified. First, the concept of a program specification should be clearly separated from that of a program: a program specification provides detailed description of requirements from which a program is constructed and based on which the correctness of a program is verified. Secondly, for the framework to be widely applicable, the concept of a program should be sufficiently general, and the framework should associate a sufficiently large set of correct programs with each program specification in order that derivation and selection of efficient programs can be discussed with sufficient generality. Thirdly, the framework should facilitate establishment of a solid theory for correctness verification of programs and component-based program construction—a correct program should be easily constructed out of simpler correct components.

The ET Framework. Declarative descriptions are only used for formulating program specifications, but not used as programs in the ET scheme. More precisely, a *program specification* is a pair $\langle D, Q \rangle$, where D is a declarative description, called a *definition part*, and Q is a set of declarative descriptions, each of which is called a *query part*. For each query part q in Q, the pair $\langle D, q \rangle$ is

M. Leuschel (Ed.): LOPSTR 2002, LNCS 2664, pp. 278–279, 2003.

regarded as a *problem description*, with which a declarative meaning is associated. A *program*, by contrast, is a set of rewriting rules, each of which defines a transformation operation on declarative descriptions. Computation consists in successive transformation of a given query part by application of rewriting rules so as to derive a simpler query part from which the answers to the specified query are readily obtained. The clear-cut separation of programs from specifications satisfies the first requirement. To fulfill the second requirement, the framework adopts equivalent transformation (ET) as its principal basis for discussing the correctness of computation: the correctness of answers is guaranteed as long as a given problem is transformed equivalently in each computation step. A specification is related with every program that always transforms a problem description covered by the specification equivalently. The framework uses in its theoretical setting a highly abstract notion of a rewriting rule (and thus of a program), i.e., a rewriting rule is mathematically identified with a binary relation on declarative descriptions. The ET principle along with the generality of the concept of a program ensures that a sufficiently large set of correct programs can be connected with each specification. It moreover enables development of a simple and general theoretical foundation for correctness verification of programs—a rewriting rule is correct with respect to a specification S if it always preserves the meaning associated with each problem description covered by S, and a program is correct if it consists only of correct rewriting rules. This foundation has two significant consequences: the correctness of one rewriting rule is completely independent of the correctness of another rewriting rule; in addition, the correctness of a program depends solely on the correctness of the rewriting rules it contains, not on any kind of their interconnection. As a result, a correct program can be constructed incrementally by addition of individual correct rewriting rules on demand—component-based program construction is inherently supported.

Program synthesis in the ET framework consists of three main phases: (1) transformation of a given specification, (2) generation of correct rewriting rules, and (3) association of some rule-firing control information with the generated rewriting rules. In the first phase, a given initial specification $\langle D, Q\rangle$ is transformed into another specification $\langle D', Q'\rangle$ that is equivalent to the initial one but has a more suitable form for generation of efficient rewriting rules in the second phase. Methods that can be applied in the first phase include modification of the underlying data domain and the specialization operation thereon [1], as well as methods from research on program transformation and partial evaluation. The second phase, which is the major part of program synthesis in this framework, consists in systematically generating (normally by means of metacomputation) from the derived specification $\langle D', Q'\rangle$ a set of correct rewriting rules. The third phase is concerned with, e.g., prioritizing the obtained rewriting rules [2].

References

1. Akama, K.: Declarative Semantics of Logic Programs on Parameterized Representation Systems. Advances in Software Science and Technology **5** (1993) 45–63
2. Akama, K., Shimizu, T., Miyamoto, E.: Solving Problems by Equivalent Transformation of Declarative Programs. J. Japanese Soc. for AI **13** (1998) 944–952

Author Index