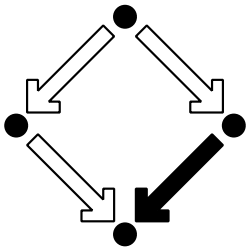
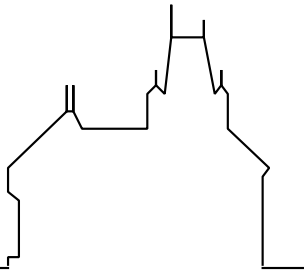


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



20th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2010

María Alpuente (ed.)

Castle of Hagenberg, Austria
July 23-25, 2010

RISC-Linz Report Series No. 10-14

Editors: RISC-Linz Faculty
K. Bosa, B. Buchberger, R. Hemmecke, T. Jebelean, E. Kartaschova, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, V. Pillwein, N. Popov, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by: Bundesministerium für Wissenschaft und Forschung, Johannes Kepler University Linz, Linzer Hochschulfonds, Doctoral Program "Computational Mathematics", Linz AG.

María Alpuente (Ed.)

Logic-Based Program Synthesis and Transformation

20th International Symposium
Hagenberg, Austria, July 23-25, 2010

Pre-Proceedings

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University in Linz

Preface

This book contains the papers presented at the 20th International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR 2010, which is held July 23-25, 2010, as a part of the RISC Summer 2010 in Hagenberg, Austria. LOPSTR 2010 is co-located with PPDP 2010, the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. Previous LOPSTR symposia were held in Coimbra (2009), Valencia (2008), Lyngby (2007), Venice (2006 and 1999), London (2005 and 2000), Verona (2004), Uppsala (2003), Madrid (2002), Paphos (2001), Manchester (1998, 1992, and 1991), Leuven (1997), Stockholm (1996), Arnhem (1995), Pisa (1994), and Louvain-la-Neuve (1993). Information about the conference can be found at: http://www.risc.jku.at/about/conferences/lopstr2010/.

The aim of the LOPSTR series is to stimulate and promote international research and collaboration in logic-based program development. LOPSTR traditionally solicits contributions, in any language paradigm, in the areas of specification, synthesis, verification, analysis, optimization, specialization, security, certification, applications and tools, program/model manipulation, and transformational techniques. LOPSTR has a reputation for being a lively, friendly forum for presenting and discussing work in progress. Formal proceedings are produced only after the symposium so that authors can incorporate this feedback in the published papers. The LOPSTR 2010 post-proceedings will be published in the Lecture Notes in Computer Science series of Springer-Verlag.

In response to the call for papers, 26 contributions were submitted from 15 different countries. The Programme Committee decided to accept eight full papers and seven extended abstracts, basing this choice on their scientific quality, originality, and relevance to the symposium. Each paper was reviewed by at least three Program Committee members or external referees. In addition to the 15 contributed papers, this volume includes the abstracts of the invited talks by three outstanding speakers: Bruno Buchberger (RISC, Johannes Kepler University Linz, Austria), Olivier Danvy (University of Aarhus, Denmark), and Johann Schumann (RIACS/NASA Ames Research Center, USA).

I want to thank the Program Committee members, who worked diligently to produce high-quality reviews for the submitted papers, as well as all the external reviewers involved in the paper selection. I am very grateful the LOPSTR 2010 Conference Chair, Temur Kutsia, and the local organizers of RISC Summer 2010 for the great job they did in preparing the conference. I also thank Andrei Voronkov for his excellent EasyChair system that automates many of the tasks involved in chairing a conference. Finally, I gratefully acknowledge the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University of Linz, which sponsored this event.

July 2010

María Alpuente
Program Chair

Organization

Program Chair

María Alpuente
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n
E-46022 Valencia, Spain
Email: alpuente@dsic.upv.es

Conference Chair

Temur Kutsia
Research Institute for Symbolic Computation
Johannes Kepler University of Linz
Altenbergerstrasse 69
A-4040 Linz, Austria
Email: kutsia@risc.uni-linz.ac.at

Program Committee

María Alpuente	Universitat Politècnica de València, Spain, chair
Sergio Antoy	Portland State University, USA
Gilles Barthe	IMDEA Software, Madrid
Manuel Carro	Technical University of Madrid, Spain
Marco Comini	University of Udine, Italy
Danny De Schreye	Katholieke Universiteit Leuven, Belgium
Santiago Escobar	Universitat Politècnica de València, Spain
Moreno Falaschi	University of Siena, Italy
Fabio Fioravanti	University of Chieti - Pescara, Italy
John Gallagher	Roskilde University, Denmark
Michael Hanus	University of Kiel, Germany
Patricia M. Hill	University of Parma, Italy
Andy King	University of Kent, UK
Temur Kutsia	Johannes Kepler University of Linz, Austria
Ralf Lämmel	Universität Koblenz-Landau, Germany
Michael Leuschel	University of Düsseldorf, Germany
Yanhong Annie Liu	State University of New York at Stony Brook, USA
Julio Mariño	Technical University of Madrid, Spain
Ricardo Peña	Complutense University of Madrid, Spain
Peter Schneider-Kamp	University of Southern Denmark
Alicia Villanueva	Universitat Politècnica de València, Spain

Additional Referees

Giorgio Bacci
Giovanni Bacci
Demis Ballis
Amir Ben-Amram
Jon Brandveini
Bernd Brassel
Maurice Bruynooghe
Khaled Bsaiesi
Rafael Caballero
Juan Manuel Crespo
Giorgio Delzanno
Carsten Fuhs
Samir Genaim
Jeremy Gibbons
José Iborra
Barton Masseyi
Fred Mesnard
Ruben Monjarazi
Manuel Montenegro
Federico Olmedo
Étienne Payet
Alberto Pettorossi
Maurizio Proietti
Andrew Santosai
Rahul Savani
Ivan Scagnetto
Thomas Stroeder
Tuncay Tekle

Table of Contents

Algorithm Synthesis by Lazy Thinking: Case Study Gröbner Bases

Bruno Buchberger

Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria, Europe
Bruno.Buchberger@risc.jku.at

In the Theorema Project, we try to computer-support (semi-automate) the exploration of mathematical theories (invention of concepts by definition, invention and proof of propositions, invention of problems, invention and verification of algorithms for solving problems). In the frame of Theorema, the speaker's "Lazy Thinking" approach is a general heuristic method for inventing (and proving) theorems and inventing (and proving) algorithms. It consists of a combination of the use of formulae schemes (for proposing potential theorems and algorithms) and the automated analysis of failures in proof attempts (for proving the appropriateness of the schemes) and the automated generation of lemmata and specifications for subalgorithms from failing proofs.

In the talk, we focus on the use of Lazy Thinking for algorithm synthesis: Starting from a formal specification P of a problem, we choose an algorithm scheme A from a library of algorithm schemes. An algorithm scheme is a formula that explains how A is composed of subalgorithms B, C, etc. ... . "Divide and Conquer" and "Critical Pair / Completion" are typical examples of algorithm schemes. Next, we attempt at proving (automatically) the correctness of the chosen scheme A for the given problem P. Typically, this attempt will fail because, at this point, nothing is known about the ingredient subalgorithms B, C, ... By a systematic failure analysis method, which is the core of the "Lazy Thinking" approach, specifications Q, R, ... for the subalgorithms B, C, ... are extracted that make the proof work. Hence, in the successful case, we arrive at a situation where one can say that, whatever algorithms B, C, ... one takes that satisfy the (automatically generated) specifications Q, R, ..., algorithm A with subalgorithms$B$, C, ... will be a correct algorithm for the given problem P.

We will first illustrate Lazy Thinking by the standard example of sorting: Starting from a specification P of the sorting problem, using various algorithm schemes, we arrive automatically at the various known sorting algorithms (in fact, at whole classes of possible algorithms).

We will then show that an algorithm for a problem as difficult as Gröbner bases construction can be synthesized, completely automatically, by Lazy Thinking. Gröbner bases are sets of multivariate polynomials with certain properties that allow to solve fundamental problems of algebraic geometry (commutative algebra) algorithmically. Gröbner bases have applications in many areas of mathematics, computer science, natural science and computational logic (e.g. for automated theorem proving in geometry). The traditional algorithm for constructing Gröbner bases, which was introduced many years ago by the speaker, is based

on the notion of "S-polyonomials" that are an analogue to the notion of "most general unifiers" in rewriting. By Lazy Thinking, this algorithm, including the central notion of "S-polynomials", can be synthesized completely automatically.

Thus, in a way, in this talk the speaker reports on how, in his mature age, he was able to automate his mathematical inventive power of his young age.

Three Syntactic Theories for Combinatory Graph Reduction

Olivier Danvy and Ian Zerny

Department of Computer Science, Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
{danvy,zerny}@cs.au.dk

Abstract. We present a purely syntactic theory of graph reduction for the canonical combinators S, K, and I, where graph vertices are represented with evaluation contexts and let expressions. We express this syntactic theory as a reduction semantics, which we refocus into the first storeless abstract machine for combinatory graph reduction, which we refunctionalize into the first storeless natural semantics for combinatory graph reduction. We then factor out the introduction of let expressions to denote as many graph vertices as possible *upfront* instead of *on demand*, resulting in a second syntactic theory, this one of term graphs in the sense of Barendregt et al. The corresponding storeless abstract machine and natural semantics follow mutatis mutandis. We then interpret let expressions as operations over a global store (thus shifting, in Strachey's words, from denotable entities to storable entities), resulting in a third syntactic theory. *The structure of the store-based abstract machine corresponding to this third syntactic theory coincides with that of Turner's original reduction machine.* The three syntactic theories presented here therefore have the following computational reality: they properly account for combinatory graph reduction As We Know It. We also outline how to extend the overall inter-derivation with the Y combinator.

1 Introduction

Fifteen years ago [4, 3, 29], Ariola, Felleisen, Maraist, Odersky and Wadler presented a purely syntactic theory for the call-by-need λ-calculus. In retrospect, their key insight was to syntactically represent 'def-use chains' for identifiers with evaluation contexts. For example, here is one of their contraction rules:

$$(\lambda x.E[x])\, v \to (\lambda x.E[v])\, v$$

In the left-hand side, an identifier, x, occurs (i.e., is 'used') in the eye of an evaluation context, E: its denotation is therefore needed.[1] This identifier is declared (i.e., is 'defined') in a λ-abstraction that is applied to a (syntactic) value v. In the right-hand side, v hygienically replaces x in the eye of E. There may be other occurrences of x in E: if another such one is needed later in the reduction sequence, this contraction rule will intervene again—it implements memoization.

[1] The notation $E[t]$ stands for a term that uniquely decomposes into a reduction context, E, and a subterm, t.

In this article, we take a next logical step and present a purely syntactic theory of graph reduction for the canonical combinators S, K and I. Our key technique is to syntactically represent def-use chains for graph vertices using evaluation contexts and let expressions declaring unique references. For example, the K combinator is traditionally specified as $K\,t_1\,t_2 = t_1$, for any terms t_1 and t_2, a specification that does not account for sharing of subterms before and after contraction. In contrast, our specification does account for sharing:

$$\begin{array}{rl} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\,t_1\ \mathsf{in} & \rightarrow \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_3 = t_1\ \mathsf{in} \\ \quad E_1[\mathsf{let}\ x_0 = x_1\,t_0\ \mathsf{in} & \qquad \mathsf{let}\ x_1 = x_2\,x_3\ \mathsf{in} \\ \quad\quad E_0[x_0]]] & \qquad E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in} \\ & \qquad\quad E_0[x_0]]] \\ & \quad \text{where } x_3 \text{ is fresh} \end{array}$$

This contraction rule should be read inside-out. In the left-hand side, i.e., in the redex, a reference, x_0, occurs in the eye of an evaluation context: its denotation is therefore needed. The definiens of x_0 is the application of a second reference, x_1, to a term t_0: the denotation of x_1 is therefore also needed. The definiens of x_1 is the application of a third reference, x_2, to a term t_1: the denotation of x_2 is therefore also needed. The definiens of x_2 is the K combinator. In the right-hand side, i.e., in the contractum, a fresh (and thus unique) reference, x_3, is introduced to denote t_1: it replaces the application of x_1 to t_0. Reducing the K combinator is thus achieved (1) by creating a fresh reference to t_1 to share any subsequent reduction in t_1,[2] and (2) by replacing the reference to the application $x_1\,t_0$ by x_3. There may be other occurrences of x_0 in E_0: if another such one is needed later in the reduction sequence, this contraction rule for K will not intervene again—its result has been memoized.

In Section 2, we fully specify our syntactic theory of combinatory graph reduction as a reduction semantics, and we then apply the first author's programme [11, 12] to derive the first storeless abstract machine and the first storeless natural semantics for combinatory graph reduction, in a way similar to what we recently did for the call-by-need λ-calculus [15].

Our syntactic theory introduces let expressions for applications on demand. In Section 3, we preprocess source terms into term graphs by introducing let expressions upfront for all source applications, and we present the corresponding reduction semantics, storeless abstract machine, and storeless natural semantics.

In Section 4, as a reality check, we map the explicit representation of graph vertices as let headers to implicit references in a global store. Again, we present the corresponding store-based reduction semantics and store-based abstract machine. This store-based abstract machine essentially coincides with Turner's original graph-reduction machine [37]. This coincidence provides an independent, objective bridge between the modern theory of combinatory graph reduction and its classical computational practice.

[2] References ensure sharing of subterms: they are uniquely defined, but they can have many uses. References can be freely duplicated, but what they refer to, i.e., their denotation, is not duplicated and is thus shared.

Prerequisites and notations: We expect an elementary awareness of the S, K and I combinators and how combinatory terms can be reduced to head normal form, either in principle (as a formal property in Combinatory Logic [6]) or in practice (as a stack-based graph-reduction machine [31, 37]). We also assume a basic familiarity with the formats of a reduction semantics, an abstract machine, and a natural semantics as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [12]; and with the concept of term graphs, as pedagogically presented in Blom's PhD dissertation [8]. In particular, we use the terms 'reduction context' and 'evaluation context' interchangeably.

For a notion of reduction $\mathcal{R}$ defined with a set of contraction rules, we define the contraction of a redex r into a contractum t as $(r, t) \in \mathcal{R}$. We use $\rightarrow_{\mathcal{R}}$ for one-step $\mathcal{R}$-reduction, and $\twoheadrightarrow_{\mathcal{R}}$ for the transitive-reflexive closure of $\rightarrow_{\mathcal{R}}$. A term t is in $\mathcal{R}$-normal form ($\mathcal{R}$-nf) if t does not contain a redex of $\mathcal{R}$.

Pictorial overview:

	terms	graphs	store
reduction semantics	Section 2.1	Section 3.1	Section 4.1
abstract machine	Section 2.2	Section 3.2	Section 4.2
natural semantics	Section 2.3	Section 3.3	

2 Three inter-derivable semantic artifacts for storeless combinatory graph reduction

Our starting point is the following grammar of combinatory terms:

$$t ::= I \mid K \mid S \mid t\,t$$

So a combinatory term is a combinator or a combination, i.e., the application of a term to another term.

We embed this grammar into a grammar of terms where sub-terms can be referred to through let expressions and where a program p is an initial term t denoted by a reference whose denotation is needed:

$$\begin{array}{l} t ::= I \mid K \mid S \mid t\,t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid x \\ p ::= \mathsf{let}\ x = t\ \mathsf{in}\ x \end{array}$$

In this grammar, a term is a combinatory term (i.e., a combinator or a combination), the declaration of a reference to a term in another term, or the occurrence of a declared reference. This grammar of terms is closed under contraction.

In our experience, however, there is a better fit for the contraction rules, namely the following sub-grammar where a denotable term is an original combinatory term or a term that generalizes the original program into declarations nested around a declared reference:

$$\begin{array}{l} p ::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\ d ::= I \mid K \mid S \mid d\,d \mid t \\ t ::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \end{array}$$

This grammar of terms excludes terms with let expressions whose body is a combinator or a combination. It is closed under contraction.

At this point, it would be tempting to define reduction contexts to reflect how references are needed in the reduction process:

$$\text{Reduction Context} \ni E ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\,d\ \mathsf{in}\ E[x]$$

The constructor "$\mathsf{let}\ x = d\ \mathsf{in}\ E$" accounts for the recursive search for the innermost reference in a term. The constructor "$\mathsf{let}\ x = E\,d\ \mathsf{in}\ E[x]$" accounts for the need of intermediate references.

In our experience, however, there is a better grammatical fit for contexts, namely one which separates the search for the innermost reference in a term and the subsequent construction that links needed references to their declaration. The former gives rise to delimited reduction contexts and the latter to def-use chains:

$$\begin{aligned} \text{Reduction Context} &\ni E ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\ \text{Def-use Chain} &\ni C ::= [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]] \end{aligned}$$

2.1 A reduction semantics

Here is the full definition of the syntax:

$$\begin{aligned} \text{Program} &\ni p ::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\ \text{Denotable Term} &\ni d ::= I \mid K \mid S \mid d\,d \mid t \\ \text{Term} &\ni t ::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \\ \text{Reduction Context} &\ni E ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\ \text{Def-use Chain} &\ni C ::= [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]] \end{aligned}$$

Reduction contexts reflect the recursive search for the innermost reference in a term. While returning from this search, def-use chains are constructed to connect each reference whose denotation is needed with its declaration site. We abbreviate $\mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]$ as $(x,\, d,\, E) \cdot C$ and we write $\Pi_0^{i=n}(x_i,\, d_i,\, E_i) \cdot C$ as short hand for $(x_n,\, d_n,\, E_n) \cdot \ldots \cdot (x_0,\, d_0,\, E_0) \cdot C$, and $|C|$ for the length of C, so that $|\Pi_0^{i=n}(x_i,\, d_i,\, E_i) \cdot [\,]| = n + 1$.

Axioms (i.e., contraction rules): Figure 1 displays the axioms. Each of (I), (K) and (S) is much as described in Section 1, with the addition of the inner def-use chain: it carries out a particular rearrangement while preserving sharing through common references.[3]

In the left-hand side of (*comb*), a reference, x_0, occurs in the eye of the current def-use chain, C: its denotation is therefore needed. Its definiens is a combination of two denotable terms, d_0 and d_1. In the right-hand side, a fresh reference, x_1, is introduced to denote d_0. This fresh reference extends the current def-use chain

[3] On the right-hand side of (I), (K) and (S), we have kept $E_0[C[x_0]]$ in order to highlight each particular rearrangement. It would be simple to "optimize" these right-hand sides by taking advantage of [a subsequent use of] (*ref*) and (*comb*), so that, e.g., the right-hand side of (K) contains $E_0[C[x_3]]$ instead.

$$
\begin{array}{rrcl}
(I) & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\ \qquad E_0[C[x_0]]] & \to & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = d_0\ \mathsf{in} \\ \qquad E_0[C[x_0]]] \\
(K) & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\ \qquad E_0[C[x_0]]]] & \to & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_3 = d_1\ \mathsf{in} \\ \qquad \mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in} \\ \qquad E_0[C[x_0]]]] \\
 & & & \text{where } x_3 \text{ is fresh} \\
(S) & \mathsf{let}\ x_3 = S\ \mathsf{in} \\ \qquad E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} \\ \qquad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\ \qquad E_0[C[x_0]]]]] & \to & \mathsf{let}\ x_3 = S\ \mathsf{in} \\ \qquad E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\ \qquad \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\ \qquad E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\ \qquad \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\ \qquad \mathsf{let}\ x_0 = (x_4\, x_6)\, (x_5\, x_6)\ \mathsf{in} \\ \qquad E_0[C[x_0]]]]] \\
 & & & \text{where } x_4,\ x_5 \text{ and } x_6 \text{ are fresh} \\
(comb) & \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in}\ E_0[C[x_0]] & \to & \mathsf{let}\ x_1 = d_0\ \mathsf{in}\ \mathsf{let}\ x_0 = x_1\, d_1\ \mathsf{in}\ E_0[C[x_0]] \\
 & & & \text{where } d_0 \text{ is not a reference} \\
 & & & \quad \text{and } x_1 \text{ is fresh} \\
(assoc) & \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]] & \to & \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ \mathsf{let}\ x_0 = t_0\ \mathsf{in}\ E_0[C[x_0]] \\
(ref) & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \to & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]
\end{array}
$$

Fig. 1. Reduction semantics for combinatory graph reduction: axioms

for d_0, thereby ensuring that any subsequent reduction in d_0 is shared.[4] This specific choice of d_0 ensures that any redex found in a subsequent search will be on the left of this combination, thereby enforcing left-most reduction.

The axiom (*assoc*) is used to flatten let expressions,[5] and the axiom (*ref*) to resolve indirect references.

The corresponding notion of reduction is $\mathcal{T}$:

$$\mathcal{T} = (I) \cup (K) \cup (S) \cup (comb) \cup (assoc) \cup (ref)$$

Reduction strategy: The reduction strategy is left-most outermost (and thus not optimal [28]): the def-use chains force us to only consider the outermost combination, and (*comb*) ensures that this outermost combination is the left-most one.

Recompositions: Figure 2 displays the recompositions of a reduction context with a term, and of a def-use chain with a reference:

Definition 1 (inside-out recomposition of contexts). *A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E, t\rangle_{io} \Uparrow_{\mathrm{rec}} t'$ holds. (See Figure 2 and also Footnote 1 for the notation $E[t]$.)*

[4] If d_0 is already a reference, it is already part of a def-use chain and no contraction need take place: $\mathsf{let}\ x_0 = x\, d_1\ \mathsf{in}\ E_0[C[x_0]]$ is not a redex.

[5] There is no need for the condition "x_1 does not occur free in E_0" since each label is unique.

Inside-out recomposition of a reduction context with a term:

$$\frac{}{\langle [\,],\, t\rangle_{io} \Uparrow_{\mathrm{rec}} t} \qquad \frac{\langle E,\, \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{io} \Uparrow_{\mathrm{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\, t\rangle_{io} \Uparrow_{\mathrm{rec}} t'}$$

Outside-in recomposition of a reduction context with a term:

$$\frac{}{\langle [\,],\, t\rangle_{oi} \Uparrow_{\mathrm{rec}} t} \qquad \frac{\langle E,\, t\rangle_{oi} \Uparrow_{\mathrm{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\, t\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x = d\ \mathsf{in}\ t'}$$

Recomposition of a def-use chain:

$$\frac{}{\langle [\,],\, x\rangle_{chain} \Uparrow_{\mathrm{rec}} x} \qquad \frac{\langle C,\, x'\rangle_{chain} \Uparrow_{\mathrm{rec}} t \qquad \langle E,\, t\rangle_{oi} \Uparrow_{\mathrm{rec}} t'}{\langle \mathsf{let}\ x' = [\,]\, d\ \mathsf{in}\ E[C[x']],\, x\rangle_{chain} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x' = x\, d\ \mathsf{in}\ t'}$$

Fig. 2. Reduction semantics for combinatory graph reduction: recompositions

Definition 2 (outside-in recomposition of contexts). *A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E,\, t\rangle_{oi} \Uparrow_{\mathrm{rec}} t'$ holds. (See Figure 2.)*

Outside-in recomposition of contexts is used as an auxiliary judgment in the recomposition of def-use chains:

Definition 3 (recomposition of def-use chains). *A def-use chain C is recomposed around a reference x into a term $t = C[x]$ whenever $\langle C,\, x\rangle_{chain} \Uparrow_{\mathrm{rec}} t$ holds.*[6] *(See Figure 2.)*

Decomposition: Decomposition implements the reduction strategy by searching for a redex and its reduction context in a term. Figure 3 displays this search as a transition system:

term-**transitions:** Given a term, we recursively dive into the bodies of its nested let expressions until its innermost reference x, which is therefore needed.

cont-**transitions:** Having found a reference x that is needed, we backtrack in search of its declaration, incrementally constructing a def-use chain for it.[7] If we do not find any redex, the term is in $\mathcal{T}$-normal form.

den-**transitions:** Having found the declaration of the reference that was needed, we check whether we have also found a redex and thus a decomposition. Otherwise, a combinator is not fully applied or a new reference is needed. We then resume a *cont*-transition, either on our way to a $\mathcal{T}$-normal form or extending the current def-use chain for this new reference.

Definition 4 (decomposition). *For any t,*

$$\langle t,\, [\,]\rangle_{term} \downarrow^{*}_{\mathrm{dec}} \begin{cases} \langle t\rangle_{nf} & \text{if } t \in \mathcal{T}\text{-}nf \\ \langle E,\, r\rangle_{dec} & \text{otherwise} \end{cases}$$

where r is the left-most outermost redex in t and E is its reduction context.

[6] As already pointed out in Footnote 1, the notation $C[x]$ stands for a term that uniquely decomposes into a def-use chain, C, and a reference, x.

[7] There is no transition for $\langle [\,],\, (x_0,\, E_0,\, C)\rangle_{cont}$ because all references are declared.

$$
\begin{aligned}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} &\downarrow_{\mathrm{dec}} \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} &\downarrow_{\mathrm{dec}} \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
\\
\langle [\,],\ t\rangle_{cont} &\downarrow_{\mathrm{dec}} \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\downarrow_{\mathrm{dec}} \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\downarrow_{\mathrm{dec}} \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
&\quad \text{where } x \neq x_0 \\
\\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]\rangle_{dec} \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} \\
&\qquad\qquad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\
&\qquad\qquad\quad E_0[C[x_0]]]]\rangle_{dec} \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} \\
&\qquad\qquad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} \\
&\qquad\qquad\quad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\
&\qquad\qquad\qquad E_0[C[x_0]]]]]\rangle_{dec} \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
&\quad \text{where } d_0 = I \ \text{ and } |C| < 1 \\
&\qquad \text{or } d_0 = K \text{ and } |C| < 2 \\
&\qquad \text{or } d_0 = S \ \text{ and } |C| < 3 \\
&\quad \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\, d_0,\ E_0,\ C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ (x_1,\ [\,],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ d_0\, d_1,\ E_0,\ C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
&\quad \text{where } d_0 \text{ is not a reference} \\
&\quad \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} &\downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]
\end{aligned}
$$

Fig. 3. Reduction semantics for combinatory graph reduction: decomposition

The transition system implementing decomposition can be seen as a big-step abstract machine [13]. As repeatedly pointed out in the first author's lecture notes at AFP 2008 [12], such a big-step abstract machine is often in defunctionalized form—as is the case here. In the present case, it can be refunctionalized into a function over source terms which is compositional. Ergo, it is expressible as a catamorphism over source terms. Further, the parameters of this catamorphism

are total functions. Therefore, the decomposition function is total. It yields either the given term if this term is in $\mathcal{T}$-normal form, or its left-most outermost redex and the corresponding reduction context.

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{T}$-normal form proceeds as follows: (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 5 (standard one-step reduction). *For any t,*

$$t \mapsto_{\mathcal{T}} t'' \quad \textit{if} \quad \begin{cases} \langle t, [\,] \rangle_{term} \downarrow^{*}_{\mathrm{dec}} \langle E, r \rangle_{dec} \\ (r, t') \in \mathcal{T} \\ \langle E, t' \rangle_{io} \Uparrow_{\mathrm{rec}} t'' \end{cases}$$

One-step reduction is a partial function because the given term may already be in $\mathcal{T}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 6 (reduction-based evaluation). *For any program p,*

$$p \mapsto^{*}_{\mathcal{T}} t \ \wedge \ t \in \mathcal{T}\text{-}nf$$

Evaluation is a partial function because it may diverge.

Most of the time, decomposition and recomposition(s) are kept implicit in published reduction semantics. We however observe that what was kept implicit is then progressively revealed as, e.g., one constructs an abstract machine to implement evaluation [19]. We believe that it is better to completely spell out reduction semantics upfront, because one is then in position to systematically calculate the corresponding abstract machines and natural semantics [12], as illustrated in the next two sections for syntactic graph reduction.

2.2 A storeless abstract machine

Reduction-based evaluation, as defined in Section 2.1, is inefficient because of its repeated decompositions and recompositions that construct each successive term in a reduction sequence. Refocusing [17] deforests these intermediate terms, and is defined very simply as continuing decomposition with the contractum and its reduction context. The reduction semantics of Section 2.1 satisfies the formal requirements for refocusing [17] and so its reduction-based evaluation function can be simplified into a reduction-free evaluation function that does not construct each successive term in reduction sequences. Reflecting the structure of the decomposition function of Figure 3, the result is an abstract machine whose 'corridor' transitions can be compressed into the abstract machine displayed in Figure 4:

$$
\begin{array}{rcl}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} & \to_{\mathrm{run}} & \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} & \to_{\mathrm{run}} & \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
 & & \\
\langle [\,],\ t\rangle_{cont} & \to_{\mathrm{run}} & \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} & \to_{\mathrm{run}} & \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
 & & \\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \to_{\mathrm{run}} & \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \to_{\mathrm{run}} & \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
 & & \text{where } x \neq x_0 \\
 & & \\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle x_0,\ d_0,\ E_0,\ C,\ E_1 \circ_{io} \mathsf{let}\ x_1 = I\ \mathsf{in}\ E\rangle_{den} \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle x_3,\ d_1,\ E',\ C,\ E_2 \circ_{io} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E\rangle_{den} \\
 & & \text{where } E' = (\mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = x_3\ \mathsf{in}\ E_0) \\
 & & \text{and } x_3 \text{ is fresh} \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle x_4,\ d_2,\ E',\ C',\ E_3 \circ_{io} \mathsf{let}\ x_3 = S\ \mathsf{in}\ E\rangle_{den} \\
 & & \text{where } E' = (\mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_5 = d_1\ \mathsf{in}\ \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_6 = d_0\ \mathsf{in}\ [\,]) \\
 & & \text{and } C' = (x_7,\ x_6,\ [\,])\cdot(x_0,\ x_5\, x_6,\ E_0)\cdot C \\
 & & \text{and } x_4, x_5, x_6 \text{ and } x_7 \text{ are fresh} \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
 & & \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \\
 & & \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\, d_0,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle E,\ (x_1,\ [\,],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ d_0\, d_1,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle x_1,\ d_0,\ [\,],\ (x_0,\ d_1,\ E_0)\cdot C,\ E\rangle_{den} \\
 & & \text{where } d_0 \text{ is not a reference and } x_1 \text{ is fresh} \\
\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle x_0,\ t_0,\ E_0,\ C,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ E\rangle_{den} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}} & \langle E,\ (x_1,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\ C)\rangle_{cont}
\end{array}
$$

Fig. 4. Storeless abstract machine for combinatory graph reduction

term-**transitions:** The *term*-transitions are the same as for decomposition.
cont-**transitions:** The *cont*-transitions are the same as for decomposition.
den-**transitions:** Having found the declaration of the reference that was needed, we check whether we have also found a redex. If so, we contract it and continue with a new needed reference. Otherwise, the *den*-transitions are the same as for decomposition.

This abstract machine uses the following two compositions of evaluation contexts:

Definition 7 (composition of evaluation contexts). *Two contexts that were constructed outside in are composed into an outside-in context as follows:*

$$[\,] \circ_{oi} E = E$$
$$(\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{oi} E = \mathsf{let}\ x = d\ \mathsf{in}\ (E' \circ_{oi} E)$$

This composition function is associative.

Definition 8 (mixed composition of evaluation contexts). *A context that was constructed inside out is composed with a context that was constructed outside in as follows:*

$$[\,] \circ_{io} E = E$$
$$(\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{io} E = E' \circ_{io} \mathsf{let}\ x = d\ \mathsf{in}\ E$$

The resulting context is constructed outside in.

Proposition 1 (full correctness). *For any program p,*

$$p \mapsto^*_{\mathcal{T}} t \ \wedge\ t \in \mathcal{T}\text{-}nf \ \Leftrightarrow\ \langle p, [\,]\rangle_{term} \to^*_{\mathrm{run}} \langle t\rangle_{nf}$$

2.3 A storeless natural semantics

The abstract machine displayed in Figure 4 is in 'defunctionalized form' [16, 33] in that its reduction contexts and the *cont*-transitions are the first-order representation of a function—or more precisely: of a continuation. Refunctionalization (i.e., the left inverse of defunctionalization [14]) yields an interpreter for graph reduction that is compositional and in continuation-passing style (CPS). The direct-style transformation (i.e., the left inverse of the CPS transformation [10]) yields an evaluation function that implements the natural semantics displayed in Figure 5. This natural semantics uses a tagged result in the form of a sum type. The left summand (tagged "*nf*") contains a normal form, and the right summand (tagged "*ide*") contains a triple grouping a reference that is needed, a context in progress, and a def-use chain for the reference. The semantics consists of four relations:

term-**relation:** Given a term, we recursively dive into the bodies of its nested let expressions until its innermost reference x, which is therefore needed. This reference then bubbles up as a tagged result in search of its declaration, while we incrementally construct a def-use chain for it.

$$\frac{t \Downarrow_{\text{term}} r \qquad (x,\, d,\, r) \Downarrow_{\text{cont}} r'}{\mathsf{let}\ x = d\ \mathsf{in}\ t \Downarrow_{\text{term}} r'} \qquad \frac{}{x \Downarrow_{\text{term}} ide(x,\, [\,],\, [\,])} \qquad \frac{}{(x,\, d,\, nf(t)) \Downarrow_{\text{cont}} nf(\mathsf{let}\ x = d\ \mathsf{in}\ t)}$$

$$\frac{(x_0,\, d,\, E_0,\, C) \Downarrow_{\text{den}} r}{(x_0,\, d,\, ide(x_0,\, E_0,\, C)) \Downarrow_{\text{cont}} r} \qquad \frac{}{(x,\, d,\, ide(x_0,\, E_0,\, C)) \Downarrow_{\text{cont}} ide(x_0,\, \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\, C)} \text{where } x \neq x_0$$

$$\frac{(x_0,\, d_0,\, E_0,\, C) \Downarrow_{\text{den}} r \qquad (E_1,\, r) \Downarrow_{\text{fill}} r' \qquad (x_1,\, I,\, r') \Downarrow_{\text{cont}} r''}{(x_1,\, I,\, E_1,\, \Pi_0^{i=0}(x_i,\, d_i,\, E_i) \cdot C) \Downarrow_{\text{den}} r''}$$

$$\frac{(x_3,\, d_1,\, (\mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = x_3\ \mathsf{in}\ E_0),\, C) \Downarrow_{\text{den}} r \qquad (E_2,\, r) \Downarrow_{\text{fill}} r' \qquad (x_2,\, K,\, r') \Downarrow_{\text{cont}} r''}{(x_2,\, K,\, E_2,\, \Pi_0^{i=1}(x_i,\, d_i,\, E_i) \cdot C) \Downarrow_{\text{den}} r''} \text{where } x_3 \text{ is fresh}$$

$$\frac{\begin{array}{r}(x_4,\, d_2,\, (\mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_5 = d_1\ \mathsf{in}\ \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_6 = d_0\ \mathsf{in}\ [\,]),\, (x_7,\, x_6,\, [\,]) \cdot (x_0,\, x_5\, x_6,\, E_0) \cdot C) \Downarrow_{\text{den}} r \\ (E_3,\, r) \Downarrow_{\text{fill}} r' \\ (x_3,\, S,\, r') \Downarrow_{\text{cont}} r''\end{array}}{(x_3,\, S,\, E_3,\, \Pi_0^{i=2}(x_i,\, d_i,\, E_i) \cdot C) \Downarrow_{\text{den}} r''} \text{where } x_4,\, x_5,\, x_6 \text{ and } x_7 \text{ are fresh}$$

$$\frac{}{(x_0,\, d_0,\, E_0,\, C) \Downarrow_{\text{den}} nf(\mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]])} \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \text{ and } \langle C,\, x_0\rangle_{chain} \Uparrow_{\text{rec}} C[x_0] \text{ and } \langle E,\, C[x_0]\rangle_{oi} \Uparrow_{\text{rec}} E_0[C[x_0]]$$

$$\frac{}{(x_0,\, x_1\, d_0,\, E_0,\, C) \Downarrow_{\text{den}} ide(x_1,\, [\,],\, (x_0,\, d_0,\, E_0) \cdot C)} \qquad \frac{(x_1,\, d_0,\, [\,],\, (x_0,\, d_1,\, E_0) \cdot C) \Downarrow_{\text{den}} r}{(x_0,\, d_0\, d_1,\, E_0,\, C) \Downarrow_{\text{den}} r} \text{where } x_1 \text{ is fresh and } d_0 \text{ is not a reference}$$

$$\frac{(x_0,\, t_0,\, E_0,\, C) \Downarrow_{\text{den}} r \qquad (x_1,\, d_1,\, r) \Downarrow_{\text{cont}} r'}{(x_0,\, \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\, E_0,\, C) \Downarrow_{\text{den}} r'} \qquad \frac{}{(x_0,\, x_1,\, E_0,\, C) \Downarrow_{\text{den}} ide(x_1,\, \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\, C)}$$

$$\frac{}{([\,],\, r) \Downarrow_{\text{fill}} r} \qquad \frac{(E,\, r) \Downarrow_{\text{fill}} r' \qquad (x,\, d,\, r') \Downarrow_{\text{cont}} r''}{(\mathsf{let}\ x = d\ \mathsf{in}\ E,\, r) \Downarrow_{\text{fill}} r''}$$

Fig. 5. Storeless natural semantics for combinatory graph reduction

cont-**relation:** Given a reference and a matching triple, we evaluate its definiens. Otherwise we retag the result and it bubbles up further.

den-**relation:** Having found the declaration of the reference that was needed, we check whether we have also found a redex. If so, we contract it and we pursue the next reference that is needed, which yields an intermediate result that bubbles up in the current delimited context with the *fill*-relation and yields another intermediate result that bubbles up with the *cont*-relation. Otherwise, a combinator is not fully applied or a new reference is needed. We then return a normal form or a triple extending the current def-use chain for this new reference.

fill-**relation:** Given a context and a tagged result, we recursively recompose it inside-out through the *cont*-relation.

Proposition 2 (full correctness). *For any program p,*

$$\langle p, [\,]\rangle_{term} \rightarrow^*_{\text{run}} \langle t\rangle_{nf} \quad \Leftrightarrow \quad p \Downarrow_{\text{term}} nf(t)$$

2.4 Summary and conclusion

Starting from a completely spelled out reduction semantics for combinatory terms, we have mechanically inter-derived a storeless abstract machine and a storeless natural semantics. Each of these semantic artifacts uses the same syntactic representations as the two others. The abstract machine and the natural semantics only differ in their representation and treatment of the congruence rules, which are determined by the reduction strategy: the natural semantics uses an implicit proof tree whereas the abstract machine uses an explicit evaluation context.

3 Preprocessing combinatory terms into term graphs

In Section 2, references are declared on demand in the reduction sequence. In this section, we factor out all possible such declarations for combinations into a preprocessing phase.

We start by restating the $(comb)$ and (S) axioms:

$$\begin{array}{lll} (comb') & \mathsf{let}\ x = d_0\, d_1\ \mathsf{in}\ E[C[x]] & \rightarrow \ \mathsf{let}\ x_0 = d_0\ \mathsf{in} \\ & & \quad \mathsf{let}\ x_1 = d_1\ \mathsf{in} \\ & & \quad \mathsf{let}\ x = x_0\, x_1\ \mathsf{in}\ E[C[x]] \\ & & \quad \text{where } d_0 \text{ is not a reference} \\ & & \quad \text{and } x_1 \text{ and } x_2 \text{ are fresh} \end{array}$$

In contrast to the $(comb)$ axiom, the restated axiom $(comb')$ declares references for both sides of a combination. Unlike in Section 2, there can thus be references to denotable terms whose denotation is not needed. In the same spirit, we restate the (S) axiom so that it declares references to both sides of any combination in the contractum:

$$
\begin{array}{lll}
(S') & \mathsf{let}\ x_3 = S\ \mathsf{in} & \to\ \mathsf{let}\ x_3 = S\ \mathsf{in} \\
& \quad E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} & \quad E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\
& \qquad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & \qquad \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\
& \qquad\quad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \qquad E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\
& \qquad\qquad E_0[C[x_0]]]]] & \qquad\quad \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\
& & \qquad\quad E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\
& & \qquad\qquad \mathsf{let}\ x_7 = x_4\, x_6\ \mathsf{in} \\
& & \qquad\qquad \mathsf{let}\ x_8 = x_5\, x_6\ \mathsf{in} \\
& & \qquad\qquad \mathsf{let}\ x_0 = x_7\, x_8\ \mathsf{in} \\
& & \qquad\qquad E_0[C[x_0]]]]] \\
& & \text{where } x_4,\ x_5,\ x_6,\ x_7 \text{ and } x_8 \text{ are fresh}
\end{array}
$$

The corresponding notion of reduction is $\mathcal{T}'$:

$$\mathcal{T}' = (I) \cup (K) \cup (S') \cup (comb') \cup (assoc) \cup (ref)$$

We split $\mathcal{T}'$ into two: a compile-time notion of reduction $\mathcal{C}$ and a run-time notion of reduction $\mathcal{R}$:

$$\mathcal{C} = (assoc) \cup (comb')$$
$$\mathcal{R} = (I) \cup (K) \cup (S') \cup (ref)$$

Each of $\mathcal{C}$ and $\mathcal{R}$ contain only left-linear axioms and no critical pairs: they are orthogonal and thus confluent [25]. Furthermore, $\mathcal{C}$ is strongly normalizing.

The $\mathcal{C}$-normal forms are contained within the following sub-grammar of terms:

$$\begin{array}{rcl} \mathsf{Denotable\ Term} \ni d & ::= & I \mid K \mid S \mid x\, x \mid x \\ \mathsf{Term} \ni t & ::= & \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \end{array}$$

In this grammar, only combinations of references are admitted and furthermore let expressions are completely flattened, in a way reminiscent of monadic normal forms [21, 22].

Proposition 3 (Preprocessing). *If* $t \twoheadrightarrow_{\mathcal{T}'} t' \ \wedge\ t' \in \mathcal{T}'\text{-}nf$ *then* $\exists t'' \in \mathcal{C}\text{-}nf\ .\ t \twoheadrightarrow_{\mathcal{C}} t'' \twoheadrightarrow_{\mathcal{R}} t'$.

Proof. Strong normalization of $\mathcal{C}$ *ensures the existence of* t''. *Confluence of* $\mathcal{T}'$ *gives* $t'' \twoheadrightarrow_{\mathcal{T}'} t'$. $\mathcal{R}$ *is closed over* $\mathcal{C}$*-nf. Thus, only* $\mathcal{R}$ *is needed in the reduction* $t'' \twoheadrightarrow_{\mathcal{R}} t'$.

We observe that a preprocessed term is a syntactic representation of a graph where every denotable term has been declared with a reference. Indeed it is straightforward to interpret preprocessed terms as term graphs:

Definition 9 (term graphs [7, Definition 4.2.6]). *A* term graph *is a tuple* $(N, lab, succ, r)$ *over a set of function symbols* F *where*

– *N is a set of unique node identifiers;*
– *$lab : N \to F$ is a labeling function mapping nodes to function symbols;*
– *$succ : N \to N^n$ is a successor function mapping nodes to an n-tuple of successor nodes for some natural number n; and*
– *$r \in N$ is the root of the term graph.*

Definition 10 (interpreting combinatory terms as term graphs). *For any preprocessed term, its term graph over the function symbols $F = \{I, K, S, A\}$ is defined as follows:*

– *N is the set of declared references in the term.*
– *lab is defined on the definiens of a reference: for a combinator, it yields the respective function symbols I, K or S; for a combination, it yields the application symbol A; and for a reference, it yields the result of applying lab to this reference, which in effect acts as an alias for a node.*[8]
– *$succ$ is defined on the definiens of references: for a combination, it yields the corresponding pair of references, and for everything else, the empty tuple.*
– *r is the innermost reference of the term.*

Using the interpretation of Definition 10, we can translate the contraction rules over combinatory terms to graph-rewriting rules [7, Section 4.4.4]. The translation of (I), (K) and (S') gives us rewriting rules with the side condition that the redex is *rooted*, meaning that there is a path from the root of the graph to the redex, which is the case here and is manifested by its def-use chain. Terms in our language are therefore a restricted form of term graphs: directed acyclic graphs with an ordering hierarchy imposed on *succ* by the scoping of nested let expressions. (In his PhD thesis [8], Blom refers to this property of term graphs as 'horizontal sharing.')

3.1 A reduction semantics

Here is the full definition of the syntax after preprocessing terms into $\mathcal{C}$-nf:

$$\begin{aligned}
\text{Program} \ni p &::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\
\text{Denotable Term} \ni d &::= I \mid K \mid S \mid x\,x \mid x \\
\text{Term} \ni t &::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \\
\text{Reduction Context} \ni E &::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\
\text{Def-use Chain} \ni C &::= [\,] \mid \mathsf{let}\ x = [\,]\,x\ \mathsf{in}\ E[C[x]]
\end{aligned}$$

Axioms: Figure 6 displays the axioms. Each of (I) and (K) is much as the corresponding axiom in Section 2, though we have specialized it with respect to the grammar of preprocessed terms. The (S) axiom is a further specialization of the (S') axiom. Specifically, since the right-hand side of any combination is known to be a reference, there is no need to introduce new let expressions to preserve sharing. As for the (ref) axiom, it is unchanged.

The notion of reduction on preprocessed terms is $\mathcal{G}$:

$$\mathcal{G} = (I) \cup (K) \cup (S) \cup (ref)$$

[8] This application is well behaved since terms are acyclic.

$$
\begin{array}{lll}
(I) & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in}\ E_0[C[x_0]]] & \rightarrow\ \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = y_0\ \mathsf{in}\ E_0[C[x_0]]] \\
(K) & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
& \qquad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} \\
& \qquad\quad E_0[C[x_0]]]] & \rightarrow\ \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
& & \qquad E_1[\mathsf{let}\ x_0 = y_1\ \mathsf{in} \\
& & \qquad\quad E_0[C[x_0]]]] \\
(S) & \qquad \mathsf{let}\ x_3 = S\ \mathsf{in} & \rightarrow\ \mathsf{let}\ x_3 = S\ \mathsf{in} \\
& \qquad E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} & \qquad E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
& \qquad\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} & \qquad\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
& \qquad\qquad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} & \qquad\qquad E_1[\mathsf{let}\ x_4 = y_2\, y_0\ \mathsf{in} \\
& \qquad\qquad\quad E_0[C[x_0]]]]] & \qquad\qquad\quad \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in} \\
& & \qquad\qquad\quad \mathsf{let}\ x_0 = x_4\, x_5\ \mathsf{in} \\
& & \qquad\qquad\quad E_0[C[x_0]]]]] \\
& & \qquad \text{where } x_4 \text{ and } x_5 \text{ are fresh} \\
(\mathit{ref}) & \qquad \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \rightarrow\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_1]]
\end{array}
$$

Fig. 6. Reduction semantics for combinatory graph reduction over preprocessed terms: axioms

Recompositions: The recompositions of contexts and def-use chains are defined in the same way as in Section 2.

Decomposition: Decomposition is much as in Section 2, though we have specialized it with respect to the grammar of preprocessed terms. Its definition is displayed in Figure 7.

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{G}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 11 (standard one-step reduction). *For any t,*

$$
t \mapsto_{\mathcal{G}} t'' \quad \mathit{if} \quad \begin{cases} \langle t,\, [\,]\rangle_{term} \downarrow^{*}_{\mathrm{dec}} \langle E,\, r\rangle_{dec} \\ (r, t') \in \mathcal{G} \\ \langle E,\, t'\rangle_{io} \Uparrow_{\mathrm{rec}} t'' \end{cases}
$$

One-step reduction is a partial function because the given term may already be in $\mathcal{G}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 12 (reduction-based evaluation). *For any program p,*

$$
p \mapsto^{*}_{\mathcal{G}} t \ \wedge\ t \in \mathcal{G}\text{-}nf
$$

Evaluation is a partial function because it may diverge.

$$
\begin{array}{rcl}
\langle \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ t_1,\ E\rangle_{term} & \downarrow_{\mathrm{dec}} & \langle t_1,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} & \downarrow_{\mathrm{dec}} & \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
\\
\langle [\,],\ t\rangle_{cont} & \downarrow_{\mathrm{dec}} & \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \downarrow_{\mathrm{dec}} & \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \downarrow_{\mathrm{dec}} & \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
 & & \quad \text{where } x \neq x_0 \\
\\
\langle x_1,\ I,\ E_1,\ \mathit{\Pi}_0^{i=0}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\,y_0\ \mathsf{in}\ E_0[C[x_0]]]\rangle_{dec} \\
 & & \quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_2,\ K,\ E_2,\ \mathit{\Pi}_0^{i=1}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\,y_1\ \mathsf{in} \\
 & & \qquad E_1[\mathsf{let}\ x_0 = x_1\,y_0\ \mathsf{in} \\
 & & \qquad\quad E_0[C[x_0]]]]\rangle_{dec} \\
 & & \quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_3,\ S,\ E_3,\ \mathit{\Pi}_0^{i=2}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\,y_2\ \mathsf{in} \\
 & & \qquad E_2[\mathsf{let}\ x_1 = x_2\,y_1\ \mathsf{in} \\
 & & \qquad\quad E_1[\mathsf{let}\ x_0 = x_1\,y_0\ \mathsf{in} \\
 & & \qquad\quad\quad E_0[C[x_0]]]]]\rangle_{dec} \\
 & & \quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
 & & \quad \text{where } d_0 = I \ \text{ and } |C| < 1 \\
 & & \qquad \text{or } d_0 = K \text{ and } |C| < 2 \\
 & & \qquad \text{or } d_0 = S \ \text{ and } |C| < 3 \\
 & & \quad \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\,y_0,\ E_0,\ C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ (x_1,\ [\,],\ (x_0,\ y_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} & \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
 & & \quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]
\end{array}
$$

Fig. 7. Reduction semantics for combinatory graph reduction over preprocessed terms: decomposition

3.2 A storeless abstract machine

The abstract machine is calculated as in Section 2.2 and displayed in Figure 8.

Proposition 4 (full correctness). *For any program p,*

$$p \mapsto_{\mathcal{G}}^{*} t \ \wedge\ t \in \mathcal{G}\text{-}nf \ \Leftrightarrow\ \langle p,\ [\,]\rangle_{term} \rightarrow_{\mathrm{run}}^{*} \langle t\rangle_{nf}$$

3.3 A storeless natural semantics

The natural semantics is calculated as in Section 2.3 and displayed in Figure 9.

$$
\begin{array}{rl}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} & \to_{\mathrm{run}}\ \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} & \to_{\mathrm{run}}\ \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
 & \\
\langle [\,],\ t\rangle_{cont} & \to_{\mathrm{run}}\ \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} & \to_{\mathrm{run}}\ \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
 & \\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \to_{\mathrm{run}}\ \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} & \to_{\mathrm{run}}\ \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
 & \quad \text{where } x \neq x_0 \\
 & \\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E_1 \circ_{io} \mathsf{let}\ x_1 = I\ \mathsf{in}\ E,\ (y_0,\ \mathsf{let}\ x_0 = y_0\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E_2 \circ_{io} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E,\ (y_1,\ E',\ C)\rangle_{cont} \\
 & \quad \text{where } E' = (\mathsf{let}\ x_1 = x_2\, x_1\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = y_1\ \mathsf{in}\ E_0) \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E_3 \circ_{io} \mathsf{let}\ x_3 = S\ \mathsf{in}\ E,\ (y_2,\ E',\ C')\rangle_{cont} \\
 & \quad \text{where } E' = (\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\ E_1) \\
 & \quad \text{and } C' = (x_4,\ y_0,\ \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in}\ [\,]) \cdot (x_0,\ x_5,\ E_0) \cdot C \\
 & \quad \text{and } x_4 \text{ and } x_5 \text{ are fresh} \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
 & \quad \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \\
 & \quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & \quad \text{and } \langle E,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\, y_0,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E,\ (x_1,\ [\,],\ (x_0,\ y_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} & \to_{\mathrm{run}}\ \langle E,\ (x_1,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\ C)\rangle_{cont}
\end{array}
$$

Fig. 8. Storeless abstract machine for combinatory graph reduction over preprocessed terms

$$\frac{t \Downarrow_{\mathrm{term}} r \qquad (x,\, d,\, r) \Downarrow_{\mathrm{cont}} r'}{\mathsf{let}\ x = d\ \mathsf{in}\ t \Downarrow_{\mathrm{term}} r'} \qquad \frac{}{x \Downarrow_{\mathrm{term}} ide(x,\, [\,],\, [\,])} \qquad \frac{}{(x,\, d,\, nf(t)) \Downarrow_{\mathrm{cont}} nf(\mathsf{let}\ x = d\ \mathsf{in}\ t)}$$

$$\frac{(x_0,\, d,\, E_0,\, C) \Downarrow_{\mathrm{den}} r}{(x_0,\, d,\, ide(x_0,\, E_0,\, C)) \Downarrow_{\mathrm{cont}} r} \qquad \frac{}{(x,\, d,\, ide(x_0,\, E_0,\, C)) \Downarrow_{\mathrm{cont}} ide(x_0,\, \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\, C)} \text{where } x \neq x_0$$

$$\frac{(E_1,\, ide(y_0,\, \mathsf{let}\ x_0 = y_0\ \mathsf{in}\ E_0,\, C)) \Downarrow_{\mathrm{fill}} r \qquad (x_1,\, I,\, r) \Downarrow_{\mathrm{cont}} r'}{(x_1,\, I,\, E_1,\, \Pi_0^{i=0}(x_i,\, y_i,\, E_i) \cdot C) \Downarrow_{\mathrm{den}} r'}$$

$$\frac{(E_2,\, ide(y_1,\, (\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = y_1\ \mathsf{in}\ E_0),\, C)) \Downarrow_{\mathrm{fill}} r \qquad (x_2,\, K,\, r) \Downarrow_{\mathrm{cont}} r'}{(x_2,\, K,\, E_2,\, \Pi_0^{i=1}(x_i,\, y_i,\, E_i) \cdot C) \Downarrow_{\mathrm{den}} r'}$$

$$\frac{\begin{array}{r}(E_3,\, ide(y_2,\, (\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\ E_1),\, (x_4,\, y_0,\, \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in}\ [\,]) \cdot (x_0,\, x_5,\, E_0) \cdot C)) \Downarrow_{\mathrm{fill}} r \\ (x_3,\, S,\, r) \Downarrow_{\mathrm{cont}} r'\end{array}}{(x_3,\, S,\, E_3,\, \Pi_0^{i=2}(x_i,\, y_i,\, E_i) \cdot C) \Downarrow_{\mathrm{den}} r'} \text{where } x_4 \text{ and } x_5 \text{ are fresh}$$

$$\frac{}{(x_0,\, d_0,\, E_0,\, C) \Downarrow_{\mathrm{den}} nf(\mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]), \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \text{ and } \langle C,\, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E,\, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]}$$

$$\frac{}{(x_0,\, x_1\, y_0,\, E_0,\, C) \Downarrow_{\mathrm{den}} ide(x_1,\, [\,],\, (x_0,\, y_0,\, E_0) \cdot C)} \qquad \frac{}{(x_0,\, x_1,\, E_0,\, C) \Downarrow_{\mathrm{den}} ide(x_1,\, \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\, C)}$$

$$\frac{}{([\,],\, r) \Downarrow_{\mathrm{fill}} r} \qquad \frac{(E,\, r) \Downarrow_{\mathrm{fill}} r' \qquad (x,\, d,\, r') \Downarrow_{\mathrm{cont}} r''}{(\mathsf{let}\ x = d\ \mathsf{in}\ E,\, r) \Downarrow_{\mathrm{fill}} r''}$$

Fig. 9. Storeless natural semantics for combinatory graph reduction over preprocessed terms

Proposition 5 (full correctness). *For any program p,*

$$\langle p,\, [\,]\rangle_{term} \rightarrow^{*}_{\mathrm{run}} \langle t\rangle_{nf} \quad \Leftrightarrow \quad p \Downarrow_{\mathrm{term}} \; nf(t)$$

3.4 Summary and conclusion

Starting from a completely spelled out reduction semantics for preprocessed combinatory term graphs, we have inter-derived a storeless abstract machine and a storeless natural semantics. As in Section 2, each of these semantic artifacts uses the same syntactic representations, and the abstract machine and the natural semantics only differ in their representation and treatment of the congruence rules.

4 Store-based combinatory graph reduction

In this section, we no longer represent graph vertices explicitly as let expressions, but implicitly as locations in a store:

$$\begin{aligned} \mathsf{Global\ Store} &\ni \sigma \\ \mathsf{Location} &\ni x,\, y \end{aligned}$$

In the storeless accounts of Sections 2 and 3, let expressions declare references to denotable terms. In the store-based account presented in this section, a global store maps locations to storable terms. Given a store σ, a location x and a storable term s, we write $\sigma[x := s]$ for the store σ' such that $\sigma'(x) = s$ and $\sigma'(x') = \sigma(x')$ for $x' \neq x$.

The syntax of Section 3 therefore specializes as follows:

$$\begin{aligned} \mathsf{Program} &\ni p ::= (x,\sigma\,) \\ \mathsf{Storable\ Term} &\ni s ::= I \mid K \mid S \mid x\,x \mid x \\ \mathsf{Ancestor\ Stack} &\ni a ::= [\,] \mid (x,x)\cdot a \end{aligned}$$

A program now pairs the root of a graph in a store with this store. Denotable terms have been replaced by storable terms. Terms and reduction contexts have been replaced by references in the store. Def-use chains have specialized into what Turner calls *ancestor stacks*. We write $|a|$ for the height of an ancestor stack a.

Translating the preprocessed terms of Section 3 to store-based terms is straightforward:

Definition 13 (let-expression based to store-based).

$$\begin{aligned} [\![\mathsf{let}\ x = d\ \mathsf{in}\ t]\!]_\sigma &= [\![t]\!]_{\sigma[x := [\![d]\!]']} \\ [\![x]\!]_\sigma &= (x,\sigma\,) \end{aligned}$$

$$\begin{aligned} [\![I]\!]' &= I \\ [\![K]\!]' &= K \\ [\![S]\!]' &= S \\ [\![x_0\,x_1]\!]' &= [\![x_0]\!]'\,[\![x_1]\!]' \\ [\![x]\!]' &= x \end{aligned}$$

$$
\begin{array}{lrcl}
(I) & (x_0,\ \sigma[x_1 := I][x_0 := x_1\, y_0]) & \to & (x_0,\ \sigma[x_1 := I][x_0 := y_0]) \\
(K) & (x_0,\ \sigma[x_2 := K][x_1 := x_2\, y_1][x_0 := x_1\, y_0]) & \to & (x_0,\ \sigma[x_2 := K][x_1 := x_2\, y_1][x_0 := y_1]) \\
(S) & (x_0,\ \sigma[x_3 := S] & \to & (x_0,\ \sigma[x_3 := S] \\
 & \quad [x_2 := x_3\, y_2] & & \quad\ [x_2 := x_3\, y_2] \\
 & \quad [x_1 := x_2\, y_1] & & \quad\ [x_1 := x_2\, y_1] \\
 & \quad [x_0 := x_1\, y_0]) & & \quad\ [x_4 := y_2\, y_0] \\
 & & & \quad\ [x_5 := y_1\, y_0] \\
 & & & \quad\ [x_0 := x_4\, x_5]) \\
 & & & \text{where } x_4 \text{ and } x_5 \text{ are fresh} \\
(ref_1) & (x_0,\ \sigma[x_0 := x_1]) & \to & (x_1,\ \sigma[x_0 := x_1]) \\
 & & & \text{where } x_0 \text{ is the graph root} \\
(ref_2) & (x_0,\ \sigma[x_0 := x_1]) & \to & (x_1,\ \sigma[x_0 := x_1][x := x_1\, y]) \\
 & & & \text{where } x \text{ is reachable from the graph root} \\
 & & & \text{and } \sigma(x) = x_0\, y \text{ for some } y
\end{array}
$$

Fig. 10. Reduction semantics for store-based combinatory graph reduction: axioms

This encoding maps the explicit representation of graph vertices as let expressions to implicit locations in a store.

4.1 A reduction semantics

Axioms: An axiom is of the form $(x,\sigma) \to (x', \sigma')$ where x and x' are the left and right root respectively. For such an axiom, a redex is a pair (x'', σ'') together with a renaming of locations defined by a structure-preserving function on storable terms, π, such that:

$$\pi(x) = x'' \text{ and } \forall y \in dom(\sigma)\ .\ \pi(\sigma(y)) = \sigma''(\pi(y)).$$

In words, the renaming must map the left root to the root of the redex, and any location in the store of the axiom must have a corresponding location in the store of the redex. As before, we write $\sigma[x := s]$ for a store mapping the location x to the storable term s.

The axioms are displayed in Figure 10. They assume that there is a path from the graph root to the redex root. This assumption mirrors the decomposition conditions in the axioms of Figure 6. Consequently, the reference axiom is split in two cases: one if the redex root is the graph root, corresponding to a decomposition into the empty def-use chain, and one if the redex root is not the graph root, corresponding to a decomposition into a non-empty def-use chain.

The notion of reduction on store-based terms is $\mathcal{H}$:

$$\mathcal{H} = (I) \cup (K) \cup (S) \cup (ref_1) \cup (ref_2)$$

$$
\begin{aligned}
\langle [\,],\, x,\sigma\,\rangle_{stack} &\downarrow_{\mathrm{dec}} \langle (x,\sigma\,)\rangle_{nf} \\
\langle (x_0,y_0)\cdot a,\, x_1,\, \sigma\rangle_{stack} &\downarrow_{\mathrm{dec}} \langle a,\, x_0,\, \sigma\rangle_{stack} \\
\\
\langle x_1,\, I,\, (x_0,y_0)\cdot a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec} \\
\langle x_2,\, K,\, (x_1,y_1)\cdot(x_0,y_0)\cdot a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec} \\
\langle x_3,\, S,\, (x_2,y_2)\cdot(x_1,y_1)\cdot(x_0,y_0)\cdot a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec} \\
\langle x_0,\, s,\, a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle a,\, x_0,\, \sigma\rangle_{stack} \\
&\qquad \text{where } s = I \ \text{ and } |a| < 1, \\
&\qquad\quad \text{or } s = K \text{ and } |a| < 2 \\
&\qquad\quad \text{or } s = S \ \text{ and } |a| < 3 \\
\langle x_0,\, x_1\, y_0,\, a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle x_1,\, \sigma(x_1),\, (x_0,y_0)\cdot a,\sigma\,\rangle_{sto} \\
\langle x_0,\, x_1,\, a,\sigma\,\rangle_{sto} &\downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec}
\end{aligned}
$$

Fig. 11. Reduction semantics for store-based combinatory graph reduction: decomposition

Recomposition: The recomposition of an ancestor stack with a store-based term relocates the root of the graph:

$$
\frac{}{\langle [\,],\, x,\sigma\,\rangle_{stack} \Uparrow_{\mathrm{rec}} (x,\sigma\,)} \qquad \frac{\langle a,\, x_0,\, \sigma\rangle_{stack} \Uparrow_{\mathrm{rec}} (x',\sigma')}{\langle (x_0,y_0)\cdot a,\, x,\sigma\,\rangle_{stack} \Uparrow_{\mathrm{rec}} (x',\sigma')}
$$

Decomposition: Decomposition is much as in Section 2 though we have further specialized it with respect to store-based graphs. The search previously done at return time is now done at call time. Starting from the root reference, x, we recursively search for a redex, incrementally constructing an ancestor stack for x. If we do not find any redex, the term is in $\mathcal{H}$-normal form. Figure 11 displays this search as a transition system:

Definition 14 (decomposition). *For any* $(x,\sigma\,)$*,*

$$
\langle x,\sigma\ (x),\, [\,],\, \sigma\rangle_{sto} \downarrow^{*}_{\mathrm{dec}} \begin{cases} \langle (x,\sigma\,)\rangle_{nf} & \textit{if } (x,\sigma\,) \in \mathcal{H}\textit{-nf} \\ \langle a,\, (x',\sigma')\rangle_{dec} & \textit{otherwise} \end{cases}
$$

where (x',σ') *is the left-most outermost redex in* $(x,\sigma\,)$ *and* a *is the ancestor stack from* x *to* x'*.*

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{H}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 15 (standard one-step reduction). *For any* $(x,\sigma\,)$*,*

$$
(x,\sigma\,) \mapsto_{\mathcal{H}} (x''',\sigma''') \quad \textit{if} \quad \begin{cases} \langle x,\sigma\ (x),\, [\,],\, \sigma\rangle_{sto} \downarrow^{*}_{\mathrm{dec}} \langle a,\, (x',\ \sigma')\rangle_{dec} \\ ((x',\ \sigma'),(x'',\ \sigma'')) \ \in \ \mathcal{H} \\ \quad \langle a,\, x'',\, \sigma''\rangle_{stack} \Uparrow_{\mathrm{rec}} (x''',\sigma''') \end{cases}
$$

$$
\begin{array}{rcl}
\langle [\,],\, x,\sigma\,\rangle_{stack} & \to_{\mathrm{run}} & \langle (x,\sigma\,)\rangle_{nf} \\
\langle (x_0,y_0)\cdot a,\, x_1,\, \sigma\rangle_{stack} & \to_{\mathrm{run}} & \langle a,\, x_0,\, \sigma\rangle_{stack} \\
 & & \\
\langle x_1,\, I,\, (x_0,y_0)\cdot a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle x_0,\, y_0,\, a,\sigma\ [x_0 := y_0]\rangle_{sto} \\
\langle x_2,\, K,\, (x_1,y_1)\cdot(x_0,y_0)\cdot a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle x_0,\, y_1,\, a,\sigma\ [x_0 := y_1]\rangle_{sto} \\
\langle x_3,\, S,\, (x_2,y_2)\cdot(x_1,y_1)\cdot(x_0,y_0)\cdot a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle y_2,\, \sigma'(y_2),\, (x_4,y_0)\cdot(x_0,y_5)\cdot a,\sigma\ '\rangle_{sto} \\
 & & \quad \text{where } \sigma' = \sigma[x_4 := y_2\, y_0] \\
 & & \qquad\qquad [x_5 := y_1\, y_0] \\
 & & \qquad\qquad [x_0 := x_4\, x_5] \\
 & & \quad \text{and } x_4 \text{ and } x_5 \text{ are fresh} \\
\langle x_0,\, s,\, a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle a,\, x_0,\, \sigma\rangle_{stack} \\
 & & \quad \text{where } s = I \ \text{ and } |a| < 1, \\
 & & \qquad \text{or } s = K \text{ and } |a| < 2 \\
 & & \qquad \text{or } s = S \ \text{ and } |a| < 3 \\
\langle x_0,\, x_1\, y_0,\, a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle x_1,\, \sigma(x_1),\, (x_0,y_0)\cdot a,\sigma\,\rangle_{sto} \\
\langle x_0,\, x_1,\, [\,],\, \sigma\rangle_{sto} & \to_{\mathrm{run}} & \langle x_1,\, \sigma(x_1),\, [\,],\, \sigma\rangle_{sto} \\
\langle x_0,\, x_1,\, (x,y)\cdot a,\sigma\,\rangle_{sto} & \to_{\mathrm{run}} & \langle x_1,\, \sigma'(x_1),\, (x,y)\cdot a,\sigma\ '\rangle_{sto} \\
 & & \quad \text{where } \sigma' = \sigma[x := x_1\, y]
\end{array}
$$

Fig. 12. Store-based abstract machine for combinatory graph reduction

One-step reduction is a partial function because the given term may already be in $\mathcal{H}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 16 (reduction-based evaluation). *For any program* $(x,\sigma\,)$*,*

$$(x,\sigma\,) \mapsto^*_{\mathcal{H}} (x',\sigma') \ \wedge\ (x',\sigma') \in \mathcal{H}\text{-}nf$$

Evaluation is a partial function because it may diverge.

4.2 A store-based abstract machine

The abstract machine is calculated as in Section 2.2. We display it in Figure 12. Its architecture is that of Turner's SK-reduction machine [37]: the left-ancestor stack is incrementally constructed at each combination; upon reaching a combinator, its arguments are found on top of the ancestor stack and a graph transformation takes place to rearrange them. In particular, our handling of stored locations coincides with Turner's indirection nodes. The only differences are that our machine accepts the partial application of combinators and that Turner's combinators are unboxed, which is an optimization.

Proposition 6 (full correctness). *For any program* $(x,\sigma\,)$*,*

$$(x,\sigma\,) \mapsto^*_{\mathcal{H}} (x',\sigma') \ \wedge\ (x',\sigma') \in \mathcal{H}\text{-}nf \ \Leftrightarrow\ \langle x,\sigma\ (x),\, [\,],\, \sigma\rangle_{sto} \to^*_{\mathrm{run}} \langle (x',\sigma')\rangle_{nf}$$

This machine is not in defunctionalized form and thus not ready for refunctionalization towards the functional representation of a store-based natural semantics à la Launchbury [27].

4.3 Summary and conclusion

Starting from a completely spelled out reduction semantics for combinatory term graphs in a store, we have inter-derived a store-based abstract machine. The structure of this store-based abstract machine coincides with that of Turner's SK-reduction machine.

5 The Y Combinator

In this section we briefly cover extending the semantics for the preprocessed terms of Section 3 with cyclic constructs. More precisely, we add a fixed-point combinator Y as traditionally specified by the equation $Yt = t(Yt)$. This equation exhibits two types of duplication:

1. the duplication of non-recursive subterms, in this case t; and
2. the duplication of locally recursive subterms, in this case Yt.

Both types of duplication can be avoided using graphs and the graphs can be represented syntactically. To syntactically capture the sharing of non-recursive subterms we can use let expressions as shown in the previous sections. To syntactically capture the sharing of locally recursive subterms we enrich the def-use chains with a letrec construct for local recursive declarations, letrec $x = d$ in t, where the denoted term, d, can refer to its reference, x, recursively. This "tying of the knot," to paraphrase Landin, is realized by the following axiom:

$$
\begin{array}{lll}
(Y) & \mathsf{let}\ x_1 = Y\ \mathsf{in} & \rightarrow\ \mathsf{let}\ x_1 = Y\ \mathsf{in} \\
& \quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} & \quad\ E_1[\mathsf{letrec}\ x_0 = y_0\, x_0 \\
& \qquad E_0[C[x_0]]] & \qquad\ \mathsf{in}\ E_0[C[x_0]]]
\end{array}
$$

Instead of containing a new subterm of the form Yt, the contractum recursively refers to itself through the recursive declaration letrec. The contractum thus shares the recursively defined subterm Yt.

Cyclic terms raise a new issue: naive contraction can bring references out of the scope of their definitions. A correct treatment is thus to λ-lift [18, 24] the cyclic definition so it can be referred to lexically in the contractum.

This situation arises in the (S) axiom. Consider the following redex and its contraction according to the (S) axiom of Figure 6:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0 \\
\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]]
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{let}\ x_4 = y_2\, x_0\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ x_5 = y_1\, x_0\ \mathsf{in} \\
\quad\quad\quad \mathsf{letrec}\ x_0 = x_4\, x_5 \\
\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]] \\
\text{where } x_4 \text{ and } x_5 \text{ are fresh}
\end{array}
$$

Here the contractum contains two occurrences of x_0 that are out of scope. By first λ-lifting the recursive binding, this scoping issue is solved:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0 \\
\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]]
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_6 = x_1\, x_6 \\
\quad\quad\quad \mathsf{in\ let}\ x_4 = y_2\, x_6\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ x_5 = y_1\, x_6\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ x_0 = x_4\, x_5\ \mathsf{in} \\
\quad\quad\quad\quad E_0[C[x_0]]]]] \\
\text{where } x_4,\ x_5 \text{ and } x_6 \text{ are fresh}
\end{array}
$$

This solution does maintain proper scoping, but it still suffers from the same duplication of locally recursive subterms that we started out with. If x_6 is needed, the same contraction will again take place instead of reusing the contractum just obtained. In other words, we would like to tie another knot.

From knot to Gordian knot: We further enrich the def-use chains with a letrec construct for local and mutually recursive declarations. The example from before can then be restated as follows:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0 \\
\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]]
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_4 = y_2\, x_0 \\
\quad\quad\quad\quad\quad\quad x_5 = y_1\, x_0 \\
\quad\quad\quad\quad\quad\quad x_0 = x_4\, x_5 \\
\quad\quad\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]] \\
\text{where } x_4 \text{ and } x_5 \text{ are fresh}
\end{array}
$$

Adding mutually recursive definitions makes it possible to share more subgraphs. Indeed, in his PhD thesis [8, Section 4.5], Blom describes how increasing the number of mutually recursive definitions strictly increases expressible sharing.

For a syntactic treatment of the cyclic λ-calculus using letrec, we refer the reader to Ariola and Felleisen's and to Nakata and Hagesawa's work [3, 30].

After specifying the syntactic treatment of cyclic programs, the story is once again compellingly simple. As in Section 2.2, an abstract machine can be calculated directly from the reduction semantics. If in defunctionalized form, a natural semantics can be further calculated from this machine as in Section 2.3. The resulting semantics are larger but the inter-derivations apply.

6 Related work

It has long been noticed that combinators make it possible to do without variables. For example, in the 1960's, Robinson outlined how this could be done to implement logics [36]. However, it took Turner to realize in the 1970's that combinatory graph reduction could be not only efficiently implementable, but also provided an efficient implementation for lazy functional languages [37]. Turner's work ignited a culture of implementation techniques in the 1980's [31], whose goal in retrospect can be characterized as designing efficient big-step graph reducers.

Due to the increased interest in graph reduction, Barendregt et al. developed term graphs and term graph rewriting [7]. Their work has since been used to model languages with sharing and to reason about program transformations in the presence of sharing [2, 20, 23, 26, 32]. Later work by Ariola and Klop provides an equational theory for term graph rewriting with cycles [5], a topic further developed by Blom in his PhD thesis [8] and by Nakata and Hagesawa since [30].

Over the 2000's, the first author and his students have investigated off-the-shelf program-transformation techniques for inter-deriving semantic artifacts [1, 16, 38]. The present work is an outgrowth of this investigation.

7 Conclusion and future work

Methodologically, mathematicians who wrote about their art (Godfrey H. Hardy, John E. Littlewood, Paul Halmos, Jacques Hadamard, George Pólya, Alexandre Gothendriek, and Donald E. Knuth for example) clearly describe how their research is typically structured in two stages: (1) an exploratory stage where they boldly move forward, discovering right and left, and (2) a descriptive stage where they retrace their steps and revisit their foray, verify it, structure it, and put it into narrative shape. As far as abstract machines are concerned, tradition has it to seek new semantic artifacts, which is characteristic of the first stage. Our work stems from this tradition, though by now it subscribes to the second stage as we field-test our inter-derivational tools.

The present article reports our field test of combinatory graph reduction. Our main result is that representing def-use chains using reduction contexts and let expressions, which, in retrospect, is at the heart of Ariola et al.'s syntactic theory of the call-by-need λ-calculus, also makes it possible to account for combinatory graph reduction. We have stated in complete detail two reduction semantics

and have inter-derived two storeless abstract machines and two storeless natural semantics. Interpreting denotable entities as storable ones in a global store, we have rediscovered David Turner's graph-reduction machine.

At this point of time, we are adding literals and strict arithmetic and logic functions, as well as garbage-collection rules such as the following one:

$$\mathsf{let}\ x = d\ \mathsf{in}\ t\ \rightarrow\ t \quad \text{if } x \text{ does not occur in } t$$

We are wondering which kind of garbage collector is fostered by the nested let expressions of the syntactic theories and also the extent to which its references are akin to Curry's apparent variables [9].

Acknowledgments: Thanks are due to Marìa Alpuente for her generous invitation to present this material at LOPSTR 2010. We are also grateful to Kenichi Asai, Mayer Goldberg, Steffen Daniel Jensen, Julia Lawall, and three anonymous reviewers for their comments on an earlier version of this article.

The first author dedicates this work to the memory of his *directeur de thèse*, Bernard Robinet (1941–2009), who introduced him to Combinatory Logic and contexts [35].

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.
2. Zena M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146(1&2):69–108, 1995.
3. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.
4. Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, January 1995. ACM Press.
5. Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3/4):207–240, 1996.
6. Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.
7. Henk P. Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In Jaco de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, number 259 in Lecture Notes in Computer Science, pages 141–158, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
8. Stefan Blom. *Term Graph Rewriting – Syntax and Semantics*. PhD thesis, Institute for Programming Research and Algorithmics, Vrije Universiteit, Amsterdam, The Netherlands, March 2001.

9. Haskell B. Curry. Apparent variables from the standpoint of Combinatory Logic. *Annals of Mathematics*, 34:381–404, 1933.
10. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).
11. Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, September 2008. ACM Press. Invited talk.
12. Olivier Danvy. From reduction-based to reduction-free normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.
13. Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.
14. Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04.
15. Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer.
16. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.
17. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.
18. Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 2004(1), July 2004. Available online at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/`. A preliminary version was presented at the Sixth International Symposium on Functional and Logic Programming (FLOPS 2002).
19. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Benjamin C. Pierce, editor, *Proceedings of the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 44, No. 1, pages 153–164, Savannah, GA, January 2009. ACM Press.
20. John R. W. Glauert, Richard Kennaway, and M. Ronan Sleep. Dactl: An experimental graph rewriting language. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Proceedings*, volume 532 of *Lecture Notes in Computer Science*, pages 378–395, Bremen, Germany, March 1990. Springer.

21. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.
22. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997.
23. Alan Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 82–91, Paris, France, July 1994. IEEE Computer Society Press.
24. Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.
25. Jan W. Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.
26. Pieter W. M. Koopman. *Functional Programs as Executable Specifications*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1990.
27. John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.
28. Julia L. Lawall and Harry G. Mairson. On global dynamics of optimal graph reduction. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 32, No. 8, pages 188–195, Amsterdam, The Netherlands, June 1997. ACM Press.
29. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
30. Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.
31. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.
32. Marinus J. Plasmeijer and Marko C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
33. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [34].
34. John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
35. Bernard Robinet. *Contribution à l'étude de réalités informatiques*. Thèse d'état, Université Pierre et Marie Curie (Paris VI), Paris, France, May 1974.
36. John A. Robinson. A note on mechanizing higher order logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 5, pages 123–133. Edinburgh University Press, 1969.
37. David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.
38. Ian Zerny. On graph rewriting, reduction and evaluation. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Trends in Functional Programming Volume 10*, Komárno, Slovakia, June 2009. Intellect Books. Granted the best student-paper award of TFP 2009. To appear.

Analysis of Air Traffic Track Data with the AutoBayes Synthesis System

Johann Schumann[1], Karen Cate[2], and Alan Lee[3]

[1] SGT, Inc./ NASA Ames, Moffett Field, CA 94035, Johann.M.Schumann@nasa.gov
[2] NASA Ames, Moffett Field, CA 94035-0001, karen.tung@nasa.gov
[3] NASA Ames, Moffett Field, CA 94035-0001, alan.g.lee@nasa.gov

Abstract. The Next Generation Air Traffic System (NGATS) is aiming to provide substantial computer support for the air traffic controllers. Algorithms for the accurate prediction of aircraft movements are of central importance for such software systems but trajectory prediction has to work reliably in the presence of unknown parameters and uncertainties. We are using the AutoBayes program synthesis system to generate customized data analysis algorithms that process large sets of aircraft radar track data in order to estimate parameters and uncertainties. In this paper, we present, how the tasks of finding structure in track data, estimation of important parameters in climb trajectories, and the detection of continuous descent approaches can be accomplished with compact task-specific AutoBayes specifications. We present an overview of the AutoBayes architecture and describe, how its schema-based approach generates customized analysis algorithms, documented C/C++ code, and detailed mathematical derivations. Results of experiments with actual air traffic control data are discussed.

1 Introduction

Commercial Air Traffic Control (ATC) has coped with the strongly increasing volume of commercial passenger and freight air traffic that causes more and more crowded airways, delays, and canceled flights. The current ATC systems rely on radar data to measure the position of the aircraft; the communication between air traffic controllers and the pilots is handled via voice communication. The Next Generation Air Traffic System (NGATS) is aiming to provide substantial computer support for the air traffic controllers in the form of alert and advisory systems as well pave the way for computerized data links between ATC and the aircraft itself.

The NASA Ames Research Center has developed the software system CTAS (Center Tracon Advisory System) [1,2], which is used as a research and development platform for many projects, e.g., the automatic recognition/prediction of separation violations, or the En-route Descent Advisory system (EDA) [3].

A central software component of the CTAS system is the Trajectory Synthesizer (TS) [4]. Based upon the current state of the aircraft (position, altitude, heading, speed), the current atmospheric data like wind speed and direction,

and the intent (e.g., flight plan), the TS will calculate a prediction of the aircraft's future movements (trajectory). The TS utilizes aerodynamic equations of motion, weather forecasts, aircraft performance models, as well the intent of the current flight segment (e.g., "climb with constant speed to 30,000ft") to generate these predictions. It is obvious that the predictions of the trajectory synthesizer should be as accurate as possible, because many other ATC algorithms use the TS and base their decision upon the predicted trajectories. On the other hand, the accuracy of a trajectory prediction depends on many factors of uncertainty: weather forecast, surveillance data (radar, ADS-B), intent (controller instruction, pilot procedures), pilot inputs (delays in reaction and erroneous pilot actions), navigation, as well as aircraft performance modeling, all of which are often unknown.

In order to obtain realistic estimates for unknown parameters, data mining and analysis can be performed on large data sets of actual aircraft trajectories. All movements of commercial aircraft are registered by land-based radar systems and recorded approximately every 12 seconds. Thus huge data sets become available, even for a restricted region or even a single airport. Such trajectory data can also be used for testing the TS software: how close is the predicted trajectory to the actual one, compared after the fact. Such tests can reveal strength and weaknesses of the TS algorithm. However, there is a catch: almost all of the recorded trajectories comprise nominal flights (e.g., a straight overflight or a standard approach). Thus a high test coverage can only be obtained when an infeasible number of trajectories are exercised. For testing purposes, therefore, data sets with a well-balanced mixture of nominal and off-nominal trajectories are desirable. Extracting such a set from a huge set of actual air traffic data again is a data mining task.

In this paper, we describe, how the AutoBayes program synthesis system can be used for the estimation of unknown parameters and the extraction of a wide variety of flight scenarios. The AutoBayes system [5,6] is a schema-based synthesis system, which takes a high-level statistical model as its input. From that, AutoBayes generates a customized data analysis algorithm and produces C or C++ code, which can be directly invoked from Matlab or Octave to actually process the data. The AutoBayes system is well suited for mining of ATC data, because each different mining task requires a different statistical model, which would require the implementation of specific algorithms or existing code and libraries must be adapted accordingly—usually a very time-consuming task. The ability of AutoBayes to quickly synthesize customized algorithms from compact declarative specifications facilitates a quick exploration of different models.

AutoBayes has been used to support data analysis tasks on sets of actual aircraft trajectories. We will describe AutoBayes models for the unsupervised clustering of track data based upon sets of features. Such structuring of a large set of aircraft movements can be used to isolate specific trajectories of interest (e.g., some unusual flight profile), or to find representative trajectories in large sets of very similar trajectories. We will also report on the use of AutoBayes

models for time series analysis to estimate elements of the flight profile such as the CAS-mach speed transition, as well as for the detection of continuous descent approaches (CDA). For each of the applications, a short, fully declarative statistical specification of the problem was given to the AutoBayes program synthesis system, which in turn generated customized C/C++ algorithms, specifically tailored toward the analysis problem at hand.

There are several approaches in the literature for the automated analysis of air traffic control data using different statistical approaches. For example, [7] uses clustering algorithms to separate out nominal trajectories from those, which do not follow a usual profile and thus could be a potential source for problems. [8] performs statistical time series analysis on air traffic data.

For our analysis, we could have directly used one of the available EM-implementations, like Autoclass [9], EMMIX [10], MCLUST [11], or WEKA [12]. However, such tools are usually designed for Gaussian distributed data only and only allow little customization. Refining the statistical model (e.g., by incorporating other probability distributions for certain variables or to introduce domain knowledge), the analysis algorithms need to be modified substantially for each problem variant, making experimentation a time-consuming and error-prone undertaking with such tools.

The rest of this paper is structured as follows: In Section 2, we give an overview of the AutoBayes program synthesis system and its architecture. Section 3 discusses how AutoBayes has been used to generate clustering algorithms for the analysis of aircraft track data. We discuss the entire AutoBayes specification, describe, how AutoBayes automatically derives solutions for clustering of non-Gaussian distributions and present experimental results. In Section 4, we describe, how change-point time series models, have been used to estimate parameters or detect profiles. Section 5 summarizes our approach and concludes.

2 The AutoBayes Program Synthesis System

AutoBayes [5,6] is a fully automatic program synthesis system that generates efficient and documented C/C++ code from abstract statistical model specifications. Developed at NASA Ames, AutoBayes is implemented in approximately 90,000 lines of documented SWI Prolog (http://www.swi-prolog.org) code. From the outside, AutoBayes looks similar to a compiler for a very high-level programming language (Figure 1): it takes an abstract problem specification in the form of a (Bayesian) statistical model and translates it into executable C/C++ code that can be called from Matlab or Octave.

Once, a specification (for examples see Sections 3 and 4) has been converted into an internal representation by the input parser, a Bayesian network [13], representing the essential information of the specification is created. The synthesis kernel uses a schema-based approach to generate a customized algorithm that solves the statistical task at hand in an imperative intermediate language. This

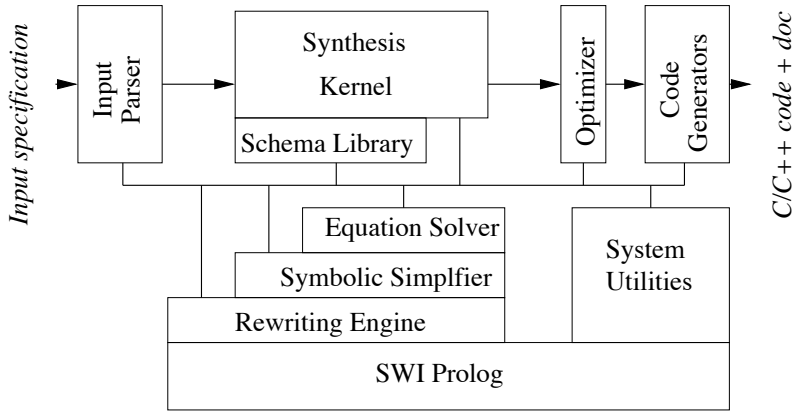


Fig. 1. The AutoBayes system architecture

program is then optimized and the code generator finally produces the desired code (currently C++ for Octave, C for Matlab, or stand-alone C).

AutoBayes uses a schema-based approach, because of the large and complex synthesis tasks, for which pure deductive synthesis (e.g., [14,15,16]) would not scale up. Each schema inside the schema library consists of applicability constraints and a template (parameterized code fragment with holes, which are Prolog variables). When a schema is applicable for the task at hand, AutoBayes tries to instantiate these parameters by direct calculations or by recursive calls to other schemas, which in turn solve the occurring subtasks. Using this mechanism, we not only generate code, but also comments and mathematical derivations, which can be automatically typeset in LaTeX (for an example see the appendix). All schemas are implemented as Prolog clauses; for guiding the search we use Prolog's backtracking mechanism and the constraints in the schemas. The constraints are formulated as conditions over the Bayesian network or as properties about the underlying model.

The schema library is organized in layers: top-level schemas comprise Bayesian network decomposition schemas, which are based upon independence theorems for Bayesian networks [13] as well as other statistical transformations. On the next layer contains schemas with statistical algorithm skeletons, like, for example the EM algorithm (expectation maximization) [17] or k-means for nearest-neighbor clustering. The bottom layer contains calls to symbolic solvers and standard numeric optimization methods, for example a Nelder-Mead simplex algorithm [18].

The schema-based synthesis system heavily relies on a powerful symbolic system. Based upon a small rewriting engine implemented in PROLOG, a large set of symbolic simplification rules, support for symbolic differentiation, as well as a number of symbolic equation solvers has been implemented. For details on the symbolic system see, e.g., [19].

3 Multivariate Clustering of Aircraft Track Data

The aircraft track data set used for this analysis is derived from radar track data. Every 12 seconds, position, altitude, heading, speeds, as well as numerous other data are recorded for each aircraft flying though a specific sector in the air space. Thus, a recording of an air traffic control center for one day usually contains the track data of thousands of aircraft; the data for each aircraft are stored as time series data. Additionally, data about wind speed and wind direction at different altitudes have been provided.

We are interested into automatically finding structure in these large data sets. Obvious categories for flight scenarios are departures and ascends, descents, or high altitude overflights. However, there is a multitude of other categories, which cannot be described as easily. We have been using multivariate clustering with AutoBayes to find such categories. As we did not want to perform a detailed time series analysis, we described each trajectory by a set of features, which were calculated from the track data (see Section 3.2) and used the feature vector as the input for multivariate clustering.

3.1 AutoBayes Clustering Models

Clustering problems can be specified as statistical mixture problems. Here, the statistical distribution or probability density function (PDF) for each feature must be known. In our case, several variables have discrete values (e.g., booleans or enumeration types for the different aircraft models), which have a discrete distribution; continuous variables include distances, times, and angles. Most clustering algorithms and tools make the assumption that all (continuous and discrete) variables are Gaussian (normal) distributed. For variables, which have a different PDF, Gaussian noise is added to the data in order to obtain a Gaussian-like distribution. However, such a model can be statistically very inaccurate, in particular, when dealing with angles. A noisy measurement of an angle close to 0° would, when considered normal distributed, yield (incorrectly) two classes with means around 0° and 360°. If arbitrary values for angles can occur, an angular rotation does not help either.

With AutoBayes, the user can directly specify customized mixture models, which correctly deal with variables having different probability distributions. Listing 1.1 shows a complete AutoBayes specification for such a clustering task. For space reasons, we focus on just three variables: heading (an angle), altitude (Gaussian distributed), and the type of the aircraft, which has discrete values. The actual string values (e.g., "B747") are converted into an enumeration type during preprocessing.

The first lines of Listing 1.1 define the model name and the global declarations for this problem. The constants N and C are instantiated when the generated code is executed, but they are assumed to be constant within the algorithm. Constraints are given after the **where** keyword. Lines 6–12 declare all distribution parameters for the statistical variables, θ_0, m for the von Mises distribution for the heading hd, μ, and σ for the altitude and ρ for the aircraft

type (e.g., $\rho_0 = \mathbf{P}(x = "B777")$). Line 12 states that each data point actually belongs to one of the specified types of aircraft. Since different classes will have different parameters, these variables are vectors over the classes. Note that all indexing is 0-based like in C. Φ is the unknown class frequency (Line 14), and c is the most likely class assignment for each aircraft track, which will be calculated and returned by the algorithm (Line 15). Statistically speaking, c is discrete distributed with probabilities Φ as reflected Line 17. Line 15 states that Φ is indeed a probability vector and has to add up to 1.

```
1  model ac_track as 'AC TRACK ANALYSIS'.
2
3  const nat N as 'NR. OF DATA POINTS'. where 0 < N.
4  const nat C as 'NR. OF CLASSES'. where 0 < C. where C << N.
5
6  double theta0_hd(0..C-1).    % THETA_0 FOR ANGLE
7  double m_hd(0..C-1).         % M FOR ANGLE
8  double mu_alt(0..C-1).       % MEAN, SIGMA FOR ALT
9  double sigma_alt(0..C-1). where 0 < sigma_alt(_).
10      % AC_TYPE: {"B777", "B737", "B755", "OTHER"}
11 double rho_AC_type(0..3, 0..C-1).
12     where 0 = sum(_i := 0..3, rho_AC_type(_i,_))-1.
13
14 double phi(0..C-1) as 'CLASS FREQUENCY'.
15         where 0 = sum(_i := 0 .. C-1, phi(_i))-1.
16 output double c(0..N-1) as 'CLASS ASSIGNMENT VECTOR'.
17     c(_) ~ discrete(vector(_i := 0 .. C-1, phi(_i))).
18
19 data double hd(0..N-1).
20     hd(_i) ~ vonmises1(theta0_hd(c(_i)),m_hd(c(_i))).
21 data double alt(0..N-1).
22     alt(_i) ~ gauss(mu_alt(c(_i)), sigma_alt(c(_i))).
23 data nat AC_type(0..N-1).
24     AC_type(_i) ~ discrete(vector(_j := 0..3,
25                                   rho_AC_type(_j,c(_i)))).
26
27 max pr({hd, alt, AC_type} |
28    {phi,theta0_hd,m_hd,mu_alt,sigma_alt,rho_AC_type})
29  for {phi,theta0_hd,m_hd,mu_alt,sigma_alt,rho_AC_type}.
```

Listing 1.1. AUTOBAYES specification for aircraft track clustering analysis

Lines 19–25 declare the data vectors and their respective distributions using the class-indexed parameters defined above. Finally, the goal statement (Lines 27–29) triggers the AUTOBAYES synthesis: maximize the probability of the data, given the distribution parameters for all distribution parameters. The solution to this task is a set of estimated parameters $(\phi, \theta_0, m, \mu, \sigma, \rho)$, which most likely explain the data. Note that the entire specification is purely declarative.

From this specification, AutoBayes generates 1032 lines of documented C code with a Matlab Mex interface in less than 5 seconds on a Mac book. In a first step, AutoBayes generates a Bayesian network, describing the specification is generated. Figure 2 shows the autogenerated graph. Shaded nodes are known variables (declared as **data**). Boxes around groups of nodes indicate that these variables are identically indexed (iid) over the number of classes C or the number of tracks N. As usual, directed arcs show the interdependency between the nodes.

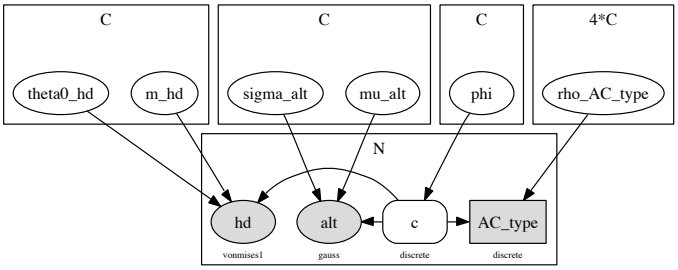


Fig. 2. Autogenerated Bayesian network for clustering specification (Listing 1.1)

Then, schemas are invoked, which detect that the problem can be solved using an iterative clustering algorithm like expectation maximisation (EM, [17]), or k-means [20]. By default, EM is selected. This algorithm schema breaks down the model and generates new subproblems, which must be solved by the recursive application of schemas. The code, which is produced by the schemas is used to assemble all parts of the EM algorithm. For our example, 4 different subproblems are produced: $\max \mathbf{P}(c \mid \phi)$ for ϕ, $\max \mathbf{P}(AC\_type \mid c, \rho_{AC})$ for ρ_{AC}, $\max \mathbf{P}(alt \mid c, \mu_{alt}, \sigma_{alt})$ for μ_{alt}, σ_{alt}, and $\max \mathbf{P}(hd \mid c, \mu_{hd}, \theta^0_{hd})$ for μ_{hd}, θ^0_{hd}.

The first subproblem can be solved symbolically using a Lagrange multiplier (for details see [5]). Each of the remaining subproblems can be solved independently. The optimization of the probability terms first triggers schemas, which replace the conditional probability by products and then attempt to optimize the atomic probability expressions. This is done in the text-book way by setting the partial derivatives to zero and solving for the unknown parameters. The appendix shows the detailed (autogenerated) derivation for the angle variable *hd*. For a detailed derivation for the Gaussian distributed variable see [5]; for the discrete variable see [6].

After all symbolic derivations have been performed, the top-level schema pulls together the individual pieces of the algorithm and, after several optimization and transformation steps, C code is generated. Please note that all schemas are independent of the specification and are not specifically tailored toward a given probability density. For each PDF, only symbolic expressions for the classical defining properties have to be provided. They can be taken directly out of a textbook.

3.2 Experiments and Results

For our clustering experiments, we used data sets containing flights around one large airport (DFW) over a period of between one and seven days. These data sets contained between a few hundred to more than ten thousand trajectories. In a first step, we extracted features from each aircraft track, so that each aircraft track was represented by a vector of features. Features included beginning or end coordinates for a track, overall changes in altitude, changes in headings, and much more. The features were kept as generic as possible but were selected to describe meaningful trajectories. For example, a low value of overall altitude changes (combined with a high altitude) can indicate a high altitude overflight. A large amount of altitude change can, when combined with other features (e.g., location of the trajectory start or end), describe take-off or landing scenarios.

A large number of customized features can be defined. They can include the straight-forward combination of information extracted from the time series, but can also include information extracted through additional preprocessing. For example, specific kinds of trajectories can (e.g., climb, holding, landing, overflight). Also distances from reference trajectories or lists of way-points can be used. The set of selected features spans a multi-dimensional hybrid space, as different features can have different probability distributions. Then, a task-specific AutoBayes model is set up and customized code is generated.

Figure 3 shows results that have been obtained using simple sets of features. In the first two panels, features included altitude, mean calibrated air speed (CAS), as well as the overall change in altitude. Trajectories belonging to different classes are shown in shades of grey. The two selected classes show low-speed approaches close to the airport (black) as compared to high altitude passages shown in grey. Panel (B) show climb approaches (grey), which characterized by increasing altitude and constant or increasing speed. The main directions of the departing aircraft, depending on the orientation of the runways can be seen clearly. Panels (C) and (D) shows members of a class of trajectories, which include many turns. Typically, navigation among way-points and holding patterns belong to this group (C). Panel (D) shows one selected trajectory, which was put into a separate class due to its many changes in heading. That flight most probably was a surveillance aircraft or a helicopter. The dots represent the individual radar measurements.

4 Change-point Detection Models

4.1 The CAS/mach Transition

Most climb procedures for commercial aircraft utilize a climb Calibrated Air Speed (CAS) and a climb mach speed. At low altitudes, the aircraft flies with the constant climb CAS measured in knots. As the aircraft ascends it reaches a particular altitude where the climb CAS is equivalent to the climb mach. This is the CAS-mach transition altitude. At this point the speed to be maintained

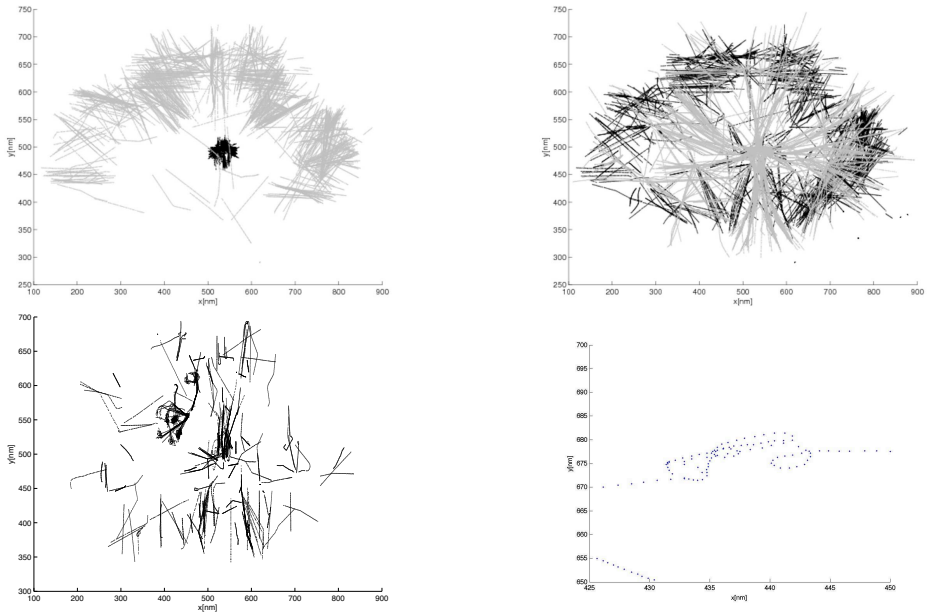


Fig. 3. Clustering results of aircraft track data

transitions from the climb CAS to the climb mach. Once the aircraft has reached its final altitude, it will continue to measure its speed in mach.

Such climb profiles are usually executed by the flight management system (FMS). The pilot has to enter the final altitude and speed (in mach) and the FMS will execute the appropriate profile. The altitude, at which the FMS performs the transition is usually kept constant. However, this parameter is not published and can vary between different air carriers. For an accurate prediction of trajectories as done by the CTAS TS software, however, good knowledge about these transition points is important.

4.2 Change-point Detection with AutoBayes

The calibrated air speed v_{CAS} and the mach number v_{mach} for such a climb scenario can be written as

$$v_{CAS} = \begin{cases} v^0_{CAS} & t \leq t_T \\ v^0_{CAS} - \Delta_C * (t - t_T) & t > t_T \end{cases}$$

$$v_{mach} = \begin{cases} v^0_{mach} + \Delta_m * (t_T - t) & t \leq t_T \\ v^0_{mach} & t > t_T \end{cases}$$

where t_T is the time when the flight regime changes (change-point). v^0_{mach}, v^0_{CAS} denote the constant mach number and CAS speed, respectively, and Δ_C, Δ_m

describe the slope of the speed changes. Prior to the transition, the CAS is kept constant, whereas the mach number increases linearly. This is due to the decreasing air pressure at high altitudes, which lowers the speed of sound. After this transition point, the mach number is kept constant for the remainder of the climb. Again, due to decreasing air pressure, the CAS is now decreasing.

Of course, all data are highly noisy, because turbulence, radar inaccuracies, and pilot maneuvers can perturb the data. Therefore, a statistical approach for finding this choice-point is needed. In our case, we simply assume that all measurements are Gaussian distributed, for example, $v_{CAS} = N(\mu_{vCAS}, \sigma_{vCAS})$. Then the problem of finding the best change point can be easily formulated as a statistical problem of change detection (e.g., [21,22]). Statistical text books usually provide examples and algorithms for just one variable. So, a manual development of the change-point detection algorithm and its implementation would be non-trivial and time-consuming.

In AutoBayes, we directly can specify our problem. Listing 1.2 contains the entire AutoBayes specification for this problem; AutoBayes then generates 684 lines of commented C code within a few seconds.

```
1  model climb_transition.
2  const nat n.  where 0 < n.
3  nat t_T as 'Transition time'. where t_T in 3..n-3.
4  double v_mach_0. double d_mach.
5  double v_CAS_0. double d_CAS.
6  const double sigma_sq.  where 0 < sigma_sq.
7  data double cas(0..n-1) as 'time series data: air speed'.
8  data double mach(0..n-1) as 'time series data: mach'.
9
10 cas(T) ~  gauss( cond(T < t_T,
11                       v_CAS_0,
12                       v_CAS_0 - (I-t_T)*d_CAS
13                  ), sqrt(sigma_sq)).
14 mach(T) ~ gauss( cond(T < t_T,
15                       v_mach_0 - (T-t_T)*d_mach,
16                       v_mach_0
17                  ), sqrt(sigma_sq)).
18 max pr({mach, cas}|{d_mach, v_mach_0, v_CAS_0, d_CAS, t_T})
19          for {v_mach_0, d_mach, v_CAS_0, d_CAS, t_T}.
```

Listing 1.2. AutoBayes specification for CAS-mach transition

Lines 1–6 declare model name and all constants and unknown parameters. Here, we assume that the uncertainty σ^2 is known, but all other parameters must be estimated. Lines 8–9 declare the known data vectors, which are time-series data here. Lines 10–17 are the core of the specification, where the probability distributions of each of the statistical variables are declared. This formulation is directly derived from the formulas shown above and uses C-style conditional expressions (**cond**(E,T,F) corresponds to E?T:F). Finally, lines 18–19 contains the goal statement.

Figure 4 shows results for three different climb scenarios: the mach number (top panel), the calibrated air speed (CAS, middle panel), and the altitude over time. The thin curves show actual aircraft data. The lines indicate the linear approximations. The location of the change point and the other unknown parameters have been estimated closely.

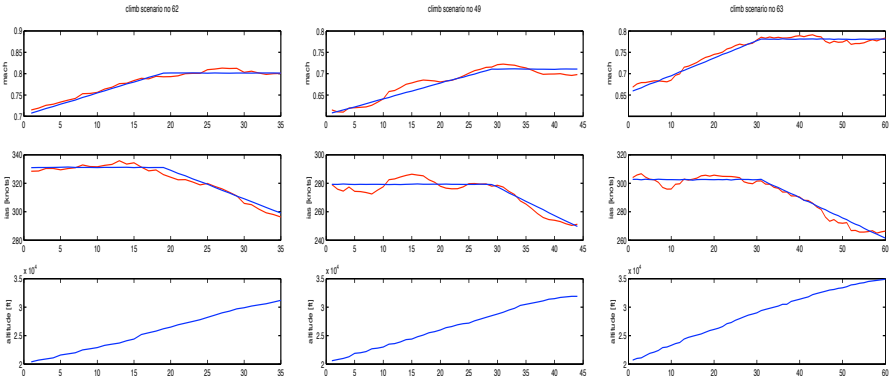


Fig. 4. Results of analysis with AutoBayes for three climb scenarios.

In a further analysis step, we analyzed more than 10,000 climb scenarios and recorded the altitude and speed at which the estimated CAS-mach transition took place. Figure 5 (left) shows the likelihood of the transition in an altitude over mach coordinate system as a set of contour lines (darker areas indicate larger likelihood). Two major transition altitudes could be detected, one at approximately 26,000ft, the other at around 31,000ft.

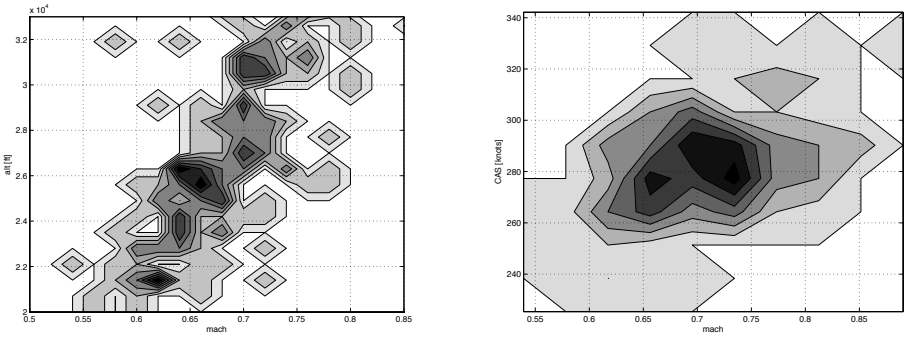


Fig. 5. Likelihood of transition point in an altitude over mach coordinate system (left) and in a CAS over mach coordinate system (right).

Figure 5 (right) shows the results of the analysis displayed as the likelihood of the transition in a CAS over mach coordinate system. This figure indicates that in most trajectories the climbs are initially performed with a CAS of 275 knots. Then the climb is continued most likely with mach 0.78 or mach 0.7. These two panels only show data for the B737 aircraft. Results for other types of aircraft show a similar behavior, but with different parameters.

4.3 CDA Detection

A continuous descent approach (CDA) is a flight profile where the aircraft continuously descends without any level-off segments. The benefits of this approach over a standard approach are a reduction in noise, emissions, fuel consumption, and flight time. For the detection of CDA-like scenarios, another AutoBayes change-point model has been used. One of the main characteristics of a CDA descent is a long stretch of almost constant vertical speed. In the data we analyzed [23], that speed was around -2,000ft/min. Figure 6 shows the horizontal profile in x-y coordinates and the altitude during the descent. Note that the long linear stretch during the descent (from appr. 35,000ft to appr. 12,000ft) indicates a constant descent speed.

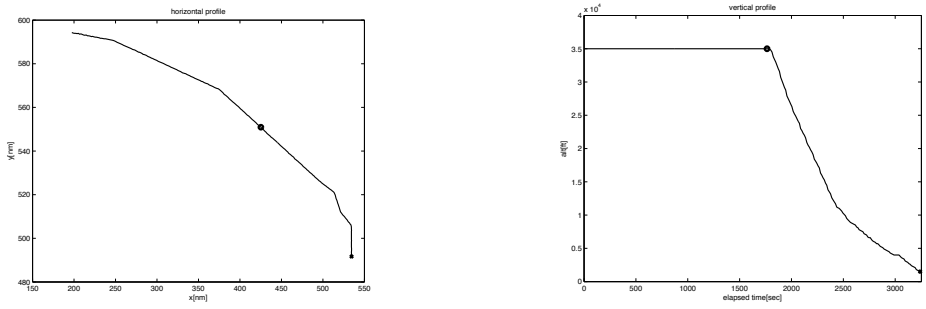


Fig. 6. Horizontal and vertical profile of a CDA approach. Start of descent and meter fix at 4,000ft are marked by a dot.

The AutoBayes specification of a CDA model entirely focuses on the time-series representing the vertical speed v_{vert} and has three distinct phases. In Phase 1, the descent has not yet started. The values of v_{vert} are around zero, but very indeterminate. Phase 2 comprises the CDA phase. Here, we assume that the vertical speed is Gaussian distributed around $\bar{v}_{cda}$ (in our case -2,000ft/min) with a reasonably σ^2 to accommodate for the noise. In Phase 3, again, only very vague assumptions about the vertical speed can be made. The time-points t_{12} and t_{23} represent the transitions. Formally, we have

$$v_{vert} = \begin{cases} N(\bar{v}, \sigma^2_{high}) & \text{for } t < t_{12} \\ N(\bar{v}_{cda}, \sigma^2_{low}) & \text{for } t_{12} \leq t < t_{23} \\ N(\bar{v}, \sigma^2_{high}) & \text{for } t_{23} \leq t \end{cases}$$

Figure 6 shows the vertical speed of the aircraft for one CDA approach. The different phases of the descent are marked. The thick horizontal lines show mean speeds for each segment ($\bar{v}, \bar{v}_{cda}$); the dashed lines show $\pm\sigma^2$ values around the means. Note that the actual vertical speed stays within these boundaries. It is obvious that this model is very primitive, yet it can capture the major vertical speed characteristics of a CDA approach. Listing 1.3 shows an excerpt of the AutoBayes specification containing the distribution of the time-series data and the goal statement. The rest of the specification is very similar to Listing 1.2.

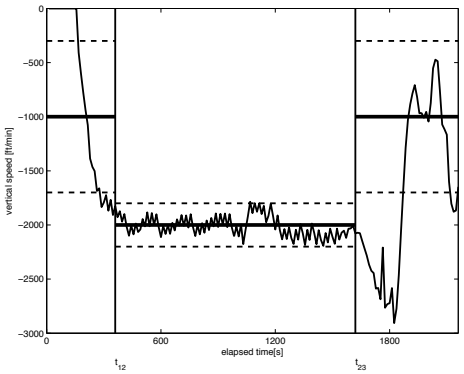


Fig. 7. Vertical speed over time for a CDA scenario.

```
1 v_vert(I) ~ gauss( cond(I < t_12  , v_bar,
2                         cond((I ≥ t_12) and (I< t_23),
3                            v_bar_cda, v_bar)),
4                    cond(I < t_12  , sigma_sq_high,
5                         cond((I ≥ t_12) and (I< t_23),
6                            sigma_sq_low, sigma_sq_high))).
7 max pr({v_vert}|{t_12, t_23}) for {t_12, t_23}.
```

Listing 1.3. AutoBayes specification for detection of CDA (excerpts)

In our analysis, we tested descent scenarios over the period of one week to find out scenarios, which were likely flown as actual CDA approaches. In a preprocessing phase, we eliminated all tracks that had longer horizontal stretches flown at low altitude. Then, we used our AutoBayes model to find the longest stretches with constant vertical speed. The length of these stretches related to the overall duration of the descent was taken as metric on how CDA-like an approach was. In the given data set, we were able to identify a number of likely CDA approaches. Many of them involved cargo or business jets and most were flown between during the night. Since standard procedures do not offer CDA approaches, this finding is consistent with the fact that approach procedures are handled more flexible during times with low traffic density.

5 Conclusions

In this paper, we discussed how the program synthesis tool AutoBayes can be used for the statistical analysis of aircraft track data. We presented experiments on multivariate clustering on trajectories, which have been represented as vectors of features, and on change-point models to estimate unknown parameters like the CAS-mach transition and the detection of constant descent approaches (CDA). For each of these tasks, we presented the AutoBayes specification, which comprises a compact, fully declarative statistical model. From that, AutoBayes generates a customized algorithm implemented in C or C++ within a few seconds run time. The schema-based synthesis approach used in AutoBayes together with its powerful symbolic system with enable the system to fully automatically synthesize complicated, but highly documented code of up to several thousand lines of C/C++.

The schema library of AutoBayes can be extended to handle different kinds of probability density functions and to incorporate different algorithm skeletons [24]. However, such extensions usually require a substantial understanding of the internal workings of AutoBayes and detailed knowledge of Prolog.

By using AutoBayes to generate customized algorithms for each individual task, a lot of otherwise necessary implementation effort can be saved, in particular as AutoBayes is capable of handling statistical models with many non-Gaussian distributions as well as incorporating domain knowledge and additional constraints in the form of (Bayesian) priors. With the generated code interfacing to Octave and Matlab, AutoBayes thus can be used to quickly and effectively explore a variety of statistical models for advanced data analysis tasks as are necessary for the analysis of Air Traffic data.

References

1. Erzberger, H.: CTAS: Computer intelligence for air traffic control in the terminal area. Technical Report NASA TM-103959, NASA (1992)
2. Denery, D.G., Erzberger, H.: The center-tracon automation system: Simulation and field testing. In: Proceedings of the Advanced Workshop on ATM (ATM 95) sponsored by the National Research Council of Italy, NASA TM-110366 (1995)
3. Coppenberger, R.A., Lanier, R., Sweet, D., Dorsky, S.: Design and development of the en route descent advisor (EDA) for conflict-free arrival metering. In: AIAA Guidance, Navigation, and Control Conference. AIAA-2004-4875. (2004)
4. Slattery, R., Zhao, Y.: Trajectory synthesis for air traffic automation. Journal of Guidance, Control and Dynamics **20** (1997) 232–238
5. Fischer, B., Schumann, J.: AutoBayes: A system for generating data analysis programs from statistical models. J. Functional Programming **13** (2003) 483–508
6. Schumann, J., Jafari, H., Pressburger, T., Denney, E., Buntine, W., Fischer, B.: AutoBayes program synthesis system users manual. Technical Report NASA/TM-2008-215366, NASA (2008)
7. Gariel, M., Srivastava, A., Feron, E.: Trajectory clustering and an application to airspace monitoring. Conf. on Intelligent Data Understanding (CIDU). (2009)

8. Meyerhoff, N., Wang, G.: Statistical analysis and time series modeling of air traffic operations data from flight service stations and terminal radar approach control facilities: Two case studies. Technical Report ADA109873, DTIC (1981)
9. Cheeseman, P., Stutz, J.: Bayesian classification (AutoClass): Theory and results. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, AAAI Press (1996) 153–180
10. McLachlan, G., Peel, D., Basford, K.E., Adams, P.: The EMMIX software for the fitting of mixtures of normal and t-components. J. Statistical Software **4** (1999)
11. Fraley, C., Raftery, A.E.: MCLUST: Software for model-based clustering, density estimation, and discriminant analysis. Technical Report 415, Department of Statistics, University of Washington (2002)
12. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. SIGKDD Explorations **11** (2009)
13. Buntine, W.L.: Operations for learning with graphical models. J. AI Research **2** (1994) 159–225
14. Whittle, J., Van Baalen, J., Schumann, J., Robinson, P., Pressburger, T., Penix, J., Oh, P., Lowry, M., Brat, G.: Amphion/NAV: Deductive synthesis of state estimation software. In: Proc. 16th Intl. Conf. Automated Software Engineering, IEEE Comp. Soc. Press (2001) 395–399
15. Ayari, A., Basin, D.: A higher-order interpretation of deductive tableau. J. Symbolic Computation **31** (2001) 487–520
16. Manna, Z., Waldinger, R.J.: Fundamentals of deductive program synthesis. IEEE Trans. Software Engineering **18** (1992) 674–704
17. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm (with discussion). J. of the Royal Statistical Society series B **39** (1977) 1–38
18. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical Recipes in C. 2nd. edn. Cambridge University Press (1992)
19. Fischer, B., Hajian, A., Knuth, K., Schumann, J.: Automatic derivation of statistical data analysis algorithms: Planetary nebulae and beyond. In: Proc. 23rd Intl. Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering, American Institute of Physics (2003) 276–291
20. Hartigan, J.: Clustering Algorithms. Wiley (1975)
21. Brodsky, B.E., Darkhovsky, B.S.: Nonparametric Methods in Change-Point Problems. Kluwer (1993)
22. Basseville, M., Nikiforov, I.V.: Detection of Abrupt Changes: Theory and Application. Prentice-Hall (1993)
23. Roach, K.: Analysis of American Airlines continuous descent approaches at Dellas/Fort Worth airport. Technical Report NTX_20080522_CDA_Meeting, North Texas Research Station (NTX) (2008)
24. Gray, A.G., Fischer, B., Schumann, J., Buntine, W.: Automatic derivation of statistical algorithms: The EM family and beyond. In: Advances in Neural Information Processing Systems 15, MIT Press (2003) 689–696
25. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer (2006)

Appendix

This appendix shows how AutoBayes solves $\max \mathbf{P}(hd \mid c, \mu_{hd}, \theta^0_{hd})$ for μ_{hd}, θ^0_{hd}, from Listing 1.1. AutoBayes automatically generated the detailed derivation typeset in LaTeX. We only added a few line-breaks and abbreviated theta0_hd by θ and m_hd by m. The solution uses a function *solve_B* to solve the Bessel function. As no closed-form solution exists, we chose to approximate this function by $solve\_B(x) \approx \tan(0.5\pi x)$ (see [25]). If necessary, an AutoBayes schema could be developed to instantiate an iterative numerical algorithm.

The conditional probability $\mathbf{P}(hd \mid c, m, \theta)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+N} \mathbf{P}(hd_i | c_i, m, \theta)$$

The probability occurring here is atomic and can thus be replaced by the respective probability density function given in the model. Summing out the expected variable c_{pv16} yields the log-likelihood function

$$\sum\nolimits_{j \in \mathrm{dom}\ c_i \sim q(i,j)}^{i=0\ldots-1+N} \log \prod_{k=0}^{-1+N} \exp\left(cos(hd_k - \theta_{c_k})\, m_{c_k}\right) \frac{1}{2\,\pi\, B_i(0, m_{c_k})}$$

which can be simplified to

$$(-1\ N\ \log 2) + (-1\ N\ \log \pi) +$$
$$(-1 \sum_{i=0}^{-1+C} \log B_i(0, m_i) \sum_{j=0}^{-1+N} q(j,i)) + \sum_{i=0}^{-1+C} m_i \sum_{j=0}^{-1+N} cos((-1\,\theta_i) + hd_j)\, q(j,i)$$

This function is then optimized w.r.t. the goal variables m and θ.

...

The function

$$(-1\ \log B\_i(0, m_l) \sum_{i=0}^{-1+N} q(i,l)) + (m_l \sum_{i=0}^{-1+N} cos((-1\,\theta_l) + hd_i)\, q(i,l))$$

is then symbolically maximized w.r.t. the goal variables m_l and θ_l. The partial differentials

$$\frac{\partial f}{\partial m_l} = (-1\, B\_i(0, m_l)^{-1}\, B\_i(1, m_l) \sum_{i=0}^{-1+N} q(i,l)) + \sum_{i=0}^{-1+N} cos((-1\,\theta_l) + hd_i)\, q(i,l)$$

$$\frac{\partial f}{\partial \theta_l} = (-1\, m_l\, sin(\theta_l) \sum_{i=0}^{-1+N} cos(hd_i)\, q(i,l)) + (cos(\theta_l)\, m_l \sum_{i=0}^{-1+N} sin(hd_i)\, q(i,l))$$

are set to zero; these equations yield the solutions

$$m_l = solve\_B(\sum_{i=0}^{-1+N} q(i,l)^{-1} \sum_{i=0}^{-1+N} cos((-1\,\theta_l) + hd_i)\, q(i,l))$$

$$\theta_l = atan2(\sum_{i=0}^{-1+N} sin(hd_i)\, q(i,l),\ \sum_{i=0}^{-1+N} cos(hd_i)\, q(i,l))$$

Towards Compositional CLP-based Test Data Generation for Imperative Languages

Elvira Albert[1], Miguel Gómez-Zamalloa[1], José Miguel Rojas[2], Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

1 Introduction

Test data generation (TDG) is the process of automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; *loop-k* (resp. *block-k*) which limits to a threshold k the number of times we iterate on loops (resp. visit blocks in the control flow graph [1]). Among the wide variety of approaches to TDG (see [20]), our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [6,12] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic execution* of the program [5,16,17,13,11,19], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach has been combined with the use of *constraint solvers* [17,11,19] in order to handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For instance, a symbolic JVM machine which integrates several constraints solvers has been designed in [17] for TDG of Java (bytecode) programs. In general, a symbolic machine requires non-trivial extensions w.r.t. a non-symbolic one like the JVM: (1) it needs to execute (imperative) code symbolically as explained above, (2) it must be able to non-deterministically execute multiple paths (as without knowledge about the input data non-determinism usually arises).

In recent work [10], we have proposed a CLP-based approach to TDG of imperative programs consisting of three main ingredients: (i) The imperative program is first translated into an equivalent CLP one, named *CLP-translated program* in what follows. The translation can be performed by partial evaluation [9] or by traditional compilation. (ii) Symbolic execution on the CLP-translated program can be performed by relying on the standard evaluation mechanism of CLP, which provides backtracking and handling of symbolic expressions for free. (iii) The use of dynamic memory requires to define heap-related operations that, during TDG, take care of constructing complex data structures with unbounded data (e.g., recursive data structures). Such operations can be implemented in CLP [10].

It is well-known that symbolic execution might become computationally intractable due to the large number of paths that need to be explored and also to

```
class R {                                  class A {
  int n; int d;                              static int abs(int x){
  void simplify(){                             if (x >= 0) return x;
    int gcd = A.gcd(n,d);                      else return -x;
    n = n/gcd; d = d/gcd;}                   }
  static R[] simp(R[] rs){                   static int gcd(int a,int b){
    int length = rs.length;                    int res;
    R[] oldRs = new R[length];                 while (b != 0){
    arraycopy(rs,oldRs,length);                  res = a%b; a = b; b = res;}
    for (int i = 0; i < length; i++)           return abs(a);
      rs[i].simplify();                      }
    return oldRs;}}                        }
```

Fig. 1. Java source of working example

the size of their associated constraints (see [18]). Currently, one of the main challenges of symbolic execution (and thus of glass-box TDG) is to scale up to larger applications. While compositionality has been applied in many areas of static analysis to improve scalability, it is less widely used in TDG (some notable exceptions in the context of dynamic testing are [7,3]). In this paper, we propose a compositional approach to static CLP-based TDG for imperative languages. In symbolic execution, compositionality means that when a method m invokes p, the execution can *compose* the *test cases* available for p (also known as *method summary* for p) with the current execution state and continue the process, instead of having to symbolically execute p. By test cases or method summary, we refer to the set of path constraints obtained by symbolically executing method p using a certain coverage criterion.

Having a compositional TDG approach in turn facilitates the handling of *native code* during symbolic execution, i.e., code which is implemented in a different language. This is achieved by modeling the behavior of native code as preconditions in the input state and postconditions in the output state. Such model can be composed with the current state during symbolic execution in the same way as the test cases inferred automatically by the testing tool are. By treating native code, we overcome one of the inherent limitations of symbolic execution (see [18]). Indeed, both scaling techniques and handling native code are considered main challenges in the fields of symbolic execution and TDG.

2 CLP-based Test Case Generation

In this section, we summarize the CLP-based approach to TDG for imperative languages introduced in [1] and recently extended to object-oriented languages with dynamic memory in [10]. For simplicity, we do not take aliasing of references into account and simplify the language by excluding inheritance and virtual invocations. However, these issues are orthogonal to compositionality and our approach could be applied to the complete framework of [10]. Also, although our approach is not tied to any particular imperative language, as regards dynamic memory, we assume a Java-like language. In [9], it has been shown that Java bytecode (and hence Java) can be translated into the equivalent CLP-programs shown below.

2.1 From Imperative to Equivalent CLP Programs

A *CLP-translated program* as defined in [10] is made up of a set of predicates. Each predicate is defined by one or more mutually exclusive clauses. Each clause receives as input a possibly empty list of arguments $Args_{in}$ and an input heap H_{in}, and returns a possibly empty output $Args_{out}$, a possibly modified output heap H_{out}, and an exceptional flag indicating whether the execution ends normally or with an uncaught exception. The body of a clause may include a set of guards (comparisons between numeric data or references, etc.) followed by different types of instructions: arithmetic operations and assignment statements, calls to other predicates, instructions to create objects and arrays and to consult the array length, read and write accesses to object fields or array positions, as defined by the following grammar:

$$
\begin{array}{rl}
\mathit{Clause} ::= & \mathit{Pred}\ (\mathit{Args}_{in},\mathit{Args}_{out},H_{in},H_{out},\mathit{ExFlag})\ \texttt{:-}\ [G,]B_1,B_2,\ldots,B_n.\\
G ::= & \mathit{Num}^*\ \mathit{ROp}\ \mathit{Num}^* \mid \mathit{Ref}_1^*\ \backslash\texttt{==}\ \mathit{Ref}_2^*\\
B ::= & \mathit{Var}\ \#\texttt{=}\ \mathit{Num}^*\ \mathit{AOp}\ \mathit{Num}^* \mid \mathit{Pred}\ (\mathit{Args}_{in},\mathit{Args}_{out},H_{in},H_{out},\mathit{ExFlag}) \mid\\
& \mathsf{new\_object}(H,C^*,\mathit{Ref}^*,H) \mid \mathsf{new\_array}(H,T,\mathit{Num}^*,\mathit{Ref}^*,H) \mid \mathsf{length}(H,\mathit{Ref}^*,\mathit{Var}) \mid\\
& \mathsf{get\_field}(H,\mathit{Ref}^*,\mathit{FSig},\mathit{Var}) \mid \mathsf{set\_field}(H,\mathit{Ref}^*,\mathit{FSig},\mathit{Data}^*,H) \mid\\
& \mathsf{get\_array}(H,\mathit{Ref}^*,\mathit{Num}^*,\mathit{Var}) \mid \mathsf{set\_array}(H,\mathit{Ref}^*,\mathit{Num}^*,\mathit{Data}^*,H)
\end{array}
$$

$$
\begin{array}{rlrl}
\mathit{Pred} ::= & \mathit{Block} \mid \mathit{MSig} & \mathit{ROp} ::= & \#> \mid \#< \mid \#>= \mid \#=< \mid \#= \mid \#\backslash=\\
\mathit{Args} ::= & [] \mid [\mathit{Data}^*|\mathit{Args}] & \mathit{AOp} ::= & + \mid - \mid * \mid / \mid \mathit{mod}\\
\mathit{Data} ::= & \mathit{Num} \mid \mathit{Ref} \mid \mathit{ExFlag} & T ::= & \mathit{bool} \mid \mathit{int} \mid C \mid \mathit{array}(T)\\
\mathit{Ref} ::= & \mathit{null} \mid \mathit{r}(\mathit{Var}) & \mathit{FSig} ::= & C{:}FN\\
\mathit{ExFlag} ::= & \mathit{ok} \mid \mathit{exc}(\mathit{Var}) & H ::= & \mathit{Var}
\end{array}
$$

Non-terminals *Block*, *Num*, *Var*, *FN*, *MSig* and *C* denote, resp., the set of predicate names, numbers, variables, field names, method signatures and class names. Clauses can define both methods which appear in the original source program (*MSig*), or additional predicates which correspond to intermediate blocks in the program (*Block*). An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constraint) variable.

Example 1. Fig. 1 shows the Java source of our running example and Fig. 2 the CLP-translated version of method `simp` obtained from the bytecode. The main features that can be observed from the translation are: (1) All clauses contain input and output arguments and heaps, and an exceptional flag. Reference variables are of the form `r(V)` and we use the same variable name `V` as in the program. E.g., argument `Rs` of `simp` corresponds to the method input argument. (2) Java exceptions are made explicit in the translated program, e.g., the second clauses for predicates `simp` and `loopbody2` capture the null-pointer exception (`NPE`). (3) Conditional statements and iteration in the source program are transformed into guarded rules and recursion in the CLP program, respectively, e.g., the for-loop corresponds to the recursive predicate `loop`. (4) Methods (like `simp`) and intermediate blocks (like `r1`) are uniformly represented by means of predicates and are not distinguishable in the translated program.

2.2 Symbolic Execution

When the imperative language does not use dynamic memory, CLP-translated programs can be executed by using the standard CLP execution mechanism with all arguments being free variables. However, in order to generate heap-allocated data

```
simp([r(Rs)],[Ret],H0,H3,EF) :- length(H0,Rs,Len), Len #>= 0, new_array(H0,'R',Len,OldRs,H1),
        arraycopy([r(Rs),r(OldRs),Len],[],H1,H2,EFp), r1([EFp,r(Rs),r(OldRs),Len],[Rt],H2,H3,EF).
simp([null],_,Hin,Hout,exc(ERef)) :- new_object(Hin,'NPE',ERef,Hout).

r1([ok,Rs,OldRs,Length],[Ret],H1,H2,EF) :- loop([Rs,OldRs,Length,0],[Ret],H1,H2,EF).
r1([exc(ERef),_,_,_],_,H,H,exc(ERef)).

loop([_,OldRs,Length,I],[OldRs],H,H,ok) :- I #>= Length.
loop([Rs,OldRs,L,I],[Ret],H1,H2,EF) :- I #< L, loopbody1([Rs,OldRs,L,I],[Ret],H1,H2,EF).

loopbody1([r(Rs),OldRs,Length,I],[Ret],H1,H2,EF) :- length(H1,Rs,L), L #>= 0, I #< L,
        get_array(H1,Rs,I,RSi), loopbody2([r(Rs),OldRs,Length,I,Rsi],[Ret],H1,H2,EF).

loopbody2([Rs,OldRs,Length,I,r(Rsi)],[Ret],H1,H3,EF) :- simplify([r(Rsi)],[],H1,H2,EFp),
        loopbody3([EFp,Rs,OldRs,Length,I],[Ret],H2,H3,EF).
loopbody2([_,_,_,_,null],_,H1,H2,exc(ERef)) :- new_object(H1,'NPE',ERef,H2).

loopbody3([ok,Rs,OldRs,L,I],[Ret],H1,H2,EF) :- Ip #= I+1, loop([Rs,OldRs,L,Ip],[Ret],H1,H2,EF).
loopbody3([exc(ERef),_,_,_,_],_,H,H,exc(ERef)).
```

Fig. 2. CLP Translation associated to bytecode of method `simp`

structures, it is required to define heap-related operations which build the heap associated with a given path by using only the constraints induced by the visited code. Fig. 3 summarizes the CLP-implementation of the operations in [10] to create heap-allocated data structures (like new_object and new_array) and to read and modify them (like set_field, etc.) which use some auxiliary predicates (like deterministic versions of member member_det, of replace replace_det, and nth0 and replace_nth for arrays) which are quite standard and hence their implementation is not shown.

The intuitive idea is that the heap during symbolic execution contains two parts: the *known part*, with the cells that have been explicitly created during symbolic execution appearing at the beginning of the list, and the *unknown part*, which is a logic variable (tail of the list) in which new data can be added.

Importantly, the definition of get_cell/3 distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause). Note the use of syntactic equality rather than unification since references at execution time can be variables. (ii) Otherwise, it reaches the unknown part of the heap (a logic variable), and it allocates the reference (in this case a variable) there (first clause).

Example 2. Let us consider a branch of the symbolic execution of method `simp` which starts from $\texttt{simp}(\mathsf{A_{in}},\mathsf{A_{out}},\mathsf{H_{in}},\mathsf{H_{out}},\mathsf{EF})$, the empty state $\phi_0 = \langle \emptyset, \emptyset \rangle$ and which (by ignoring the call to `arraycopy` for simplicity) executes the predicates $\texttt{simp}_1 \rightarrow \mathsf{length} \rightarrow \geq \rightarrow \mathsf{new\_array} \rightarrow \texttt{r1}_1 \rightarrow \texttt{loop}_1 \rightarrow \geq \rightarrow \texttt{true}$. The subindex 1 indicates that we pick up the first rule defining a predicate for execution. As customary in CLP, a state ϕ consists of a set of bindings σ and a constraint store θ. The final state of the above derivation is $\phi_f = \langle \sigma_f, \theta_f \rangle$ with $\sigma_f = \{\mathsf{A_{in}} = [\mathsf{r(Rs)}],\ \mathsf{A_{out}} = [\mathsf{r(C)}],\ \mathsf{H_{in}} = [(\mathsf{Rs}, \mathsf{array}(\mathsf{T}, \mathsf{L}, \_))|\_],\ \mathsf{H_{out}} = [(\mathsf{C}, \mathsf{array}(\mathsf{'R'}, \mathsf{L}, \_))|\mathsf{H_{in}}],\ \mathsf{EF} = \mathsf{ok}\}$ and $\theta_f = \{\mathsf{L} = 0\}$. Observe that a heap is represented as a list of locations which are pairs made up of a unique reference and a cell, which in turn can be an object or an array. This can be read as "if the array at location Rs in the input heap has length 0, then it is not modified and a new array of length 0 is returned". This derivation corresponds to the first test case in Table 3 where a graphical representation for the heap is used.

2.3 Method Summaries obtained by TDG

It is well-known that, in symbolic execution, the execution tree to be traversed is in general infinite. This is because iterative constructs such as loops and recursion

```
new_object(H,C,Ref,H') :- build_object(C,Ob), new_ref(Ref), H' = [(Ref,Ob)|H].
new_array(H,T,L,Ref,H') :- build_array(T,L,Arr), new_ref(Ref), H' = [(Ref,Arr)|H].

length(H,Ref,L) :- get_cell(H,Ref,Cell), Cell = array(_,L,_).

get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN, Ob = object(T,Fields),
                           T = C, member_det(field(FN,V),Fields).
get_array(H,Ref,I,V) :- get_cell(H,Ref,Arr), Arr = array(_,_,Xs), nth0(I,Xs,V).

set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN, Ob = object(T,Fields),
                              T = C, replace_det(Fields,field(FN,_),field(FN,V),Fds'),
                              set_cell(H,Ref,object(T,Fds'),H').
set_array(H,Ref,I,V,H') :- get_cell(H,Ref,Arr), Arr = array(T,L,Xs),
                           replace_nth0(Xs,I,V,Xs'), set_cell(H,Ref,array(T,L,Xs'),H').
```

```
get_cell(H,Ref,Cell) :- var(H), !, H = [(Ref,Cell)|_].
get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).

set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !, H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H''], set_cell(H',Ref,Cell,H'').
```

Fig. 3. Heap operations for symbolic execution [10]

usually induce an infinite number of execution paths when executed without input values. It is therefore essential to establish a *termination criterion* which, in the context of TDG, is usually defined in terms of the so-called *coverage criterion* (see Sec. 1). The concept of *method summary* corresponds to the finite representation of its symbolic execution for a given coverage criterion.

Definition 1 (method summary). *Let $\mathcal{T}_m^C$ be the finite symbolic execution tree of method m obtained by using a coverage criterion C. Let B be the set of successful branches in $\mathcal{T}_m^C$ and $m(Args_{in},Args_{out},H_{in},H_{out},EF)$ be its root. A* method summary *for m w.r.t. C, denoted $\mathcal{S}_m^C$, is the set of 6-tuples associated to each branch $b \in B$ of the form: $\langle\sigma(Args_{in}),\sigma(Args_{out}),\sigma(H_{in}),\sigma(H_{out}),\sigma(EF),\theta\rangle$, where σ and θ are the set of bindings and constraint store, resp., associated to b.*

Each tuple in a summary is said to be a *(test) case* of the summary, denoted c, and its associated *state* ϕ_c comprises its corresponding σ and θ, also referred to as *context* ϕ_c. Intuitively, a method summary can be seen as a *complete specification* of the method for the considered coverage criterion, so that each summary case corresponds to the *path constraints* associated with each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a *partial specification* for the method, as long as there are incomplete branches in the finite tree.

Example 3. Table 1 shows the summary obtained by symbolically executing method `simplify` using the *block-2* coverage criterion of [1] (see Sec. 1): The summary contains 5 cases, which correspond to the different execution paths induced by calls to methods `gcd` and `abs`. For the sake of clarity, we adopt a graphical representation for the input and output heaps. Heap locations are shown as arrows labeled with their reference variable names. Split-circles represent objects of type `R` and fields `n` and `d` are shown in the upper and lower part, respectively. Exceptions are shown as starbursts, like in the special case of the fraction "0/0", for which an arithmetic

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(B)		B → C/0	B → G/0	ok	C<0, J=-C, G=C/J
r(B)		B → C/0	B → 1/0	ok	C>0
r(B)		B → 0/0	B → 0/0, 0 → AE	exc(0)	
r(B)		B → C/D	B → H/J	ok	D<0, K=-D, H=C/K, C *mod* D=0, J=D/K
r(B)		B → C/D	B → H/1	ok	D>0, C *mod* D=0, C/D=H

Table 1. Summary of method `simplify`

exception (`AE`) is thrown due to a division by zero. In summary examples of Tables 2 and 3, split-rectangles represent arrays, with the length of the array in the upper part and its list of values (in Prolog syntax) in the lower one.

In a subsequent stage, it is possible to produce actual values from the obtained path constraints (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore obtaining executable test-cases. However, this is not an issue of this paper and we will rely on the method summaries only in what follows.

3 Towards a Compositional CLP-based TDG Approach

The goal of this section is to study the compositionality of the CLP-based approach to TDG of imperative languages presented in the previous section.

3.1 Composition in Symbolic Execution

Let us assume that during the symbolic execution of a method m, there is a method invocation to p within a state ϕ. In the context of our CLP approach, the challenge is to define a composition operation so that, instead of symbolically executing p its (previously computed) summary $\mathcal{S}_p$ is reused. Notice that the use of a summary for p must be equivalent as if p were indeed symbolically executed within m, as in the non-compositional manner.

Fig. 4 shows such a composition operation (predicate compose_summary/1). The idea is therefore to replace during symbolic execution a method invocation I by a call compose_summary(I) when there is a summary available for I. Intuitively, given the variables of the call to p, with their associated state ϕ, compose_summary/1 produces a branch (on backtracking) for each *compatible* case $c \in \mathcal{S}_p$, composing its state ϕ_c with ϕ and producing a new state ϕ' to continue the symbolic execution on. We assume that the summary for a method p is represented as a set of facts of the form summary(p,SA_{in},SA_{out},SH_{in},SH_{out},SEF,θ). Roughly speaking, state ϕ_c is *compatible* with ϕ if: 1) the bindings and constraints on the arguments can be conjoined, and 2) the structures of the input heaps match. This means that, for each location which is present in both heaps, its associated cells match, which in turn requires that their associated bindings and constraints can be conjoined. Note that compatibility of a case is checked on the fly, so that if ϕ is not compatible with ϕ_c some call in compose_summary/1 will fail.

As it can be observed by looking at the code of compose_summary/1, the input and output arguments, and the exception flags are simply unified, while the

```
compose_summary(Call) :-
    Call =..[M,A_in,A_out,H_in,H_out,EF],
    summary(M,SA_in,SA_out,SH_in,SH_out,SEF,σ),
    SA_in = A_in, SA_out = A_out, SEF = EF,
    compose_hin(H_in,SH_in),
    compose_hout(H_in,SH_out,H_out).
    load_store(σ).

compose_hin(_,SH) :- var(SH), !.
compose_hin(_,[]).
compose_hin(H,[(R,Ce)|SH]) :-
    get_cell(H,R,Ce'), unify_cells(Ce',Ce),
    compose_hin(H,SH).

unify_cells(ob(T,Fs),ob(T,Fs')) :-
    unify_fields(Fs,Fs').
unify_cells(array(T,L,Vs),array(T,L,Vs)).

unify_fields(_,Fs) :- var(Fs),!.
unify_fields(_,[]).
unify_fields(Fs,[field(FName,FV)|RFs]) :-
    member_det(field(FName,FV'),Fs), !,
    FV = FV', unify_fields(Fs,RFs).

compose_hout(H,SH,H) :- var(SH),!.
compose_hout(H,[],H).
compose_hout(H_in,[(Ref,C)|SH_out],H_out) :-
    get_cell(H_in,Ref,C'), combine_cell(C',C,C''),
    set_cell(H_in,(Ref,C''),H'),
    compose_hout(H',SH_out,H_out).

combine_cell(ob(T,Fs),ob(T,Fs'),ob(T,Fs'')) :-
    set_fields(Fs,Fs',Fs'').
combine_cell(array(T,L,Vs),array(T,L,Vs'),
                array(T,L,Vs'')) :-
    set_array_vs(Vs,Vs',Vs'').

set_fields(Fs,Fs',Fs) :- var(Fs'),!.
set_fields(Fs,[],Fs).
set_fields(Fs,[field(FN,FV)|RFs],Fs'') :-
    replace_det(Fs,field(FN,_),field(FN,FV),Fs'),
    set_fields(Fs',RFs,Fs'').

set_array_vs(Vs,Vs',Vs) :- var(Vs'),!.
set_array_vs([],[],[]).
set_array_vs([_|RVs],[V|RVs'],[V|RVs'']) :-
    set_array_vs(RVs,RVs',RVs'').
```

Fig. 4. The composition operation

constraint store θ is trivially incorporated by means of predicate load_store/1. The heaps however require a more sophisticated treatment, mainly due to the underlying representation of sets (of objects and fields) as Prolog lists. Predicate compose_hin/2 composes the input heap of the summary case $\mathsf{SH_{in}}$ with the current heap $\mathsf{H_{in}}$ producing the composed input heap in $\mathsf{H_{in}}$. To accomplish this, compose_hin/2 traverses each cell in $\mathsf{SH_{in}}$, and binds the appropriate variables in $\mathsf{H_{in}}$ with the appropriate sub-terms in the cell. Essentially, it produces an equivalent effect as if a sequence of get_field/4 operations for each field in each object in $\mathsf{SH_{in}}$ are performed over $\mathsf{H_{in}}$ (or get_array/4 for arrays). Similarly, the effect of compose_hout/3 is equivalent as if a sequence of set_field/5 operations for each field in each object in $\mathsf{SH_{out}}$ are performed (or set_array/5 for each array element), starting with $\mathsf{H_{in}}$, and gradually obtaining new heaps until getting the composed output heap $\mathsf{H_{out}}$.

Example 4. When symbolically executing `simp`, the call simplify($\mathsf{A_{in}}$,$\mathsf{A_{out}}$,$\mathsf{H_{in}}$,$\mathsf{H_{out}}$,EF) arises in one of the branches in state $\sigma = \{\mathsf{A_{in}}=[\mathsf{r(E0)}], \mathsf{A_{out}}=[], \mathsf{H_{in}}=[(0, \mathsf{array}(\mathsf{R'}, \mathsf{L}, [\mathsf{E0}|\_])), (\mathsf{Rs}, \mathsf{array}(\mathsf{R'}, \mathsf{L}, [\mathsf{E0}|\_]))|\mathsf{RH_{in}}]\}$ and $\theta = \{\mathsf{L} \geq 0\}$. The composition of this state with the second summary case of `simplify` succeeds and produces the state $\sigma' = \sigma \cup \{\mathsf{E0}=\mathsf{B}, \mathsf{RH_{in}}=[(\mathsf{B}, \mathsf{ob}(\mathsf{R'}, [\mathsf{field}(\mathsf{n}, \mathsf{C}), \mathsf{field}(\mathsf{d}, 0)]))|\_], \mathsf{H_{out}}=[\ldots, (\mathsf{B}, \mathsf{ob}(\mathsf{R'}, [\mathsf{field}(\mathsf{n}, 1), \mathsf{field}(\mathsf{d}, 0)]))|\_]\}$ and $\theta'=\{\mathsf{L} \geq 0, \mathsf{C} > 0\}$. The dots in $\mathsf{H_{out}}$ denote the rest of the cells in $\mathsf{H_{in}}$.

3.2 Compositional TDG Schemata

In order to design a compositional, incremental approach to TDG, the composition operation can be incorporated in two main different ways within the testing process:

Context-sensitive. Usually, starting from an entry method m (and possibly a set of preconditions), TDG performs a top-down symbolic execution such that, when a

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
[B,C,0]		H	H	ok	$\emptyset$
[r(B),null,C]		B → L [V|_]	B → L [V|_] 1 → NPE	exc(1)	C>0, L>0
[null,B,C]		H	2 → NPE	exc(2)	C>0
[B,C,D]		H	3 → AE	exc(3)	D<0
[r(B),r(C),1]		B → L1 [V|_] C → L2 [V2|_]	B → L1 [V|_] C → L2 [V|_]	ok	L1>1, L2>0

Table 2. Summary of method `arraycopy`

method call p is found, its code is executed from the actual state ϕ. In a context-sensitive approach, once a method is executed, we store the summary computed for p in the context ϕ. If we later reach another call to p within a (possibly different) context ϕ', we first check if the stored context is sufficiently general. In such case, we can adapt the existing summary for p to the current context ϕ' (by relying on the operation in Fig. 4). At the end of each execution, it can be decided which of the computed (context-sensitive) summaries are stored for future use.

Context-insensitive. Another possibility is to perform the TDG process in a context-insensitive way. This can be done by first computing the call graph for the entry method m and the strongly connected components (SCC) for such graph. The SCCs can be traversed in reverse topological order starting from the SCC which does not depend on any other. The idea is that each SCC is symbolically executed from its entry w.r.t. the most general context (i.e., *true*). If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts.

In general terms, the advantage of the context-insensitive approach is that composition can always be performed. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts were not sufficiently general). In such case, it had been more efficient to obtain the summary without assuming any context. It remains as future work to formalize the above approaches (and possibly hybrid variants) and experimentally evaluate them.

3.3 Handling Native Code during Symbolic Execution

An inherent limitation of symbolic execution is the handling of native code. This is code usually implemented in another language. In the context of hybrid approaches to TDG which combine symbolic and concrete execution, a solution is concolic execution [8] where concrete execution is performed on random inputs and path constraints are collected at the same time; native code is executed for concrete values. Although we believe that such approach could be also adapted to our CLP framework, we concentrate here on a purely symbolic approach. In this case, the only possibility is to model the behavior of the native code by means of specifications. Such specifications can be in turn treated as summaries for the corresponding native methods. They can be declared by the code provider or automatically inferred by

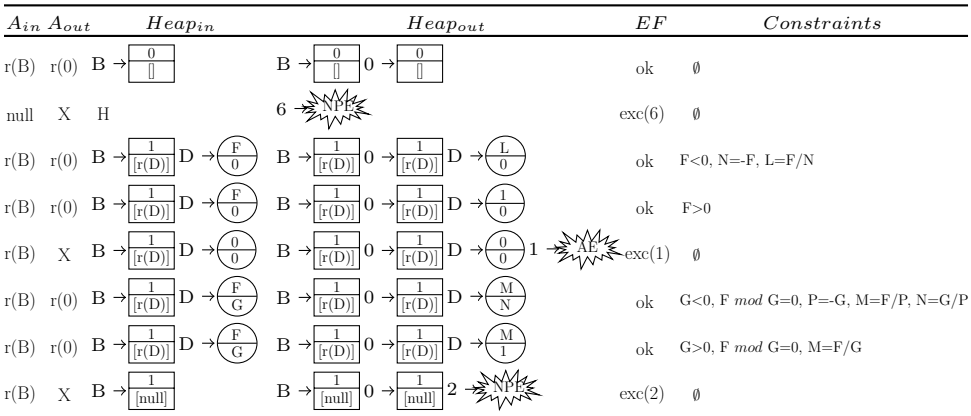


Table 3. Summary of method `simp`

a TDG tool for the corresponding language. Interestingly, the composition operator uses them exactly in the same way as it uses the summaries obtained by applying our own symbolic execution mechanism. Let us see an example.

Example 5. Assume that method `arraycopy` is native. A (complete) specification of `arraycopy` can be provided by means of a corresponding method summary, as shown in Table 2, where we have (manually) specified five cases: the first one for arrays of length zero, the second and third ones for null arrays, the fourth one for a negative length, and finally a normal execution of non-null arrays. Now, by using our compositional reasoning, we can continue symbolic execution for `simp` by composing the specified summary of `arraycopy` within the actual context. In Table 3, we show the entire summary of method `simp` for a *block-2* coverage criterion obtained by relying on the summaries for `simplify` and `arraycopy` shown before.

A practical question is how specifications for native code should be provided. A standard way is to use assertions (e.g., in JML in the case of Java) which could be parsed and easily transformed into our Prolog syntax.

3.4 Completeness w.r.t. Coverage Criteria

When selecting a coverage criterion during TDG, it must be ensured that the test cases obtained in the summary meet the coverage established by the criterion, i.e., *completeness* w.r.t. the criterion in guaranteed. In compositional TDG, the following question arises: *given an existing summary S for p w.r.t. the coverage criterion C, if while computing a summary for m w.r.t. C a call to p occurs and we compose S with the current state, is it ensured that the final summary obtained for m meets criterion C?* This does not hold for all coverage criteria, e.g., for statement coverage.

Example 6. Consider the following simple method:

```
m(int a,int b){if (a > 0 || b > 0) S;}
```

Without assuming any context, a summary with a single case $\{\texttt{a} > 0, \texttt{b} \leq 0\}$ is sufficient to achieve statement coverage. However, if `m` is called from an outer scope with a more restricted context (e.g., $\texttt{a} < 0$), such summary is no longer valid to achieve statement coverage since statement `S` would not be visited.

Intuitively, completeness in compositional reasoning for a coverage criterion is ensured when the following property holds: *given a summary S obtained for m in a context ϕ w.r.t. C, test cases for a more restricted context ϕ' can be obtained by adapting S to context ϕ'.* For instance, the block-k coverage criterion used in the examples of the paper is compositional. This is because, by restricting the context to ϕ', we are pruning the symbolic execution tree obtained for ϕ. Hence, the test cases for ϕ' can be obtained from the test cases in the summary obtained for ϕ (by adapting them to ϕ'). On the contrary, when such property does not hold, we say that the coverage criterion is *not* compositional.

Another interesting issue to study is whether different (compositional) criteria can be combined during TDG. In other words, *given a summary computed for p w.r.t. a criteria C', can we use it when computing test cases for m w.r.t. a criteria C?* By focusing on block-k, assume that C' corresponds to $k = 3$ and C to $k = 2$. We can clearly adapt the summaries obtained for $k = 3$ to the current criteria $k = 2$. Even more, if one uses the whole summary for $k = 3$, the required coverage $k = 2$ is ensured, although unnecessary test cases are introduced.

4 Conclusions and Future Work

Much effort has been devoted in the area of symbolic execution to alleviate scalability problems and three main approaches co-exist which, in a sense, complement each other. Probably, the most widely used is abstraction, a well-known technique to reduce large data domains of a program to smaller domains (see [15]). Another scaling technique which is closely related to abstraction is path merging [4,14], which consists in defining points where the merging of symbolic execution should occur. In this abstract, we have focused on yet another complementary approach, *compositional* symbolic execution, a technique widely used in static analysis but notably less common in the field of TDG. We have outlined how the CLP-based approach to TDG of imperative languages can be applied in a compositional way. This is essential in order to make the approach scalable and, as we have seen, also provides a practical way of dealing with native code.

In a longer version of this abstract, we plan to extend our work along several directions. From the theoretical side, we plan to develop concrete TDG strategies which rely on the composition operator (along the lines discussed in Sec. 3.2). We also want to further study completeness issues by characterizing the features that a coverage criterion must have in order to enable (complete) compositional reasoning. On the practical side, we want to be able to generate JUnit test cases from our Prolog terms and, also, be able to translate specifications provided by means of assertions into our Prolog format. Finally, we will carry out an experimental evaluation in the PET system, a Partial Evaluation-based TDG tool [2].

Acknowledgements

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, number 5438 in LNCS, pages 4–23. Springer-Verlag, March 2009.
2. E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 25–28, Madrid, Spain, January 2010. ACM Press.
3. Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2008.
4. Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *DATE*, pages 825–830. IEEE, 2008.
5. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.
6. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
7. Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 47–54. ACM, 2007.
8. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
9. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51:1409–1427, October 2009.
10. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming*, 2010. To appear.
11. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.
12. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Automated Software Engineering*, pages 219–228, 2000.
13. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
14. Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.
15. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.
16. C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
17. R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.
18. Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.
19. T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *19th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'09)*, 2009.
20. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Abstract Diagnosis of First Order Functional Logic Programs

Giovanni Bacci[1] and Marco Comini[1]

Dipartimento di Matematica e Informatica
University of Udine

Abstract We present a generic scheme for the abstract debugging of functional-logic programs. We associate to our programs a semantics based on a (continuous) immediate consequence operator, $\mathcal{P}[\![\mathcal{R}]\!]$, which models correctly the typical cogent features of modern functional-logic languages (non-deterministic, non-strict functions defined by non-confluent programs and call-time choice behaviour). 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.

1 Introduction

There has been a lot of work on debugging for functional-logic languages, amongst all [14,8,7,9,6,1], mostly following the declarative debugging approach. Declarative debugging 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 questions on correctness of solutions to 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 [10] is a framework parametric w.r.t. an abstract program property which be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. It is based on using the immediate consequence operator T_P to identify bugs in logic programs. 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-logic programming which applies the ideas of [10] to debug a functional-logic program. We associate a (continuous) immediate consequence operator $\mathcal{P}[\![\mathcal{R}]\!]$ to program $\mathcal{R}$ which allows us to derive the concrete semantics for $\mathcal{R}$. Then, we formulate an efficacious debugging methodology based on abstract interpretation which proceeds by approximating the $\mathcal{P}[\![\mathcal{R}]\!]$ operator by means of an abstract $\mathcal{P}^{\alpha}[\![\mathcal{R}]\!]$ operator. We show that, given the abstract intended specification $\mathcal{I}^{\alpha}$ of the semantics of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single step

of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ and, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

Among other valuable facilities, this debugging approach supports the development of cogent diagnostic tools that find program errors without having to determine symptoms in advance. The key issue of this approach is the goal-independence of the concrete semantics, meaning that the semantics is defined by collecting the observable properties starting with "most general" calls, while still providing a complete characterization of the program behavior.

2 Notations

Let us briefly introduce the notations used in the paper. Since we are interested in modern first-order functional-logic programs we will use the formalism of [12,4]. In the following a program is a special case of Graph Rewriting System $(\Sigma, \mathcal{R})$, where the signature is partitioned into two disjoint sets $\Sigma \coloneqq \mathcal{C} \uplus \mathcal{D}$, where term graphs (simply terms) are just labelled DAGs (instead of general admissible term graphs), without multiple occurrences of the same variable, and where the head of program rules $l \to r$ is a pattern. A term π is a *pattern* if it is of the form $f(\boldsymbol{c})$ where $f/n \in \mathcal{D}$ and $\boldsymbol{c}$ is a n-tuple of constructor terms (not sharing subterms, variables included). A pattern π is *most general* if $\boldsymbol{c}$ is a n-tuple of distinct variables. $\mathbb{MGP}$ is the set of all the most general patterns. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$, $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called constructor terms. Throughout this paper, $\mathcal{V}$ will denote a countably infinite set of variables. By $Var(s)$ we denote the set of variables occurring in the syntactic object s. A *fresh* variable is a variable that appears nowhere else. $s \ll S$ indicates that the syntactic object s is a fresh variant of an object in S. $t|_p$ is the subterm at the position p of t. $t[r]_p$ is the term t with the subterm at the position p replaced with r. Syntactic equality of terms is represented by $\equiv$. $\theta\restriction_s$ denotes the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by ε. $mgu(t, s)$ denotes "the" *most general unifier* homomorphism of term graphs t and s. By abuse of notation it denotes also its induced substitution. We will also denote by $mgu(\sigma_1, \ldots, \sigma_n)$ the most general unifier of substitutions $\sigma_1, \ldots, \sigma_n$ [15]. $t_0 \overset{\sigma}{\underset{\mathcal{R}}{\rightsquigarrow}}{}^* t_n$ denotes a *needed* narrowing derivation $t_0 \overset{\sigma_1}{\underset{\mathcal{R}}{\rightsquigarrow}} \ldots \overset{\sigma_n}{\underset{\mathcal{R}}{\rightsquigarrow}} t_n$ such that $\sigma = (\sigma_1 \ldots \sigma_n)\restriction_t$. If $t \overset{\sigma}{\underset{\mathcal{R}}{\rightsquigarrow}}{}^* s$ the pair $\sigma \cdot s$ is said a *partial answer* for t and when $s \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ we call it a *computed answer*.

3 The semantic framework

In this section we introduce the concrete semantics of our framework, which is suitable to model correctly the typical cogent features of modern functional-logic languages:

– non-deterministic, non-strict functions defined by non-confluent programs;
– the call-time choice behaviour (where the values of the arguments of an operation are determined before the operation is evaluated)

Actually we have obtained this semantics by optimal abstraction of a (much more) concrete semantics modeling needed narrowing trees, thus its correctness w.r.t. needed narrowing comes by construction and standard Abstract Interpretation results [11].

3.1 Incremental Answer Semantics

To deal with non-strict operations, as done by [8], we consider signatures $\Sigma_\bot$ that are extended by a special constructor symbol $\bot$ to represent undefined values, e.g. $S(S(\bot))$ denotes a number greater than 1 where the exact value is undefined. Such partial terms are considered as finite approximations of possibly infinite values. We denote by $\mathcal{T}(\mathcal{C}_\bot, \mathcal{V})$ the set of *partial constructor terms*.

Our fixpoint semantics is based on interpretations that consist of families of Incremental Answer Trees of the "most general calls" of a program. Intuitively an Incremental Answer Tree collects the "relevant history" of the computation of all computed answers of a goal, abstracting from function calls and focusing only on the way in which the answer is (incrementally) constructed.

Definition 1 (Incremental Answer Trees). *Given a term t over signature $\Sigma_\bot = \mathcal{C}_\bot \uplus \mathcal{D}$, an* incremental answer tree *for t is a tree of partial answers T such that*

1. *its root is $\varepsilon \cdot \tau_\bot(t)$, where $\tau_\bot(t)$ denotes the partial constructor term obtained by replacing all $\mathcal{D}$-rooted subterms of t with the $\bot$ symbol,*
2. *all sibling nodes are different partial answers and*
3. *for any partial answer $\sigma \cdot s$ parent of $\sigma' \cdot s'$ there exists a constructor substitution θ, a position p and $t \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{V}) \setminus \{\bot\}$ such that $s|_p = \bot$ and $s' \equiv (s\theta)[t]_p$.*

Given two incremental answer trees T and T' for t, $T \sqsubseteq T'$ if and only if every path in T starting from the root, is also a path of T'.

The semantic domain $\mathbb{C}_\Sigma$ is the set of all incremental answer trees ordered by $\sqsubseteq$. $\mathbb{C}_\Sigma$ is a complete lattice. We denote its bottom $\varepsilon \cdot \bot$ by $\bot_\mathbb{C}$.

The intuition is that for any parent node we choose an occurrence of a $\bot$ and we evaluate it until at least another constructor is added. All narrowing steps that do not introduce a constructor are collapsed together (see Example 1).

The idea behind our proposal is that of building a family of incremental answer trees "modulo variance", one tree for each most general call $f(x_1, \ldots, x_n)$ (which are a finite number). We quotient w.r.t. variance because the actual choice of variables names in $f(x_1, \ldots, x_n)$ has to be irrelevant. For notational convenience we represent this family as a function in the following way.

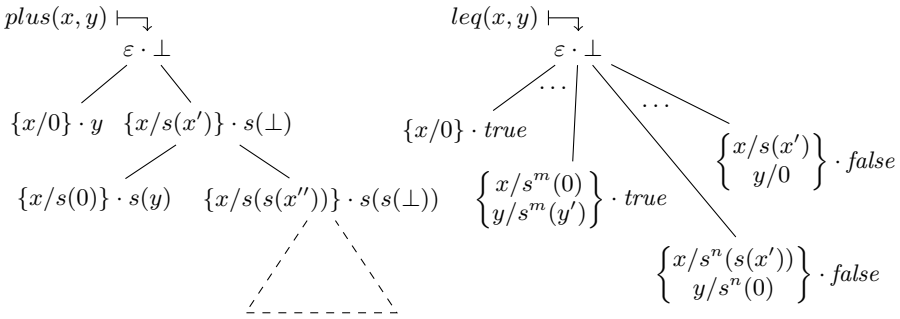


Figure 1. Semantics for *plus* and *leq*

Definition 2 (Interpretations). *An* interpretation *is a function* $\mathbb{MGP} \to \mathbb{C}_\Sigma$ modulo variance *such that, for every* $\pi \in \mathbb{MGP}$, $\mathcal{I}(\pi)$ *is an incremental answer tree for* π. *Two functions* I, J *are variants if for each* $\pi \in \mathbb{MGP}$ *there exists a renaming* ρ *such that* $I(\pi\rho) \equiv J(\pi)\rho$.

The semantic domain $\mathbb{I}_\Sigma$ *is the set of all interpretations ordered by the pointwise extension (modulo variance) of* $\sqsubseteq$. $\mathbb{I}_\Sigma$ *is a complete lattice. Its bottom is the constant function* $\lambda f(\boldsymbol{x}).\bot_{\mathbb{C}}$.

Example 1. The incremental answer trees depicted in Figure 1 denotes the semantics of the program

```
plus Z   y = y                  leq Z _ = True
plus (S x) y = S (plus x y)     leq (S x) Z = False
                                leq (S x) (S y) = leq x y
```

Notice that both trees are infinite in different ways. The one associated to $plus(x, y)$ has infinite height, since *plus* builds the solution one S at a time. The one associated to $leq(x, y)$ has infinite width, since *leq* delivers just true/-false. □

While trees are easy to understand and to manage in implementations, they are quite big to show in a paper and technical definitions tends to be more cluttered. Thus, for the sake of compactness and comprehension, we prefer to use in the following an isomorphic representation where we collect in a set all the nodes of a tree but annotating with • the new root constructors introduced by expanding a $\bot$ in a parent node. This annotation conveys all the needed information about inner tree nodes. We call these sets *annotated-sets*.

For example, the annotated-set representation of the trees of Figure 1 is

$$\begin{cases} plus(x,y) \mapsto \{\, \varepsilon \cdot \bot, \{x/0\} \cdot \overset{\bullet}{y}, \{x/s(x')\} \cdot \overset{\bullet}{s}(\bot), \\ \qquad\qquad \{x/s(0)\} \cdot \overset{\bullet}{s}(\overset{\bullet}{y}), \{x/s(s(x'))\} \cdot \overset{\bullet}{s}(\overset{\bullet}{s}(\bot)), \dots \,\} \\ leq(x,y) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \overset{\bullet}{true}, \dots, \{x/s^i(0), y/s^i(y')\} \cdot \overset{\bullet}{true}, \dots \\ \qquad\qquad \{x/s(x'), y/0\} \cdot \overset{\bullet}{true}, \dots, \{x/s^{i+1}(x'), y/s^i(0)\} \cdot \overset{\bullet}{true}, \dots \,\} \end{cases}$$

Note that solutions for *plus* have $\dot{s}$ everywhere, because s are introduced one at a time. For a rule like $doub(x) \to s(s(doub(x)))$, where two s are introduced in each step, solutions are instead $\{\bot, \dot{s}(s(\bot)), \dot{s}(s(\dot{s}(s(\bot)))), \ldots\}$.

An important ingredient of the definition of our semantics is the evaluation of concrete terms occurring in the body of a program rule w.r.t. a current interpretation as defined next. Note that the evaluation of a constructor symbol can be considered as a degenerate case of a defined symbol that is just interpreted as itself. Thus, in order to have much compact definitions, in the following we implicitly extend any interpretation $\mathcal{I}$ to constructors as $\mathcal{I}(c(x_1, \ldots, x_n)) := \{\varepsilon \cdot c(x_1, \ldots, x_n)\}$.

Definition 3 (Evaluation of terms). *Given an interpretation $\mathcal{I}$ we define by syntactic induction the* semantic evaluation *of a term t in $\mathcal{I}$ as $\mathcal{E}[\![x]\!]_{\mathcal{I}} := \{\varepsilon \cdot x\}$,*

$$\mathcal{E}[\![f(\boldsymbol{t})]\!]_{\mathcal{I}} := \left\{ (\vartheta\eta)|_{\boldsymbol{t}} \cdot r\eta \;\middle|\; \begin{array}{l} \sigma_i \cdot s_i \in \mathcal{E}[\![t_i]\!]_{\mathcal{I}} \text{ for } i = 1, \ldots, n \\ \vartheta = mgu(\sigma_1, \ldots, \sigma_n),\ \mu \cdot r \ll \mathcal{I}(f(\boldsymbol{x})) \\ \exists \eta = \mathring{mgu}_{Var(r)}(f(\boldsymbol{x})\mu, f(\boldsymbol{s})\vartheta) \end{array} \right\} \tag{3.1}$$

where $\mathring{mgu}_V(t, s)$, for an annotation-free pattern t, a term s with no node in common with t and a set V of variables, is defined as the substitution σ_h associated to the mgu homomorphism h for t and s, $h\colon t \oplus s \to r$, such that the following conditions hold:

1. *every $p \in \mathcal{N}_r$ is annotated if and only if $h^{-1}(p)$ has an annotated node*
2. *for every $x \in Var(t) \setminus V$ if p is an annotated node of $\sigma_h(x)$ then*
   - *p is shared with $\sigma_h(y)$ for some $y \in V$, or*
   - *there exists $q \in \mathcal{N}_t \cap h^{-1}(p)$ such that $\mathcal{L}_t(q) \in \mathcal{C}$*

$\mathring{mgu}_V(t, s)$ is an extension of the standard most general unifier which takes into account annotations and a target set of variables $Var(t) \setminus V$. A variable in the target cannot be bound with an annotated term t', *unless* t' is unified with another non-variable term. Intuitively, when we perform $\mathring{mgu}_V(t, s)$, variables not in the target $Var(t) \setminus V$ are those which will be bound to outermost-needed terms in subsequent narrowing steps, thus we have to evaluate them. On the contrary, variables in the target should not be evaluated to preserve laziness.

Equation (3.1) is the core of all the evaluation: first it "mounts" together all possible contributes of subterms $\boldsymbol{t}$ ensuring that their local instantiations are compatible with each other, then it takes a fresh partial answer[1] $\mu \cdot r$ from $\mathcal{I}(f(\boldsymbol{x}))$ and performs unification respecting annotations by $\mathring{mgu}_{Var(r)}(f(\boldsymbol{x})\mu, f(\boldsymbol{s})\vartheta)$. This ensures that only the evaluations of terms $\boldsymbol{t}$ which are necessary to evaluate $f(\boldsymbol{t})$ are used. In other words, (3.1) uses the partial answer $\mu \cdot r \in \mathcal{I}(f(\boldsymbol{x}))$ as "big-step" rule for f, namely $f(\boldsymbol{x})\mu \to r$, moreover the existence of $\eta = \mathring{mgu}_{Var(r)}(f(\boldsymbol{x})\mu, f(\boldsymbol{s}))$ ensures that any annotated term in the codomain of η correspond to a term which is root-needed for $f(\boldsymbol{t})$.

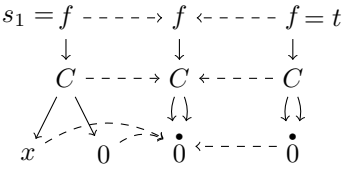
[1] Note that in the case the symbol f is a constructor c the partial answer is $\varepsilon \cdot c(\boldsymbol{x})$.

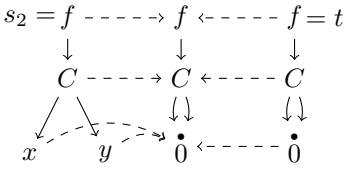
Example 2. Consider the following program and its interpretation $\mathcal{I}$:

```
coin = Z
coin = S Z
sub2 (S (S x)) = x
f (C x y) = 0
f (C x 0) = S x
pair(x) = C x x
```

$$I \coloneqq \begin{cases} \mathit{coin} \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{0},\ \varepsilon \cdot \dot{s}(0)\} \\ \mathit{sub2}(x) \mapsto \{\varepsilon \cdot \bot,\ \{x/s(s(x'))\} \cdot \dot{x'}\} \\ \mathit{pair}(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(x,x)\} \\ f(z) \mapsto \{\varepsilon \cdot \bot,\ \{z/C(x,y)\} \cdot \dot{0}, \\ \qquad\qquad \{z/C(x,0)\} \cdot \dot{S}(x)\} \end{cases}$$

For the goal term $g \coloneqq \mathit{pair}(\mathit{sub2}(s(\mathit{coin})))$, we have $\varepsilon \cdot \dot{C}(\dot{0}, \dot{0}) \in \mathcal{E}[\![g]\!]_{\mathcal{I}}$. In order to evaluate $\mathcal{E}[\![f(g)]\!]_{\mathcal{I}}$ we have to decide if there exist $\vartheta_1 \coloneqq \mathring{mgu}_{\{x\}}(s_1, t)$ and $\vartheta_2 \coloneqq \mathring{mgu}_{\emptyset}(s_2, t)$ respectively, where





we see that ϑ_1 exists and is equal to $\{x/\dot{0}\}$, while ϑ_2 does not exists. Even if t and s_2 unify in the standard sense, x and y have to be bound to an annotated term, namely $\dot{0}$, but y is not in $V = \emptyset$. □

Now we can define our concrete semantics.

Definition 4 (Concrete Fixpoint Semantics). *Let $\mathcal{I}$ be an interpretation and $\mathcal{R}$ be a program. The immediate consequences operator is*

$$\mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{I}} \coloneqq \lambda f(\boldsymbol{x}).\ \{\varepsilon \cdot \bot\} \cup \left\{ (\{\boldsymbol{x}/\boldsymbol{t}\}\sigma)\restriction_{\boldsymbol{x}} \cdot \dot{s} \;\middle|\; \begin{array}{l} f(\boldsymbol{t}) \to r \ll \mathcal{R}, \\ \sigma \cdot s \in \mathcal{E}[\![r]\!]_{\mathcal{I}},\ s \neq \bot \end{array} \right\}$$

Since $\mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{I}}$ is continuous, we can define our fixpoint semantics as $\mathcal{F}[\![\mathcal{R}]\!] \coloneqq \mathit{lfp}\,(\mathcal{P}[\![\mathcal{R}]\!]) = \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}\omega$.

Example 3. Let $\mathcal{R}$ be the program

```
from n = n : from (S n)
take Z _ = []
take (S n) (x:xs) = x : take n xs
m1 = take (S (S Z)) (from n) where n free
m2 = take (S (S Z)) (from coin)
```

The second iteration of the fixpoint computation $\mathcal{P}[\![\mathcal{R}]\!]\uparrow 2$ is

$$\begin{cases} \mathit{from}(n) \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}s(n){:}\bot\} \\ \mathit{coin} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot 0,\ \varepsilon\cdot s(0)\} \\ \mathit{take}(n,xs) \mapsto \{\varepsilon\cdot\bot,\ \{n/0\}\cdot[\,],\ \{n/s(n'),xs/x'{:}xs'\}\cdot x'{:}\bot, \\ \qquad\qquad \{n/s(0),xs/x'{:}xs'\}\cdot x'{:}[\,], \\ \qquad\qquad \{n/s(s(n'')),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}\bot\} \\ \mathit{m1} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot\} \\ \mathit{m2} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot\bot{:}\bot,\ \varepsilon\cdot 0{:}\bot,\ \varepsilon\cdot s(0){:}\bot\} \end{cases}$$

while the resulting semantics $\mathcal{F}[\![\mathcal{R}]\!]$ is

$$\begin{cases} \mathit{from}(n) \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}s(n){:}\bot,\ \varepsilon\cdot n{:}s(n){:}s(s(n)){:}\bot,\dots\} \\ \mathit{coin} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot 0,\ \varepsilon\cdot s(0)\} \\ \mathit{take}(n,xs) \mapsto \{\varepsilon\cdot\bot,\{n/0\}\cdot[\,],\ \{n/s(n'),xs/x'{:}xs'\}\cdot x'{:}\bot, \\ \qquad\qquad \{n/s(0),xs/x'{:}xs'\}\cdot x'{:}[\,], \\ \qquad\qquad \{n/s(s(n'')),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}\bot, \\ \qquad\qquad \{n/s(s(0)),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}[\,],\ \dots\} \\ \mathit{m1} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}s(n){:}\bot,\ \varepsilon\cdot n{:}s(n){:}[\,]\} \\ \mathit{m2} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot\bot{:}\bot,\ \varepsilon\cdot\bot{:}s(\bot){:}\bot,\ \varepsilon\cdot 0{:}\bot,\ \varepsilon\cdot 0{:}s(0){:}\bot, \\ \qquad \varepsilon\cdot 0{:}s(0){:}[\,],\ \varepsilon\cdot s(0){:}\bot,\ \varepsilon\cdot s(0){:}s(s(0)){:}\bot,\ \varepsilon\cdot s(0){:}s(s(0)){:}[\,]\} \end{cases}$$

It is worth noting that we do not require any conditions on programs. This is particularly important in view of application of the semantics in Abstract Diagnosis, where buggy programs cannot reasonably satisfy *a priori* any condition at all.

Theorem 1 (Correctness). *Let t be a term, and let $\mathcal{R}$ be a functional logic program. The following statements hold:*

1. *if $\sigma\cdot s_\bot \in \mathcal{E}[\![t]\!]_{\mathcal{F}[\![\mathcal{R}]\!]}$ then $\exists s \in \mathcal{T}(\Sigma,\mathcal{V})$ such that $t \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^{*}\ s$ and $\tau_\bot(s) = s_\bot$*
2. *if $t \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^{*}\ s$ then $\exists \vartheta \le \sigma$ and $s_\bot = \tau_\bot(s)\vartheta$ such that $\vartheta\cdot s_\bot \in \mathcal{E}[\![t]\!]_{\mathcal{F}[\![\mathcal{R}]\!]}$*

3.2 Abstraction Scheme

In this section, starting from the fixpoint semantics in Section 3.1, we develop an abstract semantics which approximates the observable behavior of the program. Program properties which can be of interest are Galois Insertions between the

concrete domain $\mathbb{C}_\Sigma$ and the abstract domain chosen to model the property. We assume familiarity with basic results of Abstract Interpretation [11].

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call an *incremental answer abstraction* that is a Galois Insertion $(\mathbb{C}_\Sigma, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \leq)$. This abstraction can by systematically lifted to a Galois Insertion $\mathbb{I} \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} [\mathbb{MGP} \to \mathbb{A}]$ by function composition (i.e., $\bar{\alpha}(f) = \alpha \circ f$).

Now we can derive the optimal abstract version of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ simply as $\mathcal{P}^\alpha[\![\mathcal{R}]\!] \coloneqq \bar{\alpha} \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \bar{\gamma}$. Abstract interpretation theory assures that $\mathcal{F}^\alpha[\![\mathcal{R}]\!] \coloneqq \mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow\omega$ is the best correct approximation of $\mathcal{F}[\![\mathcal{R}]\!]$. Correct means $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{F}^\alpha[\![\mathcal{R}]\!]$ and best means that it is the minimum (w.r.t. $\leq$) of all correct approximations. If $\mathbb{A}$ is Nötherian the abstract fixpoint is reached in a finite number of steps, that is, there exists a finite natural number h such that $\mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow\omega = \mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow h$.

A case study: The domain *depth(k)* An interesting finite abstraction of an infinite set of constructor terms are sets of terms up to a particular depth k, which has already been used in call-pattern analysis for functional-logic programs [13], the abstract diagnosis of functional programs [3] and logic programs [10] or in the abstraction of term rewriting systems [5] (with $k = 1$).

Now we show how to approximate an infinite set of incremental answer trees by means of a *depth*(k) cut $\lfloor\!\!\lfloor_k$ 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 $\widehat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent each term obtained by instantiating the variables of $\widehat{\mathcal{V}}$ with terms built over $\mathcal{V}$.

We extend $\lfloor\!\!\lfloor_k$ to partial answers $\{x_1/t_1, \ldots, x_n/t_n\} \cdot t_0$ essentially by cutting all t_i. However in case there is a shared term s between t_0 and another t_i and s is at depth greater than k in one of the two, in both terms we replace s with two (different) fresh variables from $\widehat{\mathcal{V}}$. Thus it can happen that variables from $\widehat{\mathcal{V}}$ appear at depth less than k.

For example, given $\alpha = \{x/s(x'), y/A(z, s(s(y)))\} \cdot \dot{B}(y, s(t), t, x')$ where $t = \dot{s}(s(\dot{z}))$, its cut $\alpha\lfloor\!\!\lfloor_2$ is $\{x/s(x'), y/A(\hat{v}_1, s(\hat{v}_2))\} \cdot \dot{B}(\hat{v}_3, s(\dot{\hat{v}}_4), \dot{s}(\dot{\hat{v}}_5), x')$.

We define the order $\leq$ for this domain by successive lifting of the order s.t. $\hat{x} \leq t$ for every term t and variable $\hat{x} \in \widehat{\mathcal{V}}$. First we extend $\leq$ by structural induction on the structure over terms, then we extend it to substitutions by pointwise extension and then over partial answers. Finally given two *depth*(k) partial answers trees T_1 and T_2, $T_1 \leq T_2$ iff for any $\alpha \in T_1$ exists a $\beta \in T_2$ such that $\alpha \leq \beta$. The set of all *depth*(k) partial answers trees ordered by $\leq$ is a complete lattice. Let $\bigvee$ be its join.

The *depth*(k) cut of a tree T is $\kappa(T) \coloneqq \bigvee\{\beta\lfloor\!\!\lfloor_k \mid \beta \in T\}$. For example, given $T = \{\varepsilon \cdot \bot,\ x/s^3(y) \cdot \dot{y},\ x/s^3(y) \cdot \dot{A}\}$, for $k = 2$, $\kappa(T) = \{\varepsilon \cdot \bot,\ x/s^3(y) \cdot \dot{y}\}$

The resulting (optimal) abstract immediate consequence operator is

$$\mathcal{P}^\kappa[\![\mathcal{R}]\!]_{I^\kappa} = \lambda f(\boldsymbol{x}).\ \{\varepsilon \cdot \bot\} \vee \bigvee \left\{ ((\{\boldsymbol{x}/\boldsymbol{t}\}\sigma)\restriction_{\boldsymbol{x}} \cdot \dot{s})\lfloor\!\!\lfloor_k \;\middle|\; \begin{array}{l} f(\boldsymbol{t}) \to r \ll \mathcal{R}, \\ \sigma \cdot s \in \mathcal{E}^\kappa[\![r]\!]_{I^\kappa},\ s \neq \bot \end{array} \right\}$$

where $\mathcal{E}^\kappa[\![x]\!]_{I^\kappa} \coloneqq \{\varepsilon \cdot x\}$ and

$$\mathcal{E}^\kappa[\![f(\boldsymbol{t})]\!]_{I^\kappa} \coloneqq \left\{ (\vartheta\eta)\restriction_{\boldsymbol{t}} \cdot r\eta \;\middle|\; \begin{array}{l} \sigma_i \cdot s_i \in \mathcal{E}^\kappa[\![t_i]\!]_{I^\kappa} \text{ for } i = 1, \ldots, n \\ \vartheta = mgu(\sigma_1, \ldots, \sigma_n),\ \mu \cdot r \ll I^\kappa(f(\boldsymbol{x})) \\ \exists \eta = \mathring{mgu}_{Var(r) \cup \widehat{\mathcal{V}}}(f(\boldsymbol{x})\mu, f(\boldsymbol{s})\vartheta) \end{array} \right\}$$

Example 4. Consider the program in Example 3 and take $k = 2$. According to the previous definition, the abstract semantics of function *from* is:

$$\left\{ \begin{array}{l} from(n) \mapsto \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot n \mathbin{:\!\!\bullet} \bot,\ \varepsilon \cdot n \mathbin{:\!\!\bullet} s(\hat{x}_1) \mathbin{:\!\!\bullet} \bot, \\ \qquad\qquad \varepsilon \cdot n \mathbin{:\!\!\bullet} s(\hat{x}_1) \mathbin{:\!\!\bullet} \hat{x}_2 \mathbin{:\!\!\bullet} \hat{x}_3,\ \varepsilon \cdot n \mathbin{:\!\!\bullet} s(\hat{x}_1) \mathbin{:\!\!\bullet} \hat{x}_2 \mathbin{:\!\!\bullet} \hat{x}_3 \,\} \end{array} \right.$$

Example 5. Consider the program $\mathcal{R}_{plus}$ of Example 1. For $k = 1$ $\mathcal{F}^\kappa[\![\mathcal{R}_{plus}]\!]$ is

$$\left\{ plus(x, y) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \overset{\bullet}{y},\ \{x/s(\hat{x}_1)\} \cdot \overset{\bullet}{s}(\hat{x}_2),\ \{x/s(\hat{x}_1)\} \cdot \overset{\bullet}{s}(\overset{\bullet}{\hat{x}_2}) \,\} \right.$$

This is essentially the same result that can be obtained by [5], with the *head* upper-closure operator, that derives the following abstract TRS:

$$\begin{array}{ll} plus^a(0^a, 0^a) \to 0^a & plus^a(0^a, s^a(\top_{nat})) \to s^a(\top_{nat}) \\ plus^a(s^a(\top_{nat}), 0^a) \to s^a(\top_{nat}) & plus^a(s^a(\top_{nat}), s^a(\top_{nat})) \to s^a(\top_{nat}) \end{array}$$

Note that the first two rules of the abstract TRS are subsumed by the partial answer $\{x/0\} \cdot \overset{\bullet}{y}$ in our semantics. □

Example 6. Let $\mathcal{R}$ be the program obtained by adding to the one of Example 2 the following rules

```
main = disj m1                    disj (C 0 (S x)) = True
m1 = pair (sub2 (S coin))
```

For $k = 2$, the abstract analysis of $\mathcal{R}$ reaches the fixpoint in 3 steps, giving

$$\left\{ \begin{array}{l} sub2(x) \mapsto \{\varepsilon \cdot \bot,\ x/s(s(\hat{x}_1)) \cdot \overset{\bullet}{\hat{x}_2}\} \\ pair(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \overset{\bullet}{C}(x, x)\} \\ m1 \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \overset{\bullet}{C}(\bot, \bot),\ \varepsilon \cdot \overset{\bullet}{C}(\overset{\bullet}{\hat{x}_2}, \overset{\bullet}{\hat{x}_2})\} \\ main \mapsto \{\varepsilon \cdot \bot\} \end{array} \right.$$

For the same program and same k [13] reaches the call pattern $disj(\top, \top) \doteq true$ causing $main \doteq true$ to be observed, which does not correspond to a concrete call pattern. However for $k \geq 3$ this false call pattern is not observed. □

It is worth noting that the resulting abstract semantics encompasses some limitations of previous works

– Since we use a "truly" goal-independent concrete semantics we obtain a much compact abstract semantics than [3]. This is exactly the reason why we are developing a compact semantics for generic TRS [2].
– For $k = 1$ if we consider TRS admissible to apply the technique of [5] we obtain the same results. However the abstract rewriting methodology of [5] requires canonicity, stratification, constructor discipline, and complete definedness for the analyses. This class of TRS is very restricted (even for functional programming) and certainly cannot cover functional-logic programs. On the contrary we have no restrictions at all.
– Since we use a careful definition of the abstraction function that uses $depth(k)$ variables instead of just a $\top$ symbol like does [13] we have some slightly better results. For the same k we do not produce all false answers which are produced by [13]. These answers won't be generated by [13] for $k + 1$, but due to the quickly growing size of $depth(k)$ our improvement can be worthy.

All examples we have showed are obtained by a proof of concept prototype.

4 Abstract diagnosis of functional-logic programs

Now, following the approach of [10], we define abstract diagnosis of functional-logic programs. The framework of abstract diagnosis [10] comes from the idea of considering the abstract versions of Park's Induction Principle[2]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. However, in the general case, diagnosing w.r.t. *abstract* program properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

There have been several proposal about Declarative Diagnosis of functional-logic languages, like [14,8,7,9]. The approach has revealed much more problematic than the logic paradigm. As can be read from [8] "A more practical problem with existing debuggers for lazy functional (logic) languages is related to the presentation of the questions asked to the oracle". Actually the call-time choice model and the peculiarity of needed narrowing strategy cannot be tackled by pure declarations about expected answers. Roughly speaking, the oracle must "understand" neededness. As already noted by [8], the "simplification" proposed in [14] to adopt the generic approach to FLP is not well-suited. [8] aims at a debugging method that asks the oracle about computed answers that do not involve function calls, plus possible occurrences of an "undefined symbol". However this is exactly the kind of information we have in our concrete semantics, which makes it a suitable starting point for our diagnosis methodology.

In the following, $\mathcal{I}^\alpha$ is the specification of the intended behavior of a program w.r.t. the property α.

Definition 5. *Let $\mathcal{R}$ be a program and α be a property.*

[2] a concept of formal verification that is undecidable in general

1. $\mathcal{R}$ *is (abstractly)* partially correct *w.r.t.* $\mathcal{I}^\alpha$ *if* $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{I}^\alpha$.
2. $\mathcal{R}$ *is (abstractly)* complete *w.r.t.* $\mathcal{I}^\alpha$ *if* $\mathcal{I}^\alpha \leq \alpha(\mathcal{F}[\![\mathcal{R}]\!])$.
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 $\alpha(\mathcal{F}[\![\mathcal{R}]\!])$ and not in terms of the (possibly less precise) abstract semantics $\mathcal{F}^\alpha[\![\mathcal{R}]\!]$. $\mathcal{I}^\alpha$ is the abstraction of the intended concrete semantics of $\mathcal{R}$. Thus, the user 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 "originating" 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 element*.

Definition 6. *Let $\mathcal{R}$ be a program, R a rule, $\mathcal{I}^\alpha$ an interpretation and $e \in \mathbb{A}$.*
R is abstractly incorrect *if $\mathcal{P}^\alpha[\![\{R\}]\!]_{\mathcal{I}^\alpha} \not\leq \mathcal{I}^\alpha$.*
e is an uncovered element *if $e \leq \mathcal{I}^\alpha$ and $e \not\leq \mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{I}^\alpha}$.*

Informally, R is abstractly incorrect if it derives a wrong abstract element from the intended semantics. e is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking these conditions requires one application of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ to $\mathcal{I}^\alpha$, while the standard detection based on symptoms would require the construction of $\alpha(\mathcal{F}[\![\mathcal{R}]\!])$ and therefore a fixpoint computation.

It is worth noting that correctness and completeness are defined in terms of $\alpha(\mathcal{F}^\alpha[\![\mathcal{R}]\!])$, 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 $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$). The issue of the precision of the abstract semantics becomes therefore relevant in establishing the relation between the two concepts.

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.

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 warning about a possible source of errors. Because of the approximation, it can happen that a (concretely) correct rule is abstractly incorrect.

However all concrete errors that are "visible"[3] in the abstract domain are detected as they lead to an abstract incorrectness.

[3] A concrete symptom is visible if its abstraction is different from the abstraction of correct answers. For example if we abstract to the length of lists, an incorrect rule producing wrong lists of the same length of the correct ones is not visible.

Theorem 4. *Let r be a rule, $\mathcal{I}$ a concrete specification. If $\mathcal{P}[\![\{r\}]\!]_{\mathcal{I}} \not\sqsubseteq \mathcal{I}$ and $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{I}}) \not\leq \alpha(\mathcal{I})$ then r is abstractly incorrect w.r.t. $\alpha(\mathcal{I})$.*

The diagnosis w.r.t. approximate properties over Nötherian domains is always effective, because the abstract specification is finite. However, 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.
- there exists no sufficient condition for completeness.

It is important to note that our method, since it has been derived by (properly) applying Abstract Interpretation techniques, is correct by construction.

We now show the *depth*(k) instance of our methodology to provide a concrete application of our Abstract Diagnosis framework. This instance turns out to encompass several proposal for Declarative Debugging. It will thus be interesting in the future to experiment with other possible instances over more sophisticated domains.

Our case study: Abstract Diagnosis on *depth*(k) Now we show how we can derive an efficacious debugger by choosing suitable instances of the general framework described above. We consider the *depth*(k) abstraction κ presented in Section 3.2.

Note that *depth*(k) partial answers that do not contain variables belonging to $\widehat{\mathcal{V}}$ actually are concrete answers. Thus an additional benefit w.r.t. the general results is that errors exhibiting symptoms without variables belonging to $\widehat{\mathcal{V}}$ are actually concrete (real) errors.

Example 7. Consider a buggy program for *from* (proposed in [7]) with rule R: $\mathit{from}(n) \rightarrow n : \mathit{from}(n)$. Use as intended specification $\mathcal{S}^{\kappa}$ the fixpoint from Example 4. For $k = 3$ we detect that rule R is abstractly incorrect since

$$\mathcal{P}^{\kappa}[\![\{R\}]\!]_{\mathcal{S}^{\kappa}} = \begin{cases} \mathit{from}(n) \mapsto \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot n \dot{:} \bot,\ \varepsilon \cdot n \dot{:} n \dot{:} \bot, \\ \qquad\qquad \varepsilon \cdot n \dot{:} n \dot{:} \hat{x}_2 \dot{:} \hat{x}_3,\ \varepsilon \cdot n \dot{:} n \dot{:} \hat{x}_2 \dot{:} \hat{x}_3 \,\} \end{cases} \not\leq \mathcal{S}^{\kappa}$$

Example 8. Consider the buggy program $\mathcal{R}_{bug}$

```
main = C (h (f x)) x
h (S x) = Z                                   R:   f x = S (h x)
```

where rule R should have been f (S x) = S Z to be correct w.r.t. the intended semantics on *depth*(k), with $k > 2$,

$$\mathcal{I}^{\kappa} = \begin{cases} f(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{s}(\bot),\ \{x/s(x')\} \cdot \dot{s}(\dot{0})\} \\ h(x) \mapsto \{\varepsilon \cdot \bot,\ \{x/s(x')\} \cdot \dot{0}\} \\ \mathit{main} \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(\bot, x),\ \varepsilon \cdot \dot{C}(\dot{0}, x)\} \end{cases}$$

This error preserves the computed answer behavior both for h and f, but not for *main*. In fact, *main* evaluates to $\varepsilon \cdot C(0, s(x'))$. Rule R is abstractly incorrect. The diagnosis method of [9,7] does not report incorrectness errors (as there are no incorrectness symptoms) while it reports the missing answer $\varepsilon \cdot C(0,0)$. □

Example 9. Consider the following bugged program for *double* w.r.t. the *depth*(2) specification $\mathcal{S}^\kappa$.

```
R1: double Z   = s Z
R2: double (S x) = S (double x)
```

$$\mathcal{S}^\kappa := \left\{ \begin{array}{l} double(x) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{0},\ \{x/s(x')\} \cdot \dot{s}(s(\hat{y})), \\ \qquad\qquad \{x/s(0)\} \cdot \dot{s}(s(\dot{\hat{y}})),\ \{x/s(s(\hat{x}_1)\} \cdot \dot{s}(s(\dot{\hat{y}}))\} \end{array} \right.$$

We can detect that both R_1 and R_2 are abstractly incorrect, since

$$\mathcal{P}^\kappa[\![\{R_1\}]\!]_{\mathcal{S}^\kappa} = \{\varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{s}(0)\}$$

$$\mathcal{P}^\kappa[\![\{R_2\}]\!]_{\mathcal{S}^\kappa} = \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot \dot{s}(\bot),\ \{x/s(0)\} \cdot \dot{s}(\dot{0}) \quad \{x/s(s(\hat{x}_1)\} \cdot \dot{s}(\dot{s}(\dot{\hat{y}}))\ \{x/s(s(\hat{x}_1)\} \cdot \dot{s}(\dot{s}(\dot{\hat{y}}))\}$$

However the debugger in [3] is not able to determine for $k = 2$ that rule R_2 is incorrect. □

We run out prototype on the possible benchmarks from the NOFIB-Buggy collection available at http://einstein.dsic.upv.es/nofib-buggy. Not all benchmarks can be checked yet since errors are in rules using higher-order or I/O and arithmetical features, which our semantic framework can not handle. However, since our methodology processes each rule independently, we can anyway find errors in the first-order rules of general programs. In fact we discovered uncovered elements at depth-4 in `boyer`; we detected at depth-1 the incorrect rules of `clausify` and `lift`[4]; we found some uncovered elements for `knights`, `pretty`, `sorting` and `fluid`[5] at depth-1, depth-8, depth-3 and depth-2, respectively.

The selection of the appropriate depth for the abstraction is a sensitive point of our approach since the abstraction might not be precise enough. For example if we consider $k = 2$ in Example 7 we do not detect the incorrectness.

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.

Is is important to note that the resulting abstract diagnosis encompasses some limitations of previous works on declarative diagnosis

– Symptoms involving infinite answers cannot be directly tackled by [9,7], while, if the error manifests in the first part of the infinite answer we detect it (Example 7).

[4] An adjustment for some rules was needed in order to simulate Haskell selection rule.
[5] Minor changes has been done to replace arithmetic operations with Peano's notation.

– If we just compare the actual and the intended semantics of a program some incorrectness bugs can be "hidden" because of an incompleteness bug. Our technique does not suffer of error interference, as each possible source of error is checked in isolation (Example 8). Moreover we detect all errors simultaneously.

As a final remark about applicability of the current proposal, let note that we can tackle higher-order features and primitive operations of functional (logic) languages in the same way proposed by [13]: by using the technique known as "defunctionalization" and by approximating, very roughly, calls to primitive operations, that are not explicitly defined by rewrite rules, just with variables over $\widehat{\mathcal{V}}$.

5 Conclusions

We have presented a generic scheme for the declarative debugging of functional programs. Our approach is based on the ideas of [3,10,1] which we apply to the diagnosis of functional-logic programs. We have presented a fixpoint semantics $\mathcal{P}[\![\mathcal{R}]\!]$ for functional programs. Our semantics allows us to model correctly the typical cogent features of modern functional-logic languages (non-deterministic, non-strict functions defined by non-confluent programs and call-time choice behaviour).

Then, we formulated an efficacious debugging methodology based on abstract interpretation which proceeds by approximating the $\mathcal{P}[\![\mathcal{R}]\!]$ operator by means of an abstract $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ operator. We showed that, given the abstract intended specification $\mathcal{I}^\alpha$ of the semantics of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single step of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ and, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

We presented a particular instance of our methodology in order to compare our proposal with [5,8,7,9,13] and showed several examples where we have better results.

For future work, we intend to extend the concrete semantics to directly incorporate higher-order functions and residuation, in order to develop Abstract Diagnosis for full multi-paradigm languages like Curry.

References

1. M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. An Integrated Framework for the Diagnosis and Correction of Rule-Based Programs. *Theoretical Computer Science*, 2010. To Appear.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and J. Iborra. A Compact Fixpoint Semantics for Term Rewriting Systems. *Theoretical Computer Science*, 2010. To Appear.
3. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected*

Papers, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.

4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
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. B. Braßel. A Technique to build Debugging Tools for Lazy Functional Logic Languages. In M. Falaschi, editor, *Proceedings of the 17th Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, pages 63–76, 2008.
7. R. Caballero-Roldán. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 8–13, New York, NY, USA, 2005. ACM Press.
8. R. Caballero-Roldán, F. J. López-Fraguas, and M. Rodríquez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proceedings of Fifth International Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 170–184, Berlin, 2001. Springer-Verlag.
9. R. Caballero-Roldán, M. Rodríquez-Artalejo, and R. del Vado-Vírseda. Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In *Proceedings of 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321, Berlin, 2008. 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. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press.
12. R. Echahed and Janodet J.-C. Admissible Graph Rewriting and Narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325–340. MIT Press, 1998.
13. M. Hanus. Call pattern analysis for functional logic programs. In *Proc. of the 10th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008.
14. L. Naish and Timothy Barbour. A Declarative Debugger for a Logical-Functional Language. *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2:91–99, 1995.
15. C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399, Berlin, 1990. Springer-Verlag.

Simple Functional Programs for Computing Reflexive-transitive Closures

Rudolf Berghammer and Sebastian Fischer

Institut für Informatik, Universität Kiel, 24098 Kiel, Germany

Abstract. We show how to derive simple functional algorithms for computing reflexive-transitive closures. The original relational description employs the notion of rectangles and leads to an algorithm schema. Using data refinement, we develop a simple Haskell program for a specific instantiation and show that it has cubic run time like Warshall's standard algorithm for transitive closures.

1 Introduction

The reflexive-transitive closure of a directed graph G is a directed graph with the same vertices as G that contains an edge from each vertex x to each vertex y if and only if y is reachable from x in G. We can represent any directed graph as (binary) relation R that relates two vertices x and y if and only if there is an edge from x to y. The reflexive-transitive closure of a graph G corresponds to the reflexive-transitive closure R^* of the binary relation R which represents G, i.e., to the least reflexive and transitive relation that contains R.

Usually, the task of computing R^* is solved by a variant of Warshall's algorithm which represents R as a Boolean matrix such that each entry that is 1 represents an edge of the graph. It first computes the transitive closure R^+ of R using Warshall's original method [8] and then obtains R^* from R^+ by putting all entries of the main diagonal of R^+ to 1. Representing the graph as a 2-dimensional Boolean array leads to a simple and efficient imperative program with three nested loops for the computation of R^+ and a single loop for the subsequent treatment of the main diagonal, i.e., to a program of time complexity $O(n^3)$ with n as cardinality of the carrier set of R.

However, in certain cases arrays are unfit for representing relations. In particular this holds if both R and R^* are of "medium density" or even sparse. Here a representation of relations by, say, successor lists, lists of pairs or even look-up tables (i.e., characteristic functions) is much more economic. But such a representation sacrifices the simplicity and efficiency of the above mentioned imperative implementation. Moreover, the traditional method of imperatively updating an array representing the graph is alien to the purely functional programming paradigm which restricts the use of side effects.

We present an alternative method for computing reflexive-transitive closures. Using relation algebra as the methodical tool, we develop a generic algorithm for computing R^*, that is based on the decomposition of R by so-called rectangles,

and provide some instantiations. Each concrete algorithm – we now speak of a program – is determined by a specific choice of the rectangles. Due to its generality, our generic algorithm not only allows simple and efficient imperative array-based implementations, but also purely functional programs based on a very simple representation of relations via successor lists. In this paper we focus on functional programming.

2 Relation-algebraic Preliminaries

We denote the set (or type) of all relations with source X and target Y by $[X \leftrightarrow Y]$ and write $R : X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$. If the sets X and Y are finite, we may consider R as a Boolean matrix with $|X|$ rows and $|Y|$ columns. Since this interpretation is well suited for many purposes, we will often use matrix notation and terminology in this paper. In particular, we talk about rows, columns and entries of relations.

We assume the reader to be familiar with the basic operations on relations, viz. R^T (transposition), $\overline{R}$ (complement), $R \cup S$ (join), $R \cap S$ (meet), $R;S$ (composition), the predicate indicating $R \subseteq S$ (inclusion) and the special relations O (empty relation), L (universal relation) and I (identity relation). Furthermore, we assume the reader to know the most fundamental laws of relation algebra like $R^{\mathsf{T}\mathsf{T}} = R$, $(R;S)^\mathsf{T} = S^\mathsf{T};R^\mathsf{T}$, $R;(S \cup T) = R;S \cup R;T$, and the *Schröder rule* stating the equivalence of $Q;R \subseteq S$, $Q^\mathsf{T};\overline{S} \subseteq \overline{R}$ and $\overline{S};R^\mathsf{T} \subseteq \overline{Q}$. We will also use the relation-algebraic specifications of the following properties: reflexivity $\mathsf{I} \subseteq R$, transitivity $R;R \subseteq R$, injectivity $R;R^\mathsf{T} \subseteq \mathsf{I}$ and surjectivity $\mathsf{L};R = \mathsf{L}$. For more details, concerning the algebraic treatment of relations, we refer to [6].

A *vector* is a relation v which satisfies the equation $v = v;\mathsf{L}$ and a *point* is an injective and surjective vector. If p is a point, then we have $p \neq \mathsf{O}$, $\overline{R;p} = \overline{R};p$, and that $R \subseteq S;p$ is equivalent to $R;p^\mathsf{T} \subseteq S$; see [6]. For vectors the targets are irrelevant. We, therefore, consider in the following mostly vectors $v : X \leftrightarrow \mathbf{1}$ with a specific singleton set $\mathbf{1} = \{\bot\}$ as target and omit in such cases the second element $\bot$ in a pair, i.e., write $x \in v$ instead of $(x, \bot) \in v$. A vector from $[X \leftrightarrow \mathbf{1}]$ can be considered as a Boolean matrix with $|X|$ rows and exactly one column, i.e., as a Boolean column vector, and *represents* the subset $\{x \in X \mid x \in v\}$ of X. In the Boolean matrix model a point from $[X \leftrightarrow \mathbf{1}]$ is a Boolean column vector in which exactly one entry is 1. This means that points represent singleton sets, or elements if we identify a singleton set with the only element it contains. Later we will use that if the point $p : X \leftrightarrow \mathbf{1}$ represents $x_1 \in X$ and the point $q : X \leftrightarrow \mathbf{1}$ represents $x_2 \in X$, then $(y, z) \in p;q^\mathsf{T}$ is equivalent to $x_1 = y$ and $x_2 = z$.

3 Rectangles and Reflexive-transitive Closures

Given a relation $R : X \leftrightarrow X$, its *reflexive-transitive closure* $R^* : X \leftrightarrow X$ is defined as the least reflexive and transitive relation that contains R and its *transitive closure* $R^+ : X \leftrightarrow X$ is defined as the least transitive relation that contains R.

It is well-known (cf. [6, 1]) that R^* and R^+ can also be specified via least fixed point constructions. The least fixed point of $\tau_R : [X \leftrightarrow X] \to [X \leftrightarrow X]$, where $\tau_R(Q) = \mathsf{I} \cup R;Q$, is R^*, and the least fixed point of $\sigma_R : [X \leftrightarrow X] \to [X \leftrightarrow X]$, where $\sigma_R(Q) = R \cup R;Q$, is R^+. From these specifications, we obtain by fixed point considerations that $\mathsf{O}^* = \mathsf{I}$, $R^* = \mathsf{I} \cup R^+$ and $(R \cup S)^* = R^*;(S;R^*)^*$. The last equation is called the *star-decomposition* rule [1]. A simple fixed point argument also shows that R is transitive if and only if $R = R^+$.

3.1 A Generic Algorithm ror Reflexive-transitive Closures

Now, assume two relations $R : X \leftrightarrow X$ and $S : X \leftrightarrow X$ of the same type. Then we have the following decomposition property for reflexive-transitive closures that is decisive for the remainder of the paper.

$$S;\mathsf{L};S \subseteq S \implies (R \cup S)^* = R^* \cup R^*;S;R^* \tag{1}$$

A proof of (1) starts with the following calculation that applies the left-hand side (the assumption on S) in the last step: $S;R^*;S;R^* \subseteq S;\mathsf{L};S;R^* \subseteq S;R^*$. This is the transitivity of $S;R^*$, which yields $(S;R^*)^+ = S;R^*$. Using this equation and star decomposition, we can prove the right-hand side of (1) as follows.

$$(R \cup S)^* = R^*;(S;R^*)^* = R^*;(\mathsf{I} \cup (S;R^*)^+) = R^*;(\mathsf{I} \cup S;R^*) = R^* \cup R^*;S;R^*$$

A relation $S : X \leftrightarrow X$ such that $S;\mathsf{L};S \subseteq S$ is called a *rectangle*. In set-theoretic notation this means that it is of the form $A \times B$, where A and B are subsets of X, and in Boolean matrix terminology this means that a permutation of rows and columns transforms S into a form such that the 1-entries form a rectangular block. If we assume a function *rectangle* to be at hand that yields for a given non-empty relation R a non-empty rectangle S such that $S \subseteq R$, then a combination of the right-hand side of (1) with the equations $R = (R \cap \overline{S}) \cup S$ and $\mathsf{O}^* = \mathsf{I}$ leads to the following recursive algorithm schema for computing R^* that uses a `let`-clause to avoid the multiple computation of $(R \cap \overline{S})^*$.

$$\begin{array}{l} rtc(R) = \texttt{if } R = \mathsf{O} \texttt{ then } \mathsf{I} \\ \qquad\qquad \texttt{else let } S = rectangle(R) \\ \qquad\qquad\qquad\quad\;\; C = rtc(R \cap \overline{S}) \\ \qquad\qquad\quad\;\; \texttt{in } \; C \cup C;S;C \end{array} \tag{2}$$

If the input is finite, this program terminates since the arguments of the recursive calls strictly become smaller until the termination condition is fulfilled.

3.2 Some Instantiations

For giving concrete code for the choice of the rectangle S in (2), we assume $R : X \leftrightarrow X$ to be a non-empty relation. Then a first possibility to obtain a non-empty rectangle inside of R is as follows. Select a point $p : X \leftrightarrow \mathbf{1}$ such that

$p \subseteq R;\mathsf{L}$. After that, select a point $q : X \leftrightarrow \mathbf{1}$ such that $q \subseteq R^{\mathsf{T}};p$. By means of the points p and q, finally, define S as $p;q^{\mathsf{T}} : X \leftrightarrow X$.

Using graph-theoretic terminology, the point p represents a vertex x of the directed graph $G = (X,R)$ with set X of vertices and set R of edges that possesses at least one successor, the vector $R^{\mathsf{T}};p$ represents the successor set $succ_R(x) := \{y \in X \mid (x,y) \in R\}$ of x and the point q represents an element of $succ_R(x)$, that is, a single successor of x. Here is the correctness proof for the above procedure. The vector property of p implies $S;\mathsf{L};S = p;q^{\mathsf{T}};\mathsf{L};p;q^{\mathsf{T}} \subseteq p;\mathsf{L};q^{\mathsf{T}} = p;q^{\mathsf{T}} = S$, i.e., that S indeed is a rectangle. Nonemptiness of S follows from the fact that p and q are points, since the assumption $S = p;q^{\mathsf{T}} = \mathsf{O}$ is equivalent to $p \subseteq \mathsf{O};q$ (see Section 2), i.e., contradicts $p \neq \mathsf{O}$. Finally, if we combine the assumption $q \subseteq R^{\mathsf{T}};p$ with the injectivity of p, this yields that S is contained in R by $S = p;q^{\mathsf{T}} \subseteq p;(R^{\mathsf{T}};p)^{\mathsf{T}} = p;p^{\mathsf{T}};R^{\mathsf{T}^{\mathsf{T}}} \subseteq R$,

Let us assume a pre-defined operation *point* for the selection of a point from a non-empty vector to be available as, for instance, in the programming language of the Kiel relation algebra tool RelView [3]. Then the above approach leads to the following refinement of (2).

$$\begin{array}{l} rtc(R) = \texttt{if } R = \mathsf{O} \texttt{ then } \mathsf{I} \\ \qquad\qquad \texttt{else let } p = point(R;\mathsf{L}) \\ \qquad\qquad\qquad\quad\;\; q = point(R^{\mathsf{T}};p) \\ \qquad\qquad\qquad\quad\;\; S = p;q^{\mathsf{T}} \\ \qquad\qquad\qquad\quad\;\; C = rtc(R \cap \overline{S}\,) \\ \qquad\qquad\qquad\;\; \texttt{in } \; C \cup C;S;C \end{array} \tag{3}$$

Because p and q are selected as points, the rectangle $S := p;q^{\mathsf{T}}$ is an atom of the ordered set $([X \leftrightarrow X], \subseteq)$. The latter means that it contains exactly one pair, or, in Boolean matrix terminology, exactly one entry is 1. So, the (proper) subrelation $R \cap \overline{S}$ of R is obtained by taking off from R the single pair (x,y), with $x \in X$ represented by p and $y \in X$ represented by q. Due to this property, (3) leads to $|R|$ recursive calls.

For presenting a second possibility to obtain a rectangle inside of the given relation $R : X \leftrightarrow X$, we assume again that $p : X \leftrightarrow \mathbf{1}$ is a point contained in the vector $R;\mathsf{L}$. If we do not select a single point q from $R^{\mathsf{T}};p$ as above, but take the entire vector $R^{\mathsf{T}};p$ instead, then in combination with $p;(R^{\mathsf{T}};p)^{\mathsf{T}} = p;p^{\mathsf{T}};R^{\mathsf{T}^{\mathsf{T}}} = p;p^{\mathsf{T}};R$ this choice yields the following instantiation of (2). In graph-theoretic terminology, this means that instead of removing a single edge in each step we remove all edges that originate from a specific vertex.

$$\begin{array}{l} rtc(R) = \texttt{if } R = \mathsf{O} \texttt{ then } \mathsf{I} \\ \qquad\qquad \texttt{else let } p = point(R;\mathsf{L}) \\ \qquad\qquad\qquad\quad\;\; S = p;p^{\mathsf{T}};R \\ \qquad\qquad\qquad\quad\;\; C = rtc(R \cap \overline{S}\,) \\ \qquad\qquad\qquad\;\; \texttt{in } \; C \cup C;S;C \end{array} \tag{4}$$

Concerning correctness, the proof of the rectangle property of $S := p;p^{\mathsf{T}};R$ via the calculation $S;\mathsf{L};S = p;p^{\mathsf{T}};R;\mathsf{L};p;p^{\mathsf{T}};R \subseteq p;\mathsf{L};p^{\mathsf{T}};R = p;p^{\mathsf{T}};R = S$ applies

again that p is a vector. To show nonemptiness of the rectangle S, we assume $S = \mathsf{O}$. Starting with this, we get

$$\begin{array}{lll} p;p^\mathsf{T};R = \mathsf{O} & \iff & (p;p^\mathsf{T})^\mathsf{T};\mathsf{L} \subseteq \overline{R} \qquad \text{Schröder rule} \\ & \iff & p;p^\mathsf{T};\mathsf{L} \subseteq \overline{R} \\ & \iff & p;\mathsf{L} \subseteq \overline{R} \qquad p \text{ surjective} \\ & \iff & R;\mathsf{L} \subseteq \overline{p} \qquad \text{Schröder rule}, \mathsf{L} = \mathsf{L}^\mathsf{T} \\ & \iff & p \subseteq \overline{R;\mathsf{L}} \end{array}$$

so that, together with $p \subseteq R;\mathsf{L}$, we arrive at the contradiction $p = \mathsf{O}$. Finally, $S = p;p^\mathsf{T};R \subseteq R$ immediately follows from the injectivity of p.

If the point p represents the element $x \in X$, then the rectangle $S = p;p^\mathsf{T};R$ is obtained from R by, as [6] calls it, "singling out" the x-row of R. This means that the x-row of S coincides with the x-row of R and all other entries of S are 0. Hence, $R \cap \overline{S}$ is obtained from R by "zeroing out" the x-row. As a consequence, (4) leads to at most $|X|$ recursive calls.

A further instantiation of (2) can be obtained if we select a *maximal rectangle* contained in the relation $R : X \leftrightarrow X$. As shown in [7], every point $p : X \leftrightarrow X$ that is contained in $R;\mathsf{L}$ gives rise to a specific maximal rectangle inside R started horizontally by $S := \overline{\overline{R};R^\mathsf{T}};p;p^\mathsf{T};R = \overline{\overline{R};R^\mathsf{T};p};(R^\mathsf{T};p)^\mathsf{T}$. Using this rectangle and a variable v within the `let`-binding to avoid the two-fold evaluation of $R^\mathsf{T};p$, we obtain the following program for computing R^*.

$$\begin{array}{lll} rtc(R) = \texttt{if } R = \mathsf{O} \texttt{ then } \mathsf{I} & & \\ \qquad \texttt{else let } & p = point(R;\mathsf{L}) & \\ & v = R^\mathsf{T};p & \\ & S = \overline{\overline{R};v};v^\mathsf{T} & \\ & C = rtc(R \cap \overline{S}) & \\ \qquad \texttt{in } & C \cup C;S;C & \end{array} \tag{5}$$

With the help of the Schröder rule we get $\mathsf{I} \subseteq \overline{\overline{R};R^\mathsf{T}}$ from $\mathsf{I};R \subseteq R$, so that the rectangle used in (4) is contained in that used in (5). As a consequence, the number of recursive calls of (5) is bounded by the number of recursive calls of (4). As experiments with RelView have shown, in practice (5) frequently leads to fewer recursive calls than (4). This is due to the fact that, with $x \in X$ represented by $p : X \leftrightarrow \mathbf{1}$, the step from R to $R \cap \overline{S}$, where $S = \overline{\overline{R};R^\mathsf{T}};p;p^\mathsf{T};R$, not only zeroes out the x-row of R but, at the same time, all rows of R identical to it. In graph-theoretic terminology this "parallel zeroing out" removes all edges from $G = (X,R)$ that start from a vertex $y \in X$ such that $succ_R(y) = succ_R(x)$.

4 Implementations in Haskell

The aim of this Section is to develop by means of data refinement an implementation of (4) in Haskell, that uses linear lists of successor lists for representing relations. To simplify the development, we assume that the finite carrier set X of the input relation $R : X \leftrightarrow X$ consists of the natural numbers $0, 1, \ldots, m$. We assume the reader to be familiar with Haskell.

4.1 From Relation Algebra to Lists of Successor Sets

To obtain a version of (4) that works on linear lists of successor sets, we represent in a first step relations of type $[X \leftrightarrow X]$ by functions from X to its powerset 2^X which map each element to its successor set. Doing so, the empty relation O is represented by $\lambda x.\emptyset$ and the identity relation I by $\lambda x.\{x\}$, where we denote anonymous functions as λ-terms.

Now, let the two relations R and C of (4) be represented by the functions $r : X \to 2^X$ and $c : X \to 2^X$, respectively. Then the selection of a point p such that $p \subseteq R;\mathsf{L}$ corresponds to the selection of an element n from X such that $r(n) \neq \emptyset$. This element may be assumed to be represented by p. Based on r, c and n, the function representing $R \cap \overline{p;p^\mathsf{T};R}$ maps all $x \neq n$ to $r(x)$ and n to $\emptyset$. Usually this function is denoted as $r[n \leftarrow \emptyset]$. For the relation $C \cup C;p;p^\mathsf{T};R;C$ we get $\lambda x.\mathtt{if}\ n \in c(x)\ \mathtt{then}\ c(x) \cup ks\ \mathtt{else}\ c(x)$ as representing function, where ks abbreviates the set $\bigcup\{c(k) \mid k \in r(n)\}$. This follows from the subsequent calculation in which x and y are arbitrary elements from the set X and in the existentially quantified formulae i, j, k range over X.

$$
\begin{array}{ll}
 & (x,y) \in C \cup C;p;p^\mathsf{T};R;C \\
\iff & (x,y) \in C \vee (x,y) \in C;p;p^\mathsf{T};R;C \\
\iff & (x,y) \in C \vee \exists\, i : (x,i) \in C \wedge \exists\, j : (i,j) \in p;p^\mathsf{T} \wedge (j,y) \in R;C \\
\iff & (x,y) \in C \vee \exists\, i : (x,i) \in C \wedge \exists\, j : i = n \wedge j = n \wedge (j,y) \in R;C \\
\iff & (x,y) \in C \vee ((x,n) \in C \wedge (n,y) \in R;C) \\
\iff & (x,y) \in C \vee ((x,n) \in C \wedge \exists\, k : (n,k) \in R \wedge (k,y) \in C) \\
\iff & y \in c(x)\ \ \vee \mathtt{if}\ n \in c(x)\ \mathtt{then}\ \exists\, k : k \in r(n) \wedge y \in c(k)\ \mathtt{else}\ \mathit{false} \\
\iff & y \in c(x) \vee \mathtt{if}\ n \in c(x)\ \mathtt{then}\ y \in \bigcup\{c(k) \mid k \in r(n)\}\ \mathtt{else}\ y \in \emptyset \\
\iff & y \in c(x) \vee y \in \mathtt{if}\ n \in c(x)\ \mathtt{then}\ ks\ \mathtt{else}\ \emptyset \\
\iff & y \in \mathtt{if}\ n \in c(x)\ \mathtt{then}\ c(x) \cup ks\ \mathtt{else}\ c(x)
\end{array}
$$

This calculation only uses well-known correspondences between logical, relation-algebraic and set-theoretic constructions and the definition of ks.

If we replace in (4) all relations by their representing functions and the selection of the point p by that of the element n from $\{x \in X \mid r(x) \neq \emptyset\}$, we arrive at the following functional program.

$$
\begin{array}{l}
rtc(r) = \mathtt{if}\ r = \lambda x.\emptyset\ \mathtt{then}\ \lambda x.\{x\} \\
\qquad\qquad \mathtt{else\ let}\ n = \mathit{elem}(\{x \in X \mid r(x) \neq \emptyset\}) \\
\qquad\qquad\qquad\qquad c = rtc(r[n \leftarrow \emptyset]) \\
\qquad\qquad\qquad\qquad ks = \bigcup\{c(k) \mid k \in r(n)\} \\
\qquad\qquad\qquad\ \mathtt{in}\ \ \lambda x.\mathtt{if}\ n \in c(x)\ \mathtt{then}\ c(x) \cup ks\ \mathtt{else}\ c(x)
\end{array}
$$

In the next steps, we represent functions by linear lists. We start with the functions which appear as results of rtc, i.e., $c : X \to 2^X$ and the anonymous ones $\lambda x.\{x\}$ and $\lambda x.\mathtt{if}\ n \in c(x)\ \mathtt{then}\ c(x) \cup ks\ \mathtt{else}\ c(x)$. By assumption, the set X consists of the natural numbers $0, 1, \ldots, m$. Hence, it seems to be obvious to represent a function from X to 2^X by a linear list of length $m+1$ such that for all x with $0 \leq x \leq m$ the x-th component of the list equals the result of the

application of the function to x, i.e., is a subset of X. In the case of $\lambda x.\{x\}$ this yields the linear list $[\{0\},\ldots,\{m\}]$ of singleton sets. Using a notation similar to Haskell's list comprehension, we denote it by $[\{x\} \mid x \in [0..m]]$. For the function $\lambda x.\texttt{if } n \in c(x) \texttt{ then } c(x) \cup ks \texttt{ else } c(x)$ we obtain, again using the list comprehension notation just introduced, the representation

$$[\texttt{if } n \in cs!!x \texttt{ then } cs!!x \cup ks \texttt{ else } cs!!x \mid x \in [0..m]], \tag{6}$$

where cs is the list representation of c and, as in Haskell, $cs!!x$ denotes the x-th component of the linear list cs. The list cs consists exactly of the lists $cs!!x$, where x ranges over $0, 1, \ldots, m$. Hence, instead of ranging in the list comprehension (6) over all these indices, we alternatively can range over all components ms of the linear list cs, that is, can replace in (6) the formula $x \in [0..m]$ by $ms \in cs$, if simultaneously the expression $cs!!x$ is replaced by ms.

Moving from the output functions to the list representations we have obtained transforms the above program into the following one.

$$\begin{aligned} rtc(r) = \texttt{if } r = \lambda x.\emptyset \texttt{ then } & [\{x\} \mid x \in [0..m]] \\ \texttt{else let } & n = elem(\{x \in X \mid r(x) \neq \emptyset\}) \\ & cs = rtc(r[n \leftarrow \emptyset]) \\ & ks = \bigcup\{cs!!k \mid k \in r(n)\} \\ \texttt{in } & [\texttt{if } n \in ms \texttt{ then } ms \cup ks \texttt{ else } ms \mid ms \in cs] \end{aligned}$$

It remains to give list representations of the input functions r and $r[n \leftarrow \emptyset]$ of this version of rtc. Here the approach used for the result functions of rtc seems not to be reasonable. In the case of $\lambda x.\emptyset$ such a representation leads to the task of testing whether all components of a list of sets are $\emptyset$. Also the selection of an n such that $0 \leq n \leq m$ and the n-th component of the list representation of r is not $\emptyset$ requires some effort. The same holds for the computation of the list representation of $r[n \leftarrow \emptyset]$ from that of r.

If, however, the input function r of rtc is represented as a linear list rs of pairs (x, xs), where x ranges over all numbers from 0 to m for which $r(x) \neq \emptyset$ and xs is a set that equals $r(x)$, and the function $r[n \leftarrow \emptyset]$ is represented in the same way, then the solutions of the just mentioned three tasks are rather simple. Testing $r = \lambda x.\emptyset$ reduces to the list-emptiness test $rs = []$. An n with $0 \leq n \leq m$ and $r(n) \neq \emptyset$ is given by the first component of the head of the list rs. Because of the relationship between the two components of each list element, with this choice of n the list representation of the function $r[n \leftarrow \emptyset]$ is obtained from that of r by simply removing the head of rs and, furthermore, the set $r(n)$ in $\bigcup\{cs!!k \mid k \in r(n)\}$ equals the second component of the head of rs. If we use pattern matching in the `let`-clause and the well-known list-operations *head* and *tail*, we obtain the following program.

$$\begin{aligned} rtc(rs) = \texttt{if } rs = [] \texttt{ then } & [\{x\} \mid x \in [0..m]] \\ \texttt{else let } & (n, ns) = head(rs) \\ & cs = rtc(tail(rs)) \\ & ks = \bigcup\{cs!!k \mid k \in ns\} \\ \texttt{in } & [\texttt{if } n \in ms \texttt{ then } ms \cup ks \texttt{ else } ms \mid ms \in cs] \end{aligned}$$

This program can also be written in the following form. It uses a non-recursive auxiliary function *step* that performs, for the head of *rs* and the result of the recursive call of *rtc* as inputs, the essential computations of the recursion.

$$\begin{aligned} step((n, ns), cs) &= \texttt{let } ks = \bigcup\{cs!!k \mid k \in ns\} \\ &\quad\ \texttt{in } [\texttt{if } n \in ms \texttt{ then } ms \cup ks \texttt{ else } ms \mid ms \in cs] \\ rtc(rs) &= \texttt{if } rs = [] \texttt{ then } [\{x\} \mid x \in [0..m]] \\ &\quad\ \texttt{else } step(head(rs), rtc(tail(rs))) \end{aligned} \tag{7}$$

Obviously, an execution of this program leads to at most $m + 1 = |X|$ recursive calls of *rtc*. Since the generation of the list $[\{x\} \mid x \in [0..m]]$ can be done in time $O(m)$ and *head* and *tail* require constant time, the total running time of program (7) depends on the time complexity of the auxiliary function *step*.

4.2 A Haskell Program for Reflexive-transitive Closures

For the following, we assume the largest number m of the set X to be at hand as Haskell constant `m` of type `Int`. To obtain an implementation of (7) in Haskell, the main task is to implement subsets of X in Haskell and to formulate a Haskell function that computes the union of two sets. From the latter, a simple recursion (that we will realize by a pre-defined higher-order Haskell function) then at once leads to the computation of the set $\bigcup\{cs!!k \mid k \in ns\}$.

An obvious implementation of subsets of X in Haskell is given by linear lists over X *without multiple occurrences of elements*. Then $\emptyset$ is implemented by `[]` and set-membership by the pre-defined Haskell operation `elem`. A straightforward implementation of set union requires quadratic time. But we can do better using that the set $X = \{0, 1, \ldots, m\}$ is linearly ordered. If we additionally demand that *all lists that occur as representations of successor sets are increasingly sorted*, then an obvious implementation of set union is given by the following Haskell function. It needs linear time to merge two sorted lists into a sorted one and to remove at the same time all multiple occurrences of elements.

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x == y = x : merge xs ys
                    | x <  y = x : merge xs (y:ys)
                    | x >  y = y : merge (x:xs) ys
```

Now, assume the set *ns* of the expression $\bigcup\{cs!!k \mid k \in ns\}$ of (7) to be given as Haskell list `ns` and let `cs` and `k` be the Haskell counterparts of *cs* and *k*. Then the Haskell list for $\bigcup\{cs!!k \mid k \in ns\}$ is obtained by merging all sorted lists `cs!!k`, where `k` ranges over the elements of `ns`. Such a repeated application of `merge` corresponds to a fold over a list containing all lists `cs!!k`, with the empty list as initial value. If we use right-fold, in Haskell realized by the higher-order

function `foldr`, we get `foldr merge [] [cs!!k | k <- ns]` as list representation of $\bigcup\{cs!!k \mid k \in ns\}$ and, thus, the following Haskell implementation of *step*.

```
step :: (Int,[Int]) -> [[Int]]  -> [[Int]]
step (n,ns) cs =
  let ks = foldr merge [] [cs!!k | k <- ns]
  in  [if n `elem` ms then merge ms ks else ms | ms <- cs]
```

In view of the main function *rtc* of (7), our Haskell implementation of successor sets leads to `[[x] | x <- [0..m]]` as Haskell code for $[\{x\} \mid x \in [0..m]]$. Also *rtc* can be seen as a right-fold over the Haskell counterpart `rs` of *rs* that realizes the repeated application of `step`. The initial value is `[[x] | x <- [0..m]]`. So, we arrive at the following Haskell implementation.

```
rtc :: [(Int,[Int])] -> [[Int]]
rtc rs = foldr step [[x] | x <- [0..m]] rs
```

For estimating the running time of `step`, assume that the length of the input lists `ns`, `cs` and of all components of `cs` is at most $m + 1$ and that `ns` and all components of `cs` are sorted. Then the computation of `ks` can be done in time $O(m^2)$ since this time suffices to generate `[cs!!k | k <- ns]` as well as to merge its components to a single list. Notice that during the merging process each argument of `merge` has at most length $m + 1$, the same also holds for the result and `merge` merges sorted lists in linear time. Also the list comprehension of `step` can be evaluated in $O(m^2)$ steps. This follows again from the assumptions on the input since they imply that each of the at most $m + 1$ membership-tests and mergings can be done in time $O(m)$. (The length of `ks` is at most $m + 1$).

If the Haskell representation `rs` of the original input relation R satisfies the requirements stated at the beginning of this section, i.e., each successor list is sorted and without multiple occurrences of elements, then all arguments of the calls of `step` that occur during the evaluation of `rtc rs` satisfy the assumptions of the running time estimation of `step`. Together with the remark made at the end of Subsection 4.1, we get the claimed running time $O(m^3)$.

Slightly dissatisfying is that the above Haskell function `rtc` depends on the constant `m` and works with two different implementations of relations. But it is not difficult to change it in such a way that it is independent of `m` and also the input `rs` is a linear list of type `[[Int]]` that contains as x-th component the (possibly empty) successor list of x. Here is the result, where `step` is as above.

```
rtc :: [[Int]] -> [[Int]]
rtc rs =
  let xs = [0..length rs - 1]
  in  foldr step [[n] | n <- xs] (zip xs rs)
```

Since the length of `rs` is $m + 1$, the linear list `xs` equals `[0..m]`. Using the predefined operation `zip` that takes two lists and returns the list of corresponding pairs, the list `rs` of successor lists is transformed into a list of pairs (x, xs),

where x ranges over $X = \{0, 1, \ldots, m\}$ and xs is the successor list of x. This is almost the same representation as used in Subsection 4.1 for the output of *rtc*; compared with Subsection 4.1 the only difference is that now a pair of the list *rs* may contain an empty second component. But this is without any effect w.r.t. correctness since a call `step (n,[]) cs` yields `cs` as its result.

5 Related Work and Concluding Remarks

Computing the reflexive-transitive closure of a relation can also be seen as a problem of graph-theory. Therefore, related to our work are all the approaches to program graph algorithms in a functional language. In the meantime functional graph algorithms have a long tradition. Here we only want to mention some papers that deal with different aspects. In [2] transformational programming is applied to derive certain functional reachability algorithms. The paper [5] deals with the specification and functional computation of the depth-first search forest and present some classical applications in the functional style. To achieve a linear running time, monads are used to mimic the imperative marking technique. Instead of regarding graphs as monolithic data as all the just mentioned papers do, in [4] graphs are inductively generated. This allows to write many graph algorithms in the typical functional style using structural recursion.

Each relation can be specified as union of disjoint rectangles. This is closely related to the inductive generation of graphs in [4] since there in each step of the generation process a graph g is extended by a new vertex x and edges connecting x with vertices that are already in g. The insertion of the edges corresponds to a union with two rectangles, one consisting of the edges with source x and one consisting of the edges with sink x. It would be interesting to explore how the inductive graph view can be transferred to relations to make algorithms on relations and relational structures (like graphs, orders, Petri nets, games) more amenable for the typical functional style.

References

1. Aarts A. et al.: Fixed point calculus. Inf. Proc. Letters 53, 131-136 (1996).
2. Berghammer R., Ehler H., Zierer H.: Development of graph algorithms by program transformation. In: Graph-Theoretic Concepts in Computer Science. LNCS 314, Springer, 206-218 (1988).
3. Berghammer R., Neumann F.: RelView — An OBDD-based Computer Algebra system for relations. In: Computer Algebra in Scientific Computing, LNCS 3718, Springer, 40-51 (2005).
4. Erwig M.: Inductive graphs and functional graph algorithms. J. Functional Programming 11, 467-492 (2001).
5. King D.J., Launchbury J.: Structuring depth-first search algorithms in Haskell. In: ACM Symp. on Principles of Programming. ACM Press, 344-356 (1995).
6. Schmidt G., Ströhlein T.: Relations and graphs. Springer (1993).
7. Schmidt G.: Rectangles, frings and inverses. In: Relations and Kleene Algebra in Computer Science. LNCS 4988. Springer, 352-366 (2008).
8. Warshall S.: A theorem on Boolean matrices. J. ACM 9, 11-12 (1962).

The First-Order Nominal Link

Christophe Calvès and Maribel Fernández

King's College London, Department of Computer Science,
Strand, London WC2R 2LS, UK
Christophe.Calves@kcl.ac.uk Maribel.Fernandez@kcl.ac.uk

Abstract. We define a morphism from nominal syntax, which supports binding, to standard (first-order) syntax. We use this morphism to extend Paterson and Wegman's linear first-order unification algorithm in order to deal with terms modulo alpha-equivalence. The nominal unification algorithm obtained is quadratic in time in general, and is linear for large classes of nominal unification problems.

1 Introduction

This paper studies the link between languages based on first-order syntax, where terms are built with function symbols and variables, and languages based on nominal syntax [6,9,10], which extends first-order syntax to provide support for binding operators. Operators with binding are ubiquitous in computer science. Programs, logic formulas, and process calculi are some examples of systems that involve binding. Program transformations and optimisations, for instance, are defined as operations on programs, and therefore work uniformly on α-equivalence classes.

In nominal syntax, *atoms* (or object-level variables) can be abstracted (we use the notation $[a]t$, where a is an atom and t is a term), and *meta-variables* (or just variables) behave like first-order variables and are used to represent unknown parts of terms. Variables may be decorated with atom permutations, generated by swappings (e.g., $(a\ b) \cdot t$ means swap a and b everywhere in t). For instance, $(a\ b)\cdot[a]a = [b]b$, and $(a\ b)\cdot[a]X = [b](a\ b)\cdot X$ (we will introduce the notation formally in Section 2). As shown in this example, permutations suspend on variables. The idea is that when a substitution is applied to X in $(a\ b)\cdot X$, the permutation will be applied to the term that instantiates X.

As atoms are just names, we want to identify $[a]a$ and $[b]b$, and even $[a]X$ and $[b]X$ under some conditions. Permutations of atoms are one of the main ingredients in the definition of α-equivalence for nominal terms, they are the translation tables from one term to another. Obviously such translations are subject to conditions, $[a]X$ and $[b]X$ can only be identified if neither a nor b are unabstracted in X. Although many first-order algorithms have been successfully extended to the nominal case (see for instance [11,2] for nominal versions of unification and matching, [5,3] for nominal equational reasoning, and [4] for definitions and applications of nominal rewriting), the management of such translations and conditions is subtle. The main contributions of this article are:

– A study of the relationship between nominal theories and first-order theories. We present a morphism between nominal and first-order theories, which enables us to extend concepts (such as equivalence relations for example), algorithms and properties from the first-order setting to the nominal setting.
– An application of the previous result to solve unification problems for nominal terms. Urban et al's original presentation of a nominal unification algorithm [11] is an extension of a first-order unification algorithm based on multiset rewriting, which can be obtained directly using the morphism. A naive implementation of this algorithm is exponential. In this paper we derive a quadratic nominal unification algorithm from Paterson and Wegman's linear unification algorithm [8] and show its correctness using the morphism.[1]

The article is structured as follows: Section 2 introduces nominal theory. Section 3 presents a morphism from nominal to first-order theory. We use this morphism in Section 4 to extend Patterson and Wegman's linear first-order unification algorithm to a quadratic nominal algorithm. Section 5 concludes the paper.

2 Background

Let Σ be a denumerable set of **function symbols** f, g, ...; $\mathcal{X}$ a denumerable set of **variables** $X, Y, \ldots$; and $\mathcal{A}$ a denumerable set of **atoms** $a, b, \ldots$ We assume that Σ, $\mathcal{X}$, and $\mathcal{A}$ are pairwise disjoint. In the intended applications, variables will be used to denote meta-level variables (unknowns), and atoms will be used to represent object level variables.

A **swapping** is a pair of (not necessarily distinct) atoms, written $(a\ b)$. **Permutations** π are lists of swappings, generated by the grammar: $\pi ::= \mathsf{Id} \mid (a\ b)\circ\pi$ where Id is the **identity permutation**. We write π^{-1} for the permutation obtained by reversing the list of swappings in π. We denote by $\pi\circ\pi'$ the permutation containing all the swappings in π followed by those in π'. The set of permutations is written Π.

We now define a minimal syntax for nominal terms using pairs instead of arbitrary tuples as in [11,4].The application of a function symbol to its arguments is also represented as a pair of the function symbol and the arguments, and permutations can be attached to any term.

Definition 1. *The set of* ***nominal terms****, denoted $\mathcal{T}_N$, is generated by the grammar*

$$s, t ::= a \mid X \mid f \mid (s_1, s_2) \mid [a]s \mid \pi{\cdot}s$$

where $[a]s$ is an abstraction, (s_1, s_2) a pair, $\pi{\cdot}s$ a suspension.

[1] Another quadratic nominal unification algorithm was independently developed by Levy and Villaret [7].

Substitutions are generated by the grammar: $\sigma ::= \mathtt{Id} \mid [X \mapsto t]\sigma$. The composition of substitutions is written $\sigma \circ \sigma'$, and is defined by:

$$\begin{array}{lcl} \mathtt{Id} \circ \sigma' & = & \sigma' \\ ([X \mapsto t]\sigma) \circ \sigma' & = & [X \mapsto t](\sigma \circ \sigma') \end{array}$$

Substitutions will act on terms by instantiating their variables. Permutations will act top-down on nominal terms and accumulate on variables. By changing the term constructors and the actions of substitutions and permutations we can obtain different kinds of languages. We explore this issue in the next section, but first we define the basic conditions that substitutions and permutations should satisfy.

Definition 2. *Let $\mathcal{T}$ be a set of terms, $\mathcal{S}_\mathcal{T}$ a set of substitutions over $\mathcal{T}$ and Π the set of permutations. A* ***substitution application function****, denoted $\llcorner$, is a function from $\mathcal{S}_\mathcal{T} \times \mathcal{T}$ to $\mathcal{T}$ such that*

$$\begin{array}{lcl} \forall t \in \mathcal{T} \quad t \llcorner \mathtt{Id} & = & t \\ \forall t \in \mathcal{T}, \sigma, \sigma' \in \mathcal{S}_\mathcal{T} \quad t \llcorner (\sigma \circ \sigma') & = & (t \llcorner \sigma) \llcorner \sigma' \end{array}$$

where $t \llcorner \sigma$ denotes the application of σ on t. When there is no ambiguity, $t \llcorner \sigma$ will be written $t\sigma$.

A permutation action $\frown$ is a function from $\pi \times \mathcal{T}$ to $\mathcal{T}$ that satisfies:

$$\begin{array}{lcl} \forall t \in \mathcal{T} \quad \mathsf{Id} \frown t & = & t \\ \forall t \in \mathcal{T}, \pi, \pi' \in \Pi \quad (\pi \circ \pi') \frown t & = & (\pi \circ \pi') \cdot t \end{array}$$

Nominal constraints have the form $a\#t$ and $s \approx_\alpha t$ (read "a fresh for t" and "s α-equivalent to t", respectively). A freshness context Δ is a set of freshness constraints of the form $a\#X$. We define the validity of constraints under a freshness context Δ inductively, by a system of axioms and rules, using $\#$ in the definition of $\approx_\alpha$ (see below). We write $ds(\pi, \pi')\#X$ as an abbreviation for $\{a\#X \mid a \in ds(\pi, \pi')\}$, where $ds(\pi, \pi') = \{a \mid \pi{\cdot}a \neq \pi'{\cdot}a\}$ is the set of atoms where π and π' differ (i.e., their difference set). Below a, b are any pair of distinct atoms.

$$\frac{}{\Delta \vdash a\#b}\,(\#_{ab}) \qquad \frac{}{\Delta \vdash a\#f}\,(\#_f) \qquad \frac{\Delta \vdash a\#s_1 \quad \Delta \vdash a\#s_2}{\Delta \vdash a\#(s_1, s_2)}\,(\#_{tup})$$

$$\frac{}{\Delta \vdash a\#[a]s}\,(\#_{absa}) \qquad \frac{\Delta \vdash a\#s}{\Delta \vdash a\#[b]s}\,(\#_{absb}) \qquad \frac{\Delta \vdash \pi^{-1}{\cdot}a\#s}{\Delta \vdash a\#\pi{\cdot}s}\,(\#_X)$$

$$\frac{a \# X \in \Delta}{\Delta \vdash a\#X}\,(\#_\Delta)$$

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(\approx_{\alpha a}) \qquad \frac{\Delta \vdash ds(\pi, \pi')\#s}{\Delta \vdash \pi{\cdot}s \approx_\alpha \pi'{\cdot}s}\,(\approx_{\alpha s})$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1 \quad \Delta \vdash s_2 \approx_\alpha t_2}{\Delta \vdash (s_1, s_2) \approx_\alpha (t_1, t_2)} (\approx_{\alpha tup}) \qquad \frac{}{\Delta \vdash f \approx_\alpha f} (\approx_{\alpha f})$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash [a]s \approx_\alpha [a]t} (\approx_{\alpha absa}) \qquad \frac{\Delta \vdash s \approx_\alpha (a\ b)\cdot t \quad a\#t}{\Delta \vdash [a]s \approx_\alpha [b]t} (\approx_{\alpha absb})$$

We remark that $\approx_\alpha$ is indeed an equivalence relation (see [11] for more details). We write $\Delta \vdash a \# t$ and $\Delta \vdash t \approx_\alpha s$ when the judgement is derivable using the system above. When Δ is empty we write simply $a \# t$ or $t \approx_\alpha s$. For example, $a\#X, b\#X \vdash [a]X \approx_\alpha [b]X$, and $[a]a \approx_\alpha [b]b$ (as indicated in the Introduction).

3 Nominal Structure: From Nominal to First-Order

Definition 3. *A **nominal structure** is a triple $(\mathcal{T}, \frown, \llcorner)$, where $\mathcal{T}$ is a set of terms, $\frown$ is a permutation action (i.e., a function from $\Pi \times \mathcal{T}$ to $\mathcal{T}$ representing the application of a permutation on a term, written $\pi \frown t$), and $\llcorner$ is a substitution application function over $\mathcal{T}$ (see Definition 2).*

If we take the set $\mathcal{T}_N$ of nominal terms and define a permutation action such that

$$\pi \frown X = \pi \cdot X \quad \pi \frown a = \pi(a) \quad \pi \frown f = f$$
$$\pi \frown (s_1, s_2) = (\pi \frown s_1, \pi \frown s_2) \quad \pi \frown [a]s = [\pi(a)]\pi \frown s$$

and a substitution action such that $(\pi \cdot X)\sigma = \pi \frown (X\sigma)$ (extended as usual to terms), we get a nominal structure with the standard notion of permutation and substitution on nominal terms.

Nominal terms not containing atoms ($\mathrm{A}(t) = \emptyset$) are actually first-order terms.

Definition 4. *The set $\mathcal{T}_f$ of **first-order terms** is generated by the grammar:*

$$s, t ::= \bullet \mid X \mid f \mid (s_1, s_2) \mid [] \ s$$

On nominal terms, permutations accumulate on variables, but in first-order syntax, variables represent first-order terms so $\pi \frown X = X$. Thus we define the permutation action on first-order terms as

$$\forall t \in \mathcal{T}_f, \pi \in \Pi \quad \pi \frown_f t = t$$

$\mathcal{T}_f$ with $\frown_f$ as permutation action and the standard substitution application function forms a nominal structure called the **first-order structure**.

Definition 5. *A **morphism** from the nominal structure $(\mathcal{T}_1, \frown_1, \llcorner_1)$ to $(\mathcal{T}_2, \frown_2, \llcorner_2)$ is a function φ from $\mathcal{T}_1$ to $\mathcal{T}_2$ such that*

$$\begin{array}{ll} \forall t \in \mathcal{T}_1, \pi \in \Pi & \varphi(\pi \frown_1 t) = \pi \frown_2 \varphi(t) \\ \forall t \in \mathcal{T}_1, \sigma \in \mathcal{S}_{\mathcal{T}_1} & \varphi(t \llcorner_1 \sigma) \ = \varphi(t) \llcorner_2 \varphi(\sigma) \end{array}$$

where $\varphi([X_1 \mapsto t_1] \ldots [X_n \mapsto t_n]) = [X_1 \mapsto \varphi(t_1)] \ldots [X_n \mapsto \varphi(t_n)]$

We present here a morphism F from nominal terms to first-order terms. Let t be a nominal term over Σ, $\mathcal{X}$ and $\mathcal{A}$. F associates to t a first-order term by "forgetting" the nominal details of t: $\bullet$ will represent atoms and [] abstraction. F is defined as follows:

$$\begin{array}{lcl} F(a) & = & \bullet \\ F(X) & = & X \\ F(f) & = & f \\ F(\pi{\cdot}t) & = & F(t) \\ F((t_1, t_2)) & = & (F(t_1), F(t_2)) \\ F([a]t) & = & []\, F(t) \end{array}$$

For example: $F(([a](a, ((a\ b){\cdot}X, Y)), f)) = ([](\bullet, (X, Y)), f)$. F can be extended to substitutions by: $F([X \mapsto t]) = [X \mapsto F(t)]$.

Proposition 1. *Let t be a nominal term and σ a substitution, then $F(t\sigma) = F(t)F(\sigma)$.*

Definition 6. *A **nominal unification problem**, or **unification problem** for short, is a set Pr of nominal constraints, for which we have to decide whether there exists a freshness context Δ and a substitution σ such that for every constraint $a \# t$ and $s \approx_\alpha u$ in Pr, $\Delta \vdash a \# t\sigma$ and $\Delta \vdash s\sigma \approx_\alpha u\sigma$. (Δ, σ) is called a solution of the nominal unification problem.*

The morphism F defined above can also be extended to unification problems: $F(s \approx_\alpha t) = F(s) \stackrel{?}{=} F(t)$, and $F(a \# t) = \top$.

Proposition 2. *Let s, t be two nominal terms.*

1. *$\Delta \vdash s \approx_\alpha t \Rightarrow F(s) = F(t)$ (i.e., $F(s)$ and $F(t)$ are syntactically equal).*
2. *If a nominal unification problem Pr has a solution (Δ, σ), then the first-order unification problem $F(Pr)$ has a solution, and $F(\sigma)$ is a solution.*

This result is interesting because it says that if the first-order problem corresponding to a nominal problem does not have any solution, then neither has the nominal problem. Because first-order unification can be solved in linear time and space, it provides a fast way to decide if some problems are solvable. It also says that solutions of nominal problems and first-order problems have the same structure, i.e., the same function symbols, abstractions, variables, tuples at the same positions.

4 Extending Paterson and Wegman's First-Order Unification Algorithm

As an application of the previous observations, in this section we derive a nominal unification algorithm from Paterson and Wegman's First-Order Linear Unification Algorithm *(FLU)* [8]. Because any unification problem can be encoded in

linear time and space into a single constraint $s \approx_\alpha t$ (see [2] for details), we only consider unification problems of the form $\{s \approx_\alpha t\}$. The set of atoms in the problem, written A_0 is defined as $A_0 = \mathrm{A}(s) \cup \mathrm{A}(t)$.

In the rest of this section, we consider a nominal graph d representing the problem $s \approx_\alpha t$. **Nominal graphs** are directed acyclic graphs, where atoms, variables and function symbols are represented by nodes labelled by the corresponding symbol. In addition, we represent pairs using a node labelled by () with two edges to the representation of each element in the pair. Similarly we introduce a node labelled by [] to represent an abstraction, and a node labelled by $\cdot$ to represent a suspension. Nodes that are labelled by the same atom or variable symbol are identified (i.e., there is at most one leaf labelled by a given atom a or variable X). We refer the reader to [2] for examples of nominal graphs, omitted here due to lack of space.

We start by giving, in Section 4.1, a theoretical characterisation of nominal unification as a relation on the nodes of d. We then use this relation to prove the correctness of the algorithm presented in Section 4.2. A concrete implementation of the algorithm in *Objective CAML* [1] is available at `http://www.dcs.kcl.ac.uk/pg/calves/qnu.ml`.

4.1 Nominal Unification as a Relation on Term Nodes

The *FLU* algorithm computes the most general unifier of two first-order terms, if one exists, by computing a binary relation on the nodes of the directed acyclic graph representing the problem. This relation basically means that related terms can be made equal (by an appropriate substitution). In nominal theory, we are interested in whether or not two terms can be made α-equivalent and not just syntactically equal. So we adapt this relation to take into account permutations and freshness constraints.

To prove the correctness of their algorithm Paterson and Wegman developed in [8] the concept of **valid** equivalence relation. We extend this notion to nominal graphs below. First we define nominal relations.

Definition 7. *A **nominal relation** R is a binary relation on nodes of a nominal graph, defined in terms of a ternary relation $\mathbb{R}_{?\approx_?,}$ between nodes and permutations, written $s\ \mathbb{R}_{?\approx_?,\pi}\ t$, and a binary relation $\mathbb{F}$ between atoms and nodes, written $a\ \mathbb{F}\ s$. Two nodes s and t are related by R, written $s\ R\ t$, if and only if there exists a permutation π such that $s\ \mathbb{R}_{?\approx_?,\pi}\ t$.*

*A nominal relation $(\mathbb{R}_{?\approx_?,}, \mathbb{F})$ is a **nominal equivalence relation** if:*

- $s\ \mathbb{R}_{?\approx_?,\mathsf{Id}}\ s$ *for any* s *(Reflexivity)*
- $s\ \mathbb{R}_{?\approx_?,\pi}\ t \Rightarrow t\ \mathbb{R}_{?\approx_?,\pi^{-1}}\ s$ *(Symmetry)*
- $s\ \mathbb{R}_{?\approx_?,\pi_1}\ u \qquad u\ \mathbb{R}_{?\approx_?,\pi_2}\ t \Rightarrow s\ \mathbb{R}_{?\approx_?,\pi_1\circ\pi_2}\ t$ *(Transitivity)*
- $s\ \mathbb{R}_{?\approx_?,\pi}\ s \Rightarrow (a\ \mathbb{F}\ s)_{a\in supp(\pi)}$, *where* $supp(\pi)$ *is the set of atoms affected by the permutation (Disagreement set)*
- $a\ \mathbb{F}\ s$ *and* $s\ \mathbb{R}_{?\approx_?,\pi}\ t \Rightarrow \pi^{-1}(a)\ \mathbb{F}\ t$ *(Congruence)*

Definition 8. *Let* $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ *be a nominal relation. We define the function* fc *over nodes as* $fc(s) = \{a \mid a\ \mathbb{F}\ s\}$.

Intuitively, $fc(s)$ represents the set of atoms that must be fresh in the term represented by s (i.e. the set of *freshness constraints* on s).

Proposition 3. *If* $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ *is a nominal equivalence relation then*

$$s\ \mathbb{R}_{?\approx?,\pi}\ t \Rightarrow fc(s) = \pi(fc(t))$$

Proof. Assume $a \in fc(s)$, then $a\ \mathbb{F}\ s$. If $s\ \mathbb{R}_{?\approx?,\pi}\ t$ then $\pi^{-1}(a)\ \mathbb{F}\ t$ (by Congruence). Hence $\pi^{-1}(a) \in fc(t)$ and $a \in \pi(fc(t))$. By Symmetry, $s\ \mathbb{R}_{?\approx?,\pi}\ t \Rightarrow t\ \mathbb{R}_{?\approx?,\pi^{-1}}\ s$, which gives $fc(t) \subseteq \pi^{-1}(fc(s))$.

If two first-order terms do not have, on their root node, the same function symbol and the same arity, then they can not be unified. Paterson and Wegman's algorithm uses this property to detect unsolvable problems. We extend this notion of compatibility to nominal graphs:

Definition 9 ($\pi, \mathbb{F}$-compatibility). *Two nodes* s *and* t *are said* $\pi, \mathbb{F}$***-compatible*** *if and only if at least one of the following properties holds:*

- $s = \pi' \cdot s'$ *and* s' *and* t *are* $(\pi'^{-1} \circ \pi), \mathbb{F}$*-compatible*
- $t = \pi' \cdot t'$ *and* s *and* t' *are* $(\pi \circ \pi'), \mathbb{F}$*-compatible*
- s *and* t *are two atoms,* $s = \pi \cdot t$, $s \notin fc(s)$ *and* $t \notin fc(t)$
- s *and* t *are two constants and* $s = t$
- s *and* t *are two pairs*
- s *and* t *are two abstractions*
- s *is a variable*
- t *is a variable*

When there is no ambiguity we will omit $\mathbb{F}$ *and write simply* π*-compatibility.*

Definition 10. *Two nodes* s *and* t*, representing first-order terms, are said* ***compatible*** *if and only if they are* $\mathsf{Id}, \emptyset$*-compatible.*

Any directed acyclic graph can generate a partial ordering on the nodes of the graph as follows:

Definition 11. *The partial order derived from the directed acyclic graph* d *is defined as:* $n_1 >_d n_2$ *if and only if there exists a path from the node* n_1 *to the node* n_2 *in* d *(*$n_1 \neq n_2$*).*

Paterson and Wegman define a *valid* equivalence relation as:

> An equivalence relation on the nodes of a dag is valid if it has the following properties:
> (i) if two function nodes are equivalent then their corresponding children are pairwise equivalent,

(ii) each equivalence class is homogeneous, that is, it does not contain two nodes with distinct function symbols,
(iii) the equivalence classes may be partially ordered by the partial order on the dag.

We extend the notion of valid equivalence relation to nominal terms, following Paterson and Wegman's definition closely:

Definition 12. *A nominal equivalence relation* $(\mathbb{R}_{?\approx_?,}, \mathbb{F})$ *on the nodes of a nominal dag is* ***valid*** *if:*

– *Propagation:*

$$\begin{array}{ll} (s_1, s_2)\ \mathbb{R}_{?\approx_?,\pi}\ (t_1, t_2) & \Rightarrow s_1\ \mathbb{R}_{?\approx_?,\pi}\ t_1 \quad s_2\ \mathbb{R}_{?\approx_?,\pi}\ t_2 \\ [a]s\ \mathbb{R}_{?\approx_?,\pi}\ [b]t & \Rightarrow s\ \mathbb{R}_{?\approx_?,(a\ \pi(b))\circ\pi}\ t \\ & \quad (\pi(b)\ \mathbb{F}\ s \qquad \text{if } a \neq \pi(b)) \\ a\ \mathbb{F}\ (s_1, s_2) & \Rightarrow a\ \mathbb{F}\ s_1 \quad a\ \mathbb{F}\ s_2 \\ a\ \mathbb{F}\ [b]s & \Rightarrow a\ \mathbb{F}\ s \text{ (if } a \neq b) \\ a\ \mathbb{F}\ [a]s & \end{array}$$

– *Compatibility: each nominal class is* homogeneous*, that is for any* s *and* t *such that* $s\ \mathbb{R}_{?\approx_?,\pi}\ t$*,* (s,t) *is* $\pi, \mathbb{F}$*-compatible.*
– *Occurrence check: the partial order derived from the directed acyclic graph, quotiented by the equivalence relation, is a partial order on the equivalence classes.*

The first rule checks that the elements of two related pairs are pairwise related, and that abstractions can be made equal by permutation. The second rule checks that there is no clash, and the third checks that there is no cycle.

Note that applying the morphism F to the propagation (resp. compatibility) rule gives exactly the rule (i) (resp. (ii)) above. We kept the essence of Paterson and Wegman's definition, adding just the management of nominal details so we can import (by F) properties from the first-order side.

Finally, we extend the notion of "minimal relation", which is based on an ordering defined as follows.

Definition 13. *The standard partial ordering over relations extends to nominal relations as follows:* $(\mathbb{R}_{?\approx_?,_1}, \mathbb{F}_1) \leq (\mathbb{R}_{?\approx_?,_2}, \mathbb{F}_2)$ *if and only if*

1. $\forall s, t, \pi\ \ s\ \mathbb{R}_{?\approx_?,_{1\pi}}\ t \Rightarrow \exists \pi' \quad s\ \mathbb{R}_{?\approx_?,_{2\pi'}}\ t \quad \{a\ \mathbb{F}_1\ t\}_{\pi(a)\neq\pi'(a)} \quad dom(\pi) \subseteq dom(\pi')$*, and*
2. $\forall a, s \quad a\ \mathbb{F}_1\ s \Rightarrow a\ \mathbb{F}_2\ s$

Once again, applying the morphism F to the definition above gives the standard definition of relation inclusion used by Paterson and Wegman.

Now we can state the same property as in [8] between valid relations and solutions of unification problems:

Proposition 4. *Two nominal terms s and t are unifiable if and only if there exist a valid nominal equivalence relation on the graph representing $s \approx_\alpha t$ such that $s \; \mathbb{R}_{?\approx_?,\mathsf{ld}} \; t$. If such a relation is minimal wrt the ordering in Definition 13, then it defines a most general unifier.*

Proof. If s and t are unifiable then let (σ, Δ) be a solution of the unification problem. Then we define R as the relation containing $s \; \mathbb{R}_{?\approx_?,\mathsf{ld}} \; t$ if and only if $s\sigma \approx_\alpha t\sigma$ under Δ, and $\mathbb{F}$ as the relation containing $a \; \mathbb{F} \; s$ if and only if $a \# s$ under Δ.

Conversely, if such a relation exists, σ can be derived from the dag by setting $\sigma(X) = \pi{\cdot}t$ if $X \; \mathbb{R}_{?\approx_?,\pi} \; t$ and t not a variable, or $\sigma(X) = X$ is there is no such t. And $\Delta = \{a \# X \mid a \; \mathbb{F} \; X \text{ and } \sigma(X) = X\}$.

4.2 A Quadratic Nominal Unification Algorithm

Paterson and Wegman's First-Order Linear Unification Algorithm (FLU) is linear in time and space in the size of the directed acyclic graph representing the problem.

The nominal version of FLU, called QNU, proceeds by computing a valid relation in the same way as FLU. There are two kinds of edges in the algorithm: the edges of d and undirected edges $s \; {}_{\pi^{-1}}\!\leftrightarrow_\pi \; t$ which represent $s \approx_\alpha \pi{\cdot}t$ (also $s \; \mathbb{R}_{?\approx_?,\pi} \; t$).

Proposition 5. QNU *is correct.*

Proof. Like FLU, QNU computes a valid nominal equivalence relation. So if the algorithm terminates without raising an error, then the computed relation is valid. Furthermore the relation is minimal. For more details we refer to [2].

Implementation. We have implemented the nominal unification algorithm in Objective Caml. Atoms, constants and variables are implemented as integers. Permutations and sets are implemented as arrays indexed by atoms. Composing and inverting permutations and permuting and computing the union of sets takes a linear time in the number of atoms in the problem. Accessing the image of an atom by permutation, or performing a membership test takes a constant time.

Proposition 6. *Let d be a nominal graph representing the problem $s \approx_\alpha t$, and let A_0 be the set of atoms in the problem. The complexity of $QNU(s,t)$ is at most $|A_0| \times (\mid F(d) \mid + n_\pi(d))$ where $|A_0|$ is the cardinal of A_0, $\mid F(d) \mid$ is the size of the first-order directed acyclic graph corresponding to d, and $n_\pi(d)$ is the number of permutation application $(\cdot)$ nodes in d.*

Proof. Let $F(d)$ be the direct acyclic graph obtained by applying F to every node in d the same way it is done with terms. Let us compare the execution of FLU and QNU. The main difference arises when checking (π)-compatibility. It may happen that two terms r and s are not π-compatible but $F(r)$ and $F(s)$ are compatible. In that case, QNU will halt on the error FAIL-CLASH while

FLU continues its execution. Otherwise, the execution of $FLU(F(s), F(t))$ and $QNU(s, t)$ pass through the same steps. Most of the operations which take n time in FLU take $|A_0| \times\ n$ in QNU. The only one which does not, is the edge creation. When creating an edge $s \xrightarrow{\pi}_e t$, s and t may be suspended terms but we want s and t not to be suspensions so we reduce them by applying permutations before creating the edge. Fortunately, this extra cost is bounded in the whole algorithm by $|A_0| \times \mid F(d) \mid$.

Remark 1. When $\mid d \mid$ is close to $\mid A_0 \mid \times (\mid F(d) \mid + n_\pi(d))$ and using mutable arrays for permutations and sets implementation, the unification algorithm becomes linear in time and space.

5 Conclusions

We have given the first formal account of the relationship between first-order and nominal languages. The morphism between structures defined in this paper extends to unification problems and we have used it to derive a quadratic nominal unification algorithm form the linear first-order unification algorithm given by Paterson and Wegman. An implementation of this algorithm is available from `http://www.dcs.kcl.ac.uk/pg/calves`.

References

1. Objective caml website:. `http://caml.inria.fr/ocaml/index.en.html`.
2. C. Calvès. Complexity and implementation of nominal algorithms. `http://www.dcs.kcl.ac.uk/pg/calves/thesis.pdf`, 2010. Ph.D. thesis, King's College London.
3. R.A. Clouston and A.M. Pitts. Nominal equational logic. *Electronic Notes in Theoretical Computer Science*, 172:223–257, 2007.
4. M. Fernandez and M.J. Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 2007.
5. M.J. Gabbay and A. Mathijssen. Nominal algebra. *Submitted STACS*, 7, 2006.
6. M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3):341–363, 2002.
7. J. Levy and M. Villaret. A quadratic nominal unification algorithm. In *Proceedings of RTA 2010*, 2010.
8. MS Paterson and MN Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186. ACM New York, NY, USA, 1976.
9. A.M. Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, 186(2):165–193, 2003.
10. M.R. Shinwell, A.M. Pitts, and M.J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274. ACM New York, NY, USA, 2003.
11. C. Urban, A.M. Pitts, and M.J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1):473–498, 2004.

Program Specialization for Verifying Infinite State Systems: An Experimental Evaluation

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[3], and Valerio Senni[2]

[1] Dipartimento di Scienze, University 'G. D'Annunzio',
Viale Pindaro 42, I-65127 Pescara, Italy
fioravanti@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[3] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present some improvements of a method for the automated verification of CTL temporal properties of infinite state reactive systems and we present an experimental evaluation of that improved method. One can apply our verification method based on program specialization to systems specified as constraint logic programs (CLP). First, we reformulate the verification method as a two-phase procedure: (1) in the first phase the CLP specification is specialized with respect to the initial state of the system and the temporal property to be verified, and (2) in the second phase we perform a bottom-up evaluation of the specialized program. In this paper we also propose some new strategies for performing program specialization during Phase (1). We evaluate the effectiveness of these new strategies, as well as that of some old ones, by presenting the results of experiments performed on several infinite state systems and temporal properties. Finally, we compare the implementation of our specialization-based verification method with various constraint-based model checking tools. The experimental results show that our specialization-based method is effective and competitive with respect to the methods used by those other tools.

1 Introduction

One of the most challenging problems in the verification of reactive systems, is the extension of the model checking technique (see [10] for a thorough overview) to infinite state systems. In model checking the evolution over time of an infinite state system is modelled as a binary transition relation over an infinite set of states and the properties of that evolution are specified by means of propositional temporal formulas. In particular, in this paper we consider the *Computation Tree Logic* (CTL, for short), which is a branching time propositional temporal logic by which one can specify, among others, the so-called *safety* and *liveness* properties [10].

Unfortunately, when considering infinite state systems, the verification of CTL formulas is an undecidable problem, in general. In order to cope with

this limitation, various *decidable subclasses* of systems and formulas have been identified (see, for instance, [1,16]). Other approaches enhance finite state model checking by using more general *deductive* techniques (see, for instance, [34,37]) or using *abstractions*, that is, mappings by which one can reduce an infinite state system to a finite state system, preserving the property of interest (see, for instance, [2,9,12,20,39]).

Also logic programming and constraint logic programming (CLP) have been proposed as frameworks for specifying and verifying properties of reactive systems. Indeed, the fixpoint semantics of logic programming languages allows us to easily represent the fixpoint semantics of various temporal logics [15,31,35]. Moreover, constraints over the integers or the rationals can be used to provide finite representations of infinite sets of states [15,19].

However, for programs which specify infinite state systems, the proof procedures normally used in constraint logic programming, such as the extension of SLDNF resolution and tabled resolution [8] to CLP, very often diverge when trying to check some given temporal properties. This is due to the limited ability of these proof procedures to cope with infinitely failed derivations. For this reason, instead of using direct program evaluation, many logic programming-based verification systems make use of reasoning techniques such as: (i) static analysis [5,15] or (ii) program transformation [17,27,29,32,36].

In this paper we further develop the verification method presented in [17] and we assess its practical value. That method is applicable to specifications of CTL properties of infinite state systems encoded as constraint logic programs and it makes use of program specialization.

The specific contributions of this paper are the following. First, we have reformulated the specialization-based verification method of [17] as a two-phase method. In Phase (1) the CLP specification is specialized w.r.t. the initial state of the system and the temporal property to be verified, and in Phase (2) the construction of the perfect model of the specialized program is performed via a bottom-up evaluation. We have shown in an experimental way that this bottom-up evaluation terminates in most examples without the need for any abstraction.

We have defined various generalization strategies which can be used during Phase (1) of our verification method, for controlling when and how to perform generalization. Selecting a good generalization strategy is not a trivial task: it must guarantee the termination of the specialization phase itself, by introducing a finite number of specialization sub-problems, and it should provide a good balance between precision and performance. Indeed, the use of a too coarse generalization strategy may prevent one from proving the properties of interest, while an unnecessarily precise strategy may lead to verification times which are too high. Since the states of the systems we consider are encoded as n-tuples of rationals, our generalization strategies have been specifically designed for CLP programs using the constraint domain of linear inequations over rationals.

We have implemented these strategies on the MAP transformation system [30], and we have applied them to several infinite state systems and properties taken from the literature. We have performed a comparative evaluation of generaliza-

tion strategies in terms of efficiency and power, based on the analysis of the experimental results.

Finally, we have compared our MAP implementation with various constraint-based model checkers for infinite state systems and, in particular, with ALV [7], DMC [15], and HyTech [22].

The paper is structured as follows. In Section 2 we recall how CTL properties of infinite state systems can be encoded by using locally stratified CLP programs. In Section 3 we present our two-phase verification method. In Section 4 we describe various strategies that can be applied during the specialization phase and the generalization techniques used for ensuring the termination of the program specialization phase. In Section 5 we report on some experiments we have performed by using a prototype implementation of the MAP transformation system. Finally, we compare the results we have obtained using the MAP system with the results we have obtained using other verification systems.

2 Specifying CTL Properties by CLP Programs

We will model an infinite state system as a *Kripke structure* and we will represent a property to be verified as a formula of the *Computation Tree Logic* (CTL, for short). The fact that a CTL formula φ holds in a state s of a Kripke structure $\mathcal{K}$ will be denoted by $\mathcal{K}, s \models \varphi$. (The reader who is not familiar with these notions may refer to the Appendix or to [10].) A Kripke structure can be encoded as a constraint logic program as indicated in the following four points [17,29,31].
(1) The set S of states is given as a set of n-tuples of the form $\langle t_1, \ldots, t_n \rangle$, where for $i = 1, \ldots, n$, the term t_i is either a rational number or an element of a finite domain. For reasons of simplicity, when denoting a state we will feel free to use a single variable X, instead of an n-tuple of variables of the form $\langle X_1, \ldots, X_n \rangle$.
(2) The set I of initial states is given as a set of clauses of the form:
$initial(X) \leftarrow c(X)$, where $c(X)$ is a constraint.
(3) The transition relation R is given as a set of clauses of the form:
$t(X, Y) \leftarrow c(X, Y)$
where $c(X, Y)$ is a constraint. Y is called a *successor state* of X. We also define a predicate ts such that, for every state X, $ts(X, Ys)$ holds iff Ys is a list of all the successor states of X, that is, for every state X, the state Y belongs to the list Ys iff $t(X, Y)$ holds. We refer to [18] for conditions that guarantee that Ys is a finite list and for an algorithm to construct the clauses defining ts from the clauses defining t.
(4) The elementary properties which are associated with each state X by the labeling function L, are given by a set of clauses of the form:
$elem(X, e) \leftarrow c(X)$
where e is a constant, that is, the name of an elementary property, and $c(X)$ is a constraint.

Given a Kripke structure $\mathcal{K}$ encoded by the clauses defining the predicates *initial*, t, ts, and *elem*, the satisfaction relation $\models$ can be encoded by a predicate *sat* defined by the following clauses [17,29,31]:

1. $sat(X, F) \leftarrow elem(X, F)$
2. $sat(X, not(F)) \leftarrow \neg sat(X, F)$
3. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1), sat(X, F_2)$
4. $sat(X, ex(F)) \leftarrow t(X, Y), sat(Y, F)$
5. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_2)$
6. $sat(X, eu(F_1, F_2)) \leftarrow sat(X, F_1), t(X, Y), sat(Y, eu(F_1, F_2))$
7. $sat(X, af(F)) \leftarrow sat(X, F)$
8. $sat(X, af(F)) \leftarrow ts(X, Ys), sat\_all(Ys, af(F))$
9. $sat\_all([\,], F) \leftarrow$
10. $sat\_all([X|Xs], F) \leftarrow sat(X, F), sat\_all(Xs, F)$

Note that $\{ex, eu, af\}$ is a complete set of operators, that is, all the CTL operators introduced in [10] can be defined in terms of ex, eu, and af. In particular, for every CTL formula φ, $ef(\varphi)$ can be defined as $eu(true, \varphi)$ and $eg(\varphi)$ can be defined as $not(af(not(\varphi)))$.

Let $P_{\mathcal{K}}$ denote the constraint logic program consisting of clauses 1–10 together with the clauses defining the predicates *initial*, t, ts, and *elem*.

Now we will present a method for proving that a given CTL formula φ holds. We start by defining a new predicate *prop* as follows:

$prop \equiv_{def} \forall X(initial(X) \rightarrow sat(X, \varphi))$

This definition can be encoded by the following two clauses:

γ_1 : $prop \leftarrow \neg negprop$

γ_2 : $negprop \leftarrow initial(X), sat(X, not(\varphi))$

The correctness of this encoding is stated by the following Theorem 1 and it is a consequence of the fact that program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is locally stratified and, hence, it has a unique perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ [4]. The proof proceeds by structural induction on the CTL formula φ and is omitted (see [18] for details).

Theorem 1 (Correctness of Encoding). *Let $\mathcal{K}$ be a Kripke structure, let I be the set of initial states of $\mathcal{K}$, and let φ be a CTL formula. Then,*

for all states $s \in I$, $\mathcal{K}, s \models \varphi$ iff $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$.

Example 1. Let us consider the reactive system depicted in Figure 1, where a term of the form $s(X_1, X_2)$ denotes the pair $\langle X_1, X_2 \rangle$ of rationals.

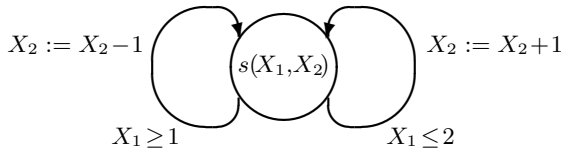


Fig. 1. A reactive system. In any initial state we have that $X_1 \leq 0$ and $X_2 = 0$. The transitions do not change the value of X_1.

The Kripke structure $\mathcal{K}$ which models that system, is defined as follows. The initial states are given by the clause:

11. $initial(s(X_1, X_2)) \leftarrow X_1 \leq 0,\ X_2 = 0$

The transition relation R is given by the clauses:

12. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \geq 1,\ Y_1 = X_1,\ Y_2 = X_2 - 1$
13. $t(s(X_1, X_2), s(Y_1, Y_2)) \leftarrow X_1 \leq 2,\ Y_1 = X_1,\ Y_2 = X_2 + 1$

The elementary property *negative* is given by the clause:

14. $elem(s(X_1, X_2), negative) \leftarrow X_2 < 0$

Let $P_{\mathcal{K}}$ denote the program consisting of clauses 1–14. We omit the clauses defining the predicate *ts*, which are not needed in this example.

Suppose that we want to verify the following property: in every initial state $s(X_1, X_2)$, where $X_1 \leq 0$ and $X_2 = 0$, the CTL formula $not(eu(true, negative))$ holds, that is, from any initial state it is impossible to reach a state $s(X_1', X_2')$ where $X_2' < 0$. By using the fact that $not(not(\varphi))$ is equivalent to φ, this property is encoded by the following two clauses:

γ_1: $prop \leftarrow \neg negprop$

γ_2: $negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ sat(s(X_1, X_2), eu(true, negative))$

Thus, by Theorem 1, in order to verify that in every initial state of $\mathcal{K}$ the CTL formula $not(eu(true, negative))$ holds, we need to show that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$. □

One of the most important features of the model checking techniques for finite state systems is the ability to find witnesses and counterexamples [10]. Our encoding of the Kripke structure can easily be extended to handle the generation of witnesses of formulas of the form $eu(\varphi_1, \varphi_2)$ and counterexamples of formulas of the form $af(\varphi)$. For details, the interested reader may refer to Section 7 of [18].

3 Verifying Infinite State Systems by Specializing CLP Programs

In this section we present a method for checking whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ is a CLP specification of an infinite state system and *prop* is a predicate encoding the satisfiability of a given CTL formula, as indicated in Section 2.

As already mentioned, the proof procedures normally used in constraint logic programming, such as the extensions to CLP of SLDNF resolution and tabled resolution, very often diverge when trying to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ by evaluating the query *prop*. This is due to the limited ability of these proof procedures to cope with infinite failure.

Also the bottom-up construction of the perfect model $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ often diverges because it does not take into account the information about the query *prop* to be evaluated and, in particular, about the initial states of the system and the formula to be verified. Indeed, by a naive bottom-up evaluation, the clauses of $P_{\mathcal{K}}$ may generate infinitely many atoms of the form $sat(s, \varphi)$. For instance, given a state s_0, an elementary property f that holds in s_0, and an infinite sequence $\{s_i \mid i \in \mathbb{N}\}$ of distinct states such that, for every $i \in \mathbb{N}$, $t(s_{i+1}, s_i)$ holds, clauses 5 and 6 generate by bottom-up evaluation the infinitely many atoms of the form: (i) $sat(s_0, f), sat(s_0, eu(true, f)), sat(s_0, eu(true, eu(true, f))), \ldots$, and of the form: (ii) $sat(s_i, eu(true, f))$, for every $i \in \mathbb{N}$.

In this paper we will show that the termination of the bottom-up construction of the perfect model may be improved by a prior application of program

specialization. In particular, in this section we will present a verification algorithm which is a reformulation of the method proposed in [17] and consists of two phases: (1) a first phase, in which we specialize the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ w.r.t. the query *prop*, thereby deriving a new program P_s, whose perfect model M_s satisfies the following equivalence: $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_s$, and (2) a second phase, in which we construct the perfect model M_s by a bottom-up evaluation.

The specialization phase, modifies the initial program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ by incorporating into the specialized program P_s the information about the initial states and the formula to be verified. The bottom-up evaluation of P_s may terminate more often than the bottom-up evaluation of the initial program because: (i) it may avoid generating a possibly infinite set of states that are unreachable from the initial states, and (ii) it will generate only specialized atoms corresponding to the subformulas of the formula to be verified.

The Verification Algorithm
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.
Output: An interpretation M_s such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M_s$.
(Phase 1) $Specialize(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}, P_s)$;
(Phase 2) $BottomUp(P_s, M_s)$

The *Specialize* procedure of Phase (1) makes use of the following transformation rules only: definition introduction, constrained atomic folding, positive unfolding, constrained atomic folding, removal of clauses with unsatisfiable body, and removal of subsumed clauses. Thus, Phase (1) is simpler than the specialization technique presented in [17] which uses some extra rules such as negative unfolding, removal of useless clauses, and contextual constraint replacement.

The Procedure *Specialize*
Input: The program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$.
Output: A stratified program P_s such that $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ iff $prop \in M(P_s)$.

$P_s := \{\gamma_1\}$; $InDefs := \{\gamma_2\}$; $Defs := \{\}$;
while there exists a clause γ in *InDefs*
do $Unfold(\gamma, \Gamma)$;
$Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$;
$P_s := P_s \cup \Phi$;
$InDefs := (InDefs - \{\gamma\}) \cup NewDefs$; $Defs := Defs \cup NewDefs$;
end-while

The *Unfold* procedure takes as input a clause $\gamma \in InDefs$ of the form $H \leftarrow c(X), sat(X, \psi)$, and returns as output a set Γ of clauses derived from γ as follows. The *Unfold* procedure first unfolds once γ w.r.t. $sat(X, \psi)$ and then applies zero or more times the unfolding as long as in the body of a clause derived from γ there is an atom of one of the following forms: (i) $t(s_1, s_2)$, (ii) $ts(s, ss)$, (iii) $sat(s, e)$, where e is an elementary property, (iv) $sat(s, not(\psi_1))$,

(v) $sat(s, and(\psi_1, \psi_2))$, (vi) $sat(s, ex(\psi_1))$, and (vii) $sat\_all(ss, \psi_1)$, where ss is a non-variable list. Then the set of clauses derived from γ by applying the unfolding rule is simplified by removing: (i) every clause whose body contains an unsatisfiable constraint, and (ii) every clause which is subsumed by a clause of the form $H \leftarrow c$, where c is a constraint. Due to the structure of the clauses defining the predicates t, ts, sat, and sat_all, the *Unfold* procedure terminates for any ground term denoting a CTL formula occurring in γ.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and the set *Defs* of clauses, called *definitions*. A definition in *Defs* is a clause of the form $newp(X) \leftarrow d(X), sat(X, \psi)$ which can be used for folding. The *Generalize&Fold* procedure introduces a set *NewDefs* of new definitions (which are then added to *Defs*) and, by folding the clauses in Γ using the definitions in $Defs \cup NewDefs$, derives a new set Φ of clauses, called specialized clauses, which are added to the program P_s. The specialized clauses in Φ are derived as follows. Suppose that η: $H \leftarrow e, L_1, \ldots, L_n$ is a clause in Γ, where for $i = 1, \ldots, n$, L_i is of the form either $sat(X, \psi_i)$ or $\neg sat(X, \psi_i)$. For $i = 1, \ldots, n$, *if* there exists a definition δ_i: $newp(X) \leftarrow d_i(X), sat(X, \psi_i)$ in *Defs* such that e implies $d_i(X)$, *then* γ is folded using δ_i, that is, $sat(X, \psi_i)$ is replaced by $newp(X)$, *else* a new definition ν_i: $newp(X) \leftarrow g(X), sat(X, \psi_i)$, such that e implies $g(X)$, is added to *NewDefs* and γ is folded using ν_i. More details on the *Generalize&Fold* procedure will be given in the next section.

An uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions and, therefore, it may determine the non-termination of the *Specialize* procedure. In order to guarantee termination, we will extend to constraint logic programs some techniques which have been proposed for controlling generalization in *positive supercompilation* [38] and *partial deduction* [25,28].

The output program P_s of the *Specialize* procedure is a *stratified* program and the procedure *BottomUp* computes the perfect model M_s of P_s by considering a stratum at a time, starting from the lowest stratum and going up to the highest stratum of P_s (see, for instance, [4]). Obviously, the model M_s may be infinite and the *BottomUp* procedure may not terminate.

In order to get a terminating procedure, we could compute an approximation of M_s by applying some standard techniques which are used in the static analysis of programs [11]. Indeed, in order to prove that $prop \in M_s$, we could construct a set $A \subseteq M_s$ such that $prop \in A$. Approximations (also called *abstractions*) are often used for the verification of infinite state systems (see, for instance, [2,5,9,12,15,20,39]). In general, integrating approximation mechanisms during the bottom-up construction of the perfect model, requires us to compute both over-approximations and under-approximations of models, because of the presence of negation. However, in this paper we will not address the issue of defining suitable approximations and we will focus our attention on the design of the *Specialize* procedure only. In Section 5 we show that after the application of our *Specialize* procedure, the construction of the model M_s terminates in several significant cases.

Example 2. Let us consider the reactive system $\mathcal{K}$ of Example 1. We want to check whether or not $prop \in M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$, where *prop* expresses the fact that $not(eu(true, negative))$ holds in every initial state.

Now we have that: (i) by using a traditional Prolog system, the evaluation of the query *prop* does not terminate in the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ because *negprop* has an infinitely failed SLD tree, (ii) by using the XSB tabled logic programming system, the query *prop* does not terminate because infinitely many *sat* atoms are tabled, and (iii) the bottom-up construction of $M(P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\})$ does not terminate because of the presence of clauses 5 and 6 as indicated at the beginning of this section.

By applying the *Specialize* procedure to the program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$ (with a suitable generalization strategy, as illustrated in the next section), we derive the following specialized program P_s:

γ_1. $prop \leftarrow \neg negprop$
γ_2'. $negprop \leftarrow X_1 \leq 0,\ X_2 = 0,\ new1(X_1, X_2)$
γ_3. $new1(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 = 0,\ Y_1 = X_1,\ Y_2 = 1,\ new2(Y_1, Y_2)$
γ_4. $new2(X_1, X_2) \leftarrow X_1 \leq 0,\ X_2 \geq 0,\ Y_1 = X_1,\ Y_2 = X_2 + 1,\ new2(Y_1, Y_2)$

Note that the *Specialize* procedure has propagated through the program P_s the constraint $X_1 \leq 0,\ X_2 = 0$ characterizing the initial states (see clause 11 of Example 1). This constraint, in fact, appears in clause γ_3 and its generalization $X_1 \leq 0,\ X_2 \geq 0$ appears in clause γ_4. The *BottomUp* procedure computes the perfect model of P_s, which is $M_s = \{prop\}$, in a finite number of steps. Thus, the property $not(eu(true, negative))$ holds in every initial state of $\mathcal{K}$. □

4 Generalization Strategies

The design of a powerful generalization strategy should meet two conflicting requirements: (i) the strategy should enforce the termination of the *Specialize* procedure, and (ii) over-generalization may produce a specialized program P_s with an infinite perfect model which may cause the non-termination of the *BottomUp* procedure. In this section we present several generalization strategies for coping with those conflicting requirements. These strategies combine various by now standard techniques used in the fields of program transformation and static analysis, such as *well-quasi orderings*, *widening*, and *convex hull* operators, and variants thereof [5,11,25,26,28,32,38]. All these strategies guarantee the termination of the *Specialize* procedure. However, as we deal with an undecidable verification problem, the power and effectiveness of the various generalization strategies can only be assessed by an experimental evaluation, which will be presented in the next section.

4.1 The *Generalize&Fold* Procedure

The *Generalize&Fold* procedure makes use of a tree, called *Definition Tree*, whose root is labelled by clause γ_2 (recall that $\{\gamma_2\}$ is the initial value of *InDefs*) and the non-root nodes are labelled by the clauses in *Defs*. Since, by construction, the clauses in *Defs* are all distinct, we will identify each clause with a node of that tree. The children of a clause γ in *Defs* are the clauses *NewDefs* derived by

applying *Unfold*(γ, Γ) followed by *Generalize&Fold*$(Defs, \Gamma, NewDefs, \Phi)$. Similarly to [25,26,28,38], our generalization technique is based on the combined use of *well-quasi ordering* relations and clause *generalization* operators. The well-quasi orderings guarantee that generalization is eventually applied, while generalization operators guarantee that each definition can be generalized a finite number of times only.

Let $\mathcal{C}$ be the set of all constraints and $\mathcal{D}$ be a fixed interpretation for the constraints in $\mathcal{C}$. We assume that: (i) every constraint in $\mathcal{C}$ is either an atomic constraint or a finite conjunction of constraints (we will denote conjunction by comma), and (ii) $\mathcal{C}$ is closed under projection, where the projection of a constraint c w.r.t. the variable X is a constraint, denoted *project*(c, X), such that $\mathcal{D} \models \forall(\mathit{project}(c, X) \leftrightarrow \exists X c)$. We define a partial order $\sqsubseteq$ on $\mathcal{C}$ as follows: for any two constraints c_1 and c_2 in $\mathcal{C}$, we have that $c_1 \sqsubseteq c_2$ iff $\mathcal{D} \models \forall\ (c_1 \rightarrow c_2)$.

Definition 1 (Well-Quasi Ordering). A *well-quasi ordering* (*wqo*, for short) on a set S is a reflexive, transitive, binary relation $\precsim$ such that, for every infinite sequence $e_0, e_1, \ldots$ of elements of S, there exist i and j such that $i < j$ and $e_i \precsim e_j$. Given e_1 and e_2 in S, we write $e_1 \approx e_2$ if $e_1 \precsim e_2$ and $e_2 \precsim e_1$. We say that a wqo $\precsim$ is *thin* iff for all $e \in S$, the set $\{e' \in S \mid e \approx e'\}$ is finite.

Definition 2 (Generalization Operator). Let $\precsim$ be a wqo on $\mathcal{C}$. A *generalization* operator on $\mathcal{C}$ w.r.t. the wqo $\precsim$, is a binary operator $\ominus$ such that, for all constraints c and d in $\mathcal{C}$, we have: (i) $d \sqsubseteq c \ominus d$, and (ii) $c \ominus d \precsim c$. (Note that, in general, $\ominus$ is not commutative.)

Similarly to the *widening* operator ∇ used in abstract interpretations [11], every infinite sequence of constraints constructed by using the generalization operator eventually stabilizes, that is, for every infinite sequence $d_0, d_1, \ldots$ of constraints, in the infinite sequence $c_0, c_1, \ldots$ defined as follows:

$c_0 = d_0$ and for any $i \geq 0$, $c_{i+1} = c_i \ominus d_{i+1}$,
there exist m and n, with $0 \leq m < n$, such that $c_m = c_n$.

The Procedure *Generalize&Fold*
Input: (i) a set *Defs* of definitions, and (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure.
Output: (i) A set *NewDefs* of new definitions, and (ii) a set Φ of folded clauses.

NewDefs $:= \emptyset$; $\Phi := \Gamma$;
while in Φ there exists a clause η: $H \leftarrow e, G_1, L, G_2$, where L is either $sat(X, \psi)$ or $\neg sat(X, \psi)$ *do*

GENERALIZE:
Let $e_p(X)$ be *project*(e, X).
1. *if* in *Defs* there exists a clause δ: $newp(X) \leftarrow d(X), sat(X, \psi)$ such that $e_p(X) \sqsubseteq d(X)$ (modulo variable renaming)
   then NewDefs := *NewDefs*
2. *elseif* there exists a clause α in *Defs* such that:
   (i) α is of the form $newq(X) \leftarrow b(X), sat(X, \psi)$, and (ii) α is the most recent ancestor of γ in the Definition Tree such that $b(X) \precsim e_p(X)$

then $NewDefs := NewDefs \cup \{newp(X) \leftarrow b(X) \ominus e_p(X), sat(X, \psi)\}$

3. *else* $NewDefs := NewDefs \cup \{newp(X) \leftarrow e_p(X), sat(X, \psi)\}$

FOLD:

$\Phi := (\Phi - \{\eta\}) \cup \{H \leftarrow e, G_1, M, G_2\}$, where M is $newp(X)$, if L is $sat(X, \psi)$, and M is $\neg newp(X)$, if L is $\neg sat(X, \psi)$

end-while

The following theorem, whose proof is given in [18], establishes that the *Specialize* procedure, which uses the *Unfold* and the *Generalize&Fold* subprocedures, always terminates and preserves the perfect model semantics.

Theorem 2 (Termination and Correctness of the *Specialize* Procedure). *For every input program $P_{\mathcal{K}} \cup \{\gamma_1, \gamma_2\}$, for every thin wqo $\preceq$, for every generalization operator $\ominus$, the Specialize procedure terminates. If P_s is the output program of the Specialize procedure, then prop $\in M(P_{\mathcal{K}})$ iff prop $\in M(P_s)$.*

4.2 Well-Quasi Orderings and Generalization Operators on Linear Constraints

In this section we describe the wqo's and the generalization operators we have used in our verification experiments.

We will consider the set Lin_k of constraints defined as follows. The atomic constraints of Lin_k are linear inequations constructed by using the predicate symbols $<$ and $\leq$, the k distinct variables $X_1, \ldots, X_k$, and the integer coefficients q_i's. The constraints in Lin_k are interpreted over the rationals in the usual way. Every constraint $c \in Lin_k$ is the conjunction $a_1, \ldots, a_m$ of $m\,(\geq 0)$ *distinct* atomic constraints and, for $i = 1, \ldots, m$, (1) a_i is of the form either $p_i \leq 0$ or $p_i < 0$, and (2) p_i is a polynomial of the form $q_0 + q_1 X_1 + \ldots q_k X_k$, where the q_i's are integer coefficients. An equation $r = s$ is considered as an abbreviation of the conjunction of the two inequations $r \leq s$ and $s \leq r$.

In every infinite state system we will consider, the state is represented as an n-tuple $\langle t_1, \ldots, t_n \rangle$, where k terms, with $k \leq n$, are rationals and the remaining $n-k$ terms are elements of a finite domain. As illustrated in Section 2, the initial states and the elementary properties are specified by constraints in Lin_k and the transition relation is specified by constraints in Lin_{2k}. (The $n-k$ components of a state which are elements of a finite domain, are specified by their values.)

Well-Quasi Orderings. Now we present three wqo's between polynomials and between constraints on Lin_k that we use in our verification examples of the next section. The three of them are based on the coefficients of the polynomials. The first wqo has been considered in [25,26,28,38] and the other two are simple variants of that wqo.

(W1) The wqo *HomeoCoeff*, denoted by $\preceq_{HC}$, compares sequences of absolute values of integer coefficients occurring in polynomials. The $\preceq_{HC}$ ordering is based on the notion of a *homeomorphic embedding* and takes into account commutativity and associativity of addition and conjunction. Given two polynomials with integer coefficients $p_1 =_{def} q_0 + q_1 X_1 + \ldots + q_k X_k$, and

$p_2 =_{def} r_0 + r_1X_1 + \ldots + r_kX_k$, we have that $p_1 \precsim_{HC} p_2$ iff there exist a permutation $\langle \ell_0, \ldots, \ell_k \rangle$ of the indexes $\langle 0, \ldots, k \rangle$ such that, for $i = 0, \ldots, k$, $|q_i| \leq |r_{\ell_i}|$. Given two atomic constraints $a_1 =_{def} p_1 < 0$ and $a_2 =_{def} p_2 < 0$, we have that $a_1 \precsim_{HC} a_2$ iff $p_1 \precsim_{HC} p_2$. Similarly, if we consider $a_1 =_{def} p_1 \leq 0$ and $a_2 =_{def} p_2 \leq 0$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_{HC} c_2$ iff there exist m *distinct* indexes $\ell_1, \ldots, \ell_m$, with $m \leq n$, such that $a_i \precsim_{HC} b_{\ell_i}$, for $i = 1, \ldots, m$.
(W2) The wqo *MaxCoeff*, denoted by $\precsim_{MC}$, compares the maximum absolute value of coefficients occurring in polynomials. For any atomic constraint a_i of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, we have that $\mathit{maxcoeff}(a_i) = max\{|q_0|, |q_1|, \ldots, |q_k|\}$, and for any two atomic constraints a_1, a_2, we have that $a_1 \precsim_{MC} a_2$ iff $\mathit{maxcoeff}(a_1) \leq \mathit{maxcoeff}(a_2)$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we have that $c_1 \precsim_{MC} c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_{MC} b_j$.
(W3) The wqo *SumCoeff*, denoted by $\precsim_{SC}$, compares the sum of the absolute values of the coefficients occurring in polynomials. For any atomic constraint a_i of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1X_1 + \ldots + q_kX_k$, , we define $\mathit{sumcoeff}(a_i) = \sum_{j=0}^{k} |q_j|$, and for any two atomic constraints a_1, a_2, we define $a_1 \precsim_{SC} a_2$ iff $\mathit{sumcoeff}(a_1) \leq \mathit{sumcoeff}(a_2)$. Given two constraints $c_1 =_{def} a_1, \ldots, a_m$, and $c_2 =_{def} b_1, \ldots, b_n$, we define $c_1 \precsim_{SC} c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_{SC} b_j$.

Generalization Operators. Now we present some generalization operators on Lin_k which we use in the verification examples of the next section.

(G1) Given any two constraints c and d, the generalization operator *Top*, denoted $\ominus_T$, returns *true*. It can be shown that $\ominus_T$ is indeed a generalization operator w.r.t. the wqo's *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*.

(G2) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and d, the generalization operator *Widen*, denoted $\ominus_W$, returns the conjunction $a_{i1}, \ldots, a_{ir}$, where $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$ (see [11] for a similar operator for static program analysis). It can be shown that $\ominus_W$ is indeed a generalization operator w.r.t. the wqo's *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*.

(G3) Given any two constraints $c =_{def} a_1, \ldots, a_m$, and $d =_{def} b_1, \ldots, b_n$, the generalization operator *WidenPlus*, denoted $\ominus_{WP}$, returns the conjunction $a_{i1}, \ldots, a_{ir}$, $b_{j1}, \ldots, b_{js}$, where: (i) $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$, and (ii) $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim c\}$. It can be shown that $\ominus_{WP}$ is indeed a generalization operator w.r.t. the wqo's *MaxCoeff* and *SumCoeff*. However, in general, $\ominus_{WP}$ is *not* a generalization operator w.r.t. *HomeoCoeff*, because the constraint $c \ominus_{WP} d$ may contain more atomic constraints than c and, thus, it may not be the case that $(c \ominus_{WP} d) \precsim_{HC} c$.

Some additional generalization operators can be defined by combining our three operators $\ominus_T$, $\ominus_W$, and $\ominus_{WP}$, with the *convex hull* operator, which is often used in the static analysis of programs [11]. Given two constraints c and d, their *convex hull*, denoted by $ch(c, d)$, is the least (w.r.t. the $\sqsubseteq$ ordering) constraint h such that $c \sqsubseteq h$ and $d \sqsubseteq h$.

Given any two constraints c and d, a wqo $\precsim$, and a generalization operator $\ominus$, we define the generalization operator $\ominus_{CH}$ as follows: $c \ominus_{CH} d =_{def} c \ominus ch(c, d)$. From the definitions of $\ominus$ and ch one can easily derive that the operator $\ominus_{CH}$ is indeed a generalization operator for c and d, that is, (i) $d \sqsubseteq c \ominus_{CH} d$, and (ii) $c \ominus_{CH} d \precsim c$.

(G4) Given any two constraints c and d, we define the generalization operator *CHWiden*, denoted $\ominus_{CHW}$, as follows: $c \ominus_{CHW} d =_{def} c \ominus_{W} ch(c, d)$.

(G5) Given any two constraints c and d, we define the generalization operator *CHWidenPlus*, denoted $\ominus_{CHWP}$, as follows: $c \ominus_{CHWP} d =_{def} c \ominus_{WP} ch(c, d)$.

Note that if we combine the generalization operator *Top* and the convex hull operator, we get the *Top* operator again.

5 Experimental Evaluation

In this section we report the results of the experiments we have performed on several examples of verification of infinite state reactive systems. We have implemented the verification algorithm presented in Section 2 by using MAP [30], an experimental system for the transformation of constraint logic programs. The MAP system is implemented in SICStus Prolog 3.12.8 and uses the `clpq` library to operate on constraints.

Now we give a brief description of the experiments we have conducted (see also Tables 1 and 2).

We have considered the following *mutual exclusion* protocols. (i) *Bakery* [23]: we have verified safety (that is, mutual exclusion) and liveness (that is, starvation freedom) in the case of two processes, and safety in the case of three processes; (ii) *MutAst* [24]: we have verified safety in the case of two processes; (iii) *Peterson* [33]: we have considered a *counting abstraction* [13] of this protocol and we have verified safety in the case of N processes; and (iv) *Ticket* [3]: we have considered the case of two processes and we have verified safety and liveness.

We have also verified safety properties of the following *cache coherence* protocols: (v) *Berkeley RISC*, (vi) *DEC Firefly*, (vii) *IEEE Futurebus+*, (viii) *Illinois University*, (ix) *MESI*, (x) *MOESI*, (xi) *Synapse N+1*, and (xii) *Xerox PARC Dragon*. These protocols are used in shared-memory multiprocessing systems for guaranteeing data consistency of the *local cache* associated with every processor [21]. We have considered *parameterized* versions of the above protocols, that is, protocols designed for an arbitrary number of processors. We have applied our verification method to the counting abstractions of the protocols described in [13].

We have also verified safety properties of the following systems. (xiii) *Barber* [6]: we have considered a parameterized version of this protocol with a single barber process and an arbitrary number of customer processes; (xiv) *Bounded* and *Unbounded Buffer*: we have considered protocols for two producers and two consumers which communicate via a bounded and an unbounded buffer, respectively (the encodings of these protocols are taken from [15]); (xv) *CSM*, which is a central server model described in [14]; (xvi) *Insertion Sort* and *Selection Sort*:

we have considered the problem of checking array bounds of these two sorting algorithms, parameterized w.r.t. the size of the array, as presented in [15]; (xvii) *Office Light Control* [7], which is a protocol for controlling how office lights are switched on or off, depending on room occupancy; (xviii) *Reset Petri Nets*, which are Petri Nets augmented with *reset arcs*: we have considered a reachability problem for a net which is a variant of one presented in [27] (unlike [27], in the net we have considered there is no bound on the number of tokens that can reside in a place and, therefore, our net is an infinite state system).

Table 1 shows the results of running the MAP system on the above examples by choosing different combinations of a wqo W and a generalization operator G introduced in Section 4. In the sequel we will denote that combination by $W\&G$. The combinations *MaxCoeff*&*CHWiden*, *MaxCoeff*&*CHWidenPlus*, *MaxCoeff*&*Top*, and *MaxCoeff*&*Widen* have been omitted because they give results which are very similar to those obtained by using *SumCoeff*, instead of *MaxCoeff*. *HomeoCoeff*&*CHWidenPlus* and *HomeoCoeff*&*WidenPlus* have been omitted because, as already mentioned in Section 4, these combinations do not satisfy the conditions given in Definition 2, and thus, they do not guarantee the termination of the specialization strategy.

If we consider *precision*, that is, the number of properties which have been proved, the best combination is *SumCoeff*&*WidenPlus* (23 properties proved out of 23) closely followed by *MaxCoeff*&*WidenPlus* and *SumCoeff*&*CHWidenPlus* (22 properties proved out of 23). The verification times for *SumCoeff*&*CHWidenPlus* are definitely worse than the ones for the other two combinations: indeed, on average, for each proved property, *SumCoeff*&*CHWidenPlus* takes 2990 milliseconds, while *MaxCoeff*&*WidenPlus* and *SumCoeff*&*WidenPlus* take 730 milliseconds and 820 milliseconds, respectively. This is due to the fact that *SumCoeff*&*CHWidenPlus* introduces more definitions than the other two strategies. (For lack of space, we do not report average times for the other generalization strategies, nor we present statistics on the number of generated definitions.) Table 1 also shows that by using the *Top* and the *Widen* generalization operators the specialization time is quite good, but the precision of verification is low. In other terms, the *Top* and *Widen* operators determine an over-generalization. In contrast, *SumCoeff*&*WidenPlus* and *MaxCoeff*&*WidenPlus* ensure a good balance between time and precision.

In conclusion, the specialization strategy which uses the generalization strategy *SumCoeff*&*WidenPlus* outperforms the others, closely followed by the specialization strategy which uses *MaxCoeff*&*WidenPlus*. In particular, the generalization strategies based either on the homeomorphic embedding (that is, *HomeoCoeff*) or the combination of the widening and convex hull operators (that is, *Widen* and *CHWiden*) are not the best choices in our examples.

In order to compare our MAP implementation of the verification method with other constraint-based model checking tools for infinite state systems, we have run the verification examples described above on the following systems: (i) ALV [7], which combines BDD-based symbolic manipulation for boolean and enumerated types, with a solver for linear constraints on integers, (ii) DMC [15],

Generalization G:	*CHWiden*		*CHWidenPlus*	*Top*		*Widen*		*WidenPlus*	
EXAMPLE wqo W:	*HC*	*SC*	*SC*	*HC*	*SC*	*HC*	*SC*	*MC*	*SC*
Bakery 2 (safety)	20	70	20	30	40	20	60	30	20
	20	50	20	20	30	20	40	30	20
Bakery 2 (liveness)	60	120	80	80	100	70	130	80	70
	40	80	60	50	60	50	90	60	50
Bakery 3 (safety)	160	800	180	2420	3010	170	750	180	160
	150	430	170	730	680	160	380	170	150
MutAst	230	440	440	2870	2490	220	370	70	140
	200	390	420	330	220	190	320	70	140
Peterson N	∞	∞	1370	∞	∞	∞	∞	210	230
	∞	410	1370	∞	30	∞	250	210	230
Ticket (safety)	30	30	20	20	30	20	30	20	40
	30	20	10	20	20	20	20	10	30
Ticket (liveness)	90	120	120	100	110	100	100	110	110
	50	70	70	60	60	60	50	60	60
Berkeley RISC	60	60	200	70	30	50	50	30	30
	60	40	170	50	20	50	30	30	30
DEC Firefly	190	120	340	100	80	180	120	30	20
	100	60	160	40	20	90	60	30	20
IEEE Futurebus+	∞	47260	47260	∞	15630	∞	4720	100	2460
	∞	290	290	∞	30	∞	230	100	270
Illinois University	50	80	40	140	90	50	70	40	20
	50	60	40	60	30	50	50	30	10
MESI	100	50	130	100	70	100	50	30	30
	80	40	120	50	20	80	40	30	30
MOESI	980	160	180	930	100	940	160	50	60
	950	60	80	860	30	910	60	50	50
Synapse N+1	30	10	10	20	20	20	10	10	10
	20	10	10	20	10	10	10	10	10
Xerox PARC Dragon	1230	80	280	1140	50	1210	70	30	40
	1180	60	260	1110	20	1160	50	30	40
Barber	41380	30150	2740	∞	∞	40750	29030	1210	1170
	3260	3100	2620	900	410	2630	1620	1170	1130
Bounded Buffer	73990	370	6790	71870	20	75330	340	3520	3540
	73190	170	6780	71850	20	74550	140	2040	2060
Unbounded Buffer	∞	∞	410	∞	∞	∞	∞	3890	3890
	310	130	410	140	10	280	100	360	360
CSM	∞	∞	4710	∞	∞	∞	∞	6380	6580
	∞	620	4700	30	20	∞	440	6300	6300
Insertion Sort	80	80	160	110	80	70	70	90	100
	80	60	150	30	20	70	50	90	100
Selection Sort	∞	∞	200	∞	∞	∞	∞	∞	190
	380	80	200	40	40	340	70	770	180
Office Light Control	40	50	50	40	30	50	50	50	50
	30	40	40	30	30	40	40	40	40
Reset Petri Nets	∞	∞	∞	∞	∞	∞	∞	0	0
	10	10	10	0	10	0	10	0	0

Table 1. Comparison of various generalization strategies used by the MAP system for proving properties of the examples of the first column. *HC*, *MC*, and *SC* denote the wqo *HomeoCoeff*, *MaxCoeff*, and *SumCoeff*, respectively. Times are expressed in milliseconds. The precision is 10 milliseconds. '0' means termination in less than 10 milliseconds. '∞' means 'No answer' within 100 seconds. For each example we have two lines: the first one shows the *total verification time* (Phases 1 and 2) and the second one shows the *program specialization time* (Phase 1 only).

which computes (approximated) least and greatest models of CLP(R) programs, and (iii) HyTech [22], a model checker for hybrid systems.

Table 2 reports the results of our experiments obtained by using various options available in those verification systems. All experiments with the MAP, ALV, DMC, and HyTech systems have been conducted on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system.

In terms of precision, MAP with the *SumCoeff&WidenPlus* option is the best system (23 properties proved out of 23), followed by DMC with the *A* option (19 out of 23), ALV with the *default* option (18 out of 23), and, finally, HyTech with the *Bw* option (17 out of 23). Among the above mentioned systems and respective options, HyTech (*Bw*) has the best average running time (70 milliseconds per proved property), followed by MAP and DMC (both 820 milliseconds), and ALV (8480 milliseconds). This is explained by the fact that the HyTech with the *Bw* option tries to prove a safety property with a very simple strategy, by constructing the reachability set backwards from the property to be proved, while the other systems apply more sophisticated techniques. Note also that the average verification times are affected by an uneven behavior on some specific examples. For instance, in the Bounded and Unbounded Buffer examples the MAP system has higher verification times with respect to the other systems, because these examples can be easily verified by backward reachability, thereby making the MAP specialization phase redundant. On the opposite side, MAP is much more efficient than the other systems in the Peterson and CSM examples.

6 Conclusions

In this paper we have proposed some improvements of the method presented in [17] for verifying infinite state reactive systems. First, we have reformulated the verification method as a two-phase method: (1) in the first phase a CLP specification of the reactive system is specialized w.r.t. the initial state and the temporal property to be verified and (2) in the second phase the perfect model of the specialized program is constructed in a bottom-up way. For Phase (1) we have considered various specialization strategies which employ different well-quasi orderings and generalization operators to guarantee the termination of the transformations. These orderings and generalization operators are either new or adapted from similar notions, such as convex hull, widening, and homeomorphic embedding, which have been defined in the context of static analysis of programs [5,11] and program specialization [25,26,28,32,38].

We have applied these specialization strategies to several examples of infinite state systems taken from the literature and we have compared the results in terms of efficiency and precision (that is, the time taken for the proofs and the number of properties which have been proved). On the basis of our experimental results we have found some strategies that outperform the others in terms of a good balance of efficiency and precision. In particular, the strategies based on the convex hull, widening, and homeomorphic embedding, do not appear to be the best strategies in our examples.

EXAMPLE	MAP	ALV				DMC		HyTech	
	SC&WidenPlus	*default*	*A*	*F*	*L*	*noAbs*	*Abs*	*Fw*	*Bw*
Bakery 2 (safety)	20	20	30	90	30	10	30	∞	20
Bakery 2 (liveness)	70	30	30	90	30	60	70	$\times$	$\times$
Bakery 3 (safety)	160	580	570	∞	600	460	3090	∞	360
MutAst	140	$\perp$	$\perp$	910	$\perp$	150	1370	70	130
Peterson N	230	71690	$\perp$	∞	∞	∞	∞	70	∞
Ticket (safety)	40	∞	80	30	∞	∞	60	∞	∞
Ticket (liveness)	110	∞	230	40	∞	∞	220	$\times$	$\times$
Berkeley RISC	30	10	$\perp$	20	60	30	30	∞	20
DEC Firefly	20	10	$\perp$	20	80	50	80	∞	20
IEEE Futurebus+	2460	320	$\perp$	∞	670	4670	9890	∞	380
Illinois University	20	10	$\perp$	∞	140	70	110	∞	20
MESI	30	10	$\perp$	20	60	40	60	∞	20
MOESI	60	10	$\perp$	40	100	50	90	∞	10
Synapse N+1	10	10	$\perp$	10	30	0	0	∞	0
Xerox PARC Dragon	40	20	$\perp$	40	340	70	120	∞	20
Barber	1170	340	$\perp$	90	360	140	230	∞	90
Bounded Buffer	3540	0	10	∞	20	20	30	∞	10
Unbonded Buffer	3890	10	10	40	40	∞	∞	∞	20
CSM	6580	79490	$\perp$	∞	∞	∞	∞	∞	∞
Insertion Sort	100	40	60	∞	70	30	80	∞	10
Selection Sort	190	∞	390	∞	∞	∞	∞	∞	∞
Office Light Control	50	20	20	30	20	10	10	∞	∞
Reset Petri Nets	0	∞	$\perp$	∞	10	0	0	∞	10

Table 2. Comparison of the MAP system with the ALV, DMC, and HyTech verification systems. Times are expressed in milliseconds. The precision is 10 milliseconds. (i) '0' means termination in less than 10 milliseconds. (ii) '$\perp$' means termination with the answer: 'Unable to verify'. (iii) '∞' means 'No answer' within 100 seconds. (iv) '$\times$' means that the test has not been performed (indeed, HyTech has no built-in for checking liveness). For the MAP system we show the total verification time with the *SumCoeff&WidenPlus* option (see the last column of Table 1). For the ALV system we show the time for four options: *default*, *A* (with approximate backward fixpoint computation), *F* (with approximate forward fixpoint computation), and *L* (with computation of loop closures for accelerating reachability). For the DMC system we show the time for two options: *noAbs* (without abstraction) and *Abs* (with abstraction). For the HyTech system we show the time for two options: *Fw* (forward reachability) and *Bw* (backward reachability).

Then, we have applied other model checking tools for infinite state systems (in particular, ALV [7], DMC [15], and HyTech [22]) to the same set of examples. The experimental results show that the specialization-based verification system (with the best performing strategies) is competitive with respect to the other tools.

Our approach is closely related to other verification methods for infinite state systems based on the specialization of (constraint) logic programs [27,29,32]. Unlike the approach proposed in [27,29] we use constraints, which give us very powerful means of dealing with infinite sets of states. The specialization-based verification method presented in [32] consists of one phase only, incorporating both top-down query directed specialization and bottom-up answer propagation. This method is restricted to definite constraint logic programs and makes use of a generalization technique which combines widening and convex hull computations for ensuring termination. In [32] only two examples have been presented (the Bakery protocol and a Petri net) and no verification times are reported. Thus, it is hard to make a comparison in terms of effectiveness with respect to the method we have presented here. However, as already mentioned, the generalization techniques based on the widening and the convex hull operators do not turn out to be the best options in our examples.

Another approach based on program transformation for verifying parameterized (and, hence, infinite state) systems has been presented in [36]. This approach is based on unfold/fold transformations which are more general than the ones used in the specialization strategy considered in this paper. However, the strategy for guiding the unfold/fold rules proposed in [36] works in fully automatic mode in a small set of examples only.

Finally, we would like to mention that our verification method can be viewed as complementary with respect to the methods based on the static analysis of constraint logic programs, such as [5,15]. These methods work by constructing approximations of program models (which can be least or greatest models, depending on the specific technique). These methods may fail to prove a property simply because they compute a subset (or a superset) of the program model.

One further enhancement of our method could be achieved by replacing the bottom-up, precise computation of the perfect model performed in our Phase (2) by an approximated computation in the style of [5,15]. Finding the optimal combination, in terms of both efficiency and precision, of program specialization and static analysis techniques for the verification of infinite state systems is an interesting direction for future research.

Acknowledgements

We thank the anonymous referees for very valuable comments. We also thank John Gallagher for providing the code of some of the systems considered in Section 5.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. *Proc. LICS'96*, pages 313–321. IEEE Press, 1996.
2. P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction (On efficient verification of parameterized systems). *International Journal of Foundations of Computer Science*, 20(5):779–801, 2009.
3. G. R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.
4. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
5. G. Banda and J. P. Gallagher. Constraint-based abstraction of a model checker for infinite state systems. *Proc. WLP 2009*. Potsdam, Germany, 2009.
6. T. Bultan. BDD vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems. *Proc. TACAS 2000*, LNCS 1785, pages 441–455. Springer, 2000.
7. T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. *Proc. ASE 2001*, pages 382–386. IEEE Press, 2001.
8. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
9. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *Proc. POPL'78*, pages 84–96. ACM Press, 1978.
12. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
13. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
14. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. *Proc. CSL '99*, *LNCS* 1683, pages 50–66. Springer, 1999.
15. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
16. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
17. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proc. VCL'01*, Tech. Rep. DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
18. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying infinite state systems by specializing constraint logic programs. Report 657, IASI-CNR, Roma, Italy, 2007.
19. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. *Proc. CONCUR'97*, LNCS 1243, pages 96–107. Springer, 1997.
20. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR '01*, Lecture Notes in Computer Science 2154, pages 426–440. Springer, 2001.
21. J. Handy. *The Cache Memory Book.* Morgan Kaufman, 1998. Second Edition.
22. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

23. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
24. D. Lesens and H. Saïdi. Abstraction of parameterized networks. *Electronic Notes of Theoretical Computer Science*, 9:41, 1997.
25. M. Leuschel. Improving homeomorphic embedding for online termination. *Proc. LOPSTR'98*, LNCS 1559, pages 199–218. Springer, 1999.
26. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*, LNCS 2566, pages 379–403. Springer, 2002.
27. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. *Proc. CL 2000*, LNAI 1861, pages 101–115. Springer, 2000.
28. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
29. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. *Proc. LOPSTR'99*, LNCS 1817, pages 63–82. Springer, 2000.
30. The MAP Transformation System. www.iasi.cnr.it/∼proietti/system.html, 2010.
31. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. CL 2000*, LNAI 1861, pages 384–398. Springer, 2000.
32. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of clp programs. *Proc. LOPSTR 2002*, LNCS 2664, pages 90–108, 2003.
33. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
34. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. *Proc. CAV'96*, LNCS 1102, pages 184–195. Springer, 1996.
35. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *Proc.CAV'97*, LNCS 1254, pages 143–154. Springer, 1997.
36. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. *Proc. TACAS 2000*, LNCS 1785, pages 172–187. Springer, 2000.
37. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15:49–74, 1999.
38. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. *Proc. ILPS'95*, pages 465–479. MIT Press, 1995.
39. L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (A survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.

Appendix

A *Kripke structure* [10] is a 4-tuple $\langle S, I, R, L\rangle$, where: (1) S is a set of *states*, (2) $I \subseteq S$ is a set of *initial states,* (3) $R \subseteq S \times S$ is a *transition relation* from states to states, and (4) L: $S \rightarrow \mathcal{P}(Elem)$ is a *labeling function* that assigns to each state $s \in S$ a subset $L(s)$ of *Elem*, that is, a set of elementary properties that hold in s. In particular, there exist the elementary properties *true*, *false*, and *initial* and we have that, for all $s \in S$, (i) $true \in L(s)$, (ii) $false \notin L(s)$, and (iii) $initial \in L(s)$ iff $s \in I$. We assume that R is a *total* relation, that is, for every state $s \in S$ there exists at least one state $s' \in S$, called a *successor state* of s, such that $s\, R\, s'$.

A *computation path* in a Kripke structure $\mathcal{K}$ is an *infinite* sequence of states $s_0\, s_1 \ldots$ such that $s_i\, R\, s_{i+1}$ for every $i \geq 0$.

The *Computation Tree Logic* (CTL, for short) [10] is a propositional temporal logic interpreted over a given Kripke structure. A CTL formula φ has the following syntax:

$\varphi ::= e \mid not(\varphi) \mid and(\varphi_1, \varphi_2) \mid ex(\varphi) \mid eu(\varphi_1, \varphi_2) \mid af(\varphi)$

where e belongs to the set *Elem* of the elementary properties.

The operators ex, eu, and af have the following semantics. The property $ex(\varphi)$ holds in a state s if there exists a successor s' of s such that φ holds in s'. The property $eu(\varphi_1, \varphi_2)$ holds in a state s if there exists a computation path π starting from s such that φ_1 holds in all states of a finite prefix of π and φ_2 holds on the rest of the path. The property $af(\varphi)$ holds in a state s if on every computation path π starting from s there exists a state s' where φ holds.

Formally, the semantics of CTL formulas is given by defining the satisfaction relation $\mathcal{K}, s \models \varphi$, which tells us when a formula φ holds in a state s of the Kripke structure $\mathcal{K}$.

Definition 3 (Satisfaction Relation for a Kripke Structure). Given a Kripke structure $\mathcal{K} = \langle S, I, R, L\rangle$, a state $s \in S$, and a formula φ, we inductively define the relation $\mathcal{K}, s \models \varphi$ as follows:

$\mathcal{K}, s \models e$ iff e is an elementary property belonging to $L(s)$

$\mathcal{K}, s \models not(\varphi)$ iff it is not the case that $\mathcal{K}, s \models \varphi$

$\mathcal{K}, s \models and(\varphi_1, \varphi_2)$ iff $\mathcal{K}, s \models \varphi_1$ and $\mathcal{K}, s \models \varphi_2$

$\mathcal{K}, s \models ex(\varphi)$ iff there exists a computation path $s_0\, s_1 \ldots$ in $\mathcal{K}$ such that $s = s_0$ and $\mathcal{K}, s_1 \models \varphi$

$\mathcal{K}, s \models eu(\varphi_1, \varphi_2)$ iff there exists a computation path $s_0\, s_1 \ldots$ in $\mathcal{K}$ such that (i) $s = s_0$ and (ii) for some $n \geq 0$ we have that: $\mathcal{K}, s_n \models \varphi_2$ and $\mathcal{K}, s_j \models \varphi_1$ for all $j \in \{0, \ldots, n-1\}$

$\mathcal{K}, s \models af(\varphi)$ iff for all computation paths $s_0\, s_1 \ldots$ in $\mathcal{K}$, if $s = s_0$ then there exists $n \geq 0$ such that $\mathcal{K}, s_n \models \varphi$. □

Verification of the Schorr-Waite algorithm – From trees to graphs

Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel

IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse

Abstract. This article proposes a method for proving the correctness of graph algorithms by manipulating their spanning trees enriched with additional references. We illustrate this concept with a proof of the correctness of a (pseudo-)imperative version of the Schorr-Waite algorithm by refinement of a functional one working on trees. It is composed of two orthogonal steps of refinement – functional to imperative and tree to graph – finally merged to obtain the result. Our imperative specifications use monadic constructs and syntax sugar, making them close to common imperative languages. This work has been realized within the Isabelle/HOL proof assistant.

Key words: Verification of imperative programs, Pointer algorithms, Program refinement

1 Introduction

The Schorr-Waite algorithm [15] is an in-place graph marking algorithm that traverses a graph without building up a stack of nodes visited during traversal. Instead, it codes the backtracking structure within the graph itself while descending into the graph, and restores the original shape on the way back to the root. Originally conceived to be a particularly space-efficient algorithm to be used in garbage collectors, it has meanwhile become a benchmark for studying pointer algorithms.

A correctness argument for the Schorr-Waite (SW) algorithm is non-trivial, and a large number of correctness proofs, both on paper and machine-assisted, has accumulated over the years. All these approaches have in common that they start from a low-level graph representation, as elements of a heap which are related by pointers (see Section 7 for a discussion).

In this paper, we advocate a development that starts from high-level structures (Section 2), in particular inductively defined trees, and exploits as far as possible the corresponding principles of computation (mostly structural recursion) and reasoning (mostly structural induction). We then proceed by refinement, along two dimensions: on the one hand, by mapping the inductively defined structures to a low-level heap representation (Sections 3 and 4), on the other hand, by adding pointers to the trees, to obtain genuine graphs (Sections 5). These two developments are joined in Section 6.

We argue that this method has several advantages over methods that manipulate lower-level structures:

- Termination of the algorithms becomes easier to prove, as the size of the underlying trees and similar measures can be used in the termination argument.
- Transformation and also preservation of structure is easier to express and to prove than when working on a raw mesh of pointers. In particular, we can state succinctly that the SW algorithm restores the original structure after having traversed and marked it.
- Using structural reasoning such as induction allows a higher degree of proof automation: the prover can apply rewriting which is more deterministic than the kind of predicate-logic reasoning that arises in a relational representation of pointer structures.

Technically, the main ingredients of our development are, on the higher level, spanning trees with additional pointers parting from the leaf nodes to represent arbitrary (finite) graphs. During the execution of the algorithm, the state space is partitioned into disjoint areas that may only be linked by pointers which satisfy specific invariants. On the lower level, we use state-transformer and state-reader monads for representing imperative programs. The two levels are related by a refinement relation that is preserved during execution of the algorithms. In this article, the refinement is carried out manually, but in the long run, we hope to largely automate this step.

Even though, taken separately, most of these ingredients are not new (see Section 7 for a discussion of related work), this paper highlights the fact that relatively complex graph algorithms can be dealt with elegantly when perceiving them as refinements of tree algorithms.

The entire development has been carried out in the Isabelle theorem prover [11], which offers a high degree of automation – most proofs are just a few lines long. The formalization itself does not exploit any specificities of Isabelle, but we use Isabelle's syntax definition facilities for devising a readable notation for imperative programs. A longer version of this paper with details of the proofs is available at [6].

2 Schorr-Waite on Pure Trees

A few words on notation before starting the development itself: Isabelle/HOL's syntax is a combination of mathematical notation and the ML language. Type variables are written $'a$, $'b$, ..., total functions from α to β are denoted by $\alpha \Rightarrow \beta$ and type constructors are post-fix by default (like $'a\ list$). $\longrightarrow/\Longrightarrow$ are both implication on term-level/meta-level where the meta-level is the domain of proofs. $[\![a_0;\ldots;a_n]\!] \Longrightarrow b$ abbreviates $a_0 \Longrightarrow (\ldots \Longrightarrow (a_n \Longrightarrow a)\ldots)$. Construction and concatenation operators on lists are represented by $x \# xs$ and $xs @ ys$. Sometimes we will judiciously choose the right level of nesting for pattern

matching in definitions, in order to take advantage of case splitting to improve automation in proofs.

The high-level version of the algorithm operates on inductively defined trees, whose definition is standard:

datatype $('a,'l)$ *tree* = *Leaf* $'l$ | *Node* $'a$ $(('a,'l)$ *tree*$)$ $(('a,'l)$ *tree*$)$

The SW algorithm requires a tag in each node, consisting of its mark, here represented by a boolean value (*True* for marked, *False* for unmarked) and a "direction" (left or right), telling the algorithm how to backtrack. We store this information as follows:

datatype *dir* = *L* | *R* **datatype** $'a$ *tag* = *Tag bool dir* $'a$

With these preliminaries, we can describe the SW algorithm. It uses two "pointers" t and p (which, for the time being, are trees): t points to the root of the tree to be traversed, and p to the previously visited node. There are three main operations:

- As long as the t node is unmarked, *push* moves t down the left subtree, turns its left pointer upwards and makes p point to the former t node. The latter node is then marked and its direction component set to "left".
- Eventually, the left subtree will have been marked, *i. e.* t's mark is *True*, or t is a Leaf. If p's direction component says "left", the *swing* operation makes t point to p's right subtree, the roles of p's left and right subtree pointers are reversed, and the direction component is set to "right".
- Finally, if, after the recursive descent, the right subtree is marked and p's direction component says "right", the *pop* operation will make the two pointers move up one level, reestablishing the original shape of t.

The algorithm is supposed to start with an empty p (a leaf), and it stops if p is empty again and t is marked. The three operations are illustrated in Figure 1 in which black circles represent marked nodes, white circles unmarked nodes, the directions are indicated by the arrows. Dots indicate intermediate steps and leaves are not represented.

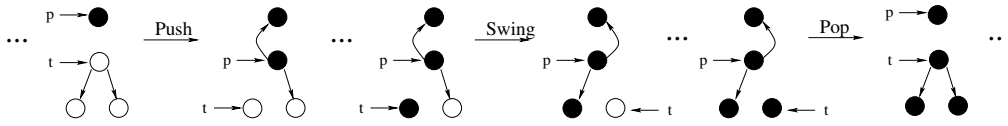


Fig. 1. Operations of the Schorr-Waite algorithm.

Our algorithm uses two auxiliary functions *sw-term* (termination condition) and *sw-body*, the body of the algorithm with three main branches as in the informal characterisation above. The function *sw-body* should not be called if t is marked and p is a Leaf, so it returns an insignificant result in this case.

```
fun sw-term :: (('a tag, 'l) tree * ('a tag, 'l) tree) ⇒ bool where
  sw-term (p, t) = (case p of
      Leaf - ⇒ (case t of Leaf - ⇒ True | (Node (Tag m d v) tlf tr) ⇒ m)
    | -      ⇒ False)

fun sw-body :: (('a tag, 'l) tree * ('a tag, 'l) tree)
         ⇒ (('a tag, 'l) tree * ('a tag, 'l) tree) where
  sw-body (p, t) = (case t of
     (Node (Tag False d v) tlf tr) ⇒ ((Node (Tag True L v) p tr), tlf)
   | - ⇒ (case p of
        Leaf - ⇒ (p, t)
      | (Node (Tag m L v) pl pr) ⇒ ((Node (Tag m R v) t pl), pr)
      | (Node (Tag m R v) pl pr) ⇒ (pr, (Node (Tag m R v) pl t))))
```

The SW algorithm on trees, *sw-tr*, is now easy to define, using the p and t pointers like a zipper data-structure [8]. We note in passing that *sw-tr* is tail recursive. If coding it in a functional programming language, your favorite compiler will most likely convert it to a while loop that traverses the tree without building up a stack.

```
function sw-tr :: (('a tag, 'l) tree * ('a tag, 'l) tree)
        ⇒ (('a tag, 'l) tree * ('a tag, 'l) tree) where
  sw-tr args = (if (sw-term args) then args else sw-tr (sw-body args))
```

We still have to prove the termination of the algorithm. We note that either the number of unmarked nodes decreases (during *push*), or it remains unchanged and the number of nodes with "left" direction decreases (during *swing*), or these two numbers remain unchanged and the p tree becomes smaller (during *pop*). This double lexicographic order is expressed in Isabelle as follows (with the predefined function *size*, and *unmarked-count* and *left-count* with obvious definitions):

```
termination sw-tr apply (relation measures [
  λ (p,t). unmarked-count p + unmarked-count t,
  λ (p,t). left-count p + left-count t,
  λ (p,t). size p])
```

Please note that the algorithm works on type ('*a tag*, '*l*) *tree* with an arbitrary type for the data in the leaf nodes, which will later be instantiated by types for references.

Let's take a look at some invariants of the algorithm. The first thing to note is that the t tree should be consistently marked: Either, it is completely unmarked, or it is completely marked. This is a requirement for the initial tree: a marked root with unmarked nodes hidden below would cause the algorithm to return prematurely, without having explored the whole tree. We sharpen this requirement, by postulating that in a t tree, the direction is "right" iff the node is marked. This is not a strict necessity, but facilitates stating our correctness theorem. We thus arrive at the following two properties *t-marked True* and *t-marked False* for t trees that are defined in one go:

```
primrec t-marked :: bool ⇒ ('a tag, 'l) tree ⇒ bool where
```

```
  t-marked m (Leaf rf) = True
| t-marked m (Node n l r) = (case n of (Tag m' d v) ⇒
    (((d = R) = m) ∧ m' = m ∧ t-marked m l ∧ t-marked m r))
```

We can similarly state a property of a p tree. We note that such a tree is composed of an upwards branch that is again a p-shaped tree, and a downwards branch that, depending on the direction, is either a previously marked t tree or an as yet unexplored (and therefore completely unmarked) t tree:

```
primrec p-marked :: ('a tag, 'l) tree ⇒ bool where
  p-marked (Leaf rf) = True
| p-marked (Node n l r) = (case n of (Tag m d v) ⇒ (case d of
      L ⇒ (m ∧ p-marked l ∧ t-marked False r)
    | R ⇒ (m ∧ t-marked True l ∧ p-marked r)))
```

We note in passing that these two properties are invariants of *sw-body*.

What should the correctness criterion for *sw-tr* be? We would like to state that *sw-tr* behaves like a traditional recursive tree traversal (implicitly using a stack!) that sets the mark to *True*. Unfortunately, SW not only modifies the mark, but also the direction, so the two components have to be taken into account:

```
fun mark-all :: bool ⇒ dir ⇒ ('a tag, 'l) tree ⇒ ('a tag, 'l) tree where
  mark-all m d (Leaf rf) = Leaf rf
| mark-all m d (Node (Tag m' d' v) l r) =
    (Node (Tag m d v) (mark-all m d l) (mark-all m d r))
```

By using the function *mark-all* we also capture the fact that the shape of the tree is unaltered after traversal. Of course, if a tree is consistently marked, it is not modified by marking with *True* and direction "right":

```
lemma t-marked-R-mark-all: t-marked True t ⟶ mark-all True R t = t
```

A key element of the correctness proof is that at each moment of the SW algorithm, given the p and t trees, we can reconstruct the shape of the original tree (if not its marks) by climbing up the p tree and putting back in place its subtrees:

```
fun reconstruct :: (('a tag, 'l) tree * ('a tag, 'l) tree) ⇒ ('a tag, 'l) tree where
  reconstruct (Leaf rf, t) = t
| reconstruct ((Node n l r), t) = (case n of (Tag m d v) ⇒ (case d of
      L ⇒ reconstruct (l, (Node (Tag m d v) t r))
    | R ⇒ reconstruct (r, (Node (Tag m d v) l t))))
```

For this reason, if two trees t and t' have the same shape (*i. e.* are the same after marking), they are also of the same shape after reconstruction with the same p.

Application of *sw-body* does not change the shape of the original tree that p and t are reconstructed to:

```
lemma sw-body-mark-all-reconstruct:
  ⟦ p-marked p; t-marked m' t; ¬ sw-term (p, t) ⟧ ⟹
  mark-all m d (reconstruct (sw-body (p, t))) = mark-all m d (reconstruct (p, t))
```

Obviously, if t is *t-marked* and we are in the final state of the recursion (*sw-term* is satisfied), then t is marked as true and p is empty. Together with the invariant of *sw-body* just identified, an induction on the form of the recursion of *sw-tr* gives us:

lemma *sw-tr-mark-all-reconstruct*:
let $(p,\ t) =$ *args in* $(\forall\ m.$ *t-marked* $m\ t \longrightarrow$ *p-marked* $p \longrightarrow$
let $(p',\ t') =$ (*sw-tr args*) *in*
mark-all True R (*reconstruct* $(p,\ t)) = t' \wedge (\exists\ rf.\ p' =$ *Leaf* $rf)))$

For a run of *sw-tr* starting with an empty p, we obtain the desired theorem (which, of course, is only interesting for the non-trivial case m=*False*):

theorem *sw-tr-correct*: *t-marked* $m\ t \implies$ *sw-tr* (*Leaf* $rf,\ t) = (p',\ t')$
$\implies t' =$ *mark-all True R* $t \wedge (\exists\ rf.\ p' =$ *Leaf* $rf)$

To show the brevity of the development, the full version of the paper [6] reproduces the entire Isabelle script up to this point, which is barely 5 pages long.

3 Imperative Language and its Memory Model

This section presents a way to manipulate low-level programs. We use a heap-transformer monad providing means to reason about monadic/imperative code along with a nice syntax, and that should allow similar executable code to be generated.

The theory Imperative_HOL [4] discussed in Section 7 already implements such a monad, however our development started independently of it and we have then used it to improve our version, without code generation for the moment.

The State Transformer Monad

In this section we define the state-reader and state-transformer monads and a syntax seamlessly mixing them. We encapsulate them in the SR – respectively ST – datatypes, as functions from a state to a return value – respectively a pair of return value and state.

We can escape from these datatypes with the *runSR* – respectively *runST* and *evalST* – functions which are intended to be used only in logical parts (theorems and proofs) and that should not be extractible.

datatype $('a,\ 's)$ *SR* $=$ *SR* $'s \Rightarrow 'a$ **datatype** $('a,\ 's)$ *ST* $=$ *ST* $'s \Rightarrow 'a \times 's$
primrec *runSR* $::$ $('a,\ 's)$ *SR* $\Rightarrow 's \Rightarrow 'a$ **where** *runSR* (*SR* $m) = m$
primrec *runST* $::$ $('a,\ 's)$ *ST* $\Rightarrow 's \Rightarrow 'a \times 's$ **where** *runST* (*ST* $m) = m$
abbreviation *evalST* **where** *evalST fm s* $==$ *fst* (*runST fm s*)

The *return* (also called *unit*) and *bind* functions for manipulating the monads are then defined classically with the infix notations $\trianglerighteq_{SR}$ and $\trianglerighteq_{ST}$ for *bind*s. We add also the function *SRtoST* translating state-reader monads to state-transformer monads and the function *thenST* (with infix notation $\triangleright_{ST}$) abbreviating binding without value transfer.

consts
returnSR :: $'a \Rightarrow ('a,\ 's)\ SR$
returnST :: $'a \Rightarrow ('a,\ 's)\ ST$
bindSR :: $('a,\ 's)\ SR \Rightarrow ('a \Rightarrow ('b,\ 's)\ SR) \Rightarrow ('b,\ 's)\ SR$ (**infixr** $\trianglerighteq_{SR}$)
bindST :: $('a,\ 's)\ ST \Rightarrow ('a \Rightarrow ('b,\ 's)\ ST) \Rightarrow ('b,\ 's)\ ST$ (**infixr** $\trianglerighteq_{ST}$)
SRtoST :: $('a,'s)\ SR \Rightarrow ('a,\ 's)\ ST$

We define also syntax translations to use the Haskell-like *do*-notation. The principal difference between the Haskell *do*-notation and this one is the use of state-readers for which order does not matter. With some syntax transformations, we can simply compose several state readers into one as well as give them as arguments to state writers, almost as it is done in imperative languages (for which state is the heap). In an adapted context – *i. e.* in $doSR\{\dots\}$ or $doST\{\dots\}$ – we can so use state readers in place of expressions by simply putting them in $\langle\dots\rangle$, the current state being automatically provided to them, only thanks to the syntax transformation which propagates the same state to all $\langle\dots\rangle$. We also add syntax for *let* ($letST\ x = a^{SR}; b^{ST}$) and *if* ($if\ (a^{SR})\ \{b^{ST}\}\ else\ \{c^{ST}\}$). For example with $f^{'a \Rightarrow ('b,\ 's)\ ST}$, $a^{('a,'s)\ SR}$, $g^{((),\ 's)\ ST}$ and $h^{'b \Rightarrow 'd}$, all these expressions are equivalent:

- $doST\ \{\ x \leftarrow f\ \langle a\rangle;\ g;\ returnST\ (h\ x)\}$
- $doST\ \{\ va \leftarrow SRtoST\ a;\ x \leftarrow f\ va;\ g;\ returnST\ (h\ x)\}$
- $ST\ (\lambda s.\ runST\ (f\ (runSR\ a\ s))\ s) \trianglerighteq (\lambda x.\ g \triangleright returnST\ (h\ x))$

We finally define the *whileST* combinator ($[v = v0]\ while\ (c)\ \{a\}$), inspired from the *while* combinator definition of the Isabelle/HOL library, the only difference with it being the encapsulation in monads:
$whileST\ b\ c\ v = (doST\{if(\langle b\ v\rangle)\{v' \leftarrow c\ v;\ whileST\ b\ c\ v'\}else\{returnST\ v\}\})$
$while\ b\ c\ v = (if\ b\ v\ then\ while\ b\ c\ (c\ v)\ else\ v)$

The Heap Transformer Monad

We define a heap we will use as the state in the state-reader/transformer monads. We represent it by an extensible record containing a field for the values.

As the Schorr-Waite algorithm doesn't need allocation of new references, our heap simply is a total function from references to values. (We use a record here because of developments already under way and needing further components.)

record $('n,\ 'v)\ heap = heap :: 'n \Rightarrow 'v$

We assume that we have a data type of references, which can either be *Null* or point to a defined location:

datatype $'n\ ref = Ref\ 'n \mid Null$

To read and write the heap, we define the corresponding primitives *read* and *write*. To access directly to the fields of structures in the heap, we also add the *get* $(a{\cdot}b)$, *rget* $(r \rightarrow b)$ and *rupdate* $(r \rightarrow b := v)$ operators, taking an access-and-update function (b) as argument.

4 Implementation for Pure Trees

In this section, we provide a low-level representation of trees as structures connected by pointers that are manipulated by an imperative program. This is the typical representation in programming languages like C, and it is also used in most correctness proofs of SW.

Data Structures

These structures are either empty (corresponding to a leaf with a null pointer, as we will see later) or nodes with references to the left and right subtree:

datatype $('a, 'r)$ *struct* = *EmptyS* | *Struct* $'a$ $('r\ ref)$ $('r\ ref)$

We define then access-and-update functions $\$v$ (value) $\$l$ (left) and $\$r$ (right) for the $('a, 'r)$ *struct* datatype, and access-and-update functions $\$mark$, $\$dir$ and $\$val$ for the $'a$ *tag* datatype.

Traditionally, in language semantics, the memory is divided into a heap and a stack, where the latter contains the variables. In our particular case, we choose a greatly simplified representation, because we just have to accommodate the variables pointing to the trees p and t. Our heap will be a type abbreviation for heaps whose values are structures:

types $('r, 'a)$ *str-heap* = $('r, ('a, 'r)$ *struct*$)$ *heap*

An Imperative Algorithm

We now have an idea of the low-level memory representation of trees and can start devising an imperative program that manipulates them (as we will see, with a similar outcome as the high-level program of Section 2). The program is a while loop, written in monadic style, that has as one main ingredient a termination condition:

constdefs
```
  sw-impl-term :: ('r ref × 'r ref) ⇒ (bool, ('r, 'v tag) str-heap) SR
  sw-impl-term vs == doSR { (case vs of (ref-p, ref-t) ⇒
   (case ref-p of
      Null ⇒ (case ref-t of Null ⇒ True | t ⇒ ((⟨ read t ⟩·$v)·$mark) )
    | - ⇒ False))}
```

The second main ingredient of the while loop is its body:

constdefs
```
  sw-impl-body :: ('r ref × 'r ref) ⇒ ('r ref × 'r ref, ('r, 'v tag) str-heap) ST
  sw-impl-body vs == (case vs of (p, t) ⇒ doST {
    if (case t of Null ⇒ True | - ⇒ (⟨read t⟩·$v)·$mark) {
      (if (⟨ p → ($v oo $dir) ⟩ = L) {   (** swing **)
          letST rt = ⟨p → $r⟩;
          p → $r := ⟨p → $l⟩;
          p → $l := t;
          p → ($v oo $dir) := R;
```

```
    returnST (p, rt)
  } else {                          (** pop **)
    letST rp = ⟨p → $r⟩;
    p → $r := t;
    returnST (rp, p) })
} else {                            (** push **)
  letST rt = ⟨t → $l⟩;
  t → $l := p;
  t → ($v oo $mark) := True;
  t → ($v oo $dir) := L;
  returnST (t, rt) } })
```

The termination condition and loop body are combined in the following imperative program:

```
constdefs sw-impl-tr :: ('r ref × 'r ref) ⇒ ('r ref × 'r ref, ('r, 'v tag) str-heap) ST
where
 sw-impl-tr pt == (doST{[vs = pt] while (¬⟨sw-impl-term vs⟩) {sw-impl-body vs}})
```

Correctness

Before we can describe the implementation of inductively defined trees in low-level memory, let us note that we need to have a means of expressing which node of a tree is mapped to which memory location. For this, we need trees adorned with address information:

```
datatype ('r, 'v) addr = Addr 'r 'v
```

For later use, we also introduce some projections (cf. definition of *tag* in Section 2) :

```
primrec addr-of-tag :: ('r, 'v) addr tag ⇒ 'r
where addr-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ ref)
primrec val-of-tag :: ('r, 'v) addr tag ⇒ 'v
where val-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ v)
```

We can now turn to characterizing the implementation relation of trees in memory, which we define gradually, starting with a relation which expresses that a (non-empty) node n with subtrees l and r is represented in state s. Remember that node n contains its address in memory. It is not possible that the structure at this address is empty. We therefore find a structure with a field corresponding to the value of n (just remove the address, which is not represented in the structure) and left and right pointers to the l and r subtrees:

```
primrec val-proj-of-tag :: ('r, 'v) addr tag ⇒ 'v tag where
 val-proj-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ (Tag m d v))

constdefs struct-alloc-in-state ::
  ('t ⇒ 'r ref) ⇒ ('r, 'v) addr tag ⇒ 't ⇒ 't ⇒ ('r,'v tag) str-heap ⇒ bool
  struct-alloc-in-state ac n l r s == (case (heap s (addr-of-tag n)) of
      EmptyS ⇒ False
    | Struct ns ref-l ref-r ⇒ ns = val-proj-of-tag n ∧ ref-l = ac l ∧ ref-r = ac r)
```

Please ignore the projection parameter *ac* for the moment. We will instantiate it with different functions, depending on whether we are working on trees (this section) or on graphs (in Section 6).

Given the representation of a node in memory, we can define the representation of a tree t in a state s: Just traverse the tree recursively and check that each node is correctly represented (*tree-alloc-in-state ac t s*). Finally, a configuration (the p and t trees) is correctly represented (*config-alloc-in-state* (p, t)) if each of the trees is, and the p variable contains a reference to the p tree, and similarly for t.

Let us now present the first intended instantiation of the *ac* parameter appearing in the above definitions: It is a function that returns the address of a non-empty node, and always *Null* for a leaf:

primrec *addr-of* :: $(('r, 'v)\ addr\ tag, 'b)\ tree \Rightarrow 'r\ ref$ **where**
  addr-of $(Leaf\ rf) = Null$
| *addr-of* $(Node\ n\ l\ r) = Ref\ (addr\text{-}of\text{-}tag\ n)$

We can now state our first result: for a couple of p and t trees correctly allocated in a state s, the low-level and high-level algorithms have the same termination behaviour:

lemma *sw-impl-term-sw-term*:
config-alloc-in-state addr-of pt $(ref\text{-}pt, s)$
  $\Longrightarrow$ *runSR* $(sw\text{-}impl\text{-}term\ ref\text{-}pt)\ s = sw\text{-}term\ pt$

Before discussing the correctness proof, let us remark that the references occurring in the trees have to be unique. Otherwise, the representation of a tree in memory might not be a tree any more, but might contain loops or joint subtrees. Given the list *reach* of references occurring in a tree, we will in the following require the p and t trees to have disjoint ("distinct") reference lists.

The correctness argument of the imperative algorithm is now given in the form of a simulation theorem: A computation with *sw-tr* carried out on trees p and t and producing trees p' and t' can be simulated by a computation with *sw-impl-tr* starting in a state implementing p and t, and ending in a state implementing p' and t'. We will not go into details here (please refer to the long version of the paper [6]), since the correctness argument for trees is just a light-weight version of the argument for graphs in Section 5.

5 Schorr-Waite on Trees with Pointers

The Schorr-Waite algorithm has originally been conceived for genuine graphs, and not for trees. We will now add "pointers" to our trees to obtain a representation of a graph as a spanning tree with additional pointers. This is readily done, by instantiating the type variables of the leaves to the type of references. Thus, a leaf can now represent a null pointer (*Leaf Null*), or a reference to r, of the form *Leaf* (*Ref r*).

For example, the graph of Figure 4 could be represented by the following tree, with references of type *nat*:

Node (*Tag False L* (*Addr 1* ()))
(*Node* (*Tag False L* (*Addr 2* ())) (*Leaf Null*) (*Leaf Null*))
(*Node* (*Tag False L* (*Addr 3* ())) (*Leaf* (*Ref 2*)) (*Leaf Null*))

A given graph might be represented by different spanning trees with additional pointers, but the choice is not indifferent: it is important that the graph is represented by a spanning tree that has an appropriate form, so that the low-level algorithm of Section 4 starts backtracking at the right moment. To characterize this form and to get an intuition for the simulation proof presented in Section 6, let's take a look at a "good" (Figure 2) and a "bad" spanning tree (Figure 3).

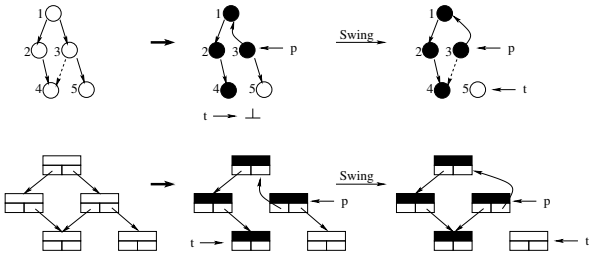


Fig. 2. Low-level graph with a "good" spanning tree

In Figure 2, the decisive step occurs when p points to node 3. Since the high-level algorithm proceeds structurally, it does not follow the additional pointer. Its t tree will therefore be a leaf, and the algorithm will start backtracking at this moment. The low-level representation does not distinguish between pointers to subtrees and additional pointers in leaf nodes, so that the t pointer will follow the link to node 4, just to discover that this node is already marked. For this reason, also the low-level algorithm will start backtracking.

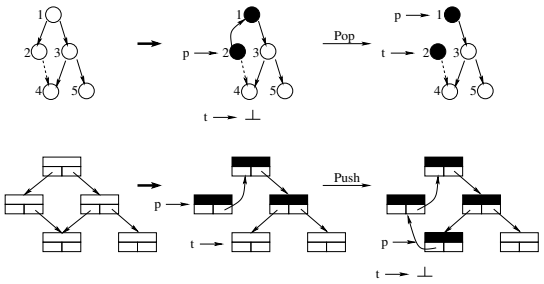


Fig. 3. The same graph with a bad spanning tree

The situation is different in Figure 3, when p reaches node 2. The high-level algorithm finishes its recursive descent at this point and starts backtracking, leaving node 4 unmarked. However, the low-level version proceeds to node 4 with its t pointer, marks it and starts exploring its subtrees. At this point, the high-level and low-level versions of the algorithm start diverging irrecoverably.

What then, more generally, is a "good" spanning tree? It is one where all additional pointers reference nodes that have already been visited before, in a pre-order traversal of the tree. This way, we can be sure that by following such an additional pointer, the low-level algorithm discovers a marked node and returns. We formalize this property by a predicate that traverses a tree recursively and checks that additional pointers only point to a set of allowed external references *extrefs*. When descending into the left subtree, we add the root to this set, and when descending into the right subtree, the root and the nodes of the left subtree.

primrec *t-marked-ext* :: $(('r, 'v)\ addr\ tag, 'r\ ref)\ tree \Rightarrow 'r\ set \Rightarrow bool$ **where**
t-marked-ext $(Leaf\ rf)\ extrefs = (case\ rf\ of\ Null \Rightarrow True \mid Ref\ r \Rightarrow r \in extrefs)$
$\mid$ *t-marked-ext* $(Node\ n\ l\ r)\ extrefs = (t\text{-}marked\text{-}ext\ l\ (insert\ (addr\text{-}of\text{-}tag\ n)\ extrefs)$
$\wedge\ t\text{-}marked\text{-}ext\ r\ ((insert\ (addr\text{-}of\text{-}tag\ n)\ extrefs) \cup set\ (reach\ l)))$

There is a similar property *p-marked-ext*, less inspiring, for p trees, which we do not give here. It uses in particular the function *reach-visited* giving the list of addresses of nodes that should have been visited.

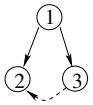


Fig. 4. A tree with additional pointers (drawn as dashed lines)

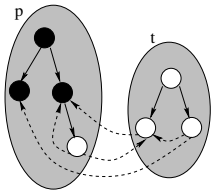


Fig. 5. Partitioning of trees

Let us insist on one point, because it is essential for the method advocated in this paper: Intuitively, we partition the state space into disjoint areas of addresses with their spanning trees, as depicted in Figure 5. The properties of these areas are described by the predicates *t-marked-ext* and *p-marked-ext*. Pointers may reach from one area into another area (the *extrefs* of the predicates). For the proof, it will be necessary to identify characteristic properties of these external references.

One of these is that all external references of a t tree only point to nodes marked as true. To prepare the ground, the following function gives us all the references of nodes that have a given mark, using the projection *nmark* for retrieving the mark from a tag:

primrec *marked-as-in* :: $bool \Rightarrow (('r, 'v)\ addr\ tag, 'l)\ tree \Rightarrow 'r\ set$ **where**
marked-as-in $m\ (Leaf\ rf) = \{\}$

```
| marked-as-in m (Node n l r) = (if (nmark n) = m then {addr-of-tag n} else {})
    ∪ marked-as-in m l ∪ marked-as-in m r
```

We can now show some invariants of function *sw-body* that will be instrumental for the correctness proof of Section 6. For example, if all external references of t only point to marked nodes of p, then this will also be the case after execution of *sw-body*:

```
lemma marked-ext-pres-t-marked-ext:
  ⟦ (p', t') = sw-body (p, t); ¬ sw-term (p, t); distinct (reach p @ reach t);
  p-marked-ext p (set (reach t)); t-marked-ext t (set (reach-visited p));
  t-marked m t; p-marked p; (set (reach-visited p)) ⊆ marked-as-in True p⟧
⟹ t-marked-ext t' (set (reach-visited p'))
```

To conclude this section, let us remark that the refinement of trees by instantiation of the leaf node type described here allows us to state some more invariants, but of course, the correctness properties of the original algorithm of Section 2 remain valid.

6 Implementation for Trees with Pointers

More surprisingly, we will now see that the low-level traversal algorithm of Section 4 also works for all graphs - without the slightest modification of the algorithm! The main justification has already been given in the previous section: There is essentially only one situation when the "pure tree" version and the "tree with pointers" version of the algorithm differ:

- In the high-level "pure tree" version, after a sequence of *push* operations, the t tree will become a leaf. This will be the case exactly when in the low-level version of the algorithm, the corresponding t pointer will become *Null*.
- In the high-level "tree with pointers" version, after some while, the t tree will also become a leaf, but in the low-level version, the t pointer might have moved on to a non-*Null* node. If the underlying spanning tree is well-formed (in the sense of *t-marked-ext*), the t will point to a marked node, so that the algorithm initiates backtracking in both cases.

We achieve this by taking into account the information contained in leaf nodes. Instead of the function *addr-of* of Section 4, we now parametrize our development with the function *addr-or-ptr* that also returns the references contained in leaf nodes:

```
primrec addr-or-ptr ::(('r, 'v) addr tag, 'r ref) tree ⇒ 'r ref where
  addr-or-ptr (Leaf rf) = rf
| addr-or-ptr (Node n l r) = Ref (addr-of-tag n)
```

We can now prove an extension of the lemma *sw-impl-term-sw-term* of Section 4 establishing the correspondence of the high-level and low-level termination conditions described above.

The proof for the simulation lemma now proceeds essentially along the same lines as before.

lemma *sw-impl-body-config-alloc-ext*:
⟦ *config-alloc-in-state addr-or-ptr* (*p*, *t*) (*vs*, *s*); *t-marked m t*; *p-marked p*;
¬ (*runSR* (*sw-impl-term vs*) *s*); *distinct* (*reach p* @ *reach t*);
p-marked-ext p (*set* (*reach t*)); *t-marked-ext t* (*set* (*reach-visited p*));
(*set* (*reach-visited p*)) ⊆ *marked-as-in True p* ⟧
⟹ *config-alloc-in-state addr-or-ptr* (*sw-body* (*p*, *t*)) (*runST* (*sw-impl-body vs*) *s*)

Now, the proof proceeds along the lines of the proof discussed in Section 4, yielding finally a preservation theorem for configurations:

lemma *impl-correct-ext*:
(∃ *m*. *t-marked m t*) ∧ *p-marked p* ∧ *distinct* (*reach p* @ *reach t*)
∧ *p-marked-ext p* (*set* (*reach t*)) ∧ *t-marked-ext t* (*set* (*reach-visited p*))
∧ (*set* (*reach-visited p*)) ⊆ *marked-as-in True p*
∧ *config-alloc-in-state addr-or-ptr* (*p*, *t*) (*vs*, *s*)
⟹ *config-alloc-in-state addr-or-ptr* (*sw-tr* (*p*, *t*)) (*runST* (*sw-impl-tr vs*) *s*)

If we start our computation with an empty p tree, some of the preconditions of this lemma vanish, as seen by a simple expansion of definitions. A tidier version of our result is then:

theorem *impl-correct-tidied*:
⟦ *t-marked m t*; *t-marked-ext t* {}; *distinct* (*reach t*);
tree-alloc-in-state addr-or-ptr t s; *sw-tr* (*Leaf Null*, *t*) = (*p′*, *t′*);
(*runST* (*sw-impl-tr* (*Null*, *addr-or-ptr t*)) *s*) = ((*p-ptr′*, *t-ptr′*), *s′*) ⟧
⟹ *tree-alloc-in-state addr-or-ptr t′ s′* ∧ *t-ptr′* = (*addr-or-ptr t′*)

7 Related Work

A considerable amount of work has accumulated on the Schorr-Waite algorithm in particular and on the verification of pointer algorithms in general. For reasons of space, we have to defer a detailed discussion to the full version [6], which also contains some references we cannot accommodate here.

Schorr-Waite: Since its publication [15], the algorithm has given rise to numerous publications which, in the early days, were usually paper proofs which often followed a transformational approach to derive an executable program. There are recently some fully automated methods that, however, are incomplete or cover only very specific correctness properties. Of more interest to us are proofs using interactive theorem provers, sparked by Bornat's proof using his Jape prover [2]. This work has later been shortened considerably in Isabelle [10], using a "split heap" representation. Similar in spirit are proofs using the Caduceus platform [7] and the KeY system [3]. A proof with the B method [1] follows the refinement tradition of program development.

Verification of imperative programs with proof assistants: There are two ways to obtain verified executable code: verify written code by abstracting it or generate it from abstract specification.

Haskabelle [13] allows to import Haskell code into Isabelle, which can then be used as specification, implementation or intermediate refinement like in [9]. Why/Krakatoa [5] is a general framework to generate proof obligations from annotated imperative programs like Java or C into proof assistants like Coq, PVS or Isabelle.

As to the second way, the Isabelle extractor generates SML, Caml, Haskell code from executable specifications. The theory Imperative_HOL [4] takes advantage of this and already implements a state-transformer monad with syntax transformations, and code extraction/generation.

We used a simple memory model as a total function from natural numbers to values which was sufficient in our case, but managing allocation could become hard. [14] compares several memory models and then presents a synthesis enjoying their respective advantages, which could be interesting for our work. Our state space partitioning bears similarities with methods advocated in Separation logic [12] – details of this connection still have to be explored.

8 Conclusions

We have presented a correctness proof of the Schorr-Waite algorithm, starting from a high-level algorithm operating on inductively defined trees, to which we add pointers to obtain genuine graphs. The low-level algorithm, written in monadic style, has been proved correct using a refinement relation with the aid of a simulation argument.

The Isabelle proof script is about 1000 lines long and thus compares favorably with previous mechanized proofs of Schorr-Waite, in particular in view of the fact that the termination of the algorithm and structure preservation of the graph after marking have been addressed. It is written in a plain style without particular acrobatics. It favors readability over compactness and can presumably be adapted to similar proof assistants without great effort.

The aim of the present paper is to advocate a development style starting from high-level, inductively defined structures and proceeding by refinement. There might be other approaches of verifying low-level heap manipulating algorithms (iterating, for example, over the number of objects stored in the heap and thus not exploiting structural properties). However, we hope to develop patterns that make refinement proofs easier and to partly automate them, thus further decreasing the proof effort.

We think that some essential concepts of our approach (representation of a graph by spanning trees with additional pointers, refinement to imperative programs in monadic style, partitioning of the heap space into subgraphs) can be adapted to other traditional graph algorithms: Often, the underlying structure of interest is indeed tree-shaped, whereas pointers are just used for optimization. In this spirit, we are currently verifying a class of BDD construction algorithms and we are planning to apply similar techniques in the context of Model Driven Engineering. Of course, we do not claim that our approach is universally applicable for graphs without deeper structure.

The present paper is primarily a case study, so there are still some rough edges: the representation of our imperative programs has to be refined, with the aim of allowing their compilation to standard programming languages like C or Java.

References

1. Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In *Formal Methods Europe (FME)*, LNCS 2805, pages 51–74, 2003.
2. Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction (MPC)*, LNCS 1837, pages 102–126, 2000.
3. Richard Bubel. The Schorr-Waite-Algorithm. In *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, chapter 15, pages 569–587. Springer Verlag, 2007.
4. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative Functional Programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOL)*, LNCS 5170, 2008.
5. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, LNCS 4590, pages 173–177. Springer, 2007.
6. Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. Verification of the Schorr-Waite algorithm - From trees to graphs, January 2010. `http://www.irit.fr/~Mathieu.Giorgino/Publications/SchorrWaite_TreesGraphs.html`.
7. Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2005.
8. Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.
9. Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4 — formally verifying a high-performance microkernel. In *International Conference on Functional Programming (ICFP)*. ACM, 2009.
10. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.
11. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.
12. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*, LNCS 2142, pages 1–19. Springer, 2001.
13. Tobias Rittweiler and Florian Haftmann. Haskabelle – converting Haskell source files to Isabelle/HOL theories, 2009. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/haskabelle.html`.
14. Norbert Schirmer and Makarius Wenzel. State spaces — the locale way. *ENTCS*, 254:161–179, 2009.
15. H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10:501–506, 1967.

Proving with ACL2 the correctness of simplicial sets in the Kenzo system.*

Jónathan Heras, Vico Pascual, and Julio Rubio

Departamento de Matemáticas y Computación, Universidad de La Rioja,
Edificio Vives, Luis de Ulloa s/n, E-26004 Logroño (La Rioja, Spain).
{jonathan.heras, vico.pascual, julio.rubio}@unirioja.es

Abstract. Kenzo is a Common Lisp system devoted to Algebraic Topology. Although Kenzo uses higher-order functional programming intensively, we show in this paper how the theorem prover ACL2 can be used to prove the correctness of first order fragments of Kenzo. More concretely, we report on the verification in ACL2 of the implementation of simplicial sets. By means of a generic instantiation mechanism, we achieve the reduction of the proving effort for each family of simplicial sets, letting ACL2 automate the routine parts of the proofs.

1 Introduction

Kenzo [4] is a computer algebra system devoted to Algebraic Topology and Homological Algebra calculations. It was created by Sergeraert following his ideas about *effective homology* [13], and has been successful in the sense that Kenzo has been capable of computing previously unknown homology groups. This implies that increasing user's trust in the system is relevant. To this aim, a wide project to apply formal methods in the study of Kenzo was launched several years ago.

One feature of Kenzo is it uses higher order functional programming to handle spaces of infinite dimension. Thus, the first attempts to apply theorem proving assistants in the analysis of Kenzo were oriented towards higher order logic tools. Concretely, the Isabelle/HOL proof assistant was used to verify a very important algorithm in Homological Algebra: the Basic Perturbation Lemma (see [2]). Let us note, however, that this formalization was related to *algorithms* and not to the real *programs* implemented in Kenzo. The problem of extracting programs from the Isabelle/HOL proofs has been dealt with in [3], but even there the programs are generated in ML, far from Kenzo. Because Kenzo is programmed in Common Lisp.

In this paper, we report on a verification on some fragments of the real Kenzo code, by means of the ACL2 theorem prover [7]. ACL2 works with first order logic. Hence, in a first step, Kenzo first order fragment are dealt with. A

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Community FP7, STREP project ForMath.

previous work on this area was published in [10], where the programs to deal with degeneracies in Kenzo were proven correct by means of ACL2.

To understand another source of first order fragments of Kenzo, let us explain the way of working with the system. As a first step, the user constructs some initial spaces (under the form of *simplicial sets* [11]) by means of some built-in Kenzo functions; then, in a second step, he constructs new spaces by applying topological constructions (as Cartesian products, loop spaces, and so on); as a third, and final, step, the user asks Kenzo for computing the homology groups of the spaces. The important point for our discussion is that only steps 2 and 3 need higher-order functional programming. The first step, the construction of *constant* spaces, can be modeled in a first order logic. Thus, this paper is devoted to an ACL2 infrastructure allowing us to prove the correctness of Kenzo programs for constructing these constant simplicial sets.

We started that work by examining each family of constant simplicial sets in Kenzo (spheres, Moore spaces, Eilenberg-MacLane spaces, and so on). By doing the first ACL2 proofs, it became quickly clear that all the proofs match a common pattern. So, we carefully analyzed which is the common part for all the constant spaces, and which is particular for each family of simplicial sets. This led us to Theorem 3 in Subsection 3.2. The proof of this theorem in ACL2 allows us to use the generic instantiation tool by Martín-Mateos et al. [9], to produce ACL2 proofs for concrete families of constant simplicial sets. This has been done, up to now, for the spheres and for the simplicial sets coming from simplicial complexes. It is worth noting that from 4 definitions and 4 theorems the instantiation tool generates (and instantiates) 15 definitions and 375 theorems. The benefits in terms of proof effort are considerable. These two issues are the main contributions of this paper.

The unique difference between actual code and ACL2 verified code is the transformation of loops into ACL2 recursive functions. Since this transformation is very well-known and quite safe, we consider that the verified code is as close as possible to the real Kenzo programs.

The methodological approach has been imported from [10]. EAT [12] was the predecessor of Kenzo. The EAT system is also based on Sergeraert's ideas, but its Common Lisp implementation was closer to the mathematical theory. This means a poorer performance (since in Kenzo the algorithms have been optimized), but also that the ACL2 verification of EAT programs is easier. Thus, the main idea is to prove first in ACL2 the correctness of EAT programs, and then, by a domain transformation, to translate the proofs to Kenzo programs. To this aim, an intermediary model, based on binary numbers, is employed (this was introduced in [10], too).

The complete ACL2 code of our formalization can be found at [5].

The organization of the paper is as follows. Section 2 is devoted to mathematical preliminaries and to their concrete materialization in the EAT and Kenzo systems. The main theorems to factorize the proofs which are common to all the spaces are presented in Section 3 (both in the EAT and in the Kenzo models).

Technical issues about the ACL2 proofs are dealt with in Section 4. The paper ends with conclusions, future work and the bibliography.

2 Simplicial models

In this section, we present the minimal mathematical preliminaries to understand the rest of the paper, and we explain how the elementary data structures are encoded in both systems EAT and Kenzo.

2.1 A mathematical model

The following definition is the most important one in this work (see [11] for the context and further details).

Definition 1 A *simplicial set* K, is a union $K = \bigcup_{q\geq 0} K^q$, where the K^q are disjoints sets, together with functions:

$$\begin{array}{lll} \partial_i^q : K^q \to K^{q-1}, & q > 0, & i = 0, \ldots, q, \\ \eta_i^q : K^q \to K^{q+1}, & q \geq 0, & i = 0, \ldots, q, \end{array}$$

subject to the relations:

$$\begin{array}{llcll} (1)\ \partial_i^{q-1}\partial_j^q & = & \partial_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (2)\ \eta_i^{q+1}\eta_j^q & = & \eta_j^{q+1}\eta_{i-1}^q & \text{if} & i > j, \\ (3)\ \partial_i^{q+1}\eta_j^q & = & \eta_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (4)\ \partial_i^{q+1}\eta_i^q & = & identity & = & \partial_{i+1}^{q+1}\eta_i^q, \\ (5)\ \partial_i^{q+1}\eta_j^q & = & \eta_j^{q-1}\partial_{i-1}^q & \text{if} & i > j+1, \end{array}$$

The functions ∂_i^q and η_i^q are called *face* and *degeneracy* maps, respectively.

The elements of K^q are called *q-simplexes*. A q-simplex x is *degenerate* if $x = \eta_i^{q-1} y$ for some simplex y, and for some degeneracy map η_i^{q-1}; otherwise x is *non degenerate*.

In the rest of the paper a non degenerate simplex will be called *geometric* simplex, to stress that only these simplexes really have a geometric meaning; the degenerate simplexes can be understood as *formal* artifacts introduced for technical (combinatorial) reasons. This becomes clear in the following discussion.

The next essential result, which follows from the commuting properties of degeneracy maps in Definition 1, was modeled and proved by means of the ACL2 theorem prover in [1].

Proposition 1 Let K be a simplicial set. Any n-simplex $x \in K^n$ can be expressed in a unique way as a (possibly) iterated degeneracy of a non-degenerate simplex y in the following way:

$$x = \eta_{j_k} \ldots \eta_{j_1} y$$

with $y \in K^r$, $k = n - r \geq 0$, and $0 \leq j_1 < \cdots < j_k < n$.

In the previous statement the super-indexes in the degeneracy maps have been skipped, since they can be inferred from the context. It is a usual practice and will be freely used in the sequel, both for degeneracy and for face maps.

This proposition allows us to encode all the elements (simplexes) of *any* simplicial set in a generic way, by means of a structure called *abstract simplex*. More concretely, an *abstract simplex* is a pair $(dgop \quad gmsm)$ consisting of a sequence of degeneracy maps *dgop* (which will be called a *degeneracy operator*) and a geometric simplex *gmsm*. The indexes in a degeneracy operator *dgop* must be in strictly decreasing order. For instance, if σ is a non-degenerate simplex, and σ' is the degenerate simplex $\eta_1\eta_2\sigma$, the corresponding abstract simplexes are respectively $(\emptyset \quad \sigma)$ and $(\eta_3\eta_1 \quad \sigma)$, as $\eta_1\eta_2 = \eta_3\eta_1$, due to equality (2) in Definition 1. Of course, the nature of geometric simplexes depends on the concrete simplicial set we are dealing with, but the notion of abstract simplex allows us a generic handling of all the elements in our proofs.

Equation (2) in Definition 1 allows one to apply a degeneracy map η_i over a degeneracy operator *dgop* to obtain a new degeneracy operator. Let us consider, for example, η_4 and the degeneracy operator $\eta_5\eta_4\eta_1$; then $\eta_4\eta_5\eta_4\eta_1 = \eta_6\eta_4\eta_4\eta_1 = \eta_6\eta_5\eta_4\eta_1$. We will use the notation $\eta_i \circ dgop$ for the resulting degeneracy operator; in our example: $\eta_4 \circ (\eta_5\eta_4\eta_1) = \eta_6\eta_5\eta_4\eta_1$.

We can also try to apply a face map ∂_i over a degeneracy operator *dgop*. But now, there are two cases, according to whether the indexes i or $i-1$ appear in *dgop*. If they do not appear, then there is a face that *survives* in the process (for instance: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). Otherwise, the cancellation equation (4) from Definition 1 applies and the result of the process is simply another degeneracy operator (example: $\partial_4\eta_5\eta_3\eta_0 = \eta_4\partial_4\eta_3\eta_0 = \eta_4\eta_0$). We will denote by $\partial_i \circ dgop$ the output degeneracy operator, in both cases (in our examples: $\partial_4 \circ (\eta_5\eta_2\eta_0) = \eta_4\eta_2\eta_0$ and $\partial_4 \circ (\eta_5\eta_3\eta_0) = \eta_4\eta_0$).

With these notational conventions, the behaviour of face and degeneracy maps over abstract simplexes is characterized as follows:

$$\eta_i^q(dgop \quad gmsm) := (\eta_i^q \circ dgop \quad gmsm)$$
$$\partial_i^q(dgop \quad gmsm) := \begin{cases} (\partial_i^q \circ dgop \quad gmsm) & \text{if } \eta_i \in dgop \vee \eta_{i-1} \in dgop \\ (\partial_i^q \circ dgop \quad \partial_k^r gmsm) & \text{otherwise;} \end{cases}$$

where

- $r = q - \{\text{number of degeneracies in } dgop\}$ and
- $k = i - \{\text{number of degeneracies in } dgop \text{ with index lower than } i\}$[1].

Note that the degeneracy map expressed in terms of abstract simplexes only affects the degeneracy operator of the abstract simplex; therefore degeneracy maps can be implemented independently from the simplicial set. On the contrary,

[1] In fact, we are still abusing the notation here, since the face of a geometric simplex is an abstract simplex and, sometimes, a degenerate one; this implies that to get a correct representation of $(\partial_i^q \circ dgop \quad \partial_k^r gmsm)$ as an abstract simplex $(dgop' \quad gmsm')$ we should compose the degeneracy operator $\partial_i^q \circ dgop$ with that coming from $\partial_k^r gmsm$.

face maps can depend on the simplicial set because, when $\eta_i \notin dgop$ and $\eta_{i-1} \notin dgop$, the application of a face map ∂_i *arrives* until the geometric simplex, and, this, of course, requires some knowledge from the concrete simplicial set where the computation is carried out.

This observation will be very important in the sequel, since it indicates which parts of the proofs could be automatized (those independent from the concrete simplicial sets), and which parts must be explicitly provided by the user.

Furthermore, it is necessary to characterize the pattern of the admissible abstract simplexes for a given simplicial set, since it will allow us to determine over which elements the proofs will be carried out. The following property gives such a characterization.

Proposition 2 Let K be a simplicial set and $absm = (dgop \;\; gmsm)$ be a pair where $dgop$ is a degeneracy operator and $gmsm$ is a geometric simplex of K. Then, $absm$ is an abstract simplex of K in dimension q if and only if the following properties are satisfied:

1. $gmsm \in K^r$, for some natural number $r \leq q$;
2. the length of the sequence of degeneracies $dgop$ is $q - r$;
3. the index of the first degeneracy in $dgop$ is lower than q.

Each concrete representation for degeneracy operators defines a different model to encode elements of simplicial sets. In the following two subsections we will explain the EAT and the Kenzo models, respectively.

2.2 The EAT model

An abstract simplex, $absm$, is represented internally in the EAT system by a Lisp object: $(dgop \;\; gmsm)$ where $dgop$ is a strictly decreasing integer list (a *degeneracy list*) which represents a sequence of degeneracy maps, and $gmsm$ is a geometric simplex (whose type is left unspecified). For example, if we retake the examples introduced in the previous subsection $(\emptyset \;\; \sigma)$ and $(\eta_3\eta_1 \;\; \sigma)$, the corresponding EAT objects are respectively $(nil \;\; \sigma)$ and $((3 \;\; 1) \;\; \sigma)$, where nil stands for the empty list in Lisp.

Now, we can implement an invariant function in Lisp, which is a predicate indicating when a Lisp pair is an abstract simplex in EAT. The invariant translates the conditions from Proposition 2:

1. $gmsm$ is an element of K^r (this information depends on K and must be implemented for each simplicial set);
2. the length of the $dgop$ list is equal to $q - r$;
3. the first element of $dgop$ is lower than q.

With respect to the EAT representation of a simplicial set, it is based on considering a simplicial set as a tuple of functional objects. Since the degeneracy maps do not depend on the simplicial set, only two functional slots are needed: one for recovering faces (on *geometric* simplexes), and other with information about the encoding of geometric simplexes.

As mentioned previously, we want to focus on the simplicial sets of the Kenzo system. The EAT model has been used as a simplified formal model to reduce the gap between the mathematical structures and their Kenzo representations.

2.3 The Kenzo model

Both EAT and Kenzo systems are based on the same Sergeraert's ideas, but the performance of the EAT system is much poorer than that of Kenzo. One of the reasons why Kenzo performs better than EAT is because of a smart encoding of degeneracy operators. Since generating and composing degeneracy lists are operations which appear in an exponential way in most of Kenzo calculations (through the Eilenberg-Zilber theorem [11]), it is clear that having a better way for storing and processing degeneracy operators is very important. But, no reward comes without a corresponding price, and the Kenzo algorithms are somehow obscured, in comparison to the clean and comprehensible EAT approach.

An abstract simplex, *absm*, is represented internally in the Kenzo system by a Lisp object: (*dgop* *gmsm*) where *dgop* is a non-negative integer coding a strictly decreasing integer list and *gmsm* is a geometric simplex. The strictly decreasing integer list represents a sequence of η operators and is coded as a *unique* integer. Let us explain this with an example: the degeneracy list (3 1) can equivalently be seen as the binary list (0 1 0 1) in which 1 is in position i if the number i is in the degeneracy list, and 0 otherwise. This list, interpreted as a binary number in the reverse order, defines the natural number 10. Thus, Kenzo encodes the degeneracy list (3 1) as the natural number 10. The empty list is encoded by the number 0. Then, the abstract simplexes ($\emptyset$ σ) and ($\eta_3\eta_1$ σ) are implemented in the Kenzo system as (0 σ) and (10 σ), respectively.

With this representation, we will say that an index is an *active bit* in a natural number representing a degeneracy operator if the index appears in the degeneracy operator.

With this representation and notation, the invariant function for abstract simplexes in Kenzo can be defined according to:

1. *gmsm* is an element of K^r;
2. the number of active bits in *dgop* is equal to $q - r$;
3. $dgop < 2^q$.

As Kenzo encodes degeneracy lists as integers, the face and degeneracy maps can be implemented using very efficient Common Lisp primitives dealing with binary numbers (such as *logxor*, *ash*, *logand*, *logbitp* and so on). This is one of the reasons why Kenzo dramatically improves the performance of its predecessor EAT. Nevertheless, these efficient operators have a more obscure semantics than their counterparts in EAT.

The Kenzo representation of a simplicial set follows the same pattern that the EAT one, that is, a simplicial set is a tuple of functional λ-expressions.

In order to establish an infrastructure to prove the objects handled in Kenzo as simplicial sets are *really* simplicial sets (in other words, they satisfy the equations in Definition 1), the following strategy has been followed (inspired from that of [10]). First, the correctness of the EAT representation will be proven. In a second step, a proof of the correctness of the domain transformations between the EAT and Kenzo representations will be built. Then, it will become easy to prove a property about a Kenzo operator by first proving the property about the EAT one (which is usually much simpler) and then translating it to Kenzo, by means of the domain transformations theorems. These tasks have been fulfilled by means of the ACL2 system, as reported in the next section.

3 Schema of the proof

3.1 Proving that EAT objects are Simplicial Sets

The first task consists in proving that the face and degeneracy operators are well defined. Let $absm$ be an abstract simplex belonging to K^q; then the face and degeneracy EAT operators must satisfy: (i) $\eta_i^q absm \in K^{q+1}$; (ii) $\partial_i^q absm \in K^{q-1}$. As the definition of the degeneracy maps over abstract simplexes is independent from the simplicial set, so is the proof of its correctness. On the contrary, the face map invariance must be proven for each particular object.

Then, as a second task, the properties stated in Definition 1 must be accomplished by the face and degeneracy maps.

As no additional information from the particular object is needed, some important equalities can be obtained for every simplicial set:

Theorem 1 Let $absm$ be an abstract simplex. Then:

$$\begin{array}{lcccl}
\eta_i^{q+1}\eta_j^q absm & = & \eta_j^{q+1}\eta_{i-1}^q absm & \text{if} & i > j, \\
\partial_i^{q+1}\eta_j^q absm & = & \eta_{j-1}^{q-1}\partial_i^q absm & \text{if} & i < j, \\
\partial_i^{q+1}\eta_i^q absm & = & absm & = & \partial_{i+1}^{q+1}\eta_i^q absm, \\
\partial_i^{q+1}\eta_j^q absm & = & \eta_j^{q-1}\partial_{i-1}^q absm & \text{if} & i > j+1,
\end{array}$$

Then all properties of Definition 1 are proven without using a particular simplicial set except relation (1). On the contrary, we must require that the particular simplicial set satisfies some properties in order to obtain the proof of the first property of Definition 1. These required conditions have been characterized by means of the following result.

Theorem 2 Let $\mathcal{E}$ be an EAT object implementing a simplicial set. If for every natural number $q \geq 2$ and for every geometric simplex $gmsm$ in dimension q the following properties hold:

1. $\forall i, j \in \mathbb{N} : i < j \leq q \Longrightarrow \partial_i^{q-1}(\partial_j^q gmsm) = \partial_{j-1}^{q-1}(\partial_i^q gmsm)$,
2. $\forall i \in \mathbb{N}$, $i \leq q$: $\partial_i^q gmsm$ is a simplex of $\mathcal{E}$ in dimension $q-1$,

then:

$\mathcal{E}$ is a simplicial set.

This theorem has been proven in ACL2, by using in particular the `cmp-d-ls` EAT function. This function takes as arguments a natural number i and a degeneracy list (that is, a strictly decreasing list of natural numbers); the function has two outputs (compare with the discussion about the two cases in the formula 2.1):

- a new degeneracy list, obtained by systematically applying equations (3), (4) and (5) of Definition 1, starting with ∂_i, and
- an index which *survives* in the previous process, or the symbol *nil* in the case where the equation (4) in Definition 1 is applied and the face map is cancelled.

For instance, with the inputs 4 and (5 2 0) the results of `cmp-d-ls` are (4 2 0) and 2 (recall the process: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). With the inputs 4 and (5 3 0), the outputs are (4 0) and *nil* (since $\partial_4\eta_5\eta_3\eta_0 = \eta_4\partial_4\eta_3\eta_0 = \eta_4\eta_0$).

This function will play an important role when tackling the same problem in Kenzo. We will say that this function implements a *face operator* (over degeneracy operators).

Thanks to Theorem 2, certifying that an EAT object is a simplicial set can be reduced to prove some basic properties of the particular object, the rest of the proof can be generated automatically by ACL2, as will be detailed in Subsection 4.2. The same schema will be applied to the Kenzo model in the next subsection.

3.2 Proving that Kenzo objects are Simplicial Sets

In [10] the correctness of Kenzo degeneracy maps was proven. Thus, we must focus here on the correctness of face maps. The most important function to this aim is called in Kenzo `1dlop-dgop`, equivalent to the previously evoked EAT function `cmp-d-ls`. The arguments and outputs are, of course, equivalent in both functions, but recall that in Kenzo a degeneracy operator is encoded as a natural number.

The proof that the function `1dlop-dgop` is equivalent to `cmp-d-ls` is not simple, for two main reasons. On the one hand, the Kenzo function and the EAT one deal with different representations of degeneracy operators. On the other hand, the Kenzo function implements an algorithm which is not intuitive and is quite different from the algorithm of the EAT version, which is closely related to the mathematical definitions. A suitable strategy, used in [10] for the degeneracy operator, to attack the proof consists in considering an intermediary representation of degeneracy operators based on binary lists (that is, lists of bits), as explained at the second paragraph of Subsection 2.3.

The plan has consisted in defining a function `1dlop-dgop-binary` implementing the application of the face operator over a degeneracy list represented as a

binary list, by means of an algorithm directly inspired from that of Kenzo. Thus, the equivalence between EAT and Kenzo face maps has been proven in two steps. We have proven the equivalence between the Kenzo function `1dlop-dgop` and the binary function `1dlop-dgop-binary`. Subsequently, it has also been proven that the binary version function and the EAT function are equivalent.

Schematically, let $\mathcal{D}_g^L$ be the set of strictly decreasing lists of natural numbers, $\mathcal{D}_g^B$ be the set of binary lists and $\mathbb{N}$ the set of natural numbers. The proof consists in verifying the commutativity of the following diagram (in which the names of the transformation functions have been omitted):

$$\begin{array}{ccccc}
\mathbb{N} \times \mathcal{D}_g^L & \rightleftarrows & \mathbb{N} \times \mathcal{D}_g^B & \rightleftarrows & \mathbb{N} \times \mathbb{N} \\
\Big\downarrow{\scriptstyle cmp-d-ls} & & \Big\downarrow{\scriptstyle 1dlop-dgop-binary} & & \Big\downarrow{\scriptstyle 1dlop-dgop} \\
\mathcal{D}_g^L \times (\mathbb{N} \cup \{nil\}) & \rightleftarrows & \mathcal{D}_g^B \times (\mathbb{N} \cup \{nil\}) & \rightleftarrows & \mathbb{N} \times (\mathbb{N} \cup \{nil\})
\end{array}$$

These functions, that implement the face operators in the different representations, receive as input two arguments (a natural number representing the index of the face map and a degeneracy operator encoded in the respective representation) and have as output a pair composed of a degeneracy operator (in the same representation of the input one) and either a natural number or the value `nil` (as explained in detail in the case of `cmp-d-ls` in the previous subsection).

Thus the commutativity of the diagram ensures the equivalence modulo the change of representation between the EAT and Kenzo models.

More concretely, both the degeneracy operator correctness and the properties included in Theorem 2, can be translated to the Kenzo system thanks to the domain transformation theorems, producing a similar structural theorem in ACL2 for the Kenzo model, which accurately corresponds with the next statement.

Theorem 3 Let $\mathcal{K}$ be a Kenzo object implementing a simplicial set, satisfying for every natural number $q \geq 2$ and for every geometric simplex $gmsm$ in dimension q the following properties:

1. $\forall i, j \in \mathbb{N} : i < j \leq q \Longrightarrow \partial_i^{q-1}(\partial_j^q gmsm) = \partial_{j-1}^{q-1}(\partial_i^q gmsm)$,
2. $\forall i \in \mathbb{N}$, $i \leq q$: $\partial_i^q gmsm$ is a simplex of $\mathcal{K}$ in dimension $q-1$,

then:

$\mathcal{K}$ is a simplicial set.

4 ACL2 technical issues

In this section we deal with three technical issues in ACL2: (1) how to prove the correctness of face operators implemented in Kenzo, (2) the definition of a generic simplicial set theory in ACL2 which can be used to certify that the so-called simplicial sets of the Kenzo system are really... simplicial sets, and (3) the instantiation of this generic tool to concrete examples of families of simplicial sets actually programmed in Kenzo.

4.1 Correctness of face and degeneracy operators

Since the degeneracy operator was studied in [10], we are going to focus on the face operator.

The left side in Figure 1 contains the *real* Common Lisp code of Kenzo for the application of a face map over a degeneracy operator. That definition receives as inputs two naturals numbers `1dlop` and `dgop` (the second one to be interpreted as a degeneracy operator) and returns two values: a natural number (encoding a degeneracy operator) and an index (observe the occurrence of the `values` primitive). The algorithm takes advantage of the encoding of degeneracy operators as numbers, by using very efficient Common Lisp (and ACL2) primitives. For example, to know if `1dlop` is an active bit of `dgop` it uses the function `(logbitp 1dlop dgop)`. Or for computing the *xor* of two numbers it uses the `logxor` operator. Thanks to this way of programming, the function does not need an iterative processing, but just a conditional distinction of cases. It is to be compared with the corresponding EAT function `cmp-d-ls` whose linear time algorithm closely follows the natural mathematical iteration (recall once more the process: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). It is easy to understand the benefits of the Kenzo approach from the performance point of view. It should also be clear that the correctness of the Kenzo function is not evident (contrary to its EAT counterpart), and then an ACL2 verification is a highly valuable objective.

```
(defun 1dlop-dgop (1dlop dgop)
  (progn
    (when (logbitp 1dlop dgop)
      (let ((share (ash -1 1dlop)))
        (declare (fixnum share))
        (return-from 1dlop-dgop
          (values
           (logxor
            (logand share (ash dgop -1))
            (logandc1 share dgop))
           nil))))
    (when (and (plusp 1dlop)
               (logbitp (1- 1dlop) dgop))
      (let ((share (ash -1 1dlop)))
        (declare (fixnum share))
        (setf share (ash share -1))
        (return-from 1dlop-dgop
          (values
           (logxor
            (logand share (ash dgop -1))
            (logandc1 share dgop))
           nil))))
    (let ((share (ash -1 1dlop)))
      (declare (fixnum share))
      (let ((right (logandc1 share dgop)))
        (declare (fixnum right))
        (values
         (logxor
            right
            (logand share (ash dgop -1)))
         (- 1dlop (logcount right)))))))
```

```
(defun 1dlop-dgop-dgop (1dlop dgop)
  (if (and (natp 1dlop) (natp dgop))
      (cond ((logbitp 1dlop dgop)
             (logxor
              (logand (ash -1 1dlop)
                      (ash dgop -1))
              (logandc1 (ash -1 1dlop)
                        dgop)))
            ((and (plusp 1dlop)
                  (logbitp (- 1dlop 1) dgop))
             (logxor
              (logand (ash (ash -1 1dlop) -1)
                      (ash dgop -1))
              (logandc1 (ash (ash -1 1dlop) -1)
                        dgop)))
            (t (logxor
                (logandc1 (ash -1 1dlop) dgop)
                (logand (ash -1 1dlop)
                        (ash dgop -1)))))
    nil))

(defun 1dlop-dgop-indx (1dlop dgop)
  (if (or (logbitp 1dlop dgop)
          (and (plusp 1dlop)
               (logbitp (- 1dlop 1) dgop)))
      nil
    (- 1dlop
       (logcount (logandc1 (ash -1 1dlop) dgop)
                 ))))
```

Fig. 1. 1dlop-dgop definition in Kenzo and in ACL2

The right box of Figure 1 contains our ACL2 definition. The function `1dlop-dgop` is separated into two functions (a different, but equivalent, alternative consists in using the `mv` ACL2 macro which returns two or more values). However, this is the only important difference between the two sides of Figure 1, since all the binary operations of Common Lisp (`logxor` and the like) are present in ACL2. Thus, the two ACL2 programs make up an accurate version of the Kenzo one. But, of course, the challenge of proving the equivalence with the (ACL2 verified) EAT version remains. Following the guidelines given in [10], and with some effort, our methodology was used with success.

Once the correctness of the face and degeneracy maps over degeneracy operators has been proven, the task of certifying properties like Theorem 1 can be carried out. First proving them using the EAT formal specification and later on translating the properties to Kenzo by means of the domain transformation theorems. Figure 2 shows our ACL2 version of Kenzo definitions of the face and degeneracy maps for simplicial sets over abstract simplexes. The name of the functions (with the prefix `imp-`) is reminiscent of the *imp-construction* introduced in [8] to explain, in an algebraic specification framework, the way in which EAT manipulates its objects. Here it is used, at a syntactical level, to get a signature closer to mathematical Definition 1 (with three arguments: an index `n`, a dimension `q` and a simplex), and then to hide the irrelevant information to the operational functions (both the ambient simplicial set and the dimension are irrelevant for the degeneracy function, but only the dimension is irrelevant for the face map). More importantly, the imp-construction allows us to organize simplicial sets as indexed families (the parameter `ss` being the "index" of the space in the family), factoring out proofs for each element of the family (the families we are thinking of having already been mentioned: spheres, Moore spaces, simplicial complexes, and so on).

The `face-kenzo` function of Figure 2 uses the previously introduced functions `1dlop-dgop-dgop` and `1dlop-dgop-indx`, and the function `face` which contains the actual Kenzo definition for faces of geometric simplexes in a simplicial set belonging to a concrete family of spaces. The `degeneracy-kenzo` function uses `dgop-ext-int` to changing of domain from lists (of natural numbers) to natural numbers; then it applies the Kenzo function `dgop*dgop` to compute the composition of degeneracy lists (the ACL2 reification and verification of these Kenzo functions were dealt with in [10]). Let us remark that the ACL2 verified code is the Kenzo code with the sole transformation depicted in Figure 1.

Each function presented in Figure 2 has an equivalent for the EAT representation. Figure 3 uses them to state the theorem `eat-property-3`, which reflects accurately the equation (3) of Definition 1. That theorem must be proven in ACL2 from scratch, by using only the EAT models. Then the domain transformation theorems are applied, and the theorem `kenzo-property-3` of Figure 3 is obtained for free in ACL2. The same schema applies for the rest of properties in Definition 1.

```
(defun face-kenzo (ss d dgop gmsm)
  (if (1dlop-dgop-dgop d dgop)
      (list (1dlop-dgop-dgop d dgop) gmsm)
    (list (1dlop-dgop-dgop d dgop) (face ss (1dlop-dgop-indx d dgop) gmsm))))

(defun imp-face-kenzo (ss d q absm)
  (declare (ignore q)) (face-kenzo ss d (car absm) (cadr absm)))

(defun degeneracy-kenzo (d dgop gmsm)
  (list (dgop*dgop (dgop-ext-int (list d)) dgop) gmsm))

(defun imp-degeneracy-kenzo (ss d q absm)
  (declare (ignore ss q)) (degeneracy-kenzo d (car absm) (cadr absm)))
```

Fig. 2. ACL2 definition of Kenzo operators over abstract simplexes

```
(defthm eat-property-3
  (implies (and (< i j) (imp-inv-eat ss q absm)))
           (equal (imp-degeneracy-eat ss (- j 1) (- q 1) (imp-face-eat ss i q absm))
                  (imp-face-eat ss i (+ q 1) (imp-degeneracy-eat ss j q absm)))))

(defthm kenzo-property-3
  (implies (and (< i j) (natp j) (natp i) (imp-inv-kenzo ss q absm))
           (equal (imp-degeneracy-kenzo ss (- j 1) (- q 1) (imp-face-kenzo ss i q absm))
                  (imp-face-kenzo ss i (+ q 1) (imp-degeneracy-kenzo ss j q absm)))))
```

Fig. 3. A Kenzo property from an EAT property

4.2 A generic simplicial set theory

The strength of Theorem 3 relies on the few preconditions needed in order to prove that a Kenzo object is a simplicial set. It is worth providing an ACL2 tool such that, when a user proves the preconditions in ACL2, the system generates automatically the complete proof.

To this aim, a generic instantiation tool [9] has been used. This tool provides a way to develop a generic theory and to instantiate the definitions and theorems of the theory for different implementations, in our case different Kenzo objects.

We must specify our simplicial sets generic theory, by means of an ACL2 tool called `encapsulate`. The signatures of the encapsulated functions are as follows:

- `(face * * *)`: to compute the face of a geometric simplex,
- `(dimension *)`: to compute the dimension of a simplex,
- `(canonical *)`: to determine if an object is a simplex in canonical form,
- `(member-ss * *)`: to know if the second argument is a simplex of the first one, a simplicial set.

To finish our generic model we have to assume the properties of Figure 4 (these properties correspond with hypothesis of Theorem 3) and to prove them with respect to a *witness*, to ensure that the consistency of the logical world is kept.

```
(defthm faceoface
  (implies (and (natp i) (natp j) (< i j) (canonical gmsm))
           (equal (face ss i (face ss j gmsm)) (face ss (- j 1) (face ss i gmsm)))))

(defthm face-dimension
 (implies (and (canonical gmsm) (natp i) (< i (dimension gmsm)))
          (equal (dimension (face ss i gmsm)) (1- (dimension gmsm)))))

(defthm face-member
 (implies (and (canonical gmsm) (member-ss ss gmsm) (natp i) (< i (dimension gmsm)))
          (member-ss ss (face ss i gmsm))))

(defthm natp-dimension
  (implies (canonical gmsm)
           (natp (dimension gmsm))))
```

Fig. 4. Assumed axioms

Once the generic theory has been built, producing a book[2] (for further reference, let us name this book by `generic-kenzo-properties-imp`), if a user gives instances for the previous definitions and proves for them the theorems in Figure 4, ACL2 produces a certification ensuring that all the properties of a simplicial set hold, achieving a proof of Theorem 3 for these instances. Let us note the importance of the automatic generation of the proof by means of some data: from 4 definitions and 4 generic theorems the instantiation tool generates (and instantiates) 15 definitions and 375 theorems, in addition to 77 definitions and 601 theorems which are included from other books. In this way, the hard task of proving that a Kenzo object is a simplicial set from scratch can be relaxed to introduce 4 definitions and to prove 4 theorems.

4.3 Obtaining ACL2 correctness certifications for concrete Kenzo simplicial sets families

We have applied the previous infrastructure to two families of simplicial sets in Kenzo: spheres (indexed by a natural number n, with $n > 0$) and simplicial sets coming from finite simplicial complexes (here, each space in the family is determined by a list of maximal simplexes).

If our intention is simply to prove the correctness of finite spaces like the previous ones, one strategy could be to verify the Kenzo function which is used to generate this kind of simplicial sets, called `build-finite-ss`. Nevertheless, our aim is also to provide proofs for infinite dimensional spaces which are offered by Kenzo to the user to initiate computations. Examples are Eilenberg-MacLane spaces (see [11]) and the universal simplicial set, usually denoted by Δ (this

[2] *Book* refers in the ACL2 jargon to a file containing definitions and statements that have been certified as admissible by the system.

particular space already played an important role in the ACL2 proof of Proposition 1 in [1]). To reach this objective, our approach is more general, as it can be applied to *any* simplicial set, regardless of its dimension.

We now explain how the generic tool is instantiated in the particular case of spheres. Spheres are produced in Kenzo by invoking the function `sphere` over a positive natural number `n`. The constructed object contains, in particular, a slot with a λ-term computing the faces of each simplex in the given sphere.

Consider that we want to write an ACL2 book `ss-sphere.lisp` which generates a proof of the fact that spheres with this Kenzo implementation are simplicial sets, through the functions and properties of the generic theory. To this aim, we include the following: `(include-book "generic-kenzo-properties-imp")`. This event generates `definstance-*simplicial-set-kenzo*`, a macro which will be used to instantiate the events from the generic book. It is now needed to define the counterparts of the generic functions. For instance, the counterpart of the function `face` will be called `face-sphere`, and will be, essentially, the λ-term previously evoked. Later on, the statements presented in the previous subsection in Fig. 4 must be also proven. For instance, the following theorem must be proven.

```
(defthm faceoface-sphere
 (implies (and (natp i) (natp j) (< i j) (canonical-sphere gmsm))
          (equal (face-sphere n i (face-sphere n j gmsm))
                 (face-sphere n (+ -1 j) (face-sphere n i gmsm)))))
```

Finally we instantiate all the events from `*simplicial-set-kenzo*`, simply by this macro call:

```
(definstance-*simplicial-set-kenzo*
 ((face face-sphere) (canonical canonical-sphere)
  (dimension dimension-sphere) (member-ss member-ss-sphere))
 "-sphere")
```

At this moment, new instantiated definitions and theorems are available in the ACL2 logical world, proving that Kenzo spheres satisfy all the conditions of Definition 1.

5 Conclusions and future work

A framework to prove the correctness of simplicial sets as implemented in the Kenzo system has been presented. As examples of application we have given a complete correctness proof of the implementation in Kenzo of spheres and of simplicial sets coming from simplicial complexes (modulo a safe translation of Kenzo programs to ACL2 syntax) has been done. By means of the same generic theory the correctness of other Kenzo simplicial sets can be proved.

Some parts of the future work are quite natural. With the acquire experience, the presented methodology could be extrapolated to other algebraic Kenzo data structures. So, this work can be considered a solid step towards our objective of verifying in ACL2 first order fragments of the Kenzo computer algebra system. Nevertheless, it is not evident how ACL2 could be used to certify the correctness

of constructors which generate new spaces from another ones because, as was explained in the Introduction, higher-order functional programming is involved.

In a different line, we want to integrate ACL2 certification capabilities in our user interface *fKenzo* [6]. The idea is to interact in a same friendly front-end with Kenzo and ACL2. For instance, and closely related to the contributions presented in this paper, the user could give information to construct a new simplicial set with Kenzo. In addition, he could provide minimal clues to ACL2 explaining why his construction is sensible (technically, he should afford the system with the ACL2 hypothesis of Theorem 3); then ACL2 would produce a complete proof of the correctness of the construction. This kind of interaction between computer algebra and theorem provers would be very valuable, but severe difficulties related to finding common representation models are yet to be overcome.

References

1. M. Andrés, L. Lambán, J. Rubio, and J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. *Proceedings ACL2 Workshop 2007. University of Austin*, pages 34–39, 2007.
2. J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4):271–292, 2008.
3. J. Aransay, C. Ballarin, and J. Rubio. Generating certified code from formal proofs: a case study in homological algebra. *Formal Aspects of Computing*, 22(2):193–213, 2010.
4. X. Dousson, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo`.
5. J. Heras. ACL2 verification of Kenzo simplicial sets, 2010. `http://www.unirioja.es/cu/joheras/ss-tool.html`.
6. J. Heras, V. Pascual, and J. Rubio. Using Open Mathematical Documents to interface Computer Algebra and Proof Assistant systems. *Lecture Notes in Artificial Intelligence*, 5625:467–473, 2009.
7. M. Kaufmann and J. S. Moore. ACL2 Home Page. `http://www.cs.utexas.edu/users/moore/acl2/`.
8. L. Lambán, V. Pascual, and J. Rubio. An object-oriented interpretation of the EAT system. *Applicable Algebra in Engineering, Communication and Computing*, 14(3):187–215, 2003.
9. F. J. Martín-Mateos, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz-Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. *Proceedings of the Third ACL2 Workshop. University of Grenoble, France*, pages 188–203, 2002.
10. F. J. Martín-Mateos, J. Rubio, and J. L. Ruiz-Reina. ACL2 verification of simplicial degeneracy programs in the Kenzo system. *Lecture Notes in Computer Science*, 5625:106–121, 2009.
11. J. P. May. *Simplicial objects in Algebraic Topology*, volume 11 of *Van Nostrand Mathematical Studies*. 1967.
12. J. Rubio, F. Sergeraert, and Y. Siret. EAT: Symbolic Software for Effective Homology Computation. Institut Fourier, Grenoble, 1990. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/#Eat`.
13. F. Sergeraert. The computability problem in Algebraic Topology. *Advances in Mathematics*, 104:1–29, 1994.

Executable Specifications in an Object Oriented Formal Notation

Ángel Herranz and Julio Mariño

Universidad Politécnica de Madrid
{aherranz|jmarino}@fi.upm.es

Abstract. Early validation of requirements is crucial for the rigorous development of software. Without it, even the most formal of the methodologies will produce the wrong outcome. One successful approach, popularised by some of the so-called *lightweight formal methods*, consists in generating (finite, small) models of the specifications. Another possibility is to build a running prototype from those specifications. In this paper we show how to obtain executable prototypes from formal specifications written in an object oriented notation by translating them into logic programs. This has a number of advantages over other lightweight methodologies. For instance, we recover the possibility of dealing with recursive data types as specifications that use them often lack finite models.

1 Introduction

Lightweight formal methods [14, 9] have become relatively popular thanks to their success in early validation of requirements, a smooth learning curve and the availability of usable tools. This simplicity is obtained by replacing formal proof – which often demands human intervention – by model checking, but this also implies giving up correctness in favour of a less stringent criterion for models.

Consider, for example, the stepwise specification of queues in Alloy [15]. The specifier might start by just sketching the interface up, like in

```
module myQueue

sig Queue { root: Node }
sig Node  { next: Node }
```

that is, stating that *queues* must have a *root node* and nodes will have a *next* node to follow. The description can be "validated" by fixing a number of Queue and Node individuals and letting a tool like *Alloy Analyzer* [2] model check the specification and show graphically the different instances found. Of course, some of these instances will be inconsistent with the intuition in the specifier's mind – e.g. unreachable nodes or cyclic queues, that can be revealed with very small models. Further constraints, like

```
fact allNodesBelongToOneQueue {
  all n:Node | one q:Queue | n in q.root.*next }
fact nextNotCyclic {no n:Node | n in n.^next}
```

can be added to the myQueue module in order to supply some of the pieces missing in the original requirements. The first *fact* states that for every node there must be some queue such that the node lies somewhere in the transitive-reflexive closure of the next relation starting with the root of that node. The second one says that no node can be in the transitive closure of the next relation starting in itself. Model checking the refined specification will generate less instances, thus allowing to explore bigger ones, which will hopefully lead to reveal more subtle corners in the requirements.

As said before, this approach is extremely attractive: requirements are refined in a stepwise manner guided by counterexamples found by means of model checking, and the whole process is performed with the help of graphical tools.

However, there are also some limitations inherent to this approach. Leaving aside the fact that total correctness of the specification is abandoned in favour of a more relaxed notion of being *not yet falsified by a counterexample*, which can make the whole enterprise unsuitable for safety critical domains, the use of model checking rather than proof based techniques also brings other negative consequences, such as limiting the choice of data types in order to keep models finite, making extremely difficult to model and reason over recursive data types like naturals, lists, trees, etc. (See [15], Ch. 4, Sec. 8.)

A natural alternative to model checking the initial requirements is to produce an executable prototype from them. Using the right language it is possible to obtain recursive code and validation can be guided by *testing*, which might also be automated by tools such as *QuickCheck* [6].

Regarding how to obtain the prototypes, there are several possibilities. One of them is to follow the *correct by construction* slogan and to produce code from the specification, either by means of a transformational approach that often requires human intervention, or by casting the original problem in some *constructive type theory* that will lead directly to an implementation in a calculus thanks to the Curry-Howard isomorphism [23, 5, 4, 22].

Another possibility is to use logic programming. In this case, executable specifications are obtained *free of charge*, as resolution or narrowing will deal with the existential variables involved in any implicit (i.e. non-constructive) specification. Readers familiar with logic programming will remember the typical examples – obtaining subtraction from addition for free, sorting algorithms from sorting test, etc. – and those familiar with logic program transformation techniques will also recognise that these can be used to turn those naive implementations into decent prototypes.

However, when it comes to practical usage, none of these formalisms can compete with the lightweight methods above, due to the great distances separating them from the notations used for modelling object oriented software.

This paper studies the synthesis of logic programs from specifications written in an object oriented notation. The specification language, Clay, is being designed around two driving ideas. First, the language must be small but make room for the basic constructs in object oriented programming. Second, specifications must

admit at least one canonical translation into executable prototypes to a allow the specifier to interactively validate her own specifications.

We contribute, on one hand, a *static* theory of types and inheritance that somehow copes with bridging the aforementioned gap between object orientation and logic programming, and, on the other, a *dynamic* part that deals with search in the presence of equality and inheritance (Section 3). To support our contributions we give a formalisation of the synthesis of logic programs from Clay specifications (Section 4) and some examples of the prototypes generated in action (Section 5).

2 Object Oriented Specifications in Clay

Clay is a lightweight evolution of SLAM-SL [10], a large object oriented notation designed to bridge the gap between formal methods and more widespread software engineering processes ([13, 12, 11]). Clay is a *desugared* version of SLAM-SL where the authors focus on formal aspects superficially treated in the cited works.

Clay is a stateless object oriented formal notation, a class-based language with a nominal type system. Classes are defined as algebraic types in the form of case classes: complete and disjoint subclasses of the defining class. Classes can be extended by subclassing. Methods are specified with pre and postconditions, first order formulae involving self (the recipient), parameters and result (the resulting object). Atomic formulae are equalities ($=$) and class membership ($:$).

An *interlingua* [3] declarative semantics for Clay is provided by translation into first-order logic. Clay tools generate an axiomatisation in Prover9/Mace4 [24] syntax. Then, early detection of inconsistencies is achieved by the combination of automatic theorem proving (Prover9) and model checking (Mace4) of the first order logic theories that reflects the structure of Clay specifications.

For the purposes of this paper, the move to logic program synthesis requires, on the *front-end* of the tools, to take some simplifying decisions in order to keep the resulting theory tractable and readable: no multiple inheritance, overloading not allowed (just method refinement) and no parametric polymorphism.

Figures 1 and 2 contain three examples of Clay specifications that will guide the whole paper. In this section, the examples will help us to present some of the most important and relevant aspects of our notation.

2.1 Modelling Data

Let us start with the specification of natural numbers (Figure 1). Instances of a class are the disjoint and complete sum of the instances of its case classes (indicated with keyword **state** due to their similarity to the design pattern *State* [8]): if n is an instance of Nat ($n : \mathsf{Nat}$) then it is an instance of Zero or, exclusively, of Succ. The following Clay formula expresses it formally:

$$\forall\ \mathsf{n} : \mathsf{Nat}\ ((\mathsf{n} : \mathsf{Zero} \vee \mathsf{n} : \mathsf{Succ}) \wedge \mathsf{n} : \mathsf{Zero} \Leftrightarrow \neg\, \mathsf{n} : \mathsf{Succ})$$

```
class Nat <: Num {
  state Zero {}
  state Succ {pred : Nat}
  modifier add (n : Nat) {
    post {     self : Zero ⇒ result = n
          ∧ self : Succ ⇒ self←pred←add(n←inc) = result }
  }
  observer even : Bool {
    post { result : True ⇔ exists n : Nat ( self = n←add(n)) }
    sol {     self : Zero ∧ result : True
         ∨ self : Succ ∧ result = self←pred←even←neg }
  }
  observer half : Nat {
    pre { self←even = Bool←mkTrue }
    post { self = result←add(result) }
  }
}
```

Fig. 1. Natural numbers in Clay

The case classes Zero and Succ introduce the constructor methods mkZero and mkSucc. Both are messages that can be sent to the object Nat (classes are objects in Clay): Nat←mkZero[1] or Nat←mkSucc(n), being n an instance of Nat. Let us use 0, 1, 2, etc. to abbreviate Nat←mkZero, Nat←mkSucc(Nat←mkZero), Nat←mkSucc(Nat←mkSucc(Nat←mkZero)), etc.

Composition. Composition is represented by fields defined in a case class. Those fields are methods that project the encapsulated information of its case. In our example the result of the expression Nat←mkSucc(n)←pred is n.

Inheritance. Classes can be extended with subclasses that inherit all the properties of the superclass. On the left of the Figure 2, class RGBNat extends Nat and therefore its instances obey the aforementioned property:

$$\forall\ \mathsf{n} : \mathsf{RGBNat}\ ((\mathsf{n} : \mathsf{Zero} \vee \mathsf{n} : \mathsf{Succ}) \wedge \mathsf{n} : \mathsf{Zero} \Leftrightarrow \neg\ \mathsf{n} : \mathsf{Succ})$$

Inheritance induces a subtype relation (<:) with all its expected rules: reflexivity, transitivity and subsumption. The most important aspect of the subtyping relation is that subclasses cannot invalidate by overriding any property specified in a superclass, otherwise the whole specification is considered inconsistent.

We consider that this approach is essential when we are specifying in the large: the specifier needs to reason locally to a class and a subclass cannot show a behaviour that forces the specifier to take into account all the subclasses. The approach adds another advantage: specifications can be much more concise since it is not needed to state already stated properties in superclasses. The main drawback is certain loss of flexibility but, on our view, the decision pays back.

[1] In Clay ← is used to indicate method invocation instead of the dot notation used in C++ and other widespread OO programming languages.

```
class RGBNat <: Nat {
  state Red {}
  state Green {}
  state Blue {}

  constructor mkRZ {
    post {     result : Zero
           ∧   result : Rojo }
  }
}

class CMYNat <: Nat {
  state Cyan {}
  state Magenta {}
  state Yellow {}

  constructor mkCZ {
    post {     result : Zero
           ∧   result : Cyan }
}
```

```
class Cell {
  state CellCase { contents : Nat }
  observer get : Nat {
    post { result = self←contents }
  }
  modifier set (v : Nat) {
    post { result←contents = v }
  }
}

class ReCell <: Cell {
  state ReCellCase { backup : Nat }
  modifier set (v : Nat) {
    post { result←backup = self←contents }
  }
  modifier restore {
    post {     result←contents = self←backup
           ∧   result←backup = self←backup
    }
  }
}
```

Fig. 2. Inheritance in Clay

2.2 Modelling Behaviour

Methods are specified with first order formulae that relate the receiver of the message (self) and the message's parameters with the answer to the message (result). Atomic formulae are equalities and class membership.

Class membership is mainly used to do pattern matching. In the specification of method add we can see how antecedents of implications distinguish the cases zero and non-zero (self : Zero and self : Succ).

Equality is particularly interesting in Clay. The predicate is implicitly indexed by the minimum subtype of the compared instances in the context in which the formula appears. In the example of add, the minimum type of result and self in the consequent of the first implication is Nat. The semantics establishes that no properties of self other than those *reachable* from Nat are enforced in result .

Let us illustrate it adding a *red* zero to a zero (0←add(RGBNat←mkRZ)). Add establishes that the result is equal, up to Nat, to RGBNat←mkRZ, so the property 0←add(RGBNat←mkRZ) : Zero holds (by the postcondition of mkRZ) and 0←add(RGBNat←mkRZ) : Red is not even a valid expression. Nevertheless, we can write the formula cn = 0←add(RGBNat←mkRZ) on a coloured natural cn : RGBNat since the equality can be indexed with Nat, the minimum common subtype of cn and 0←add(RGBNat←mkRZ).

Keywords **modifier** and **observer** are merely type informative (the result of a modifier is an instance of the class being specified) and has no influence in the semantics of the methods.

The specifier can feed the tools with extra executable specifications. The specification of even includes two postconditions (**post** and **sol**). The postcondition can be considered too hard to be used in the prototype and the specifier added a *solution* with the recursive decision procedure. In contrast, observer half is specified by the very natural property $2\times n\leftarrow\mathsf{half} = n$.

Cell and ReCell examples on the right of Figure 2 are brought from [1]. Instances of Cell are storage-cell objects encapsulating a natural number that can be changed (set) and read (get). The extension of Cell with a restore option yields ReCell. We can observe the conciseness of the overriding of set in ReCell since properties of Cell are inherited.

2.3 Interacting with Clay

In Section 5 we will check the actual results of our prototype to some examples that we present in the following lines:

- Specification of add is an implicit one. Will our prototype be able to compute $42\leftarrow\mathsf{half}$? What should be the result for $27\leftarrow\mathsf{half}$, where precondition does not hold?
- The Clay equality is particularly curious. What would be the answers to the following Clay atomic formulae?
  - $\mathsf{Nat}\leftarrow\mathsf{mkZero} = \mathsf{CMYNat}\leftarrow\mathsf{mkRZ}$
  - $\mathsf{Nat}\leftarrow\mathsf{mkCZ} = \mathsf{CMYNat}\leftarrow\mathsf{mkRZ}$
  - $\mathsf{Nat}\leftarrow\mathsf{mkZero}\leftarrow\mathsf{inc} = \mathsf{CMYNat}\leftarrow\mathsf{mkRZ}\leftarrow\mathsf{inc}$
- Finally, to check that we are enabling the specifier to write concise specifications, what will be the answers to expressions like
  $\mathsf{ReCell}\leftarrow\mathsf{mkReCell}\leftarrow\mathsf{set}(0)\leftarrow\mathsf{set}(1)\leftarrow\mathsf{get}$ and
  $\mathsf{ReCell}\leftarrow\mathsf{mkReCell}\leftarrow\mathsf{set}(0)\leftarrow\mathsf{set}(1)\leftarrow\mathsf{restore}\leftarrow\mathsf{get}$?

3 Translating Clay Specifications into Logic Programs

Given a Clay specification we will synthesise facts that represent its abstract syntax tree: classes, inheritance, case classes, fields, and method's pre and postconditions. In Section 4 we give a formalisation of the translation, for the moment Figure 3 contains the informal meaning of the target predicates.

The heart of our prototype is a common theory for all specifications: *the Clay theory*. The most important predicates of this axiomatisation are (`instanceof/2`, `reduce/2`, and `eq/3`), definitions that rely on the facts translated from the source specifications. Their meaning:

- Predicate `instanceof(`NF`,`A`)` is a generator of instances NF of a class A. NF is a normal form of an instance of A: a flexible representation as an incomplete data structure that we will study in Section 3.1.

`class(`C`)`	C is a Clay's class identifier.
`inherits(`A`,[`B`])`	B is the superclass of A
`cases(`C`,`Cs`)`	Cs is the list with cases classes of class C
`fields(`C`,`Fs`)`	Fs is the association list with the field names and field types of the case class C
`msgtype(`C`,`M`)`	M is a message identifier of a method defined or overridden in class C
`pre(`C`,`S`,`M`,`As`)`	Precondition of sending message M with arguments As to instance S in class C
`post(`C`,`S`,`M`,`As`,`R`)`	Postcondition that establishes that R is the resulting instance of sending message M with arguments As to instance S in class C

Fig. 3. Representing Clay in Prolog

- Predicate `eq(`A`,`NF_1`,`NF_2`)`, the Clay's equality, decides if the representation of two instances are indistinguishable in class A.
- Finally, predicate `reduce(`E`,`NF`)` *reduces* any Clay object expression E to its normal form. Predicates `eq` and `reduce` will be study in Section 3.3.

3.1 Representing Clay Instances in Prolog

How will we represent the natural number 1 (an instance of Nat)? We need to capture all the information of known superclasses (Num) and to capture all the information about the specific case class (Nat). We can use a list where each element contains the part of the representation for a given class of the instance: `(`C`,`S`,`F`)` where C is the class, S is the particular case class, and F is an association list of field names to the representation of the instances of their classes. Let us show the representation of number 1:

```
[('Num','Num',[]), ('Nat','Succ',[(pred,[('Num','Num',[]),
                                         ('Nat','Zero',[])])])]
```

Under subtyping, during a deduction process where 1 is expected the successor of a *red 0* could appear. If we follow our rules, the representation of the coloured number would be:

```
[('Num','Num',[]), ('Nat','Succ',[(pred,[('Num','Num',[]),
                                         ('Nat','Zero',[]),
                                         ('RGBNat','Red',[])])])]
```

The representation of 0 and red 0 are partially the same:

```
[('Num','Num',[]),('Nat','Zero',[])]
[('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])]
```

We propose to *make room* for not yet known information of subclasses. Our proposal is to use an incomplete data structure where the incomplete part represents the *room* for the information of potential subclasses of `Nat`:

```
[('Num','Num',[]),('Nat','Zero',[]),_Room]
[('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[]),_Room]
```

Apart from carrying all the information needed by methods specified in the superclasses, our normal form has the following properties:

- Information of the case classes allows us to reflect the disjoint sum (case classes) of products (fields).
- The incomplete part might be instantiated with data of an instance of a subclass (like RGBNat) during the deduction process. The most interesting benefit is that the instantiation can be implemented with the unification of our logic language engine. The example above shows how a red 0 *fits* in a 0.

3.2 Instance of

Predicate ":" (instance of) is translated into the Prolog predicate `instanceof/2`. `Instanceof/2` generates the representation of *all* instances (first argument) of all classes (second argument) of a specification. Let us see some outputs of this predicate:

```
?- instanceof(O,C).
C = 'Num', O = [('Num','Num',[])|_] ? ;
C = 'Bool', O = [('Bool','True',[])|_] ? ;
...
C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])|_] ? ;
C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero,[]),('RGBNat','Green',[])|_]␣?
```

Thanks to our incomplete structures every instance of a subclass is an instance of a superclass, a technique that makes the desirable property of subsumption to be a theorem in our Prolog axiomatisation.

3.3 Equality

Clay equality (=) is the other predicate used in the atomic formulae of Clay in this work. Our translation of Clay equality into Prolog consists of two steps: a reduction to normal form of the objects expressions and the unification of the obtained representations. Let us see a description of the implementation of the reduction step and postpone the formalisation of the translation of the equality literals to section 4. `Reduce/2` relates terms that represent abstract syntax trees of Clay expressions with its normal form (assume `:- op(200,yfx,'<--').` declared to emulate the Clay syntax):

```
reduce(O<--M,NF) :- M =.. [Mid|Args],
                    reduce(O,ONF), reduceall(Args,ArgsNF),
                    knownclasses(ONF,Cs),
                    checkpreposts(Cs,ONF,Mid,ArgsNF,NF,defined).
```

`Reduceall/2` reduces a list of expressions, second argument of `knownclasses/2` contains the known classes (`Cs`) of the recipient of the message and `checkpreposts` checks pre and postconditions of every class of `Cs` in which method `Mid` is defined.

Safe Inheritance. We already mentioned in Section 2 the danger of overriding the properties of methods in subclasses: the practical impossibility of reasoning in large programs. The above implementation of predicate `reduce/2` will fail if any postcondition in the inheritance hierarchy is inconsistent with the postconditions specified in superclasses.

4 The Translation, Formalised

In this section we show an abstract representation of specifications in Clay. A specification (*Spec*) is a partial function from class identifiers (*CI*) to class specifications (*CS*). A class specification is a triple with the identifier of the superclass, a state environment (*SE*) and a method environment (*ME*). A state environment is a partial function from class identifiers (the case classes) to field environments (*FE*). A field environment is a partial function from method identifiers (*MI*) to class identifiers. A method environment is a partial function from method identifiers (*MI*) to method specifications (*MS*). A method specification is a triple with a method declaration (*MD*), and two formulae (*Form*) that represent the precondition and the postcondition. A method declaration is a partial function from object variables (*OV*) to class identifiers.

Figure 4 contains the mathematical definition of the abstract syntax. We use the following conventions in our metalanguage: A and B for class identifiers (*CI*), e, a and b for object expressions (*Expr*), x for object variables (*OV*), m for method identifiers (*MI*), and F and G for formulae (*Form*).

Formulae (F, G)	$Form ::= e_1 = e_2 \mid e : A$
	$\mid \neg F \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid F \Leftrightarrow G$
	$\mid \forall\, x : A(F) \mid \exists\, x : A(F)$
Expressions (a, b, e)	$Expr ::= A \mid x \mid e \leftarrow m(e_1, e_2, \ldots e_n)$
Specifications	$Spec = CI \nrightarrow CS$
Class Specifications	$CS = CI \times SE \times ME$
State Environments	$SE = CI \nrightarrow FE$
Field Environments	$FE = MI \nrightarrow CI$
Method Environments	$ME = MI \nrightarrow MS$
Method Specifications	$MS = MD \times Form \times Form$
Method Declarations	$MD = OV \nrightarrow CI$

Fig. 4. Clay's abstract syntax

Figure 6 shows the translation of Clay specifications, into Prolog programs (*Prog*), although extended programs (*EG*, *EC*, *EP*) allowing general first-order constructs in goals are used as an intermediate format and then translated into

$$
\begin{array}{lrcl}
\textbf{Terms}\,(s,t) & \mathit{Term} & ::= & x \mid f(t_1,\ldots,t_n) \\
\textbf{Atoms}\,(A,B) & \mathit{Atom} & ::= & p(t_1,\ldots,t_n) \\
\textbf{Literals}\,(L) & \mathit{Lit} & ::= & A \mid \neg A \\
\textbf{Goals}\,(G) & \mathit{Goal} & = & \mathit{Lit}^* \\
\textbf{Horn Clauses}\,(C) & \mathit{Clause} & = & \mathit{Atom} \times \mathit{Goal} \\
\textbf{Programs}\,(P,Q) & \mathit{Prog} & = & \mathbb{P}\,\mathit{Clause} \\
\textbf{Extended Goals}(F) & \mathit{EG} & ::= & \mathit{Lit} \mid F \wedge F' \mid F \vee F' \mid F \Rightarrow F' \mid F \Leftrightarrow F' \\
 & & & \mid \forall\, x : A(F) \mid \exists\, x : A(F) \\
\textbf{Ext. Clauses}(K) & \mathit{EC} & = & \mathit{Atom} \times \mathit{EG} \\
\textbf{Ext. Programs} & \mathit{EP} & = & \mathbb{P}\,\mathit{EC}
\end{array}
$$

Fig. 5. Prolog's abstract syntax

standard logic programs via a Lloyd-Topor like transform ([20, 19]). A concise abstract syntax of logic programs is given in Figure 5. The part of translation dealing with formulae and expressions is shown in Figure 7. Due to space limitations, the definitions of several auxiliary functions are given informally.

5 Experimental Results

Let us show some details of the code generated for the examples in our test set.

Recursive definitions. Our implementation of the Lloyd-Topor transformation applied to the postcondition of `add` produces four clauses:

```
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   \+ instanceof(_self, 'Zero'),
   \+ instanceof(_self, 'Succ').
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   \+ instanceof(_self, 'Zero'),
   reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
   eq('Nat', _result, _NF_Nat_mkSucc).
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   eq('Nat', _result, _n),
   \+ instanceof(_self, 'Succ').
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   eq('Nat', _result, _n),
   reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
   eq('Nat', _result, _NF_Nat_mkSucc).
```

Since not enough intelligence has been imprinted, the first and the last clause are, respectively, the result of falsifying the antecedents of implications and checking the consequents. None of those clauses yield any new result in our case: `self` cannot be and instance of `Zero` and an instance of `Succ` and the last

$\mathsf{tr\_spec} : \mathit{Spec} \to \mathit{Prog}$
$\mathsf{tr\_spec}[\![S]\!] = \mathsf{Clay\_Theory} \cup \bigcup_{A \in \mathrm{dom}\, S} \mathsf{lloyd\_topor} \circ \mathsf{tr\_class}[\![A, S(A)]\!]$

$\mathsf{lloyd\_topor} : \mathit{EP} \to \mathit{Prog}$
$\mathsf{lloyd\_topor}[\![\mathit{ep}]\!] =$ The Lloyd-Topor Transformation ([20, 19])

$\mathsf{tr\_class} : \mathit{CI} \times \mathit{CS} \to \mathit{EP}$
$\mathsf{tr\_class}[\![A, (\mathit{super}, \mathit{states}, \mathit{methods})]\!] = \{(\mathtt{class}(\mathsf{tr\_exp}[\![A]\!]), \top)$
$,(\mathtt{instanceof}(\mathsf{tr\_exp}[\![A]\!], \mathsf{tr\_meta}[\![A]\!]), \top)$
$,(\mathtt{inherits}(\mathsf{tr\_exp}[\![A]\!], \mathsf{tr\_exp}[\![\mathit{super}]\!]), \top)\}$
$\bigcup \mathsf{tr\_se}[\![A, \mathit{states}]\!] \bigcup \mathsf{tr\_me}[\![A, \mathit{methods}]\!]$

$\mathsf{tr\_se} : \mathit{CI} \times \mathit{SE} \to \mathit{EP}$
$\mathsf{tr\_se}[\![A, \mathit{states}]\!] = \{(\mathtt{cases}(\mathsf{tr\_exp}[\![A]\!], \mathsf{tr\_set}[\![\mathrm{dom}\, \mathit{states}]\!]), \top)\}$
$\bigcup_{A \in \mathrm{dom}\, \mathit{states}} (\mathsf{tr\_stt}[\![A, B, \mathit{states}(B)]\!] \cup \mathsf{tr\_fe}[\![A, B, \mathit{states}(B)]\!])$

$\mathsf{tr\_stt} : \mathit{CI} \times \mathit{CI} \times \mathit{FE} \to \mathit{EP}$
$\mathsf{tr\_stt}[\![A, B, \mathit{fields}]\!] = \{(\mathtt{msgtype}(\mathsf{tr\_exp}[\![A]\!], \mathtt{mk}{+}\mathsf{tr\_exp}[\![B]\!]), \top)$
$, (\mathtt{pre}(\mathsf{tr\_meta}[\![A]\!], \mathtt{\_self}, \mathtt{mk}{+}\mathsf{tr\_exp}[\![B]\!], \mathsf{tr\_fv}[\![\mathit{fields}]\!]),$
$\mathtt{instanceof}(\mathtt{\_self}, \mathsf{tr\_meta}[\![A]\!])$
$\wedge\ \mathsf{tr\_fety}[\![\mathit{fields}]\!])$
$, (\mathtt{post}(\mathsf{tr\_meta}[\![A]\!], \mathtt{\_self}, \mathtt{mk}{+}\mathsf{tr\_exp}[\![B]\!], \mathsf{tr\_fv}[\![\mathit{fields}]\!], \mathtt{\_result}),$
$\mathtt{instanceof}(\mathtt{\_result}, \mathsf{tr\_meta}[\![B]\!])$
$\wedge\ \mathtt{project}(\mathtt{\_result}, \mathsf{tr\_exp}[\![B]\!], \mathsf{tr\_fenv}[\![\mathit{fields}]\!]))$

$\mathsf{tr\_fe} : \mathit{CI} \times \mathit{CI} \times \mathit{FE} \to \mathit{EP}$
$\mathsf{tr\_fe}[\![A, B, \mathit{fields}]\!] = \bigcup_{m \in \mathrm{dom}\, \mathit{fields}} \mathsf{tr\_field}[\![A, B, \mathit{fields}(m), i, m]\!]$

$\mathsf{tr\_field} : \mathit{CI} \times \mathit{CI} \times \mathit{CI} \times \mathit{MI} \times \mathbb{N} \to \mathit{EP}$
$\mathsf{tr\_field}[\![A, B, C, i, \mathit{fn}]\!] = \{(\mathtt{msgtype}(\mathsf{tr\_exp}[\![A]\!], \mathsf{tr\_mi}[\![\mathit{fn}]\!]), \top)$
$, (\mathtt{pre}(\mathsf{tr\_meta}[\![A]\!], \mathtt{\_self}, \mathsf{tr\_mi}[\![\mathit{fn}]\!], \mathtt{[]}),$
$\mathtt{instanceof}(\mathtt{\_self}, \mathsf{tr\_meta}[\![B]\!]))$
$, (\mathtt{post}(\mathsf{tr\_meta}[\![A]\!], \mathtt{\_self}, \mathsf{tr\_mi}[\![\mathit{fn}]\!], \mathtt{[]}, \mathtt{\_result}),$
$\mathtt{instanceof}(\mathtt{\_result}, \mathsf{tr\_meta}[\![C]\!])$
$\wedge\ \mathtt{project}(\mathtt{\_self}, \mathsf{tr\_exp}[\![B]\!], \mathsf{tr\_anon}[\![i, \mathit{fn}, C]\!]))$

$\mathsf{tr\_me} : \mathit{CI} \times \mathit{ME} \to \mathit{EP}$
$\mathsf{tr\_me}[\![A, B, \mathit{methods}]\!] = \bigcup_{m \in \mathrm{dom}\, \mathit{methods}} \mathsf{tr\_ms}[\![A, \mathit{methods}(m)]\!]$

$\mathsf{tr\_ms} : \mathit{CI} \times \mathit{MI} \times \mathit{MS} \to \mathit{EP}$
$\mathsf{tr\_ms}[\![A, m, (\mathit{md}, \mathit{pre}, \mathit{post})]\!] = \mathsf{tr\_md}[\![A, m, \mathit{md}]\!] \cup$
$\{(\mathtt{pre}(\mathsf{tr\_exp}[\![A]\!], \mathtt{\_self}, \mathsf{tr\_mi}[\![m]\!], \mathsf{tr\_args}[\![\mathit{md}]\!]),$
$\mathsf{tr\_form}[\![\mathit{pre}]\!])$
$, (\mathtt{post}(\mathsf{tr\_exp}[\![A]\!], \mathtt{\_self}, \mathsf{tr\_mi}[\![m]\!], \mathsf{tr\_args}[\![\mathit{md}]\!], \mathtt{\_result}),$
$\mathsf{tr\_form}[\![\mathit{post}]\!])$

$\mathsf{tr\_md} : \mathit{CI} \times \mathit{MD} \to \mathit{EP}$
$\mathsf{tr\_md}[\![A, m]\!] = \{(\mathtt{msgtype}(\mathsf{tr\_exp}[\![A]\!], \mathsf{tr\_mi}[\![m]\!]), \top)\}$

$\mathsf{tr\_meta}$ generates the class a given class A is instance of: $\mathtt{meta}A$
$\mathsf{tr\_set}$ generates a Prolog list with valid Prolog terms that represent the set of class identifiers
$\mathsf{tr\_mi}$ generates a valid Prolog term to represent a message identifier

Fig. 6. Translating Clay specifications

$\mathsf{tr\_form} : \mathit{Form} \to \mathit{EG}$

$\mathsf{tr\_form}[\![e_1 = e_2]\!]$ = `reduce(tr_exp`$[\![e_1]\!]$`, _NF1)` $\wedge$ `reduce(tr_exp`$[\![e_2]\!]$`, _NF2)` $\wedge$ `eq(tr_exp`$[\![A]\!]$`, _NF1, _NF2)`
where A is the minimum common type of e_1 and e_2
and `_NF1` and `_NF2` are new variables

$\mathsf{tr\_form}[\![e : A]\!]$ = `reduce(tr_exp`$[\![e]\!]$`, _NF)` $\wedge$ `instanceof(_NF, tr_exp`$[\![A]\!]$`)`
where `_NF` is a new variable

$\mathsf{tr\_form}[\![\neg F]\!]$ = `negate(tr_form`$[\![F]\!]$`)`

$\mathsf{tr\_form}[\![F * G]\!]$ = $\mathsf{tr\_form}[\![F]\!] * \mathsf{tr\_form}[\![G]\!]$
where $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

$\mathsf{tr\_form}[\![\forall\, x : A(F)]\!] = \forall\, \mathsf{tr\_exp}[\![x]\!](\mathsf{tr\_form}[\![x : A]\!] \Rightarrow \mathsf{tr\_form}[\![F]\!])$

$\mathsf{tr\_form}[\![\exists\, x : A(F)]\!] = \exists\, \mathsf{tr\_exp}[\![x]\!](\mathsf{tr\_form}[\![x : A]\!] \wedge \mathsf{tr\_form}[\![F]\!])$

$\mathsf{tr\_exp} : \mathit{Expr} \to \mathit{Term}$

$\mathsf{tr\_exp}[\![A]\!] = \mathsf{mk\_const}[\![A]\!]$

$\mathsf{tr\_exp}[\![x]\!] = \mathsf{mk\_var}[\![x]\!]$

$\mathsf{tr\_exp}[\![e \leftarrow m(e_1, \ldots, e_n)]\!] = \mathsf{send}(\mathsf{tr\_exp}[\![e]\!], \mathsf{tr\_mi}[\![m]\!](\mathsf{tr\_exp}[\![e_1]\!], \ldots, \mathsf{tr\_exp}[\![e_n]\!])$

tr_fv generates a list with a variable per field
tr_fety generates `instanceof` calls given a list of vars and a list of classes
tr_fenv generates a partial function from field names to vars
tr_args generates a list of vars per arg in a message

Fig. 7. Translating Clay formulae and expressions

clause is establishing that result = n and result = self + n forcing self to be 0 and repeating the results of third clause. The result of some messages[2]:

```
?- reduce('Nat'<--mkZero,_Zero), reduce('Nat'<--mkSucc(_Zero),_One),
   reduce(_Zero<--add(_One),_One), reduce(_One<--add(_Zero),_One),
   reduce('Nat'<--mkSucc(_One),_Two), reduce(_Two<--add(Two),_Four).
Zero = Zero{},
One = Succ{pred : Zero{}},
Two = Succ{pred : Succ{pred : Zero{}}}
Four = Succ{pred : Succ{pred : Succ{pred : Succ{pred : Zero{}}}}}?
```

Our next example is the translation of half. For the moment we can see that the synthesised code is structurally the same that the specified in Clay:

```
post('Nat', _self, half, [], _result) :-
   instanceof(_result, 'Nat'),
   reduce(_result<--add(_result), _NF__result_add),
   eq('Nat', _self, _NF__result_add).
```

Will our prototype find the resulting instance of `42←half`? And, what about `27←half`?

```
?- num2nat(42,_E42), num2nat(21,_E21),
   reduce(_E42<--half,NF21), reduce(_E21,NF21).
NF21 = [('Num','Num',[]),('Nat','Succ',[(pred,[...])])] ?
```

[2] Prolog terms that represent normal forms are presented in a more readable format.

```
yes
?- num2nat(27,_E27), reduce(_E27<--half,_).
no
```

`Yes`[3], is the answer to 42⟵half with variable `NF21` representing the normal form of our natural number 21. Since precondition of 27⟵half (27⟵even : True) does not hold our prototype's answer is `no`.

Equality. Let us show the answers to queries about the equality of several coloured and non-coloured naturals:

```
?- reduce('RGBNat'<--mkZeroRed,ZR),
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',ZR,ZC).
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_],
ZR = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])|_] ?
?- reduce('Nat'<--mkZero,Z)
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',Z,ZC).
Z = [('Num','Num',[]),('Nat','Zero',[])|_],
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_] ?
```

Overriding. We show now the effects of the safe inheritance:

```
?- reduce('Cell'<--mkCellCase(0),R).
R = 'CellCase'{contents : 'Zero'{}}
?- reduce('Cell'<--mkCellCase(0)<--set('Nat'<--mkSucc(0)),R).
R = 'CellCase'{contents : 'Succ'{pred : 'Zero'{}}}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}}
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}}
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--restore<--get,R).
R = 'Zero'{}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--restore<--get,R).
no
```

Performance. We finish the presentation of our results with a pair of performance figures. Our experiments have been produced in a Ubuntu box running GNU/Linux 2.6.31-21 SMP on a machine with an Intel Dual Core CPU T7200@2.00GHz, 4096KB of cache and 2 GB of RAM. Our Prolog engine is Ciao 1.13.0-11293. In the following table we show the performance of our prototypes. Column *depth* indicates the limit in the depth for the iterative deepening strategy for predicate `instanceof`.

Test	**Depth**	**Time** (secs)
Generation of 1000 natural numbers	6000	1.5
`42⟵even`	1000	20.0

The Clay specifications of `Bool`, `Num`, `Nat`, `RGBNat`, `CYMNat`, `Cell`, `ReCell`, their translation into Prolog and the Prolog implementation of the Clay theory, can be found at http://babel.ls.fi.upm.es/~angel/papers/2010lopstr-code.tgz.

[3] Predicate `num2nat/2` translates Prolog positive integers into Clay abstract syntax trees that represent such numbers.

6 Related Work and Conclusions

We have presented the compilation scheme of an object oriented formal notation into logic programs. This allows the generation of executable prototypes that can help in validating requirements, e.g. by means of testing.

In this paper we have focused on the generation of code from implicit method specifications, specially in presence of recursive definitions, something which is seldom supported by other lightweight methods and tools.

Early experiments with our prototype compiler show the feasibility of the approach, but also the limitations of a naive application of Prolog's standard search mechanisms. In fact, obtaining an efficient search scheme is one of the challenging aspects of this work. Our current implementation combines several techniques: Lloyd-Topor transforms of first-order formulae, constructive negation, iterative deepening search, to name a few.

Comparison between the Prolog code obtained from our compiler and that crafted by hand still shows room for improvement. However, there exist more mature tools (see, for instance, ProB [17, 18]) which also generate logic programs from formal specifications, that show that certain extensions of logic programming (constraints, coroutining, etc.) can help in dramatically improving the efficiency of the resulting code in an automated way.

Certain features of object oriented programming (e.g. mutable state) have been left out of this presentation. Studying the introduction of state in our code generation scheme would help in applying the ideas presented in this paper to other object oriented formal notations like VDM++, Object-Z, Troll or OASIS [7, 25, 16, 21].

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Alloy Website. http://alloy.mit.edu.
3. Jeffrey Van Baalen and Richard E. Fikes. The role of reversible grammars in translating between representation languages. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 562–571, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
5. James L. Caldwell. Extracting general recursive program schemes in Nuprl's type theory. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*, pages 233–244, London, UK, 2001. Springer-Verlag.
6. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.
7. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
9. Vinu George and Rayford Vaughn. Application of lightweight formal methods in requirement engineering. *CrossTalk - The Journal of Defense Software Engineering*, January 2003.
10. Ángel Herranz and Juan José Moreno-Navarro. On the design of an object-oriented formal notation. In *Fourth Workshop on Rigorous Object Oriented Methods, ROOM 4*. King's College, London, March 2002.
11. Ángel Herranz and Juan José Moreno-Navarro. Formal extreme (and extremely formal) programming. In Michele Marchesi and Giancarlo Succi, editors, *4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003*, number 2675 in LNCS, pages 88–96, Genova, Italy, May 2003.
12. Ángel Herranz and Juan José Moreno-Navarro. Rapid prototyping and incremental evolution using SLAM. In *14th IEEE International Workshop on Rapid System Prototyping, RSP 2003)*, San Diego, California, USA, June 2003.
13. Ángel Herranz and Juan José Moreno-Navarro. *Design Pattern Formalization Techniques*, chapter Modeling and Reasoning about Design Patterns in SLAM-SL. IGI Publishing, March 2007. Other ISBN: 978-1-59904-221-3.
14. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.
15. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
16. Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL a language for object-oriented specification of information systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.
17. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
18. Michael Leuschel, Dominique Cansell, and Michael Butler. Validating and animating higher-order recursive functions in B. In Jean-Raymond Abrial and Uwe Glässer, editors, *Festschrift for Egon Börger*, 2007.
19. John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *J. Log. Program.*, 1(3):225–240, 1984.
20. John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
21. Oscar Pastor Lopez, Fiona Hayes, and Stephen Bear. *OASIS: An Object-Oriented Specification Language.*, volume 593 of *Lecture Notes in Computer Science*, pages 348–363. Springer, Berlin, Heidelberg, January 1992.
22. Ulf Norell. Dependently typed programming in Agda. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 1–2, New York, NY, USA, 2009. ACM.
23. Nicolas Oury and Wouter Swierstra. The power of Pi. *SIGPLAN Not.*, 43(9):39–50, 2008.
24. Prover9 and Mace4 Website. http://www.cs.unm.edu/%7emccune/mace4.
25. Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Debugging with Incomplete and Dynamically Generated Execution Trees*

David Insa and Josep Silva

Universidad Politécnica de Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{dinsa,jsilva}@dsic.upv.es

Abstract. Declarative debugging is a powerful debugging technique that has been adapted to practically all programming languages. However, the technique suffers from important scalability problems in both time and memory. With realistic programs the huge size of the execution tree handled makes the debugging session impractical and too slow to be productive. In this work, we present a new architecture for declarative debuggers in which we adapt the technique to work with incomplete execution trees. This allows us to avoid the problem of loading the whole execution tree in main memory and solve the memory scalability problems. We also provide the technique with the ability to debug execution trees that are only partially generated. This allows the programmer to start the debugging session even before the execution tree is computed. This solves the time scalability problems. We have implemented the technique and show its practicality with several experiments conducted with real applications.

1 Introduction

Declarative debugging is a semi-automatic debugging technique that has been extended to practically all paradigms, and many techniques [9, 3, 14, 6, 5] have been defined to improve the original proposal [12]. The technique produces a dialogue between the debugger and the programmer to find the bugs. Essentially, it relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, declarative debuggers compare the results of sub-computations with what the programmer intended. By asking the programmer questions or using a formal specification the system can identify precisely the location of a program's bug.

Traditionally, declarative debugging consists of two sequential phases: The construction of an *Execution Tree* (ET) which is an intermediate data structure that represents the execution of the program including all subcomputations; and

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08).

the exploration of the ET with a given strategy to find the bug. A survey can be found in [15].

This technique is very powerful thanks to the ET, because it guarantees that the bug will be found whenever the programmer answers the questions of the debugger. Unfortunately, with realistic programs, the ET can be huge (indeed gigabytes) and this is the main drawback of this debugging technique, because scalability has not been solved yet: If the ET is stored in main memory, the debugger is out of memory with big ETs that do not fit. If, on the other hand, it is stored in a database, debugging becomes a slow task because some questions need to explore a big part of the ET; and also because storing the ET in the database is a time-consuming task.

Modern declarative debuggers allow the programmer to freely explore the ET with graphical user interfaces (GUI) that represent computations [4]. The scalability problem also translates to these features, because showing the whole ET (or even the part of the ET that participates in a subcomputation) is often not possible due to memory overflow reasons.

In some languages, the scalability problem is inherent to the current technology that supports the language and cannot be avoided with more accurate implementations. For instance, in Java, current declarative debuggers (e.g., JavaDD [7] and DDJ [4]) are based on the *Java Platform Debugger Architecture* (JPDA) [10] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface which helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Unfortunately, the time scalability problem described before also translates to this architecture, and hence, any debugger implemented with the JPDA will suffer the scalability problems. For instance, we conducted some experiments to measure the time needed by JPDA to produce the ET[1] of a collection of medium/large benchmarks. Results are shown in column `ET time` of Table 1. Note that, in order to generate the ET, the JVM with JPDA needs some minutes, thus the debugging session would not be able to generate the first question until this time.

In this work we propose a new implementation model that solves the three scalability problems, namely, memory, time and graphical visualization of the ET. Clearly, the ET is the bottleneck of the technique, and sometimes (e.g., in Java) it is not possible to generate it fast. Therefore, our model is based on the following question: *Is it possible to start the debugging session before having computed the whole ET?* The answer is yes.

We propose a framework in which the debugger uses the (incomplete) ET while it is being dynamically generated. Roughly speaking, two processes run in parallel. The first process generates the ET and stores it into both a database (the whole ET) and main memory (a part of the ET). The other process starts the debugging session by only using the part of the ET already generated. Moreover, we use a three-cache memories system to speedup the generation of questions

[1] These times corresponds to the execution of the program, the production of the ET and its storage in a database.

and to guarantee that the debugger is never out of memory (including the GUI components).

2 A New Architecture for Declarative Debuggers

This section presents a new architecture in which declarative debugging is not done in two sequential phases, but in two concurrent phases; that is, while the ET is being generated, the debugger is able to produce questions. This new architecture solves the scalability problems of declarative debugging. In particular, we use a database to store the whole ET, and only a part of it is loaded to main memory.

Moreover, in order to make the algorithms that traverse the ET independent of the database caching problems, we use a three-tier architecture where all the components have access to a *virtual execution tree* (VET). The VET is a data structure which is identical to the ET except that some nodes are missing (not generated yet) or incomplete (they only store a part of the method invocation) Hence, standard strategies can traverse the VET because the structure of the ET is kept.

The VET is produced while running the program. For each method invocation, a new node is added to it with the method parameters and the context before the call. The result and the context after the call are only added to the node when the method invocation finishes.

Let us explain the components of the architecture with the diagram in Figure 1. Observe that each tier contains a cache that can be seen as a view of the VET. Each cache is used for a different task:

Persistence cache. It is used to store the nodes of the VET in the database. Therefore, when the whole VET is in the database, the persistence cache is not used anymore. Basically, it specifies the maximum number of completed nodes that can be stored in the VET. This bound is called *persistence bound* and it ensures that main memory is never overflowed.

Logic cache. It defines a subset of the VET. This subset contains a limited number of nodes (in the following, *logic bound*), and these nodes are those with the highest probability of being asked, therefore, they should be retrieved from the database. This allows us to load in a single database transaction those nodes that are going to be probably asked and thus reducing the number of accesses to the database.

Presentation cache. It contains the part of the VET that is shown to the user in the GUI. The number of nodes in this cache should be limited to ensure that the GUI is not out of memory or it is too slow. The presentation cache defines a subtree inside the logic cache. Therefore, all the nodes in the presentation cache are also nodes of the logic cache. Here, the subtree is defined by selecting one root node and a depth (in the following, *presentation bound*).

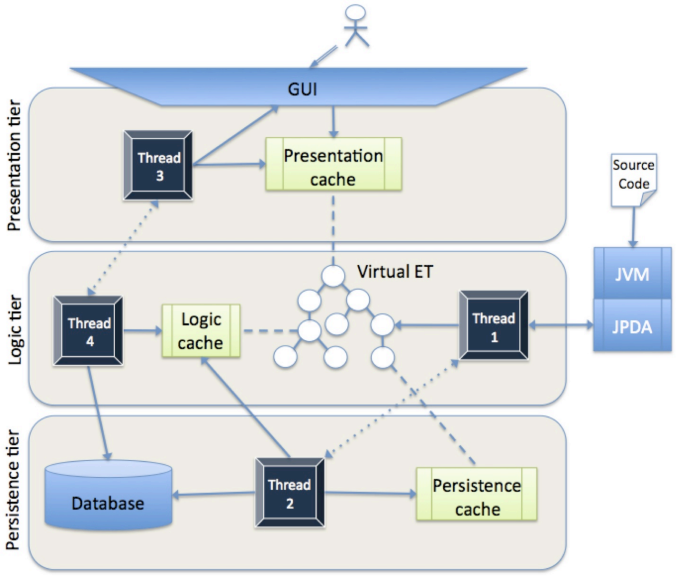
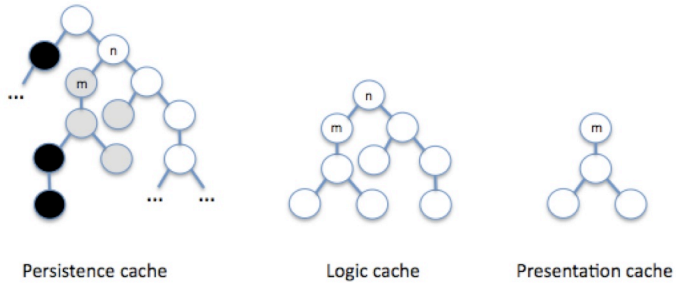


Fig. 1. Architecture of a scalable declarative debugger

The whole VET does not usually fit in main memory. Therefore, a mechanism to remove nodes from it and store them in a database is needed. When the number of complete nodes in the VET is close to the persistence bound, some of them are moved to the database, and only their identifiers remain in the VET. This allows the debugger to keep the whole ET structure in main memory and use identifiers to retrieve nodes from the database when needed.

Example 1. Consider the following trees:



The tree at the left is the VET of a debugging session where gray nodes are those already completed (their associated method invocation already finished); black nodes are completed nodes that are only stored in the database (only their identifiers are stored in the VET), and white nodes are nodes that have not been completed yet (they represent a method invocation that has not finished yet). It could be possible that some of the white nodes had a children not generated

yet. Note that this VET is associated with an instant of the execution; and new nodes could be generated later. The tree in the middle is the part of the VET referenced by the logic cache, in this case it is a tree, being n the root node and a depth of four, but in general it could contain unconnected nodes. Similarly, the tree at the right is the part of the VET referenced by the presentation cache, with m the root node and a depth of three. Note that the presentation cache is a subset of the logic cache.

The behavior of the debugger is controlled by four threads that run in parallel, one for the presentation tier (thread 3), two for the logic tier (threads 1 and 4) and one for the persistence tier (thread 2). Threads 1 and 2 control the generation of the VET and its storage in the database. They collaborate via synchronizations and message passing. Threads 3 and 4 communicate with the user and generate the questions. They also collaborate and are independent of threads 1 and 2. A description of the threads and their behavior specified with pseudo-code follows:

Thread 1 (Contruction of the VET) This thread is in charge of constructing the VET. It is the only one that communicates with the JPDA and JVM. Therefore, we could easily construct a declarative debugger for another language (e.g., C++) by only replacing this thread. Basically, this thread executes the program and for every method invocation performed, it constructs a new node stored in the VET. When the number of complete nodes (given by function *completeNodes*) is close to the persistence bound, this thread sends to thread 2 the *wake up* signal. Then, thread 2 moves some nodes to the database. If the persistence bound is reached, thread 1 sleeps until enough nodes have been removed from the VET and it can continue generating new nodes.

Algorithm 1 Contruction of the VET (Thread 1)

Input: A source program $\mathcal{P}$
Output: A VET $\mathcal{V}$
Initialization: $\mathcal{V} = \emptyset$

repeat
(1) Run $\mathcal{P}$ with JPDA and catch event e
  case e **of**
  new method invocation I:
(2) create a new node N with I
(3) add N to $\mathcal{V}$
  method invocation I ended:
(4) complete node N associated with I
(5) **If** completeNodes($\mathcal{V}$)== persistenceBound/2
(6) **then** send to thread 2 the wake up signal
(7) **If** completeNodes($\mathcal{V}$)== persistenceBound
(8) **then** sleep
until P finishes or the bug is found

Thread 2 (Controlling the size of the VET) This thread ensures that the VET always fits in main memory. It controls what nodes of the VET should be stored in main memory, and what nodes should be stored in the database. When the number of completed nodes in the VET is close to the persistence bound thread 1 wakes up thread 2 that removes some[2] nodes from the VET and copies them to the database. It uses the logic cache to decide what nodes to store in the database. Concretely, it tries to store in the database as many nodes as possible that are not in the logic cache, but always less than the persistence bound divided by two. When it finishes, it sends to thread 1 the *wake up* signal and sleeps.

Algorithm 2 Controlling the size of the VET (Thread 2)

Input: A VET $\mathcal{V}$
Output: An ET stored in a database

```
repeat
1) Sleep until wake up signal is received
   repeat
2)     Look into the persistence cache for the next completed node N of the VET
3)     if N is not found
4)     then wake up thread 1
5)          break
6)     else store N in the database
7)     if N is the root node then exit
```

Thread 3 (Interface communication) This thread is the only one that communicates with the user. It controls the information shown in the GUI with the presentation cache. According to the user's answers, the strategy selected, and the presentation bound, this thread selects the root node of the presentation cache. This task is done question after question according to the programmer answers, ensuring that the question asked (using function *AskQuestion*), its parent, and as many descendants as the presentation bound allows, are shown in the GUI.

Algorithm 3 Interface communication (Thread 3)

Input: Answers of the user
Output: A buggy node

```
repeat
(1) ask thread 4 to produce a question
(2) update presentation cache and GUI visualization
(3) answer = AskQuestion(question)
(4) send answer to thread 4
until a buggy node is found
```

[2] In our implementation, it removes half of the nodes. Our experiments reveal that this is a good choice because it keeps threads 1 and 2 continuously running in a producer-consumer manner.

Thread 4 (Selecting questions) This thread chooses the next question according to a given strategy using function *SelectNextQuestion* that implements standard strategies. With the node selected, the logic cache is updated and all the nodes in the logic cache are loaded from the database. This is done with function *UpdateLogicCache* that uses the node selected as the root, and the logic bound to compute the logic cache. All the nodes that belong to the new logic cache and that do not belong to the previous logic cache are loaded from the database using function *FromDatabaseToET*.

Algorithm 4 Selecting questions (Thread 4)

Input: A strategy $\mathcal{S}$ and a VET $\mathcal{V}$
Output: A buggy node

repeat
(1) question = SelectNextQuestion($\mathcal{V}$,$\mathcal{S}$)
(2) missingNodes = UpdateLogicCache()
(3) **If** (question $\notin \mathcal{V}$) **then** $\mathcal{V}$ = FromDatabaseToET($\mathcal{V}$,missingNodes)
(4) send question to thread 3
(5) get answer from thread 3
until a buggy node is found

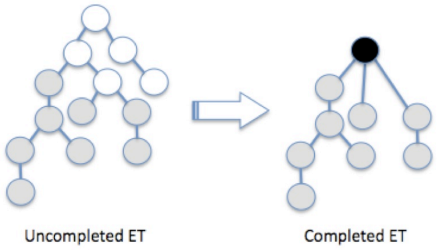
2.1 Redefining the Strategies for Declarative Debugging

In Algorithm 4, a strategy is used to generate the sequence of questions by selecting nodes in the VET. Nevertheless, all declarative debugging strategies in the literature have been defined for ETs and not for VETs where incomplete nodes can exist. All of them assume that the information of all ET nodes is available. Clearly, this is not true in our context and thus, the strategies would fail. For instance, the first node asked by the strategy top-down and its variants is always the root node of the ET. However, this node is the last node completed by Algorithm 3. Hence, these strategies could not even start until the whole ET is completed, and this is exactly the problem that we want to avoid.

Therefore, in this section we propose a redefinition of the strategies for declarative debugging so that they can work with VETs.

A first solution could be to define a transformation from a VET with incomplete nodes to a VET where all nodes are completed. This can be done by inserting a new root node with the equation $1 = 0$. Then, the children of this node would be all the completed nodes whose parent is incomplete. In this way, (i) all nodes of the produced ET would be completed and could be asked; (ii) the parent-child relation is kept in all the subtrees of the ET; and (iii) it is guaranteed that at least one bug (the root node) exists. If the debugging session finishes with the root node as buggy, it means that the node with the "real" bug (if any) has not been completed yet.

Example 2. Consider the following VETs:



In the VET at the left, gray nodes are completed, and white nodes are incomplete. This VET can be transformed into the VET at the right where all nodes are completed. The new artificial root is the black node which ensures that at least one buggy node exists.

From an implementation point of view, this transformation is inefficient and costly because the VET is being generated continuously by thread 1, and hence, this transformation should be done repeatedly question after question. In contrast, a more efficient solution is to redefine the strategies so that they ignore incomplete nodes. For instance, top-down [1] only asks for the descendants of a node that are completed and that do not have a completed ancestor. Similarly, Binks top-down [2] would ask first for the completed descendant that in turn contains more completed descendants. D&Q [12] would ask for the node that divides the VET in two subtrees with the same number of completed nodes, and so on. We refer the interested reader to the source code of our implementation that is publicly available and where all strategies have been reimplemented for VETs.

Even though the architecture presented has been discussed in the context of Java, it can work for other languages with very few changes. Observe that the part of an algorithmic debugger that is language-dependent is the front-end, and our technique relies on the back-end. Once the VET is generated, the back-end can handle the VET mostly independent of the language. In particular, the strategies can traverse the VET being unaware of the meaning of the questions.

3 Implementation

We have implemented the technique presented in this paper and integrated it into DDJ 2.4. The implementation has been tested with a collection of real applications (e.g., an interpreter, a parser, a debugger, etc).

Table 1 summarizes the results of the experiments performed. These experiments have been done in an Intel Core2 Quad 2.67 GHz with 2GB RAM.

The first column contains the names of the benchmarks. For each benchmark, the second and third columns give an idea of the size of the execution. Fourth and fifth columns are time measures. Finally, sixth and seventh columns show memory bounds. Concretely, column `variables number` shows the number of variables participating (possibly repeated) in the execution considered. Column `ET size` shows the size in Mb of the ET when it is completed, this measure

Benchmark	var. num.	ET size	ET time	node time	cache lim.	ET depth
argparser	8.812	2 Mb	22 s.	407 ms.	7/7	7
cglib	216.931	200 Mb	230 s.	719 ms.	11/14	18
kxml2	194.879	85 Mb	1318 s.	1844 ms.	6/6	9
javassist	650.314	459 Mb	556 s.	844 ms.	7/7	16
jtstcase	1.859.043	893 Mb	1913 s.	1531 ms.	17/26	57
HTMLcleaner	3.575.513	2909 Mb	4828 s.	609 ms.	4/4	17

Table 1. Benchmark results

has been taken from the size of the ET in the database (of course, it includes compaction). Column `ET time` is the time needed to finish the ET. Column `node time` is the time needed to complete the first node of the ET. Column `cache limit` shows the presentation bound and the depth of the logic cache of these benchmarks. After these bounds, the computer was out of memory. Finally, column `ET depth` shows the depth of the ET after it was constructed.

Observe that a standard declarative debugger is hardly scalable to these real programs. With the standard technique, even if the ET fits in main memory or we use a database, the programmer must wait for a long time until the ET is completed and the first question can be asked. In the worst case, this time is more than one hour. Contrarily, with the new technique, the debugger can start to ask questions before the ET is completed. Note that the time needed to complete the first node is always less than two seconds. Therefore, the debugging session can start almost instantaneously.

The last two columns of the table give an idea of how big is the ET shown in the GUI before it is out of memory. In general, five levels of depth is enough to see the question asked and the part of the computation closely related to this question. In the experiments only HTMLcleaner was out of memory when showing five levels of the ET in the GUI.

All the information related to the experiments, the source code of the benchmarks, the bugs, the source code of the tool and other material can be found at http://www.dsic.upv.es/~jsilva/DDJ

4 Conclusions

Declarative debugging is a powerful debugging technique that has been adapted to practically all programming languages. The main problem of the technique is its low level of scalability both in time and memory. With realistic programs the huge size of the internal data structures handled makes the debugging session impractical and too slow to be productive.

In this work, we propose the use of VETs as a suitable solution to these problems. This data structure has two important advantages: It is prepared to be partially stored in main memory, and completely stored in secondary memory. This ensures that it will always fit in main memory and thus solves the memory scalability problem. In addition, it can be used during a debugging session before

it is completed. For this, we have implemented a version of standard declarative debugging strategies able to work with VETs. This solves the time scalability problem as demonstrated by our experiments.

In our implementation, the programmer can control how much memory is used by the GUI components, and by the strategies thanks to the use of three cache memories. The most important result is that experiments confirm that, even with large programs and long running computations, a debugging session can start to ask questions after only few seconds.

References

1. E. Av-Ron. *Top-Down Diagnosis of Prolog Programs*. PhD thesis, Weizmanm Institute, 1984.
2. D. Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.
3. R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.
4. R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science, 2006.
5. R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for maude functional modules. *Electronic Notes Theoretical Computer Science*, 238(3):63–81, 2009.
6. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.
7. H. Girgis and B. Jayaraman. JavaDD: a Declarative Debugger for Java. Technical Report 2006-07, University at Buffalo, March 2006.
8. G. Kokai, J. Nilson, and C. Niss. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press, 1999.
9. I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
10. Sun Microsystems. Java Platform Debugger Architecture - JPDA. Available from URL: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.
11. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
12. E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
13. J. Silva. An Empirical Evaluation of Algorithmic Debugging Strategies. Technical Report DSIC-II/10/09, UPV, 2009. Available from URL: `http://www.dsic.upv.es/~jsilva/research.htm#techs`.
14. J. Silva. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140, 2006.
15. J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.

A Semantics to Generate the Context-sensitive Synchronized Control-Flow Graph

Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit

Universidad Politécnica de Valencia, Camino de Vera S/N, E-46022 Valencia, Spain
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

Abstract. The CSP language allows the specification and verification of complex concurrent systems. Many analyses for CSP exist that have been successfully applied in different industrial projects. However, the cost of the analyses performed is usually very high, and sometimes prohibitive, due to the complexity imposed by the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations. In this work, we define a data structure that allows us to statically simplify a specification before the analyses. This simplification can drastically reduce the time needed by many CSP analyses. We also introduce an algorithm able to automatically generate this data structure from a CSP specification. The algorithm has been proved correct and its implementation for the CSP's animator ProB is publicly available.

1 Introduction

The *Communicating Sequential Processes* (CSP) [3, 13] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [5], reliability analysis [4], refinement checking [12], etc.) which are often based on a data structure able to represent all computations of a specification.

Recently, a new data structure called *Context-sensitive Synchronized Control-Flow Graph* (CSCFG) has been proposed [7]. This data structure is a graph that allows us to finitely represent possibly infinite computations, and it is particularly interesting because it takes into account the context of process calls, and thus it allows us to produce analyses that are very precise. In particular, some analyses (see, e.g., [8, 9]) use the CSCFG to simplify a specification with respect to some term by discarding those parts of the specification that cannot be executed before the term and thus they cannot influence it. This simplification is automatic and thus it is very useful as a preprocessing stage of other analyses.

However, computing the CSCFG is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is the reason why there does not exist any correctness result which formally relates the CSCFG of a specification to its execution. This result is needed to prove important properties (such as correctness and completeness) of the techniques based on the CSCFG.

In this work, we formally define the CSCFG and a technique to produce the CSCFG of a given CSP specification. Roughly, we instrument the CSP standard semantics (Chapter 7 in [13]) in such a way that the execution of the instrumented semantics produces as a side-effect the portion of the CSCFG associated with the performed computation. Then, we define an algorithm which uses the instrumented semantics to build the complete CSCFG associated with a CSP specification. This algorithm executes the semantics several times to explore all possible computations of the specification, producing incrementally the final CSCFG.

2 The Syntax and Semantics of CSP

In order to make the paper self-contained, this section recalls CSP's syntax and semantics [3, 13]. For concretion, and to facilitate the understanding of the following definitions and algorithm, we have selected a subset of CSP that is sufficiently expressive to illustrate the method, and it contains the most important operators that produce the challenging problems such as deadlocks, non-determinism and parallel execution.

We use the following domains: process names ($M, N \ldots \in Names$), processes ($P, Q \ldots \in Procs$) and events ($a, b \ldots \in \Sigma$). A CSP specification is a finite set of process definitions $N = P$ with $P = M \mid a \rightarrow P \mid P \sqcap Q \mid P \,\Box\, Q \mid P \underset{X \subseteq \Sigma}{||} Q \mid STOP$. Therefore, processes can be a call to another process or a combination of the following operators:

Prefixing ($a \rightarrow P$) Event a must happen before process P.

Internal choice ($P \sqcap Q$) The system chooses non-deterministically to execute one of the two processes P or Q.

External choice ($P \,\Box\, Q$) It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Synchronized parallelism ($P \underset{X \subseteq \Sigma}{||} Q$) Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes, it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* (represented by $|||$) where no synchronizations exist (i.e., $X = \emptyset$).

Stop ($STOP$) Synonym of deadlock: It finishes the current process.

We now recall the standard operational semantics of CSP as defined by Roscoe [13]. It is presented in Fig. 1 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. In the following, we assume that the system starts with an initial state `MAIN`, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma^\tau = \Sigma \cup \{\tau\}$. Events in Σ are visible from the external

environment, and can only happen with its co-operation (e.g., actions of the user). Event τ is an internal event that cannot be observed from outside the system and it happens automatically as defined by the semantics. In order to perform computations, we construct an initial state and (non-deterministically) apply the rules of Fig. 1.

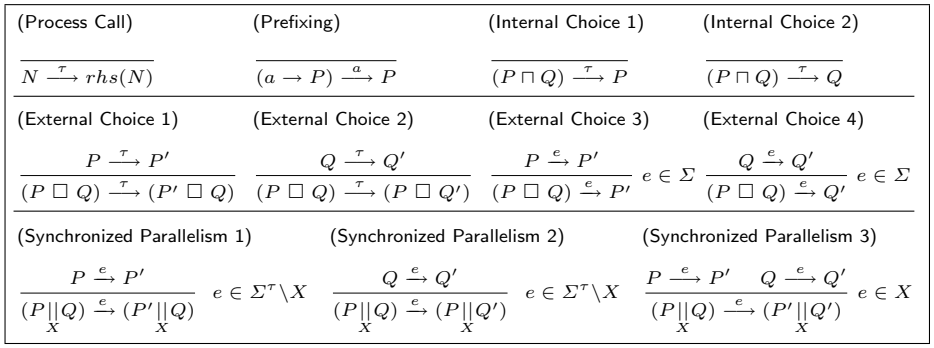


Fig. 1. CSP's operational semantics

3 Context-sensitive Synchronized Control-Flow Graphs

The CSCFG was proposed in [7, 9] as a data structure able to finitely represent all possible (often infinite) computations of a CSP specification. This data structure is particularly useful to simplify a CSP specification before its static analysis. The simplification of industrial CSP specifications allows us to drastically reduce the time needed to perform expensive analyses such as model checking. Algorithms to construct CSCFGs have been implemented [8] and integrated into the most advanced CSP environment ProB [6]. In this section we introduce a new formalization of the CSCFG that directly relates the graph construction to the control-flow of the computations it represents.

A CSCFG is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. In addition, the source position (in the specification) of each literal (i.e., events, operators and process names) is also included in the CSCFG. This is very useful because it provides the CSCFG with the ability to determine what parts of the source code have been executed and in what order. The inclusion of source positions in the CSCFG implies an additional level of complexity in the semantics, but the benefits of providing the CSCFG with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP

specification and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. *(Specification position) A* specification position *is a pair* (N, w) *where* $N \in \mathcal{N}$ *and* w *is a sequence of natural numbers (we use* Λ *to denote the empty sequence). We let* $\mathcal{P}os(o)$ *denote the specification position of an expression* o*. Each process definition* $N = P$ *of a CSP specification is labelled with specification positions. The specification position of its left-hand side is* $\mathcal{P}os(N) = (N, 0)$*. The right-hand side (abbrev.* rhs*) is labelled with the call* $\texttt{AddSpPos}(P, (N, \Lambda))$*; where function* `AddSpPos` *is defined as follows:*

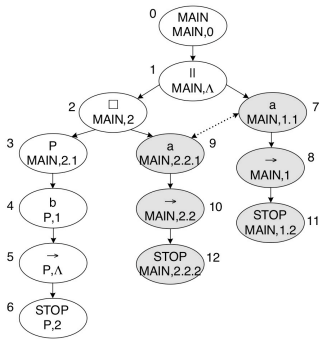
$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \text{if } P \in \mathcal{N} \\ STOP_{(N,w)} & \text{if } P = STOP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \text{if } P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2)) & \text{if } P = Q\ op\ R\ \ \forall\ op \in \{\sqcap, \Box, ||\} \end{cases}$$

We often use $\mathcal{P}os(\mathcal{S})$ *to denote a set with all positions in a specification* $\mathcal{S}$*.*

Example 1. Consider the CSP specification in Fig. 2(a) where literals are labelled with their associated specification positions (they are underlined) so that labels are unique.

$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \parallel^{\underline{(\texttt{MAIN},\Lambda)}}_{\{\texttt{a}\}}$
$(\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \Box_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$
$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}}$

(a) CSP specification



(b) CSCFG

Fig. 2. CSP specification and its associated CSCFG

In the following, specification positions will be represented with greek letters $(\alpha, \beta, \ldots)$ and we will often use indistinguishably an expression and its associated specification position when it is clear from the context (e.g., in Example 1 we will refer to $(\texttt{P}, 1)$ as `b`).

In order to introduce the definition of CSCFG, we need first to define the concepts of *control-flow*, *path* and *context*.

Definition 2. *(Control-flow) Given a CSP specification $\mathcal{S}$, the* control-flow *is a transitive relation between the specification positions of $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the* control *of α* can pass *to β iff*

i) $\alpha = N \ \wedge \ \beta = \mathit{first}((N, \Lambda))$ with $N = rhs(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap, \square, ||\} \ \wedge \ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha = \beta.1 \ \wedge \ \beta = \ \rightarrow$
iv) $\alpha = \ \rightarrow \ \wedge \ \beta = \mathit{first}(\alpha.2)$

where $\mathit{first}(\alpha)$ is defined as follows: $\mathit{first}(\alpha) = \begin{cases} \alpha.1 & \text{if } \alpha = \ \rightarrow \\ \alpha & \text{otherwise} \end{cases}$

We say that a specification position α is executable *in $\mathcal{S}$ iff the control can pass from the initial state (i.e.,* `MAIN`*) to α.*

For instance, in Example 1, the control can pass from $(\texttt{MAIN}, 2.1)$ to $(\texttt{P}, 1)$ due to rule i), from $(\texttt{MAIN}, 2)$ to $(\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2.2.1)$ due to rule ii), from $(\texttt{MAIN}, 2.2.1)$ to $(\texttt{MAIN}, 2.2)$ due to rule iii), and from $(\texttt{MAIN}, 2.2)$ to $(\texttt{MAIN}, 2.2.2)$ due to rule iv).

As we will work with graphs whose nodes are labelled with positions, we use $l(n)$ to refer to the label of node n.

Definition 3. *(Path) Given a labelled graph $\mathcal{G} = (N, E)$, a* path *between two nodes n_1, $m \in N$, $Path(n_1, m)$, is a sequence $n_1, \ldots, n_k$ such that $n_k \mapsto m \in E$ and for all $1 \leq i < k$ we have $n_i \mapsto n_{i+1} \in E$. The path is* loop-free *if for all $i \neq j$ we have $n_i \neq n_j$.*

Definition 4. *(Context) Given a labelled graph $\mathcal{G} = (N, E)$ and a node $n \in N$, the* context *of n, $\mathcal{C}on(n) = \{m \mid l(m) = M$ with $(M = P) \in \mathcal{S}$ and there exists a loop-free path $m \mapsto^* n\}$.*

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. This is represented by the set of process calls in the computation that were done before the specified node. For instance, the CSCFG associated with the specification in Example 1 is shown in Fig. 2(b). In this graph we have that $\mathcal{C}on(4) = \{0, 3\}$, i.e., `b` is being executed after having called processes `MAIN` and `P`. Note that focussing on a process call node we can use the context to identify loops; i.e., we have a loop whenever $n \in \mathcal{C}on(m)$ with $l(n) = l(m) \in Names$. Note also that the CSCFG is unique for a given CSP specification [9].

Definition 5. *(Context-sensitive Synchronized Control-Flow Graph) Given a CSP specification $\mathcal{S}$, its* Context-sensitive Synchronized Control-Flow Graph *(CSCFG) is a labelled directed graph $\mathcal{G} = (N, E_c, E_l, E_s)$ where N is a set of nodes such that $\forall \ n \in N.\ l(n) \in \mathcal{P}os(\mathcal{S})$ and $l(n)$ is executable in $\mathcal{S}$; and edges are divided into three groups:* control-flow edges *(E_c),* loop edges *(E_l) and* synchronization edges *(E_s).*

- *E_c is a set of one-way edges (denoted with $\mapsto$) representing the possible control-flow between two nodes. Control edges do not form loops. The root of the tree formed by E_c is the position of the initial call to* `MAIN`*.*

- *E_l is a set of one-way edges (denoted with $\rightsquigarrow$) such that $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1)$ and $l(n_2)$ are (possibly different) process calls that refer to the same process $M \in \mathcal{N}$ and $n_2 \in \mathcal{C}on(n_1)$.*
- *E_s is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two event nodes ($l(n) \in \Sigma$).*
- *Given a CSCFG, every node labelled (M, Λ) has one and only one incoming edge in E_c; and every process call node has one and only one outgoing edge which belongs to either E_c or E_l.*

Example 2. Consider again the specification of Example 1, shown in Fig. 2(a), and its associated CSCFG, shown in Fig. 2(b). For the time being, the reader can ignore the numbering and color of the nodes; they will be explained in Section 4. Each process call is connected to a subgraph which contains the right-hand side of the called process. For convenience, in this example there are no loop edges;[1] there are control-flow edges and one synchronization edge between nodes $(\texttt{MAIN}, \texttt{2.2.1})$ and $(\texttt{MAIN}, \texttt{1.1})$ representing the synchronization of event `a`.

Note that the CSCFG shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order. Therefore, it is not only useful as a program comprehension tool, but it can be used for program simplification. For instance, with a simple backwards traversal from `a`, the CSCFG reveals that the only part of the code that can be executed before `a` is the underlined part:

$$\underline{\texttt{MAIN}} = (\underline{\texttt{a}} \rightarrow \texttt{STOP}) \underset{\underline{\{\texttt{a}\}}}{\parallel} (\texttt{P} \,\underline{\Box}\, (\underline{\texttt{a}} \rightarrow \texttt{STOP}))$$
$$\texttt{P} = \texttt{b} \rightarrow \texttt{STOP}$$

Hence, the specification can be significantly simplified for those analyses focussing on the occurrence of event `a`.

4 An Algorithm to Generate the CSCFG

This section introduces an algorithm able to generate the CSCFG associated with a CSP specification. The algorithm uses an instrumented operational semantics of CSP which (i) generates as a side-effect the CSCFG associated with the computation performed with the semantics; (ii) it controls that no infinite loops are executed; and (iii) it ensures that the execution is deterministic.

Algorithm 1 controls that the semantics is executed repeatedly in order to deterministically execute all possible computations—of the original (non-deterministic) specification—and the CSCFG for the whole specification is constructed incrementally with each execution of the semantics. The key point of the algorithm is the use of a stack that records the actions that can be performed by the semantics. In particular, the stack contains tuples of the form

[1] We refer the reader to [10] where an example with loop edges is discussed.

$(rule, rules)$ where $rule$ indicates the rule that must be selected by the semantics in the next execution step, and $rules$ is a set with the other possible rules that can be selected. The algorithm uses the stack to prepare each execution of the semantics indicating the rules that must be applied at each step. For this, function `UpdStack` is used; it basically avoids to repeat the same computation with the semantics. When the semantics finishes, the algorithm prepares a new execution of the semantics with an updated stack. This is repeated until all possible computations are explored (i.e., until the stack is empty).

The standard operational semantics of CSP [13] can be non-terminating due to infinite computations. Therefore, the instrumentation of the semantics incorporates a loop-checking mechanism to ensure termination.

Algorithm 1 General Algorithm

Build the initial state of the semantics: $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, \emptyset, \bullet, (\emptyset, \emptyset), \emptyset, \emptyset)$
repeat
  repeat
    Run the rules of the instrumented semantics with the state $state$
  until no more rules can be applied
  Get the new state: $state = (\_, G, \_, (\emptyset, S_0), \_, \zeta)$
  $state = (\texttt{MAIN}_{(\texttt{MAIN},0)}, G, \bullet, (\texttt{UpdStack}(S_0), \emptyset), \emptyset, \emptyset)$
until $\texttt{UpdStack}(S_0) = \emptyset$
return G
where function `UpdStack` is defined as follows:

$$\texttt{UpdStack}(S) = \begin{cases} (rule, rules \backslash \{rule\}) : S' & \text{if } S = (\_, rules) : S' \text{ and } rule \in rules \\ \texttt{UpdStack}(S') & \text{if } S = (\_, \emptyset) : S' \\ \emptyset & \text{if } S = \emptyset \end{cases}$$

The instrumented semantics used by Algorithm 1 is shown in Fig. 3. It is an operational semantics where we assume that every literal in the specification has been labelled with its specification position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple $(P, G, m, (S, S_0), \Delta, \zeta)$, where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the CSCFG constructed so far), m is a numeric reference to the current node in G, (S, S_0) is a tuple with two stacks (where the empty stack is denoted by $\emptyset$) that contains the rules to apply and the rules applied so far, Δ is a set of references to nodes used to draw synchronizations in G and ζ is a graph like G, but it only contains the part of the graph generated for the current computation, and it is used to detect loops. The basic idea of the graph construction is to record the current control with a fresh reference[2] n by connecting it to its parent m. We use the notation $G[n \overset{m}{\mapsto} \alpha]$ either to introduce a node in G or as a condition on G (i.e., G contains node n). This node has reference n, is labelled with specification position α and its parent is m. The edge introduced can be a control, a synchronization or a loop edge. This notation is very convenient because it allows us to add nodes to G,

[2] We assume that fresh references are numeric and generated incrementally.

but also to extract information from G. For instance, with $G[3 \overset{m}{\mapsto} \alpha]$ we can know the parent of node 3 (the value of m), and the specification position of node 3 (the value of α).

Note that the initial state for the semantics used by Algorithm 1 has $\texttt{MAIN}_{(\texttt{MAIN},0)}$ in the control. This initial call to `MAIN` does not appear in the specification, thus we label it with a special specification position $(\texttt{MAIN}, 0)$ which is the root of the CSCFG (see Fig. 2(b)). Note that we use $\bullet$ as a reference in the initial state. The first node added to the CSCFG (i.e., the root) will have parent reference $\bullet$. Therefore, here $\bullet$ denotes the empty reference because the root of the CSCFG has no parent.

An explanation for each rule of the semantics follows.

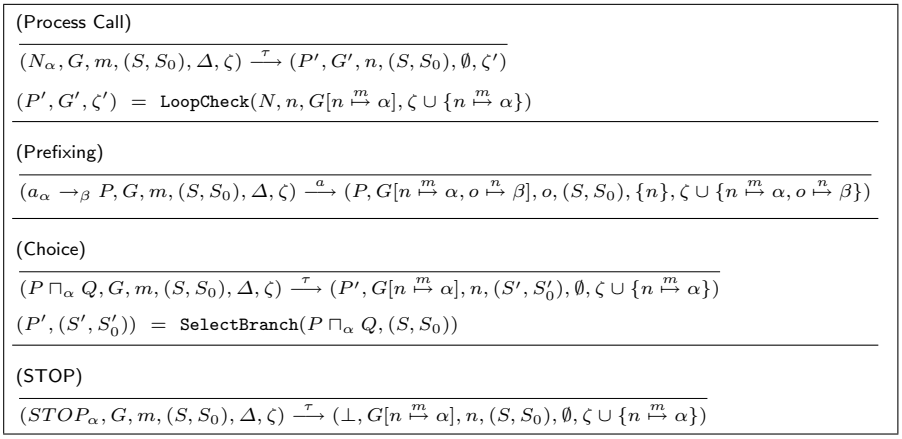


Fig. 3. An instrumented operational semantics that generates the CSCFG

(Process Call) The called process N is unfolded, node n (a fresh reference) is added to the graphs G and ζ with specification position α and parent m. In the new state, n represents the current reference. The new expression in the control is P', computed with function `LoopCheck` which is used to prevent infinite unfolding and is defined below. No event can synchronize in this rule, thus Δ is empty.

$$\texttt{LoopCheck}(N, n, G, \zeta) = \begin{cases} (\circlearrowleft_s(rhs(N)), G[n \rightsquigarrow s], \zeta \cup \{n \rightsquigarrow s\}) & \text{if } \exists s.s \overset{t}{\mapsto} N \in G \\ & \wedge s \in Path(0, n) \\ (rhs(N), G, \zeta) & \text{otherwise} \end{cases}$$

Function `LoopCheck` checks whether the process call in the control has not been already executed (if so, we are in a loop). When a loop is detected, a loop edge between nodes n and s is added to the graph G and to ζ; and the right-hand side of the called process is labelled with a special symbol $\circlearrowleft_s$. This label is later

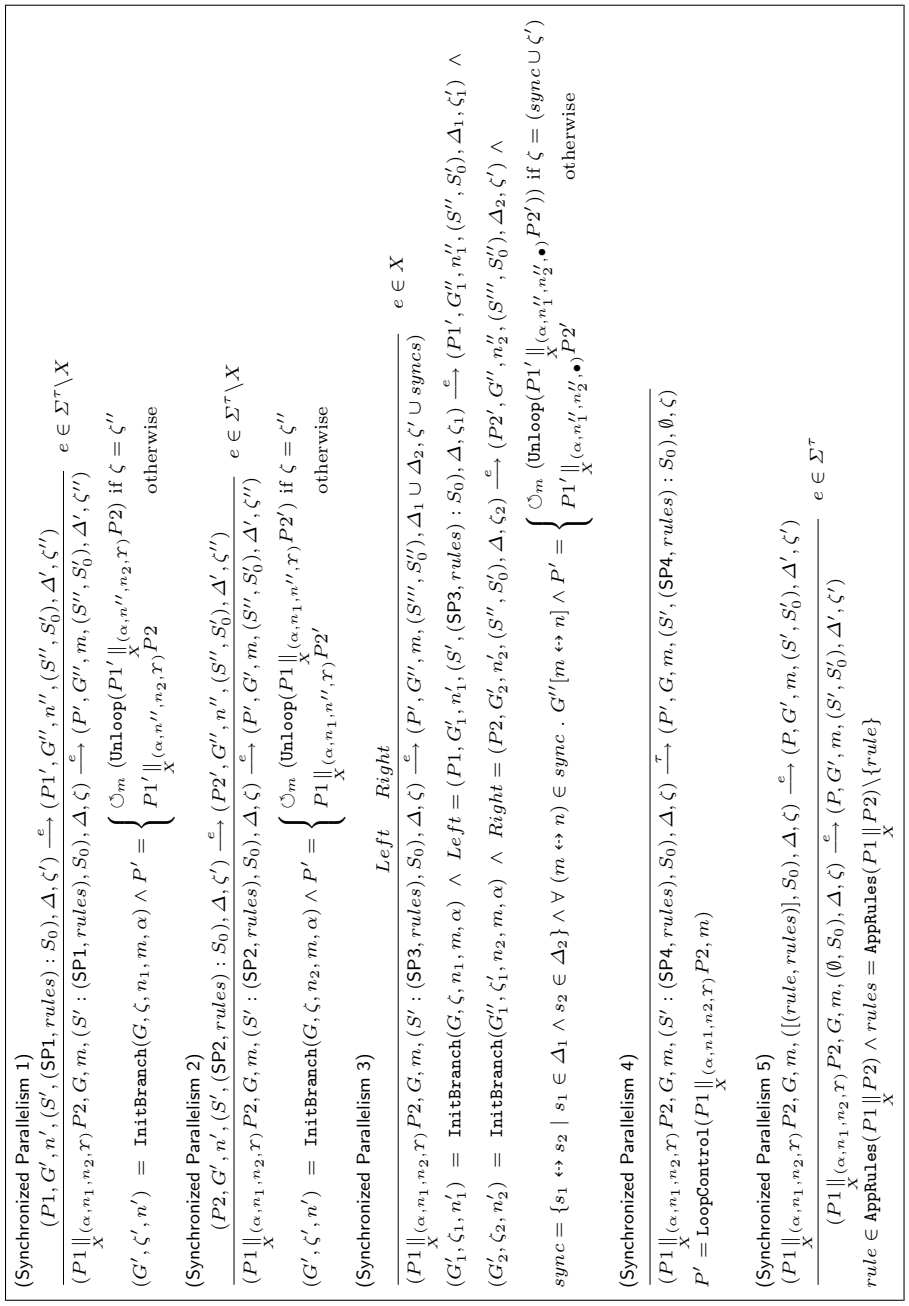


Fig. 3. An instrumented operational semantics that generates the CSCFG (cont.)

used by rule (Synchronized Parallelism 4) to decide whether the process must be stopped. The loop symbol $\circlearrowleft$ is labelled with the position s of the process call of the loop. This is used to know what is the reference of the process' node if it is unfolded again.
(Prefixing) This rule adds nodes n (the prefix) and o (the prefixing operator) to the graphs G and ζ. In the new state, o becomes the current reference. The new control is P. The set Δ is $\{n\}$ to indicate that event a has occurred and it must be synchronized when required by (Synchronized Parallelism 3).
(Choice) The only sources of non-determinism are choice operators (different branches can be selected for execution) and parallel operators (different order of branches can be selected for execution). Therefore, every time the semantics executes a choice or a parallelism, they are made deterministic thanks to the information in the stack S. Both internal and external can be treated with a single rule because the CSCFG associated to a specification with external choices is identical to the CSCFG associated to the specification with the external choices replaced by internal choices. This rule adds node n to the graphs which is labelled with the specification position α and has parent m. In the new state, n becomes the current reference. No event can synchronize in this rule, thus Δ is empty.

Function `SelectBranch` is used to produce the new control P' and the new tuple of stacks (S', S'_0), by selecting a branch with the information of the stack. Note that, for simplicity, the lists constructor ":" has been overloaded, and it is also used to build lists of the form $(A : a)$ where A is a list and a is the last element:

$$\texttt{SelectBranch}(P \sqcap_\alpha Q, (S, S_0)) = \begin{cases} (P, (S', (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{if } S = S' : (\mathsf{C1}, \{\mathsf{C2}\}) \\ (Q, (S', (\mathsf{C2}, \emptyset) : S_0)) & \text{if } S = S' : (\mathsf{C2}, \emptyset) \\ (P, (\emptyset, (\mathsf{C1}, \{\mathsf{C2}\}) : S_0)) & \text{otherwise} \end{cases}$$

If the last element of the stack S indicates that the first branch of the choice ($\mathsf{C1}$) must be selected, then P is the new control. If the second branch must be selected ($\mathsf{C2}$), the new control is Q. In any other case the stack is empty, and thus this is the first time that this choice is evaluated. Then, we select the first branch (P is the new control) and we add $(\mathsf{C1}, \{\mathsf{C2}\})$ to the stack S_0 indicating that $\mathsf{C1}$ has been fired, and the remaining option is $\mathsf{C2}$.

For instance, when the CSCFG of Fig. 2(b) is being constructed and we reach the choice operator (i.e., $(\texttt{MAIN}, 2)$), then the left branch of the choice is evaluated and $(\mathsf{C1}, \{\mathsf{C2}\})$ is added to the stack to indicate that the left branch has been evaluated. The second time it is evaluated, the stack is updated to $(\mathsf{C2}, \emptyset)$ and the right branch is evaluated. Therefore, the selection of branches is predetermined by the stack, thus, Algorithm 1 can decide what branches are evaluated by conveniently handling the information of the stack.
(Synchronized Parallelism 1 and 2) The stack determines what rule to use when a parallelism operator is in the control. If the last element in the stack is $\mathsf{SP1}$, then (Synchronized Parallelism 1) is used. If it is $\mathsf{SP2}$, (Synchronized Parallelism 2) is used.

In a synchronized parallelism composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes for

both processes can be added interwoven to the graph. Hence, the semantics needs to know in every state the references to be used in both branches. This is done by labelling each parallelism operator with a tuple of the form $(\alpha, n_1, n_2, \Upsilon)$ where α is the specification position of the parallelism operator; n_1 and n_2 are respectively the references of the last node introduced in the left and right branches of the parallelism, and they are initialised to $\bullet$; and Υ is a node reference used to decide when to unfold a process call (in order to avoid infinite loops), also initialised to $\bullet$. The sets Δ' and ζ'' are passed down unchanged so that another rule can use them if necessary. In the case that ζ is equal to ζ'', meaning that nothing has change in this derivation, this rule detects that the parallelism is in a loop; and thus, in the new control the parallelism operator is labelled with $\circlearrowleft$ and all the other loop labels are removed from it (this is done by a trivial function `Unloop`). These rules develop the branches of the parallelism until they are finished or until they must synchronize. They use function `InitBranch` to introduce the parallelism into the graph and into ζ the first time it is executed and only if it has not been introduced in a previous computation. For instance, consider a state where a parallelism operator is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, when, e.g., rule (Synchronized Parallelism 1) is applied, a node $1 \overset{0}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G and the parallelism operator is relabelled to $((\texttt{MAIN}, \Lambda), \texttt{x}, \bullet, \bullet)$ where x is the new reference associated with the left branch. After executing function `InitBranch`, we get a new graph and a new reference. Its definition is the following:

$$\texttt{InitBranch}(G, \zeta, n, m, \alpha) = \begin{cases} (G[o \overset{m}{\mapsto} \alpha], \zeta \cup \{o \overset{m}{\mapsto} \alpha\}, o) & \text{if } n = \bullet \\ (G, \zeta, n) & \text{otherwise} \end{cases}$$

(Synchronized Parallelism 3) It is applied when the last element in the stack is SP3. It is used to synchronize the parallel processes. In this rule, Υ is replaced by $\bullet$, meaning that a synchronization edge has been drawn and the loops could be unfolded again if it is needed. The set *sync* of all the events that have been executed in this step must be synchronized. Therefore, all the events occurred in the subderivations of $P1$ (Δ_1) and $P2$ (Δ_2) are mutually synchronized and added to both G'' and ζ'.

(Synchronized Parallelism 4) This rule is applied when the last element in the stack is SP4. It is used when none of the parallel processes can proceed (because they already finished, deadlocked or were labelled with $\circlearrowleft$). When a process is labelled as a loop with $\circlearrowleft$, it can be unlabelled to unfold it once[3] in order to allow the other processes to continue. This happens when the looped process is in parallel with other process and the later is waiting to synchronize with the former. In order to perform the synchronization, both processes must continue, thus the loop is unlabelled. Hence, the system must stop only when both parallel processes are marked as a loop. This task is done by function `LoopControl`.

[3] Only once because it will be labelled again by rule (Process Call) when the loop is repeated. In [10], we present an example with loops where this situation happens.

It decides if the branches of the parallelism should be further unfolded or they should be stopped (e.g., due to a deadlock or an infinite loop):

$$\texttt{LoopControl}(P \underset{X}{\|}_{(\alpha,p,q,\Upsilon)} Q, m) =$$

$$\begin{cases} \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q_{\circlearrowleft},\bullet)} Q'_{\circlearrowleft}) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \wedge Q' = \circlearrowleft_{q_{\circlearrowleft}}(Q'_{\circlearrowleft}) \\ \circlearrowleft_m(P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',\bullet)} \bot) & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge (Q' = \bot \vee (\Upsilon = p_{\circlearrowleft} \wedge Q' \neq \circlearrowleft_{\_}(\_))) \\ P'_{\circlearrowleft} \underset{X}{\|}_{(\alpha,p_{\circlearrowleft},q',p_{\circlearrowleft})} Q' & \text{if } P' = \circlearrowleft_{p_{\circlearrowleft}}(P'_{\circlearrowleft}) \ \wedge Q' \neq \bot \wedge \ \Upsilon \neq p_{\circlearrowleft} \wedge Q' \neq \circlearrowleft_{\_}(\_) \\ \bot & \text{otherwise} \end{cases}$$

where $(P', p', Q', q') \in \{(P, p, Q, q), (Q, q, P, p)\}$.

When one of the branches has been labelled as a loop, there are three options: (i) The other branch is also a loop. In this case, the whole parallelism is marked as a loop labelled with its parent, and Υ is put to $\bullet$. (ii) Either it is a loop that has been unfolded without drawing any synchronization (this is known because Υ is equal to the parent of the loop), or the other branch already terminated (i.e., it is $\bot$). In this case, the parallelism is also marked as a loop, and the other branch is put to $\bot$ (this means that this process has been deadlocked). Also here, Υ is put to $\bullet$. (iii) If we are not in a loop, then we allow the parallelism to proceed by unlabelling the looped branch. When none of the branches has been labelled as a loop, $\bot$ is returned representing that this is a deadlock, and thus, stopping further computations.

(Synchronized Parallelism 5) This rule is used when the stack is empty. It basically analyses the control and decides what are the applicable rules of the semantics. This is done with function `AppRules` which returns the set of rules R that can be applied to a synchronized parallelism $P \underset{X}{\|} Q$:

$$\texttt{AppRules}(P \underset{X}{\|} Q) = \begin{cases} \{\mathsf{SP1}\} & \text{if } \tau \in \texttt{FstEvs}(P) \\ \{\mathsf{SP2}\} & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \in \texttt{FstEvs}(Q) \\ R & \text{if } \tau \notin \texttt{FstEvs}(P) \wedge \tau \notin \texttt{FstEvs}(Q) \ \wedge \ R \neq \emptyset \\ \{\mathsf{SP4}\} & \text{otherwise} \end{cases}$$

where

$$\begin{cases} \mathsf{SP1} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \wedge e \notin X \\ \mathsf{SP2} \in R & \text{if } \exists e \in \texttt{FstEvs}(Q) \wedge e \notin X \\ \mathsf{SP3} \in R & \text{if } \exists e \in \texttt{FstEvs}(P) \ \wedge \exists e \in \texttt{FstEvs}(Q) \wedge e \in X \end{cases}$$

Essentially, `AppRules` decides what rules are applicable depending on the events that could happen in the next step. These events can be inferred by using function `FstEvs`. In particular, given a process P, function `FstEvs` returns the set of events that can fire a rule in the semantics using P as the control. Therefore, rule (Synchronized Parallelism 5) prepares the stack allowing the semantics to proceed with the correct rule.

$$\texttt{FstEvs}(P) =$$

$$\begin{cases} \{a\} & \text{if } P = a \to Q \\ \emptyset & \text{if } P = \circlearrowleft Q \ \vee \ P = \bot \\ \{\tau\} & \text{if } P = M \vee P = \texttt{STOP} \vee P = Q \sqcap R \vee P = (\bot \underset{X}{\parallel} \bot) \\ & \quad \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \circlearrowleft R) \ \vee \ P = (\circlearrowleft Q \underset{X}{\parallel} \bot) \ \vee \ P = (\bot \underset{X}{\parallel} \circlearrowleft R) \\ & \quad \vee (P = (\circlearrowleft Q \underset{X}{\parallel} R) \wedge \texttt{FstEvs}(R) \subseteq X) \vee (P = (Q \underset{X}{\parallel} \circlearrowleft R) \wedge \texttt{FstEvs}(Q) \subseteq X) \\ & \quad \vee (P = Q \underset{X}{\parallel} R \wedge \texttt{FstEvs}(Q) \subseteq X \wedge \texttt{FstEvs}(R) \subseteq X \wedge \bigcap_{M \in \{Q,R\}} \texttt{FstEvs}(M) = \emptyset) \\ E & \text{otherwise, where } P = Q \underset{X}{\parallel} R \wedge E = (\texttt{FstEvs}(Q) \cup \texttt{FstEvs}(R)) \setminus \\ & \qquad\qquad (X \cap (\texttt{FstEvs}(Q) \setminus \texttt{FstEvs}(R) \cup \texttt{FstEvs}(R) \setminus \texttt{FstEvs}(Q))) \end{cases}$$

(STOP) Whenever this rule is applied, the subcomputation finishes because $\bot$ is put in the control, and this special constructor has no associated rule. A node with the `STOP` position is added to the graph.

We illustrate this semantics with a simple example.

Example 3. Consider again the specification in Example 1. Due to the choice operator, in this specification two different events can occur, namely `b` and `a`. Therefore, Algorithm 1 performs two iterations (one for each computation) to generate the final CSCFG. Figure 2(b) shows the CSCFG generated where white nodes were produced in the first iteration; and grey nodes were produced in the second iteration. For the interested reader, in [10] all computation steps executed by Algorithm 1 to obtain the CSCFG associated with the specification in Example 1 are explained in detail.

5 Correctness

In this section we state the correctness of the proposed algorithm by showing that (i) the graph produced by the algorithm for a CSP specification $\mathcal{S}$ is the CSCFG of $\mathcal{S}$; and (ii) the algorithm terminates, even if non-terminating computations exist for the specification $\mathcal{S}$.

Theorem 1 (Correctness). *Let $\mathcal{S}$ be a CSP specification and G the graph produced for $\mathcal{S}$ by Algorithm 1. Then, G is the CSCFG associated with $\mathcal{S}$.*

This theorem can be proved by showing first that each step performed with the standard semantics has an associated step in the instrumented semantics; and that the specification position of the expression in the control is added to the CSCFG as a new node which is properly inserted into the CSCFG. This can be proved by induction on the length of a derivation in the standard semantics. Then, it must be proved that the algorithm performs all possible computations. This can be done by showing that every non-deterministic step of the semantics is recorded in the stack with all possible rules that can be applied; and the

algorithm traverses the stack until all possibilities have been evaluated. The interesting case of the proof happens when the computation is infinite. In this case, the context of a process call must be repeated because the number of process calls is finite by definition. Therefore, in this case the proof must show that functions `LoopCheck` and `LoopControl` correctly finish the computation. The proof of this theorem can be found in [10].

Theorem 2 (Termination). *Let $\mathcal{S}$ be a CSP specification. Then, the execution of Algorithm 1 with $\mathcal{S}$ terminates.*

The proof of this theorem must ensure that all derivations of the instrumented semantics are finite, and that the number of derivations fired by the algorithm is also finite. This can be proved by showing that the stacks never grow infinitely, and they will eventually become empty after all computations have been explored. The proof of this theorem can be found in [10].

6 Conclusions

This work introduces an algorithm to build the CSCFG associated with a CSP specification. The algorithm uses an instrumentation of the standard CSP's operational semantics to explore all possible computations of a specification. The semantics is deterministic because the rule applied in every step is predetermined by the initial state and the information in the stack. Therefore, the algorithm can execute the semantics several times to iteratively explore all computations and hence, generate the whole CSCFG. The CSCFG is generated even for non-terminating specifications due to the use of a loop detection mechanism controlled by the semantics. This semantics is an interesting result because it can serve as a reference mark to prove properties such as completeness of static analyses based on the CSCFG. The way in which the semantics has been instrumented can be used for other similar purposes with slight modifications. For instance, the same design could be used to generate other graph representations of a computation such as Petri nets [11].

On the practical side, we have implemented a tool called *SOC* [8] which is able to automatically generate the CSCFG of a CSP specification. The CSCFG is later used for debugging and program simplification. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [2, 6], that shows the maturity and usefulness of this tool and of CSCFGs. The last release of *SOC* implements the algorithm described in this paper. However, in the implementation the algorithm is much more complex because it contains some improvements that significantly speed up the CSCFG construction. The most important improvement is to avoid repeated computations. This is done by: (i) state memorization: once a state already explored is reached the algorithm stops this computation and starts with another one; and (ii) skipping already performed computations: computations do not start from `MAIN`, they start from the next non-deterministic state in the execution (this is provided by the information of the stack).

The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

References

1. Brassel, B., Hanus, M., Huch, F., Vidal, G.: A Semantics for Tracing Declarative Multi-paradigm Programs. In: Moggi, E., Warren, D.S. (eds.) 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 179–190. ACM, New York, NY, USA (2004)
2. Butler, M., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heildeberg (2005)
3. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA (1985)
4. Kavi, K.M., Sheldon, F.T., Shirazi, B., Hurson, A.R.: Reliability Analysis of CSP Specifications using Petri Nets and Markov Processes. In: 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS'95), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)
5. Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)
6. Leuschel, M., Butler, M.: ProB: an Automated Analysis Toolset for the B Method. Journal of Software Tools for Technology Transfer. 10(2), 185–203 (2008)
7. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Static Slicing of CSP Specifications. In: Hanus, M. (ed.) 18th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'08), pp. 141–150. Technical report, DSIC-II/09/08, Universidad Politécnica de Valencia (July 2008)
8. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: SOC: a Slicer for CSP Specifications. In: Puebla, G., Vidal, G. (eds.) 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09), pp. 165–168. ACM, New York, NY, USA (2009)
9. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: The MEB and CEB Static Analysis for CSP Specifications. In: Hanus, M. (ed.) LOPSTR 2008, Revised Selected Papers. LNCS, vol. 5438, pp. 103–118. Springer, Heildeberg (2009)
10. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Semantics to Generate the Context-sensitive Synchronized Control-Flow Graph (extended). Technical report DSIC, Universidad Politécnica de Valencia. Accessible via `http://www.dsic.upv.es/~jsilva`, Valencia, Spain, June 2010.
11. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: Transforming Communicating Sequential Processes to Petri Nets. In: Topping, B.H.V., Adam, J.M., Pallarés, F.J., Bru, R., Romero, M.L. (eds.) Seventh Int'l Conf. on Engineering Computational Technology (ICECT'10). Civil-Comp Press, Stirlingshire, Scotland (to appear 2010)
12. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, London (1995)
13. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)

Mikiβ : A General GUI Library for Visualizing Proof Trees

– System Description and Demonstration –

Kanako Sakurai and Kenichi Asai

Ochanomizu University, Japan
{sakurai.kanako,asai}@is.ocha.ac.jp

Abstract. This paper describes and demonstrates Mikiβ, a general graphical user interface (GUI) library we are developing for visualizing proof trees. Using Mikiβ, one can construct a proof tree step by step by selecting a judgement and clicking a rule to apply, without worrying about rewriting various parts of a proof tree through uni cation, copying similar expressions for each judgement, or how much space is necessary to complete proof trees. To cope with many di erent kinds of proof trees, Mikiβ is designed to work with user-de ned judgements and inference rules. Although Mikiβ is still in its infancy, we have used it to visualize typing derivations for the simply-typed lambda calculus extended with delimited continuation constructs, system F, as well as logical proof trees for sequent calculus.

Key words: graphical user interface (GUI), proof tree, type system, lambda calculus, system F, sequent calculus, OCaml, LablTk

1 Introduction

A proof tree is used in various places. We write a proof tree to infer a type of an expression and to prove a logical formula, for example. It is also useful for educational purposes to understand the behavior of type systems and deduction systems. However, writing a proof tree by hand is not so straightforward. It is often di cult to predict how big the proof tree will become. We also need to copy similar expressions many times. Even worse, we have to rewrite almost whole the tree when a metavariable is instantiated to something else because of uni cation.

These problems could be avoided if we construct a graphical user interface (GUI) for writing proof trees. However, writing one GUI is not enough. We design new type systems for di erent purposes, and we want to visualize proof trees for these new systems, too. But those who design type systems want to concentrate on type systems and do not usually want to bother themselves about making GUI.

As an attempt to resolve this situation, we are developing a general GUI system, Mikiβ, for visualizing proof trees. To use Mikiβ, one is required to supply

de nition for judgements and inference rules in a speci c way. Then, Mikiβ uses the de nition to construct a GUI system for it. Mikiβ is still an on-going work and requires users to write quite a lot of things. However they are mostly about the inference system itself and not about GUI. Thus, we expect that Mikiβ users can concentrate on the inference system itself to obtain GUI for it.

After showing the overview of Mikiβ in Section 2, we show how to construct a GUI for the simply-typed lambda calculus as an example (Section 3). In Section 4, we show other systems we have implemented. We mention related work in Section 5 and the paper concludes in Section 6.

2 Overview of Mikiβ

The ultimate goal of the Mikiβ project is to build a system in which a GUI is constructed from the de nition of judgements and inference rules. Toward this goal, the current Mikiβ aims at separating GUI parts from the de nition of inference rules as much as possible, so that users need not care about GUI very much to construct a GUI system.

Mikiβ is being developed with LablTk in OCaml. It o ers a data structure for a tree and related functions as well as a GUI library to initialize a window and to register buttons. Since Mikiβ o ers only GUI-related functions, it does not o er functions regarding inference rules. Thus, users of Mikiβ need to understand the inference rules deeply and de ne them properly.

To build a GUI system using Mikiβ, one has to provide a syntax of judgements (together with a lexer and parer) and four functions for:

- extracting a GUI object identi er,
- generating a new metavariable,
- uni cation, and
- drawing expressions

for each user-de ned data type.

3 Using Mikiβ

In this section, we use the simply-typed lambda calculus to show how to construct a GUI with Mikiβ. The syntax and typing rules are shown in Fig. 1. The de nition is standard except for the inclusion of `(TWEAK)`. Usually, we avoid `(TWEAK)` by regarding a context Γ as a *set* of bindings. However, the precise speci cation would then require a de nition of sets. Here, we represent Γ as an ordered list of bindings and use `(TWEAK)` instead.

Note that variables appearing in typing rules (x, T, Γ, etc.) are metavariables. They are replaced with concrete values, when typing rules are applied to construct a proof tree.

Syntax			Typing	
t ::=		terms:		
	x	variables	$\dfrac{}{x : T, \Gamma \vdash x : T}$	(TVAR)
	λx:T.t	abstraction		
	t t	application	$\dfrac{x : T_1, \Gamma \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}$	(TABS)
T ::=		types:		
	B	base type		
	T → T	type of functions	$\dfrac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(TAPP)
Γ ::=		contexts:		
	∅	empty context		
	x:T, Γ	variable binding	$\dfrac{\Gamma \vdash t : T}{x : T_1, \Gamma \vdash t : T}$	(TWEAK)
j ::=	Γ ⊢ t : T	judgements:		

Fig. 1. The simply-typed lambda calculus

3.1 Definition of Syntax

Mikiβ o ers a data type `tree_t` of proof trees to be visualized in GUI.

```
type tree_t =
    Axiom of judge_t
  | Infer of judge_t * tree_t list (* assumptions *)
```

The type `judge_t` is a type for judgements to be de ned by Mikiβ users. For the simply-typed lambda calculus, de nition of `judge_t` becomes as shown in Fig. 2 in the typewriter font. It is a straightforward transcription of Fig. 1 into OCaml, except that variables are assigned an independent type rather than using a raw string. This is because we want to assign single GUI object to the semantically same variables.

```
type 'a meta_t =
    string ref * 'a option ref * id_t

type var_t =
    V of string * id_t
  | MetaVar of var_t meta_t

type term_t =
    Var of var_t * id_t
  | Abs of var_t * term_t * id_t
  | App of term_t * term_t * id_t
  | MetaTerm of term_t meta_t
```

```
type type_t =
    TVar of string * id_t
  | Fun of type_t * type_t * id_t
  | MetaType of type_t meta_t

type env_t =
    Empty of id_t
  | Cons of
      var_t * type_t * env_t * id_t
  | MetaEnv of env_t meta_t

type judge_t =
  Judge of env_t * term_t * type_t * id_t
```

Fig. 2. The syntax de nition of simply-typed lambda calculus in Mikiβ

For this type de nition, we need to add two things to use it in Mikiβ. They are shown in italic in Fig. 2.

GUI object identifiers Each constructor needs a GUI object identi er `id_t`. It enables Mikiβ to relate OCaml data with GUI objects on a display. Users do not need to know how the identi er is de ned internally.

Metavariables In the typing rules of Fig. 1, each variable is regarded as a metavariable. To represent a metavariable, we add it to the type de nition in Fig. 2. For example, a metavariable for types is de ned as follows:

```
MetaType of type_t meta_t
```

where `'a meta_t` is de ned as follows:

```
type 'a meta_t = string ref * 'a option ref * id_t
```

A metavariable has a name and a pointer to another data of the same type. When a metavariable is instantiated to a concrete value, the pointer to the value is set in the metavariable. Mikiβ will o er functions operating `'a meta_t`.

Besides the above two additions, users are currently requested to write two more functions as well as a lexer and a parser to convert concrete syntax into the above data type. One is a function to extract an identi er from data, and the other is a function to generate a new metavariable. For example, we de ne the two functions for `type_t` data type as follows. Here, `new_meta` is a function to generate a new value of type `'a meta_t` using a given string.

```
(* Extracting id_t *)
let get_type_id tp = match tp with
  TVar(_, id) | Fun(_, _, id) | MetaType(_, _, id) -> id

(* Generating a new metavariable *)
let new_type () = MetaType(new_meta "T")
```

3.2 Definition of Inference Rules

Thanks to the introduction of metavariables, users can directly de ne inference rules as a tree structure. For example, `(TABS)` is de ned as follows. Here, `NEW` is a prede ned dummy identi er in Mikiβ.

```
let t_abs() =
  (* generating new metavaiables *)
  let e = new_env() in    (* context *)
  let tm = new_term() in  (* term *)
  let x = new_var() in    (* variable *)
  let tp1 = new_type() in (* type *)
  let tp2 = new_type() in
  (* define the inference rule using metavariables *)
  Infer(Judge(e, Abs(x, tp1, tm, NEW), Fun(tp1, tp2, NEW), NEW),
        [Infer(Judge(Cons(x, tp1, e, NEW), tm, tp2, NEW), [])])
```

Whenever we need to supply a value of type `id_t`, we use `NEW`. Assumptions of the inference rules are encoded as a list of `Infer` nodes with an empty assumption list. It indicates that we need to nd their proofs before completing a proof tree. The other inference rules are written similarly. Inference rules de ned in this way are registered to `infer_button_list`, which is used to create a button for each inference rule on a display.

```
let infer_button_list = [("TVAR", t_var); ("TABS", t_abs);
                         ("TAPP", t_app); ("TWEAK", t_weak)]
```

3.3 Definition of Unification and Application of Inference Rules

A proof tree is constructed in three steps:

1. selection of a judgement and an inference rule,
2. uni cation of the judgement and the conclusion of the inference rule, and
3. substitution of the judgement with the instantiated inference rule.

For example, application of `(TABS)` to $a : int \vdash (\lambda x : T_1.x)\ a : T_2$ (written in bold font) is depicted in Fig. 3.

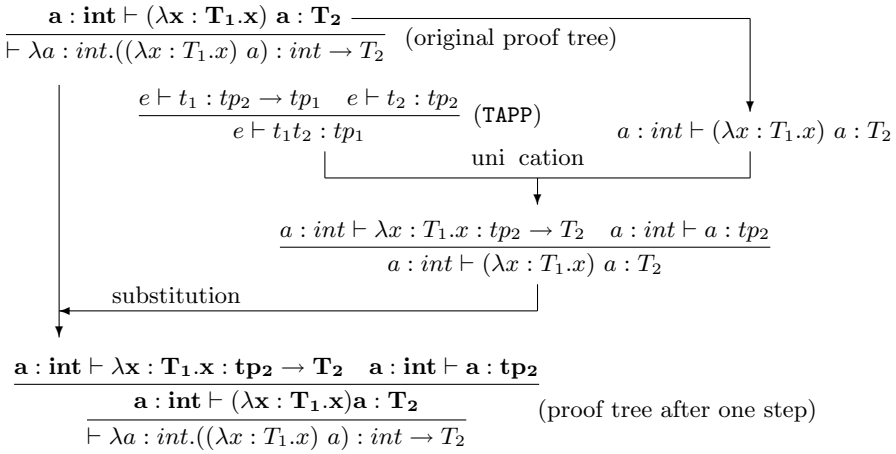


Fig. 3. Applying an inference rule with uni cation

Among the three steps, (1) and (3) are taken care of by Mikiβ. For (2), users are currently requested to write a uni cation function for the user-de ned data. For example, de ne a function `deref_type` that dereferences a metavariable:

```
let rec deref_type tp = match tp with
    MetaType(_, {contents = Some(tp')}, _) -> deref_type tp'
  | _ -> tp
```

Then, the uni cation function for types can be de ned as follows:

```
let rec unify_type tp1 tp2 =
  match (deref_type tp1, deref_type tp2) with
  (* case1 : meta type and meta type *)
      (MetaType(_, op1, _), (MetaType(_, op2, _) as tp2')) ->
       if op1 != op2
       then op1 := Some(tp2')
  (* case2 : meta type and concrete type *)
    | (MetaType(s, op, _), tp) | (tp, MetaType(s, op, _)) ->
       if occur s tp  (* check if s occurs in tp *)
       then raise Unify_Error
       else op := Some(tp)
  (* case3 : concrete type and concrete type *)
    | (Fun(tp1, tp2, _),Fun(tp3, tp4, _)) ->
       unify_type tp1 tp3; unify_type tp2 tp4
    | (TVar(tv1, _), TVar(tv2, _)) -> unify_tvar tv1 tv2
    | _ -> raise Unify_Error
```

Since uni cation functions have a uniform structure, we expect to generate them from the data de nition in the future.

3.4 Definition of Drawing

Since the object layout di ers from one system to another, users specify how data is layout on a display in Mikiβ. For this purpose, Mikiβ o ers a few drawing functions.

- `create_str : string → id_t`
  To create and layout a string object on a display.
- `combineH : id_t list → id_t`
  To layout objects horizontally and make them one object.
- `combineV : id_t → id_t → id_t`
  To layout two objects vertically and make them one object. (In case of an expression which has only horizontal layout, this function is not necessary.)

Using these functions, users specify a drawing function in a way GUI identi ers are combined through `combineH` and `combineV`. For example, a function drawing `type_t` data type is as follows:

```
let rec draw_type tp = match (deref_type tp) with
    TVar(tv, _) ->
    let tv' = draw_tvar tv in
    TVar(tv', get_tvar_id tv')
  | Fun(tp1, tp2, _) ->
    let tp1' = draw_type tp1 in
```

```
    let tp2' = draw_type tp2 in
    Fun(tp1',tp2',
        combineH [get_type_id tp1'; create_str " -> ";
                  get_type_id tp2'])
  | MetaType(str, op , _) -> MetaType(str, op, create_str !str)
```

3.5 Main and Supporting Functions

Up to here, we have shown how to de ne inference rules to be used in Mikiβ. It is mostly independent of GUI and users of Mikiβ can concentrate on the speci cation of judgements and inference rules.

To build a GUI system, the de nition of `judge_t` and other helper functions de ned by users are packaged into a module and passed to the tree functor provided by Mikiβ. The tree functor contains various GUI-related functions, such as:

Function Registration At start up, each function in `infer_button_list` is assigned a button on the display.

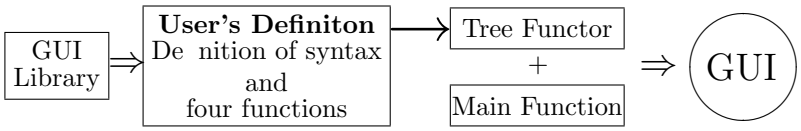
Object Selection When a mouse is clicked, the clicked object is searched and its GUI object identi er is returned. To select a user-de ned data, one currently has to specify a fold-like function for the data.

Inference Rule Application When a button is pressed, the selected function (inference rule) is used to grow a proof tree.

Replacing Metavariable Names A name of a metavariable can be changed consistently by right-clicking a mouse and inserting a new name into a text box.

Undo Because uni cation is implemented as side-e ects, it is not straightforward to undo the inference. Mikiβ supports undo by recording all the user interaction and redo it from the original judgement.

The complete GUI system is then obtained by linking the tree functor with the main function that creates an empty window, initializes it, and launches the event loop. The following diagram shows the overall structure of the GUI system:



To use the GUI system, we take the following steps:

1. Enter an expression you want to infer to a text box at the top, and press the Go button (Fig. 4).
2. Select a target judgement with a mouse click (Fig. 5) and press a button that you want to apply to the target judgement (Fig. 6). In case the selected inference rule is inapplicable, nothing will happen.
3. Repeat the second step until the proof tree is complete (Fig. 7).

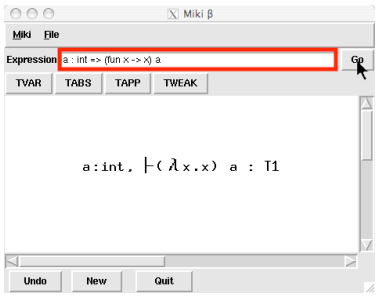


Fig. 4. Enter a judgement

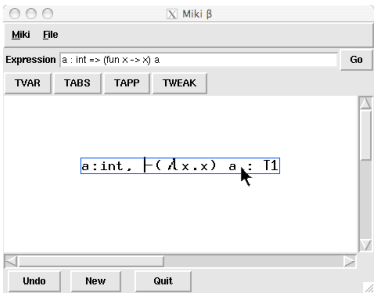


Fig. 5. Select a target judgement

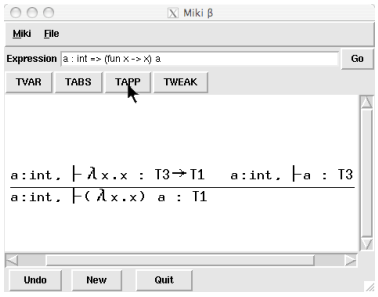


Fig. 6. Apply an inference rule

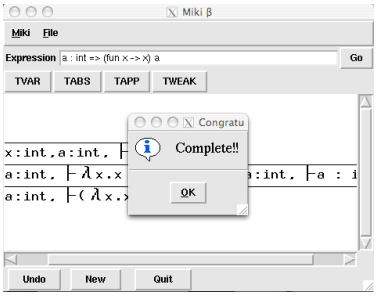


Fig. 7. Inference nished

4 Examples

In this section, we show several experiences of using Mikiβ.

4.1 Type system for shift and reset

The control operators, shift and reset, are introduced by Danvy and Filinski [2] to capture the current continuation up to an enclosing delimiter. Their monomorphic type system is a generalization of the one for lambda calculus and mentions the *answer type* of the enclosing context. The judgement has the following form:

$$\Gamma; \alpha \vdash e : \tau; \beta$$

It reads: under a type environment Γ, a term e has a type τ, and the evaluation of e changes the answer type from α to β. With this type system, the rule for application, for example, is as follows:

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \quad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \delta}\ (\mathtt{app})$$

Although conceptually simple, it is extremely hard to keep track of all these types. With Mikiβ, one can simply input all the inference rules and obtain a GUI system for it that takes care of all the types. We could actually give the de nition in less than an hour.

4.2 System F

System F adds two inference rules to the simply-typed lambda calculus [5]:

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}\ (\texttt{T-TABS}) \qquad \frac{\Gamma \vdash t : \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

Here, we notice that `(T-TAPP)` uses a type substitution at conclusion. It means that to use `(T-TAPP)` to grow a proof tree, we need to unify the type $T_2[X \mapsto T_1]$ with the type of the current judgement. However, without knowing the structure of T_2, it is impossible in general to perform uni cation at this stage. It becomes possible only when T_2 is instantiated to a concrete type.

To remedy this situation, we change the inference rules `(TAPP)` and `(T-TAPP)` so that all the leaf judgements have only metavariables as types:

$$\frac{\Gamma \vdash t_1 : T \quad T = T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}\ (\texttt{TAPP})$$

$$\frac{\Gamma \vdash t : T \quad T = \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

We introduced a new judgement of the form $T_1 = T_2$. It enables us to defer uni cation. The new judgement can be written as an axiom in Mikiβ:

```
type judge_t =
    Judge of env_t * term_t * type_t * id_t
  | Equal of type_t * type_t * id_t          (* new judgement *)

let t_equal() =
  let tp = new_type() in (* generate a metavariable *)
  Axiom(Equal(tp, tp, NEW), NEW)
```

At the time of writing, however, we still have to modify the Mikiβ system itself to accommodate the application of substitutions. We are currently gathering what operations are needed to express various inference systems.

4.3 Sequent Calculus

It is straightforward to specify the inference rules for sequent calculus. However, judgements in sequent calculus have additional exibility. For example, in the `(∨L)` rule below:

$$\frac{\Gamma, A \vdash \Delta \quad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi}\ (\vee\texttt{L})$$

formulas in the antecedent (Γ, Σ) and the succedent (Δ, Π) are separated at an arbitrary place to obtain the two premises. To support such arbitrary separation, we had to introduce into Mikiβ a mechanism to insert a cursor in the formulas and to choose a separation point. Since it is expected that such mechanism is needed in other systems (such as parsing in the natural language processing), we hope to support it in Mikiβ in the future.

More generally, we hope to include a general way to incorporate arbitrarily complex programmable operations into Mikiβ.

5 Related Work

Although the goal is quite different, visualising proof trees has much in common with theorem proving. Actually, growing a proof tree can be regarded as decomposing and proving goals in theorem proving. A theorem prover typically implements various kinds of automation, such as tactics found in Coq [1]. It is an interesting challenge to incorporate such automation in Mikiβ. It would then become possible to apply `(TWEAK)` automatically before `(TVAR)`.

Geuvers and Jojgov [4] study incomplete proof trees with metavariables and discuss their semantics. Although they do not mention GUI, their result could be regarded as a theoretical basis of our work.

PLT Redex [3] is a domain specific language to specify operational semantics. Given a definition of operational semantics, PLT Redex offers (among other features) graphical output of reduction of terms.

6 Conclusion and Future Work

In this paper, we have described Mikiβ, our on-going project to visualize proof trees. Given definition for judgements and inference rules, Mikiβ produces a GUI system for it. Although users are required to write certain amount of definition in OCaml, Mikiβ mostly takes care of GUI parts and users can concentrate on the formalization of inference rules. Our experience of using Mikiβ shows that it is at least useful to visualize relatively simple systems.

To cope with more complex systems, we still need to enhance GUI parts of Mikiβ. We are currently implementing various systems in Mikiβ to extract what features we need to specify them in a GUI-independent way. Through this process, we hope to make Mikiβ a more general and useful tool for visualizing proof trees.

Acknowledgment

We received many useful comments from the anonymous reviewers.

References

1. Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, EATCS Series, Berlin: Springer (2004).
2. Olivier Danvy, and Andrzej Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).
3. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, *Semantics Engineering with PLT Redex*, Cambridge: MIT Press (2009).
4. Herman Geuvers and Gueorgui I. Jojgov. "Open Proofs and Open Terms: A Basis for Interactive Logic," In Proc. of CSL 2002. LNCS 2471, pp.537–552.
5. Benjamin C. Pierce, *Types and Programming Languages*, Cambridge: MIT Press (2002).

On Inductive Proofs by Extended Unfold/fold Transformation Rules

Hirohisa Seki *

Dept. of Computer Science, Nagoya Inst. of Technology,
Showa-ku, Nagoya, 466-8555 Japan
seki@nitech.ac.jp

Abstract. We present an extended framework for unfold/fold transformation of strati ed logic programs. We extend our previous transformation system which contains, among others, negative unfolding with a new application condition, by introducing *extended* negative unfolding. The application of extended negative unfolding allows an unfolding clause to have existential variables in its body, while conventional negative unfolding does not. Moreover, we complement our previous transformation system with another two transformation rules, simultaneous folding and negative folding. The correctness of the extended transformation system with these three rules is shown in the sense of the perfect model semantics. We also examine the use of simultaneous folding for proving properties of programs in literature. We show by examples that our unfold/fold transformation system with extended negative unfolding, when used together with Lloyd-Topor transformation, can be used for verifying recursively de ned properties of programs. The example illustrated in the paper shows that, although such properties are provable by using simultaneous folding, our inductive proof method can solve them in a simpler and more intuitive manner.

1 Introduction

Since the seminal work by Tamaki and Sato [13], a vast number of papers on unfold/fold transformation rules for logic programs have been reported (see an excellent survey [5] and references therein). Among them, *negative unfolding* is a transformation rule, which applies unfolding to a negative literal in the body of a clause. When used together with the other transformation rules, negative unfolding is shown to play an important role in program transformation, construction (e.g., [2]) and veri cation (e.g., [6], [10]). In [12], we proposed a framework for unfold/fold transformation of strati ed programs which extends previous frameworks in the literature, and identi ed a new application condition on negative unfolding which ensures the preservation of our intended semantics of a program. One of the motivations of this paper is to further study negative unfolding proposed in the literature, and propose an extension of our previous

* This work was partially supported by JSPS Grant-in-Aid for Scienti c Research (C) 21500136.

transformation system, especially for the purpose of using the transformation rules for inductive proofs.

Negative unfolding in our previous transformation system [12] is proposed by Pettorossi and Proietti for locally strati ed programs [6] (PP-negative unfolding for short). One of the restrictions on applying PP-negative unfolding is that it does not allow an unfolding clause to have existential variables in its body. A variable occurring in the body of a clause is called *existential*, if it does not appear in the head of the clause. However, such a clause often appears in many applications; for example, a simple reachability property (which speci es that a state in which proposition p holds is reachable) in concurrent systems can be encoded as follows [10]:

$$ef(X) \leftarrow p(X)$$
$$ef(X) \leftarrow trans(X, Y), ef(Y),$$

where the predicate $trans$ represents the transition relation of the system considered, and $p(X)$ is true if the proposition p holds in state X. Note that the second clause of the de nition of ef has an existential variable Y in its body.

In this paper, we therefore propose an extension of PP-negative unfolding, called *extended* negative unfolding (*ENU* for short). In ENU, the above-mentioned restriction is eliminated, thereby making an unfolded clause with a negative literal in its body amenable to subsequent transformations, which will hopefully lead to a successful inductive proof. We show such an example in Sect. 3.1. Moreover, we complement our previous transformation system with another two transformation rules, simultaneous folding and negative folding, and show the correctness of the extended transformation system in the sense of the perfect model semantics.

We then examine the use of simultaneous folding in inductive proofs in the literature. We show by examples that our transformation system with ENU, when used together with Lloyd-Topor transformation [4], can inductively prove properties of programs which were originally proved by using simultaneous folding in the literature [10]. We show that our inductive proof method can prove such properties in a simpler and more intuitive manner.

The organization of this paper is as follows. In Section 2, after giving some preliminaries, we recall our basic unfold/fold transformation transformation system for strati ed programs. In Section 3, we introduce into the system three new transformation rules: extended negative unfolding, simultaneous folding and negative folding. We then explain by an example how our inductive proof method via unfold/fold transformations inductively proves properties of programs. The correctness of the extended transformation system is shown in Section 3.3. Finally, we discuss about the related work and give a summary of this work in Section 4. [1]

Throughout this paper, we assume that the reader is familiar with the basic concepts of logic programming, which are found in [4, 8].

[1] Due to space constraints, we omit most proofs and some details, which will appear in the full paper.

2 A Framework for Unfold/Fold Transformation

In this section, we review a framework for our basic unfold/fold transformation of strati ed programs [12]. Although we con ne the framework to *stratified* programs here for simplicity, it is possible to extend it to *locally stratified constraint* programs as in [2].

2.1 Transformation Rules

We rst explain our transformation rules here, and then recall some conditions imposed on the transformation rules which are necessary for correctness of transformation.

We divide the set of the predicate symbols appearing in a program into two disjoint sets: *primitive* predicates and *non-primitive* predicates.[2] This partition of the predicate symbols is arbitrary and it depends on an application area of the user. We call an atom (a literal) with primitive predicate symbol a *primitive atom* (*primitive literal*), respectively. A clause with primitive (resp., non-primitive) head atom is called *primitive* (resp., *non-primitive*).

The set of all clauses in program P with the same predicate symbol p in the head is called the de nition of p and denoted by $Def(p, P)$. The predicate symbol of the head of a clause is called the *head predicate* of the clause. The head and the body of a clause C are denoted by $hd(C)$ and $bd(C)$, respectively.

Given a clause C, a variable in $bd(C)$ is said to be *existential*, if it does not appear in $hd(C)$. The other variables in C are called *free* variables. The conventional representation of a clause $C : H \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$, where $A_1, \ldots, A_n$ are a conjunction of literals, is equivalent to the formula: $H \leftarrow \exists X_1 \ldots X_m(A_1, \ldots, A_n)$, when the existential variables $X_1 \ldots X_m$ $(m \geq 0)$ in C are made explicit. In general, an existentially quanti ed conjunction M of the form: $\exists X_1 \ldots X_m(A_1, \ldots, A_n)$ $(m \geq 0, n \geq 0)$ is called a *molecule*, where $X_1 \ldots X_m$ are distinct variables called *existential variables* in M. We denote by $Vf(M)$ the set of free variables in M. A molecule without free variables is said to be *closed*, while a molecule without free variables nor existential variables is said to be *ground*. Two molecules M and N are considered to be identical, denoted by $M = N$, if M is obtained from N through permutation of conjuncts and renaming of existential variables. When more than one molecule is involved, they are assumed to have disjoint sets of variables, unless otherwise stated.

A *stratification* is a total function σ from the set $Pred(P)$ of all predicate symbols appearing in P to the set N of natural numbers. It is extended to a function from the set of literals to N in such a way that, for a positive literal A, $\sigma(A) = i$, where i is the strati cation of predicate symbol of A. We assume that σ satis es the following: For every primitive atom A, $\sigma(A) = 0$. For a positive literal A, $\sigma(\neg A) = \sigma(A) + 1$. For a conjunction of literals $G = l_1, \ldots, l_k$ $(k \geq 0)$, $\sigma(G) = 0$ if $k = 0$ and $\sigma(G) = \max\{\sigma(l_i) : i = 1, \ldots, k\}$ otherwise.

[2] In [6], primitive (non-primitive) predicates are called as *basic* (*non-basic*), respectively.

For a strati ed program P, we denote its perfect model [8] by $M(P)$. Let A be a ground atom true in $M(P)$. A nite successful ground SLS-derivation T with its root $\leftarrow A$ is called a *proof* of A by P, and we say that A has a proof T by P. The de nition of proof is extended from a ground atom to a conjunction of ground literals, i.e., a ground molecule, in a straightforward way.

In our framework, we assume that an initial program, from which an unfold/fold transformation sequence starts, has the structure speci ed in the following de nition.[3]

Definition 1. Initial Program Condition

Let P_0 be a program, divided into two disjoint sets of clauses, P_{pr} and P_{np}, where P_{pr} (P_{np}) is the set of primitive (non-primitive) clauses in P_0, respectively. Then, P_0 satis es the *initial program condition*, if the following conditions hold:

1. No non-primitive predicate appears in P_{pr}.
2. P_0 is a *stratified* program, with a strati cation σ, called the *initial stratification*. Moreover, for every non-primitive predicate symbol p, $\sigma(p)$ is de ned by $\sigma(p) = \max\{\sigma(bd(C)) \mid C \in Def(p, P_0)\}$.
3. There exists a total function $level : Pred(P_0) \mapsto \{0, 1, \ldots, I\}$, where I is a positive integer called the *maximum level* of the program. For every primitive (resp. non-primitive) predicate symbol p, $level(p) = 0$ (resp., $1 \leq level(p) \leq I$). We de ne the *level of an atom* (or *literal*) A, denoted by $level(A)$, to be the level of its predicate symbol, and the *level of a clause* C to be the level of its head. Then, for each clause $C \in P_0$, it holds that $level(hd(C)) \geq level(A)$ for every *positive* literal A in the body of C. □

Remark 1. The above de nition follows the generalized framework in [14], thereby eliminating some restrictions in the original one [13]. In [13], the number of levels is two; each predicate in an initial program is classi ed as either *old* or *new*. The de nition of a new predicate consists of a single clause whose body contains positive literals with old predicates only, thus a recursive de nition of a new predicate is not allowed. Moreover, it has no primitive predicates. □

Example 1. Adopted from [10]. Suppose that P_0 consists of the following clauses. Predicate *gen* generates a string consisting of only 0's, while the *test* predicate checks whether a given list can be transformed (through a nite number of applications of *trans*) into a string consisting only 1's.

$$
\begin{array}{lll}
thm(X) & \leftarrow gen(X), test(X) & \\
gen([]) & \leftarrow & \\
gen([0|X]) & \leftarrow gen(X) & \\
test(X) & \leftarrow canon(X) & \\
test(X) & \leftarrow trans(X, Y), test(Y) & \\
canon([]) & \leftarrow & \\
canon([1|X]) & \leftarrow canon(X) & \\
trans([0|X], [1|X]) & \leftarrow & \\
trans([1|T], [1|T1]) & \leftarrow trans(T, T1) &
\end{array}
$$

[3] When we say "an initial program P_0" hereafter, P_0 is assumed to satisfy the initial program condition in Def. 1.

We note that the second clause of the de nition of predicate *test* has an existential variable Y in its body.

It is a user's choice to partition the predicate symbols in $Pred(P_0)$ into either primitive or non-primitive ones. We assume here that predicate symbols *test*, *canon* and *trans* are primitive, while the other predicate symbols are non-primitive. We assign each non-primitive predicate symbol its level as follows: $level(gen) = 1$, $level(thm) = 2$. Since P_0 is a de nite program, we assume that the stratum of each non-primitive predicate is 1. Then, it is easy to see that P_0 satis es the initial program condition.

We consider another partition of the predicate symbols and an assignment of *level* in Example 4. □

We now recall the de nitions of our basic transformation rules.

Positive Unfolding. Let C be a renamed apart clause in a strati ed program P of the form: $H \leftarrow G_1, A, G_2$, where A is an atom, and G_1 and G_2 are (possibly empty) conjunctions of literals. Let $D_1, \ldots, D_k$ with $k \geq 0$, be all clauses of program P, such that A is uni able with $hd(D_1), \ldots, hd(D_k)$, with most general uni ers (m. g. u.) $\theta_1, \ldots, \theta_k$, respectively.
By (*positive*) *unfolding* C w.r.t. A, we derive from P the new program P' by replacing C by $C_1, \ldots, C_k$, where C_i is the clause $(H \leftarrow G_1, bd(D_i), G_2)\theta_i$, for $i = 1, \ldots, k$.
In particular, if $k = 0$, i.e., there exists no clause in P whose head is uni able with A, then we derive from P the new program P' by deleting clause C.
Negative Unfolding. [4] Let C be a renamed apart clause in a strati ed program P of the form: $H \leftarrow G_1, \neg A, G_2$, where A is an atom, and G_1 and G_2 are (possibly empty) conjunctions of literals. Let $D_1, \ldots, D_k$ with $k \geq 0$, be all clauses of program P, such that A is uni able with $hd(D_1), \ldots, hd(D_k)$, with most general uni ers $\theta_1, \ldots, \theta_k$, respectively. Assume that:
(PP1) $A = hd(D_1)\theta_1 = \ldots = hd(D_k)\theta_k$, that is, for each i $(1 \leq i \leq k)$, A is an instance of $hd(D_i)$,
(PP2) for each i $(1 \leq i \leq k)$, D_i has no existential variables, and
(PP3) from $\neg(bd(D_1)\theta_1 \vee \ldots \vee bd(D_k)\theta_k)$, we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of conjunctions of literals, with $r \geq 0$, by rst pushing $\neg$ inside and then pushing $\vee$ outside.

By *negative unfolding* w.r.t. $\neg A$, we derive from P the new program P' by replacing C by $C_1, \ldots, C_r$, where C_i is the clause $H \leftarrow G_1, Q_i, G_2$, for $i = 1, \ldots, r$.
In particular: (i) if $k = 0$, i.e., there exists no clause in P whose head is uni able with A, then we derive from P the new program P' by deleting $\neg A$ from the body of clause C, and (ii) if for some i $(1 \leq i \leq k)$, $bd(D_i) = true$, i.e., D_i is a unit clause, then we derive from P the new program P' by deleting clause C.
Folding Let P_0 be an initial program and A be an atom. A molecule M is said to be a P-*expansion* of A (by a clause D) if there is a clause D in P_0 and a substitution θ of free variables of $hd(D)$ such that $hd(D)\theta = A$ and $bd(D)\theta = M$.

[4] The following de nition of negative unfolding is due to Pettorossi and Proietti [6].

Let C be a clause in P of the form: $H \leftarrow \exists X_1 \ldots X_n(M, N)$, where M and N are molecules, and $X_1 \ldots X_n$ are some free variables in M. If M is a P-expansion of A (by a renamed apart clause D), the result of *folding* C w.r.t. M *by* P_0 is the clause γ: $H \leftarrow \exists X_1 \ldots X_n(A, N)$. By applying *folding* to C w.r.t. M using clause D, we derive from P the new program $P' = (P \setminus \{C\}) \cup \{\gamma\}$.
The clause C is called the *folded clause* and D the *folding clause* (or *folder clause*).
The folding operation is said to be *reversible* if M is the only P-expansion of A in the above de nition.
Replacement Rule A *replacement rule* R is a pair $M_1 \Rightarrow M_2$ of molecules, such that $Vf(M_1) \supseteq Vf(M_2)$, where $Vf(M_i)$ is the set of free variables in M_i $(1 \leq i \leq 2)$. Let C be a clause of the form: $A \leftarrow M$. Assume that there is a substitution θ of free variables of M_1 such that M is of the form: $\exists X_1 \ldots X_n(M_1\theta, N)$ for some molecule N and some variables $X_1 \ldots X_n$ $(n \geq 0)$ in $Vf(M_1\theta)$. Then, the result of *applying* R to $M_1\theta$ in C is the clause γ: $A \leftarrow \exists X_1 \ldots X_n(M_2\theta, N)$. By applying *replacement rule* R to C w.r.t. $M_1\theta$, we derive from P the new program $P' = (P \setminus \{C\}) \cup \{\gamma\}$.
A replacement rule $M_1 \Rightarrow M_2$ is said to be *correct* w.r.t. an initial program P_0, if, for every ground substitution θ of free variables in M_1 and M_2, it holds that $M_1\theta$ has a proof by P_0 i $M_2\theta$ has a proof by P_0. □

Remark 2. We note that we do not explicitly consider *definition introduction* as a transformation rule, which allows us to de ne a new predicate in terms of old (i.e., previously de ned) predicates. The reason is that new predicates introduced in the course of transformation can be assumed to be present in an initial program from scratch as is done in [13]. □

We can now de ne a transformation sequence as follows:

Definition 2. Transformation Sequence
Let P_0 be an initial program (thus satisfying the conditions in Def.1), and $\mathcal{R}$ be a set of replacement rules correct w.r.t. P_0. A sequence of programs $P_0, \ldots, P_n$ is said to be a *transformation sequence* with the input $(P_0, \mathcal{R})$, if each P_i $(n \geq i \geq 1)$ is obtained from P_{i-1} by applying to a *non-primitive* clause in P_{i-1} one of the following transformation rules: (i) positive unfolding, (ii) negative unfolding, (iii) *reversible* folding *by* P_0, and (iv) some replacement rule in $\mathcal{R}$. □

We note that every primitive clause in P_0 remains *untransformed* at any step in a transformation sequence.

Example 2. (Continued from Example 1.) Let C_0 be the single clause de ning predicate *thm* in P_0. After applying positive unfolding several times to C_0 and its derived clauses, we obtain the following:
C_1 : $thm([]) \quad \leftarrow$
C_2 : $thm([0|X]) \leftarrow gen(X), canon(X)$
C_3 : $thm([0|X]) \leftarrow gen(X), trans(X, Y), test([1|Y])$
Next,we apply to C_3 replacement rule R: $test([1|Y]) \Rightarrow test(Y)$, noting that, for every ground substitution θ of free variable Y, it holds that $test([1|Y])\theta$ has a proof by P_0 i $test(Y)\theta$ has a proof by P_0. Then, we obtain the following clause:

$C_4 : \ thm([0|X]) \leftarrow gen(X), trans(X,Y), test(Y).$

Let $P_1 = P_0 \setminus \{C_0\} \cup \{C_1, C_2, C_4\}$.

We discuss again the correctness of the above application of replacement rule R in Example 3 and Example 4. □

2.2 Application Conditions of Transformation Rules

In the following, we will brie y explain the conditions imposed on the transformation rules to preserve the perfect model semantics of a program. The conditions, which are summarized in Table 1, are intended for the rules to satisfy the following two properties: (i) the preservation of the initial strati cation σ of an initial program P_0, and (ii) the preservation of an invariant of the size (according to a suitable measure μ) of the proofs[5] of an atom true in P_0.

The following de nition of the well-founded measure μ is a natural extension of that in [14], where μ is de ned in terms of an SLD-derivation.

Definition 3. Weight-Tuple, Well-founded Measure μ

Let P_0 be an initial program with the maximum level I and A be a ground atom true in $M(P_0)$. Let T be a proof of A by P_0, and let w_i $(1 \leq i \leq I)$ be the number of selected non-primitive positive literals of T with level i. Then, the *weight-tuple* of T is an I-tuple $\langle w_1, \ldots, w_I \rangle$.

The *well-founded measure* $\mu(A)$ of A is de ned by $\mu(A) := \min\{w \mid w$ is the weight-tuple of a proof of $A\}$, where $\min S$ is the minimum of set S under the lexicographic ordering[6] over N^I, and N is the set of natural numbers.

For a ground molecule L, $\mu(L)$ is de ned similarly. For a *closed* molecule M, $\mu(M) := \min\{w \mid w$ is the weight-tuple of a proof of a ground instance of $M\}$. □

Note that the above de ned measure μ is *well-founded* over the set of ground molecules which have proofs by P_0. By de nition, for a ground primitive atom A true in $M(P_0)$, $\mu(A) = \langle 0, \ldots, 0 \rangle = \mathbf{0}$ (I-tuple).

To specify the condition of folding, we need a notion of a *descent level* of a clause, originally due to Tamaki-Sato [14] for de nite programs. We recall that each clause $C \in P_0$ is assigned its level, $level(C)$ (Def. 1).

Definition 4. Descent Level of a Clause

Let C be a clause appearing in a transformation sequence starting from an initial program P_0. The *descent level* of C, denoted by $dl(C)$, is de ned inductively as follows:

1. If C is in P_0, $dl(C) = level(C)$, where $level(C)$ is the level of C.
2. If C is rst introduced as the result of applying positive unfolding to some clause C' in P_i $(0 \leq i)$ w.r.t. a positive literal A in C', then

$$dl(C) = \begin{cases} dl(C') & \text{if } A \text{ is primitive,} \\ \min\{dl(C'), level(A)\} & \text{otherwise} \end{cases}$$

[5] Recall that, by a proof, we mean a nite successful ground SLS-derivation. See 2.1.

[6] We use the inequality signs $>$, $\geq$ and so on to represent this lexicographic ordering.

Table 1. Conditions Imposed on the Transformations Rules [12]: The application conditions for extended negative unfolding, simultaneous folding and negative folding are newly given in this paper.

transformation rule	preservation of σ	preservation of μ-completeness
de nition (init. program P_0) (Def. 1)	$\exists$ init. strati cation σ	$\exists\, level : Pred(P_0) \to N$ s.t. $P_0 \ni \forall\, C : H \leftarrow L_1, \ldots, L_k$, $level(H) \geq level(L_i)$, if L_i is pos.
pos. unfolding	–	–
neg. unfolding	–	μ-consistent
folding	σ-consistent	TS-folding condition
replacement $M_1 \Rightarrow M_2$	σ-consistent $\sigma(M_1) \geq \sigma(M_2)$	μ-consistent $\mu(M_1) \geq \mu(M_2)$
extended neg. unfolding	–	–
simultaneous folding	σ-consistent	TS-folding condition
negative folding	–	–

3. If C is rst introduced as the result of applying either negative unfolding, folding, or a replacement rule to some clause C', then $dl(C) = dl(C')$. □

A Condition on Negative Unfolding. The application of negative unfolding is said to be *consistent* with μ (or μ-consistent), if it does not increase the *positive* occurrences of a non-primitive literal in the body of any derived clause. That is, in the de nition of negative unfolding (PP3), every positive literal (if any) in Q_i is *primitive*, for $i = 1, \ldots, r$.

Conditions on Folding Suppose that reversible folding rule *by* P_0 with folding clause D is applied to folded clause C. Then, the application of folding is said to be *consistent with* σ (σ*-consistent* for short), if the stratum of head predicate of D is less than or equal to that of the head of C, i.e., $\sigma(hd(C)) \geq \sigma(hd(D))$. Moreover, the application of folding is said to satisfy *TS-folding condition*, if the descent level of C is *smaller* than the level of D.

Conditions on Replacement Rules Let R be a replacement rule of the form $M_1 \Rightarrow M_2$, which is correct w.r.t. P_0. Then, R is said to be σ*-consistent* if $\sigma(M_1) \geq \sigma(M_2)$, and it is said to be μ*-consistent* if $\mu(M_1\theta) \geq \mu(M_2\theta)$ for any ground substitution θ for $Vf(M_1)$ such that $M_1\theta$ and $M_2\theta$ are provable by P_0.

The following shows the correctness of our basic transformation system.

Proposition 1. Correctness of Transformation [12]
Let P_0 be an initial program with the initial strati cation σ, and $\mathcal{R}$ be a set of replacement rules correct w.r.t. P_0. Let $P_0, \ldots, P_n$ $(n \geq 0)$ be a transformation sequence with the input $(P_0, \mathcal{R})$, where (i) every application of negative unfolding is μ-consistent, (ii) every application of reversible folding is σ-consistent and satis es TS-folding condition, and (iii) every application of replacement rule is consistent with σ and μ. Then, $M(P_n) = M(P_0)$. □

3 An Extended Framework for Unfold/fold Transformation System

In this section, we consider an extension of our basic unfold/fold transformation system in Sect. 2, by introducing three new transformation rules: *extended negative unfolding*, *simultaneous folding* and *negative folding*. We then show the correctness of these rules in the extended transformation system.

3.1 Negative Unfolding with Existential Variables

Definition 5. Extended Negative Unfolding

Let C be a renamed apart clause in a strati ed program P of the form: $H \leftarrow G_1, \neg A, G_2$, where A is an atom, and G_1 and G_2 are (possibly empty) conjunctions of literals. Let $D_1, \ldots, D_k$ with $k \geq 0$, be all clauses of program P, such that A is uni able with $hd(D_1), \ldots, hd(D_k)$, with most general uni ers $\theta_1, \ldots, \theta_k$, respectively. Assume that:

(PP1) $A = hd(D_1)\theta_1 = \ldots = hd(D_k)\theta_k$, that is, for each i $(1 \leq i \leq k)$, A is an instance of $hd(D_i)$, and
(ENU) for some i $(1 \leq i \leq k)$, there exists an existential variable in $bd(D_i)\theta_i$.

Then, for each i $(1 \leq i \leq k)$, we introduce a new non-primitive predicate $newp_i$ and its de nition clause: $newp_i(\tilde{X}_i) \leftarrow bd(D_i)\theta_i$, where $\tilde{X}_i$ is a (possibly empty) sequence of free variables in $hd(D_i)\theta_i$.

By *extended negative unfolding* w.r.t. $\neg A$, we derive from P the new program P' by replacing C by C', where C' is the clause $H \leftarrow G_1, \neg newp_1(\tilde{X}_1), \ldots, \neg newp_k(\tilde{X}_k), G_2$.

In particular: (i) if for some i $(1 \leq i \leq k)$, $bd(D_i) = true$, i.e., D_i is a unit clause, then we derive from P the new program P' by deleting clause C, and (ii) in condition (ENU), if $bd(D_i)\theta_i$ $(1 \leq i \leq k)$ consists of a single atom without existential variables, then we can use $bd(D_i)\theta_i$ in place of $newp_i(\tilde{X}_i)$ in C'.

The descent level of C' is the same as that of C. □

As is noted in Remark 2, the de nition clauses newly introduced in extended negative unfolding are assumed to be present in an initial program P_0 from scratch. The correctness of our transformation system with extended negative unfolding is shown in Proposition 2. We note that no additional conditions are needed for applying extended negative unfolding.

In inductive proofs via unfold/fold transformations, the notion of *useless* predicates is useful, which is originally due to Pettorossi and Proietti [6].

Definition 6. Useless Predicate

The set of the *useless* predicates of a program P is the maximal set U of predicates of P such that a predicate p is in U if, for the body of each clause of $Def(p, P)$, it has a positive literal whose predicate is in U. □

Let A be a ground atom with predicate symbol p. Then, it is easy to see that, if p is a useless predicate of P, then $M(P) \models \neg A$.

Example 3. Inductive Proof via Transformations (Continued from Example 1)

Suppose that we would like to show the first order formula φ: $\forall X\,(gen(X) \rightarrow test(X))$, which means that any string X generated by gen satisfies $test(X)$, i.e., any string consisting of only 0's can be transformed (through a finite number of applications of $trans$) into a string consisting only 1's.

Fig. 1 shows a proof process via unfold/fold transformations, where extended negative unfolding (*ENU* for short) is used. We first consider the *statement* [4] $f \leftarrow \varphi$, where f is a new predicate. Then, we apply Lloyd-Topor transformation to $f \leftarrow \varphi$, which preserves the perfect model semantics [6], obtaining the clauses $\{C_f, C_{nf}\}$, where predicate nf is a new predicate introduced by Lloyd-Topor transformation and its intended meaning is the *n*egation of f.[7]

By applying positive unfolding to C_{nf} w.r.t. $gen(X)$, we have clauses $\{(1), (2)\}$. Since the body of the 2nd clause of the definition of *test* has an existential variable, we apply ENU to (1), obtaining clauses $\{(3), (4)\}$, where $newp_1$ is a new predicate symbol introduced in ENU. Note that the simplification rule (ii) in Def. 5 is used to derive (3). Similarly, applying ENU to (2), we have clauses $\{(5), (6)\}$, where $newp_2$ is a new predicate symbol introduced in ENU.

Applying negative unfolding to (3) w.r.t. $\neg canon([])$, we obtain a new clause (7) with $\bot$ (*false*) in its body, thus it is deleted.

On the other hand, we reconsider replacement rule R: $test([1|X]) \Rightarrow test(X)$, which was used in Example 2. Since *test*, *canon* and *trans* are primitive predicate symbols, it is immediate from definition that $\mu(test([1|X])\theta) = \mu(test(X)\theta) = \mathbf{0}$ for any ground substitution θ s.t. $M(P_0) \models test([1|X])\theta$, thus R is μ-consistent. Therefore, we can apply replacement rule R to clause (8), obtaining clause (9). From (9) and (10), we have clause (11) by negative unfolding, which is further folded using C_{nf} as a folder clause, obtaining clause (12).

Let $P_0' = P_0 \cup \{C_f, C_{nf}, (4), (6)\}$ and $P_1' = P_0' \setminus \{C_{nf}\} \cup \{(12)\}$. P_0' is a stratified program, as we can extend the initial stratification σ by $\sigma(nf) = 1$, $\sigma(f) = 2$, $\sigma(newp_1) = \sigma(newp_2) = 0$. We also extend *level* by $level(nf) = 2 > level(gen)$, $level(f) = 1$, $level(newp_1) = level(newp_2) = 1 > \max\{level(trans), level(test)\}$. Therefore, P_0' satisfies the initial program condition. Since the above transformation sequence satisfies the conditions in Table 1, it holds that $M(P_0') = M(P_1')$. Note that nf is a useless predicate of P_1'. We thus have that $M(P_1') \models \neg nf$, which means that $M(P_1') \models f$, thus $M(P_0') \models f$. Therefore, it follows that $M(P_0') \models \varphi$, which is to be proved. □

3.2 Simultaneous Folding

The following is the definition of simultaneous folding, which is originally given for definite programs in [3, 10]. Figure 2 is a schematic explanation of simultaneous folding.

[7] Since our proof of f is based on proof by contradiction, our task is to show that the definition of predicate nf is transformed so that it is either an empty set or nf is a useless predicate in a transformed program.

C_f :	$f \leftarrow \neg nf$	
C_{nf} :	$nf \leftarrow gen(X), \neg test(X)$	
(1) :	$nf \leftarrow \neg test([])$	(pos. unfold C_{nf})
(2) :	$nf \leftarrow gen(X), \neg test([0\|X])$	
(3) :	$nf \leftarrow \neg canon([]), \neg newp_1$	(ext. neg. unfold (1))
(4) :	$newp_1 \leftarrow trans([], Y), test(Y)$	
(5) :	$nf \leftarrow gen(X), \neg canon([0\|X]), \neg newp_2(X)$	(ext. neg. unfold (2))
(6) :	$newp_2(X) \leftarrow trans([0\|X], Y), test(Y)$	
(7) :	$nf \leftarrow \bot$	(neg. unfold (3))
(8) :	$newp_2(X) \leftarrow test([1\|X])$	(pos. unfold (6))
(9) :	$newp_2(X) \leftarrow test(X)$	(replace (8))
(10) :	$nf \leftarrow gen(X), \neg newp_2(X)$	(neg. unfold (5))
(11) :	$nf \leftarrow gen(X), \neg test(X)$	(neg. unfold (10))
(12) :	$nf \leftarrow nf$	(fold (11))

Fig. 1. A Proof of Example 3 via Unfold/fold Transformations: In clause (7), $\bot$ means $false$, and the clause is deleted by negative unfolding.

Definition 7. Simultaneous Folding

Let $S_{fd} = \{C_1, \ldots, C_m\}$ be a set of clauses in P_k, each of which is of the form: $H \leftarrow \exists X_1 \ldots X_n(M_i, N)$, where M_i and N are molecules, and $X_1 \ldots X_n$ are some free variables in M_i, for $i = 1, \ldots, m$. Let $S_{fg} = \{D_1, \ldots, D_m\}$ be a set of clauses in an initial program P_0. Moreover, let f be a bijection (called a *folding bijection*) between S_{fd} and S_{fg} such that the following hold:

1. for each pair $(C_i, D_i) \in f$ $(i = 1, \ldots, m)$, M_i is a P-expansion of a single common atom A (by a renamed apart clause D_i), and
2. $A = hd(D_1)\theta_1 = \cdots = hd(D_m)\theta_m$.

Then, by *simultaneously folding* C_i w.r.t. M_i *by* P_0, we derive from P_k the new program $P_{k+1} = (P_k \setminus \{C_1, \ldots, C_m\}) \cup \{\gamma\}$, where γ is the clause: $H \leftarrow \exists X_1 \ldots X_n(A, N)$. C_i is called a *folded clause* and D_i a *folding clause* (or *folder clause*) $(i = 1, \ldots, m)$.

The simultaneous folding operation is said to be *set reversible* if $M_1, \ldots, M_m$ are all the P-expansions of A by P_0 in the above de nition. If γ is rst introduced as the result of simultaneous folding with folded clauses $C_1, \ldots, C_m$, then the descent level of γ is de ned by $dl(\gamma) = \max\{dl(C_1), \ldots, dl(C_m)\}$. □

Conditions on Simultaneous Folding Suppose that simultaneous folding which is set reversible is applied to folded clause C_i $(i = 1, \ldots, m)$ with folding clause $D_i \in P_0$. Let f be a folding bijection, i.e., $(C_i, D_i) \in f$. Then, the application of simultaneous folding is said to be *σ-consistent*, if each pair (C_i, D_i) satis es σ-consistency, i.e., $\sigma(H) \geq \sigma(A)$ in Def. 7. Moreover, the application of simultaneous folding is said to satisfy *TS-folding condition*, if each pair (C_i, D_i) satis es TS-folding condition, i.e., the descent level of C_i is *smaller* than the level of D_i for each i. □

The correctness of simultaneous folding when incorporated into our transformation system is shown in Proposition 2.

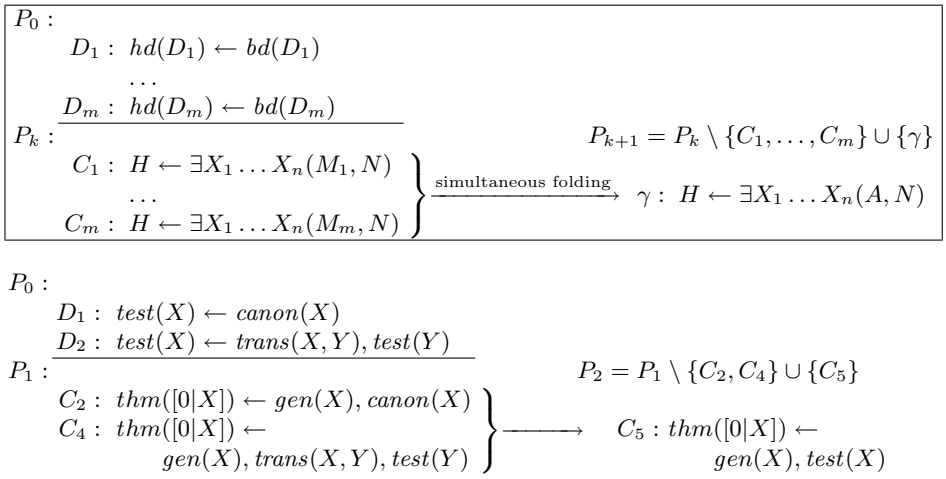


Fig. 2. A Schema of Simultaneous Folding (above) and Example 4 (below)

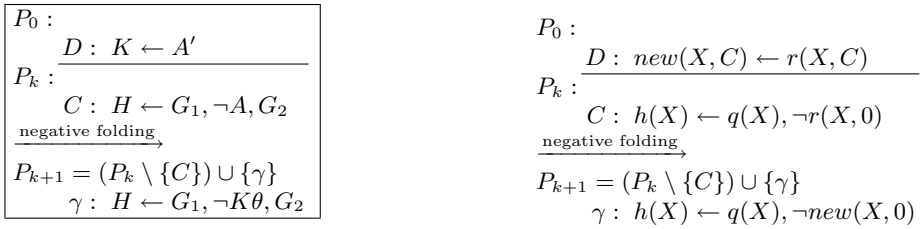
Example 4. (Continued from Example 2) We show here that the derivations given in [10] can be performed also in our framework. Suppose that we would like to apply simultaneous folding to C_2, C_4 in P_1 using the de nition clauses of *test* as the folders (see Fig. 2). However, our partition of predicate symbols in $Pred(P_0)$ and assignment function *level* in Example 1 do not satisfy the application condition of simultaneous folding (i.e., TS-folding condition). We thus choose another partition of $Pred(P_0)$ and *level*; predicate symbol *test* is non-primitive, $level(test) = 2$ this time, and the others are the same as before. Then, we can apply the above-mentioned simultaneous folding, obtaining clause C_5: $thm([0|X]) \leftarrow gen(X), test(X)$. Let $P_2 = P_1 \setminus \{C_2, C_4\} \cup \{C_5\}$.

However, the price for adopting this partition and *level* is that we should show that replacement rule R: $test([1|X]) \Rightarrow test(X)$ used in Example 2 is μ-consistent. This is shown in our framework using a method similar to the *syntactic goal replacement* [10].

Consider again the formula φ: $\forall X\,(gen(X) \rightarrow test(X))$ in Example 3. Instead of proving φ directly, Roychoudhury et al. [10] try to prove a more general formula φ_1: $\forall X\ (thm(X) \leftrightarrow gen(X))$. From the de nition of *thm*, we have that $M(P_0) \models \forall X\ (thm(X) \leftrightarrow gen(X) \wedge test(X))$, which then implies that $M(P_0) \models \varphi$.

To show φ_1, we apply folding to C_5 in P_2 using the de nition clause of *thm* as the folder, obtaining clause C_6: $thm([0|X]) \leftarrow thm(X)$. Let $P_3 = P_2 \setminus \{C_5\} \cup \{C_6\}$. Then, the de nition of predicate *thm* in P_3 consists of clauses $\{C_1, C_6\}$, whose syntactic structure is exactly the same as that of predicate *gen* (modulo predicate names). From this, we conclude that φ_1 holds. □

We are now in a position to compare the inductive proof using extended negative unfolding (ENU) in Example 3 and the above-mentioned proof using simultaneous folding in Example 4. As far as this particular formula φ is con-



Conditions on Negative Folding (i) D is a single clause consisting of the de nition of predicate K, and (ii) there exists a substitution θ such that $A'\theta = A$ and $Vf(K) = Vf(A')$. Note from (i) that $\sigma(K) = \sigma(A')$.

Fig. 3. A Schema of Negative Folding (left) and an Example (right): A, A' are atoms, and G_1, G_2 are (possibly empty) conjunctions of literals. By applying *negative folding* to C w.r.t. $\neg A$ using clause D, we derive from P_k the new program $P_{k+1} = (P_k \setminus \{C\}) \cup \{\gamma\}$. The descent level of γ is the same as that of C.

cerned, we can say that the proof method using ENU is: (i) simpler, because it is free to show that replacement rule R is μ-consistent, and (ii) more intuitive, because we can show the original proof obligation φ directly.

3.3 Negative Folding and the Correctness of the Extended Transformation System

In [2], Fioravanti, Pettorossi and Proietti consider *negative folding* (*NF* for short) and show that it is useful for program derivation. Pettorossi, Proietti and Senni also use NF for veri cation of CTL* properties of nite state reactive systems [7]. Fig. 3 shows a schematic description of NF and its application conditions.

We can show that our previous transformation system can be extended to incorporate NF, as well as ENU and simultaneous folding. The notion of a transformation sequence in Def. 2 is modi ed so that it contains these rules and every application of simultaneous folding is set reversible. Then, we have the following:

Proposition 2. Correctness of Extended Unfold/fold Transformation
Let P_0 be an initial program with the initial strati cation σ, and $\mathcal{R}$ be a set of replacement rules correct w.r.t. P_0. Let $P_0, \ldots, P_n$ $(n \geq 0)$ be a transformation sequence with the input $(P_0, \mathcal{R})$, where every application of negative unfolding, folding, and replacement rule satis es the conditions in Prop. 1. Moreover, suppose that every application of simultaneous folding is σ-consistent and satis es TS-folding condition. Then, $M(P_n) = M(P_0)$. □

In our framework, both ENU and NF can be regarded as instances of replacement rule. Therefore, the correctness of ENU and NF when incorporated into our unfold/fold transformation system is immediate from Proposition 1.

One might think that the same transformation of ENU would be realized by NF, together with a series of transformations consisting of de nition introduction, positive unfolding, folding and PP-negative unfolding. Such transformations would be possible in our extended framework, while they will not always be allowed in the frameworks by [6, 2] due to their folding conditions. Moreover, the constraints on assigning levels to newly introduced predicates in ENU will be weaker than those in the case of NF; a level assigned to $newp_i$ in ENU can be less than or equal to that required in the case of NF. Therefore, ENU will allow more freedom in choosing levels of $newp_i$, which will make the newly introduced de nition clauses amenable to subsequent transformations.

4 Related Work and Concluding Remarks

We have proposed a new transformation rule called *extended* negative unfolding (ENU), where an unfolding clause is allowed to have existential variables, thereby eliminating a restriction of the conventional negative unfolding (PP-NU). By virtue of this extension, an unfolded clause obtained by ENU will be amenable to subsequent transformations. We have shown such an example in Sect. 3.1. Moreover, we have complemented our previous transformation system [12] with another two transformation rules, simultaneous folding (SF) and negative folding (NF), and showed the correctness of the extended transformation system in the sense of the perfect model semantics.

We have also examined the use of SF in inductive proofs. We have illustrated by examples that our transformation system with ENU, when used together with Lloyd-Topor transformation, can inductively prove properties of problems which were proved by using SF in the literature. Although the examples were originally given in [10] to show that SF is essentially more powerful than conventional folding, the proof illustrated here have shown that, as far as these particular examples are concerned, our inductive proof method using ENU can show them in a simpler and more intuitive manner.

There are several precursors of the present paper, where unfold/fold transformation is used for proving program properties, including, among others, Pettorossi and Proietti [6], Fioravanti et al. [2] and Roychoudhury et al. [10].

The transformation system by Fioravanti et al. [2] contains PP-NU, SF and NF, while their folding does not allow *recursive* folding; the de nition of a folder clause cannot contain recursive clauses. The same restriction is imposed on the framework in [6]. Moreover, a replacement rule in the frameworks by Pettorossi and Proietti [6] and Fioravanti et al. [2] is restricted to the form where the literals occurring in the rule are primitive, while non-primitive predicates are allowed in the replacement rule in our framework. Due to this generality, both ENU and NF can be regarded as instances of replacement rule, as discussed in Sect. 3.3.

On the other hand, the transformation system by Roychoudhury et al. [10] employed a very general measure for proving the correctness. Their frameworks [10, 9], however, have no NU. In fact, the correctness proof in [9] depends on the preservation of the semantic kernel [1], which is not preserved in general when

NU and replacement rule are applied. Some semantic issues on NU in the answer set semantics are studied in [11].

One of the motivations of this work is to understand the close relationship between program transformation and inductive theorem proving, and apply transformation techniques to veri cation problems. Pettorossi, Proietti and Senni [7] recently propose a transformational method for verifying CTL* properties of nite state reactive systems, where they consider a (locally) strati ed program with in nite terms (*ω-programs*). We believe that our framework of unfold/fold transformation will be also applicable to such strati ed ω-programs. We hope that our results reported in this paper will be a contribution to promote further cross-fertilization between program transformation and model checking.

Acknowledgement The author would like to thank anonymous reviewers for their constructive and useful comments on the previous version of the paper.

References

1. Aravindan, C. and Dung, P. M., On the Correctness of Unfold/fold Transformation of Normal and Extended Logic Programs, *J. Logic Programming*, 24(3), 295-322, 1995.
2. Fioravanti, F., Pettorossi, A. and Proietti, M., Transformation Rules for Locally Strati ed Constraint Logic Programs, *Proc. Program Development in Computational Logic*, LNCS 3049, pp. 291-339, 2004.
3. Gergatsoulis, M. and Katzouraki, M., Unfold/Fold Transformations For De nite Clause Programs, *Proc. PLILP'94*, LNCS 844, pp. 340-354, 1994.
4. Lloyd, J. W. , *Foundations of Logic Programming*, Springer, 1987, Second edition.
5. Pettorossi, A. and Proietti, M., Transformation of Logic Programs: Foundations and Techniques, *J. Logic Programming*, 19/20:261-320, 1994.
6. Pettorossi, A. and Proietti, M., Perfect Model Checking via Unfold/Fold Transformations, *Proc. CL2000*, LNAI 1861, pp. 613-628, 2000.
7. Pettorossi, A., Proietti, M. and Senni, V., Deciding Full Branching Time Logic by Program Transformations, *Proc. LOPSTR'09*, LNCS 6037, pp. 5-21, 2010.
8. Przymusinski, T.C., On the Declarative and Procedural Semantics of Logic Programs. *J. Automated Reasoning*, 5(2):167–205, 1989.
9. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C. R. and Ramakrishnan, I. V., Beyond Tamaki-Sato Style Unfold/fold Transformations for Normal Logic Programs, *Int. J. Foundations of Computer Science* 13(3), 387-403, 2002.
10. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C. R. and Ramakrishnan, I. V., An Unfold/fold Transformation Framework for De nite Logic Programs, *ACM Trans. on Programming Languages and Systems* 26(3), 464-509, 2004.
11. Seki, H., On Negative Unfolding in the Answer Set Semantics, *Proc. LOPSTR'08*, LNCS 5438, pp. 82-96, 2010.
12. Seki, H., On Inductive and Coinductive Proofs via Unfold/fold Transformations, *Proc. LOPSTR'09*, LNCS 6037, pp. 82-96, 2010.
13. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, *Proc. 2nd Int. Conf. on Logic Programming*, 127-138, 1984.
14. Tamaki, H. and Sato, T., A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, *Technical Report*, No. 86-4, Ibaraki Univ., 1986.

Dependency Triples for Improving Termination Analysis of Logic Programs with Cut*

Thomas Ströder[1], Peter Schneider-Kamp[2], and Jürgen Giesl[1]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
{stroeder,giesl}@informatik.rwth-aachen.de
[2] IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

Abstract. In very recent work, we introduced a non-termination preserving transformation from logic programs with cut to definite logic programs. While that approach allows us to prove termination of a large class of logic programs with cut automatically, in several cases the transformation results in a non-terminating definite logic program.
In this paper we extend the transformation such that logic programs with cut are no longer transformed into definite logic programs, but into dependency triple problems. By the implementation of our new method and extensive experiments, we empirically evaluate the practical benefit of our contributions.

1 Introduction

Automated termination analysis for logic programs has been widely studied, see, e.g., ([3–5, 13, 15, 16, 19]). Still, virtually all existing techniques only prove universal termination of *definite* logic programs, which do not use the *cut* "!".[3] But most realistic Prolog programs make use of the cut or related operators such as *negation as failure* ("\+") or *if then else* ("–> ... ; ..."), which can be expressed using cuts. In [18] we introduced a non-termination preserving automated transformation from logic programs with cut to definite logic programs. The transformation consists of two stages. The first stage is based on constructing a so-called *termination graph* from a given logic program with cut. The second stage is the generation of a definite logic program from this termination graph. In this paper, we improve the second stage of the transformation. Instead of generating a definite logic program from the termination graph, we now

* Supported by the DFG grant GI 274/5-2, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Natural Science Research Council.

[3] An exception is [12], which presents a transformation of "safely-typed" logic programs to term rewrite systems. However, the resulting rewrite systems are quite complex and since there is no implementation of [12], it is unclear whether they can indeed be handled by existing termination tools from term rewriting. Moreover, [12] does not allow programs with arbitrary cuts (e.g., it does not operate on programs like the one in Ex. 1).

generate a dependency triple problem. The goal is to improve the power of the approach, i.e., to succeed also in many cases where the transformation of [18] yields a non-terminating definite logic program.

Dependency triples were introduced in [14] and improved further to the so-called *dependency triple framework* in [17]. The idea was to adapt the successful dependency pair framework [2, 8–10] from term rewriting to (definite) logic programming. This resulted in a completely modular method for termination analysis of logic programs which even allowed to combine "direct" and "transformational" methods within the proof of one and the same program. The experiments in [17] showed that this leads to the most powerful approach for automated termination analysis of definite logic programs so far. Our aim is to benefit from this work by providing an immediate translation from logic programs with cut (resp. from their termination graphs) to dependency triple problems.

This paper is structured as follows. After a short section on preliminaries, we recapitulate the construction of termination graphs in Sect. 3 and we demonstrate their transformation into definite logic programs in Sect. 4. Then, in Sect. 5 we illustrate the idea of dependency triples and introduce a novel transformation of termination graphs into dependency triple problems. We show that this new transformation has significant practical advantages in Sect. 6 and, finally, conclude in Sect. 7.

2 Preliminaries

See e.g. [1] for the basics of logic programming. We distinguish between individual cuts to make their scope explicit. Thus, we use a signature Σ containing all predicate and function symbols as well as all labeled versions of the cut operator $\{!_m/0 \mid m \in \mathbb{N}\}$. For simplicity we just consider terms $\mathcal{T}(\Sigma, \mathcal{V})$ and no atoms, i.e., we do not distinguish between predicate and function symbols.[4] A *query* is a sequence of terms from $\mathcal{T}(\Sigma, \mathcal{V})$. Let $\mathit{Goal}(\Sigma, \mathcal{V})$ be the set of all queries, where $\square$ is the empty query. A *clause* is a pair $H \leftarrow B$ where the *head* H is from $\mathcal{T}(\Sigma, \mathcal{V})$ and the *body* B is a query. A *logic program* $\mathcal{P}$ (possibly with cut) is a finite sequence of clauses. $\mathit{Slice}(\mathcal{P}, t)$ are all clauses for t's predicate, i.e., $\mathit{Slice}(\mathcal{P}, p(t_1, ..., t_n)) = \{c \mid c = \text{“}p(s_1, ..., s_n) \leftarrow B\text{”} \in \mathcal{P}\}$.

A substitution σ is a function $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ and we often denote its application to a term t by $t\sigma$ instead of $\sigma(t)$. As usual, $\mathit{Dom}(\sigma) = \{X \mid X\sigma \neq X\}$ and $\mathit{Range}(\sigma) = \{X\sigma \mid X \in \mathit{Dom}(\sigma)\}$. The restriction of σ to $\mathcal{V}' \subseteq \mathcal{V}$ is $\sigma_{|\mathcal{V}'}(X) = \sigma(X)$ if $X \in \mathcal{V}'$, and $\sigma_{|\mathcal{V}'}(X) = X$ otherwise. A substitution σ is the *most general unifier* (mgu) of s and t iff $s\sigma = t\sigma$ and, whenever $s\gamma = t\gamma$ for some other unifier γ, there exists a δ such that $X\gamma = X\sigma\delta$ for all $X \in \mathcal{V}(s) \cup \mathcal{V}(t)$. If s and t have no mgu, we write $s \not\sim t$.

Let Q be a query $A_1, \ldots, A_m$, let c be a clause $H \leftarrow B_1, \ldots, B_k$. Then Q' is a *resolvent* of Q and c using θ (denoted $Q \vdash_{c,\theta} Q'$) if θ is the mgu of A_1 and H, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$.

[4] To ease the presentation, in the paper we exclude terms with cuts $!_m$ as proper subterms.

A *derivation* of a program $\mathcal{P}$ and Q is a possibly infinite sequence $Q_0, Q_1, \ldots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$ for some substitution θ_{i+1} and some fresh variant c_{i+1} of a clause from $\mathcal{P}$. For a derivation $Q_0, \ldots, Q_n$ as above, we also write $Q_0 \vdash^n_{\mathcal{P},\theta_1 \ldots \theta_n} Q_n$ or $Q_0 \vdash^n_{\mathcal{P}} Q_n$, and we also write $Q_i \vdash_{\mathcal{P}} Q_{i+1}$ for $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$. The query Q *terminates* for $\mathcal{P}$ if all derivations of $\mathcal{P}$ and Q are finite, i.e., if $\vdash_{\mathcal{P}}$ is terminating for Q. $Answer(\mathcal{P}, Q)$ is the set of all substitutions δ such that $Q \vdash^n_{\mathcal{P},\delta} \square$ for some $n \in \mathbb{N}$.

Finally, to denote the term resulting from replacing all occurrences of a function symbol f in a term t by another function symbol g, we write $t[f/g]$.

3 Termination Graphs

To illustrate the concepts and the contributions of this paper, we use the following leading example. While this example has been designed for this purpose, as demonstrated in Sect. 6, our contributions also have a considerable effect for termination analysis of "general" logic programs with cut.

Example 1. The following clauses define a (simplified) variant of the logic program `Stroeder09/giesl97.pl` from the Termination Problem Data Base [23] that is used in the annual international Termination Competition [22]. This example formulates a functional program from [7, 24] with nested recursion as a logic program. Here, the predicate p is used to compute the predecessor of a natural number while eq is used to unify two terms.

$$\mathsf{f}(\mathsf{0}, Y) \leftarrow !, \mathsf{eq}(Y, \mathsf{0}). \tag{1}$$

$$\mathsf{f}(X, Y) \leftarrow \mathsf{p}(X, P), \mathsf{f}(P, U), \mathsf{f}(U, Y). \tag{2}$$

$$\mathsf{p}(\mathsf{0}, \mathsf{0}). \tag{3}$$

$$\mathsf{p}(\mathsf{s}(X), X). \tag{4}$$

$$\mathsf{eq}(X, X). \tag{5}$$

Note that when ignoring cuts, this logic program is not terminating for the set of queries $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$. To see this, consider the following derivation: $\mathsf{f}(\mathsf{0}, A) \vdash \mathsf{p}(\mathsf{0}, P), \mathsf{f}(P, U), \mathsf{f}(U, A) \vdash_{\{P/\mathsf{0}\}} \mathsf{f}(\mathsf{0}, U), \mathsf{f}(U, A) \vdash \mathsf{eq}(U, \mathsf{0}), \mathsf{f}(U, A) \vdash_{\{U/\mathsf{0}\}} \mathsf{f}(\mathsf{0}, A)$. Clearly, this leads to an infinite (looping) derivation.

Fig. 1 recapitulates the formulation of the operational semantics of logic programming with cut that we introduced in [18]. A formal proof on the correspondence of our inference rules to the semantics of the Prolog ISO standard [6] can be found in [20]. The formulation with our inference rules is particularly suitable for an extension to *classes* of queries in Fig. 2, and for synthesizing cut-free programs in Sect. 4 or dependency triples in Sect. 5. Our semantics is given by seven inference rules. They operate on *states* that do not just represent the current goal, but also the backtrack information that is needed to describe the effect of cuts. The backtrack information is given by a sequence of goals (separated by "|") which are optionally labeled by the program clause i that has to be applied to the goal next and by a number m that determines how cuts will

$$\frac{\square \mid S}{S}\ (\textsc{Suc}) \qquad \frac{?_m \mid S}{S}\ (\textsc{Fail}) \qquad \frac{!_m, Q \mid S \mid ?_m \mid S'}{Q \mid ?_m \mid S'}\ (\textsc{Cut})^{\text{where } S \text{ contains no } ?_m} \qquad \frac{!_m, Q \mid S}{Q}\ (\textsc{Cut})^{\text{where } S \text{ contains no } ?_m}$$

$$\frac{t, Q \mid S}{(t,Q)^{i_1}_m \mid \ldots \mid (t,Q)^{i_k}_m \mid ?_m \mid S}\ (\textsc{Case})$$ where t is neither a cut nor a variable, m is greater than all previous marks, and $\mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$ with $i_1 < \ldots < i_k$

$$\frac{(t,Q)^i_m \mid S}{B'_i\sigma, Q\sigma \mid S}\ (\textsc{Eval})^{\text{where } c_i = H_i \leftarrow B_i,\ mgu(t,H_i)=\sigma,\ B'_i = B_i[!\ /\ !_m].} \qquad \frac{(t,Q)^i_m \mid S}{S}\ (\textsc{Backtrack})^{\text{where } c_i = H_i \leftarrow B_i \text{ and } t \not\sim H_i.}$$

Fig. 1. Operational Semantics by Concrete Inference Rules

be labeled when evaluating this goal later on. Moreover, our states also contain explicit *marks* $?_m$ to mark the end of the scope of a cut $!_m$.

For the query $\mathsf{f}(\mathsf{0}, A)$ in Ex. 1, we obtain the following derivation with the rules of Fig. 1: $\mathsf{f}(\mathsf{0},A) \vdash_{\textsc{Case}} \mathsf{f}(\mathsf{0},A)^1_1 \mid \mathsf{f}(\mathsf{0},A)^2_1 \mid ?_1 \vdash_{\textsc{Eval}} !_1, \mathsf{eq}(A,\mathsf{0}) \mid \mathsf{f}(\mathsf{0},A)^2_1 \mid ?_1 \vdash_{\textsc{Cut}} \mathsf{eq}(A,\mathsf{0}) \mid ?_1 \vdash_{\textsc{Case}} \mathsf{eq}(A,\mathsf{0})^5_2 \mid ?_2 \mid ?_1 \vdash_{\textsc{Eval}} \square \mid ?_2 \mid ?_1 \vdash_{\textsc{Suc}} ?_2 \mid ?_1 \vdash_{\textsc{Fail}} ?_1 \vdash_{\textsc{Fail}} \varepsilon$. Thus, when considering cuts, our logic program terminates for the query $\mathsf{f}(\mathsf{0}, A)$, and, indeed, it terminates for all queries from the set $\{\mathsf{f}(t_1,t_2) \mid t_1 \text{ is ground}\}$. For further details on the intuition behind the inference rules, we refer to [18].

To show termination for infinite sets of queries (e.g., $\{\mathsf{f}(t_1,t_2) \mid t_1 \text{ is ground}\}$), we need to represent classes of queries by abstract states. To this end, in [18] we introduced *abstract terms* and a set $\mathcal{A}$ of *abstract variables*, where each $T \in \mathcal{A}$ represents a fixed but arbitrary term. $\mathcal{N}$ consists of all "ordinary" variables in logic programming. Then, as *abstract terms* we consider all terms from the set $\mathcal{T}(\Sigma, \mathcal{V})$ where $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$. *Concrete terms* are terms from $\mathcal{T}(\Sigma, \mathcal{N})$, i.e., terms containing no abstract variables. For any set $\mathcal{V}' \subseteq \mathcal{V}$, let $\mathcal{V}'(t)$ be the variables from $\mathcal{V}'$ occurring in the term t. To determine by which terms an abstract variable may be instantiated, we add a knowledge base $KB = (\mathcal{G}, \mathcal{U})$ to each state, where $\mathcal{G} \subseteq \mathcal{A}$ and $\mathcal{U} \subseteq \mathcal{T}(\Sigma,\mathcal{V}) \times \mathcal{T}(\Sigma,\mathcal{V})$. Instantiations γ that respect KB may only instantiate the variables in $\mathcal{G}$ by ground terms. And $(s,s') \in \mathcal{U}$ means that we are restricted to instantiations γ where $s\gamma \not\sim s'\gamma$, i.e., s and s' may not become unifiable when instantiating them with γ. We call a substitution γ that respects the information in KB a *concretization* w.r.t. KB.

Fig. 2 shows the abstract inference rules introduced in [18]. They work on classes of queries represented by abstract terms with a knowledge base. Except for Backtrack and Eval, the adaption of the concrete inference rules to corresponding abstract inference rules is straightforward.

For Backtrack and Eval we must consider that the set of queries represented by an abstract state may contain both queries where the concrete Eval rule and where the concrete Backtrack rule is applicable. Thus, the abstract Eval rule has two successors corresponding to these two cases. As abstract variables not known to represent ground terms may share variables, we have to replace all variables by fresh abstract variables in Eval's left successor state which corresponds to the application of the concrete Eval rule. For the backtrack

$$\frac{\Box \mid S; KB}{S; KB}\ (\textsc{Suc}) \qquad \frac{?_m \mid S; KB}{S; KB}\ (\textsc{Fail})$$

$$\frac{!_m, Q \mid S \mid ?_m \mid S'; KB}{Q \mid ?_m \mid S'; KB}\ (\textsc{Cut})\ \begin{array}{l}\text{where } S \\ \text{contains no} \\ ?_m\end{array} \qquad \frac{!_m, Q \mid S; KB}{Q; KB}\ (\textsc{Cut})\ \begin{array}{l}\text{where } S \\ \text{contains no} \\ ?_m\end{array}$$

$$\frac{t, Q \mid S; KB}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S; KB}\ (\textsc{Case})$$ where t is neither a cut nor a variable, m is greater than all previous marks, and $\mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$ with $i_1 < \ldots < i_k$

$$\frac{(t, Q)^i_m \mid S; KB}{S; KB}\ (\textsc{Backtrack})$$ where $c_i = H_i \leftarrow B_i$ and there is no concretization γ w.r.t. KB such that $t\gamma \sim H_i$.

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{U})}{B'_i\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{U}\sigma|_{\mathcal{G}}) \qquad S; (\mathcal{G}, \mathcal{U} \cup \{(t, H_i)\})}\ (\textsc{Eval})$$

where $c_i = H_i \leftarrow B_i$ and $mgu(t, H_i) = \sigma$. W.l.o.g., for all $X \in \mathcal{V}$, $\mathcal{V}(\sigma(X))$ only contains fresh abstract variables not occurring in $t, Q, S, \mathcal{G}$, or $\mathcal{U}$. Moreover, $\mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$ and $B'_i = B_i[!/!_m]$.

$$\frac{S; (\mathcal{G}, \mathcal{U})}{S'; (\mathcal{G}', \mathcal{U}')}\ (\textsc{Instance})$$ if there is a μ such that $S = S'\mu$, $\mu|_{\mathcal{N}}$ is a variable renaming, $\mathcal{V}(T\mu) \subseteq \mathcal{G}$ for all $T \in \mathcal{G}'$, and $\mathcal{U}'\mu \subseteq \mathcal{U}$.

$$\frac{S \mid S'; KB}{S; KB \qquad S'; KB}\ (\textsc{Parallel})$$ if $AC(S) \cap AM(S') = \varnothing$

The *active cuts* $AC(S)$ are all m where $!_m$ is in S or $(t, q)^i_m$ is in S and c_i's body has a cut. The *active marks* $AM(S)$ are all m where $S = S' \mid ?_m \mid S''$ and $S' \neq \varepsilon \neq S''$.

$$\frac{t, Q; (\mathcal{G}, \mathcal{U})}{t; (\mathcal{G}, \mathcal{U}) \qquad Q\mu; (\mathcal{G}', \mathcal{U}\mu)}\ (\textsc{Split})$$ where μ replaces all (abstract and non-abstract) variables from $\mathcal{V} \setminus \mathcal{G}$ by fresh abstract variables and $\mathcal{G}' = \mathcal{G} \cup \mathit{ApproxGnd}(t, \mu)$.

Here, we assume that we have a *groundness analysis* function $\mathit{Ground}_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$, see, e.g., [11]. Then we have $\mathit{ApproxGnd}(p(t_1, \ldots, t_n), \mu) = \{\mathcal{A}(t_j\mu) \mid j \in \mathit{Ground}_{\mathcal{P}}(p, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$.

Fig. 2. Abstract Inference Rules

possibilities and the knowledge base, however, we may only use an answer substitution $\sigma|_{\mathcal{G}}$ restricted to abstract variables known to represent ground terms. The reason is that, due to backtracking, any substitutions of non-abstract variables may become canceled. For the abstract variables $T \in \mathcal{G}$, their instantiation with the answer substitution of the abstract Eval rule corresponds to a case analysis over the shape of the ground terms that T is representing. Thus, in case of a successful unification we know that these terms must have a certain shape and we can keep this information also after backtracking. Fig. 3 shows how the rules of Fig. 2 can be applied to the initial state $\mathsf{f}(T_1, T_2)$ with the knowledge base $(\{T_1\}, \varnothing)$, which represents the set of queries $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$. In Fig. 3 we applied the Eval rule to Node B, for example. Its left successor corresponds to the case where T_1 represents the ground term $\mathsf{0}$ and, thus, the goal $\mathsf{f}(T_1, T_2)$ unifies with the head of the first clause of the program. Here we can replace all occurrences of T_1 by $\mathsf{0}$, as (due to $T_1 \in \mathcal{G}$) $\mathsf{0}$ is the term represented by T_1. In contrast, as $T_2 \notin \mathcal{G}$, the replacement of T_2 with the fresh variable T_3 is not performed in the second backtracking goal $\mathsf{f}(\mathsf{0}, T_2)^2_1$. The right successor of Node B corresponds to all cases where the unification with the head of Clause (1) fails.

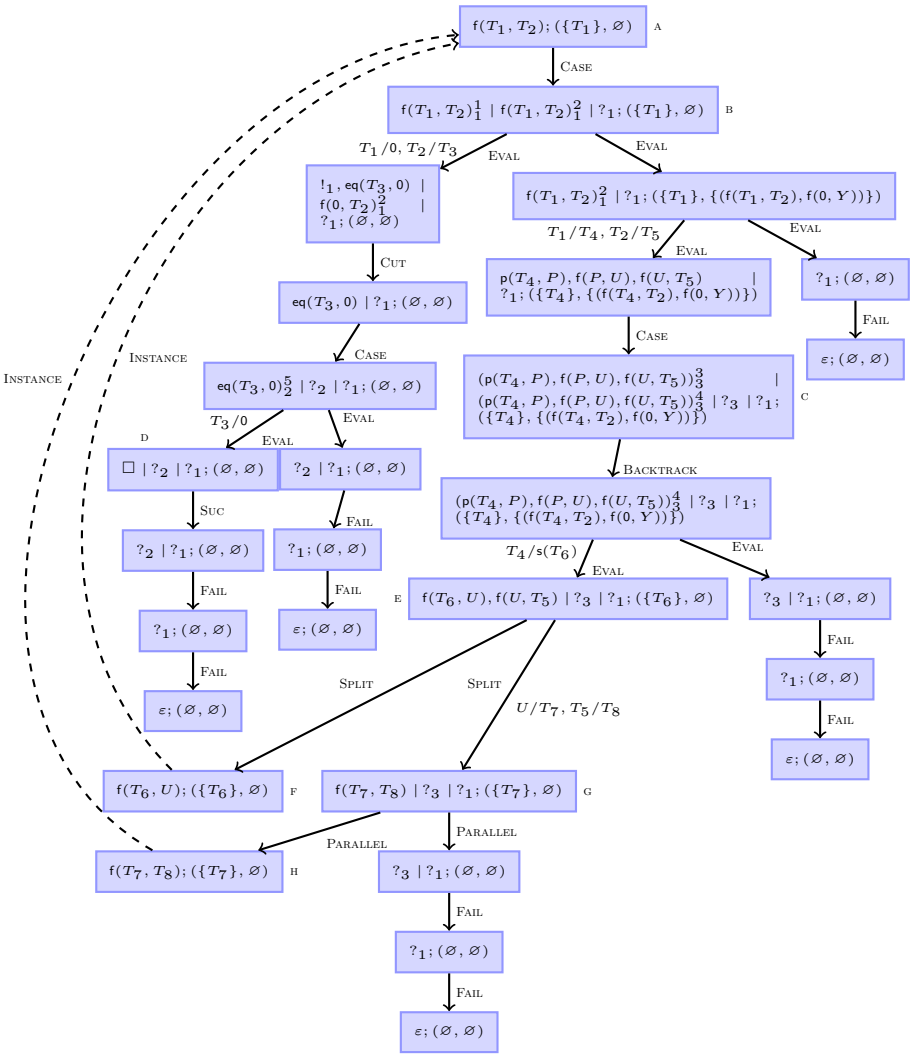


Fig. 3. Termination Graph for Ex. 1.

While the abstract EVAL rule is already sufficient for a sound simulation of all concrete derivations, the abstract BACKTRACK rule is virtually always needed for a successful termination analysis, since otherwise, the application of the abstract inference rules would almost always yield an infinite tree. To apply the abstract BACKTRACK rule to an abstract state, we have to ensure that this state does not represent any queries where the concrete EVAL rule would be applicable. In Fig. 3 we applied the BACKTRACK rule to Node C, for example. This is crucial for the success of the termination proof. As we know from the application of the EVAL rule to Node B, we are in the case where the first argument T_4 does not unify with $\mathsf{0}$. Hence, Clause (3) is not applicable to the first goal $\mathsf{p}(T_4, P)$ and

thus, we can safely apply the BACKTRACK rule to C.

With the first seven rules, we would obtain infinite trees for non-terminating programs. Even for terminating programs we may obtain infinite trees as there is no bound on the size of the terms represented by abstract variables. For a finite analysis we have to refer back to already existing states. This is done by the INSTANCE rule. The intuition for this rule is that we can refer back to a state representing a superset of queries compared to the current state. This can be ensured by finding a matcher μ which matches the more general state to the more specific state. This rule can also be used to generalize states instead of referring back to existing states. This is needed in case of "repeatedly growing" terms which would otherwise never lead to a state where we can find an already existing instance. Considering our example graph in Fig. 3, we applied the INSTANCE rule to refer Node F back to Node A with the matching substitution $\{T_1/T_6, T_2/U\}$.

Still, this is not enough to always obtain a finite termination graph. On the one hand, the evaluation of a program may yield growing backtracking sequences which never lead to a state with an already existing instance. On the other hand, the number of terms in a query may also grow and cause the same problem. For the first situation we need the PARALLEL rule which can separate the backtracking sequence into two states. The second problem is solved by the SPLIT rule which splits off the first term of a single query. Both rules may lose precision, but are often needed for a finite analysis. To reduce the loss of precision, we approximate the answer substitutions of evaluations for the first successors of SPLIT nodes using a groundness analysis. This analysis determines whether some variables will be instantiated by ground terms in every successful derivation of SPLIT's left child. Then in the right child, these variables can be added to $\mathcal{G}$ and all other variables are replaced by fresh abstract variables (this is necessary due to sharing). In Fig. 3 we applied the PARALLEL rule to Node G. In this way, we created its child H which is an instance of the earlier Node A. The SPLIT rule is used to separate the goal $\mathsf{f}(T_6, U)$ from the remainder of the state in Node E. The resulting child F is again an instance of A. For E's second child G, the groundness analysis found out that the variable U in the goal $\mathsf{f}(U, T_5)$ must be instantiated with a ground term T_7 during the evaluation of node F. Therefore, T_7 is added to the set $\mathcal{G}$ in Node G. This groundness information is important for the success of the termination proof. Otherwise, the state G would also represent non-terminating queries and hence, the termination proof would fail.

Using these rules, for Ex. 1 we obtain the termination graph depicted in Fig. 3. A *termination graph* is a finite graph where no rule is applicable to its leaves and where there is no cycle which only uses the INSTANCE rule. Note that by applying an adequate strategy, we can obtain a termination graph for *any* logic program automatically [18, Thm. 2]. To ease presentation, in the graph of Fig. 3, we always removed those abstract variables from the knowledge base that do not occur in any goal of the respective state. A termination graph without leaves that start with variables is called *proper*. (Our termination proof fails if the graph contains leaves starting with abstract variables, since they stand for any possible query.) Again, we refer to [18] for further details and explanations.

4 Transformation into Definite Logic Programs

We now explain the second stage of the transformation from [18], i.e., the transformation of termination graphs into definite logic programs. For more details and formal definitions see [18].

Termination graphs have the property that each derivation of the original program corresponds to a path through the termination graph. Thus, infinite derivations of the original program correspond to an infinite traversal of cycles in the graph. The basic idea of the transformation is to generate (cut-free) clauses for each cycle in the graph. Then termination of the (definite) logic program consisting of these clauses implies termination of the original program. To simulate the traversal of cycles, we generate clauses for paths starting at the child of an INSTANCE or SPLIT node or at the root node and ending in a SUC or INSTANCE node or in a left child of an INSTANCE or SPLIT node while not traversing other INSTANCE nodes or left children of INSTANCE or SPLIT nodes. The formal definition of paths for which we generate clauses is given below. Here, for a termination graph G, let $\textsc{Instance}(G)$ denote all nodes of G to which the rule INSTANCE has been applied (i.e., F and H). The sets $\textsc{Split}(G)$ and $\textsc{Suc}(G)$ are defined analogously. For any node n, let $Succ(i, n)$ denote the i-th child of n.

Definition 2 (Clause Path [18]). *A path $\pi = n_1 \dots n_k$ in G is a* clause path *iff $k > 1$ and*

- $n_1 \in Succ(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$ *or n_1 is the root of G,*
- $n_k \in \textsc{Suc}(G) \cup \textsc{Instance}(G) \cup Succ(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$,
- *for all $1 \leq j < k$, we have $n_j \notin \textsc{Instance}(G)$,*[5] *and*
- *for all $1 < j < k$, we have $n_j \notin Succ(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$.*

In the graph of Fig. 3 we find two clause paths ending in INSTANCE nodes. One path is from the root node A to the INSTANCE node F and one from the root node A to the INSTANCE node H. We introduce a fresh predicate symbol p_n for each node n, where these predicates have all distinct variables occurring in the node n as arguments.

For an INSTANCE node, however, we use the same predicate as for its child while applying the matching substitution used for the instantiation. Hence, for the nodes A, F, H in Fig. 3, we obtain the terms $\mathsf{p}_{\text{A}}(T_1, T_2)$, $\mathsf{p}_{\text{A}}(T_6, U)$, and $\mathsf{p}_{\text{A}}(T_7, T_8)$. To generate clauses for every clause path, we have to consider the substitutions along the paths and successively apply them to the heads of the new clauses. Thus, for the clause path from A to F, we obtain the clause $\mathsf{p}_{\text{A}}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p}_{\text{A}}(T_6, U)$.

However, for cycles traversing right children of SPLIT nodes, the newly generated clause should contain an additional intermediate body atom. This is due to the fact that the derivation along such a path is only possible if the goal corresponding to the left child of the respective SPLIT node is successfully evaluated

[5] Note that $n_k \in Succ(1, \textsc{Instance}(G))$ is possible although $n_{k-1} \notin \textsc{Instance}(G)$, since n_k may have more than one parent node in G.

first. Hence, we obtain the clause $\mathsf{p}_{\text{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\text{A}}(T_6, T_7), \mathsf{p}_{\text{A}}(T_7, T_8)$ for the path from A to H. To capture the evaluation of left children of SPLIT nodes, we also generate clauses corresponding to evaluations of left SPLIT children, i.e., paths in the graph from such nodes to SUC nodes, possibly traversing cycles first. Thus, the path from A to the only SUC node D is also a clause path. To transform it into a new clause, we have to apply the substitutions between the respective SPLIT node and the end of the path. For SUC nodes, we do not introduce new predicates. Hence, we obtain the fact $\mathsf{p}_{\text{A}}(\mathsf{0}, \mathsf{0})$ for the path from A to D. Thus, the resulting definite logic program for the termination graph from Fig. 3 is the following. Note that here, T_5, T_6, T_7, T_8 are considered as normal variables.

$$\mathsf{p}_{\text{A}}(\mathsf{0}, \mathsf{0}). \tag{6}$$

$$\mathsf{p}_{\text{A}}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p}_{\text{A}}(T_6, U). \tag{7}$$

$$\mathsf{p}_{\text{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\text{A}}(T_6, T_7), \mathsf{p}_{\text{A}}(T_7, T_8). \tag{8}$$

Below we give the formal definition for the clauses and queries generated for clause paths. To ease the presentation, we assume that for any path π, we do not traverse a BACKTRACK, FAIL, or SUC node or the right successor of an EVAL node after traversing the left successor of an EVAL node. The more general case can be found in [18] and also in the extended definitions and proofs in [21].

Definition 3 (Logic Programs from Termination Graph [18]). *Let G be a termination graph whose root n is $(f(T_1, \ldots, T_m), (\{T_{i_1}, \ldots, T_{i_k}\}, \varnothing))$. We define $\mathcal{P}_G = \bigcup_{\pi \text{ clause path in } G} \mathit{Clause}(\pi)$ and $\mathcal{Q}_G = \{\mathsf{p}_n(t_1, \ldots, t_m) \mid t_{i_1}, \ldots, t_{i_k}$ are ground$\}$. For a path $\pi = n_1 \ldots n_k$, let $\mathit{Clause}(\pi) = \mathit{Ren}(n_1)\sigma_\pi \leftarrow I_\pi, \mathit{Ren}(n_k)$. For $n \in \text{SUC}(G)$, $\mathit{Ren}(n)$ is $\square$ and for $n \in \text{INSTANCE}(G)$, it is $\mathit{Ren}(\mathit{Succ}(1, n))\mu$ where μ is the substitution associated with the* INSTANCE *node n. Otherwise, $\mathit{Ren}(n)$ is $\mathsf{p}_n(\mathcal{V}(n))$ where $\mathcal{V}(S; KB) = \mathcal{V}(S)$.*

Finally, σ_π and I_π are defined as follows. Here for a path $\pi = n_1 \ldots n_j$, the substitutions μ and σ are the labels on the outgoing edge of $n_{j-1} \in \text{SPLIT}(G)$ and $n_{j-1} \in \text{EVAL}(G)$, respectively.

$$\sigma_{n_1 \ldots n_j} = \begin{cases} \mathit{id} & \text{if } j = 1 \\ \sigma_{n_1 \ldots n_{j-1}}\, \mu & \text{if } n_{j-1} \in \text{SPLIT}(G),\ n_j = \mathit{Succ}(2, n_{j-1}) \\ \sigma_{n_1 \ldots n_{j-1}}\, \sigma & \text{if } n_{j-1} \in \text{EVAL}(G),\ n_j = \mathit{Succ}(1, n_{j-1}) \\ \sigma_{n_1 \ldots n_{j-1}} & \text{otherwise} \end{cases}$$

$$I_{n_j \ldots n_k} = \begin{cases} \square & \text{if } j = k \\ \mathit{Ren}(\mathit{Succ}(1, n_j))\sigma_{n_j \ldots n_k},\ I_{n_{j+1} \ldots n_k} & \text{if } n_j \in \text{SPLIT}(G), n_{j+1} = \mathit{Succ}(2, n_j) \\ I_{n_{j+1} \ldots n_k} & \text{otherwise} \end{cases}$$

Unfortunately, in our example, the generated program (6)-(8) is not (universally) terminating for all queries of the form $\mathsf{p}_{\text{A}}(t_1, t_2)$ where t_1 is a ground term. To see this, consider the query $\mathsf{p}_{\text{A}}(\mathsf{s}(\mathsf{s}(\mathsf{0})), Z)$. We obtain the following derivation.

$$\mathsf{p}_{\text{A}}(\mathsf{s}(\mathsf{s}(\mathsf{0})), Z) \vdash_{(8)} \mathsf{p}_{\text{A}}(\mathsf{s}(\mathsf{0}), T_7), \mathsf{p}_{\text{A}}(T_7, Z) \vdash_{(7)} \mathsf{p}_{\text{A}}(\mathsf{0}, U), \mathsf{p}_{\text{A}}(T_7, Z) \vdash_{(6)} \mathsf{p}_{\text{A}}(T_7, Z)$$

The last goal has infinitely many successful derivations. The reason why the transformation fails is that in the generated logic program, we cannot distinguish between the evaluation of intermediate goals and the traversal of cycles of the termination graph, since we only have one evaluation mechanism. We often encounter such problems when the original program has clauses whose body contains at least two atoms $q_1(\ldots)$, $q_2(\ldots)$, where both predicates q_1 and q_2 have recursive clauses and where the call of q_2 depends on the result of q_1. This is a very natural situation occurring in many practical programs (cf. our experiments in Sect. 6). It is also the case in our example for the second clause (2) (here we have the special case where both q_1 and q_2 are equal).

5 Transformation into Dependency Triple Problems

To solve the problem illustrated in the last section, we modify the second stage of the transformation to construct dependency triple problems [17] instead of definite logic programs. The advantage of dependency triple problems is that they support two different kinds of evaluation which suit our needs to handle the evaluation of intermediate goals and the traversal of cycles differently.

The basic structure in the dependency triple framework is very similar to a clause in logic programming. Indeed, a *dependency triple* (DT) [14] is just a clause $H \leftarrow I, B$ where H and B are atoms and I is a sequence of atoms. Intuitively, such a DT states that a call that is an instance of H can be followed by a call that is an instance of B if the corresponding instance of I can be proven.

Here, a "derivation" is defined in terms of a chain. Let $\mathcal{D}$ be a set of DTs, $\mathcal{P}$ be the program under consideration, and $\mathcal{Q}$ be the class of queries to be analyzed.[6] A (possibly infinite) sequence $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathcal{D}$ is a $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$-*chain* iff there are substitutions θ_i, σ_i and an $A \in \mathcal{Q}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, we have $\sigma_i \in Answer(\mathcal{P}, I_i\theta_i)$ and $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$. Such a tuple $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ is called a *dependency triple problem* and it is *terminating* iff there is no infinite $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$-chain.

As an example, consider the DT problem $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ with $\mathcal{D} = \{d_1\}$ where $d_1 = \mathsf{p}(\mathsf{s}(X), Y) \leftarrow \mathsf{eq}(X, Z), \mathsf{p}(Z, Y)$, $\mathcal{Q} = \{\mathsf{p}(t_1, t_2) \mid t_1 \text{ is ground}\}$, and $\mathcal{P} = \{\mathsf{eq}(X, X)\}$. Now, "$d_1\ d_1$" is a $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ chain. To see this, assume that $A = \mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{0})$. Then $\theta_0 = \{X/\mathsf{s}(\mathsf{0}), Y/\mathsf{0}\}$, $\sigma_0 = \{Z/\mathsf{s}(\mathsf{0})\}$, and $\theta_1 = \{X/\mathsf{0}, Y/\mathsf{0}\}$.

In this section we show how to synthesize a DT problem from a termination graph built for a logic program with cut such that termination of the DT problem implies termination of the original program w.r.t. the set of queries for which the termination graph was constructed. This approach is far more powerful than first constructing the cut-free logic program as in Sect. 4 and then transforming it into a DT problem. Indeed, the latter approach would fail for our leading example (as the cut-free program (6)-(8) is not terminating), whereas the termination proof succeeds when generating DT problems directly from the termination graph.

Like in the transformation into definite logic programs from [18], we have to prove that there is no derivation of the original program which corresponds to a

[6] For simplicity, we use a set of initial queries instead of a general call set as in [17].

path traversing the cycles in the termination graph infinitely often.

To this end, we build a set $\mathcal{D}$ of DTs for paths in the graph corresponding to cycles and a set $\mathcal{P}$ of program clauses for paths corresponding to the evaluation of intermediate goals. For the component $\mathcal{Q}$ of the resulting DT problem, we use a set of queries based on the root node.

We now illustrate how to use this idea to prove termination of Ex. 1 by building a DT problem for the termination graph from Fig. 3. We again represent each node by a fresh predicate symbol with the different variables occurring in the node as arguments. However, as before, for an INSTANCE node we take the predicate symbol of its child instead where we apply the matching substitution used for the respective instantiation and we do not introduce any predicates for SUC nodes. But in contrast to Sect. 4 and [18], we use different predicates for DTs and program clauses. In this way, we can distinguish between atoms used to represent the traversal of cycles and atoms used as intermediate goals.

To this end, instead of clause paths we now define *triple paths* (that are used to build the component $\mathcal{D}$ of the resulting DT problem) and *program paths* (that are used for the component $\mathcal{P}$ of the DT problem). Triple paths lead from the root node or the successor of an INSTANCE node to the beginning of a cycle, i.e., to an INSTANCE node or the successor of an INSTANCE node where we do not traverse other INSTANCE nodes or their children. Compared to the clause paths of Def. 2, triple paths do not start or stop in left successors of SPLIT nodes, but in contrast they may traverse them. Since finite computations are irrelevant for building infinite chains, triple paths do not stop in SUC nodes either.

Thus, we have two triple paths from the root node A to the INSTANCE nodes F and H. We also have to consider intermediate goals, but this time we use a predicate symbol p_{A} for the intermediate goal and a different predicate symbol q_{A} for the DTs. Hence, we obtain $\mathsf{q}_{\text{A}}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}_{\text{A}}(T_6, U)$ and $\mathsf{q}_{\text{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\text{A}}(T_6, T_7), \mathsf{q}_{\text{A}}(T_7, T_8)$.

Concerning the evaluation for left successors of SPLIT nodes, we build program clauses for the component $\mathcal{P}$ of the DT problem. The clauses result from *program paths* in the termination graph. These are paths starting in a left successor of a SPLIT node and ending in a SUC node. However, in addition to the condition that we do not traverse INSTANCE nodes or their successors, such a path may also not traverse another left successor of a SPLIT node as we are only interested in completely successful evaluations. Thus, the right successor of a SPLIT node must be reached. As the evaluation for left successors of SPLIT nodes may also traverse cycles before it reaches a fact, we also have to consider paths starting in the left successor of a SPLIT node or the successor of an INSTANCE node and ending in an INSTANCE node, a successor of an INSTANCE node, or a SUC node. Compared to the clause paths of Def. 2, the only difference is that program paths do not stop in left successors of SPLIT nodes. Hence, we have two program paths from the root node A to the only SUC node D and to the INSTANCE node H. We also have to consider intermediate goals for the constructed clauses. Thus, we result in the fact $\mathsf{p}_{\text{A}}(0, 0)$ and the clause $\mathsf{p}_{\text{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\text{A}}(T_6, T_7), \mathsf{p}_{\text{A}}(T_7, T_8)$.

So we obtain the DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ for the termination graph G from Fig. 3 where $\mathcal{D}_G$ contains the DTs

$$\mathsf{q}_{\mathrm{A}}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}_{\mathrm{A}}(T_6, U).$$
$$\mathsf{q}_{\mathrm{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\mathrm{A}}(T_6, T_7), \mathsf{q}_{\mathrm{A}}(T_7, T_8).$$

and $\mathcal{P}_G$ consists of the following clauses.

$$\mathsf{p}_{\mathrm{A}}(\mathsf{0}, \mathsf{0}).$$
$$\mathsf{p}_{\mathrm{A}}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_{\mathrm{A}}(T_6, T_7), \mathsf{p}_{\mathrm{A}}(T_7, T_8).$$

Hence, there are three differences compared to the program (6)-(8) we obtain following Def. 3: (i) we do not obtain the fact $\mathsf{q}_{\mathrm{A}}(\mathsf{0}, \mathsf{0})$ for the dependency triples; (ii) we do not obtain the clause $\mathsf{p}_{\mathrm{A}}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p}_{\mathrm{A}}(T_6, U)$; and (iii) we use p_{A} instead of q_{A} in the intermediate goal of the second dependency triple. The latter two differences are essential for success on this example as a ground "input" for p_{A} on the first argument guarantees a ground "output" on the second argument. Note that this is not the case for the program according to Def. 3.

In our example, $\mathcal{Q}_G$ contains all queries $\mathsf{q}_{\mathrm{A}}(t_1, t_2)$ where t_1 is ground. Then this DT problem is easily shown to be terminating by our automated termination prover AProVE (or virtually any other tool for termination analysis of definite logic programs by proving termination of $\mathcal{D}_G \cup \mathcal{P}_G$ for the set of queries $\mathcal{Q}_G$).

Now we formally define how to obtain a DT problem from a termination graph. To this end, we first need the notions of triple and program paths to characterize those paths in the termination graph from which we generate the DTs and clauses for the DT problem.

Definition 4 (Triple Path, Program Path). *A path $\pi = n_1 \dots n_k$ in G is a* triple path *iff $k > 1$ and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \textsc{Instance}(G))$ *or n_1 is the root of G,*
- $n_k \in \textsc{Instance}(G) \cup \mathit{Succ}(1, \textsc{Instance}(G))$*,*
- *for all $1 \leq j < k$, we have $n_j \notin \textsc{Instance}(G)$, and*
- *for all $1 < j < k$, we have $n_j \notin \mathit{Succ}(1, \textsc{Instance}(G))$.*

A path $\pi = n_1 \dots n_k$ in G is a program path *iff $k > 1$ and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$*,*
- $n_k \in \textsc{Suc}(G) \cup \textsc{Instance}(G) \cup \mathit{Succ}(1, \textsc{Instance}(G))$*,*
- *for all $1 \leq j < k$, we have $n_j \notin \textsc{Instance}(G)$,*
- *for all $1 < j < k$, we have $n_j \notin \mathit{Succ}(1, \textsc{Instance}(G))$, and*
- *for all $1 < j \leq k$, we have $n_j \notin \mathit{Succ}(1, \textsc{Split}(G))$.*

Now, we define the DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ for a termination graph G. The set $\mathcal{D}_G$ contains clauses for all triple paths, the queries $\mathcal{Q}_G$ contain all instances represented by the root node, and $\mathcal{P}_G$ contains clauses for all program paths.

Definition 5 (DT Problem from Termination Graph). *Let G be a termination graph whose root is $(f(T_1, \dots, T_m), (\{T_{i_1}, \dots, T_{i_k}\}, \varnothing))$. The DT problem*

$(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ *is defined by* $\mathcal{D}_G = \bigcup_{\pi \text{ triple path in } G} \mathit{Triple}(\pi)$, $\mathcal{Q}_G = \{\mathsf{q}_n(t_1, \ldots, t_m) \mid t_{i_1}, \ldots, t_{i_k}$ *are ground*$\}$ *where* q_n *is the fresh predicate chosen for the root node by* Ren_t, *and* $\mathcal{P}_G = \bigcup_{\pi \text{ program path in } G} \mathit{Clause}(\pi)$.

For a path $\pi = n_1 \ldots n_k$, *we define* $\mathit{Clause}(\pi) = \mathit{Ren}(n_1)\sigma_\pi \leftarrow I_\pi, \mathit{Ren}(n_k)$ *and* $\mathit{Triple}(\pi) = \mathit{Ren}_t(n_1)\sigma_\pi \leftarrow I_\pi, \mathit{Ren}_t(n_k)$. *Here, Ren and* Ren_t *are defined as in Def. 3 but* Ren_t *uses* q_n *instead of* p_n *for any node* n.

We now state the central theorem of this paper where we prove that termination of the resulting DT problem implies termination of the original logic program with cut for the set of queries represented by the root state of the termination graph. For the proof we refer to [21].

Theorem 6 (Correctness). *If* G *is a proper termination graph for a logic program* $\mathcal{P}$ *such that* $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ *is terminating, then all concrete states represented by* G*'s root node have only finite derivations w.r.t. the inference rules of Fig. 1.*

6 Implementation and Experiments

We implemented the new transformation in our fully automated termination prover AProVE and tested it on all 402 examples for logic programs from the Termination Problem Data Base (TPDB) [23] used for the annual international Termination Competition [22]. We compared the implementation of the new transformation (AProVE DT) with the implementation of the previous transformation into definite logic programs from [18] (AProVE Cut), and with a direct transformation into term rewrite systems ignoring cuts (AProVE Direct) from [16]. We ran the different versions of AProVE on a 2.67 GHz Intel Core i7 and, as in the international Termination Competition, we used a timeout of 60 seconds for each example. For all versions we give the number of examples which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total runtime ("Total") in seconds. For those examples where termination could be proved, we indicate how many of them contain cuts. For the details of this empirical evaluation and to run the three versions of AProVE on arbitrary examples via a web interface, we refer to `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.

	AProVE Direct	AProVE Cut	AProVE DT
Successes	243	259	**315**
– with cut	10	78	**82**
– without cut	233	181	**233**
Failures	144	129	**77**
Timeouts	15	14	**10**
Total	2485.7	3288.0	**2311.6**

Table 1. Experimental results on the Termination Problem Data Base

As shown in Table 1, the new transformation significantly increases the number of examples that can be proved terminating. In particular, we obtain 56 additional proofs of termination compared to the technique of [18]. And indeed, for all examples where AProVE Cut succeeds, AProVE DT succeeds, too. Note that while [18] is very successful on examples with cut, its performance is significantly worse than that of AProVE Direct on the other examples of the TPDB.

While we conjecture that our new improved transformation is *always* more powerful than the transformation from [18], a formal proof of this conjecture is not straightforward. The reason is that the clause paths of [18] differ from the triple and program paths in our new transformation. Hence we cannot compare the transformed problems directly.

In addition to being more powerful, the new version using dependency triples is also more efficient than any of the two other versions, resulting in fewer timeouts and a total runtime that is less than the one of the direct version and only 70% of the version corresponding to [18]. However, AProVE DT sometimes spends more time on failing examples, as the new transformation may result in DT problems where the termination proof fails later than for the logic programs resulting from [18].

7 Conclusion

We have shown that the termination graphs introduced by [18] can be used to obtain a transformation from logic programs with cut to dependency triple problems. Our experiments show that this new approach is both considerably more powerful and more efficient than a translation to definite logic programs as in [18]. As the dependency triple framework allows a modular and flexible combination of arbitrary termination techniques from logic programming and even term rewriting, the new transformation to dependency triples can be used as a frontend to any termination tool for logic programs (by taking the union of $\mathcal{D}_G$ and $\mathcal{P}_G$ in the resulting DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$) or term rewriting (by using the transformation of [17]).

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.
2. T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1,2):133–178, 2000.
3. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.
4. M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.
5. D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19,20:199–260, 1994.
6. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, New York, 1996.

7. J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.
9. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
10. N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.
11. J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.
12. M. Marchiori. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*, volume 1101 of *LNCS*, pages 375–390, 1996.
13. F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.
14. M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.
15. M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. *Theory and Practice of Logic Programming*, 2010. To appear.
16. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 10(1):2:1–49, 2009.
17. P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In *LOPSTR '09*, volume 6037 of *LNCS*, pages 37–51, 2010.
18. P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut. In *ICLP '10*, *Theory and Practice of Logic Programming*, 2010. To appear. Extended version and experimental details at `http://aprove.informatik.rwth-aachen.de/eval/cut/`.
19. A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.
20. T. Ströder. Towards Termination Analysis of Real Prolog Programs. Diploma Thesis, RWTH Aachen, 2010. `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.
21. T. Ströder, P. Schneider-Kamp, and J. Giesl. Dependency Triples for Improving Termination Analysis of Logic Programs with Cut. Technical Report AIB 2010-12, RWTH Aachen, 2010. `http://aib.informatik.rwth-aachen.de/`.
22. The Termination Competition. `http://www.termination-portal.org/wiki/Termination_Competition`.
23. The Termination Problem Data Base 7.0 (December 11, 2009). `http://termcomp.uibk.ac.at/status/downloads/`.
24. C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994.

A Hybrid Approach to Conjunctive Partial Deduction*

Germán Vidal

MiST, DSIC, Universidad Politécnica de Valencia, Spain
gvidal@dsic.upv.es

Abstract. Conjunctive partial deduction is a well-known technique for the partial evaluation of logic programs. The original formulation follows the so called online approach where all termination decisions are taken on-the-fly. In contrast, offline partial evaluators first analyze the source program and produce an annotated version so that the partial evaluation phase should only follow these annotations to ensure the termination of the process. In this work, we introduce a lightweight approach to conjunctive partial deduction that combines some of the advantages of both online and offline styles of partial evaluation.

1 Introduction

The main goal of *partial evaluation* [8] is program specialization. Essentially, given a program and *part* of its input data—the so called *static* data—a partial evaluator returns a new, residual program which is specialized for the given data. Basically, given a program and a partial call, the essential components of partial evaluation are: the construction of a *finite* representation—generally a graph—of the possible executions of the program call, followed by the systematic extraction of a *residual* program (i.e., the partially evaluated program) from this graph. Intuitively, optimization can be achieved by compressing paths in the graph and by renaming expressions while removing unnecessary function symbols.

In the context of logic programming, partial evaluation is also known as *partial deduction* [13]. Roughly speaking, given a logic program P and a set of atoms $\mathcal{A} = \{A_1, \ldots, A_n\}$, one should construct finite—possibly incomplete—SLD trees for the atomic goals $\leftarrow A_1, \ldots, \leftarrow A_n$, such that every leaf is either successful, a failure, or only contains atoms that are instances of $\{A_1, \ldots, A_n\}$; this is the so-called *closedness* condition [13]. The residual program has a *resultant* of the form $A_i\sigma \leftarrow Q$ for every root-to-leaf derivation $\leftarrow A_i \leadsto^*_\sigma \leftarrow Q$ in the SLD trees.

From an algorithmic perspective, one usually starts with an initial set $\mathcal{A}_1 = \{A_1\}$ and builds a finite SLD tree for $\leftarrow A_1$; then, every atom in the leaves of this SLD tree which is not an instance of A_1 is added to the set, thus obtaining $\mathcal{A}_2$, and so forth. In order to keep the sequence $\mathcal{A}_1, \mathcal{A}_2, \ldots$ finite, some *generalization* is often required (e.g., by replacing some predicate arguments by fresh variables).

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2010/042, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08).

One of the main drawbacks of partial deduction is the fact that the atoms in the leaves of every SLD tree are partially evaluated independently. Usually, this implies a significant loss of accuracy. In order to overcome this drawback, a new framework called *conjunctive partial deduction* (CPD) was introduced [5]. Loosely speaking, the main difference with standard partial deduction is that it considers the partial evaluation of non-atomic goals. Here, one usually starts with an initial set $\mathcal{C}_1 = \{C_1\}$, where C_1 is a conjunction of atoms, and builds a finite SLD tree for $\leftarrow C_1$; then, every leaf in the SLD tree is added to the set and so forth. Trivially, this process is usually infinite. Now, in order to keep the sequence $\mathcal{C}_1, \mathcal{C}_2, \ldots$ finite, generalization does not suffice and the conjunctions in the leaves of the SLD trees should often be *split up* to avoid conjunctions that keep growing infinitely.

Depending on *when* control issues—like deciding which atoms should or should not be unfolded or how conjunctions should be split up—are addressed, two main approaches to partial evaluation can be distinguished. In *offline* approaches to partial evaluation these decisions are taken beforehand by means of static analysis (where we know which parameters are known but not their values). In contrast, *online* partial evaluators take decisions on the way (so that actual values of static data are available). While offline partial evaluators are usually faster, online ones produce more accurate results (though, from a theoretical point of view, they are equally powerful [3]).

In this work, we introduce a hybrid approach to CPD of definite logic programs as follows:

Pre-processing stage: First, we apply a simple call and success pattern analysis that identifies which predicate arguments will be ground *at run-time*. Note that this contrasts with previous approaches where run-time information is hardly considered. A termination analysis is then applied to identify possibly non-terminating calls (for the computed call patterns). Finally, we use a syntactic characterization to identify *non-regular* predicates whose unfolding might give rise to infinitely growing conjunctions during partial evaluation.

Partial evaluation: This stage follows the scheme of traditional CPD but includes some significant differences: splitting of conjunctions is determined from the information gathered by the call and success pattern analysis and from the computed set of non-regular predicates (rather than inspecting the history of partially evaluated queries); secondly, our procedure includes no generalization but simply gives up when termination cannot be ensured (thus returning calls to the original, not renamed predicates); finally, non-leftmost unfolding is allowed as long as the selected atom is terminating (at run-time) according to the termination analysis performed in the pre-processing stage.

Post-processing stage: Finally, we extract the residual clauses from the computed partial evaluations using the standard notion of resultant. In principle, we only compute one-step resultants (which increases the opportunities for *folding* back the calls of the SLD-trees). As in the original CPD framework, all conjunctions are renamed (except for those where termination could not be ensured and we gave up). Moreover, we apply a standard post-unfolding transformation where calls to *intermediate* predicates are unfolded.

Our technique can be seen as a lightweight approach to CPD that combines some of the advantages of both online and offline styles of partial evaluation. Some of the most noteworthy features of the new proposal are the following:

- The new scheme is conceptually simpler since the main partial evaluation stage just performs unfolding and (a limited form of) splitting. Moreover, we do not require SLD trees to be *weakly fair* [5], sometimes a too restrictive condition, but rely on avoiding the non-leftmost unfolding of potentially non-terminating calls (at run-time). We note that unfolding non-terminating atoms is not a problem for ensuring termination of partial evaluation (since we use an embedding ordering for this purpose) but is essential to preserve correctness w.r.t. finite failures.
- Our scheme is potentially faster since some of the most expensive operations, generalization and splitting, do not exist anymore (generalization) or are much simpler (splitting) thanks to the use of information gathered by the static analyses.
- In contrast to previous approaches, we keep some run-time information at partial evaluation time and ensure that it is correctly propagated; actually, this is one of the main reasons to avoid generalization, since it might involve a dramatic loss of groundness and sharing information. This could be essential to generate residual programs including other, orthogonal optimizations. For instance, one could consider the replacement of some sequential conjunctions by concurrent conjunctions in residual programs; for this purpose, run-time groundness information is needed.

The new approach is still rather preliminary and further research is needed to check its viability over larger programs including built-in's and negation. Nevertheless, a prototype implementation has been undertaken and can be tested at `http://kaz.dsic.upv.es/lite.html`. Despite the simplicity of the prototype (a few hundred lines of Prolog code), the results for definite logic programs are not far from those obtained with mature CPD systems like ECCE [11].

2 Pre-Processing Stage

Our pre-processing stage consists of three different analyses. The first one is a simple call and success pattern analysis like that introduced in [12]. Basically, the analysis infers for every predicate p/n a number of call/success patterns of the form $p/n : in \mapsto out$ such that in and out are subsets of $\{1, \ldots, n\}$ denoting the arguments out of p/n which are definitely ground after a successful derivation, assuming that it is called with ground arguments in. This information could also be provided by the user (as in Mercury [14]).

The second analysis is a standard left-termination analysis (i.e., an analysis for universal termination under Prolog's left-to-right computation rule) like those based on the abstract binary unfoldings [4], size-change analysis [1], etc. This information, together with the call and success pattern analysis, will be essential to determine when non-leftmost unfolding is admissible at specialization time.

Finally, we introduce a syntactic characterization that allows us to identify which predicate calls might give rise to queries with infinite atoms. Our formulation is somehow a generalization of the notion of *B-stratifiable* programs in

[7]. The main difference is that we consider a flexible computation rule while [7] considers a fixed left-to-right rule and thus their notion is more restrictive.

In the following, we say that the *call graph* of a program P is a directed graph that contains the predicate symbols of P as vertices and an edge from predicate p/n to predicate q/m for each clause $p(t_1, \ldots, t_n) \leftarrow body$ and atom $q(s_1, \ldots, s_m)$ of *body*.

Definition 1 (strongly regular logic programs). *Let P be a logic program and let $CG_1, \ldots, CG_n$ be the strongly connected components (SCC) in the call graph of P. We say that P is strongly regular if there is no clause $p(t_1, \ldots, t_n) \leftarrow body$ such that body contains two atoms $q(s_1, \ldots, s_m)$ and $r(l_1, \ldots, l_k)$ such that q/m and r/k belong to the same SCC of p/n.*

Intuitively speaking, strongly regular programs cannot produce infinitely growing conjunctions at partial evaluation time when a flexible computation rule is considered.

When a program P is not strongly regular, we identify the predicates that are responsible of violating the strongly regular condition: we say that a predicate p/n is *non-regular* if there is a clause $p(t_1, \ldots, t_n) \leftarrow body$ and *body* contains two atoms with predicates q/m and r/k that belong to the same SCC of p/n.

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time.

Example 1. Consider the following Prolog program from the DPPD library [10]:

```
applast(L,X,Last) :- append(L,[X],LX), last(Last,LX).
last(X,[X]).
last(X,[H|T]) :- last(X,T).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Here, there are three SCCs, {`applast/3`}, {`append/3`} and {`last/2`}, but no clause violates the strongly regular condition. In contrast, the following program (also from the DPPD library [10]):

```
flipflip(XT,YT) :- flip(XT,TT), flip(TT,YT).
flip(leaf(X),leaf(X)).
flip(tree(L,I,R),tree(FR,I,FL)) :- flip(L,FL), flip(R,FR).
```

is not strongly regular. Here, we have two SCCs, {`flipflip/2`} and {`flip/2`}, and the second clause of `flip/2` violates the strongly regular condition. As a consequence, we say that `flip/2` is a non-regular predicate.

3 Partial Evaluation Stage

In this section, we present the main stage of our CPD procedure. In the following, sequences of objects (e.g., sequences of atoms, call patterns, etc) are just denoted by juxtaposition. Given a sequence $ts = t_1 \cdots t_n$, we let $ts|_i = t_i$, $1 \leq i \leq n$, and $ts|_{i..j} = t_i \cdots t_j$ for all $i \leq j$. Also, $|ts| = n$ denotes the length of the

sequence. We consider the usual list notation over sequences, so that the empty sequence is denoted by $[\,]$ and the sequence with first element q and rest qs is denoted by $q : qs$. Given sequences $qs = q_1 \cdots q_n$ and $qs' = q'_1 \cdots q'_m$, we let $qs[qs']_i = q_1 \cdots q_{i-1} q'_1 \cdots q'_m q_{i+1} \cdots q_n$. We say that two queries qs and qs' are *variants*, denoted by $qs \approx qs'$, if there is a renaming substitution σ such that $qs\sigma = qs'$

At partial evaluation time, we avoid infinite unfolding by means of a well-known strategy based on the use of the *homeomorphic embedding* ordering [9]. The embedding relation $\trianglerighteq$ is defined as the least relation satisfying

- $x \trianglerighteq y$ for all variables x, y;
- $f(t_1, \ldots, t_n) \trianglerighteq s$ if $t_i \trianglerighteq s$ for some $i \in \{1, \ldots, n\}$;
- $f(t_1, \ldots, t_n) \trianglerighteq f(s_1, \ldots, s_n)$ if $t_i \trianglerighteq s_i$ for all $i = 1, \ldots, n$.

Given an SLD-resolution step

$$\leftarrow q_1 \cdots q_i \cdots q_n \;\leadsto_\sigma\; \leftarrow (q_1 \cdots q_{i-1} q'_1 \cdots q'_m q_{i+1} \cdots q_n)\sigma$$

with selected atom q_i using clause $q \leftarrow q'_1 \cdots q'_m$, $\sigma = mgu(q_i, q)$, we say that q_i is the (covering) *ancestor* [2] of atoms $q'_1\sigma, \ldots, q'_m\sigma$. This notion is extended to SLD-derivations in the natural way.

Basically, we ensure termination by avoiding the unfolding of those calls that embed some of their ancestors.

As it is common practice (see, e.g., [6]), our partial evaluation semantics distinguishes two levels: a *global level* which is concerned on keeping the sequence of partially evaluated queries finite, and a *local level* that controls the unfolding of each query. *Global states* have the form $\langle\langle qs_1 \cdots qs_n,\ ps_1 \cdots ps_n,\ gs\rangle\rangle$ where $qs_1 \cdots qs_n$ is a sequence of queries, $ps_1 \cdots ps_n$ is the associated sequence of call patterns (i.e., every ps_i denotes the sequence of call patterns for the atoms in the query qs_i), and gs is the set of already partially evaluated queries.

The *initial global state* has the form $\langle\langle qs, ps, \emptyset\rangle\rangle$, where qs is the input query for the partial evaluation process and ps is the sequence of call patterns for the atoms in this query. From this state, we get an *initial local state* $\langle qs, \emptyset, ps, \emptyset, \{qs\}\rangle$ that starts the actual partial evaluation process. The rules of the global level can be seen in Fig. 1 and are relatively simple:

- rule restart initiates the unfolding of every query in the sequence which does not embed some previously partially evaluated query;
- rule stop discards all other queries, where $\langle\langle\ \rangle\rangle$ denotes a *final* global state.

Observe that rule restart is nondeterministic and, thus, one usually gets a tree structure. Also, note that the call pattern in the derived state of rule restart is not exactly ps_i but ps_i^r: here, we denote with ps_i^r a refinement of ps_i by taking into account the propagation of groundness information assuming the successful evaluation of all previous queries in the sequence. This information can easily be obtained from the call and success pattern analysis; here, we skip the technical details to keep the presentation simple.

Local states have the form $\langle qs, as, ps, ls, gs\rangle$ where qs is a query, as is a sequence of ancestors for qs (i.e., $as|_i$ is the set of ancestors of atom $qs|_i$), ps is

a sequence of call patterns (i.e., $ps|_i$ denotes the call pattern of atom $qs|_i$), ls is the *local stack* which stores the set of queries already processed in the local level, and gs is the *global stack* which stores the set of queries already processed in the global level. While the local stack is mainly used for memoization, the global stack is needed to ensure the termination of the partial evaluation process.

Before we introduce the transition rules of the local level, we need some preparatory definitions.

Definition 2 (unfoldable atom, unfold). *Let $qs = q_1 \cdots q_n$ be a query, $n \geq 1$, and let $as = a_1 \cdots a_n$ be the associated sequence of ancestors. We say that an atom $qs|_i$, $1 \leq i \leq |qs|$, is unfoldable if*

1. *there is no $q' \in as|_i$ such that $qs|_i \trianglerighteq q'$ and*
2. *$qs|_i$ is terminating for the call pattern $ps|_i$ (according to the termination analysis performed in the pre-processing stage).*

In general, there might be more than one unfoldable atom in a query. Here, we let function unfold(qs, as) *return the index of the leftmost unfoldable atom.*

While the first condition above is required to avoid infinite unfolding, the second one is need to ensure the correctness of CPD w.r.t. finite failures (instead of requiring the construction of *weakly fair* SLD-trees as in [5]).

In our calculus, we consider two different forms of splitting. The first one is based on the notion of *independence* and allows us to split up a query when there are two subsequences that do not share variables and, thus, can be evaluated independently without any (serious) loss of accuracy.

Definition 3 (independent splitting, i-split). *Let $qs = q_1 \cdots q_n$ be a query, $n \geq 2$, and $ps = p_1 \cdots p_n$ be the associated call patterns. We say that the pair of natural numbers $\langle i, j \rangle$ is an independent splitting for qs and ps if*

- $1 \leq i < j \leq |qs|$ *and*
- $fvars(qs|_{1..i}, ps|_{1..i}) \cap fvars(qs|_{i+1..j}, ps|_{i+1..j}) = \emptyset$,

where the auxiliary function $fvars(qs, ps)$ returns the (possibly) non-ground variables of qs according to the call patterns ps (i.e., all variables in qs but those which appear in an argument addressed by ps).

In general, there might be more than one independent splitting for a given query and call patterns. Here, we let i-split(qs, ps) *return a pair of numbers $\langle i, j \rangle$ that maximize the value of j (hopefully $j = |qs|$).*[1]

E.g., given the query $qs = \texttt{append}(\texttt{X}, \texttt{Y}, \texttt{L}_1)\texttt{append}(\texttt{X}, \texttt{Z}, \texttt{L}_2)\texttt{append}(\texttt{L}_1, \texttt{L}_2, \texttt{R})$ and the call patterns $ps = [1,2][1,2][\,]$, we have i-split$(qs, ps) = \langle 1, 2 \rangle$ (i.e., qs is split into two independent subsequences $qs|_{1..1} = \texttt{append}(\texttt{X}, \texttt{Y}, \texttt{L}_1)$ and $qs|_{2..2} = \texttt{append}(\texttt{X}, \texttt{Z}, \texttt{L}_2)$ and the remaining sequence is $qs|_{3..3} = \texttt{append}(\texttt{L}_1, \texttt{L}_2, \texttt{R})$), since $fvars(\texttt{append}(\texttt{X}, \texttt{Y}, \texttt{L}_1), [1,2]) = \{L_1\}$, $fvars(\texttt{append}(\texttt{X}, \texttt{Z}, \texttt{L}_2), [1,2]) = \{L_2\}$ and the intersection is $\emptyset$.

[1] In [5], the notion of *maximally connected subconjunction* is introduced for a similar purpose.

Note that, given i-split$(qs, ps) = \langle i, j\rangle$, both $qs|_{1..i}$ and $qs|_{i+1..j}$ are independent according to ps but might share variables with $qs|_{j+1..n}$ (so that splitting up the query might involve some loss of accuracy anyway). Moreover, note that call patterns refer to ground arguments at *run-time*, so it might happen that $qs|_{1..i}$ and $qs|_{i+1}..j$ are not independent at partial evaluation time.

Our second form of splitting often involves a serious loss of accuracy and is only used when termination cannot be guaranteed otherwise.

Definition 4 (regular splitting, r-split). *Let $qs = q_1 \cdots q_n$ be a query, $n \geq 1$. We say that the tuple of integers $\langle i_1, \ldots, i_n\rangle$ is a regular splitting for qs if*

- $1 \leq i_1 < i_2 < \ldots < i_n \leq |qs|$ *and*
- *the sequence $qs|_{in}$ contains at most one call to a non-regular predicate for all $in \in \{1..i_1, (i_1+1)..i_2, (i_2+1)..i_3, \ldots, (i_n+1)..|qs|\}$.*

In general, there might be more than one regular splitting for a given query. Here, we let r-split(qs) *return any of them.*[2]

Let us consider again the programs of Example 1. In this case, it is easy to check that r-split(`append`$(\mathtt{L}, [\mathtt{X}], \mathtt{LX}), \mathtt{last}(\mathtt{Last}, \mathtt{LX})) = \{2\}$ (i.e., no splitting is done), but r-split(`flip`$(\mathtt{L}, \mathtt{FL}), \mathtt{flip}(\mathtt{R}, \mathtt{FR})) = \{1, 2\}$ (since `flip/2` is non-regular).

The transition rules of the local level are shown in Fig. 1. Let us briefly explain the rules of the calculus:

variant: If a query is a variant of another query in the local stack (i.e., it is *closed* [13]), we stop the derivation by introducing the special symbol $\diamond$. In contrast to previous approaches, we do not consider a query closed when it is an *instance* of a previously unfolded query. This is also justified by our interest in avoiding the loss of groundness and sharing information as much as possible; for instance, if one accepts that a query $q(X, X)$ is closed w.r.t. $q(A, B)$, then all previous assumptions about the independence of variables A and B in $q(A, B)$ would be wrong.

independent splitting: If a query has a prefix that contains two subsequences which are independent (i.e., do not share variables according to the associated call patterns), this rule non-deterministically continues with the evaluation of any of the subsequences (by passing the control to the global level).

unfold: This rule selects the leftmost unfoldable atom of the current query (i.e., one that does not embed any ancestor and that, moreover, is terminating for the considered call pattern) and then performs an unfolding step. Here, $\langle qs|_i, ps|_i\rangle \rightsquigarrow_\sigma \langle qs', ps'\rangle$ denotes a slight extension of SLD resolution so that $qs|_i \rightsquigarrow_\sigma qs'$ is a standard SLD-resolution step and ps' are the new call patterns for qs'; the formal definition is not difficult but we skip the details for simplicity. In the derived state, we denote by $as[\overline{as|_i \cup \{qs|_i\}}]_i$ the result of replacing the i-th element of the sequence of ancestors as by $|qs'|$ repetitions of the set $as|_i \cup \{qs|_i\}$ (i.e., all atoms introduced in the same SLD-step share the same set of ancestors). Note that this rule is also non-deterministic when there are several matching clauses for the selected atom.

[2] In the implemented partial evaluator, we just traverse qs from left to right and start a new subsequence every time a call to a non-regular predicate is found.

global level

(restart) $$\frac{\nexists qs' \in gs.\ qs_i \trianglerighteq qs',\ i \in \{1,\ldots,n\}}{\langle\langle qs_1 \cdots qs_n,\ ps_1 \cdots ps_n,\ gs\rangle\rangle \rightarrow \langle qs_i, [\,], ps_i^r, [\,], \{qs_i\} \cup gs\rangle}$$

(stop) $$\frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ i \in \{1,\ldots,n\}}{\langle\langle qs_1 \cdots qs_n,\ ps_1 \cdots ps_n,\ gs\rangle\rangle \rightarrow_{qs_i} \langle\langle\ \rangle\rangle}$$

local level

(variant) $$\frac{\exists qs' \in ls.\ qs \approx qs'}{\langle qs, as, ps, ls, gs\rangle \overset{v}{\Rightarrow} \langle \diamond, as, ps, ls, gs\rangle}$$

(independent splitting)
$$\frac{\mathsf{i\text{-}split}(qs, ps) = \langle i, j\rangle}{\langle qs, as, ps, ls, gs\rangle \overset{i}{\Rightarrow} \langle\langle qs|_{1..i} qs|_{i+1..j} qs|_{j+1..|qs|},\ ps|_{1..i} ps|_{i+1..j} ps|_{j+1..|ps|},\ gs\rangle\rangle}$$

(unfold) $$\frac{\mathsf{unfold}(qs, as) = i\ \wedge\ \langle qs|_i, ps|_i\rangle \rightsquigarrow_\sigma \langle qs', ps'\rangle}{\langle qs, as, ps, ls, gs\rangle \overset{u}{\Rightarrow}_\sigma \langle qs[qs']_i, as[\overline{as|_i \cup \{qs|_i\}}]_i, ps[ps']_i, \{qs\} \cup ls, gs\rangle}$$

(regular splitting)
$$\frac{\mathsf{r\text{-}split}(qs) = \langle i_1, \ldots, i_n\rangle\ \wedge\ in_0 = 1..i_1,\ in_1 = (i_1+1)..i_2, \ldots,\ in_n = (i_n+1)..|qs|}{\langle qs, as, ps, ls, gs\rangle \overset{r}{\Rightarrow} \langle\langle qs|_{in_0} qs|_{in_1} \cdots qs|_{in_n},\ ps|_{in_0} ps|_{in_1} \cdots ps|_{in_n},\ gs\rangle\rangle}$$

Fig. 1. Partial evaluation semantics

regular splitting: When none of the previous rules apply, this means that termination of partial evaluation cannot be ensured. Here, we are forced to split up the query even if it might involve a serious loss of accuracy. This rule proceeds similarly to the rule for independent splitting and pass the control to the global level.

We assume that the transition rules are tried in the textual order so that, whenever a rule is applicable, the remaining rules (if any) are discarded (this could easily be ensured by adding more premises but we prefer to keep the current formulation for clarity).

4 Post-Processing Stage

Once the partial evaluation stage terminates, we produce renamed, residual rules associated to the transitions of the partial evaluation semantics as follows:

– For every unfolding step $\langle qs, as, ps, ls, gs\rangle \overset{u}{\Rightarrow}_\sigma \langle qs', as', ps', ls', gs'\rangle$, we produce a binary clause of the form $ren(qs)\sigma \leftarrow ren(qs')$, where $ren(qs)$ and $ren(qs')$ are *renamings* (atoms with fresh predicate symbols) of queries qs and qs', respectively. Actually, we observed that our scheme returns binary residual programs often (in particular, when the source programs are B-stratifiable [7]).

– For every branching performed with the rules for independent or regular splitting $\langle qs, as, ps, ls, gs\rangle \stackrel{s}{\Rightarrow} \langle\langle qs_1 \cdots qs_n,\ \_,\ \_\rangle\rangle$, with $s \in \{i, r\}$, we produce a residual clause of the form $ren(qs) \leftarrow ren(qs_1), \ldots, ren(qs_n)$.
– Finally, for every global transition with rule stop $\langle\langle qs_1 \cdots qs_n, \_, \_]\rangle\rangle \rightarrow_{qs_i} \langle\langle\ \rangle\rangle$ we produce a residual clause of the form $ren(qs_i) \leftarrow qs_i$. Here, in contrast to previous approaches, we prefer to keep qs_i and add the (required) clauses of the original program to the residual one (instead of using generalization at partial evaluation time). This case does not occur so often in practice (see the next section) and allows us to correctly propagate some run-time properties (like groundness and sharing information) that the use of generalization might destroy.

We do not present the details of the renaming function here since it is a standard renaming as introduced in, e.g., [5].

Besides the extraction of renamed, residual clauses, we also apply a simple post-unfolding transformation. Basically, we unfold all *intermediate* predicates, i.e., predicates that are only called from one program point. This is very effective for reducing the size of the residual program and is rather simple to implement.

5 Concluding Remarks and Future Work

The correctness of our approach would not be difficult to prove. In general, it can be seen as an instance of the CPD framework [5]. The main difference comes from the fact that our "one-step" unfoldings do not generally fulfill the weakly fair condition of [5] (which is required for ensuring correctness w.r.t. finite failures). This is solved in our approach by requiring non-leftmost unfolding to be only applicable over terminating calls.

A prototype implementation of the partial evaluator described so far has been undertaken. It consists of approx. 1000 lines of SWI Prolog code (including the call and success pattern analysis, comments, etc). The only missing component is the left-termination analysis (so the preservation of finite failures is not yet ensured). A web interface to our tool is publicly available at `http://kaz.dsic.upv.es/lite.html`. We have tested it by running the benchmarks of the DPPD library [10] that do not contain built-in's nor negation. The following table summarizes the results; for every benchmark, we show the number of inferences (using the `time/1` utility of SWI Prolog) of a run-time query in the original and partially evaluated programs:

benchmark	*advisor*	*applast*	*depth*	*doubleapp*	*ex_depth*	*flip*	*matchapp*	*regexp.r1*
original	4	58	24	50	24	34	374	73
residual	0	29	1	34	15	47	23	10

benchmark	*regexp.r2*	*regexp.r3*	*relative*	*rev_acc_type*	*rotateprune*	*transpose*
original	28	41	96	35	32	58
residual	8	12	3	34	45	0

Observe that only 2 (out of 14) benchmarks show a slight slowdown. In the remaining ones, the speedups are generally good (and sometimes impressive, when the residual program is reduced to a collection of facts, e.g., the case of *transpose*, or some non-trivial optimization is achieved, the case of *matchapp*). We

also note that, despite the fact that our approach does not include generalization, only 3 benchmarks required the clauses of the original program because of the application of the global rule `stop` (the case of benchmarks *ex_depth*, *flip*, and *rotateprune*).

As for future work, we plan to extend the current approach in order to deal with built-in's and negation, so that a more extensive testing can be done. On the other hand, we will explore the addition of (run-time) variable sharing information. Besides improving the accuracy of splitting, the combination of (run-time) sharing and groundness information can be very useful at partial evaluation time in order to produce residual programs where some sequential conjunctions are automatically replaced by *concurrent* conjunctions. Some preliminary experiments in this direction (using the `concurrent/3` predicate of SWI Prolog) have shown promising results.

References

1. A. Ben-Amram and M. Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C.R. Ramakrishnan and Jakob Rehof, editors, *Proc. of TACAS'07*, pages 46–55. Springer LNCS 5028, 2008.
2. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs. In *Proc. 1991 Int'l Symp. on Logic Programming*, pages 117–131, 1991.
3. N.H. Christensen and R. Glück. Offline Partial Evaluation Can Be as Accurate as Online Partial Evaluation. *ACM Transactions on Programming Languages and Systems*, 26(1):191–220, 2004.
4. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
5. 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.
6. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
7. J. Hruza and P. Stepánek. Speedup of logic programs by binarization and partial deduction. *TPLP*, 4(3):355–380, 2004.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
9. M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.
10. M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks, 2007. Available at URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.
11. M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and Their Web Interfaces. In *Proc. of PEPM'06*, pages 88–94. IBM Press, 2006.
12. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of LOPSTR 2008*, Springer LNCS 5438, pages 119-134, 2009.
13. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
14. Z. Somogyi. A System of Precise Modes for Logic Programs. In E.Y. Shapiro, editor, *Proc. of ICLP'86*, pages 769–787. The MIT Press, Cambridge, MA, 1986.

Non-termination Analysis of Logic Programs using Types

Dean Voets* and Danny De Schreye

Department of Computer Science, K.U.Leuven, Belgium
Celestijnenlaan 200A, 3001 Heverlee
{Dean.Voets, Danny.DeSchreye }@cs.kuleuven.be

Abstract. In recent years techniques and systems have been developed to prove non-termination of logic programs for certain classes of queries. In previous work, we developed such a system based on mode-information and a form of loop checking performed at compile time.
In the current paper we improve this technique by integrating type information in the analysis and by applying non-failure analysis and program specialization. It turns out that there are several classes of programs for which existing non-termination analyzers fail and for which our extended technique succeeds in proving non-termination.
Keywords: non-termination analysis, types, non-failure analysis, program specialization

1 Introduction

In the past five years, techniques have been developed to analyze the non-termination – as opposed to termination – of logic programs. The main motivation for this work is to provide precision results for termination analyzers.

Given some program P, a termination analyzer can be considered *precise* on P, if the class of programs it proves terminating for P is exactly the class of the terminating ones. Of course due to the undecidability of the halting problem, an analyzer can not be precise on all programs. Non-termination analysis can be used to show the precision of an analyzer. Given a program P, a termination analyzer is shown to be precise on P, if the class of queries for which it proves termination is the complement of the class of queries proved non-terminating by a non-termination analyzer.

Techniques for non-termination analysis should not be confused with those for loop checking. The former are compile-time techniques, while the latter are performed at run-time.

The first and most well-known non-termination analyzer is *NTI* ([6]). Recently, we developed a slightly more precise analyzer, *P2P*, presented in [9].

Both non-termination analyzers contain two phases in their analysis. In a first phase, they compute a suitable, finite approximation of all the computations for the considered class of queries. This phase is similar to what is done in

* Supported by the Fund for Scientific Research - FWO-project G0561-08

many other program analysis techniques, such as abstract interpretation, partial deduction or program specialization, and even a number of approaches to termination analysis. In a second phase, the non-termination analyzers analyze these finite approximations of the computations, to detect loops. They prove that these loops correspond to infinite derivations in concrete computations of the program for the considered queries.

In *NTI*, the finite approximation of the computations are binary unfoldings of the given program. Detecting loops is based on a special more general than relation, called Δ-*more general*. If the body of a binary clause is Δ-more general than the head, a loop is detected and non-termination is proved.

In [9], the finite approximations of the computations is the *moded SLD-tree*. This concept has been introduced in [8], in the context of termination and non-termination prediction. A moded SLD-tree is a symbolic approximation of concrete derivations, based on modes. It's finiteness is ensured by applying a *complete* loop check on the symbolic derivations, called *LP-check*. Due to completeness of this loop check, finiteness of the resulting tree is ensured. For its second phase, the analysis in [9] introduces a *sound* loop check on the moded SLD-tree. Each observed loop corresponds to a non-terminating computation for the concrete derivations.

We illustrate both techniques with an example.

Example 1. `a(f(X),s(Y)):- a(X,s(s(Y))).`

NTI proves non-termination by showing that the body is Δ-more general than the head. The first argument satisfies this more general relation because it is replaced by a more general term. The second argument requires the *filter* `instance of s(Y)`, to satisfy the Δ-more general relation. *NTI* concludes that this program is non-terminating for each query `a(I,J)` with `I` more general than `f(X)` and `J` an instance of `s(Y)`.

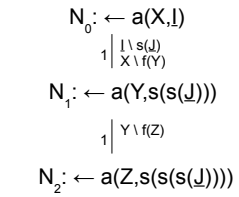


Fig. 1. Moded SLD-tree of Example 1

P2P constructs the *moded SLD-tree* of Figure 1. In this tree, I and J are *input modes*, denoting unknown ground terms. *P2P* proves that the path from N_1 to N_2 can be repeated infinitely often for any query `a(X,s(J))` with `X` a variable and J a variable-free term. □

In [9], the non-termination analyzer *P2P* was applied to all non-terminating programs of the benchmark of the annual termination competition[1]. It turned

[1] Available at http://www.lri.fr/~marche/termination-competition/2007/

out that *P2P* was able to prove non-termination for all non-terminating programs in the benchmark.

Although we experienced this as a successful experiment for the technique and our results were an improvement of the success rate of *NTI* on the same benchmark, the experiment mostly shows that the benchmark does not offer sufficient challenges for non-termination analysis. As a result, our work since then has focussed on two new directions. One is to identify classes of programs for which current non-termination analyzers fail. A second is to investigate whether the inclusion of type-information, in addition to modes, may improve the power of our analyzer.

Considering the first of these questions, a limitation of both *NTI* and *P2P* is that they only detect non-terminating derivations if, within these derivations, some fixed sequence of clauses can be applied repeatedly. The following example violates this restriction.

Example 2. The program, *longer*, loops for any query `longer(L)`, with `L` a non-empty list of zeros. The predicate *zeros/1* checks if the list contains only zeros. At the recursive call, a zero is added to the list in the previous call.

```
longer([0|L]):-
        zeros(L),                    zeros([]).
        longer([0,0|L]).             zeros([0|L]):- zeros(L).
```

The list in the recursive call is longer than the original one and thus, the number of applications of the recursive clause for *zeros/1* increases in each recursion. Therefore, no fixed sequence of clauses can be repeated infinitely and both *NTI* and *P2P* fail to prove non-termination of this example. □

In this paper, we overcome this limitation by using non-failure information. *Non-failure analysis* ([2]) detects classes of goals that can be guaranteed not to fail, given mode and type information. Its applications include inferring minimal computational costs, guiding transformations and debugging. To use the information provided by non-failure analysis in the non-termination analysis of [9], type information must be added to the symbolic derivation tree. We add this information using regular types ([10]).

There are many variants of this class of programs. Instead of having an increasing number of applications of a same clause, we could have a predicate that always succeeds but in which alternative clauses are used for obtaining success. Again, *NTI* and *P2P* will fail to prove non-termination, because the sequence of clause applications is not repeated. Non-failure information though allows to abstract away from the details on how success was reached for these computations.

Another limitation of both *NTI* and *P2P* is related to aliased variables. We illustrate this with an example from [7].

Example 3. `append([],L,L).` `append([H|T],L,[H|R]):- append(T,L,R).`

The query `append(X,X,X)` succeeds once with a computed answer substitution `X/[]`. The program loops after backtracking.

NTI needs a filter for the second argument to prove non-termination of this example. However, these filters are only allowed on argument positions that don't share variables with terms on other positions. Because all argument positions contain the same variable, *NTI* fails to prove non-termination of this example.

Similarly, *P2P* needs an *input-generalization* (see Section 4.2) to prove non-termination for this example. Due to the shared variables, this input-generalization is not allowed for this query. □

Non-termination of the last example can be proven by specializing *append* for the considered query. *Program specialization* ([3]) transforms a logic program and class of queries to a specialized, more efficient program. In [5], the authors showed that program specialization improves the results of termination analyzers. In this paper, we show that this is also the case for non-termination analyzers.

Example 4. Specializing *append* for the query `append(X,X,X)`, using the *ECCE* specializer of [3], produces the following program:

```
append(X,X,X):- app1(X).          app1([]).
app2([H|T]):- app2(T).            app1([A,B|C]):- app2(C).
```

Both *NTI* and *P2P* prove non-termination of this specialized program. □

In this paper, we extend the technique of [9] with type information. Then, we use non-failure analysis and specialization to deal with the classes of programs illustrated by the examples above. In addition to these two classes of programs, there are combinations of them that yield a fairly large class of new programs that we can prove non-terminating using non-failure analysis combined with program specialization.

The paper is structured as follows. In the next section we introduce some preliminaries. In Section 3, we add type information to the moded SLD-tree defined in [8] and we introduce a special transition to treat non-failing goals. In Section 4, we adapt our non-termination condition of [9] for these extended symbolic derivation trees and we show that program specialization can be used to improve on the results of our non-termination condition. Finally, Section 5 concludes the paper. Due to space restrictions, all proofs have been omitted. A version of the paper containing all proofs can be found online[2].

2 Preliminaries

2.1 Logic Programming

We assume the reader is familiar with standard terminology of logic programs, in particular with SLD-resolution as described in [4]. Variables are denoted by character strings beginning with a capital character. Predicates, functions and constant symbols are denoted by character strings beginning with a lower case character. We denote the set of terms constructible from a program P, by

[2] http://www.cs.kuleuven.be/~dean/

$Term_P$. Two atoms are called *variants* if they are equal up to variable renaming. An atom A is *more general* than an atom B, if there exists a substitution θ, such that $A\theta = B$.

We restrict our attention to definite logic programs. A logic program P is a finite set of clauses of the form $H \leftarrow A_1, ..., A_n$, where H and each A_i are atoms. A goal G_i is a headless clause $\leftarrow A_1, ..., A_n$. A query, Q, is a conjunction of atoms $A_1, \ldots, A_n$. Without loss of generality, we assume that Q consists only of one atom.

Let P be a logic program and G_0 a goal. G_0 is evaluated by building a *generalized SLD-tree* GT_{G_0} as defined in [8], in which each node is represented by $N_i : G_i$, where N_i is the name of the node and G_i is a goal attached to the node. Let A_i and A_j be the selected atoms at two nodes N_i and N_j, respectively. A_i is an *ancestor* of A_j, denoted $A_i \prec_{anc} A_j$, if the proof of A_i goes through the proof of A_j. Throughout the paper, we choose to use the best-known *depth-first, left-most* control strategy, as is used in Prolog, to select goals and atoms. So by the *selected atom* in each node $N_i :\leftarrow A_1, ..., A_n$, we refer to the left-most atom A_1. For any node $N_i : G_i$, we use A_i^1 to refer to the selected atom in G_i.

A derivation step is denoted by $N_i : G_i \Rightarrow_C N_{i+1} : G_{i+1}$, meaning that applying a clause C to G_i produces $N_{i+1} : G_{i+1}$. Any path of such derivation steps starting at the root node $N_0 : G_0$ of GT_{G_0} is called a *generalized SLD-derivation*.

2.2 Types

In this paper, we extend the moded SLD-tree defined in [8] by adding a special operation for nodes with a non-failing selected atom. To use non-failure information, type information of the partially instantiated goals must be available. Since logic programs are untyped, this type information will be inferred. Many techniques and tools exist to infer type definitions for a given logic program, for example [1]. We will describe types using *regular types* as defined in [10]. The set of type symbols is denoted by Σ_τ.

Definition 1. *Let P be a logic program. A **type rule** for a type symbol $T \in \Sigma_\tau$ is of the form $T \rightarrow c_1; \ldots; c_i; f_{i+1}(\bar{\tau}_{i+1}); \ldots; f_k(\bar{\tau}_k)$, $(k \geq 1)$, where $c_1, \ldots, c_i$ are constants, $f_{i+1}, \ldots, f_k$ are distinct function symbols associated with T and $\bar{\tau}_j$ $(i+1 \leq j \leq k)$ are tuples of corresponding type symbols of Σ_τ. A **type definition** $\mathcal{T}$ is a finite set of type rules, where no two rules contain the same type symbol on the left hand side, and for each type symbol occurring in the type definition, there is a type rule defining it.* □

A *predicate signature* declares one type symbol for each argument position of a given predicate. A *well-typing* $\langle \mathcal{T}, \mathcal{S} \rangle$ of a program P, is a pair consisting of a type definition $\mathcal{T}$ and a set $\mathcal{S}$ containing one predicate signature for each predicate of P, such that the types of the actual parameters passed to a predicate are an instance of the predicate's signature. For this paper, we use *PolyTypes* ([1]) to infer signatures and type definitions. In [1], it has been proven that these inferred signatures and type definitions are a well-typing for the given program.

Types allow to give a more precise description of the possible values during evaluation at different argument positions. The set of terms constructible from a certain type definition, is called the *denotation of the type*. We represent the denotation of type T by $Den(T)$.

Given a type definition $\mathcal{T}$, for every type T defined by $\mathcal{T}$, we introduce an infinite set of fresh variables Var_T. For any two types $T_1 \neq T_2$, we impose that $Var_{T_1} \cap Var_{T_2} = \emptyset$.

Definition 2. *Let $T_1, \ldots, T_n$ be types defined by a type definition $\mathcal{T}$ defining type symbols $\bar{\tau}$. The* ***denotation*** *$Den(T_i)$ of $T_i (1 \leq i \leq n)$, defined by $T_i \rightarrow c_1; \ldots; c_j; f_{j+1}(\bar{\tau}_{j+1}); \ldots; f_k(\bar{\tau}_k)$, is recursively defined as:*

- *every variable in Var_{T_i} is an element of $Den(T_i)$*
- *every constant $c_p, 1 \leq p \leq j$, is an element of $Den(T_i)$*
- *for every type term $f_p(\tau_1, \ldots \tau_l), j < p \leq k$,: if $t_1 \in Den(\tau_1), \ldots, t_l \in Den(\tau_l)$, then $f_i(t_1, \ldots, t_l) \in Den(T_i)$* □

Example 5. For the program *longer* of Example 2, the following types and signatures are inferred by *PolyTypes*:

$T_{lz} \rightarrow [\,]; [T_0 \mid T_{lz}]$	$longer(T_{lz})$
$T_0 \rightarrow 0$	$zeros(T_{lz})$

$Den(T_0)$ is the set containing 0 and all variables of Var_{T_0}. $Den(T_{lz})$ contains: $[\,], Y, [0,0], [0,X,0], [X|Y], \ldots$, with $X \in Var_{T_0}$ and $Y \in Var_{T_{lz}}$. □

2.3 Non-failure analysis and program specialization

Given type and mode information, *non-failure analysis* detects goals that can be guaranteed not to fail, i.e. they either succeed or go in an infinite loop. We use the non-failure analysis technique of [2] in this paper.

Example 6. For Example 2, non-failure analysis proves that *zeros/1* is non-failing if its argument is an input mode of type T_{lz}. It cannot show that *longer/1* with an argument of type T_{lz} is non-failing, because $longer([\,])$ fails. □

Program specialization aims at transforming a given logic program into an equivalent but more efficient program for a certain query. This query can be partially instantiated, yielding a specialized program for a class of queries. Program specialization has received a lot of attention in the community, see for example [3]. In this paper, we use the specialization tool *ECCE*, [3], to generate specialized programs.

3 Moded-Typed SLD-trees and the *NFG* transition

3.1 Moded-typed SLD-trees and loop checking

As stated in the introduction, we want to prove non-termination of partially instantiated queries, given mode and type information of the variables in the

query. In such a partially instantiated query, variables representing unknown terms are labeled *input modes*. To every input mode, a type is assigned. An atom Q is a *moded-typed* atom if some variables of Q are input modes, otherwise, it is a *concrete* atom. The terms represented by input modes in an atom Q are restricted to terms of their respective type. Note that the user does not have to declare the types associated to input modes, because the inferred well-typing declares a unique type for every subterm of atoms defined by the program.

Because a variable associated with an input mode represents an unknown term, the effect of an input mode can be approximated by treating it as a special variable I, such that I may be substituted by a constant or compound term of the correct type. In the remainder of the paper, we denote a special variable, an *input variable*, by underlining the variables name.

Definition 3. *Let P be a program, $Q = p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)$ a moded-typed atom and $\langle \mathcal{T}, \mathcal{S} \rangle$ a well-typing for P. The **moded-typed SLD-tree** of P for Q, is a pair $(GT_{G_0}, \langle \mathcal{T}, \mathcal{S} \rangle)$, where GT_{G_0} is the generalized SLD-tree for $P \cup \{\leftarrow p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)\}$, with each $\underline{I_i}$ being a special variable not occurring in any T_j. The special variables $\underline{I_1}, ..., \underline{I_m}$ are called **input variables**.* □

As mentioned, an input variable $\underline{I}$ may be substituted by a term $f(t_1, \ldots, t_n)$. If $\underline{I}$ is substituted by $f(t_1, \ldots, t_n)$, all variables in $t_1, \ldots, t_n$ also become input variables. In particular, when unifying $\underline{I}$ with a normal variable X, X becomes an input variable. We refer to Figure 2 for an illustration of (part of) a moded-typed SLD-tree. Some branches are missing, because the figure also illustrates a loop check, that we introduce below.

A moded-typed atom A represents a set of concrete atoms, called the *denotation* of A. This is the set of atoms obtained by replacing the input modes by arbitrary terms of their respective type.

We introduce the following notation. Let $s_1, \ldots, s_n$ be subterms of A, such that no s_i is a subterm of s_j. Let $t_1, \ldots, t_n$ be terms. $A(t_1 \rightarrow s_1, \ldots, t_n \rightarrow s_n)$ denotes the atom obtained by replacing each occurrence of terms $s_1, \ldots, s_n$ by corresponding terms $t_1, \ldots, t_n$.

Definition 4. *Let A be a moded-typed atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables with types $T_1, \ldots, T_n$, respectively. The **denotation** of A, $Den(A)$, is*

$$\{A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n}) \mid t_1 \in Den(T_1), \ldots, t_n \in Den(T_n)\}$$ □

Note that non-termination of an atom implies non-termination of all more general atoms. Therefore, we consider a moded-typed atom A to represent all atoms of the denotation of A, as well as all more general atoms. We call this set the *extended denotation* of A.

Definition 5. *Let A be a moded-typed atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables with types $T_1, \ldots, T_n$, respectively. The **extended denotation** of A, $Ext(A)$, is*

$$\{I \mid I' \in Den(A), I \text{ is more general than } I'\}$$ □

Note that this extended denotation includes the atoms of the denotation and that this concept can be adapted to moded-typed goals in a straightforward way.

Moded-typed SLD-trees represent all derivations for a certain class of queries. For most programs and classes of queries, this tree is infinite. In order to obtain a finite tree, we can use a complete loop check, to cut all infinite derivations from the tree.

In [8], it is proven that every infinite generalized SLD-derivation contains an infinite chain of *loop goals*. A derivation is cut by the complete loop check *LP-check* ([8]), if a prefix of such a chain is encountered. *LP-check* uses a *repetition number* defining how long the chain of loop goals can become before it is cut. For more information about this loop check, we refer to [8].

Example 7. Figure 2 shows the moded-typed SLD-tree for *longer*, using *LP-check* with repetition number 2.

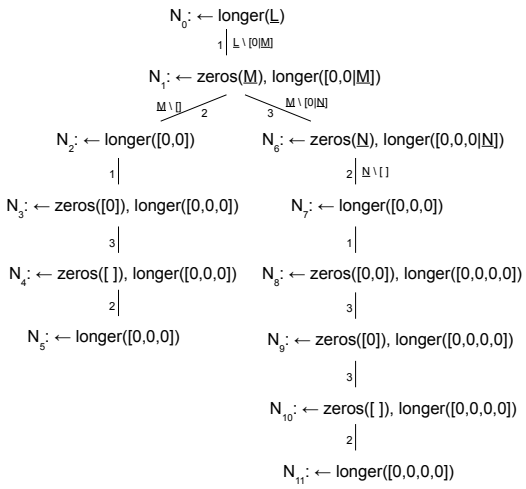


Fig. 2. Moded-typed SLD-tree for the *longer* program, using *LP-check* with repetition number 2

At node N_5, *LP-check* cuts clause 1 because node N_5 is a loop goal of node N_2. Likewise, *LP-check* cuts clause 3 at node N_6 and clause 1 at node N_{11}. □

3.2 The *NFG* transition

As illustrated in Example 2, one of the main limitations of current non-termination analysis techniques is that non-termination can only be proven for programs and queries repeating a fixed sequence of clauses. To overcome this limitation, we extend the moded-typed SLD-tree of Definition 3 by adding a special transition *NFG*, to treat non-failing selected atoms. This transition allows to abstract from the sequence of clauses needed to solve the non-failing atom. Note that during the evaluation of a non-failing atom, normal variables may be bound to terms of their respective type. Therefore, we approximate the application of this unknown

sequence of clauses by substituting all normal variables in the selected atom by fresh input variables of the correct type. In the next subsection, the correlation between these resulting moded-typed SLD-trees and concrete SLD-trees is given.

Definition 6. *Let $N_i :\leftarrow A_1, A_2, \ldots, A_n$ be a node in a moded-typed SLD-tree, with A_1 a non-failing atom. Let $V_1, \ldots, V_m$ be all normal variables of A_1, corresponding to types $T_1, \ldots, T_m$, respectively. Let $\underline{V_1}, \ldots, \underline{V_m}$ be new input variables of types $T_1, \ldots, T_m$, respectively. Then, an NFG transition, $N_i :\leftarrow A_1, \ldots, A_n \Rightarrow_{NFG} N_{i+1} :\leftarrow A_2\theta, \ldots, A_n\theta$, can be applied to N_i, with substitution $\theta = \{V_1 \setminus \underline{V_1}, \ldots, V_m \setminus \underline{V_m}\}$.* □

Note that we can also allow an *NFG* transition to be applied to a goal containing only one atom, resulting in a refutation.

Definition 7. *Let P be a logic program, Q a moded-typed atom and $\langle \mathcal{T}, \mathcal{S} \rangle$ a well-typing for P. The **moded-typed SLD-tree with NFG** is the pair $(GT_{(NFG,G_0)}, \langle \mathcal{T}, \mathcal{S} \rangle)$, where $GT_{(NFG,G_0)}$ is obtained from the generalized SLD-tree GT_{G_0} for $P \cup \leftarrow Q$, by, at each node: $N_i :\leftarrow A_1, A_2, \ldots, A_n$, with A_1 a non-failing atom, additionally applying the NFG transition.* □

Without proof, we state that *LP-check* is also a complete loop check for moded-typed SLD-trees with *NFG*. From here on, when we refer to a moded-typed SLD-tree, we mean the finite part of a moded-typed SLD-tree with *NFG*, obtained by using *LP-check* with some repetition number, r.

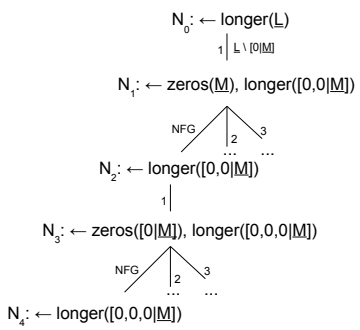


Fig. 3. Moded-typed SLD-tree with *NFG* of the *longer* program

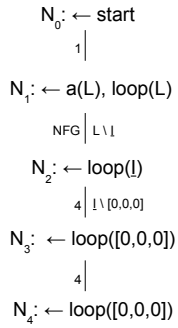


Fig. 4. Moded-typed SLD-tree of Example 10

Example 8. Figure 3 shows the moded-typed SLD-tree of Example 2 for the query *longer*($\underline{L}$) using 3 as a repetition number. The selected atom at node N_1, *zeros*($\underline{M}$), is non-failing and can be solved using an *NFG* transition. Since there are no normal variables in the selected atom, there are no substitutions for this transition. At node N_3, the *NFG* transition is applied as well.

At node N_4, *LP-check* cuts the derivation because of the chain of loop goals $N_0 \Rightarrow_1 \ldots N_2 \Rightarrow_1 \ldots N_4 \Rightarrow_1$. □

3.3 Correlation with concrete queries

The correlation between derivations in moded-typed SLD-trees without *NFG* transitions and concrete SLD-derivations is rather simple. For every derivation in such a moded-typed SLD-tree from $N_0 : G_0$ to $N_i : G_i$, there exists a non-empty subset of the concrete goals of the extended denotation of G_0 on which the same sequence of clauses can be applied. This subset of goals can be obtained from G_0 by applying the composition of all substitutions on input variables from N_0 to N_i.

Example 9. In the derivation from N_0 to N_8 in Figure 2, there are three substitutions on input variables: $\underline{L} \setminus [0|\underline{M}]$, $\underline{M} \setminus [0|\underline{N}]$ and $\underline{N} \setminus []$. Applying these substitutions to the query $longer(\underline{L})$ gives the concrete query $longer([0,0])$. □

For typed SLD-trees with *NFG* transitions, there is no such clear correspondence. Solving a non-failing atom corresponds to a potentially infinite sequence of clause applications. Thus, for an *NFG* transition $N_i : G_i \rightarrow_{NFG} N_j : G_j$, it is possible that no concrete state corresponding to G_j ever occurs in a concrete derivation. However, this will not cause any problems for our analysis. If our analysis detects and reports a non-terminating computation for a branch in the moded-typed SLD-tree which has an *NFG* transition, then a corresponding concrete atom either finitely succeeds or non-terminates. In both cases, reporting non-termination is correct.

There is a second problem with the correspondence between *NFG* transitions and concrete derivations. Input variables introduced by *NFG* give an overestimation of the possible values after evaluating the non-failing selected atom in a concrete derivation. Therefore, substitutions on such input variables further down the tree might be impossible and thus, cannot be allowed. We illustrate this with an example.

Example 10.
```
start:- a(L), loop(L).
loop([0,0,0]):- loop([0,0,0]).
```
```
a([0]).
a([0,0]).
```

T_{lz} is a correct type definition for all arguments of atoms. Non-failure analysis ([2]) shows that `a(L)` is non-failing if `L` is a free variable.

Figure 4 shows a part of the moded-typed SLD-tree using repetition number 2 for the query `start`. At node N_1, the non-failing atom `a(L)` is solved by replacing the variable `L` by a new input variable `I`. The path from N_3 to N_4 is a loop. However, this loop cannot be reached because `I` is substituted by `[0,0,0]`, but the program only allows it to be `[0]` or `[0,0]`. □

We introduce the following proposition showing the correlation between moded-typed SLD-trees and concrete SLD-trees.

Proposition 1. *Let T be a moded-typed SLD-tree with NFG for a goal G_0 and N_i a node of T. Let θ be the composition of all substitutions on input variables from N_0 to N_i.*

If the derivation from N_0 to N_i does not contain a substitution on input variables introduced by NFG transitions, then all goals in $Ext(G_0\theta)$ can be evaluated to goals in $Ext(G_i)$ or loop w.r.t. an NFG transition. □

4 Typed non-termination analysis with non-failing goals

In this section, we reformulate our non-termination conditions of [9] for moded-typed SLD-trees with *NFG*. We prove that these conditions imply non-termination. We then show that program specialization can be used to further extend the applicability of these conditions.

4.1 Non-termination of inclusion loops

To prove non-termination, we prove that a path between two nodes $N_b :\leftarrow A_b^1, \ldots$ and $N_e :\leftarrow A_e^1, \ldots$ in a moded-typed SLD-derivation can be repeated infinitely often. To find such a path, we check three properties. Because the rules in the path must be applicable independent of the values of the input variables, no substitutions on the input variables may occur in the path from N_b to N_e. Because this path should be a loop, A_b^1 must be an ancestor of A_e^1. Finally, because the goals corresponding to N_e must be able to repeat the loop, $Ext(A_e^1)$ must be a subset of $Ext(A_b^1)$. We can show that these three conditions imply non-termination.

Definition 8. *In a moded-typed SLD-derivation with NFG D, nodes $N_i : G_i$ and $N_j : G_j$ are an **inclusion loop**, $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$, if:*

- *No substitutions on input variables occur in the path from N_i to N_j.*
- *$A_i^1 \prec_{anc} A_j^1$.*
- *$Ext(A_j^1) \subset Ext(A_i^1)$* □

An inclusion loop $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$ corresponds to an infinite loop for every goal of the extended denotation of G_i.

Theorem 1 (Sufficiency of the inclusion loop). *Let $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$ be an inclusion loop in a moded-typed SLD-derivation with NFG D of a program P and a moded-typed query Q, then, every goal of the extended denotation of G_i is non-terminating w.r.t. P.* □

Due to Proposition 1, an inclusion loop $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$ proves non-termination for a subset of concrete goals of the extended denotation of the top level goal, if no substitutions on input variables introduced by *NFG* transitions occur in the path from N_0 to N_i.

Example 11. The moded-typed SLD-tree of Figure 3 can be used to prove non-termination of the *longer* program. The path from N_2 to N_4 satisfies the conditions of the inclusion loop. There are no substitutions on input variables between these nodes, the ancestor relation holds and $Ext(longer([0, 0, 0 \mid \underline{M}]))$ is a subset of $Ext(longer([0, 0 \mid \underline{M}]))$.

Since there are no input variables added by *NFG* transitions, Proposition 1 proves that all atoms in $Ext(longer([0 \mid \underline{M}]))$, with $\underline{M}$ of type T_{lz}, are non-terminating. □

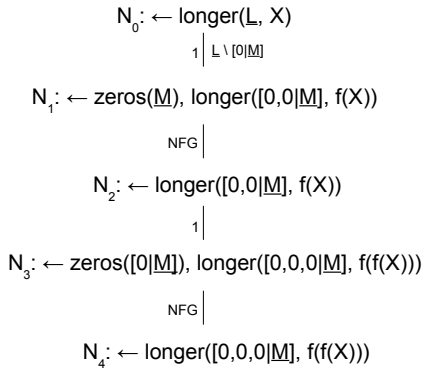


Fig. 5. Moded-typed SLD-tree of Example 12

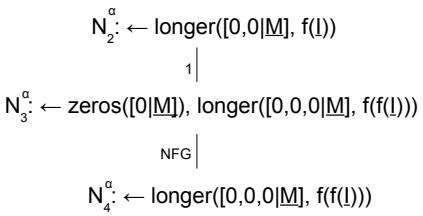


Fig. 6. Input-generalized derivation

Because extended denotations are usually infinite sets, the third condition of Definition 8, $Ext(A_j^1) \subset Ext(A_i^1)$, is hard to verify. In [9], we introduced a syntactic condition, based on a unifiability test between atoms, to verify whether the inclusion holds. This syntactic condition, formulated in Proposition 1 of [9], is still applicable in our current setting and can be used to automatically check Definition 8.

4.2 Input-generalizations

In [9], we have also shown that the applicability of the inclusion loop can be extended using a notion called *input-generalizations*.

Example 12. We show the need for input-generalizations with an adapted version of the program of Example 2.

```
longer([0|L], X):-
        zeros(L),
        longer([0,0|L], f(X)).

zeros([]).
zeros([0|L]):- zeros(L).
```

For this program, the following types and signatures are inferred:

$T_{lz} \rightarrow [\,]; [T_0 \mid T_{lz}]$ $\quad$ $T_f \rightarrow f(T_f)$ $\quad$ $longer(T_{lz}, T_f)$
$T_0 \rightarrow 0$ $\quad$ $zeros(T_{lz})$

Figure 5 shows the moded-typed SLD-derivation for the moded-typed query `longer(L, X)`, with `L` of type T_{lz}, using 3 as a repetition number. All pairs of nodes in the derivations fail the third condition of Definition 8. □

To prove non-termination for such a program, we define an input-generalization. This input-generalization is such that proving non-termination of an input-generalized selected atom implies non-termination of the original goal.

Definition 9. *A moded-typed atom A^α is an input-generalization of a moded-typed atom A, if there exist terms $t_1, \ldots, t_n$ in A, such that no t_i is a subterm of t_j, and fresh moded-typed variables $\underline{I_1}, \ldots, \underline{I_n}$ with the same types as*

$t_1, \ldots, t_n$, *respectively, such that* $A^\alpha = A(\underline{I_1} \to t_1, \ldots, \underline{I_n} \to t_n)$ *and* $Var(A^\alpha) \cap Var((t_1, \ldots, t_n)) = \emptyset$. □

Example 13. Let A be the atom $longer([0, X, X], Y)$:

- $longer([0|\underline{I}], Y)$ is an input-generalization of A, with $(\underline{I} \to [X, X])$,
- $longer(\underline{I_1}, \underline{I_2})$ is an input-generalization of A, with $(\underline{I_1} \to [0, X, X], \underline{I_2} \to Y)$,
- $longer([0, X|\underline{I}], Y)$ is not an input-generalization of A. The replacement is $(\underline{I} \to [X])$. This violates the condition of the empty intersection of the variable sets. □

To check if a path is non-terminating w.r.t. an input-generalized goal, we define an *input-generalized derivation*. This derivation is constructed by applying a path in a given derivation to the input-generalized selected atom of the first node in the path.

Definition 10. *Let* N_i *and* N_j *be nodes in a moded-typed SLD-derivation with NFG* D*, such that* $A_i^1 \prec_{anc} A_j^1$*. Let* $\langle C_1, \ldots, C_n \rangle$ *be the sequence of derivation steps and transitions from* N_i *to* N_j *and let* A^α *be an input-generalization of* A_i^1*.*

If $\langle C_1, \ldots, C_n \rangle$ *can be applied to* $\leftarrow A^\alpha$*, the resulting derivation is called the* ***input-generalized derivation*** D' *for* A^α*. The* ***input-generalized nodes*** N_i^α *and* N_j^α *are the first and last node of* D'*, respectively.* □

Note that such a path in a moded-typed SLD-derivation is applicable to the input-generalized atom, if the selected atoms at all *NFG* transitions are still non-failing.

Non-termination of the input-generalized derivation implies non-termination of the original goal.

Proposition 2 (Non-termination with input-generalization). *Let* $N_i : G_i$ *and* $N_j : G_j$ *be nodes in a moded-typed SLD-derivation with NFG* D *of a program* P *for a query* I *and let* A^α *an input-generalization of* A_i^1*. Let* N_i^α *and* N_j^α *be input-generalized nodes in the input-generalized derivation* D' *for* A^α*.*

If $N_i^\alpha \overset{inc}{\to} N_j^\alpha$*, then every concrete goal of the extended denotation of* G_i *is non-terminating w.r.t. program* P*.* □

Example 14. To prove non-termination of the program in Example 12, $longer([0, 0|\underline{M}], f(\underline{I}))$ can be used as an input-generalization of A_2^1 in Figure 5.

Figure 6, shows the input-generalized derivation of N_2 to N_4 for $longer([0, 0 \mid \underline{M}], f(\underline{I}))$. This derivation is an inclusion loop: $N_2^\alpha \overset{inc}{\to} N_4^\alpha$. Therefore, non-termination of this program w.r.t. the concrete goals of $Ext(\leftarrow longer([0, 0 \mid \underline{M}], f(X)))$ is proven by Proposition 2. □

4.3 Program specialization

In Example 3, we discussed a class of programs, related to aliased variables, for which current non-termination analyzers fail to prove non-termination. We

illustrated that program specialization can be used to prove non-termination of such programs. The main intuition is that non-termination analysis techniques have difficulties with treating queries with aliased variables. Program specialization often reduces the aliasing, due to argument filtering. So, in the context of aliasing, applying program specialization often improves the applicability of the analysis.

Program specialization can also be used in combination with the *NFG* transition. When solving a non-failing atom, all variables in the atom are substituted with new input variables. These input variables give an overestimation of the possible values after evaluating the non-failing atom. Program specialization can produce more instantiated, but equivalent clauses. These more instantiated clauses give a better approximation of the possible values after evaluating the non-failing atom. We illustrate this with an example.

Example 15. The following program generates the infinite list of Hamming numbers in symbolic notation. *hamming/0* starts the computation initializing the list of hamming numbers to `[s(0)]`. `hamming([N|Ns])` keeps a list of hamming numbers, ordered from small to large. Three new hamming numbers are generated using *times/3*, which defines multiplication on the symbolic notation. Then, *insert/3* merges these three numbers with `Ns` and removes duplicates, resulting in the list for the next iteration. The code for *insert/3* and *times/3* is omitted.

```
hamming:- hamming([s(0)]).
hamming([N|Ns]):- times(s(s(0)),N,N2), times(s(s(s(0))),N,N3),
     times(s(s(s(s(s(0))))),N,N5), insert([N2,N3,N5],Ns,Res),
     hamming(Res).
```

PolyTypes infers the following type definition:

$T_s \rightarrow 0; s(T_s)$ $\qquad\qquad$ $T_l \rightarrow [\,]; [T_s \mid T_l]$

The arguments of *hamming/1* and *insert/3* are of type T_l, the other arguments are of type T_s. Non-failure analysis shows that *insert/3* and *times/3* are non-failing if their last arguments are free variables.

When building the moded-typed SLD-tree for the query `hamming`, solving the non-failing atoms by applying an *NFG* transition, the moded-typed goal `hamming(Res)` is obtained at node N_7. In this goal, `Res` is introduced by an *NFG* transition. When applying clause 2 to this goal, `Res` is substituted by a compound term and thus, non-termination of `hamming` can not be proven.

To prove non-termination of `hamming`, one needs to know that the list in the recursive call `hamming(Res)` is again of the form `[N|Ns]`. This can be done using the program specialization technique: more specific programs. This technique generates a more instantiated version of clause 2:

```
hamming([N|Ns]):- times(s(s(0)),N,N2), times(s(s(s(0))),N,N3),
     times(s(s(s(s(s(0))))),N,N5), insert([N2,N3,N5],Ns,[R|Res]),
     hamming([R|Res]).
```

Non-termination of this specialized program using *NFG* transitions to solve *times/3* and *insert/3* is straightforward. □

5 Conclusion and Future work

In this paper, we identified classes of logic programs for which current analyzers fail to prove non-termination and we extended our non-termination analysis of [9] to overcome these limitations. As in [9], non-termination is proven by constructing a symbolic derivation tree, representing all derivations for a class of queries, and then proving that a path in this tree can be repeated infinitely.

The most important class of programs for which current analyzers fail, are programs for which no fixed sequence of clauses can be repeated infinitely. We have shown that non-failure information ([2]) can be used to abstract away from the exact sequence of clauses needed to solve non-failing goals. To use this non-failure information, type information is added to the symbolic derivation tree and a special *NFG* transition is introduced to solve non-failing atoms. As far as we know, this is the first time that non-failure information is used for non-termination analysis.

We illustrated that program specialization ([3]) can be used to overcome another limitation of current analyzers. If non-termination can not be proven due to aliased variables, redundant argument filtering may remove these duplicated variables from the program. Specialization can also be used in combination with the *NFG* transition. This transition approximates the effect of solving the atom. Program specialization may produce more instantiated clauses, giving a better approximation of the possible values after solving the non-failing goal.

In very recent work, we implemented the extensions proposed in this paper[3].

References

1. M. Bruynooghe, J.P. Gallagher, and W. Van Humbeeck. Inference of well-typing for logic programs with application to termination analysis. In *SAS 2005*, volume 3672 of *LNCS*, pages 35–51. Springer, 2005.
2. Saumya K. Debray, Pedro López-García, and Manuel V. Hermenegildo. Non-failure analysis for logic programs. In *ICLP*, pages 48–62, 1997.
3. M. Leuschel. Logic program specialisation. In *Partial Evaluation*, volume 1706 of *LNCS*, pages 155–188. Springer, 1998.
4. John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
5. M.T. Nguyen, M. Bruynooghe, D. De Schreye, and M. Leuschel. Program specialisation as a preprocessing step for termination analysis. In A. Geser and H. Sondergaard, editors, *WST 2006*, pages 7–11, 2006.
6. Étienne Payet and Frédéric Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006.
7. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.
8. Yi-dong Shen, Danny de Schreye, and Dean Voets. Termination prediction for general logic programs. *Theory Pract. Log. Program.*, 9(6):751–780, 2009.
9. D. Voets and D. De Schreye. A new approach to non-termination analysis of logic programs. In P. M. Hill and D. S. Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 220–234. Springer, 2009.
10. E. Yardeni and E. Y. Shapiro. A type system for logic programs. *J. Log. Program.*, 10(1/2/3&4):125–153, 1991.

[3] The analyzer and experiments can be found at http://www.cs.kuleuven.be/~dean

Author Index