Danny De Schreye (Ed.)

Logic-Based Program Synthesis and Transformation

LNCS 6037

19th International Symposium, LOPSTR 2009
Coimbra, Portugal, September 2009
Revised Selected Papers

Lecture Notes in Computer Science 6037

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Danny De Schreye (Ed.)

Logic-Based Program Synthesis and Transformation

19th International Symposium, LOPSTR 2009
Coimbra, Portugal, September 2009
Revised Selected Papers

Springer

Volume Editor

Danny De Schreye
K.U.Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
E-mail: danny.deschreye@cs.kuleuven.be

Library of Congress Control Number: 2010924453

CR Subject Classification (1998): F.3, D.3, D.2, F.4.1, I.2.3, F.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-12591-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-12591-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains a selection of the papers presented at the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2009) held September 9-11, 2009 in Coimbra, Portugal. Information about the conference can be found at `http://www.cs.kuleuven.be/conference/lopstr09+`. Previous LOPSTR symposia were held in 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).

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. LOPSTR traditionally solicits papers in the areas of specification, synthesis, verification, transformation, analysis, optimization, composition, security, reuse, applications and tools, component-based software development, software architectures, agent-based software development, and program refinement. 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 any feedback in the published papers.

I would like to thank all those who submitted contributions to LOPSTR in the categories of full papers and extended abstracts. Each submission was reviewed by at least three Program Committee members. The committee decided to accept three full papers for immediate inclusion in the final proceedings, and ten papers were accepted after revision and another round of reviewing. In addition to the accepted papers, the program also included an invited talk by Germán Vidal (Technical University of Valencia).

I am grateful to the Program Committee members who worked hard to produce high-quality reviews for the submitted papers in a tight schedule, as well as all the external reviewers involved in the paper selection. I also would like to thank Andrei Voronkov for his excellent EasyChair system that automates many of the tasks involved in chairing a conference.

LOPSTR 2009 was co-located with PPDP 2009 and CSL 2009. Many thanks to the local organizers of these events, in particular, to Ana Almeida, the LOPSTR 2009 Local Organization Chair.

January 2010 Danny De Schreye

Conference Organization

Program Chair

Danny De Schreye
Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
Email: danny.deschreye@cs.kuleuven.be

Local Organization Chair

Ana Almeida
Departamento de Matematica
Faculdade de Ciencias e Tecnologia
Universidade de Coimbra
Coimbra, Portugal
Email: amca@mat.uc.pt

Program Committee

Slim Abdennadher	German University Cairo, Egypt
María Alpuente Frasnedo	Technical University of Valencia, Spain
Roberto Bagnara	University of Parma, Italy
Danny De Schreye	K.U. Leuven, Belgium (Chair)
John Gallagher	Roskilde University, Denmark
Robert Glück	University of Copenhagen, Denmark
Michael Hanus	University of Kiel, Germany
Reinhard Kahle	Universidade Nova de Lisboa, Portugal
Andy King	University of Kent, UK
Michael Leuschel	University of Düsseldorf, Germany
Fabio Martinelli	Istituto di Informatica e Telematica Pisa, Italy
Fred Mesnard	Université de La Réunion, France
Mario Ornaghi	Università degli Studi di Milano, Italy
Germán Puebla	Technical University of Madrid, Spain
Sabina Rossi	Università Ca' Foscari di Venezia, Italy
Josep Silva	Technical University of Valencia, Spain
Peter Schneider-Kamp	University of Southern Denmark, Denmark
Tom Schrijvers	K.U. Leuven, Belgium
Petr Stepanek	Charles University Prague, Czech Republic
Wim Vanhoof	University of Namur, Belgium

Organizing Committee

Ana Almeida
Pedro Quaresma
Reinhard Kahle

External Reviewers

Jesper Louis Andersen
Ulrich Berger
Pedro Cabalar
François Degrave
Camillo Fiorentini
Emilio Jesus Gallego Arias
Pepe Iborra
Leanid Krautsevich
Gift Nuka
Paolo Pilozzi
Juan Rodriguez-Hortalá
Anton Setzer
Peter Van Weert
Gianluigi Zavattaro
Federico Bergenti
Carl Friedrich Bolz
Gabriele Costa
Marc Denecker
Sebastian Fischer
Michael Gelfond
Haythem Ismail
Joao Leite
Etienne Payet
Frank Raiser
Cesar Sanchez
Maja Tonnesen
Dean Voets

Table of Contents

Towards Scalable Partial Evaluation of Declarative Programs*

Germán Vidal

DSIC, Universidad Politécnica de Valencia, Spain
gvidal@dsic.upv.es

1 Introduction

Partial evaluation is a well-known technique for program specialization [4]. 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. The residual program is then used for performing the remaining computations—those that depend on the so-called *dynamic* data.

There are two main approaches to partial evaluation, depending on the way termination issues are addressed. On the one hand, *online* partial evaluators take decisions on the fly while the constructs of the source code are partially evaluated and the corresponding residual program is built. *Offline* partial evaluators, on the other hand, require a *binding-time analysis* (BTA) to be run before specialization, which annotates the source code to be specialized. Basically, every call of the source program is annotated as either unfold (to be executed by the partial evaluator) or memo (to be executed at run time, i.e., memoized), and every argument is annotated as static (known at specialization time) or dynamic (only definitely known at run time). Offline partial evaluators are usually faster but less accurate than online ones since the BTA phase is performed—and also termination issues are addressed—using an *approximation* of the static data.

There are several basic properties of a partial evaluator that can be addressed:

- *correctness:* is the specialized program equivalent to the original one for the considered static data?
- *accuracy:* is the residual program a good specialization of the original program for the static data? is it fast enough compared to a hand-written specialization?
- *efficiency:* is the partial evaluator fast? does it scale up well to large source programs?
- *predictability:* is it possible to determine the achievable run time speedup *before* partial evaluation starts?

Here, we are mainly concerned with efficiency issues in offline partial evaluation.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by *Generalitat Valenciana* under grant ACOMP/2009/017, and by *UPV* (programs PAID-05-08 and PAID-06-08).

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 1–4, 2010.

2 Accuracy vs. Efficiency

Clearly, there is a trade-off between accuracy and efficiency. For instance, some accurate BTAs are rather inefficient because the termination analysis and the algorithm for propagating static information should be interleaved, so that every time a call is annotated as memo, the termination analysis has to be re-executed to take into account that some bindings will not be propagated anymore.

Consider, for instance, the following logic programming clause:

$$p(s(X), s(Y)) \leftarrow q(X, Z),\ p(Z, Y).$$

with variable X static[1] and variable Y dynamic. A traditional BTA initially marks every predicate call as unfold and proceeds as follows:

- First, it runs a procedure for propagating static information (i.e., a sort of groundness analysis). Let us assume that, in every successful computation for $q(X, Z)$ with X ground, variable Z becomes ground too. Therefore, by assuming a fixed left-to-right selection strategy, this procedure may conclude that variable Z in $p(Z, Y)$ is static too.
- Now, a termination analysis—that takes into account which variables are marked as static—is used to infer annotations for predicate calls. We also consider a fixed left-to-right selection strategy (i.e., a so called left-termination analysis). Let us now assume that this analysis is not able to ensure termination for predicate q and, thus, it is now marked as memo.
- Since q is annotated as memo, the call to q will not be unfolded at partial evaluation time. Therefore, we should run again the procedure for propagating static information, now assuming that q is not unfolded. Clearly, this will imply that we cannot ensure that variable Z is static in $p(Z, Y)$ anymore. Therefore, since the static/dynamic annotations have changed, the termination analysis should also be run again, and so forth. This iterative process is computationally very expensive, so it does seem a good candidate as a basis for designing a scalable partial evaluator.

Our recent work [1,6,7,8,9] shows that this drawback can be overcome by using instead a *strong* termination analysis [3], i.e., an analysis that considers termination for every possible selection or evaluation strategy. In this case, both tasks—termination analysis and propagation of static information—can be kept independent, so that the termination analysis is done once and for all before the propagation phase, resulting in major efficiency improvements over previous approaches.

For instance, given the previous clause, $p(s(X), s(Y)) \leftarrow q(X, Z),\ p(Z, Y).$, the BTA would now proceed as follows:

- First, the strong termination analysis is executed. For this purpose, we have adapted the size-change analysis originally introduced for functional programs by Lee, Jones and Ben-Amram [5]. Roughly speaking, this analysis

[1] For simplicity, here we assume that a static variable is ground.

traces the size changes of arguments when going from one call to another by means of so called *size-change graphs*.
The strong termination analysis for logic programs was introduced in [9] and later refined and extended in [7,6]. Basically, for every program clause $H \leftarrow B_1, \ldots, B_n$, the analysis constructs n size-change graphs, each of them stating the relation between the sizes of the arguments of H and the sizes of the arguments of every atom B_i in the body. For instance, for the above clause, the analysis constructs two size-change graphs, one that relates the sizes of $(s(X), s(Y))$ and (X, Z), and another one that relates the sizes of $(s(X), s(Y))$ and (Z, Y).[2] The output of the analysis is then a set of conditions for the termination of every predicate call that depends on which variables are marked as static.

– Then, in a second step, the BTA applies a standard procedure for the propagation of static/dynamic information that uses the output of the strong termination analysis to infer the right annotations for both predicate calls and their arguments. The details of this procedure can be found in [7].

As expected, the accuracy of the resulting scheme is not comparable to that of previous approaches, but can nevertheless be improved in a number of ways:

– Firstly, the information gathered from a left-termination analysis (which would be run only once) can still be used to improve the accuracy in those cases where the order of evaluation is partially known. For instance, for the clause above, if we know that $q(X, Z)$ always terminates with a left-to-right selection rule (e.g., because q is not recursive), then one can safely mark q as unfold and take it into account in the size-change analysis to propagate some additional static information to the calls that occur to its right ($p(Z, Y)$ in the example, so that the size relation between $s(X)$ and Z can again be inferred, as in the traditional approach).
– Secondly, we could allow the user to provide manual annotations to improve the accuracy in some cases. We have experimentally checked that even for large programs, a few annotations suffice to get an optimal result [7].
– We may also replace some memo and/or dynamic annotations by online, a new annotation that delays the corresponding decisions to partial evaluation time. We note, however, that one should be very careful with these annotations since they may involve expensive computations at partial evaluation time (i.e., some heuristics is needed to decide when replacing memo/dynamic by online might be critical to get a good specialization).

3 Concluding Remarks

Although there is ample room for improving accuracy, we consider our approach a promising framework for developing scalable partial evaluators for declarative

[2] Observe that, in contrast to the traditional approach that uses a left-termination analysis, now we cannot infer any size relation between $s(X)$ and Z (and, as a consequence, the termination of p cannot be proved). Some possibilities to improve this situation are mentioned in the following.

programs. A promising line of research is based on the use of SAT solving techniques to improve the accuracy of the BTA while still keeping its scalability. For instance, one could extend the SAT-based approach to size-change analysis of [2] in order to compute unfold/memo annotations in polynomial time (despite the fact that a left-termination analysis is considered).

Finally, we note that the underlying techniques are essentially the same no matter the considered declarative programming language. Actually, we have applied similar principles to the partial evaluation of both functional (logic) programs [8,1] and logic programs [9,7,6].

Acknowledgements. I would like to thank the participants of LOPSTR 2009 for their useful feedback. I would also like to thank Michael Codish for suggesting the use of SAT-based techniques to improve the accuracy of the BTA.

References

1. Arroyo, G., Ramos, J.G., Silva, J., Vidal, G.: Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 60–76. Springer, Heidelberg (2007)
2. Ben-Amram, A., Codish, M.: A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2007. LNCS, vol. 5028, pp. 46–55. Springer, Heidelberg (2008)
3. Bezem, M.: Strong Termination of Logic Programs. Journal of Logic Programming 15(1,2), 79–97 (1993)
4. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
5. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. In: SIGPLAN Notices (Proc. of POPL 2001), vol. 28, pp. 81–92 (2001)
6. Leuschel, M., Tamarit, S., Vidal, G.: Fast and Accurate Size-Change Strong Termination Analysis with an Application to Partial Evaluation. In: Escobar, S. (ed.) WFLP 2009. LNCS, vol. 5979, pp. 111–127. Springer, Heidelberg (2009)
7. Leuschel, M., Vidal, G.: Fast Offline Partial Evaluation of Large Logic Programs. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 119–134. Springer, Heidelberg (2009)
8. Ramos, J.G., Silva, J., Vidal, G.: Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In: Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2005), pp. 228–239. ACM Press, New York (2005)
9. Vidal, G.: Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In: Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM 2007), pp. 51–60. ACM Press, New York (2007)

Deciding Full Branching Time Logic by Program Transformation

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

[1] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[2] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
proietti@iasi.cnr.it

Abstract. We present a method based on logic program transformation, for verifying Computation Tree Logic (CTL*) properties of finite state reactive systems. The finite state systems and the CTL* properties we want to verify, are encoded as logic programs on infinite lists. Our verification method consists of two steps. In the first step we transform the logic program that encodes the given system and the given property, into a *monadic ω-program*, that is, a stratified program defining nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold rules that preserve the perfect model of the initial program. In the second step we verify the property of interest by using a proof method for monadic ω-programs.

1 Introduction

The branching time temporal logic CTL* is among the most popular temporal logics that have been proposed for verifying properties of reactive systems [4]. A finite state reactive system, such as a protocol, a concurrent system, or a digital circuit, is formally specified as a Kripke structure and the property to be verified is specified as a CTL* formula. Thus, the problem of checking whether or not a reactive system satisfies a given property is reduced to the problem of checking whether or not a Kripke structure is a model of a CTL* formula.

There is a vast literature on the problem of model checking for the CTL* logic and, in particular, its two fragments: (i) the Computational Tree Logic CTL, and (ii) the Linear-time Temporal Logic LTL (see [2] for a survey). Most of the known model checking algorithms for CTL* either combine model checking algorithms for CTL and LTL [2], or use techniques based on translations to automata on infinite trees [6].

In this paper we extend to CTL* a method proposed in [11] for LTL. We encode the satisfaction relation of a CTL* formula φ with respect to a Kripke structure $\mathcal{K}$ by means of a locally stratified logic program $P_{\mathcal{K},\varphi}$. The program $P_{\mathcal{K},\varphi}$ belongs to a class of programs, called ω-programs, which define predicates on infinite lists. Predicates of this type are needed because the definition of the satisfaction relation is based on the infinite computation paths of $\mathcal{K}$. The semantics of $P_{\mathcal{K},\varphi}$ is provided by its unique *perfect model* [12] which for ω-programs is defined in terms of a non-Herbrand interpretation for infinite lists.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 5–21, 2010.

Our verification method consists of two steps. In the first step we transform the program $P_{\mathcal{K},\varphi}$ into a *monadic* ω-program, that is, a stratified program that defines nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules similar to those presented in [5,14,15] according to a strategy which is a variant of the *specialization strategy* presented in [5]. Similarly to [5,14], the use of those unfold/fold rules guarantees the preservation of the perfect model of $P_{\mathcal{K},\varphi}$.

In the second step of our verification method we apply a proof method for monadic ω-programs which is sound and complete with respect to the perfect model semantics.

The paper is structured as follows. In Section 2 we introduce the class of ω-programs and we show how to encode the satisfaction relation for any given Kripke structure and CTL* formula as an ω-program. In Section 3 we present our verification method. In particular, in Section 3.1 we present the specialization strategy for transforming an ω-program into a monadic ω-program and in Section 3.2 we present the proof method for monadic ω-programs. Finally, in Section 4 we discuss related work in the area of model checking and logic programming.

2 Encoding CTL* Model Checking as a Logic Program

In this section we describe a method which, given a Kripke structure $\mathcal{K}$ and a CTL* *state formula* φ, allows us to construct a logic program $P_{\mathcal{K},\varphi}$ and to define a nullary predicate *prop* such that φ is true in $\mathcal{K}$, written $\mathcal{K} \vDash \varphi$, iff *prop* is true in the perfect model of $P_{\mathcal{K},\varphi}$, written $M(P_{\mathcal{K},\varphi}) \vDash \mathit{prop}$. Thus, the problem of checking whether or not $\mathcal{K} \vDash \varphi$ holds, also called the problem of model checking φ with respect to $\mathcal{K}$, is reduced to the problem of testing whether or not $M(P_{\mathcal{K},\varphi}) \vDash \mathit{prop}$ holds.

Now we briefly recall the definition of the temporal logic CTL* (see [2] for more details). A Kripke structure is a 4-tuple $\langle \Sigma, s_0, \rho, \lambda \rangle$, where: (i) $\Sigma = \{s_0, \ldots, s_h\}$ is a finite set of *states*, (ii) $s_0 \in \Sigma$ is the *initial state*, (iii) $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, and (iv) $\lambda : \Sigma \to \mathcal{P}(\mathit{Elem})$ is a total function that assigns to every state $s \in \Sigma$ a subset $\lambda(s)$ of the set *Elem* of *elementary properties*. A *computation path* of $\mathcal{K}$ from a state s is an infinite list $[a_0, a_1, \ldots]$ of states such that $a_0 = s$ and, for every $i \geq 0$, $(a_i, a_{i+1}) \in \rho$. Given an infinite list $\pi = [a_0, a_1, \ldots]$ of states, by π_j, for any $j \geq 0$, we denote the infinite list which is the suffix $[a_j, a_{j+1}, \ldots]$ of π.

Definition 1 (CTL* Formulas). Given a set *Elem* of elementary properties, a CTL* formula φ is either a *path formula* φ_p or a *state formula* φ_s defined as follows:

(*path formulas*) $\varphi_p ::= \varphi_s \mid \neg\varphi_p \mid \varphi_p \wedge \varphi_p \mid \mathsf{X}\, \varphi_p \mid \varphi_p\, \mathsf{U}\, \varphi_p$
(*state formulas*) $\varphi_s ::= d \mid \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \mathsf{E}\, \varphi_p$

where $d \in \mathit{Elem}$.

As the following definition formally specifies, (i) $\mathsf{X}\,\varphi$ holds on a computation path π if φ holds in the second state of π, (ii) $\varphi_1\,\mathsf{U}\,\varphi_2$ holds on a computation path π if φ_2 holds in a state s of π and φ_1 holds in every state preceding s in π, and (iii) $\mathsf{E}\,\varphi$ holds in a state s if there exists a computation path starting from s on which φ holds.

Definition 2 (Satisfaction Relation for CTL*). Let $\mathcal{K} = \langle \Sigma, s_0, \rho, \lambda \rangle$ be a Kripke structure. For any CTL* formula φ and infinite list $\pi \in \Sigma^\omega$, the relation $\mathcal{K}, \pi \vDash \varphi$ is inductively defined as follows:

$\mathcal{K}, \pi \vDash d$ iff $\pi = [a_0, a_1, \ldots]$ and $d \in \lambda(a_0)$
$\mathcal{K}, \pi \vDash \neg\,\varphi$ iff $\mathcal{K}, \pi \nvDash \varphi$
$\mathcal{K}, \pi \vDash \varphi_1 \wedge \varphi_2$ iff $\mathcal{K}, \pi \vDash \varphi_1$ and $\mathcal{K}, \pi \vDash \varphi_2$
$\mathcal{K}, \pi \vDash \mathsf{X}\,\varphi$ iff $\mathcal{K}, \pi_1 \vDash \varphi$
$\mathcal{K}, \pi \vDash \varphi_1\,\mathsf{U}\,\varphi_2$ iff there exists $i \geq 0$ such that $\mathcal{K}, \pi_i \vDash \varphi_2$ and, for all $0 \leq j < i$, $\mathcal{K}, \pi_j \vDash \varphi_1$
$\mathcal{K}, \pi \vDash \mathsf{E}\,\varphi$ iff $\pi = [a_0, a_1, \ldots]$ and there exists a computation path π' from a_0 such that $\mathcal{K}, \pi' \vDash \varphi$.

Given a *state formula* φ, we say that $\mathcal{K}$ is a *model* of φ, written $\mathcal{K} \vDash \varphi$, iff there exists an infinite list $\pi \in \Sigma^\omega$ such that the first state of π is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \vDash \varphi$ holds.

The above definition of the satisfaction relation for CTL* formulas is a shorter, yet equivalent, version of the usual definition one can find in the literature [2].

In order to encode the satisfaction relation for CTL* formulas as a logic program, we will introduce in the next section a class of logic programs, called *ω-programs*. In this class the arguments of predicates may denote infinite lists.

2.1 Syntax and Semantics of ω-Programs

Let us consider a Kripke structure $\mathcal{K}$. Let us also consider a first order language $\mathcal{L}_\omega$ given by a set *Var* of variables, a set *Fun* of function symbols, and a set *Pred* of predicate symbols. We assume that *Fun* includes: (i) the set Σ of the states of $\mathcal{K}$, each state being a constant of $\mathcal{L}_\omega$, (ii) the set *Elem* of the elementary properties of $\mathcal{K}$, each elementary property being a constant of $\mathcal{L}_\omega$, and (iii) the binary function symbol $[\_|\_]$ which is the constructor of infinite lists. Thus, for instance, $[H|T]$ is the infinite list whose head is H and whose tail is the infinite list T.

We assume that $\mathcal{L}_\omega$ is a typed language [9] with the following three basic types: (i) `fterm`, which is the type of finite terms, (ii) `state`, which is the type of states, and (iii) `ilist`, which is the type of infinite lists of states. Every function symbol in $\mathit{Fun} - (\Sigma \cup \{[\_|\_]\})$, with arity $n\,(\geq 0)$, has type $\texttt{fterm} \times \cdots \times \texttt{fterm} \rightarrow \texttt{fterm}$, where `fterm` occurs n times to the left of $\rightarrow$. Every function symbol in Σ has arity 0 and type `state`. The function symbol $[\_|\_]$ has type $\texttt{state} \times \texttt{ilist} \rightarrow \texttt{ilist}$. A predicate symbol of arity $n\,(\geq 0)$ in *Pred* has type of the form $\tau_1 \times \cdots \times \tau_n$, where $\tau_1, \ldots, \tau_n \in \{\texttt{fterm}, \texttt{state}, \texttt{ilist}\}$. An *ω-program* is a logic program constructed

as usual (see, for instance, [9]) from symbols in the typed language $\mathcal{L}_\omega$. In what follows, for reasons of simplicity, we will feel free to say 'program', instead of 'ω-program'.

Given a term or a formula t, by $vars(t)$ we denote the set of variables occurring in t. The same notation will be used for sets of terms and sets of formulas. The *existential closure* of a formula φ, denoted $\exists(\varphi)$, is the formula $\exists X_1 \ldots \exists X_n\, \varphi$ where $\{X_1, \ldots, X_n\}$ is the set of the *free variables* occurring in φ. The *universal closure* of a formula φ, denoted $\forall(\varphi)$, is defined in a similar way by using $\forall$, instead of $\exists$. Note that if $vars(\varphi) = \emptyset$, then $\exists(\varphi)$ is φ itself.

An interpretation for our typed language $\mathcal{L}_\omega$, called ω-interpretation, is given as follows. Let HU be the Herbrand universe constructed from the set $Fun - (\Sigma \cup \{[\_|\_]\})$ of function symbols and let Σ^ω be the set of the infinite lists of states. An ω-interpretation I is an interpretation such that: (i) I assigns to the types `fterm`, `state`, and `ilist`, respectively, the sets HU, Σ, and Σ^ω, (ii) I assigns to the function symbol $[\_|\_]$ the function $[\_|\_]_I$ such that, for any state $a \in \Sigma$ and infinite list $[a_1, a_2, \ldots] \in \Sigma^\omega$, $[a|[a_1, a_2, \ldots]]_I$ is the infinite list $[a, a_1, a_2, \ldots]$, (iii) I is an Herbrand interpretation for all function symbols in $Fun - (\Sigma \cup \{[\_|\_]\})$, and (iv) I assigns to every n-ary predicate $p \in Pred$ of type $\tau_1 \times \ldots \times \tau_n$ a relation on $D_1 \times \cdots \times D_n$, where, for $i = 1, \ldots, n$, D_i is either HU or Σ or Σ^ω, according to the case where τ_i is either `fterm` or `state` or `ilist`, respectively. We say that an ω-interpretation I is an *ω-model* of a program P iff for every clause $\gamma \in P$ we have that $I \models \forall(\gamma)$.

A *level mapping* is a function $\ell : Pred \to \mathbb{N}$. A level mapping is extended to literals as follows: for any literal L having predicate p, if L is a positive literal, then $\ell(L) = \ell(p)$ and, if L is a negative literal then $\ell(L) = \ell(p) + 1$. An ω-clause γ of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$ is *stratified* w.r.t. ℓ if, for $i = 1, \ldots, m$, $\ell(H) \geq \ell(L_i)$. An ω-program P is *stratified* if there exists a level mapping ℓ such that all clauses of P are stratified w.r.t. ℓ.

A *valuation* is a function $v : Var \to HU \cup \Sigma \cup \Sigma^\omega$ such that: (i) if X has type `fterm` then $v(X) \in HU$, (ii) if X has type `state` then $v(X) \in \Sigma$, and (iii) if X has type `ilist` then $v(X) \in \Sigma^\omega$. For any term t, literal L, and clause γ, we define $v(t)$, $v(L)$, and $v(\gamma)$, by induction on the structure of t, L, and γ, respectively. We will say that $v(t)$, $v(L)$, and $v(\gamma)$, is 'a term', 'a literal', and 'a clause', respectively, also when they are infinite structures.

We extend the notion of *Herbrand base* [9] to the case of ω-programs by introducing the set $\mathcal{B}_\omega$ defined as follows:

$$\mathcal{B}_\omega = \{p(v(X_1), \ldots, v(X_n)) \mid p \text{ is an } n\text{-ary predicate symbol and } v \text{ is a valuation}\}$$

Thus, any ω-interpretation can be identified with a subset of $\mathcal{B}_\omega$.

A *local stratification* is a function σ: $\mathcal{B}_\omega \to W$, where W is the set of countable ordinals. Given $A \in \mathcal{B}_\omega$, we define $\sigma(\neg A) = \sigma(A) + 1$. Given a clause γ of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$ in an ω-program P and a local stratification σ, we say that γ is *locally stratified* w.r.t. σ if for $i = 1, \ldots, m$, for every valuation v, $\sigma(v(H)) \geq \sigma(v(L_i))$. An ω-program P is *locally stratified w.r.t. σ*, or σ is a

local stratification for P, if every clause in P is locally stratified w.r.t. σ. An ω-program P is *locally stratified* if there exists a local stratification σ such that P is *locally stratified w.r.t.* σ.

Clearly, every stratified ω-program is a locally stratified ω-program. Similarly to the case of logic programs, for every locally stratified ω-program P (and, hence, for every stratified ω-program P), we can construct a unique *perfect* ω-*model* (or *perfect model*, for short) denoted by $M(P)$ [1,12] (an instance of this construction is presented in Example 1).

Definition 3 (Monadic ω-Programs). A *monadic* ω-clause is an ω-clause of the form $A_0 \leftarrow L_1 \wedge \ldots \wedge L_m$, with $m \geq 0$, such that: (i) A_0 is an atom of the form p_0 or $q_0([s|X_0])$, where q_0 is a predicate of type `ilist` and $s \in \Sigma$, (ii) for $i = 1, \ldots, m$, L_i is either an atom A_i or a negated atom $\neg A_i$, where A_i is of the form p_i or $q_i(X_i)$, and q_i is a predicate of type `ilist`, and (iii) there exists a level mapping ℓ such that, for $i = 1, \ldots, m$, if L_i is an atom and $vars(A_0) \not\supseteq vars(L_i)$, then $\ell(A_0) > \ell(L_i)$ else $\ell(A_0) \geq \ell(L_i)$. A *monadic* ω*-program* is a finite set of monadic ω-clauses.

Note that in Definition 3 the predicate symbols $p_0, q_0, \ldots, p_m, q_m$ and the variables $X_0, \ldots, X_m$ are *not* necessarily distinct. Condition (iii) ensures that a monadic ω-program is stratified. This condition, which is actually stronger than stratification, is also needed for guaranteeing the completeness of the proof method for monadic ω-programs (see Section 3.2).

Example 1. Let r, q, and p be predicates of type `ilist`. The following set of clauses is a monadic ω-program P (and, thus, also an ω-program):

$$
\begin{array}{lll}
p([a|X]) \leftarrow p(X) & q([a|X]) \leftarrow q(X) & r([a|X]) \leftarrow r(X) \\
p([b|X]) \leftarrow \neg\, q(X) & q([a|X]) \leftarrow \neg\, r(X) & r([b|X]) \leftarrow \\
 & q([b|X]) \leftarrow q(X) &
\end{array}
$$

Program P is stratified by the level mapping $\ell : \mathit{Pred} \rightarrow \mathbb{N}$ such that $\ell(p) = 2$, $\ell(q) = 1$, and $\ell(r) = 0$. The perfect model $M(P)$ is constructed starting from the ground atoms of level 0 and going up, level-by-level, as we now indicate. We start from the ground atoms of level 0, that is, the ground atoms with predicate r. For all $w \in \{a, b\}^\omega$, $r(w) \in M(P)$ iff $w \in a^*b(a + b)^\omega$. Thus, $r(w) \notin M(P)$ iff $w \in a^\omega$, that is, $\neg\, r(w)$ holds in $M(P)$ iff $w \in a^\omega$. Then we consider the ground atoms of level 1, that is, the ground atoms with predicate q. For all $w \in \{a, b\}^\omega$, $q(w) \in M(P)$ iff $w \in (a+b)^*a^\omega$ (that is, w has finitely many occurrences of b). Thus, $\neg\, q(w)$ holds in $M(P)$ iff $w \in (a^*b)^\omega$ (that is, w has infinitely many occurrences of b). Finally, we consider the ground atoms of level 2, that is, the ground atoms with predicate p. For all $w \in \{a, b\}^\omega$, $p(w) \in M(P)$ iff $w \in (a^*b)(a^*b)^\omega$, that is, $p(w) \in M(P)$ iff $w \in (a^*b)^\omega$.

2.2 Encoding the CTL* Satisfaction Relation as an ω-Program

Given a Kripke structure $\mathcal{K}$ and a CTL* state formula φ, we introduce a locally stratified ω-program $P_{\mathcal{K},\varphi}$ which defines, among others, the following three predicates: (i) the unary predicate *path* such that $path(\pi)$ holds iff π is an infinite list

representing a computation path of $\mathcal{K}$, (ii) the binary predicate *sat* that encodes the satisfaction relation for CTL* formulas, in the sense that for all computation paths π and CTL* formulas ψ, we have that $M(P_{\mathcal{K},\varphi}) \models sat(\pi, \psi)$ iff $\mathcal{K}, \pi \models \psi$, and (iii) the nullary predicate *prop* that encodes the property φ to be verified, in the sense that *prop* holds iff there exists an infinite list π whose first element is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \models \varphi$.

When writing terms that encode CTL* formulas, such as the second argument of the predicate *sat*, we will use the function symbols e, x, and u standing for the operator symbols E, X, and U, respectively.

Definition 4 (Encoding Program). Given a Kripke structure $\mathcal{K} = \langle \Sigma, s_0, \rho, \lambda \rangle$ and a CTL* formula φ, the *encoding program* $P_{\mathcal{K},\varphi}$ is the following ω-program:

1. $prop \leftarrow sat([s_0|X], \varphi)$
2. $sat([S|X], F) \leftarrow elem(F, S)$
3. $sat(X, not(F)) \leftarrow \neg\, sat(X, F)$
4. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1) \wedge sat(X, F_2)$
5. $sat([S|X], x(F)) \leftarrow sat(X, F)$
6. $sat(X, u(F_1, F_2)) \leftarrow sat(X, F_2)$
7. $sat([S|X], u(F_1, F_2)) \leftarrow sat([S|X], F_1) \wedge sat(X, u(F_1, F_2))$
8. $sat([S|X], e(F)) \leftarrow exists\_sat(S, F)$
9. $exists\_sat(S, F) \leftarrow path([S|Y]) \wedge sat([S|Y], F)$
10. $path(X) \leftarrow \neg\, notpath(X)$
11. $notpath([S_1, S_2|X]) \leftarrow \neg\, tr(S_1, S_2)$
12. $notpath([S|X]) \leftarrow notpath(X)$

together with the clauses defining the predicates *tr* and *elem*, where:
(1) for all states $s_1, s_2 \in \Sigma$, $tr(s_1, s_2)$ holds iff $(s_1, s_2) \in \rho$, and
(2) for every property $d \in Elem$ and state $s \in \Sigma$, $elem(d, s)$ holds iff $d \in \lambda(s)$.

Clause 1 of Definition 4 asserts that the property φ holds for an infinite list of states whose first element is s_0. Clauses 2–9 define the satisfaction relation $sat(X, \varphi)$ for any infinite list X and CTL* formula φ. The definition of $sat(X, \varphi)$ is by structural induction on φ. Clauses 10–12 establish that $path(X)$ holds iff for every pair (a_i, a_{i+1}) of consecutive elements on the infinite list X, we have that $(a_i, a_{i+1}) \in \rho$. Indeed, clauses 11 and 12 establish that $notpath(X)$ holds iff in the list X there exist two consecutive elements a_i and a_{i+1} such that $(a_i, a_{i+1}) \notin \rho$.

The program $P_{\mathcal{K},\varphi}$ is locally stratified w.r.t. the stratification function σ from ground literals to natural numbers, defined as follows (in what follows, for any CTL* formula χ, we will denote by $|\chi|$ the number of occurrences of function symbols in r): for all states $a \in \Sigma$, for all infinite lists $\pi \in \Sigma^\omega$, and for all CTL* formulas ψ, (i) $\sigma(prop) = |\varphi| + 1$, where $prop \leftarrow sat([s_0|X], \varphi)$, (ii) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, (iii) $\sigma(exists\_sat(a, \psi)) = |\psi| + 2$, (iv) $\sigma(path(\pi)) = 2$, (v) $\sigma(notpath(\pi)) = 1$, (vi) for every ground atom A, $\sigma(\neg A) = \sigma(A) + 1$, and (vii) in all other cases σ returns 0.

Example 2. Let us consider: (i) the set $Elem = \{a, b, tt\}$ of elementary properties, where tt is the elementary property which holds in every state, and

(ii) the Kripke structure $\mathcal{K} = \langle\{s_0, s_1, s_2\}, s_0, \rho, \lambda\rangle$, where ρ is the transition relation $\{(s_0, s_0), (s_0, s_1), (s_1, s_1), (s_1, s_2), (s_2, s_1)\}$ and λ is the function such that $\lambda(s_0) = \{a\}$, $\lambda(s_1) = \{b\}$, and $\lambda(s_2) = \{a\}$. Let us also consider the formula $\varphi = \mathsf{E}\,(a\,\mathsf{U}\,\neg\,\mathsf{E}\,(tt\,\mathsf{U}\,\neg\,(tt\,\mathsf{U}\,b)))$, which can be abbreviated as $\mathsf{E}\,(a\,\mathsf{U}\,\mathsf{AGF}\,b)$, where: (i) for every state formula ψ, $\mathsf{F}\,\psi$ (read '*eventually* ψ') stands for $tt\,\mathsf{U}\,\psi$, and $\mathsf{G}\,\psi$ (read '*always* ψ') stands for $\neg\,\mathsf{F}\neg\,\psi$, and (ii) for every path formula ψ, $\mathsf{A}\,\psi$ (read '*for all computation paths* ψ') stands for $\neg\,\mathsf{E}\,\neg\,\psi$. The encoding program $P_{\mathcal{K},\varphi}$ is as follows:

$prop \leftarrow sat([s_0|X], e(u(a, not(e(u(tt, not(u(tt, b))))))))$
$tr(s_0, s_0) \leftarrow$ $tr(s_0, s_1) \leftarrow$ $tr(s_1, s_1) \leftarrow$ $tr(s_1, s_2) \leftarrow$ $tr(s_2, s_1) \leftarrow$
$elem(a, s_0) \leftarrow$ $elem(b, s_1) \leftarrow$ $elem(a, s_2) \leftarrow$ $elem(tt, S) \leftarrow$

together with clauses 2–12 of Definition 4 defining the predicates *sat*, *path*, and *notpath*.

Since $\mathcal{K} \models \varphi$ holds iff there exists an infinite list $\pi \in \Sigma^\omega$ such that the first state of π is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \models \varphi$ holds (see Definition 2), we have that the correctness of $P_{\mathcal{K},\varphi}$ can be expressed by stating that $\mathcal{K} \models \varphi$ holds iff $M(P_{\mathcal{K},\varphi}) \models \exists X\, sat([s_0|X], \varphi)$ iff (by clause 1 of Definition 4) $M(P_{\mathcal{K},\varphi}) \models prop$. The correctness of $P_{\mathcal{K},\varphi}$ is stated in the following theorem.

Theorem 1 (Correctness of the Encoding Program). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and a state formula φ. Then, $\mathcal{K} \models \varphi$ iff $M(P_{\mathcal{K},\varphi}) \models prop$.*

3 Transformational CTL* Model Checking

In this section we present a technique based on program transformation for checking whether or not, for any given structure $\mathcal{K}$ and state formula φ, $M(P_{\mathcal{K},\varphi}) \models prop$ holds, where $P_{\mathcal{K},\varphi}$ is constructed as indicated in Definition 4 above. Our technique consists of two steps. In the first step we transform the ω-program $P_{\mathcal{K},\varphi}$ into a *monadic* ω-program T such that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$. In the second step we check whether or not $M(T) \models prop$ holds by using a proof method for monadic ω-programs.

3.1 Transformation to Monadic ω-Programs

The first step of our model checking technique is realized by applying specialized versions of the following transformation rules: *definition introduction* and *elimination*, *instantiation*, *positive* and *negative unfolding*, *clause deletion*, *positive* and *negative folding* (see, for instance, [5,14,15]). These rules are applied according to a strategy which is a variant of the specialization strategy presented in [5].

Our specialization strategy starts off from the clause γ_1: $prop \leftarrow sat([s_0|X], \varphi)$ in $P_{\mathcal{K},\varphi}$ (see clause 1 in Definition 4) and a set of clauses, called *InDefs* which is initialized to $\{\gamma_1\}$. Then, our strategy iteratively applies two procedures: (i) the *instantiate-unfold* procedure, and (ii) the *define-fold* procedure. At each iteration,

the set *InDefs* is transformed into a set *Ds* of monadic ω-clauses, at the expense of possibly introducing some auxiliary, non-monadic clauses which are stored in the set *NewDefs*. These auxiliary clauses are given as input to a subsequent iteration of the strategy. The strategy terminates when no new auxiliary clauses are introduced and, when this happens, in a final step we apply the definition elimination rule by keeping only the clauses whose head predicate is either *prop* or a predicate on which *prop* depends.

The Specialization Strategy.
Input: An ω-program $P_{\mathcal{K},\varphi}$ for a Kripke structure $\mathcal{K}$ and a state formula φ.
Output: A monadic ω-program T such that $M(P_{\mathcal{K},\varphi}) \vDash \mathit{prop}$ iff $M(T) \vDash \mathit{prop}$.

$Q := P_{\mathcal{K},\varphi}$; $\quad \mathit{InDefs} := \{\mathit{prop} \leftarrow \mathit{sat}([s_0|X], \varphi)\}$; $\quad \mathit{Defs} := \mathit{InDefs}$;

while $\mathit{InDefs} \neq \emptyset$ **do**
  instantiate-unfold$(Q, \mathit{InDefs}, \mathit{Cs})$;
  define-fold$(\mathit{Cs}, \mathit{Defs}, \mathit{NewDefs}, \mathit{Ds})$;
  $Q := (Q - \mathit{InDefs}) \cup \mathit{NewDefs} \cup \mathit{Ds}$;
  $\mathit{InDefs} := \mathit{NewDefs}$; $\quad \mathit{Defs} := \mathit{Defs} \cup \mathit{NewDefs}$
od;

$T := \{\gamma \mid \gamma \in Q$ and the head predicate of γ is either *prop* or a predicate on which *prop* depends$\}$.

Let us now introduce two notions which are needed for presenting the *instantiate-unfold* and the *define-fold* procedures. A *definition clause* is a non-monadic ω-clause of the form $H \leftarrow A$ where: (1) H is an atom of the form p or $q(X)$, where q is a predicate of type `ilist`, (2) A is an atom, and (3) $\mathit{vars}(H) = \mathit{vars}(A)$. A *quasi-monadic clause* is an ω-clause of the form $H \leftarrow L_1 \wedge \ldots \wedge L_k$, with $k \geq 0$, such that: (i) H is an atom of the form p or $q([s|X])$, where p is a predicate of type `ilist` and $s \in \Sigma$, and (ii) for $i = 1, \ldots, k$, there exists a variable Y (possibly equal to X) of type `ilist` such that $\mathit{vars}(L_i) \subseteq \{Y\}$.

The *instantiate-unfold* procedure transforms a given set *InDefs* of definition clauses into a set *Cs* of quasi-monadic clauses by: (1) instantiating each clause in *InDefs*, (2) applying the positive (or negative) unfolding rule to clauses of the form $p([s|X]) \leftarrow B_L \wedge L \wedge B_R$, whenever L is a positive literal (or a negative literal, respectively), and (3) deleting subsumed clauses.

Given a clause δ, a variable X, and a term t, we denote by $\delta\{X/t\}$ the clause δ with every occurrence of X replaced by t.

The *instantiate-unfold* Procedure.
Input: An ω-program Q and a set $\mathit{InDefs} \subseteq Q$ of definition clauses.
Output: A set *Cs* of quasi-monadic clauses.

(*Instantiation*)
Let Y be a new variable of type `ilist` and let Σ be the set of states of $\mathcal{K}$;

$S := \{\, \delta\{X/[s|Y]\} \mid \delta \in \mathit{InDefs}$ and $\mathit{vars}(\delta) = \{X\}$ and $s \in \Sigma\} \cup$
  $\{\, \delta \mid \delta \in \mathit{InDefs}$ and $\mathit{vars}(\delta) = \emptyset\}$;

$Cs := \emptyset$;

(*Unfolding*)
while there exists a clause γ in S **do**

(*Case* 1. *Positive Unfolding*)
if (i) γ is of the form $H \leftarrow B_L \wedge A \wedge B_R$, where A is an atom,
(ii) $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$, and
(iii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i)
then $S := (S - \{\gamma\}) \cup \{H \leftarrow B_L \wedge B_1\vartheta_1 \wedge B_R, \ldots, H \leftarrow B_L \wedge B_m\vartheta_m \wedge B_R\}$

(*Case* 2. *Negative Unfolding*)
elseif (i) γ is of the form $H \leftarrow B_L \wedge \neg A \wedge B_R$, where A is an atom,
(ii) $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$,
(iii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i), and
(iv) for $i = 1, \ldots, m$, $vars(B_i) \subseteq vars(K_i)$
then from $B_L \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge B_R$ we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of conjunctions of literals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\vee$ outside;
$S := (S - \{\gamma\}) \cup \{H \leftarrow Q_1, \ldots, H \leftarrow Q_r\}$

(*Case* 3. *No Unfolding*)
else $S := S - \{\gamma\}$; $Cs := Cs \cup \{\gamma\}$ **fi**
od;

(*Subsumption*)
while there exists a unit clause γ_1 in Cs of the form $H \leftarrow$ and a variant of a clause γ_2 in $Cs - \{\gamma_1\}$ of the form $H \leftarrow B$ **do** $Cs := Cs - \{\gamma_2\}$ **od**

The *define-fold* procedure transforms the quasi-monadic ω-clauses of Cs into monadic ω-clauses by applying the definition introduction rule and the (positive or negative) folding rule. In particular, for any given quasi-monadic clause γ: $H \leftarrow L_1 \wedge \ldots \wedge L_k$ in Cs and for $i = 1, \ldots, m$, the *define-fold* procedure performs the following steps.

Let L_i be either the positive literal A_i or the negative literal $\neg A_i$. We consider the following two cases. Case (1): If in $Defs \cup NewDefs$ there is a clause δ_i of the form $K_i \leftarrow A_i$, then γ is folded using δ_i, that is, the occurrence of L_i in the body of γ is replaced either (i) by K_i, if $L_i = A_i$ (positive folding), or (ii) by $\neg K_i$, if $L_i = \neg A_i$ (negative folding). Case (2): Otherwise, if in $Defs \cup NewDefs$ there is no clause of the form $K_i \leftarrow A_i$, then the definition clause δ_i: $K_i \leftarrow A_i$, where K_i has a new predicate symbol $newp_i$, is added to $NewDefs$ (by applying the definition introduction rule). Then, clause γ is folded using the newly introduced clause δ_i as in Case (1).

The clause $H \leftarrow M_1 \wedge \ldots \wedge M_k$ derived by folding γ using clauses $\delta_1, \ldots, \delta_k$ is a monadic ω-clause. Indeed, we have that: (1) H is either of the form p or of the form $q([s|X])$ (because γ is quasi-monadic), (2) for $i = 1, \ldots, k$, M_i is either the atom K_i or the negated atom $\neg K_i$, where K_i is either of the form $newp_i$ or

of the form $newp_i(Y)$ (this follows from the definition of δ_i and the fact that γ is quasi-monadic), and (3) Condition (iii) of Definition 3 holds by defining ℓ as follows: let σ be the stratification function for the encoding program $P_{\mathcal{K},\varphi}$ (see Section 2.2), (i) $\ell(prop) = \sigma(prop) = |\varphi| + 1$, and (ii) for every predicate $newp_i$ that occurs in the head of a clause $K_i \leftarrow A_i$ introduced during any execution of the *define-fold* procedure, $\ell(newp_i) = \sigma(A'_i)$, where A'_i is any ground instance of A_i. For example, if we introduce the definition clause $newp_i(X) \leftarrow sat(X, e(u(a, b)))$, then we define $\ell(newp_i) = \sigma(sat(\pi, e(u(a, b)))) = |e(u(a, b)))| + 1 = 5$, where π is any infinite list. Note that ℓ does not depend on the particular instance of A_i, because the value of σ is independent of the infinite list which (possibly) occurs as an argument of A_i.

The *define-fold* Procedure.
Input: (i) A set *Cs* of quasi-monadic clauses and (ii) a set *Defs* of definition clauses;
Output: (i) A set *NewDefs* of definition clauses, and (ii) a set *Ds* of monadic ω-clauses.

NewDefs := $\emptyset$; *Ds* := $\emptyset$;

for each clause γ in *Cs* **do**
  let the clause γ be of the form $H \leftarrow L_1 \wedge \ldots \wedge L_k$;
  for $i = 1, \ldots, k$ **do**
    let L_i be either A_i or $\neg A_i$, for some atom A_i;
(*Definition Introduction*)
    if a clause δ with body A_i has a variant in *Defs* $\cup$ *NewDefs*
    then take K_i to be the head of δ
    else take K_i to be: (i) $newp_i(Y)$, if $vars(A_i) = \{Y\}$, and (ii) $newp_i$, if $vars(A_i) = \emptyset$, where $newp_i$ is a new predicate symbol;
      NewDefs := *NewDefs* $\cup \{K_i \leftarrow A_i\}$ **fi**;
(*Positive or Negative Folding*)
    if L_i is A_i **then** $M_i := K_i$ **else** $M_i := \neg K_i$ **fi**
  od; *Ds* := *Ds* $\cup \{H \leftarrow M_1 \wedge \ldots \wedge M_k\}$
od

The specialization strategy, which from the initial program $P_{\mathcal{K},\varphi}$ produces the final program T, is correct w.r.t. the perfect model semantics, in the sense that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$. This correctness result can be proved similarly to [5,14]. Note that the instantiation rule that we use in the *unfold* procedure, is not present in [5,14], but its application can be viewed as an unfolding of an additional atom $ilist(X)$ defined by the clauses: $ilist([s_0|Y]) \leftarrow, \ldots, ilist([s_h|Y]) \leftarrow$, where $\Sigma = \{s_0, \ldots, s_h\}$ is the set of states of $\mathcal{K}$.

Our specialization strategy terminates for every input program $P_{\mathcal{K},\varphi}$ because: (i) both the *instantiate-unfold* and *define-fold* procedures terminate, and (ii) the while loop of the strategy terminates.

The termination of the *instantiate-unfold* procedure is a consequence from the following properties. (1) The Instantiation and Subsumption steps terminate. (2) The predicates *path*, *tr*, and *elem* do not depend on themselves in program $P_{\mathcal{K},\varphi}$. (3) For each clause in $P_{\mathcal{K},\varphi}$ defining the predicate *notpath*, either the predicate of the body literal does not depend on *notpath* (see clause 11) or the term occurring in the body is a proper subterm of the term occurring in the head (see clause 12). (4) For each clause in $P_{\mathcal{K},\varphi}$ whose head is of the form $sat(l_1, \psi_1)$ and for each literal of the form $sat(l_2, \psi_2)$ occurring (positively or negatively) in the body of that clause, *either* ψ_2 is a proper subterm of ψ_1 *or* $\psi_1 = \psi_2$ and l_2 is a proper subterm of l_1. (5) For each state s and formula ψ, the literal $esists\_sat(s, \psi)$ depends on itself through a call to the predicate *sat* (see clauses 8 and 9) and by consuming at least one operator e in the formula ψ. (6) The applicability conditions given in the *instantiate-unfold* procedure (see Point (iii) of Case 1 and Case 2) do not allow the unfolding of a clause γ if this unfolding instantiates a variable in γ.

The termination of the *define-fold* procedure is straightforward.

Finally, the proof of termination of the while loop of the specialization strategy follows from the fact that only a finite number of definition clauses can be introduced by the *define-fold* procedure. Indeed, every definition clause is of the form $H \leftarrow A$, where: (i) A is an atom in the finite set $\Delta = \{notpath([s|X]) \mid s \in \Sigma\} \cup \{exists\_sat(s, \psi) \mid s \in \Sigma$ and ψ is a subformula of $\varphi\} \cup \{sat(X, \psi) \mid \psi$ is a subformula of $\varphi\}$, and (ii) for any $A \in \Delta$ the *define-fold* procedure introduces at most one definition clause.

Theorem 2 (Correctness and Termination of the Specialization Strategy). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and a state formula φ. The specialization strategy terminates for the input program $P_{\mathcal{K},\varphi}$ and returns an output program T such that: (i) T is a monadic ω-program and (ii) $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$.*

Example 3. Let us consider program $P_{\mathcal{K},\varphi}$ of Example 2. Our specialization strategy starts off from the sets $Q = P_{\mathcal{K},\varphi}$ and $InDefs = Defs = \{\gamma_1\}$, where γ_1 is the following definition clause (that is, clause 1 of $P_{\mathcal{K},\varphi}$):

γ_1: $prop \leftarrow sat([s_0|X], e(u(a, not(e(u(tt, not(u(tt, b)))))))))$

In the first execution of the loop body of our strategy we apply the *instantiate-unfold* procedure to the set *InDefs*. We get the set $Cs = \{\gamma_2, \gamma_3\}$ of quasi-monadic clauses, where:

γ_2: $prop \leftarrow \neg\, notpath([s_0|X]) \wedge sat(X, u(a, not(e(u(tt, not(u(tt, b)))))))$
γ_3: $prop \leftarrow \neg\, notpath([s_0|X]) \wedge \neg\, exists\_sat(s_0, u(tt, not(u(tt, b))))$

Then, by applying the *define-fold* procedure, we get the set $NewDefs = \{\gamma_4, \gamma_5, \gamma_6\}$ of definition clauses and the set $Ds = \{\gamma'_2, \gamma'_3\}$ of monadic ω-clauses, where:

γ_4: $p_1(X) \leftarrow notpath([s_0|X])$
γ_5: $p_2(X) \leftarrow sat(X, u(a, not(e(u(tt, not(u(tt, b)))))))$
γ_6: $p_3 \leftarrow exists\_sat(s_0, u(tt, not(u(tt, b))))$
γ'_2: $prop \leftarrow \neg\, p_1(X) \wedge p_2(X)$
γ'_3: $prop \leftarrow \neg\, p_1(X) \wedge \neg\, p_3$

At the end of the first execution of the body of the while loop of our strategy, we get: $Q = (P_{\mathcal{K},\varphi} - \{\gamma_1\}) \cup \{\gamma_2', \gamma_3'\}$, $InDefs = \{\gamma_4, \gamma_5, \gamma_6\}$, and $Defs = \{\gamma_1\} \cup \{\gamma_4, \gamma_5, \gamma_6\}$. Since $InDefs \neq \emptyset$ the execution of the while loop continues. After a few more executions of the loop body, the *define-fold* procedure does not introduce any new clause in *NewDefs*. Thus, we get $InDefs = \emptyset$ and we derive the final program Q. By keeping every clause in Q whose head predicate is either *prop* or a predicate on which *prop* depends, we get the following monadic ω-program T:

$prop \leftarrow \neg\, p_1(X) \wedge p_2(X)$
$prop \leftarrow \neg\, p_1(X) \wedge \neg\, p_3$
$p_1([s_0|X]) \leftarrow p_1(X)$
$p_1([s_1|X]) \leftarrow p_4(X)$
$p_1([s_2|X]) \leftarrow$
$p_2([s_0|X]) \leftarrow \neg\, p_3$
$p_2([s_0|X]) \leftarrow p_2(X)$
$p_2([s_1|X]) \leftarrow \neg\, p_5$
$p_2([s_2|X]) \leftarrow \neg\, p_6$
$p_2([s_2|X]) \leftarrow p_2(X)$
$p_3 \leftarrow \neg\, p_1(X) \wedge \neg\, p_7(X)$
$p_3 \leftarrow \neg\, p_1(X) \wedge p_8(X)$
$p_4([s_0|X]) \leftarrow$
$p_4([s_1|X]) \leftarrow p_4(X)$
$p_4([s_2|X]) \leftarrow p_9(X)$
$p_5 \leftarrow \neg\, p_4(X) \wedge p_8(X)$
$p_6 \leftarrow \neg\, p_9(X) \wedge \neg\, p_7(X)$
$p_6 \leftarrow \neg\, p_9(X) \wedge p_8(X)$
$p_7([s_0|X]) \leftarrow p_7(X)$
$p_7([s_1|X]) \leftarrow$
$p_7([s_2|X]) \leftarrow p_7(X)$
$p_8([s_0|X]) \leftarrow \neg\, p_7(X)$
$p_8([s_0|X]) \leftarrow p_8(X)$
$p_8([s_1|X]) \leftarrow p_8(X)$
$p_8([s_2|X]) \leftarrow \neg\, p_7(X)$
$p_8([s_2|X]) \leftarrow p_8(X)$
$p_9([s_0|X]) \leftarrow$
$p_9([s_1|X]) \leftarrow p_4(X)$
$p_9([s_2|X]) \leftarrow$

3.2 A Proof Method for Monadic ω-Programs

In this section we present the second step of our model checking technique. In particular, we present a method for checking whether or not $M(P) \models F$ holds, for any monadic ω-program P and any formula F which is either of the form p or of the form $\exists\, X(L_1 \wedge \ldots \wedge L_n)$, with $n \geq 1$, where, for $i = 1, \ldots, n$, L_i is either a positive literal $q_i(X)$ or a negative literal $\neg q_i(X)$. In what follows the set of the formulas F of this form will be denoted by $\mathcal{F}$. In particular, our method allows us to check whether or not $M(T) \models prop$ holds for the monadic ω-program T that we derive by the specialization strategy presented in Section 3.1.

First, we introduce the notion of a *derivation tree* and, then, the notion of a *proof* of a formula F in $\mathcal{F}$ w.r.t. a monadic ω-program P. Every node of a derivation tree has: (i) a *depth* which is the number of its ancestor nodes (in particular, the root has depth 0), and (ii) a *label* which is either

(1) the empty conjunction *true*, or
(2) the empty disjunction *false*, or
(3) a literal of the form: either p, or $\neg\, p$, or $q(X)$, or $\neg\, q(X)$, or
(4) a formula of the form: either $\exists X\, (L_1 \wedge \ldots \wedge L_n)$ or $\neg\, \exists X\, (L_1 \wedge \ldots \wedge L_n)$, with $n \geq 1$, where, for $i = 1, \ldots, n$, L_i is either $q_i(X)$ or $\neg\, q_i(X)$.

We denote by $\mathcal{L}$ the set of formulas of the forms (3) and (4). Let us also introduce the following notation: (i) for any atom A, $\overline{A}$ denotes $\neg A$ and $\overline{\neg A}$ denotes A, and (ii) for any formula B, $\overline{\exists X\, B}$ denotes $\neg \exists X\, B$.

In order to construct a derivation tree of a formula in $\mathcal{F}$ w.r.t. a given monadic ω-program P, we begin by rewriting the program P as follows. (Recall that in the body of a monadic ω-clause at most one variable occurs in a literal and two distinct literals may have a variable in common.) For every clause $H \leftarrow B$ in P and for every variable Y in $vars(B) - vars(H)$, we replace the literals $L_1, \ldots, L_m$

of B such that $vars(L_1) = \ldots = vars(L_m) = \{Y\}$ by the formula $\exists Y\,(L_1 \wedge \ldots \wedge L_m)$. Thus, every clause in P is rewritten as $H \leftarrow F_1 \wedge \ldots \wedge F_k$, where, for $i = 1, \ldots, k$, F_i is a formula in $\mathcal{L}$.

For instance, clause $q_0([s|X]) \leftarrow q_1(X) \wedge q_2(Y) \wedge p_1 \wedge \neg\, q_3(Y) \wedge p_2$ is rewritten as $q_0([s|X]) \leftarrow q_1(X) \wedge \exists Y\,(q_2(Y) \wedge \neg\, q_3(Y)) \wedge p_1 \wedge p_2$.

Definition 5 (Derivation Tree). Given a monadic ω-program P and a formula F in $\mathcal{F}$, a *derivation tree* of F w.r.t. P is a *finite* tree T constructed as follows:

1. the root node is labeled by F, and if F is of the form $\exists X\,(L_1 \wedge \ldots \wedge L_n)$ then the root node has n children labeled by $L_1, \ldots, L_n$, respectively,
2. if a non-root node N is labeled by: (i) *true*, or (ii) *false*, or (iii) $\exists X\, B$, or (iv) $\neg\, \exists X\, B$ (that is, N is not labeled by a literal), then N is a leaf,
3. for every integer $d \geq 0$, consider the nodes $N_1, \ldots, N_\ell$, with $\ell \geq 1$, of depth d:
   if there exists an integer c, with $0 \leq c < d$, such that for every literal L labeling a node of depth d, there exists a node of depth c labeled by L
   then the nodes $N_1, \ldots, N_\ell$ are leaves
   else choose a state $s \in \Sigma$ and, for $i = 1, \ldots, \ell$, if the node N_i is labeled by a literal L_i, then construct a child node of N_i with label F, for each formula F in the set $\mathcal{C}_i$ of formulas in $\mathcal{L} \cup \{true, false\}$ constructed from the state s, the literal L_i, and the program P, as we now indicate. There are two cases.
   Case (i): L_i is an atom $q(X)$ (or p). If in P there is no clause whose head is $q([s|X])$ (or p), then take $\mathcal{C}_i$ to be $\{false\}$. Otherwise, choose a clause $q([s|X]) \leftarrow F_1 \wedge \ldots \wedge F_k$ (or $p \leftarrow F_1 \wedge \ldots \wedge F_k$) in P, where, for $i = 1, \ldots, k$, $F_i \in \mathcal{L}$. If $k = 0$ then take $\mathcal{C}_i$ to be $\{true\}$, else take $\mathcal{C}_i$ to be $\{F_1, \ldots, F_k\}$.
   Case (ii): L_i is a negated atom $\neg q(X)$ (or $\neg p$). Let $q([s|X]) \leftarrow B_1, \ldots, q([s|X]) \leftarrow B_k$ (or $p \leftarrow B_1, \ldots, p \leftarrow B_k$) be all clauses in P whose head is $q([s|X])$ (or p). If $k = 0$ then take $\mathcal{C}_i$ to be $\{true\}$. If $k \geq 1$ and there exists i, with $1 \leq i \leq k$, such that B_i is the empty conjunction, then take $\mathcal{C}_i$ to be $\{false\}$. Otherwise, for $i = 1, \ldots, k$, choose a formula $F_i \in \mathcal{L}$ such that $B_i = G_1 \wedge F_i \wedge G_2$, where G_1 and G_2 are (possibly empty) conjunctions, and take $\mathcal{C}_i$ to be $\{\overline{F}_1, \ldots, \overline{F}_k\}$.

By construction, for any derivation tree T there exist: (i) an integer m which is the maximal depth of a node of T, and (ii) a *least* integer c, with $0 \leq c < m$, such that for every literal L labeling a node of depth m, there exists a node of depth c labeled by L. Now, we introduce a relation r_T between literals as follows. For any two literals L_1 and L_2, $r_T(L_1, L_2)$ holds iff: (i) there exists a node M of depth c in T whose label is L_1, (ii) there exists a node N of depth m in T whose label is L_2, and (iii) M is an ancestor of N in T. We denote by r_T^+ the transitive closure of r_T.

Proposition 1. *Let P be a monadic ω-program and F be a formula in $\mathcal{F}$. (i) Every derivation tree T of F w.r.t. P is minimal, in the sense that no proper subtree of T is itself a derivation tree of F w.r.t. P. (ii) There exists a finite number of derivation trees of F w.r.t. P.*

Now we present the definitions of proof and refutation, which are based on the notion of derivation tree.

Definition 6 (Proof and Refutation). Let P be a monadic ω-program and F be a formula in $\mathcal{F}$. We say that F has a *proof* w.r.t. P iff there exists a derivation tree T of F w.r.t. P which satisfies the following conditions:

1. every leaf N of T is labeled by: either (i) *true*, or (ii) a literal of the form p, or $\neg p$, or $q(X)$, or $\neg q(X)$, or (iii) a formula of the form $\exists X\, B$ that has a proof w.r.t. P, or (iv) a formula of the form $\neg\, \exists X\, B$ such that $\exists X\, B$ has a refutation w.r.t. P,
2. for every positive literal L labeling a leaf of T, $r_T^+(L, L)$ does not hold.

We say that F has a *refutation* w.r.t. P iff no derivation tree of F w.r.t. P is a proof of F w.r.t. P.

By Proposition 1 it is decidable whether or not a there exists a proof of a formula in $\mathcal{F}$ w.r.t. a monadic ω-program. Moreover, by induction on the level of the predicates occurring in the monadic ω-program P, we can show that our proof method is sound and complete for showing that a formula in the set $\mathcal{F}$ is true in the perfect model of P. Thus, we have the following result.

Theorem 3. *Let P be a monadic ω-program and F a formula in $\mathcal{F}$. Then: (i) there is an algorithm to check whether or not F has a proof w.r.t. P, and (ii) F has a proof w.r.t. P iff $M(P) \models F$.*

Now we present an example of application of the second step of our transformational method for proving CTL* properties of the Kripke structures which encode reactive systems.

Example 4. Let us consider: (i) the monadic ω-program T, obtained as the output of our specialization strategy (see Example 3), and (ii) the formula *prop*, that encodes the CTL* property φ of the Kripke structure $\mathcal{K}$ introduced in Example 2. We can construct a proof for the formula *prop* w.r.t. T as shown by the various derivation trees depicted in Figure 1. As a consequence, we have that $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$ holds and, thus, the formula φ holds in the Kripke structure $\mathcal{K}$.

4 Related Work and Concluding Remarks

Various logic programming techniques and tools have been developed for model checking. In particular, tabled resolution has been shown to be quite effective for implementing a modal μ-calculus model checker for CCS value passing programs [13]. Techniques based on logic programming, constraint solving, abstract interpretation, and program transformation have been proposed for performing CTL model checking of finite and infinite state systems (see, for instance, [3,5,8,10]). In this paper we have extended to CTL* model checking the transformational approach which was proposed for LTL model checking in [11].

The main contributions of this work are the following. (i) We have proposed a method for specifying CTL* properties of reactive systems based on ω-programs,

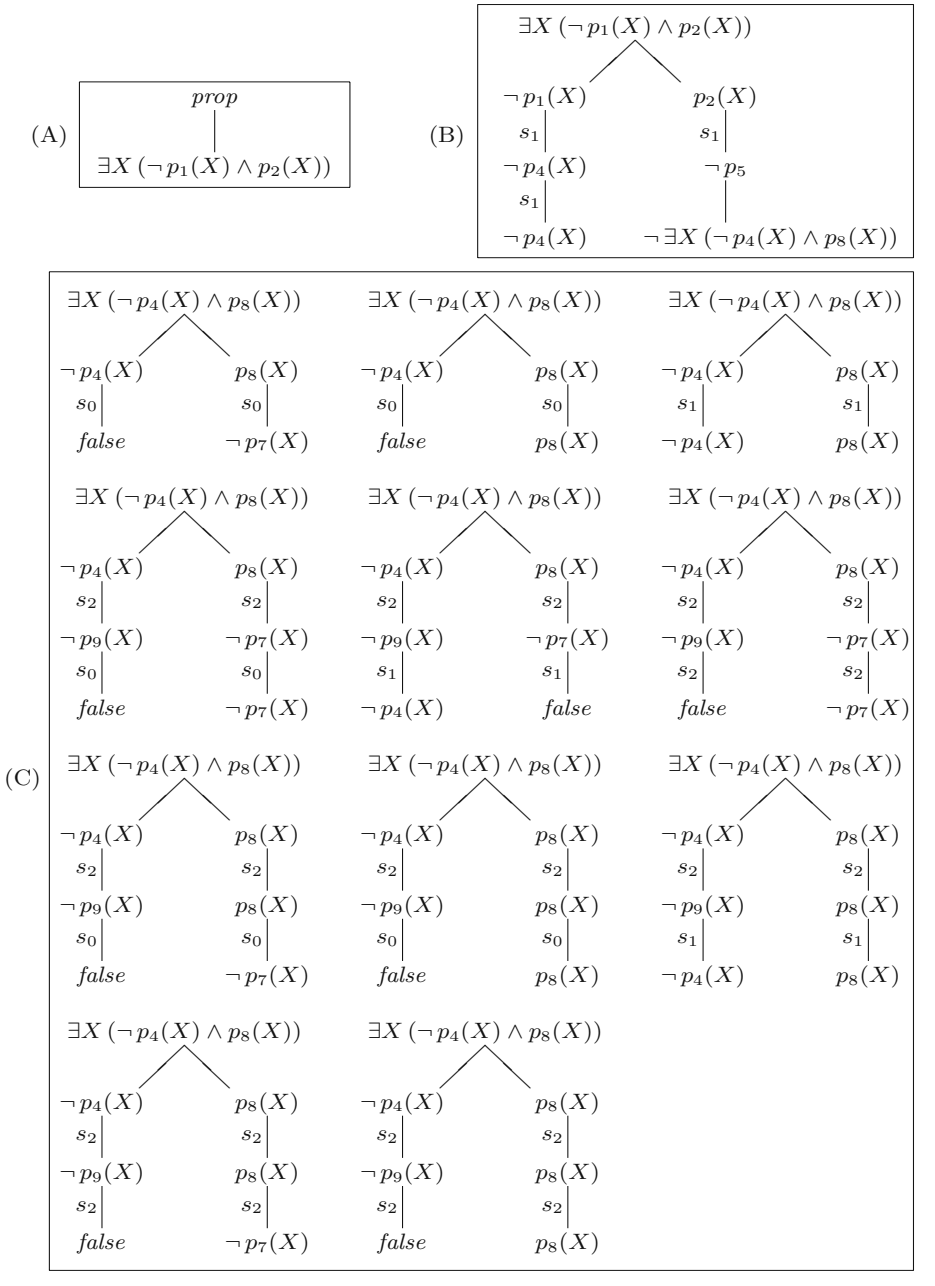


Fig. 1. The tree in (A) is a proof of *prop* w.r.t. T. The tree in (B) is a proof of $\exists X\,(\neg p_1(X) \wedge p_2(X))$ w.r.t. T. The 11 trees in (C) are all the derivation trees of $\exists X\,(\neg p_4(X) \wedge p_8(X))$ w.r.t. T and none of them is a proof. The labels of the arcs are the states $s \in \Sigma$ to be chosen according to Point (3) of Definition 5.

that is, logic programs acting on infinite lists. This method is a proper extension of the methods for specifying CTL or LTL properties, because CTL and LTL are fragments of CTL*. (ii) We have introduced the subclass of monadic ω-programs for which the satisfaction relation w.r.t. the perfect model is decidable. This subclass of programs properly extends the class of *linear monadic* ω-programs introduced in [11]. (iii) Finally, we have shown that we can transform, by applying semantics preserving unfold/fold rules, the logic programming specification of a CTL* property into a monadic ω-program.

Our transformation strategy can be viewed as a specialization of the Encoding Program (see Definition 4) w.r.t. a given Kripke structure $\mathcal{K}$ and a given CTL* formula φ. However, it should be noted that this program specialization could not be achieved by using partial deduction techniques (see [7] for a brief survey). Indeed, our specialization strategy performs instantiation and negative unfolding steps that cannot be realized by partial deduction.

Our two step verification approach bears some similarities with the automata-theoretic approach to CTL* model checking, where the specification of a finite state system and a CTL* formula are translated into alternating tree automata [6]. The automata-theoretic approach is quite appealing because many useful techniques are available in the field of automata theory. However, we believe that also our approach has its advantages because of the following reasons. (1) The specification of properties of reactive systems, together with the transformation of these specifications into monadic ω-programs, and the proofs of properties of monadic ω-programs, can all be done within the single framework of logic programming, while in the automata-theoretic approach one has to translate the temporal logic formalism into the distinct formalism of automata theory. (2) The translation of a specification into a monadic ω-program can be performed by using semantics preserving transformation rules, thereby avoiding the burden of proving the correctness of the translation by *ad-hoc* methods. (3) Finally, due its generality, we believe that our approach can be extended without much effort to the case of infinite state systems.

Issues that can be investigated in the future include: (i) the complexity of our verification method and, in particular, an efficient implementation of the proof method presented in Section 3.2, (ii) the relationship between monadic ω-programs and alternating tree automata, (iii) the applicability of our transformational approach to other logics, such as the monadic second order logic of successors, and (iv) the experimental evaluation of the efficiency of our transformational approach by considering various test cases and comparing its performance in practical examples w.r.t. that of other model checking techniques known in the literature.

References

1. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. Journal of Logic Programming 19, 20, 9–71 (1994)
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

3. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. International Journal on Software Tools for Technology Transfer 3(3), 250–270 (2001)
4. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. Journal of the ACM 33(1), 151–178 (1986)
5. Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying CTL properties of infinite state systems by specializing constraint logic programs. In: Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL 2001, Florence (Italy), Technical Report DSSE-TR-2001-3, pp. 85–96. University of Southampton, UK (2001)
6. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)
7. Leuschel, M.: Logic program specialisation. In: Hatcliff, J., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 155–188. Springer, Heidelberg (1999)
8. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialization. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 63–82. Springer, Heidelberg (2000)
9. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
10. Nilsson, U., Lübcke, J.: Constraint logic programming for local and symbolic model-checking. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 384–398. Springer, Heidelberg (2000)
11. Pettorossi, A., Proietti, M., Senni, V.: Transformational verification of linear temporal logic. In: 24th Italian Conference on Computational Logic Ferrara, Italy (CILC 2009), June 24-26 (2009), http://www.ing.unife.it/eventi/cilc09
12. Przymusinski, T.C.: On the declarative semantics of stratified deductive databases and logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 193–216. Morgan Kaufmann, San Francisco (1988)
13. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
14. Seki, H.: Unfold/fold transformation of stratified programs. Theoretical Computer Science 86, 107–139 (1991)
15. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: Tärnlund, S.-Å. (ed.) Proceedings of the Second International Conference on Logic Programming (ICLP 1984), pp. 127–138. Uppsala University, Uppsala (1984)

A Transformational Approach for Proving Properties of the CHR Constraint Store

Paolo Pilozzi*, Tom Schrijvers**, and Maurice Bruynooghe

Department of Computer Science, K.U. Leuven, Belgium
{Paolo.Pilozzi,Tom.Schrijvers,Maurice.Bruynooghe}@cs.kuleuven.be

Abstract. Proving termination of, or generating efficient control for Constraint Handling Rules (CHR) programs requires information about the kinds of constraints that can show up in the CHR constraint store. In contrast to Logic Programming (LP), there are not many tools available for deriving such information for CHR. Hence, instead of building analyses for CHR from scratch, we define a transformation from CHR to Prolog and reuse existing analysis tools for Prolog.

The proposed transformation has been implemented and combined with PolyTypes 1.3, a type analyser for Prolog, resulting in an accurate description of the types of CHR programs. Moreover, the transformation is not limited to type analysis. It can also be used to prove other properties of the constraints showing up in constraint stores, using tools for Prolog.

Keywords: Constraint Handling Rules, Program Transformation.

1 Introduction

Proving termination of, or generating efficient control for Constraint Handling Rules (CHR) programs requires information about the kinds of constraints that can show up in the CHR constraint store. In particular, type information is useful in this context. When used as a basis for determining the possible calls to the program, it leads to compiler optimisations [11], more precise termination conditions [4,7] and more refined interpretations for proving termination [1,9].

In Logic Programming (LP), many tools are available for performing such analyses [5,2,12]. Hence, instead of building analyses for CHR from scratch, it is interesting to explore whether one can define transformations from CHR to Prolog and reuse existing analysis tools for Prolog to obtain properties about the constraints that are in the CHR constraint store during computations.

One approach would be to build a faithful CHR meta-interpreter in Prolog and to analyse this meta-interpreter or to transform the CHR program into a Prolog meta-program and to analyse the meta-program. A difficulty with this approach is capturing the "fire-once" policy of CHR which prescribes that a rule cannot be applied twice to the same combination of constraints. This policy

* Supported by I.W.T. - Flanders (Belgium).
** Post-Doctoral Researcher of F.W.O. - Flanders (Belgium).

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 22–36, 2010.

prevents the infinite application of propagation rules, that add constraints to the store without removing any. The approach in [10] has a problem with this.

Fortunately, it often suffices to have an over-approximation of the constraints that can show up in the constraint store. In that case, one does not need a meta-interpreter or transformation that rigorously preserves the run-time behaviour of the CHR program and one can simply ignore the "fire-once" policy. This sometimes results in the presence of constraints in the approximated store that cannot be present at run-time, e.g., because some rule needs different occurrences of the same constraint before it can fire. But this is not too much of a problem, if only because one is typically interested in a whole class of queries (initial constraint stores), and queries in the class can have multiple occurrences of constraints, hence rules that need multiple occurrences can fire anyway.

For CHR, some direct approaches were developed [3, 11], mainly based on approaches developed for LP. Direct approaches usually make use of abstract interpretation. For CHR, not much work has been done on the topic of abstract interpretation [11] and thus not many analyses resulted from it. The transformational approach hasn't received much attention either. To the best of our knowledge, except for the termination preserving transformation discussed in [10], no transformational approaches have been attempted.

We have implemented the transformation and combined it with PolyTypes 1.3 [2] in a tool called CHRTypes. We plan to use CHRTypes as a source of information to obtain the call types of a CHR(Prolog) program. The computed call types can then be used as input to our termination analyser CHRisTA [8], instead of providing them ourselves, resulting in a fully automated termination analyser for CHR(Prolog).

The paper is organised as follows. In the next section we introduce CHR syntax and the abstract CHR semantics. Then, in Section 3, we discuss a transformation of CHR(Prolog) to Prolog. Section 4, discusses the application of our transformation to type analysis of CHR(Prolog), using PolyTypes 1.3 (based on [2]) on the transformed programs. Then, in Section 5, we evaluate our transformational approach using CHRTypes, a fully automated type analyser for CHR(Prolog). Finally, in Section 6, we conclude the paper.

2 Preliminaries

2.1 CHR Syntax

CHR is intended as a programming language for implementing constraint solvers. To implement these solvers, a user can define *CHR rules* which rewrite conjunctions of *constraints*. The constraints of a CHR program are special first-order predicates $c(t_1, \ldots, t_n)$ on terms, like the atoms of an LP program. There are two kinds of constraints defined in a CHR program: *CHR constraints* are user-defined and solved by the CHR program. *Built-in constraints* are pre-defined and solved by an underlying constraint theory, CT, defined in the host language. We consider Prolog, thus definite LP with a left-to-right selection rule, as host language. We assume the reader to be familiar with Prolog syntax and semantics.

A *CHR program*, P, is a finite set of *CHR rules*, defining the transitions of the program. To provide the analyser with information about the built-ins one can add some Prolog clauses that capture their essential properties. In CHR, there are three different kinds of rules. *Simplification rules* replace CHR constraints by new CHR and built-in constraints. On the presence of CHR constraints, *propagation rules* only add new constraints. Finally, *simpagation rules* replace CHR constraints by new constraints, given the presence of other CHR constraints.

Let H_k, H_r and C denote conjunctions of CHR constraints and let G and B denote conjunctions of built-in constraints. Then, a simplification rule takes the form, $R \mathbin{@} H_r \Leftrightarrow G \mid B, C$, a propagation rule the form, $R \mathbin{@} H_k \Rightarrow G \mid B, C$, and a simpagation rule the form, $R \mathbin{@} H_k \setminus H_r \Leftrightarrow G \mid B, C$. Like in Prolog syntax, we write a conjunction of constraints as a sequence of conjuncts separated by commas. Rules are named by adding "*rulename* @" in front of the rule.

Example 1 (Merge-sort). The program below implements the merge-sort algorithm. The query $mergesort(L)$, with L a list of natural numbers of length exactly 2^n, yields a tree-representation of the order, which then is rewritten into a sorted list of elements. Note that in this version of merge-sort we represent the natural numbers using a symbolic form: $0, s(0), s(s(0)), \ldots$.

$$
\begin{aligned}
&R_1 \mathbin{@} msort([]) \Leftrightarrow true.\\
&R_2 \mathbin{@} msort([L|Ls]) \Leftrightarrow r(0, L), msort(Ls).\\
&R_3 \mathbin{@} r(D, L1), r(D, L2) \Leftrightarrow leq(L1, L2) \mid r(s(D), L1), a(L1, L2).\\
&R_4 \mathbin{@} a(L1, L2) \setminus a(L1, L3) \Leftrightarrow leq(L2, L3) \mid a(L2, L3).
\end{aligned}
$$

The first two rules decompose a list of elements, while adding new $r/2$ constraints to the store. The constraints $r(D, L)$ represent trees of depth D (initially 0) and root value L. The third and fourth rule perform the actual merge-sorting. The third rule joins two trees of equal depth. It replaces both trees by a new tree of incremented depth, where the largest root becomes a child node of the smallest hence the branch is ordered. Note that the initial list needs to have a length that is a power of 2 to ensure that one ends with a single tree. The order in a branch is represented by $a/2$ constraints. Finally, the fourth rule merge-sorts different branches of a tree into a single branch, i.e., an ordered list of elements. □

2.2 The Abstract CHR Semantics

In general, CHR is defined as a state transition system. In its simplest form, called the *abstract semantics*, it defines a state as a conjunction, or alternatively a multi-set, of constraints, called the *constraint store*.

Definition 1 (CHR state). *A CHR state S is a conjunction or multi-set of built-in and CHR constraints. An* initial state *or* query *is a finite conjunction of constraints. In a* final state *or* answer*, either the built-in constraints are inconsistent (*failed state*) or no more transitions are possible.* □

The rules of a CHR program determine the possible transitions between constraint stores. Since the abstract semantics ignores the fire-once policy, we have

that all three kinds of rules are essentially simplification rules. Consider for example the propagation rule, $R \ @ \ H_k \Rightarrow B, C$. Given the abstract CHR semantics, it is equivalent to the simplification rule, $R \ @ \ H_k \Leftrightarrow H_k, B, C$. Similarly, a simpagation rule, $R \ @ \ H_k \setminus H_r \Leftrightarrow B, C$, can be represented as a simplification rule, $R \ @ \ H_k, H_r \Leftrightarrow H_k, B, C$.

The transition relation relates consecutive CHR states on the presence of applicable CHR rules. The built-ins are non-deterministically solved by the CT.

Definition 2 (Transition relation). *Let θ denote a substitution corresponding to the bindings generated when resolving built-in constraints. Let σ denote a matching substitution of the variables in the head and an answer substitution of the variables appearing in the guard but not in the head. The* transition relation*, $\rightarrow$, between CHR states, given a constraint theory CT for the built-ins and a CHR program P for the CHR constraints, is defined as follows.*

1. ***Solve transition**:*
   ***if** $S = b \wedge S'$ and $CT \models b\theta$ **then** $S \rightarrow S'\theta$*
2. ***Simplification**:*
   ***given** a fresh variant of a rule in P: $H_r \Leftrightarrow G \mid B, C$*
   ***if** $S = H'_r \wedge S'$ and $CT \models (H'_r = H_r\sigma) \wedge G\sigma$ **then** $S \rightarrow (B \wedge C \wedge S')\sigma$*

We assume built-ins not to introduce new CHR constraints and thus solving these can only generate binding for variables. If built-in constraints cannot be solved by the CT, the CHR program fails. □

Note that by adding variable bindings to the constraint store (solving built-ins), a guard can become true. Also note that the selection of an answer substitution for the local variables in the guard is a committed choice. To denote the host language CT we write CHR(CT), e.g. CHR(Prolog).

3 Transforming CHR(Prolog) to Prolog

In Section 2.2, we discussed the representation of the three kinds of CHR rules into simplification rules, thus safely over-approximating the contents of the constraint store with respect to the original theoretical CHR semantics. This choice was motivated in the introduction. We assume this transformation to take place prior to the transformation to Prolog that we discuss in this section.

3.1 Representing the CHR Constraint Store in Prolog

The CHR constraint store is a conjunction of constraints. To represent it in Prolog, we fix some order and represent it as a list, called the *storelist*. The code handling the firing of a rule will cope with the fact that the storelist is equivalent to any of its permutations. That there are $n!$ permutations for an n-element store is of no concern as the transformed program will be analysed, not executed.

Thus, for a constraint store $S = constr_1 \wedge constr_2 \wedge \ldots \wedge constr_n$, we obtain as a possible storelist representation $R = [constr_1, constr_2, \ldots, constr_n]$. Note that

according to the abstract CHR semantics, a CHR query is an initial constraint store. Its representation by a storelist in Prolog is therefore identical to that of any other constraint store.

3.2 Representing CHR Rules in Prolog

A CHR rule defines transitions between constraint stores. Which transitions are applicable for a constraint store is determined by the presence of matching constraints for the heads of rules such that the guards of these rules are entailed. Multiple rules can be simultaneously applicable, in which case CHR commits to a particular choice. The following example illustrates this.

Example 2 (Non-determinism). Consider an initial store $a \wedge b$ for the program:

$$R_1 \ @ \ a \Leftrightarrow c. \qquad R_2 \ @ \ b \Leftrightarrow d. \qquad R_3 \ @ \ a, d \Leftrightarrow a, a, b.$$

The program may or may not end for the initial constraint store. If we apply the first rule, then the program terminates immediately. If we apply the second and third rule repeatedly, then the program runs forever. □

To model possible constraint stores that can exist during execution of a CHR program, it suffices to represent the non-determinism of CHR by search in Prolog. This is achieved by transforming every CHR rule to a Prolog clause of the $rule/2$ predicate. The clause describes the relationship between the store before and after rule application. To perform the matching between the store and the head of the rule, it is checked whether the storelist starts with the constraints in the rule head. Thus, a CHR rule of the form:

$$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_l, C_1, \ldots, C_m$$

becomes a Prolog clause:

$$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \ \texttt{:-} \ G_1, \ldots, G_k.$$

Here, $H_1, \ldots, H_n$ are head constraints. Built-in guards and bodies are represented respectively by $G_1, \ldots, G_k$ and $B_1, \ldots, B_l$. The CHR body constraints are represented by $C_1, \ldots, C_m$. Note that the head of the CHR rule is represented as a list with a variable as tail. This tail binds with the unused constraints in the current store. When the guards succeed, the new store consists of these unused constraints extended with the new constraints from the body.

As the CHR(Prolog) program has no rules for the built-in predicates, we need to add to the translation, rules that process them. For each built-in predicate p/n, there is therefore a clause $rule([p(X_1, \ldots, X_n)|R], R) \ \texttt{:-} \ p(X_1, \ldots, X_n)$ present in the transformed CHR(Prolog) program.

3.3 Representing the Abstract Semantics of CHR in Prolog

The operational semantics of CHR programs is already largely represented by the rule clauses. Matching of constraints in the store with heads of the rules is

done by unification with the storelist. The resulting store is contained in the second argument of the rule clause. We only have to call rules repeatedly.

$$goal(S) \text{ :- } perm(S, PS), rule(PS, NS), goal(NS). \qquad goal(\_).$$

Note that we must permute the storelist – the call $perm(P, PS)$ – to bring the matching constraints to the front. Also note that whenever the program cannot call any of the $rule/2$ clauses, it will end up in a refutation, representing termination in CHR. In fact, any call to $goal/1$ can result in a refutation. Nevertheless, no further approximations of the contents of the CHR constraint store result from this. Finally, notice that CHR queries are represented by a call to $goal/1$ with a storelist representation of the CHR query as argument.

Example 3 (Transforming merge-sort). We revisit merge-sort from Example 1 and transform every rule into its clausal form. First, we represent all rules by simplification rules. This is already the case for the first three rules. The fourth rule on the other hand is a simpagation rule and is transformed into

$$R_4 \ @ \ a(L1, L2), a(L1, L3) \Leftrightarrow leq(L2, L3) \mid a(L1, L2), a(L2, L3).$$

Next, the CHR program is transformed into the following Prolog program.

$goal(S)$:- $perm(S, PS), rule(PS, NS), goal(NS)$.
$goal(\_)$.

$rule([msort([])|R], R)$.
$rule([msort([L|List])|R], [r(0, L), msort(List)|R])$.
$rule([r(D, L1), r(D, L2)|R], [r(s(D), L1), a(L1, L2)|R])$:- $leq(L1, L2)$.
$rule([a(L1, L2), a(L1, L3)|R], [a(L1, L2), a(L2, L3)|R]$:- $leq(L2, L3)$.

A query for the transformed program is of the form $goal([msort(L)])$, where L is a list of natural numbers in symbolic form, as in Example 1. □

3.4 Transformation Summary

To transform CHR states to Prolog queries, we introduce a mapping, $\alpha : S \to Q$, from a constraint store, S, to a Prolog query, Q, of the form $goal(R)$. Here, R is the storelist representation of S, as defined in Subsection 3.1. We define also the inverse of α as $\gamma = \alpha^{-1}$.

We introduce an operator, $C2P$, transforming a CHR program, P, to a Prolog program, $\wp$, and define it as follows.

Definition 3 ($C2P$). *A CHR program P is transformed into the following Prolog program $\wp = C2P(P)$.*

- *The Prolog program $\wp$ contains following clauses:*

```
goal(S) :-                perm(L, [X|P]) :-        del(X, [Y|T], [Y|R]) :-
    perm(S, PS),              del(X, L, L1),            del(X, T, R).
    rule(PS, NS),             perm(L1, P).          del(X, [X|T], T).
    goal(NS).             perm([], []).
goal(_).
```

- *The Prolog program $\wp$ contains for every rule,*
  $$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_l, C_1, \ldots, C_m.$$
  in P, where $B_1, \ldots, B_l$ are added built-in constraints and $C_1, \ldots, C_m$ are added CHR constraints, the following clause:
  $$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \text{ :- } G_1, \ldots, G_k.$$
- *The Prolog program $\wp$ contains for every built-in predicate p/n in P, a clause:*
  $$rule([p(X_1, \ldots, X_n)|R], R) \text{ :- } p(X_1, \ldots, X_n).$$ □

We connect the CHR program, P, and its corresponding Prolog program, $\wp$, using α. We show that if a transition exists between two CHR states S and S', then there must exist a corresponding derivation in the transformed program. This derivation, however, is, in contrast to the CHR transition, no single-step operation. Between a call to $goal/1$ and a next call to $goal/1$, one needs to resolve the calls to $perm/2$ and $rule/2$, implementing the CHR transition. This property is illustrated in the diagram of Figure 1.

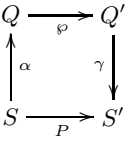


Fig. 1. Connection $\wp$ and P. Here, Q and Q' are Prolog queries and S and S' CHR states. The vertical arrows represent mappings of α and γ. The horizontal arrows represent CHR transitions in P and derivations in $\wp$.

Theorem 1 (Connecting $\wp$ and P). *Consider CHR(Prolog). Let S be a constraint store and let Q be a goal statement such that $Q = \alpha(S)$. Let S' be the constraint store that is the result of applying a CHR transition on S, given by a CHR program P. Then, there exists a partial LD-derivation in $\wp = C2P(P)$ (SLD with Prolog's left to right selection rule) that starts in Q and leads to a goal Q' with the property that $S' = \gamma(Q')$.* □

Proof. We need to show that for every transition between consecutive states S and S' in the CHR program P, as given by the abstract CHR semantics, there exists a corresponding partial LD-derivation for the definite program $C2P(P)$ and the goal $Q = \alpha(S)$ that leads to some Q' with the property that $\gamma(Q') = S'$.

According to Definition 2, we can distinguish between two kind of transitions. We proof the property for each of them. In the proof, we use multi-sets to represent the constraint store, and use set-notation to denote them.

Solve transition. In this case, the constraint store S holds a constraint b that refers to a built-in Prolog predicate b/n and that can be successfully solved by the underlying Prolog system. More precisely, the constraint store contains

$$\{b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_k\}$$

for some $k \geq 0$. Solving $b(X_1, \ldots, X_n)\sigma$ results in the application of a substitution θ and the resulting constraint store S' equals $\{c_1, \ldots, c_k\}\theta$. The goal

Q corresponding to the initial constraint store S is of the form

$$goal([a_1, \ldots, a_{k+1}])$$

with $[a_1, \ldots, a_{k+1}]$ a permutation of $[b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_k]$. When performing a resolution step on this goal with the program $C2P(P)$, either the fact $goal(\_)$ is applied, yielding a refutation, or the clause

$$goal(A) \text{ :- } perm(A, Ap), rule(Ap, An), goal(An)$$

is applied, yielding the resolvent

$$perm([a_1, \ldots, a_{k+1}], Ap), rule(Ap, An), goal(An).$$

The latter is of interest to us. Resolving the subgoal $perm([a_1, \ldots, a_{k+1}], Ap)$ completely, leads to goals of the form

$$rule([b_1, \ldots, b_{k+1}], An), goal(An)$$

with $[b_1, \ldots, b_{k+1}]$ a permutation of $[a_1, \ldots, a_{k+1}]$ and hence a permutation of $[b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_k]$. Let us consider one with $b_1 = b(X_1, \ldots, X_n)\sigma$. In that case, we can write the goal as

$$rule([b(X_1, \ldots, X_n)\sigma, b_2, \ldots, b_{k+1}], An), goal(An)$$

with $[b_2, \ldots, b_{k+1}]$ a permutation of $[c_1, \ldots, c_k]$.
We can solve this goal with

$$rule([b(X_1, \ldots, X_n)|R], R) \text{ :- } b(X_1, \ldots, X_n),$$

yielding the new goal

$$b(X_1, \ldots, X_n)\sigma, goal([b_2, \ldots, b_{k+1}]).$$

Solving the built-in predicate results — by our assumptions about the constraint store — in the substitution θ and the new goal $goal([b_2, \ldots, b_{k+1}]\theta)$ with $[b_2, \ldots, b_{k+1}]\theta$ a permutation of $[c_1, \ldots, c_k]\theta$. Hence,

$$\gamma(goal([b_2, \ldots, b_{k+1}]\theta) = \{c_1, \ldots, c_k\}\theta = S'$$

which completes the proof for this case.

Simplification. In this case, the constraint store S holds a number of constraints $h_1, \ldots, h_n$ that match the head of a CHR rule in P, i.e. it is of the form

$$\{h_1, \ldots, h_n, c_1, \ldots c_k\}$$

for some $k \geq 0$. The CHR rule is of the form

$$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_r \mid B_1, \ldots, B_l, C_1, \ldots, C_m.$$

Moreover, it is the case that there exists a matching substitution σ such that, for all $i \in [1..n]$ $H_i\sigma = h_i$ and that the guard $(G_1, \ldots, G_r)\sigma$ can be successfully solved, resulting in a substitution θ. As a next state, we thus obtain

$$S' = \{B_1, \ldots, B_l, C_1, \ldots, C_m\}\sigma\theta \uplus \{c_1, \ldots, c_k\}\theta.$$

The goal Q corresponding to the initial constraint store S is of the form

$$goal([a_1, \ldots, a_{n+k}])$$

with $[a_1, \ldots, a_{n+k}]$ a permutation of $[h_1, \ldots, h_n, c_1, \ldots, c_k]$. Similar as in the first case, we can apply the clause

$$goal(A) \texttt{ :- } perm(A, Ap), rule(Ap, An), goal(An)$$

and obtain the resolvent

$$perm([a_1, \ldots, a_{n+k}], Ap), rule(Ap, An), goal(An).$$

Completely resolving the subgoal $perm([a_1, \ldots, a_{n+k}], Ap)$ leads to goals of the form

$$rule([b_1, \ldots, b_{n+k}], An), goal(An)$$

with $[b_1, \ldots, b_{n+k}]$ a permutation of $[a_1, \ldots, a_{n+k}]$ and hence a permutation of $[h_1, \ldots, h_n, c_1, \ldots, c_k]$. Let us consider one with $b_1 = h_1, \ldots, b_n = h_n$. In that case, we can write the goal as

$$rule([h_1, \ldots, h_n, b_{n+1} \ldots, b_{n+k}], An), goal(An)$$

with $[b_{n+1}, \ldots, b_{n+k}]$ a permutation of $[c_1, \ldots, c_k]$.
We can solve this goal with

$$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \texttt{ :- } G_1, \ldots, G_r;$$

we have that, for all i, $h_i = H_i\sigma$, hence we obtain the resolvent

$$G_1\sigma, \ldots, G_r\sigma, goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]).$$

By our assumptions about the store, resolving the guards $G_1\sigma, \ldots, G_n\sigma$ one by one results in an accumulated substitution θ and the new goal

$$goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]\theta)$$

with $[b_{n+1} \ldots, b_{n+k}]\theta$ a permutation of $[c_1, \ldots c_k]\theta$. Hence,

$$\gamma(goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]\theta) =$$
$$\{B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, c_1, \ldots, c_k\}\theta = S'$$

which completes the proof for this case. □

This relation establishes that the analysis of properties of constraints, part of the CHR constraint store during execution of a CHR(Prolog) program, can take place on its transformed program $C2P(P)$ instead. After all, for every two consecutive CHR states, consecutive calls to $goal/1$ with storelist representations of these states exist. Stating the inverse is not true. For the transformed program, a refutation exists for every call to $goal/1$ and matching in CHR is replaced by the more general concept of unification in Prolog.

4 Application of the Transformation to Type Analysis

In the previous section, we discussed the correctness of our transformation for the analysis of constraints present in constraint stores during computations of a CHR(Prolog) program P. That is, for every constraint store $\{c_1, \ldots, c_n\}$ that appears during the execution of P starting from some initial constraint store S, there is a corresponding goal $goal([c_1, \ldots, c_n])$ that appears in the execution of $C2P(P)$ starting from the goal $goal(\alpha(S))$. After introducing the notion of call set, we can formulate this in a more precise way as a corollary of the above theorem. Restricting the call set of $C2P(P)$ to the predicate $goal/1$ yields an over-approximation of the call set of the constraint store, defined similar to the call set of a Prolog program.

Definition 4 (Call set of a Prolog program). *Let S be a set of atomic goals. The call set, $Call(P, S)$, is the set of all atoms A, such that a variant of A is the selected atom in some derivation for (P, Q), for some $Q \in S$.* □

Corollary 1. *Let S be the initial constraint store of a CHR program P. Let C be the set $Call(C2P(P), \{\alpha(S)\})$ restricted to calls of the predicate $goal/1$. Then, $\{S' \mid \exists c \in C$ such that $\gamma(c) = S'\}$ is an over-approximation of the constraint stores that can occur during execution of the CHR program with S.* □

The corollary implies that the transformation preserves properties of the constraints in constraint stores that may occur during execution of the original CHR program. Although the transformation yields an over-approximation, it provides us with accurate information regarding the calls in the CHR program. Such information is derived top-down and does not depend much on the presence of a fire-once policy as argued in the introduction. After all, multiple applications of rules are considered anyways as we analyse the constraint store for classes of initial constraint stores. Approximations resulting from the refutations due to the fact $goal(\_)$, do not influence the analysis of calls either.

Using unification instead of matching can introduce unwanted constraints. Consider for example the CHR rule, $a(1) \Leftrightarrow c$, and an initial constraint store $\{a(X)\}$, where X is some free variable. Then in CHR, the program would terminate immediately as the constraint in the store cannot be matched with the head of the rule. When transforming the CHR rule to a Prolog clause using $C2P$, we obtain: $rule([a(1)|R], [c|R])$; and for the the initial constraint store: $goal([a(X)])$. Due to unification in Prolog, the transformed program does allow the application of the transformed CHR rule. Consequently, a type analyser for Prolog will derive incorrectly that the constraint c can be part of the constraint store. Making the process of matching explicit would resolve the issue, however, doing so requires the use of Prolog built-ins. Type analysers for Prolog, typically, have a problem with this. In general, they do not take information of Prolog systems into account. Making matching explicit can thus only result in inaccurate types. This is in contrast to unification, where we might overestimate the constraints in constraint stores, but will compute the correct types.

A consequence of the corollary is that useful analyses of the resulting Prolog program are those that infer properties about the call set of the program. So, analyses that derive properties about the success set are useless (Unless combined with magic set transformation, so that the success set characterises the call set).

As an illustration of a useful analysis, below we apply the inference of well-typings [2] on the resulting Prolog programs. As well-typed programs cannot go wrong, all calls are well-typed and in particular, the type of the $goal/1$ predicate provides a well-typing for the constraints that can appear in the constraint store.

Other potentially useful analyses are mode analyses. As modes of arguments positions are not useful, one should derive modes annotating the type components; [13] is an example of such an analysis. Then the modes annotating the type of the elements in the list, that is the type of the $goal/1$ predicate (the *constraint* type below) would provide information about the modes of the constraints in the store. Finally, a proof of termination of $goal(\alpha(S))$ is a sufficient condition for termination of the CHR program. However, as CHR programs with propagation are transformed into simplification only programs, we introduce non-termination. Therefore, proving termination on the transformed programs can only be done for programs without propagation [10].

The next example demonstrates a type analysis on C2P(merge-sort) from Example 1. First, all rules become simplification rules, as in Example 3. Then, the program is transformed according to Definition 3:

$$
\begin{array}{lll}
goal(S) \text{ :-} & perm(L,[X|P]) \text{ :-} & del(X,[Y|T],[Y|R]) \text{ :-} \\
\quad perm(S,PS), & \quad del(X,L,L1), & \quad del(X,T,R). \\
\quad rule(PS,NS), & \quad perm(L1,P). & del(X,[X|T],T). \\
\quad goal(NS). & perm([],[]). & \\
goal(\_). & &
\end{array}
$$

$$
leq(s(X),s(Y)) \text{ :- } leq(X,Y). \qquad leq(0,X).
$$

$$
\begin{array}{l}
rule([msort([])|T],T). \\
rule([msort([L|List])|T],[r(0,L),msort(List)|T]). \\
rule([r(D,L1),r(D,L2)|T],[r(s(D),L1),a(L1,L2)|T]) \text{ :- } leq(L1,L2). \\
rule([a(L1,L2),a(L1,L3)|T],[a(L1,L2),a(L2,L3)|T]) \text{ :- } leq(L2,L3).
\end{array}
$$

Notice that we have added a definition for the built-in $leq/2$ for the sake of the analysis. Performing a type analysis on the transformed program with PolyTypes 1.3, yields the following result:

Type definitions:

$constraint \rightarrow msort(list); a(sym\_nat_1, sym\_nat_1); r(sym\_nat_2, sym\_nat_1)$
$list \rightarrow []; [sym\_nat_1|list]$
$sym\_nat_1 \rightarrow 0; s(sym\_nat_1)$
$sym\_nat_2 \rightarrow 0; s(sym\_nat_2)$
$storelist \rightarrow []; [constraint|storelist]$

Signatures:
$goal(storelist)$
$perm(storelist, storelist)$
$del(constraint, storelist, storelist)$
$leq(sym\_nat_1, sym\_nat_1)$

For readability, we have replaced the type names generated by PolyTypes 1.3 by more meaningful ones. Note that sym_nat_1 and sym_nat_2 are equivalent types representing symbolic natural numbers. As these types do not interact through unification, the type inference keeps them separate.

For the analysis of types in CHR, we are only interested in the types present in the storelist of the transformed program. This is given by the signature for $goal/1$. It expresses that the type of its argument is a *storelist*. That is, a list of elements of type *constraint*. Thus, terms of the form $msort(list)$, $a(sym\_nat_1, sym\_nat_1)$ or $r(sym\_nat_2, sym\_nat_1)$. Hence PolyTypes 1.3 correctly derives the types of the constraints that can occur in the storelist and thus in the constraint store.

The reason why PolyTypes is able to derive that we are using a list of symbolic integer values in $msort/1$ is because we have provided for the definition of $leq/2$. This is noticeable as its signature is present in the output generated by PolyTypes. Would we have not provided the implementation of $leq/2$, PolyTypes could not have derived the type definition $(sym\_nat_1 \rightarrow 0; s(sym\_nat_1))$ and would have concluded that $(sym\_nat_1 \rightarrow)$ is a type that can cover any term.

One could add a query to the program. Adding a query can only increase the type inferred by the PolyTypes analysis. For example, adding the CHR query $msort(l(s(s(s(0))), l(s(s(0)), n)))$, which translates into the Prolog query $goal([msort(l(s(s(s(0))), l(s(s(0)), n)))])$, will extend the type definition *list* as the argument of $msort/1$ in the query uses different list constructors than those in the *msort*-rules. That is, we use $l/2$ as list constructor yielding the type definitions $(list \rightarrow list_1; list_2)$, $(list_1 \rightarrow []; [sym\_nat_1|list_1])$, and $(list_2 \rightarrow n; l(sym\_nat_1, list_2))$. Actually, the obtained type is then a grave overestimation of the actual contents of the constraint store as no CHR rule can fire on the query. Here a call type analysis [6] would give more precise results.

Instead of specifying an initial query, one could also specify the type of the initial query, i.e. specifying that a call to $goal/1$ has the type *storelist* and providing for the initial type(s) for *constraint*, e.g. $constraint \rightarrow msort(list)$. Translating these types into input for PolyTypes, the tool will then extend these types and obtain the same types as the ones shown above.

5 Evaluation of the Transformation

The transformation to Prolog from Definition 3 has been implemented and integrated with PolyTypes 1.3 in a tool called CHRTypes[1]. CHRTypes derives the

[1] Available at http://www.cs.kuleuven.be/∼paolo/c2p/

Table 1. Benchmark results CHRTypes = C2P + PolyTypes 1.3

CHR(swi)	T(sec)	CHR(swi)	T(sec)	CHR(swi)	T(sec)
ackermann	0.067	$color_3$	0.068	*concat*	0.066
$constr_{121}$	0.068	$constr_{122}$	0.062	*dcons2sat*	0.582
dfsearch	0.067	*dijkstra*	0.070	*fib_bu*	0.074
fib_td	0.068	*gcd*	0.068	*genchrnet*	0.178
gencss	0.181	*gensccs*	0.277	*hamming*	0.118
knapsack	0.101	*lazychr*	0.158	*linpoleq*	0.072
mergesort	0.082	*nqueen*	0.090	*oddeven*	0.073
power	0.067	$primes_3$	0.079	*revlist*	0.064
rsa	0.113	*shortest_path*	0.069	*solvecases*	0.203
strips	0.089	$trans\_closure_1$	0.072	*unionfind*	0.087
uf_opt	0.152	*weight*	0.078	*ztoa*	0.065

types of CHR(Prolog) programs in a fully automated way, using only PolyTypes 1.3 and our transformation.

Using CHRTypes, we ran 10 tests on a benchmark of 98 CHR(Prolog) programs using a system with an $Intel(R)\ Pentium(R)\ D\ CPU\ 2.80GHz$ and $2G$ of RAM. In Table 1, we have listed the averages of these results for a representative subset of the CHR(Prolog) programs in the benchmark. Next to a set of 28 constructed examples, $compl_i$ and $constr_i$, the greater part of the benchmark consists of practical programs, among which the more complex CHR(Prolog) programs that we are aware of. The 37 example programs originating from WebCHR[2] consist of both small (such as *gcd*) and regular sized (such as *unionfind*) programs. The 33 designed by us consist of small (such as *revlist*), large (such as *genchrnet*), and large and complex (such as *gensccs*) programs.

For all programs, CHRTypes computes the correct types within $0.6s$. For each program, a correct classification of signatures for CHR constraints and Prolog built-ins is given together with the correct set of type definitions.

In the next example, we show the output generated by CHRTypes for mergesort, however, implemented here for integers and not their symbolic counterparts. As such, we use =</2 instead of $leq/2$. The definition of =</2 is not made explicit in the program as it is provided by the host language Prolog.

Example 4 (Output of CHRTypes). CHRTypes generates the following output for integer merge-sort:

```
% Type definitions:
        (P1 →)      (t42(P1) → []; [P1|t42(P1)])      (t37 → 0; s(t37))
% Signature CHR constraints:
        mergesort(t42(P1))      edge(P1, P1)      root(t37, P1)
% Signature Prolog Built-ins:
        true      P1=<P1
% Parsing: 0.0110s; Transforming: 0.0588s; PolyTypes 1.3: 0.0094s (Total: 0.0793s)
```

In the output of CHRTypes for this version of merge-sort (implemented for integers), the arguments of =</2 can take any term. This is because PolyTypes

[2] http://chr.informatik.uni-ulm.de/~webchr/

does not know the implementation of =</2. Nevertheless, PolyTypes correctly infers that both arguments of =</2 must be of the same type. □

The current version of CHRTypes consists of a pipeline of three tools. A parser which actually computes more information than required by the next stage, a transformer interpreting the result of the parser and outputting a Prolog program and finally PolyTypes, which reads the Prolog program and computes the types. Although far from optimal, from a performance point of view, the benchmark results show that performance is acceptable.

This shows in the results we obtained. Although we haven't included in Table 1 the timings for each separate tool, it is generally the case that the time required to compute the well-typing is not proportional to the time required to parse and to transform. Consider for example the following timings for $unionfind$ and gcd, both representative for a normal and toy program, respectively. The $unionfind$ program requires $0.012s$ to parse, $0.051s$ to transform, and $0.022s$ to compute the types. The gcd program requires much less time to compute the types, $0.009s$, but still needs a lot of time to parse, $0.011s$, and to transform, $0.045s$.

This can easily be overcome by better integration of the tools. The parser can easily be stripped of unnecessary components and could integrate with the transformation. PolyTypes could be adapted to accept a program as a list of clauses, avoiding as such the reading and writing of the Prolog program.

Although we demonstrated our transformation by a fully automated type analyser for CHR(Prolog), we could have adapted our system to other kinds of analyses of the calls in CHR programs, such as groundness information, modes and even call types, provided the appropriate analysis tools for Prolog.

6 Conclusion

We have presented a transformation from CHR(Prolog) programs to Prolog programs that respects the abstract CHR semantics. The transformed program describes transitions between storelists. Analysing it with respect to the storelist yields an overestimate of the CHR constraint store.

This way, existing tools for LP can be used to analyse the contents of the CHR constraint store. We have demonstrated this in the context of a type analysis, using the tool PolyTypes 1.3 integrated in our system CHRTypes, and obtained accurate type descriptions for the CHR constraints and Prolog built-ins of a CHR(Prolog) program. CHRTypes is therefore also applicable to pure Prolog programs providing the same accuracy as PolyTypes on Prolog programs.

PolyTypes 1.3 does not take query information into account. There are however other tools which do so, such as the one in [5] for deriving call types. We have demonstrated that given a CHR query specification, there is a straightforward representation into a Prolog query specification for the transformed program. Essentially such a representation from CHR to Prolog corresponds to making the constraint store explicit as a list, enumerating the constraints in the store.

Our transformation does not prioritise on the rules to apply first. In most practical implementations, there is however some kind of selection rule, e.g. based on

rule orderings. In the context of termination this information is essential to prove termination of certain programs. Future work will therefore be directed towards a better understanding of this problem. We will also apply groundness and call type analysis on the result of the transformation and will, based on CHRTypes, develop a call type analyser for integration with CHRisTA [8], yielding the first fully automated termination analyser for CHR(Prolog).

References

1. Bruynooghe, M., Codish, M., Gallagher, J.P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM Transactions on Programming Languages and Systems 29(2), 10 (2007)
2. Bruynooghe, M., Gallagher, J.P., Van Humbeeck, W.: Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 35–51. Springer, Heidelberg (2005)
3. Coquery, E., Fages, F.: A Type System for CHR. In: CHR 2005: Proceedings of the 2nd International Workshop on Constraint Handling Rules, pp. 19–33 (2005)
4. De Schreye, D., Decorte, S.: Termination of Logic Programs: The Never-Ending Story. Journal of Logic Programming 19/20, 199–260 (1994)
5. Janssens, G., Bruynooghe, M.: Deriving descriptions of possible values of program variables by means of abstract interpretation. Journal of Logic Programming 13(2-3), 199–260 (1992)
6. Nguyen, M.T.: Termination Analysis: Crossing Paradigm Borders. PhD thesis, Katholieke Universiteit Leuven, Departement Computer Wetenschappen, Belgium (2009)
7. Pilozzi, P., De Schreye, D.: Termination Analysis of CHR Revisited. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 501–515. Springer, Heidelberg (2008)
8. Pilozzi, P., De Schreye, D.: Automating termination proofs for CHR. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 504–508. Springer, Heidelberg (2009)
9. Pilozzi, P., De Schreye, D.: Proving termination by invariance relations. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 499–503. Springer, Heidelberg (2009)
10. Pilozzi, P., Schrijvers, T., De Schreye, D.: Proving termination of CHR in Prolog: A transformational approach. In: WST 2007: Proceedings of the 9th International Workshop on Termination (2007)
11. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. PhD thesis, Katholieke Universiteit Leuven, Departement Computer Wetenschappen, Belgium (2005)
12. Schrijvers, T., Bruynooghe, M., Gallagher, J.P.: From monomorphic to polymorphic well-typings and beyond. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 152–167. Springer, Heidelberg (2009)
13. Vanhoof, W., Bruynooghe, M., Leuschel, M.: Binding-time analysis for mercury. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 189–232. Springer, Heidelberg (2004)

The Dependency Triple Framework for Termination of Logic Programs*

Peter Schneider-Kamp[1], Jürgen Giesl[2], and Manh Thang Nguyen[3]

[1] IMADA, University of Southern Denmark, Denmark
[2] LuFG Informatik 2, RWTH Aachen University, Germany
[3] Department of Computer Science, K.U. Leuven, Belgium

Abstract. We show how to combine the two most powerful approaches for automated termination analysis of logic programs (LPs): the *direct* approach which operates directly on LPs and the *transformational* approach which transforms LPs to term rewrite systems (TRSs) and tries to prove termination of the resulting TRSs. To this end, we adapt the well-known *dependency pair framework* from TRSs to LPs. With the resulting method, one can combine arbitrary termination techniques for LPs in a completely modular way and one can use both direct and transformational techniques for different parts of the same LP.

1 Introduction

When comparing the direct and the transformational approach for termination of LPs, there are the following advantages and disadvantages. The *direct* approach is more efficient (since it avoids the transformation to TRSs) and in addition to the TRS techniques that have been adapted to LPs [13,15], it can also use numerous other techniques that are specific to LPs. The *transformational* approach has the advantage that it can use *all* existing termination techniques for TRSs, not just the ones that have already been adapted to LPs.

Two of the leading tools for termination of LPs are Polytool [14] (implementing the direct approach and including the adapted TRS techniques from [13,15]) and AProVE [7] (implementing the transformational approach of [17]). In the annual *International Termination Competition*,[1] AProVE was the most powerful tool for termination analysis of LPs (it solved 246 out of 349 examples), but Polytool obtained a close second place (solving 238 examples). Nevertheless, there are several examples where one tool succeeds, whereas the other does not.

This shows that both the direct and the transformational approach have their benefits. Thus, one should combine these approaches *in a modular way*. In other words, for one and the same LP, it should be possible to prove termination of some parts with the direct approach and of other parts with the transformational

* Supported by *FWO/2006/09: Termination analysis: Crossing paradigm borders* and by the *Deutsche Forschungsgemeinschaft (DFG)*, grant *GI 274/5-2*.

[1] `http://www.termination-portal.org/wiki/Termination_Competition`

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 37–51, 2010.

approach. The resulting method would improve over both approaches and can also prove termination of LPs that cannot be handled by one approach alone.

In this paper, we solve that problem. We build upon [15], where the well-known *dependency pair* (DP) method from term rewriting [2] was adapted in order to apply it to LPs directly. However, [15] only adapted the most basic parts of the method and moreover, it only adapted the classical variant of the DP method instead of the more powerful recent *DP framework* [6,8,9] which can combine different TRS termination techniques in a completely flexible way.

After providing the necessary preliminaries on LPs in Sect. 2, in Sect. 3 we adapt the DP framework to the LP setting which results in the new *dependency triple (DT) framework*. Compared to [15], the advantage is that now arbitrary termination techniques based on DTs can be applied in any combination and any order. In Sect. 4, we present three termination techniques within the DT framework. In particular, we also develop a new technique which can transform *parts* of the original LP termination problem into TRS termination problems. Then one can apply TRS techniques and tools to solve these subproblems.

We implemented our contributions in the tool Polytool and coupled it with AProVE which is called on those subproblems which were converted to TRSs. Our experimental evaluation in Sect. 5 shows that this combination clearly improves over both Polytool or AProVE alone, both concerning efficiency and power.

2 Preliminaries on Logic Programming

We briefly recapitulate needed notations. More details on logic programming can be found in [1], for example. A *signature* is a pair (Σ, Δ) where Σ and Δ are finite sets of function and predicate symbols and $\mathcal{T}(\Sigma, \mathcal{V})$ resp. $\mathcal{A}(\Sigma, \Delta, \mathcal{V})$ denote the sets of all terms resp. atoms over the signature (Σ, Δ) and the variables $\mathcal{V}$. We always assume that Σ contains at least one constant of arity 0. A *clause* c is a formula $H \leftarrow B_1, \ldots, B_k$ with $k \geq 0$ and $H, B_i \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$. A finite set of clauses $\mathcal{P}$ is a (definite) *logic program*. A clause with empty body is a *fact* and a clause with empty head is a *query*. We usually omit "$\leftarrow$" in queries and just write "$B_1, \ldots, B_k$". The empty query is denoted $\square$.

For a *substitution* $\delta : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$, we often write $t\delta$ instead of $\delta(t)$, where t can be any expression (e.g., a term, atom, clause, etc.). If δ is a variable renaming (i.e., a one-to-one correspondence on $\mathcal{V}$), then $t\delta$ is a *variant* of t. We write $\delta\sigma$ to denote that the application of δ is followed by the application of σ. A substitution δ is a *unifier* of two expressions s and t iff $s\delta = t\delta$. To simplify the presentation, in this paper we restrict ourselves to ordinary unification with occur check. We call δ the *most general unifier (mgu)* of s and t iff δ is a unifier of s and t and for all unifiers σ of s and t, there is a substitution μ such that $\sigma = \delta\mu$.

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 δ (denoted $Q \vdash_{c,\delta} Q'$) if $\delta = mgu(A_1, H)$, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\delta$. A *derivation* of a program $\mathcal{P}$ and a query Q is a possibly infinite sequence $Q_0, Q_1, \ldots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_i,\delta_i} Q_{i+1}$ for some substitution δ_i and some renamed-apart variant c_i of

a clause of $\mathcal{P}$. For a derivation $Q_0, \ldots, Q_n$ as above, we also write $Q_0 \vdash^n_{\mathcal{P},\delta_0\ldots\delta_{n-1}} Q_n$ or $Q_0 \vdash^n_{\mathcal{P}} Q_n$, and we also write $Q_i \vdash_{\mathcal{P}} Q_{i+1}$ for $Q_i \vdash_{c_i,\delta_i} Q_{i+1}$. A LP $\mathcal{P}$ is *terminating* for the query Q if all derivations of $\mathcal{P}$ and Q are finite. The *answer set* $Answer(\mathcal{P}, Q)$ for a LP $\mathcal{P}$ and a query Q is the set of all substitutions δ such that $Q \vdash^n_{\mathcal{P},\delta} \square$ for some $n \in \mathbb{N}$. For a set of atomic queries $\mathcal{S} \subseteq \mathcal{A}(\Sigma, \Delta, \mathcal{V})$, we define the *call set* $Call(\mathcal{P}, \mathcal{S}) = \{A_1 \mid Q \vdash^n_{\mathcal{P}} A_1, \ldots, A_m,\ Q \in \mathcal{S},\ n \in \mathbb{N}\}$.

Example 1. The following LP $\mathcal{P}$ uses "$\mathsf{s2m}$*" to create a matrix M of variables for fixed numbers X and Y of rows and columns. Afterwards, it uses "*$\mathsf{subs\_mat}$*" to replace each variable in the matrix by the constant "*a*".*

$\mathsf{goal}(X, Y, Msu) \leftarrow \mathsf{s2m}(X, Y, M), \mathsf{subs\_mat}(M, Msu).$
$\mathsf{s2m}(\mathsf{0}, Y, [\,]).$ $\quad \mathsf{s2m}(\mathsf{s}(X), Y, [R|Rs]) \leftarrow \mathsf{s2}\ell(Y, R), \mathsf{s2m}(X, Y, Rs).$
$\mathsf{s2}\ell(\mathsf{0}, [\,]).$ $\quad \mathsf{s2}\ell(\mathsf{s}(Y), [C|Cs]) \leftarrow \mathsf{s2}\ell(Y, Cs).$
$\mathsf{subs\_mat}([\,], [\,]).$ $\quad \mathsf{subs\_mat}([R|Rs], [SR|SRs]) \leftarrow \mathsf{subs\_row}(R, SR), \mathsf{subs\_mat}(Rs, SRs).$
$\mathsf{subs\_row}([\,], [\,]).$ $\quad \mathsf{subs\_row}([E|R], [\mathsf{a}|SR]) \leftarrow \mathsf{subs\_row}(R, SR).$

For example, for suitable substitutions δ_0 and δ_1 we have $\mathsf{goal}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Msu) \vdash_{\delta_0,\mathcal{P}} \mathsf{s2m}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), M), \mathsf{subs\_mat}(M, Msu) \vdash^8_{\delta_1,\mathcal{P}} \square$. *So* $Answer(\mathcal{P}, \mathsf{goal}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Msu))$ *contains $\delta = \delta_0\delta_1$, where $\delta(Msu) = [[\mathsf{a}]]$.*

We want to prove termination of this program for the set of queries $\mathcal{S} = \{\mathsf{goal}(t_1, t_2, t_3) \mid t_1$ and t_2 are ground terms $\}$. Here, we obtain

$$Call(\mathcal{P}, \mathcal{S}) \subseteq \mathcal{S} \cup \{\{\mathsf{s2m}(t_1, t_2, t_3) \mid t_1 \text{ and } t_2 \text{ ground}\} \cup \{\mathsf{s2}\ell(t_1, t_2) \mid t_1 \text{ ground}\} \cup \{\mathsf{subs\_row}(t_1, t_2) \mid t_1 \in List\} \cup \{\mathsf{subs\_mat}(t_1, t_2) \mid t_1 \in List\}$$

where $List$ is the smallest set with $[\,] \in List$ and $[t_1 \mid t_2] \in List$ if $t_2 \in List$.

3 Dependency Triple Framework

As mentioned before, we already adapted the basic DP method to the LP setting in [15]. The advantage of [15] over previous direct approaches for LP termination is that (a) it can use different well-founded orders for different "loops" of the LP and (b) it uses a constraint-based approach to search for arbitrary suitable well-founded orders (instead of only choosing from a fixed set of orders based on a given small set of norms). Most other direct approaches have only one of the features (a) or (b). Nevertheless, [15] has the disadvantage that it does not permit the combination of arbitrary termination techniques in a flexible and modular way. Therefore, we now adapt the recent DP framework [6,8,9] to the LP setting. Def. 2 adapts the notion of *dependency pairs* [2] from TRSs to LPs.[2]

Definition 2 (Dependency Triple). *A* dependency triple *(DT) is a clause $H \leftarrow I, B$ where H and B are atoms and I is a list of atoms. For a LP $\mathcal{P}$, the set of its dependency triples is $DT(\mathcal{P}) = \{H \leftarrow I, B \mid H \leftarrow I, B, \ldots \in \mathcal{P}\}$.*

[2] While Def. 2 is essentially from [15], the rest of this section contains new concepts that are needed for a flexible and general framework.

Example 3. The dependency triples $DT(\mathcal{P})$ of the program in Ex. 1 are:

$$\mathsf{goal}(X, Y, Msu) \leftarrow \mathsf{s2m}(X, Y, M). \quad (1)$$
$$\mathsf{goal}(X, Y, Msu) \leftarrow \mathsf{s2m}(X, Y, M), \mathsf{subs\_mat}(M, Msu). \quad (2)$$
$$\mathsf{s2m}(\mathsf{s}(X), Y, [R|Rs]) \leftarrow \mathsf{s2}\ell(Y, R). \quad (3)$$
$$\mathsf{s2m}(\mathsf{s}(X), Y, [R|Rs]) \leftarrow \mathsf{s2}\ell(Y, R), \mathsf{s2m}(X, Y, Rs). \quad (4)$$
$$\mathsf{s2}\ell(\mathsf{s}(Y), [C|Cs]) \leftarrow \mathsf{s2}\ell(Y, Cs). \quad (5)$$
$$\mathsf{subs\_mat}([R|Rs], [SR|SRs]) \leftarrow \mathsf{subs\_row}(R, SR). \quad (6)$$
$$\mathsf{subs\_mat}([R|Rs], [SR|SRs]) \leftarrow \mathsf{subs\_row}(R, SR), \mathsf{subs\_mat}(Rs, SRs). \quad (7)$$
$$\mathsf{subs\_row}([E|R], [\mathsf{a}|SR]) \leftarrow \mathsf{subs\_row}(R, SR). \quad (8)$$

Intuitively, a dependency triple $H \leftarrow I, B$ 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. To use DTs for termination analysis, one has to show that there are no infinite "chains" of such calls. The following definition corresponds to the standard definition of *chains* from the TRS setting [2]. Usually, $\mathcal{D}$ stands for the set of DTs, $\mathcal{P}$ is the program under consideration, and $\mathcal{C}$ stands for $Call(\mathcal{P}, \mathcal{S})$ where $\mathcal{S}$ is the set of queries to be analyzed for termination.

Definition 4 (Chain). *Let $\mathcal{D}$ and $\mathcal{P}$ be sets of clauses and let $\mathcal{C}$ be a set of atoms. A (possibly infinite) list $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathcal{D}$ is a $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-*chain *iff there are substitutions θ_i, σ_i and an $A \in \mathcal{C}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, we have $\sigma_i \in Answer(\mathcal{P}, I_i\theta_i)$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}$.*[3]

Example 5. For $\mathcal{P}$ and $\mathcal{S}$ from Ex. 1, the list (2), (7) is a $(DT(\mathcal{P}), Call(\mathcal{P}, \mathcal{S}), \mathcal{P})$-chain. To see this, consider $\theta_0 = \{X/\mathsf{s}(\mathsf{0}), Y/\mathsf{s}(\mathsf{0})\}$, $\sigma_0 = \{M/[[C]]\}$, and $\theta_1 = \{R/[C], Rs/[\,], Msu/[SR, SRs]\}$. Then, for $A = \mathsf{goal}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Msu) \in \mathcal{S}$, we have $H_0\theta_0 = \mathsf{goal}(X, Y, Msu)\theta_0 = A\theta_0$. Furthermore, we have $\sigma_0 \in Answer(\mathcal{P}, \mathsf{s2m}(X, Y, M)\theta_0) = Answer(\mathcal{P}, \mathsf{s2m}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), M))$ and $\theta_1 = mgu(B_0\theta_0\sigma_0, H_1) = mgu(\mathsf{subs\_mat}([[C]], Msu), \mathsf{subs\_mat}([R|Rs], [SR|SRs]))$.

Thm. 6 shows that termination is equivalent to absence of infinite chains.

Theorem 6 (Termination Criterion). *A LP $\mathcal{P}$ is terminating for a set of atomic queries $\mathcal{S}$ iff there is no infinite $(DT(\mathcal{P}), Call(\mathcal{P}, \mathcal{S}), \mathcal{P})$-chain.*

Proof. For the "if"-direction, let there be an infinite derivation $Q_0, Q_1, \ldots$ with $Q_0 \in \mathcal{S}$ and $Q_i \vdash_{c_i, \delta_i} Q_{i+1}$. The clause $c_i \in \mathcal{P}$ has the form $H_i \leftarrow A_i^1, \ldots, A_i^{k_i}$. Let $j_1 > 0$ be the minimal index such that the first atom A'_{j_1} in Q_{j_1} starts an infinite derivation. Such a j_1 always exists as shown in [17, Lemma 3.5]. As we started from an atomic query, there must be some m_0 such that $A'_{j_1} =$

[3] If $\mathcal{C} = Call(\mathcal{P}, \mathcal{S})$, then the condition "$B_i\theta_i\sigma_i \in \mathcal{C}$" is always satisfied due to the definition of "*Call*". But our goal is to formulate the concept of "chains" as general as possible (i.e., also for cases where $\mathcal{C}$ is an arbitrary set). Then this condition can be helpful in order to obtain as few chains as possible.

$A_0^{m_0}\delta_0\delta_1\ldots\delta_{j_1-1}$. Then "$H_0 \leftarrow A_0^1, \ldots, A_0^{m_0-1}, A_0^{m_0}$" is the first DT in our $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}), \mathcal{P})$-chain where $\theta_0 = \delta_0$ and $\sigma_0 = \delta_1\ldots\delta_{j_1-1}$. As $Q_0 \vdash_{\mathcal{P}}^{j_1} Q_{j_1}$ and $A_0^{m_0}\theta_0\sigma_0 = A'_{j_1}$ is the first atom in Q_{j_1}, we have $A_0^{m_0}\theta_0\sigma_0 \in Call(\mathcal{P},\mathcal{S})$.

We repeat this construction and let j_2 be the minimal index with $j_2 > j_1$ such that the first atom A'_{j_2} in Q_{j_2} starts an infinite derivation. As the first atom of Q_{j_1} already started an infinite derivation, there must be some m_{j_1} such that $A'_{j_2} = A_{j_1}^{m_{j_1}}\delta_{j_1}\ldots\delta_{j_2-1}$. Then "$H_{j_1} \leftarrow A_{j_1}^1, \ldots, A_{j_1}^{m_{j_1}-1}, A_{j_1}^{m_{j_1}}$" is the second DT in our $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}), \mathcal{P})$-chain where $\theta_1 = mgu(A_0^{m_0}\theta_0\sigma_0, H_{j_1}) = \delta_{j_1}$ and $\sigma_1 = \delta_{j_1+1}\ldots\delta_{j_2-1}$. As $Q_0 \vdash_{\mathcal{P}}^{j_2} Q_{j_2}$ and $A_{j_1}^{m_{j_1}}\theta_1\sigma_1 = A'_{j_2}$ is the first atom in Q_{j_2}, we have $A_{j_1}^{m_{j_1}}\theta_1\sigma_1 \in Call(\mathcal{P},\mathcal{S})$. By repeating this construction infinitely many times, we obtain an infinite $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}), \mathcal{P})$-chain.

For the "only if"-direction, assume that $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ is an infinite $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}), \mathcal{P})$-chain. Thus, there are substitutions θ_i, σ_i and an $A \in Call(\mathcal{P},\mathcal{S})$ 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 $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$. Due to the construction of $DT(\mathcal{P})$, there is a clause $c_0 \in \mathcal{P}$ with $c_0 = H_0 \leftarrow I_0, B_0, R_0$ for a list of atoms R_0 and the first step in our derivation is $A \vdash_{c_0,\theta_0} I_0\theta_0, B_0\theta_0, R_0\theta_0$. ¿From $\sigma_0 \in Answer(\mathcal{P}, I_0\theta_0)$ we obtain the derivation $I_0\theta_0 \vdash_{\mathcal{P},\sigma_0}^{n_0} \square$ and consequently, $I_0\theta_0, B_0\theta_0, R_0\theta_0 \vdash_{\mathcal{P},\sigma_0}^{n_0} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0$ for some $n_0 \in \mathbb{N}$. Hence, $A \vdash_{\mathcal{P},\theta_0\sigma_0}^{n_0+1} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0$. As $\theta_1 = mgu(B_0\theta_0\sigma_0, H_1)$ and as there is a clause $c_1 = H_1 \leftarrow I_1, B_1, R_1 \in \mathcal{P}$, we continue the derivation with $B_0\theta_0\sigma_0, R_0\theta_0\sigma_0 \vdash_{c_1,\theta_1} I_1\theta_1, B_1\theta_1, R_1\theta_1, R_0\theta_0\sigma_0\theta_1$. Due to $\sigma_1 \in Answer(\mathcal{P}, I_1\theta_1)$ we continue with $I_1\theta_1, B_1\theta_1, R_1\theta_1, R_0\theta_0\sigma_0\theta_1 \vdash_{\mathcal{P},\sigma_1}^{n_1} B_1\theta_1\sigma_1, R_1\theta_1\sigma_1, R_0\theta_0\sigma_0\theta_1\sigma_1$ for some $n_1 \in \mathbb{N}$.

By repeating this, we obtain an infinite derivation $A \vdash_{\mathcal{P},\theta_0\sigma_0}^{n_0+1} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0 \vdash_{\mathcal{P},\theta_1,\sigma_1}^{n_1+1} B_1\theta_1\sigma_1, R_1\theta_1\sigma_1, R_0\theta_0\sigma_0\theta_1\sigma_1 \vdash_{\mathcal{P},\theta_2\sigma_2}^{n_2+1} B_2\theta_2\sigma_2, \ldots \vdash_{\mathcal{P},\theta_3\sigma_3}^{n_2+1} \ldots$ Thus, the LP $\mathcal{P}$ is not terminating for A. ¿From $A \in Call(\mathcal{P},\mathcal{S})$ we know there is a $Q \in \mathcal{S}$ such that $Q \vdash_{\mathcal{P}}^{n} A, \ldots$ Hence, $\mathcal{P}$ is also not terminating for $Q \in \mathcal{S}$. □

Termination techniques are now called *DT processors* and they operate on so-called *DT problems* and try to prove absence of infinite chains.

Definition 7 (DT Problem). *A* DT problem *is a triple $(\mathcal{D},\mathcal{C},\mathcal{P})$ where $\mathcal{D}$ and $\mathcal{P}$ are finite sets of clauses and $\mathcal{C}$ is a set of atoms. A DT problem $(\mathcal{D},\mathcal{C},\mathcal{P})$ is* terminating *iff there is no infinite $(\mathcal{D},\mathcal{C},\mathcal{P})$-chain.*

A DT processor *Proc takes a DT problem as input and returns a set of DT problems which have to be solved instead. Proc is* sound *if for all non-terminating DT problems $(\mathcal{D},\mathcal{C},\mathcal{P})$, there is also a non-terminating DT problem in Proc($(\mathcal{D},\mathcal{C},\mathcal{P})$). So if Proc($(\mathcal{D},\mathcal{C},\mathcal{P})$) = $\varnothing$, then termination of $(\mathcal{D},\mathcal{C},\mathcal{P})$ is proved.*

Termination proofs now start with the *initial* DT problem $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}), \mathcal{P})$ whose termination is equivalent to the termination of the LP $\mathcal{P}$ for the queries $\mathcal{S}$, cf. Thm. 6. Then sound DT processors are applied repeatedly until all DT problems have been simplified to $\varnothing$.

4 Dependency Triple Processors

In Sect. 4.1 and 4.2, we adapt two of the most important DP processors from term rewriting [2,6,8,9] to the LP setting. In Sect. 4.3 we present a new DT processor to convert DT problems to DP problems.

4.1 Dependency Graph Processor

The first processor decomposes a DT problem into subproblems. Here, one constructs a *dependency graph* to determine which DTs follow each other in chains.

Definition 8 (Dependency Graph). *For a DT problem $(\mathcal{D}, \mathcal{C}, \mathcal{P})$, the nodes of the $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-*dependency graph *are the clauses of $\mathcal{D}$ and there is an arc from a clause c to a clause d iff "c, d" is a $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-chain.*

Example 9. For the initial DT problem $(DT(\mathcal{P}), Call(\mathcal{P}, \mathcal{S}), \mathcal{P})$ of the program in Ex. 1, we obtain the following dependency graph.

As in the TRS setting, the dependency graph is not computable in general. For TRSs, several techniques were developed to over-approximate dependency graphs automatically, cf. e.g. [2,9]. Def. 10 adapts the estimation of [2].[4] This estimation ignores the intermediate atoms I in a DT $H \leftarrow I, B$.

Definition 10 (Estimated Dependency Graph). *For a DT problem $(\mathcal{D}, \mathcal{C}, \mathcal{P})$, the nodes of the* estimated $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-dependency graph *are the clauses of $\mathcal{D}$ and there is an arc from $H_i \leftarrow I_i, B_i$ to $H_j \leftarrow I_j, B_j$, iff B_i unifies with a variant of H_j and there are atoms $A_i, A_j \in \mathcal{C}$ such that A_i unifies with a variant of H_i and A_j unifies with a variant of H_j.*

For the program of Ex. 1, the estimated dependency graph is identical to the real dependency graph in Ex. 9.

Example 11. To illustrate their difference, consider the LP $\mathcal{P}'$ with the clauses $\mathsf{p} \leftarrow \mathsf{q(a)}, \mathsf{p}$ and $\mathsf{q(b)}$. We consider the set of queries $\mathcal{S}' = \{\mathsf{p}\}$ and obtain $Call(\mathcal{P}', \mathcal{S}') = \{\mathsf{p}, \mathsf{q(a)}\}$. There are two DTs $\mathsf{p} \leftarrow \mathsf{q(a)}$ and $\mathsf{p} \leftarrow \mathsf{q(a)}, \mathsf{p}$. In the estimated dependency graph for the initial DT problem $(DT(\mathcal{P}'), Call(\mathcal{P}', \mathcal{S}'), \mathcal{P}')$, there is an arc from the second DT to itself. But this arc is missing in the real dependency graph because of the unsatisfiable body atom $\mathsf{q(a)}$.

The following lemma proves the "soundness" of estimated dependency graphs.

[4] The advantage of a general concept of dependency graphs like Def. 8 is that this permits the introduction of better estimations in the future without having to change the rest of the framework. However, a general concept like Def. 8 was missing in [15], which only featured a variant of the estimated dependency graph from Def. 10.

Lemma 12. *The estimated $(\mathcal{D},\mathcal{C},\mathcal{P})$-dependency graph over-approximates the real $(\mathcal{D},\mathcal{C},\mathcal{P})$-dependency graph, i.e., whenever there is an arc from c to d in the real graph, then there is also such an arc in the estimated graph.*

Proof. Assume that there is an arc from the clause $H_i \leftarrow I_i, B_i$ to $H_j \leftarrow I_j, B_j$ in the real dependency graph. Then by Def. 4, there are substitutions σ_i and θ_i such that θ_{i+1} is a unifier of $B_i\theta_i\sigma_i$ and H_j. As we can assume H_j and B_i to be variable disjoint, $\theta_i\sigma_i\theta_{i+1}$ is a unifier of B_i and H_j. Def. 4 also implies that for all DTs $H \leftarrow I, B$ in a $(\mathcal{D},\mathcal{C},\mathcal{P})$-chain, there is an atom from $\mathcal{C}$ unifying with H. Hence, this also holds for H_i and H_j. □

A set $\mathcal{D}' \neq \varnothing$ of DTs is a *cycle* if for all $c, d \in \mathcal{D}'$, there is a non-empty path from c to d traversing only DTs of $\mathcal{D}'$. A cycle $\mathcal{D}'$ is a *strongly connected component (SCC)* if $\mathcal{D}'$ is not a proper subset of another cycle. So the dependency graph in Ex. 9 has the SCCs $\mathcal{D}_1 = \{(4)\}$, $\mathcal{D}_2 = \{(5)\}$, $\mathcal{D}_3 = \{(7)\}$, $\mathcal{D}_4 = \{(8)\}$. The following processor allows us to prove termination separately for each SCC.

Theorem 13 (Dependency Graph Processor). *We define Proc$(\,(\mathcal{D},\mathcal{C},\mathcal{P})\,)$ $= \{(\mathcal{D}_1,\mathcal{C},\mathcal{P}),\ldots,(\mathcal{D}_n,\mathcal{C},\mathcal{P})\}$, where $\mathcal{D}_1,\ldots,\mathcal{D}_n$ are the SCCs of the (estimated) $(\mathcal{D},\mathcal{C},\mathcal{P})$-dependency graph. Then Proc is sound.*

Proof. Let there be an infinite $(\mathcal{D},\mathcal{C},\mathcal{P})$-chain. This infinite chain corresponds to an infinite path in the dependency graph (resp. in the estimated graph, by Lemma 12). Since $\mathcal{D}$ is finite, the path must be contained entirely in some SCC $\mathcal{D}_i$. Thus, $(\mathcal{D}_i,\mathcal{C},\mathcal{P})$ is non-terminating. □

Example 14. For the program of Ex. 1, the above processor transforms the initial DT problem $(DT(\mathcal{P}), Call(\mathcal{P},\mathcal{S}),\mathcal{P})$ to $(\mathcal{D}_1, Call(\mathcal{P},\mathcal{S}),\mathcal{P})$, $(\mathcal{D}_2, Call(\mathcal{P},\mathcal{S}),\mathcal{P})$, $(\mathcal{D}_3, Call(\mathcal{P},\mathcal{S}),\mathcal{P})$, and $(\mathcal{D}_4, Call(\mathcal{P},\mathcal{S}),\mathcal{P})$. So the original termination problem is split up into four subproblems which can now be solved independently.

4.2 Reduction Pair Processor

The next processor uses a *reduction pair* $(\succsim,\succ)$ and requires that all DTs are weakly or strictly decreasing. Then the strictly decreasing DTs can be removed from the current DT problem. A *reduction pair* $(\succsim,\succ)$ consists of a quasi-order $\succsim$ on atoms and terms (i.e., a reflexive and transitive relation) and a well-founded order $\succ$ (i.e., there is no infinite sequence $t_0 \succ t_1 \succ \ldots$). Moreover, $\succsim$ and $\succ$ have to be *compatible* (i.e., $t_1 \succsim t_2 \succ t_3$ implies $t_1 \succ t_3$).[5]

Example 15. We often use reduction pairs built from norms and level mappings [3]. A norm is a mapping $\|\cdot\| : \mathcal{T}(\Sigma,\mathcal{V}) \to \mathbb{N}$. A level mapping is a mapping $|\cdot| : \mathcal{A}(\Sigma,\Delta,\mathcal{V}) \to \mathbb{N}$. Consider the reduction pair $(\succsim,\succ)$ induced[6]

[5] In contrast to "reduction pairs" in rewriting, we do not require $\succsim$ and $\succ$ to be closed under substitutions. But for automation, we usually choose relations $\succsim$ and $\succ$ that result from polynomial interpretations which are closed under substitutions.

[6] So for terms t_1, t_2 we define $t_1 \,{}_{(}\succsim_{)}\, t_2$ iff $\|t_1\| \,{}_{(}\geq_{)}\, \|t_2\|$ and for atoms A_1, A_2 we define $A_1 \,{}_{(}\succsim_{)}\, A_2$ iff $|A_1| \,{}_{(}\geq_{)}\, |A_2|$.

by the norm $\|X\| = 0$ *for all variables* X, $\|[\,]\| = 0$, $\|\mathsf{s}(t)\| = \|[s \mid t]\| = 1+\|t\|$ *and the level mapping* $|\mathsf{s2m}(t_1,t_2,t_3)| = |\mathsf{s2}\ell(t_1,t_2)| = |\mathsf{subs\_mat}(t_1,t_2)| = |\mathsf{subs\_row}(t_1,t_2)| = \|t_1\|$. *Then* $\mathsf{subs\_mat}([[C]],[SR \mid SRs]) \succ \mathsf{subs\_mat}([\,],SRs)$, *as* $|\mathsf{subs\_mat}([[C]],[SR \mid SRs])| = \|[[C]]\| = 1$ *and* $|\mathsf{subs\_mat}([\,],SRs)| = \|[\,]\| = 0$.

Now we can define when a DT $H \leftarrow I, B$ is decreasing. Roughly, we require that $H\sigma \succ B\sigma$ must hold for every substitution σ. However, we do not have to regard *all* substitutions, but we may restrict ourselves to such substitutions where all variables of H and B on positions that are "taken into account" by $\succsim$ and $\succ$ are instantiated by ground terms.[7] Formally, a reduction pair $(\succsim, \succ)$ is *rigid* on a term or atom t if we have $t \approx t\delta$ for all substitutions δ. Here, we define $s \approx t$ iff $s \succsim t$ and $t \succsim s$. A reduction pair $(\succsim, \succ)$ is rigid on a set of terms or atoms if it is rigid on all its elements. Now for a DT $H \leftarrow I, B$ to be decreasing, we only require that $H\sigma \succ B\sigma$ holds for all σ where $(\succsim, \succ)$ is rigid on $H\sigma$.

Example 16. The reduction pair from Ex. 15 is rigid on the atom $A = \mathsf{s2m}([[C]], [SR \mid SRs])$, *since* $|A\delta| = 1$ *holds for every substitution* δ. *Moreover, if* $\sigma(Rs) \in \mathit{List}$, *then the reduction pair is also rigid on* $\mathsf{subs\_mat}([R \mid Rs],[SR \mid SRs])\sigma$. *For every such* σ, *we have* $\mathsf{subs\_mat}([R \mid Rs],[SR \mid SRs])\sigma \succ \mathsf{subs\_mat}(Rs, SRs)\sigma$.

We refine the notion of "decreasing" DTs $H \leftarrow I, B$ further. Instead of only considering H and B, one should also take the intermediate body atoms I into account. To approximate their semantics, we use *interargument relations*. An *interargument relation* for a predicate p is a relation $IR_p = \{p(t_1,\ldots,t_n) \mid t_i \in \mathcal{T}(\Sigma,\mathcal{V}) \wedge \varphi_p(t_1,\ldots,t_n)\}$, where (1) $\varphi_p(t_1,\ldots,t_n)$ is a formula of an arbitrary Boolean combination of inequalities, and (2) each inequality in φ_p is either $s_i \succsim s_j$ or $s_i \succ s_j$, where s_i, s_j are constructed from $t_1,\ldots,t_n$ by applying function symbols of $\mathcal{P}$. IR_p is *valid* iff $p(t_1,\ldots,t_n) \vdash^m_{\mathcal{P}} \square$ implies $p(t_1,\ldots,t_n) \in IR_p$ for every $p(t_1,\ldots,t_n) \in \mathcal{A}(\Sigma,\Delta,\mathcal{V})$.

Definition 17 (Decreasing DTs). *Let* $(\succsim, \succ)$ *be a reduction pair, and* $\mathfrak{R} = \{IR_{p_1},\ldots,IR_{p_k}\}$ *be a set of valid interargument relations based on* $(\succsim, \succ)$. *Let* $c = H \leftarrow p_1(\boldsymbol{t}_1),\ldots,p_k(\boldsymbol{t}_k), B$ *be a DT. Here, the* $\boldsymbol{t}_i$ *are tuples of terms.*

The DT c *is* weakly decreasing *(denoted* $(\succsim, \mathfrak{R}) \models c$*) if* $H\sigma \succsim B\sigma$ *holds for any substitution* σ *where* $(\succsim, \succ)$ *is rigid on* $H\sigma$ *and where* $p_1(\boldsymbol{t}_1)\sigma \in IR_{p_1},\ldots,$ $p_k(\boldsymbol{t}_k)\sigma \in IR_{p_k}$. *Analogously,* c *is* strictly decreasing *(denoted* $(\succ, \mathfrak{R}) \models c$*) if* $H\sigma \succ B\sigma$ *holds for any such* σ.

Example 18. Recall the reduction pair from Ex. 15 and the remarks about its rigidity in Ex. 16. When considering a set $\mathfrak{R}$ *of trivial valid interargument relations like* $IR_{\mathsf{subs\_row}} = \{\mathsf{subs\_row}(t_1,t_2) \mid t_1,t_2 \in \mathcal{T}(\Sigma,\mathcal{V})\}$, *then the DT (7) is strictly decreasing. Similarly,* $(\succ, \mathfrak{R}) \models (4)$, $(\succ, \mathfrak{R}) \models (5)$, *and* $(\succ, \mathfrak{R}) \models (8)$.

We can now formulate our second DT processor. To automate it, we refer to [15] for a description of how to synthesize valid interargument relations and how to find reduction pairs automatically that make DTs decreasing.

[7] This suffices, because we require $(\succsim, \succ)$ to be *rigid on* $\mathcal{C}$ in Thm. 19. Thus, $\succsim$ and $\succ$ do not take positions into account where atoms from $\mathit{Call}(\mathcal{P},\mathcal{S})$ have variables.

Theorem 19 (Reduction Pair Processor). *Let* $(\succsim, \succ)$ *be a reduction pair and let* $\mathfrak{R}$ *be a set of valid interargument relations. Then Proc is sound.*

$$Proc(\,(\mathcal{D},\mathcal{C},\mathcal{P})\,) = \begin{cases} \{(\mathcal{D} \setminus \mathcal{D}_\succ, \mathcal{C}, \mathcal{P})\}, \textit{ if} \\ \quad \bullet\ (\succsim, \succ) \textit{ is rigid on } \mathcal{C} \textit{ and} \\ \quad \bullet\ \textit{there is } \mathcal{D}_\succ \subseteq \mathcal{D} \textit{ with } \mathcal{D}_\succ \neq \varnothing \textit{ such that } (\succ, \mathfrak{R}) \models c \\ \qquad \textit{for all } c \in \mathcal{D}_\succ \textit{ and } (\succsim, \mathfrak{R}) \models c \textit{ for all } c \in \mathcal{D} \setminus \mathcal{D}_\succ \\ \{(\mathcal{D},\mathcal{C},\mathcal{P})\}, \textit{otherwise} \end{cases}$$

Proof. If $Proc(\,(\mathcal{D},\mathcal{C},\mathcal{P})\,) = \{(\mathcal{D},\mathcal{C},\mathcal{P})\}$, then *Proc* is trivially sound. Now we consider the case $Proc(\,(\mathcal{D},\mathcal{C},\mathcal{P})\,) = \{(\mathcal{D}\setminus\mathcal{D}_\succ,\mathcal{C},\mathcal{P})\}$. Assume that $(\mathcal{D}\setminus\mathcal{D}_\succ,\mathcal{C},\mathcal{P})$ is terminating while $(\mathcal{D},\mathcal{C},\mathcal{P})$ is non-terminating. Then there is an infinite $(\mathcal{D},\mathcal{C},\mathcal{P})$-chain $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ where at least one clause from $\mathcal{D}_\succ$ appears infinitely often. There are $A \in \mathcal{C}$ and substitutions θ_i, σ_i such that $\theta_0 = mgu(A, H_0)$ and for all i, we have $\sigma_i \in Answer(\mathcal{P}, I_i\theta_i)$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}$. We obtain

$$\begin{array}{ll}
& H_i\theta_i \\
\approx & H_i\theta_i\sigma_i\theta_{i+1} \qquad \text{(by rigidity, as } H_i\theta_i = B_{i-1}\theta_{i-1}\sigma_{i-1}\theta_i \\
& \qquad \text{and } B_{i-1}\theta_{i-1}\sigma_{i-1} \in \mathcal{C}) \\
\succsim & B_i\theta_i\sigma_i\theta_{i+1} \qquad \text{(since } (\succsim, \mathfrak{R}) \models c_i \text{ where } c_i \text{ is } H_i \leftarrow I_i, B_i, \\
& \qquad \text{as } (\succsim, \succ) \text{ is also rigid on any instance of } H_i\theta_i, \\
& \qquad \text{and since } \sigma_i \in Answer(\mathcal{P}, I_i\theta_i) \text{ implies } I_i\theta_i\sigma_i\theta_{i+1} \vdash^n_{\mathcal{P}} \square \\
& \qquad \text{and } \mathfrak{R} \text{ are valid interargument relations)} \\
= & H_{i+1}\theta_{i+1} \qquad \text{(since } \theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})) \\
\approx & H_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} \text{ (by rigidity, as } H_{i+1}\theta_{i+1} = B_i\theta_i\sigma_i\theta_{i+1} \text{ and } B_i\theta_i\sigma_i \in \mathcal{C}) \\
\succsim & B_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} \text{ (since } (\succsim, \mathfrak{R}) \models c_{i+1} \text{ where } c_{i+1} \text{ is } H_{i+1} \leftarrow I_{i+1}, B_{i+1}) \\
= & \ldots
\end{array}$$

Here, infinitely many $\succsim$-steps are "strict" (i.e., we can replace infinitely many $\succsim$-steps by $\succ$-steps). This contradicts the well-foundedness of $\succ$. □

So in our example, we apply the reduction pair processor to all 4 DT problems in Ex. 14. While we could use different reduction pairs for the different DT problems,[8] Ex. 18 showed that all their DTs are strictly decreasing for the reduction pair from Ex. 15. This reduction pair is indeed rigid on $Call(\mathcal{P}, \mathcal{S})$. Hence, the reduction pair processor transforms all 4 remaining DT problems to $(\varnothing, Call(\mathcal{P}, \mathcal{S}), \mathcal{P})$, which in turn is transformed to $\varnothing$ by the dependency graph processor. Thus, termination of the LP in Ex. 1 is proved.

4.3 Modular Transformation Processor to Term Rewriting

The previous two DT processors considerably improve over [15] due to their increased modularity.[9] In addition, one could easily adapt more techniques from

[8] Using different reduction pairs for different DT problems resulting from one and the same LP is for instance necessary for programs like the *Ackermann* function, cf. [15].

[9] In [15] these two processors were part of a fixed procedure, whereas now they can be applied to any DT problem at any time during the termination proof.

the DP framework (i.e., from the TRS setting) to the DT framework (i.e., to the LP setting). However, we now introduce a new DT processor which allows us to apply any TRS termination technique immediately to LPs (i.e., without having to adapt the TRS technique). It transforms a DT problem for LPs into a DP problem for TRSs.

Example 20. The following program $\mathcal{P}$ from [11] is part of the Termination Problem Data Base (TPDB) *used in the* International Termination Competition. *Typically,* cnf*'s first argument is a Boolean formula (where the function symbols* n, a, o *stand for the Boolean connectives) and the second is a variable which will be instantiated to an equivalent formula in conjunctive normal form. To this end,* cnf *uses the predicate* tr *which holds if its second argument results from its first one by a standard transformation step towards conjunctive normal form.*

$$\begin{array}{ll}
\mathsf{cnf}(X,Y) \leftarrow \mathsf{tr}(X,Z), \mathsf{cnf}(Z,Y). & \mathsf{cnf}(X,X).\\
\mathsf{tr}(\mathsf{n}(\mathsf{n}(X)),X). & \mathsf{tr}(\mathsf{o}(X_1,Y),\mathsf{o}(X_2,Y)) \leftarrow \mathsf{tr}(X_1,X_2).\\
\mathsf{tr}(\mathsf{n}(\mathsf{a}(X,Y)),\mathsf{o}(\mathsf{n}(X),\mathsf{n}(Y))). & \mathsf{tr}(\mathsf{o}(X,Y1),\mathsf{o}(X,Y2)) \leftarrow \mathsf{tr}(Y1,Y2).\\
\mathsf{tr}(\mathsf{n}(\mathsf{o}(X,Y)),\mathsf{a}(\mathsf{n}(X),\mathsf{n}(Y))). & \mathsf{tr}(\mathsf{a}(X_1,Y),\mathsf{a}(X_2,Y)) \leftarrow \mathsf{tr}(X_1,X_2).\\
\mathsf{tr}(\mathsf{o}(X,\mathsf{a}(Y,Z)),\mathsf{a}(\mathsf{o}(X,Y),\mathsf{o}(X,Z))). & \mathsf{tr}(\mathsf{a}(X,Y1),\mathsf{a}(X,Y2)) \leftarrow \mathsf{tr}(Y1,Y2).\\
\mathsf{tr}(\mathsf{o}(\mathsf{a}(X,Y),Z),\mathsf{a}(\mathsf{o}(X,Z),\mathsf{o}(Y,Z))). & \mathsf{tr}(\mathsf{n}(X_1),\mathsf{n}(X_2)) \leftarrow \mathsf{tr}(X_1,X_2).
\end{array}$$

Consider the queries $\mathcal{S} = \{\mathsf{cnf}(t_1,t_2) \mid t_1$ is ground$\} \cup \{\mathsf{tr}(t_1,t_2) \mid t_1$ is ground$\}$. By applying the dependency graph processor to the initial DT problem, we obtain two new DT problems. The first is $(\mathcal{D}_1, Call(\mathcal{P},\mathcal{S}),\mathcal{P})$ where $\mathcal{D}_1$ contains all recursive tr*-clauses. This DT problem can easily be solved by the reduction pair processor. The other resulting DT problem is*

$$(\{\mathsf{cnf}(X,Y) \leftarrow \mathsf{tr}(X,Z), \mathsf{cnf}(Z,Y)\}, Call(\mathcal{P},\mathcal{S}),\mathcal{P}). \tag{9}$$

To make this DT strictly decreasing, one needs a reduction pair $(\succsim, \succ)$ where $t_1 \succ t_2$ holds whenever $\mathsf{tr}(t_1,t_2)$ is satisfied. This is impossible with the orders $\succ$ in current direct LP termination tools. In contrast, it would easily be possible if one uses other orders like the recursive path order *[5] which is well established in term rewriting. This motivates the new processor presented in this section.*

To transform DT to DP problems, we adapt the existing transformation from logic programs $\mathcal{P}$ to TRSs $\mathcal{R}_\mathcal{P}$ from [17]. Here, two new n-ary function symbols p_{in} and p_{out} are introduced for each n-ary predicate p:

- Each fact $p(\boldsymbol{s})$ of the LP is transformed to the rewrite rule $p_{in}(\boldsymbol{s}) \to p_{out}(\boldsymbol{s})$.
- Each clause c of the form $p(\boldsymbol{s}) \leftarrow p_1(\boldsymbol{s}_1), \ldots, p_k(\boldsymbol{s}_k)$ is transformed into the following rewrite rules:

$$\begin{array}{r}
p_{in}(\boldsymbol{s}) \to u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))\\
u_{c,1}(p_{1_{out}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s})) \to u_{c,2}(p_{2_{in}}(\boldsymbol{s}_2), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1))\\
\ldots\\
u_{c,k}(p_{k_{out}}(\boldsymbol{s}_k), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1) \cup \ldots \cup \mathcal{V}(\boldsymbol{s}_{k-1})) \to p_{out}(\boldsymbol{s})
\end{array}$$

  Here, the $u_{c,i}$ are new function symbols and $\mathcal{V}(\boldsymbol{s})$ are the variables in $\boldsymbol{s}$. Moreover, if $\mathcal{V}(\boldsymbol{s}) = \{x_1, \ldots, x_n\}$, then "$u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))$" abbreviates the term $u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), x_1, \ldots, x_n)$, etc.

So the fact $\mathsf{tr}(\mathsf{n}(\mathsf{n}(X)), X)$ is transformed to $\mathsf{tr}_{in}(\mathsf{n}(\mathsf{n}(X)), X) \to \mathsf{tr}_{out}(\mathsf{n}(\mathsf{n}(X)), X)$ and the clause $\mathsf{cnf}(X, Y) \leftarrow \mathsf{tr}(X, Z), \mathsf{cnf}(Z, Y)$ is transformed to

$$\mathsf{cnf}_{in}(X, Y) \to \mathsf{u}_1(\mathsf{tr}_{in}(X, Z), X, Y) \tag{10}$$

$$\mathsf{u}_1(\mathsf{tr}_{out}(X, Z), X, Y) \to \mathsf{u}_2(\mathsf{cnf}_{in}(Z, Y), X, Y, Z) \tag{11}$$

$$\mathsf{u}_2(\mathsf{cnf}_{out}(Z, Y), X, Y, Z) \to \mathsf{cnf}_{out}(X, Y) \tag{12}$$

To formulate the connection between a LP and its corresponding TRS, the sets of queries that should be analyzed for termination have to be represented by an *argument filter* π where $\pi(f) \subseteq \{1, \ldots, n\}$ for every n-ary $f \in \Sigma \cup \Delta$. We extend π to terms and atoms by defining $\pi(x) = x$ if x is a variable and $\pi(f(t_1, \ldots, t_n)) = f(\pi(t_{i_1}), \ldots, \pi(t_{i_k}))$ if $\pi(f) = \{i_1, \ldots, i_k\}$ with $i_1 < \ldots < i_k$.

Argument filters specify those positions which have to be instantiated with ground terms. In Ex. 20, we wanted to prove termination for the set $\mathcal{S}$ of all queries $\mathsf{cnf}(t_1, t_2)$ or $\mathsf{tr}(t_1, t_2)$ where t_1 is ground. These queries are described by the filter with $\pi(\mathsf{cnf}) = \pi(\mathsf{tr}) = \{1\}$. Hence, we can also represent $\mathcal{S}$ as $\mathcal{S} = \{A \mid A \in \mathcal{A}(\Sigma, \Delta, \mathcal{V}), \pi(A) \text{ is ground}\}$. Thm. 21 shows that instead of proving termination of a LP $\mathcal{P}$ for a set of queries $\mathcal{S}$, it suffices to prove termination of the corresponding TRS $\mathcal{R}_\mathcal{P}$ for a corresponding set of terms $\mathcal{S}'$. As shown in [17], here we have to regard a variant of term rewriting called *infinitary constructor rewriting*, where variables in rewrite rules may only be instantiated by *constructor terms*,[10] which however may be *infinite*. This is needed since LPs use unification, whereas TRSs use matching for their evaluation.

Theorem 21 (Soundness of the Transformation [17]). *Let $\mathcal{R}_\mathcal{P}$ be the TRS resulting from transforming a LP $\mathcal{P}$ over a signature (Σ, Δ). Let π be an argument filter with $\pi(p_{in}) = \pi(p)$ for all $p \in \Delta$. Let $\mathcal{S} = \{A \mid A \in \mathcal{A}(\Sigma, \Delta, \mathcal{V}), \pi(A)$ is finite and ground$\}$ and $\mathcal{S}' = \{p_{in}(\boldsymbol{t}) \mid p(\boldsymbol{t}) \in \mathcal{S}\}$. If the TRS $\mathcal{R}_\mathcal{P}$ terminates for all terms in $\mathcal{S}'$, then the LP $\mathcal{P}$ terminates for all queries in $\mathcal{S}$.*

The DP framework for termination of term rewriting can also be used for infinitary constructor rewriting, cf. [17]. To this end, for each defined symbol f, one introduces a fresh *tuple symbol* $f^\sharp$ of the same arity. For a term $t = g(\boldsymbol{t})$ with defined root symbol g, let $t^\sharp$ denote $g^\sharp(\boldsymbol{t})$. Then the set of *dependency pairs* for a TRS $\mathcal{R}$ is $DP(\mathcal{R}) = \{\ell^\sharp \to t^\sharp \mid \ell \to r \in \mathcal{R}, t \text{ is a subterm of } r \text{ with defined root symbol}\}$. For instance, the rules (10) - (12) give rise to the following DPs.

$$\mathsf{cnf}^\sharp_{in}(X, Y) \to \mathsf{tr}^\sharp_{in}(X, Z) \tag{13}$$

$$\mathsf{cnf}^\sharp_{in}(X, Y) \to \mathsf{u}^\sharp_1(\mathsf{tr}_{in}(X, Z), X, Y) \tag{14}$$

$$\mathsf{u}^\sharp_1(\mathsf{tr}_{out}(X, Z), X, Y) \to \mathsf{cnf}^\sharp_{in}(Z, Y) \tag{15}$$

$$\mathsf{u}^\sharp_1(\mathsf{tr}_{out}(X, Z), X, Y) \to \mathsf{u}^\sharp_2(\mathsf{cnf}_{in}(Z, Y), X, Y, Z) \tag{16}$$

Termination problems are now represented as *DP problems* $(\mathcal{D}, \mathcal{R}, \pi)$ where $\mathcal{D}$ and $\mathcal{R}$ are TRSs (here, $\mathcal{D}$ is usually a set of DPs) and π is an argument filter. A

[10] As usual, the symbols on root positions of left-hand sides of rewrite rules are called *defined* symbols and all remaining function symbols are *constructors*. A *constructor term* is a term built only from constructors and variables.

list $s_1 \to t_1, s_2 \to t_2, \ldots$ of variants from $\mathcal{D}$ is a $(\mathcal{D}, \mathcal{R}, \pi)$-*chain* iff for all i, there are substitutions σ_i such that $t_i\sigma_i$ rewrites to $s_{i+1}\sigma_{i+1}$ and such that $\pi(s_i\sigma_i)$, $\pi(t_i\sigma_i)$, and $\pi(q)$ are finite and ground, for all terms q in the reduction from $t_i\sigma_i$ and $s_{i+1}\sigma_{i+1}$. $(\mathcal{D}, \mathcal{R}, \pi)$ is *terminating* iff there is no infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain.

Example 22. For instance, "(14), (15)" is a chain for the argument filter π with $\pi(\mathsf{cnf}^\sharp_{in}) = \pi(\mathsf{tr}_{in}) = \{1\}$ and $\pi(u_1^\sharp) = \pi(\mathsf{tr}_{out}) = \{1, 2\}$. To see this, consider the substitution $\sigma = \{X/\mathsf{n}(\mathsf{n}(\mathsf{a})), Z/\mathsf{a}\}$. Now $\mathsf{u}_1^\sharp(\mathsf{tr}_{in}(X, Z), X, Y)\sigma$ reduces in one step to $\mathsf{u}_1^\sharp(\mathsf{tr}_{out}(X, Z), X, Y)\sigma$ and all instantiated left- and right-hand sides of (14) and (15) are ground after filtering them with π.

To prove termination of a TRS $\mathcal{R}$ for all terms $\mathcal{S}'$ in Thm. 21, now it suffices to show termination of the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi)$. Here, one has to make sure that $\pi(DP(\mathcal{R}_\mathcal{P}))$ and $\pi(\mathcal{R}_\mathcal{P})$ satisfy the *variable condition*, i.e., that $\mathcal{V}(\pi(r)) \subseteq \mathcal{V}(\pi(\ell))$ holds for all $\ell \to r \in DP(\mathcal{R}) \cup \mathcal{R}$. If this does not hold, then π has to be refined (by filtering away more argument positions) until the variable condition is fulfilled. This leads to the following corollary from [17].

Corollary 23 (Transformation Technique [17]). *Let $\mathcal{R}_\mathcal{P}, \mathcal{P}, \pi$ be as in Thm. 21, where $\pi(p_{in}) = \pi(p^\sharp_{in}) = \pi(p)$ for all $p \in \Delta$. Let $\pi(DP(\mathcal{R}_\mathcal{P}))$ and $\pi(\mathcal{R}_\mathcal{P})$ satisfy the variable condition and let $\mathcal{S} = \{A \mid A \in \mathcal{A}(\Sigma, \Delta, \mathcal{V}), \pi(A)$ is finite and ground$\}$. If the DP problem $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi)$ is terminating, then the LP $\mathcal{P}$ terminates for all queries in $\mathcal{S}$.*

Note that Thm. 21 and Cor. 23 are applied right at the beginning of the termination proof. So here one immediately transforms the full LP into a TRS (or a DP problem) and performs the whole termination proof on the TRS level. The disadvantage is that LP-specific techniques cannot be used anymore. It would be better to only apply this transformation for those parts of the termination proof where it is necessary and to perform most of the proof on the LP level.

This is achieved by the following new transformation processor within our DT framework. Now one can first apply other DT processors like the ones from Sect. 4.1 and 4.2 (or other LP termination techniques). Only for those subproblems where a solution cannot be found, one uses the following DT processor.

Theorem 24 (DT Transformation Processor). *Let $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ be a DT problem and let π be an argument filter with $\pi(p_{in}) = \pi(p^\sharp_{in}) = \pi(p)$ for all predicates p such that $\mathcal{C} \subseteq \{A \mid A \in \mathcal{A}(\Sigma, \Delta, \mathcal{V}), \pi(A)$ is finite and ground$\}$ and such that $\pi(DP(\mathcal{R}_\mathcal{D}))$ and $\pi(\mathcal{R}_\mathcal{P})$ satisfy the variable condition. Then Proc is sound.*

$$Proc(\,(\mathcal{D}, \mathcal{C}, \mathcal{P})\,) = \begin{cases} \varnothing, & \text{if } (DP(\mathcal{R}_\mathcal{D}), \mathcal{R}_\mathcal{P}, \pi) \text{ is a terminating DP problem} \\ \{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}, & \text{otherwise} \end{cases}$$

Proof. If $Proc(\,(\mathcal{D}, \mathcal{C}, \mathcal{P})\,) = \{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}$, then soundness is trivial. Now let $Proc(\,(\mathcal{D}, \mathcal{C}, \mathcal{P})\,) = \varnothing$. Assume there is an infinite $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-chain $(H_0 \leftarrow I_0, B_0)$, $(H_1 \leftarrow I_1, B_1), \ldots$ Similar to the proof of Thm. 6, we have

$$A \vdash_{H_0 \leftarrow I_0, B_0, \theta_0} I_0\theta_0, B_0\theta_0 \vdash^{n_0}_{\mathcal{P}, \sigma_0} B_0\theta_0\sigma_0 \vdash_{H_1 \leftarrow I_1, B_1, \theta_1} I_1\theta_1, B_1\theta_1 \vdash^{n_1}_{\mathcal{P}, \sigma_1} B_0\theta_1\sigma_1 \ldots$$

For every atom $p(t_1,\ldots,t_n)$, let $\overline{p(t_1,\ldots,t_n)}$ be the term $p_{in}(t_1,\ldots,t_n)$. Then by the results on the correspondence between LPs and TRSs from [17] (in particular [17, Lemma 3.4]), we can conclude

$$\overline{A}\theta_0\sigma_0\,(\xrightarrow{\varepsilon}_{\mathcal{R}_\mathcal{D}}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\,\overline{B_0}\theta_0\sigma_0,\quad \overline{B_0}\theta_0\sigma_0\theta_1\sigma_1\,(\xrightarrow{\varepsilon}_{\mathcal{R}_\mathcal{D}}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\,\overline{B_1}\theta_0\sigma_0\theta_1\sigma_1,\ldots$$

Here, $\to_{\mathcal{R}}$ denotes the rewrite relation of a TRS $\mathcal{R}$, $\xrightarrow{\varepsilon}$ resp. $\xrightarrow{>\varepsilon}$ denote reductions on resp. below the root position and $\to^*$ resp. $\to^+$ denote zero or more resp. one or more reduction steps. This implies

$$\overline{A}^{\sharp}\theta_0\sigma_0\,(\xrightarrow{\varepsilon}_{DP(\mathcal{R}_\mathcal{D})}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\,\overline{B_0}^{\sharp}\theta_0\sigma_0,\ \overline{B_0}^{\sharp}\theta_0\sigma_0\theta_1\sigma_1\,(\xrightarrow{\varepsilon}_{DP(\mathcal{R}_\mathcal{D})}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\,\overline{B_1}^{\sharp}\theta_0\sigma_0\theta_1\sigma_1,$$

etc. Let σ be the infinite substitution $\theta_0\sigma_0\theta_1\sigma_1\theta_2\sigma_2\ldots$ where all remaining variables in σ's range can w.l.o.g. be replaced by ground terms. Then we have

$$\overline{A}^{\sharp}\sigma\quad(\xrightarrow{\varepsilon}_{DP(\mathcal{R}_\mathcal{D})}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\quad\overline{B_0}^{\sharp}\sigma\quad(\xrightarrow{\varepsilon}_{DP(\mathcal{R}_\mathcal{D})}\circ\,\xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_\mathcal{P}})^{+}\quad\overline{B_1}^{\sharp}\sigma\ldots,\qquad(17)$$

which gives rise to an infinite $(DP(\mathcal{R}_\mathcal{D}),\mathcal{R}_\mathcal{P},\pi)$-chain. To see this, note that $\pi(A)$ and all $\pi(B_i\theta_i\sigma_i)$ are finite and ground by the definition of chains of DTs. Hence, this also holds for $\pi(\overline{A}^{\sharp}\sigma)$ and all $\pi(\overline{B_i}^{\sharp}\sigma)$. Moreover, since $\pi(DP(\mathcal{R}_\mathcal{D}))$ and $\pi(\mathcal{R}_\mathcal{P})$ satisfy the variable condition, all terms occurring in the reduction (17) are finite and ground when filtering them with π. □

Example 25. We continue the termination proof of Ex. 20. Since the remaining DT problem (9) could not be solved by direct termination tools, we apply the DT processor of Thm. 24. Here, $\mathcal{R}_\mathcal{D} = \{(10),(11),(12)\}$ and hence, we obtain the DP problem $(\{(13),\ldots,(16)\},\mathcal{R}_\mathcal{P},\pi)$ where $\pi(\mathsf{cnf}) = \pi(\mathsf{tr}) = \{1\}$. On the other function symbols, π is defined as in Ex. 22 in order to fulfill the variable condition. This DP problem can easily be proved terminating by existing TRS techniques and tools, e.g., by using a recursive path order.

5 Experiments and Conclusion

We have introduced a new DT framework for termination analysis of LPs. It permits to split termination problems into subproblems, to use different orders for the termination proof of different subproblems, and to transform subproblems into termination problems for TRSs in order to apply existing TRS tools. In particular, it subsumes and improves upon recent direct and transformational approaches for LP termination analysis like [15,17].

To evaluate our contributions, we performed extensive experiments comparing our new approach with the most powerful current direct and transformational tools for LP termination: Polytool [14] and AProVE [7].[11] The *International Termination Competition* showed that direct termination tools like Polytool and

[11] In [17], Polytool and AProVE were compared with three other representative tools for LP termination analysis: TerminWeb [4], cTI [12], and TALP [16]. Here, TerminWeb and cTI use a direct approach whereas TALP uses a transformational approach. In the experiments of [17], it turned out that Polytool and AProVE were considerably more powerful than the other three tools.

transformational tools like AProVE have comparable power, cf. Sect. 1. Nevertheless, there exist examples where one tool is successful, whereas the other fails.

For example, AProVE fails on the LP from Ex. 1. The reason is that by Cor. 23, it has to represent $Call(\mathcal{P}, \mathcal{S})$ by an argument filtering π which satisfies the variable condition. However, in this example there is no such argument filtering π where $(DP(\mathcal{R}_\mathcal{P}), \mathcal{P}, \pi)$ is terminating. In contrast, Polytool represents $Call(\mathcal{P}, \mathcal{S})$ by type graphs [10] and easily shows termination of this example.

On the other hand, Polytool fails on the LP from Ex. 20. Here, one needs orders like the recursive path order that are not available in direct termination tools. Indeed, other powerful direct termination tools such as TerminWeb [4] and cTI [12] fail on this example, too. The transformational tool TALP [16] fails on this program as well, as it does not use recursive path orders. In contrast, AProVE easily proves termination using a suitable recursive path order.

The results of this paper combine the advantages of direct and transformational approaches. We implemented our new approach in a new version of Polytool. Whenever the transformation processor of Thm. 24 is used, it calls AProVE on the resulting DP problem. Thus, we call our implementation "PolyAProVE".

In our experiments, we applied the two existing tools Polytool and AProVE as well as our new tool PolyAProVE to a set of 298 LPs. This set includes all LP examples of the TPDB that is used in the *International Termination Competition*. However, to eliminate the influence of the translation from Prolog to pure logic programs, we removed all examples that use non-trivial built-in predicates or that are not definite logic programs after ignoring the cut operator. This yields the same set of examples that was used in the experimental evaluation of [17]. In addition to this set we considered two more examples: the LP of Ex. 1 and the combination of Examples 1 and 20. For all examples, we used a time limit of 60 seconds corresponding to the standard setting of the competition.

Below, we give the results and the overall time (in seconds) required to run the tools on all 298 examples.

	PolyAProVE	AProVE	Polytool
Successes	**237**	232	218
Failures	**58**	58	73
Timeouts	**3**	8	7
Total Runtime	**762.3**	2227.2	588.8
Avg. Time	**2.6**	7.5	2.0

Our experiments show that PolyAProVE solves all examples that can be solved by Polytool or AProVE (including both LPs from Ex. 1 and 20). PolyAProVE also solves all examples from this collection that can be handled by any of the three other tools TerminWeb, cTI, and TALP. Moreover, it also succeeds on LPs whose termination could not be proved by any tool up to now. For example, it proves termination of the LP consisting of the clauses of both Ex. 1 and 20 together, whereas all other five tools fail. Another main advantage of PolyAProVE compared to powerful purely transformational tools like AProVE is a substantial increase in efficiency. PolyAProVE needs only about one third (34%) of the total

runtime of AProVE. The reason is that many examples can already be handled by the direct techniques introduced in this paper. The transformation to term rewriting, which incurs a significant runtime penalty, is only used if the other DT processors fail. Thus, the performance of PolyAProVE is much closer to that of direct tools like Polytool than to that of transformational tools like AProVE.

For details on our experiments and to access our collection of examples, we refer to `http://aprove.informatik.rwth-aachen.de/eval/PolyAProVE/`

References

1. Apt, K.R.: From Logic Programming to Prolog. Prentice Hall, London (1997)
2. Arts, T., Giesl, J.: Termination of Term Rewriting using Dependency Pairs. Theoretical Computer Science 236(1,2), 133–178 (2000)
3. Bossi, A., Cocco, N., Fabris, M.: Norms on Terms and their use in Proving Universal Termination of a Logic Program. Th. Comp. Sc. 124(2), 297–328 (1994)
4. Codish, M., Taboch, C.: A Semantic Basis for Termination Analysis of Logic Programs. Journal of Logic Programming 41(1), 103–123 (1999)
5. Dershowitz, N.: Termination of Rewriting. J. Symb. Comp. 3(1,2), 69–116 (1987)
6. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
7. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the DP Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
8. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and Improving Dependency Pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
9. Hirokawa, N., Middeldorp, A.: Automating the Dependency Pair Method. Information and Computation 199(1,2), 172–199 (2005)
10. Janssens, G., Bruynooghe, M.: Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. Journal of Logic Programming 13(2,3), 205–258 (1992)
11. Jurdzinski, M.: LP Course Notes, `http://www.dcs.warwick.ac.uk/mju/CS205/`
12. Mesnard, F., Bagnara, R.: cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. Theory and Practice of Logic Programming 5(1,2), 243–257 (2005)
13. Nguyen, M.T., De Schreye, D.: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 311–325. Springer, Heidelberg (2005)
14. Nguyen, M.T., De Schreye, D.: Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 210–218. Springer, Heidelberg (2007)
15. Nguyen, M.T., Giesl, J., Schneider-Kamp, P., De Schreye, D.: Termination Analysis of Logic Programs based on Dependency Graphs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 8–22. Springer, Heidelberg (2008)
16. Ohlebusch, E., Claves, C., Marché, C.: TALP: A Tool for the Termination Analysis of Logic Programs. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 270–273. Springer, Heidelberg (2000)
17. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Proofs for Logic Programs by Term Rewriting. ACM Transactions on Computational Logic 11(1) (2009)

Goal-Directed and Relative Dependency Pairs for Proving the Termination of Narrowing*

José Iborra[1], Naoki Nishida[2], and Germán Vidal[1]

[1] DSIC, Universidad Politécnica de Valencia, Spain
{jiborra,gvidal}@dsic.upv.es
[2] Graduate School of Information Science, Nagoya University, Nagoya, Japan
nishida@is.nagoya-u.ac.jp

Abstract. In this work, we first consider a *goal-oriented* extension of the dependency pair framework for proving termination w.r.t. a given set of initial terms. Then, we introduce a new result for proving *relative* termination in terms of a dependency pair problem. Both contributions put together allow us to define a simple and powerful approach to analyzing the termination of *narrowing*, an extension of rewriting that replaces matching with unification in order to deal with logic variables. Our approach could also be useful in other contexts where considering termination w.r.t. a given set of terms is also natural (e.g., proving the termination of functional programs).

1 Introduction

Proving that a program terminates is a fundamental problem that has been extensively studied in almost all programming paradigms. In *term rewriting*, where termination analysis has attracted considerable attention (see, e.g., the surveys of Dershowitz [8] and Steinbach [23]), the termination of a rewrite system is usually proved for all possible reduction sequences.

In some cases, however, one is only interested in those sequences that start from a distinguished set of terms. This case has been already considered in some previous works, e.g., for proving the termination of logic programs [20], for proving the termination of Haskell programs [10], and for proving the termination of *narrowing* [25], an extension of rewriting to deal with logic variables. Unfortunately, these works do not focus on proving termination from an initial set of terms—only consider this problem to some extent—and are difficult to generalize.

In this paper, we first extend the well-known *dependency pair* framework [3,11] for proving the termination of rewriting in order to only consider derivations

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by *Generalitat Valenciana* under grants ACOMP/2009/017 and GV/2009/024, and by *UPV* (programs PAID-05-08 and PAID-06-08). The second author has been partially supported by a grant from Nagoya Industrial Science Research Institute.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 52–66, 2010.

from a given initial set of terms. The fundamental improvements are twofold: firstly, we introduce a notion of chain which considers only reachable loops, thus reducing the number of pairs to consider; secondly, we also present a notion of usable rules that regards as usable only those rules which occur in the derivation from an initial term, allowing us to reduce the number of rules.

As a second contribution of this paper, we study a direct application of the dependency pair approach to the solution of relative termination problems when the involved TRSs form a hierarchical combination. Roughly speaking, we can study whether $\mathcal{R}$ terminates relative to $\mathcal{B}$ (i.e., whether all $\to_{\mathcal{R}} \cup \to_{\mathcal{B}}$ reductions contain only finitely many $\to_{\mathcal{R}}$ steps) in those cases where $\mathcal{B}$ does not make calls to functions defined in $\mathcal{R}$. Although this application is arguably folklore in the literature (see, e.g., the work of [24]), to our knowledge this is the first time that the necessary conditions have been ascertained and proved in a formal publication.

Finally, we illustrate the usefulness of our developments by applying them to proving the termination of narrowing starting from initial terms. Our results are more general and potentially more accurate than previous approaches (e.g., [25]). Moreover, our approach could also be useful in other contexts where considering termination w.r.t. a given set of terms is also a natural requirement, like the approach to proving the termination of Haskell programs of [10] or that to proving the termination of logic programs by translating them to rewrite systems of [20].

The paper is organized as follows. After introducing some preliminaries in the next section, we present the goal-directed dependency pair framework in Section 3. Then, Section 4 first states a useful result for proving the relative termination of a rewrite system and, then, presents a new approach for proving the termination of narrowing. Finally, Section 5 reports on the implementation of a termination prover based on the ideas of this paper and concludes. An extended version including proofs of technical results can be found in [15].

2 Preliminaries

We assume familiarity with basic concepts of term rewriting and narrowing. We refer the reader to, e.g., [4] and [13] for further details.

Terms and Substitutions. A *signature* $\mathcal{F}$ is a set of function symbols. We often write $\mathsf{f}/n \in \mathcal{F}$ to denote that the arity of function f is n. Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \emptyset$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume that $\mathcal{F}$ always contains at least one constant $\mathsf{f}/0$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. A *position* p in a term t is represented by a finite sequence of natural numbers, where ϵ denotes the root position. Positions are used to address the nodes of a term viewed as a tree. The root symbol of a term t is denoted by $\mathsf{root}(t)$. We let $t|_p$ denote the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. $\mathcal{V}\mathsf{ar}(t)$ denotes the set of variables appearing in t. A term t is *ground* if $\mathcal{V}\mathsf{ar}(t) = \emptyset$. We write $\mathcal{T}(\mathcal{F})$ as a shorthand for the set of ground terms $\mathcal{T}(\mathcal{F}, \emptyset)$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}\mathsf{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. The set of variables introduced by a substitution σ is denoted by $\mathcal{R}\mathsf{an}(\sigma) = \cup_{x \in \mathcal{D}\mathsf{om}(\sigma)} \mathcal{V}\mathsf{ar}(x\sigma)$. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution σ to a term t by $t\sigma$ (rather than $\sigma(t)$). The identity substitution is denoted by id.

TRSs and Rewriting. A set of rewrite rules $l \to r$ such that l is a non-variable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$.

We use the notation $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ to point out that $\mathcal{D}$ are the defined function symbols and $\mathcal{C}$ are the constructors of a signature $\mathcal{F}$, with $\mathcal{D} \cap \mathcal{C} = \emptyset$. The domains $\mathcal{T}(\mathcal{C}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C})$ denote the sets of *constructor terms* and *ground constructor terms*, respectively. A substitution σ is (ground) *constructor*, if $x\sigma$ is a (ground) constructor term for all $x \in \mathcal{D}\mathsf{om}(\sigma)$.

A TRS $\mathcal{R}$ is a *constructor system* if the left-hand sides of its rules have the form $\mathsf{f}(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. A term t is *linear* if every variable of $\mathcal{V}$ occurs at most once in t. A TRS $\mathcal{R}$ is *left-linear* if l is linear for every rule $l \to r \in \mathcal{R}$.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_{\mathcal{R}}$ as follows: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_{\mathcal{R}} t$ iff there exists a position p in s, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is often denoted by $s \to_{p, l \to r} t$ to make explicit the position and rule used in this step. The instantiated left-hand side $l\sigma$ is called a *redex*.

A term t is called *irreducible* or in *normal form* in a TRS $\mathcal{R}$ if there is no term s with $t \to_{\mathcal{R}} s$. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_{\mathcal{R}} s$ means that t can be reduced to s in $\mathcal{R}$ in zero or more steps; we also use $t \to^n_{\mathcal{R}} s$ to denote that t can be reduced to s in exactly n rewrite steps.

Narrowing. Given a TRS $\mathcal{R}$ and two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have that $s \leadsto_{\mathcal{R}} t$ is a *narrowing step* iff there exist

- a non-variable position p of s,
- a variant $R = (l \to r)$ of a rule in $\mathcal{R}$,
- a substitution $\sigma = \mathsf{mgu}(s|_p, l)$ which is the most general unifier of $s|_p$ and l,

and $t = (s[r]_p)\sigma$. We often write $s \leadsto_{p,R,\theta} t$ (or simply $s \leadsto_\theta t$) to make explicit the position, rule, and substitution of the narrowing step, where $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(s)}$. A *narrowing derivation* $t_0 \leadsto^*_\sigma t_n$ denotes a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$).

3 Goal-Directed Dependency Pairs

In this section, we present a goal-directed extension of the well-known *dependency pair* (DP) framework [3,11]. Our framework is *goal-directed* since only derivations starting from a given set of terms, denoted by means of an *initial goal*, are considered.

Definition 1 (Initial Goal). *Let $\mathcal{R}$ be a TRS over $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ and $t_0 = \mathsf{f}(x_1, \ldots, x_n)$ be a term. We say that t_0 is an initial goal for $\mathcal{R}$ if $\mathsf{f} \in \mathcal{D}$ is a defined function symbol and $x_1, \ldots, x_n \in \mathcal{V}$ are distinct variables.*

Intuitively speaking, an initial goal t_0 represents the set $\lceil t_0 \rceil$ of (non necessarily ground) *constructor* instances of the term t_0, i.e.,

$$\lceil t_0 \rceil \;=\; \{\; t_0\sigma \mid \sigma \text{ is a constructor substitution} \;\}$$

For instance, given the signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ with $\mathsf{f}/1 \in \mathcal{D}$ and $\mathsf{z}/0, \mathsf{s}/1 \in \mathcal{C}$, the initial goal $\mathsf{f}(x)$ represents the set $\lceil \mathsf{f}(x) \rceil = \{\mathsf{f}(x), \mathsf{f}(\mathsf{z}), \mathsf{f}(\mathsf{s}(x)), \mathsf{f}(\mathsf{s}(\mathsf{z})), \ldots\}$. In the following, we say that a set of terms T is terminating if there is no term $t_1 \in T$ such that an infinite sequence of the form $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ exists.

It is worthwhile to observe that there is no loss of generality in our notion of initial goal since any arbitrary term t could be used as an initial goal by just adding a new rule, $\mathsf{goal}(x_1, \ldots, x_n) \to t$, where goal is a fresh function symbol with $\mathcal{V}\mathsf{ar}(t) = \{x_1, \ldots, x_n\}$, and then considering $\mathsf{goal}(x_1, \ldots, x_n)$ as initial goal.

Two key ingredients of the DP approach are the notion of *dependency pairs* and that of *chains* of dependency pairs. The first notion remains unchanged in our setting. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, for each $\mathsf{f}/n \in \mathcal{F}$, we let $\mathsf{f}^\sharp/n$ be a fresh *tuple symbol*; we often write F instead of $\mathsf{f}^\sharp$ in the examples. Given a term $\mathsf{f}(t_1, \ldots, t_n)$ with $\mathsf{f} \in \mathcal{D}$, we let $t^\sharp$ denote $\mathsf{f}^\sharp(t_1, \ldots, t_n)$.

Definition 2 (Dependency Pair [3]). *Given a TRS $\mathcal{R}$ over a signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$, the associated set of dependency pairs, $DP(\mathcal{R})$, is defined as follows.*[1]

$$DP(\mathcal{R}) = \{l^\sharp \to t^\sharp \mid l \to r \in \mathcal{R},\; r|_p = t,\; \textit{and}\; \mathsf{root}(t) \in \mathcal{D}\}$$

In order to formalize our definition of chains, we first introduce the notion of *reachable calls* from a given term. Formally, given a TRS $\mathcal{R}$ and a term t, we define the set of reachable calls, $calls_{\mathcal{R}}(t)$, from t in $\mathcal{R}$ as follows:

$$calls_{\mathcal{R}}(t) = \{\; s|_p \;\mid\; t \to^*_{\mathcal{R}} s,\; \text{with } \mathsf{root}(s|_p) \in \mathcal{D} \text{ for some position } p \;\}$$

Also, given a set of terms T, we let $calls_{\mathcal{R}}(T) = \bigcup_{t \in T} calls_{\mathcal{R}}(t)$.

Definition 3 (Chain). *Let $\mathcal{R}$ and $\mathcal{P}$ be TRSs over the signatures $\mathcal{F}$ and $\mathcal{F}^\sharp$, respectively. Let t_0 be an initial goal. A (possibly infinite) sequence of pairs $s_1 \to t_1$, $s_2 \to t_2$, ... from $\mathcal{P}$ is a $(t_0, \mathcal{P}, \mathcal{R})$-chain if there is a substitution $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that the following conditions hold:*[2]

[1] Note that if $\mathcal{R}$ is a TRS, so is $DP(\mathcal{R})$.

[2] As in [3], we assume fresh variables in every (occurrence of a) dependency pair and that the domain of substitutions may be infinite.

- *there exists a term* $s \in calls_{\mathcal{R}}(\lceil t_0 \rceil)$ *such that* $s^\sharp = s_1\sigma$ *and*
- $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$ *for every two consecutive pairs in the sequence.*

The chain is minimal iff all $t_i\sigma$ *are terminating w.r.t.* $\mathcal{R}$.

Note that the only difference with the standard notion of chains is that only chains which are *reachable* from (an instance of) the initial goal are considered.

Now, without further ado, we introduce our termination criterion:

Theorem 1 (Termination Criterion). *Let* $\mathcal{R}$ *be a TRS and* t_0 *be an initial goal. All derivations starting from a term in* $\lceil t_0 \rceil$ *in* $\mathcal{R}$ *are finite iff there are no infinite minimal* $(t_0, DP(\mathcal{R}), \mathcal{R})$*-chains.*

As in the standard DP framework [11], and in order to ease the automation of the proof search, we introduce a *goal-directed* DP (GDP) framework as follows:

Definition 4 (GDP Problems and Processors). *A GDP problem is a tuple* $(t_0, \mathcal{P}, \mathcal{R}, f)$ *consisting of two TRSs* $\mathcal{R}$ *and* $\mathcal{P}$ *over the signatures* $\mathcal{F}$ *and* $\mathcal{F}^\sharp$*, respectively, an initial goal* t_0 *for* $\mathcal{R}$*, and a minimality flag* $f \in \{\mathbf{m}, \mathbf{a}\}$ *where* $\mathbf{m}$ *and* $\mathbf{a}$ *stand for "minimal" and "arbitrary", respectively. A GDP problem is finite if there is no associated infinite (minimal if* f *is* $\mathbf{m}$*)* $(t_0, \mathcal{P}, \mathcal{R})$*-chain, and infinite if it is not finite or if* $\lceil t_0 \rceil$ *does not terminate in* $\mathcal{R}$*. A (standard) DP problem is a tuple* $(\mathcal{P}, \mathcal{R}, f)$ *consisting of* $\mathcal{P}$*,* $\mathcal{R}$ *and* f *described above.*

A GDP processor *is a function Proc which takes a GDP problem and returns either a new set of GDP problems or fails. Proc is sound if for any GDP problem* $\mathcal{M}$*,* $\mathcal{M}$ *is finite whenever all GDP problems in* $Proc(\mathcal{M})$ *are finite. Proc is complete if for any GDP problem* $\mathcal{M}$*,* $\mathcal{M}$ *is infinite whenever* $Proc(\mathcal{M})$ *fails or contains an infinite GDP problem.*

Following [11], one can construct a tree whose root is labeled with the problem $(t_0, DP(\mathcal{R}), \mathcal{R}, \mathbf{m})$ and whose nodes are produced by application of sound GDP processors. If no leaf of the tree is a failure, then $\lceil t_0 \rceil$ is terminating in $\mathcal{R}$. Otherwise, if all the processors used on the path from the root to the failure node are complete, then $\lceil t_0 \rceil$ is not terminating in $\mathcal{R}$.

3.1 Dependency Graphs

The auxiliary notion of *initial pairs* denotes the pairs that match an initial goal:

Definition 5 (Initial Pairs). *Let* $(t_0, \mathcal{P}, \mathcal{R}, f)$ *be a GDP problem. The associated set of initial pairs is given by* $\{s \to t \in DP(\mathcal{R}) \mid t_0^\# \sigma = s \text{ for some subst. } \sigma\}$.

Note that the set of initial pairs associated to a GDP problem $(t_0, \mathcal{P}, \mathcal{R}, f)$ need not belong to the current set of pairs $\mathcal{P}$.

We now recall the standard notion of dependency graph [11]:

Definition 6 (Dependency Graph). *Given a GDP problem* $(t_0, \mathcal{P}, \mathcal{R}, f)$*, its dependency graph is a directed graph where the nodes are the pairs of* $\mathcal{P}$*, and there is an edge from* $s \to t \in \mathcal{P}$ *to* $u \to v \in \mathcal{P}$ *iff* $s \to t, u \to v$ *is a* $(t_0, \mathcal{P}, \mathcal{R})$*-chain.*

Although we consider the standard definition of dependency graph, since our notion of chain is different, the dependency graph of a GDP problem may contain less pairs than the dependency graph of the corresponding standard DP problem; in our case, all the pairs which are not reachable from the initial goal are removed from the dependency graph.

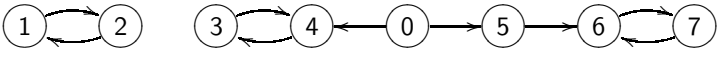
Dependency graphs are not generally computable and, thus, several approximations have been defined. Instead of adapting one of these approximations, we show how any arbitrary approach can easily be reused in our context:

Theorem 2 (Estimated Dependency Graph). *Let $(t_0, \mathcal{P}, \mathcal{R}, f)$ be a GDP problem. Let $\mathcal{G}_0$ and $\mathcal{G}$ be estimated dependency graphs (according to [11]) for the DP problems $(DP(\mathcal{R}), \mathcal{R}, f)$ and $(\mathcal{P}, \mathcal{R}, f)$, respectively.*

Assuming that the nodes of $\mathcal{G}_0$ and $\mathcal{G}$ are shared in the obvious way, the estimated dependency graph of the GDP problem $(t_0, \mathcal{P}, \mathcal{R}, f)$ is the restriction of $\mathcal{G}$ to those nodes which are reachable from an initial pair in $\mathcal{G}_0$.

The graph obtained is an over-estimation of the dependency graph.

Example 1. Consider the following dependency graph $\mathcal{G}_0$ whose nodes are labeled with $(0), (1), \ldots, (7)$:



Given a GDP problem where the only initial pair is (0), we have that pairs (1) and (2) do not belong to its dependency graph since they are not reachable.

By using the estimated dependency graph, it is immediate to define a sound GDP processor that takes a GDP problem $(t_0, \mathcal{P}, \mathcal{R}, f)$ and divides the problem into its strongly connected components (SCC) as usual. Note that, in contrast to the standard processor, we remove those SCCs which are not reachable from the initial pairs.

3.2 Usable Rules

Another way for removing pairs from $\mathcal{P}$ is based on the notion of *reduction pair* $(\succsim, \succ)$.[3] For this purpose, we first need to introduce the notion of *argument filtering* [16]. An argument filtering over a signature $\mathcal{F}$ is a function π such that, for every symbol $\mathsf{f}/n \in \mathcal{F}$, we have either $\pi(\mathsf{f}) \in \{1, \ldots, n\}$ or $\pi(\mathsf{f}) \subseteq \{1, \ldots, n\}$. Argument filterings are extended to terms as follows:[4]

- $\pi(x) = x$ for all $x \in \mathcal{V}$;
- $\pi(\mathsf{f}(t_1, \ldots, t_n)) = \pi(t_i)$ if $\pi(\mathsf{f}) = i$;
- $\pi(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\pi(t_{i_1}), \ldots, \pi(t_{i_m}))$ if $\pi(\mathsf{f}) = \{i_1, \ldots, i_m\}$ and $1 \leq i_i \leq n$.

[3] A pair of orders $(\succsim, \succ)$ is a reduction pair if $\succsim$ is a quasi-order and $\succ$ is a well-founded order where $\succsim$ is closed under contexts, and both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary) [16].

[4] By abuse of notation, we keep the same symbol for the original function and the filtered function with a possibly different arity.

Given a TRS $\mathcal{R}$, we let $\pi(\mathcal{R}) = \{\pi(l) \to \pi(r) \mid l \to r \in \mathcal{R}\}$.

For any relation $\succ$, we let $\succ_\pi$ be the relation where $t \succ_\pi u$ holds iff $\pi(t) \succ \pi(u)$. For any TRS $\mathcal{P}$ and any relation $\succ$, we let $\mathcal{P}_\succ = \{s \to t \in \mathcal{P} \mid s \succ t\}$, i.e., $\mathcal{P}_\succ$ contains those rules of $\mathcal{P}$ which decrease w.r.t. $\succ$.

Theorem 3 (Reduction Pair Processor). *Let $(\succsim, \succ)$ be a reduction pair and π be an argument filtering. Given a GDP problem $(t_0, \mathcal{P}, \mathcal{R}, f)$, if Proc returns:*

- *$(t_0, \mathcal{P} \setminus \mathcal{P}_{\succ_\pi}, \mathcal{R}, f)$, if $\mathcal{P}_{\succ_\pi} \cup \mathcal{P}_{\succsim_\pi} = \mathcal{P}$, $\mathcal{P}_{\succ_\pi} \neq \emptyset$, and $\mathcal{R}_{\succsim_\pi} = \mathcal{R}$;*
- *$(t_0, \mathcal{P}, \mathcal{R}, f)$, otherwise;*

then Proc is sound and complete.

Basically, this processor can be used to remove the strictly decreasing pairs of $\mathcal{P}$ when the remaining pairs of $\mathcal{P}$ and all rules of $\mathcal{R}$ are weakly decreasing. In fact, a weak decrease is not required for all the rules but only for the *usable* rules [12]. These rules are a superset of the rules that may be used to connect dependency pairs in a chain. For GDP problems, a notion of usable rules also removes those rules which are not reachable from the initial goal.

There are several approaches for approximating the usable rules of a problem. In this section we show how any of them can be adapted to our goal-directed setting, using as an example the usable rules of [12]. For this purpose, we first need the following auxiliary notion: given an argument filtering π and a term t, we let $RegPos_\pi(t)$ denote the *regarded positions* of t w.r.t. π; formally, $RegPos_\pi(t) = \{\epsilon\} \cup \{i.p \mid t = \mathsf{f}(t_1, \cdots, t_n), p \in RegPos_\pi(t_i), \text{ and } i \in \pi(f)\}$. In essence, the regarded positions of t w.r.t. π are those positions of t which are not dropped by the filtering π.

In the following, given a TRS $\mathcal{R}$ and symbol $\mathsf{f} \in \mathcal{F}$, we let $\mathsf{Def}_\mathcal{R}(\mathsf{f}) = \{l \to r \in \mathcal{R} \mid \mathsf{root}(l) = \mathsf{f}\}$ and $\mathcal{R}'_\mathsf{f} = \mathcal{R} \setminus \mathsf{Def}_\mathcal{R}(\mathsf{f})$.

Definition 7 (Estimated Usable Rules w.r.t. an Argument Filtering [12]). *For any TRS $\mathcal{R}$ and any argument filtering π, we define*

- *$\mathcal{U}^\pi_\mathcal{R}(x) = \emptyset$ for $x \in \mathcal{V}$.*
- *$\mathcal{U}^\pi_\mathcal{R}(\mathsf{f}(t_1, \cdots, t_n)) = \mathsf{Def}_\mathcal{R}(\mathsf{f}) \cup \bigcup_{l \to r \in \mathsf{Def}_\mathcal{R}(\mathsf{f})} \mathcal{U}^\pi_{\mathcal{R}'_\mathsf{f}}(r) \cup \bigcup_{i \in RegPos_\pi(\mathsf{f})} \mathcal{U}^\pi_{\mathcal{R}'_\mathsf{f}}(t_i)$*

For any set of rules $\mathcal{P}$ we define $\mathcal{U}^\pi_\mathcal{R}(\mathcal{P}) = \bigcup_{l \to r \in \mathcal{P}} \mathcal{U}^\pi_\mathcal{R}(r)$.

Definition 8 (Estimated Goal-Directed Usable Rules w.r.t. an Argument Filtering). *For a graph $\mathcal{G}$ and two set of nodes $\mathcal{I}$ and $\mathcal{P}$, let $PATH_\mathcal{G}(\mathcal{I}, \mathcal{P})$ denote the smallest set of nodes of $\mathcal{G}$ which contains both $\mathcal{I}$ and $\mathcal{P}$ and all the nodes which are in a path from some node in $\mathcal{I}$ to some node in $\mathcal{P}$ in $\mathcal{G}$.*

Let $(t_0, \mathcal{P}, \mathcal{R}, f)$ be a GDP problem, $\mathcal{P}_0$ its initial pairs, π an argument filtering, and $\mathcal{G}_0$ a (standard) estimated dependency graph for the DP problem $(DP(\mathcal{R}), \mathcal{R}, f)$. The goal-directed usable rules of $\mathcal{P}$ in $\mathcal{R}$ w.r.t. π and t_0 are defined as follows:

$$\mathcal{GU}(t_0, \mathcal{P}, \mathcal{R}, \pi) = \begin{cases} \mathcal{U}^\pi_\mathcal{R}(PATH_{\mathcal{G}_0}(\mathcal{P}_0, \mathcal{P})) & \text{if } \mathcal{V}\mathsf{ar}(\pi(t)) \subseteq \mathcal{V}\mathsf{ar}(\pi(s)) \\ & \text{for all } s \to t \in PATH_{\mathcal{G}_0}(\mathcal{P}_0, \mathcal{P}) \\ \mathcal{R} & \text{otherwise} \end{cases}$$

Example 2. Let us consider again the dependency graph $\mathcal{G}_0$ from Example 1, together with the set $\mathcal{I} = \{(0)\}$. Then,

- if $\mathcal{P}$ is the SCC $\{(6), (7)\}$ then $PATH_{\mathcal{G}_0}(\mathcal{I}, \mathcal{P}) = \{(0), (5), (6), (7)\}$;
- if $\mathcal{P} = \{(3), (4)\}$, then $PATH_{\mathcal{G}_0}(\mathcal{I}, \mathcal{P}) = \{(0), (3), (4)\}$.

The goal-directed usable rules coincide with the usable rules for all the chains from an initial pair to the pairs in $\mathcal{P}$. While this means that they are a superset of the usable rules of $\mathcal{P}$, they are still advantageous as they are applicable in cases where the usable rules are not. In particular, *minimality* is not required. This is critical in Section 4 where we consider the termination of narrowing as a problem of relative termination, since in this context minimality does not generally hold.

Theorem 4 (Reduction Pair Processor with Goal-Directed Usable Rules w.r.t. Argument Filterings). *Let $(\succsim, \succ)$ be a reduction pair and π be an argument filtering. Given a GDP problem $(t_0, \mathcal{P}, \mathcal{R}, f)$, if Proc returns:*

- *$(t_0, \mathcal{P} \setminus \mathcal{P}_{\succ_\pi}, \mathcal{R}, \mathbf{m})$, if f is $\mathbf{m}$, $\mathcal{P}_{\succ_\pi} \cup \mathcal{P}_{\succsim_\pi} = \mathcal{P}$, $\mathcal{R}_{\succsim_\pi} \supseteq \mathcal{U}^\pi_{\mathcal{R}}(\mathcal{P})$, and $\succsim$ is $\mathcal{C}_\mathcal{E}$-compatible,*[5]
- *$(t_0, \mathcal{P} \setminus \mathcal{P}_{\succ_\pi}, \mathcal{R}, \mathbf{a})$, if f is $\mathbf{a}$, $\mathcal{P}_{\succ_\pi} \cup \mathcal{P}_{\succsim_\pi} = \mathcal{P}$, and $\mathcal{R}_{\succsim_\pi} \supseteq \mathcal{GU}^\pi(t_0, \mathcal{P}, \mathcal{R}, \pi)$;*
- *$(t_0, \mathcal{P}, \mathcal{R}, f)$, otherwise;*

then Proc is sound and complete.

Example 3. Consider the following GDP problem

$$(\mathsf{goal}(x), \{\mathsf{ADD}(\mathsf{s}(x), y) \to \mathsf{ADD}(x, y)\}, \mathcal{R}_{add}, \mathbf{a})$$

where:

$$\mathcal{R}_{add} = \left\{ \begin{array}{rcl} \mathsf{goal}(x) & \to & \mathsf{add}(x, \mathsf{gen}) \\ \mathsf{add}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{add}(x, y)) \\ \mathsf{add}(\mathsf{zero}, y) & \to & y \\ \mathsf{gen} & \to & \mathsf{s}(\mathsf{gen}) \\ \mathsf{gen} & \to & \mathsf{zero} \end{array} \right\} \quad DP(\mathcal{R}_{add}) = \left\{ \begin{array}{rcl} \mathsf{GOAL}(x) & \to & \mathsf{ADD}(x, \mathsf{gen}) \\ \mathsf{GOAL}(x) & \to & \mathsf{GEN} \\ \mathsf{ADD}(\mathsf{s}(x), y) & \to & \mathsf{ADD}(x, y) \\ \mathsf{GEN} & \to & \mathsf{GEN} \end{array} \right\}$$

Using the default argument filtering which filters nothing, we have that

- the usable rules w.r.t. $\mathsf{goal}(x)$ include only the rules for gen.

Using the argument filtering defined as $\pi(\mathsf{add}) = \{1\}$ (*i.e.*, $\pi(\mathsf{add}(x, y)) = \mathsf{add}(x)$ and the identity otherwise), we have that

- the usable rules w.r.t. $\mathsf{goal}(x)$ are the empty set.

Since the rules of gen are increasing, the finiteness of the GDP problem can only be proved in the second case.

[5] A quasi-rewrite order $\succsim$ is *$\mathcal{C}_\mathcal{E}$-compatible* if for a new function symbol c, $\mathsf{c}(x, y) \succsim x$ and $\mathsf{c}(x, y) \succsim y$.

As this section has shown, using this notion of usable rules it is straightforward to adapt an existing reduction pair processor to the goal-directed setting. Adapting other processors to the framework is straightforward too, since every GDP chain is a DP chain and, thus, soundness is preserved as long as the processor does not introduce new pairs in the graph. In any case, discussing the details is out of the scope of this paper.

4 Goal-Directed Termination of Narrowing

In this section, we consider the termination of *narrowing* [22] and show how this problem can be reduced to proving the *relative* termination (see below) of a TRS from an initial set of terms, so that the GDP framework introduced in the previous section can be steadily applied.

4.1 Relative Termination

First, we show that it is possible to cast a relative termination problem as a standard DP problem as long as the systems involved satisfy the condition that they form hierarchical combinations.

Definition 9 (Hierarchical Combination [19]). *A system $\mathcal{R}_0 \cup \mathcal{R}_1$ is the* hierarchical combination *(HC) of a base $\mathcal{R}_0$ over $\mathcal{F}_0 = \mathcal{D}_0 \uplus \mathcal{C}_0$ and an extension $\mathcal{R}_1$ over $\mathcal{F}_1 = \mathcal{D}_1 \uplus \mathcal{C}_0$ if and only if $\mathcal{D}_0 \cap \mathcal{D}_1 = \varnothing$ and $\mathcal{C}_0 \cap \mathcal{D}_1 = \varnothing$.*

Let us now recall the notion of relative termination:

Definition 10 (Relative Termination). *Given two relations $\to_R$ and $\to_E$ we define the compound relation $\to_R/\to_E$ as $\to_E^* \cdot \to_R \cdot \to_E^*$.*

Given two TRSs $\mathcal{R}_1$ and $\mathcal{R}_0$, we say that $\mathcal{R}_1$ terminates w.r.t. $\mathcal{R}_0$ if the relation $\to_{\mathcal{R}_1}/\to_{\mathcal{R}_0}$ is terminating, i.e., if every (possibly infinite) $\to_{\mathcal{R}_0} \cup \to_{\mathcal{R}_1}$ derivation contains only finitely many $\to_{\mathcal{R}_0}$ steps.

Note that sequences of $\to_{\mathcal{R}_0}$ steps are "collapsed" and seen as a single $\to_{\mathcal{R}_1}/\to_{\mathcal{R}_0}$ step. Hence, an infinite $\to_{\mathcal{R}_1}/\to_{\mathcal{R}_0}$ derivation must contain an infinite number of $\to_{\mathcal{R}_1}$ steps, and thus by assumption only finite $\to_{\mathcal{R}_0}$ subderivations.

We say that a term t is $\to_{\mathcal{R}}$-terminating w.r.t. $\mathcal{B}$ if there is no infinite $\to_{\mathcal{R}}/\to_{\mathcal{B}}$ derivation issuing from t. Similarly, we say that T is $\to_{\mathcal{R}}$-terminating w.r.t. $\mathcal{B}$ if every term in T is $\to_{\mathcal{R}}$-terminating w.r.t. $\mathcal{B}$.

We make use of the standard notion of minimal (*non-terminating*) term, i.e., a term which starts an infinite derivation while all its proper subterms are terminating. We say that a term that is not $\to_{\mathcal{R}}$-terminating w.r.t. $\mathcal{B}$ is $\to_{\mathcal{R}}/\to_{\mathcal{B}}$-*minimal* if all its proper subterms are $\to_{\mathcal{R}}$-terminating w.r.t. $\mathcal{B}$.

Lemma 1. *Let $\mathcal{R}$ and $\mathcal{B}$ be two TRSs over $\mathcal{F}_{\mathcal{R}}$ and $\mathcal{F}_{\mathcal{B}}$ respectively, such that $\mathcal{R} \cup \mathcal{B}$ is the HC of the base $\mathcal{B}$ and the extension $\mathcal{R}$. Every $\to_{\mathcal{R}}/\to_{\mathcal{B}}$-minimal term $t_0 \in \mathcal{T}(\mathcal{F}_{\mathcal{B}} \cup \mathcal{F}_{\mathcal{R}}, \mathcal{V})$ starting an infinite $\to_{\mathcal{R}}/\to_{\mathcal{B}}$ derivation is of the form $t_0 = \mathsf{f}(\overline{u})$, where f is a defined symbol from $\mathcal{F}_{\mathcal{R}}$.*

Now we state the main result of this section. In order to prove relative termination of a TRS $\mathcal{R}$ w.r.t. a TRS $\mathcal{B}$, as long as they form an HC, one only needs to prove that the pairs of $\mathcal{R}$ are strongly decreasing, while the pairs of $\mathcal{B}$ can be ignored, even if $\mathcal{B}$ is not terminating.

Theorem 5 (Relative Termination Criterion). *Let $\mathcal{R}$ and $\mathcal{B}$ be two TRSs such that $\mathcal{R}\cup\mathcal{B}$ is the HC of the base $\mathcal{B}$ and the extension $\mathcal{R}$. Then, $\mathcal{R}$ terminates w.r.t. $\mathcal{B}$ if and only if there are no infinite $(DP(\mathcal{R}), \mathcal{R}\cup\mathcal{B})$-chains.*

Example 4. Let $\mathcal{R} = \{\mathsf{f} \to \mathsf{gen}\}$ and $\mathcal{B} = \{\mathsf{gen} \to \mathsf{f}\}$ be TRSs, which are trivially terminating. Moreover, the DP Problem $(DP(\mathcal{R}), \mathcal{R}\cup\mathcal{B}) = (\varnothing, \mathcal{R}\cup\mathcal{B})$ is trivially finite (here we assume that gen is a constructor symbol in $\mathcal{R}$, hence the set of dependency pairs $DP(\mathcal{R})$ is empty). However, $\mathcal{R}$ is not terminating w.r.t. $\mathcal{B}$ since we have the following infinite derivation: $\mathsf{f} \to \mathsf{gen} \to \mathsf{f} \to \ldots$

In contrast to the termination criterion in [9] for relative termination, the HC property is required in Theorem 5, i.e., the HC property is necessary to extend the DP framework for proving relative termination. Note also that it does not suffice to prove the absence of *minimal* chains, as the following example shows:

Example 5. Let $\mathcal{R} = \{\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(x)\}$ and $\mathcal{B} = \{\mathsf{gen} \to \mathsf{s}(\mathsf{gen})\}$. We have that $DP(\mathcal{R}) = \{\mathsf{F}(\mathsf{s}(x)) \to \mathsf{F}(x)\}$ and there are no infinite minimal chains. However there is an infinite chain with $\sigma = \{x \mapsto \mathsf{gen}\}$.

The relative termination criterion can be combined with Theorem 1 for relative termination from an initial goal.

Corollary 1 (Goal-Directed Relative Termination Criterion). *Let $\mathcal{R}$ and $\mathcal{B}$ be two TRSs such that $\mathcal{R}\cup\mathcal{B}$ is the HC of the base $\mathcal{B}$ and the extension $\mathcal{R}$. Then, all derivations starting from a term in $\lceil t_0 \rceil$ in $\mathcal{R}$ terminate w.r.t. $\mathcal{B}$ if and only if there are no infinite $(t_0, DP(\mathcal{R}), \mathcal{R}\cup\mathcal{B})$-chains.*

4.2 Termination of Narrowing via Relative Termination

Recently, [18,25] introduced a termination analysis for narrowing which is roughly based on the following process[6]. First, following [2,7], logic variables are replaced with a fresh function, called gen, which can be seen as a *data generator* that can be non-deterministically reduced to any ground (constructor) term. A first result relates the termination of narrowing in the original TRS and the *relative* termination of rewriting using occurrences of gen to replace logic variables. However, in order to avoid dealing with relative termination, [25] considers the use of an argument filtering to filter away occurrences of gen in the considered computations so that relative termination and termination coincide. Finally, termination is analyzed using the DP framework [11] for proving the termination of rewriting over the filtered terms.

[6] The termination analysis of logic programs of [20] follows a similar pattern but logic variables are replaced with *infinite* terms (the net effect, though, is similar).

This approach has several problems all related to the use of an argument filtering to filter the occurrences of gen. The most important one is that the search for an argument filtering that allows to prove termination is exponential in the arities of the signature, and even worse, this search cannot be casted as an optimization problem. This leads to the application of complex heuristics (as in [20]) which complicate the approach and diminish the effectiveness of the automation. Another issue is related to collapsing rules. Consider, for instance, a *collapsing* rule, i.e., a rule of the form $\mathsf{f}(x, y) \to y$, together with the argument filtering $\pi(\mathsf{f}) = \{1\}$. The filtered rule $\mathsf{f}(x) \to y$ contains an extra variable, y, and no refinement[7] of π will be able to eliminate it.[8]

Here, we argue that there is a better way to approach this problem. Instead of using a global argument filtering to filter away occurrences of gen, we propose to not filter them at all, and instead use the GDP framework developed in Section 3. As we show next this effectively solves the mentioned issues.

In order to formalize our approach, we first need to recall some existing notation and terminology from the literature.

Given a left-linear constructor TRS $\mathcal{R}$ over the signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$, we define the generator of $\mathcal{R}$, $\mathsf{GEN}(\mathcal{R})$, as the following set of rules:

$$\mathsf{GEN}(\mathcal{R}) \;=\; \{\;\; \mathsf{gen} \to \mathsf{c}(\overbrace{\mathsf{gen}, \ldots, \mathsf{gen}}^{n \text{ times}}) \;\;\mid\;\; \mathsf{c}/n \in \mathcal{C},\; n \geqslant 0 \;\;\}$$

Given a left-linear constructor TRS $\mathcal{R}$, we denote by $\mathcal{R}_{\mathsf{gen}}$ the set of rules resulting from augmenting $\mathcal{R}$ with $\mathsf{GEN}(\mathcal{R})$, in symbols $\mathcal{R}_{\mathsf{gen}} = \mathcal{R} \cup \mathsf{GEN}(\mathcal{R})$.

Following [25], variables are then replaced by generators in the obvious way: given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we let $\widehat{t} = t\sigma$, with $\sigma = \{x \mapsto \mathsf{gen} \mid x \in \mathcal{V}\mathsf{ar}(t)\}$. Also, given a TRS $\mathcal{R}$, possibly with *extra variables*,[9] we denote by $\widehat{\mathcal{R}}$ the result of replacing every extra variable in $\mathcal{R}$ (if any) with gen.

Note that $\widehat{t}$ is ground for any term t since all variables occurring in t are replaced by the function gen. As for $\widehat{\mathcal{R}}$, we note that it contains no extra variables by definition.

The completeness of replacing logic variables by generators is stated in [2,7]:

Lemma 2 (Completeness). *Let $\mathcal{R}$ be a left-linear constructor TRS over a signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ and $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ be a term. If $s \leadsto_{p,R,\sigma} t$ in $\mathcal{R}$, then $\widehat{s} \to^*_{\mathsf{GEN}(\mathcal{R})} \widehat{s}\sigma \to_{p,R} \widehat{t}$ in $\mathcal{R}_{\mathsf{gen}}$.*

In the following, we say that a set of terms T is $\leadsto_{\mathcal{R}}$-terminating if there is no term $t_1 \in T$ such that an infinite sequence of the form $t_1 \leadsto_{\mathcal{R}} t_2 \leadsto_{\mathcal{R}} \ldots$ exists.

In [25], the (possibly infinite) set of initial terms T was described by means of an *abstract* term $\mathsf{f}(m_1, \ldots, m_n)$, where f is a defined function symbol and $m_1, \ldots, m_n$ are either $\mathbf{g}$ (a definitely **g**round constructor term) or $\mathbf{v}$ (a possibly

[7] An argument filtering π' is a refinement of another argument filtering π if it filters the same or more arguments, i.e., either $\pi'(\mathsf{f}) = \pi(\mathsf{f})$ or $\pi'(\mathsf{f}) \subseteq \pi(\mathsf{f})$ for every f.

[8] This is not a limitation of [20] since the considered rewrite systems that are produced from the translation of logic programs never have collapsing rules.

[9] Extra variables are variables that appear in the rhs of a rule but not in its lhs.

variable constructor term). Given an abstract term t^α, we let $\gamma(t^\alpha)$ denote the set of terms that can be obtained by replacing every argument $\mathbf{g}$ with a ground constructor term and every argument $\mathbf{v}$ with any arbitrary constructor term. Then, [25] shows that the termination of narrowing can be recast in terms of the relative termination of rewriting as follows:

Theorem 6 (Termination of Narrowing). *Let $\mathcal{R}$ be a left-linear constructor TRS and t^α an abstract term. Then, $\gamma(t^\alpha)$ is $\rightsquigarrow_{\mathcal{R}}$-terminating if $\widehat{\gamma(t^\alpha)}$ is $\to_{\mathcal{R}}$-terminating relative to $\mathsf{GEN}(\mathcal{R})$.*

In the next result we apply the framework of goal-directed dependency pairs to solve this kind of problems. First, let us recall that our GDP problems consider an initial goal rather than an abstract term. So we embed the abstract goal into the TRS by means of an additional rule. To be precise, given an abstract term $t^\alpha = \mathsf{f}(t_1, \ldots, t_n)$ with m occurrences of $\mathbf{g}$, we let $goal(t^\alpha)$ be the rule

$$\mathsf{goal}(x_{j_1}, \ldots, x_{j_m}) \to \mathsf{f}(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ are (fresh) distinct variables and $j_1, \ldots, j_m$ are the positions of the $\mathbf{g}$ arguments of t^α. Given a TRS $\mathcal{R}$ and an abstract term t^α, we denote by $\mathcal{R}_{t^\alpha}$ the TRS that extends $\mathcal{R}$ with this rule; formally, $\mathcal{R}_{t^\alpha} = \mathcal{R} \cup \{goal(t^\alpha)\}$.

In other words, we replace $\mathbf{v}$ arguments of the abstract term t^α by extra variables. Extra variables occur very naturally in the context of narrowing since they behave as *free* variables which can only be instantiated to finite constructor terms. In our context they are simply replaced by occurrences of gen in $\widehat{\mathcal{R}}$.

The following result combining Theorems 1 and 6 is the basis of our termination proving method.

Theorem 7. *Let $\mathcal{R}$ be a left-linear constructor TRS (possibly with extra variables) and t^α an abstract term. Then, $\gamma(t^\alpha)$ is $\rightsquigarrow_{\mathcal{R}}$-terminating if the GDP problem $(\mathsf{goal}(x_1, \ldots, x_n), DP(\widehat{\mathcal{R}_{t^\alpha}}), \widehat{\mathcal{R}_{t^\alpha}} \cup \mathsf{GEN}(\mathcal{R}), \mathbf{a})$ is finite, where $\mathsf{goal}(x_1, \ldots, x_n)$ is the left-hand side of $goal(t^\alpha)$.*

Example 6. Consider the following TRS that is part of Example 3:

$$\mathcal{R} = \left\{ \begin{array}{l} \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \\ \mathsf{add}(\mathsf{zero}, y) \to y \end{array} \right\}$$

For the abstract term $t^\alpha = \mathsf{add}(\mathbf{g}, \mathbf{v})$, the initial GDP problem $(\mathsf{goal}(x), DP(\widehat{\mathcal{R}_{t_\alpha}}), \widehat{\mathcal{R}_{t_\alpha}} \cup \mathsf{GEN}(\mathcal{R}), \mathbf{a})$ is reduced to the finite GDP problem in Example 3. Therefore, $\gamma(\mathsf{add}(\mathbf{g}, \mathbf{v}))$ is $\rightsquigarrow_{\mathcal{R}}$-terminating.

Example 7 ([20]). Consider the TRS and its associated set of pairs:

$$\left(\begin{array}{rl} \mathsf{p_{in}}(\mathsf{g}(X)) & \to \mathsf{u_3}(\mathsf{p_{in}}(X), X) \\ \mathsf{p_{in}}(X) & \to \mathsf{u_1}(\mathsf{q_{in}}(\mathsf{f}(Y)), X) \\ \mathsf{q_{in}}(\mathsf{g}(Y)) & \to \mathsf{q_{out}}(\mathsf{g}(Y)) \\ \mathsf{u_1}(\mathsf{q_{out}}(\mathsf{f}(Y)), X) & \to \mathsf{u_2}(\mathsf{p_{in}}(Y), X, Y) \\ \mathsf{u_2}(\mathsf{p_{out}}(Y), X, Y) & \to \mathsf{p_{out}}(X) \\ \mathsf{u_3}(\mathsf{p_{out}}(X), X) & \to \mathsf{p_{out}}(\mathsf{g}(X)) \end{array} \quad \begin{array}{rl} \mathsf{P_{in}}(\mathsf{g}(X)) & \to \mathsf{U_3}(\mathsf{p_{in}}(X), X) \\ \mathsf{P_{in}}(\mathsf{g}(X)) & \to \mathsf{P_{in}}(X) \\ \mathsf{P_{in}}(X) & \to \mathsf{U_1}(\mathsf{q_{in}}(\mathsf{f}(Y)), X) \\ \mathsf{P_{in}}(X) & \to \mathsf{Q_{in}}(\mathsf{f}(Y)) \\ \mathsf{U_1}(\mathsf{q_{out}}(\mathsf{f}(Y)), X) & \to \mathsf{U_2}(\mathsf{p_{in}}(Y), X, Y) \\ \mathsf{U_1}(\mathsf{q_{out}}(\mathsf{f}(Y)), X) & \to \mathsf{P_{in}}(Y) \\ \mathsf{GOAL}(X) & \to \mathsf{P_{in}}(X) \end{array} \right)$$

where the last pair is the initial pair, added to model the abstract goal $\mathsf{p_{in}}(\mathbf{g})$. A similar rule is implicitly added to $\mathcal{R}$, together with the rules for gen. Also, note the presence of extra variables, which in our approach would be replaced by calls to gen. An estimation of the dependency graph can detect that the only SCC is the pair $\mathsf{P_{in}}(\mathsf{g}(X)) \to \mathsf{P_{in}}(X)$, and that there is only the trivial path from the initial pair to this pair which includes both. As the goal-directed usable rules are the empty set, it is trivial to find an RPO that orients this pair and solves the termination problem. On the other hand, the technique of [20] needs a global argument filtering that removes every extra variable, which ultimately has to filter either the argument of $\mathsf{P_{in}}$ or g, precluding a successful termination proof. The same remark applies to [18,25] after filtering out the extra variables.

5 Results and Discussion

The technique for proving termination of narrowing introduced in the previous section is not directly comparable to [18,25] due to the loss of minimality. This means that many desirable techniques, such as the subterm criterion of [14], cannot be applied without restrictions. The same remarks apply when comparing our new approach to the infinitary rewriting framework of [20], which also employs a global argument filtering and heuristics.

In order to see how well the new approach behaves, we benchmark it versus [25,18] and [20]. We emply a set of examples generated from the LP category of the Termination Problem Database (TPDB) 5.0 [17] using the transform of [20], minus those examples containing cuts, impure primitives or arithmetic. A huge number of techniques for termination have been developed in the recent years. In order to provide a fair comparison we focus on a small set of these: the dependency graph processor, the RPO reduction pair implemented by means of the SAT encoding of [5,21,6] extended to account for our notion of goal-directed usable rules, the subterm criterion of [14] and the narrowing and instantiation graph refinement processors. The (very fast) subterm criterion processor is included to illustrate the shortcomings of losing minimality.

We have implemented our approach in the termination tool *Narradar*, available at `http://safe-tools.dsic.upv.es/narradar`. The tool recognizes the TPDB format [17] with extensions for expressing initial goals and narrowing. In order to perform the test Narradar was extended to implement the approach of [18,25] and [20]. For [18,25] Narradar uses a simple heuristic which always filters the innermost position. Also, although this approach does not consider extra variables, we filter them from the initial problem assuming that this is safe. For [20] Narradar uses the unbounded positions heuristic which does a type analysis of the logic program and computes an optimal heuristic.

All the problems were run on a 2.5Ghz Intel CPU with a 60 seconds timeout. The results are displayed in the table below, where the new approach is the best performer overall, even though the average success time is higher. It is easy to see why: the RPO constraints generated are slightly more complex, and the fast subterm criterion processor cannot be employed. There were four examples

which Narradar failed to solve where the other techniques succeeded. These fall into two categories: the incompleteness of the generator approach due to the problem of admissible derivations [25] (e.g. `SGST06/toyama.pl`), or the inability of the RPO processor to solve a given problem where the subterm criterion succeeds easily (e.g. `SGST06/prime.pl`). While the former is an intrinsic limitation of the approach, the latter can be fixed by means of more powerful termination processors. The full results, including the proofs generated by Narradar, are available at `http://www.dsic.upv.es/~gvidal/lopstr09`. Let us remark that the results for [20] must be regarded as orientative only, for it is in fact a technique for the termination of logic programs.

	Successes	Failures	Timeouts	Average success time
Narradar	183	113	9	1.67 seconds
[25]	167	122	16	0.33 seconds
[20]	178	111	16	0.29 seconds

Future work. Although the new technique outperforms the state of the art, there is still ample room for improvement ahead. We could pursue the approach of [2] and replace every extra variable by the generator of the terms it can get instantiated to, by means of a previous static analysis step. Moreover, minimality could be recovered under some conditions. Finally, although we have defined our method for left-linear constructor systems, it remains to be seen whether this restriction can be lifted using the LL-DPs of [1].

Acknowledgements. The first author is in debt to Peter Schneider-Kamp for many insightful discussions during a research visit to South Denmark. We also thank Raúl Gutiérrez for his numerous relevant remarks on a draft of this paper, and Nao Hirokawa for his insights on relative termination.

References

1. Alpuente, M., Escobar, S., Iborra, J.: Termination of Narrowing Using Dependency Pairs. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 317–331. Springer, Heidelberg (2008)
2. Antoy, S., Hanus, M.: Overlapping Rules and Logic Variables in Functional Logic Programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 87–101. Springer, Heidelberg (2006)
3. Arts, T., Giesl, J.: Termination of Term Rewriting Using Dependency Pairs. Theoretical Computer Science 236(1-2), 133–178 (2000)
4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
5. Codish, M., Lagoon, V., Stuckey, P.J.: Solving partial order constraints for lpo termination. CoRR, abs/cs/0512067 (2005)
6. Codish, M., Schneider-Kamp, P., Lagoon, V., Thiemann, R., Giesl, J.: Sat solving for argument filterings. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 30–44. Springer, Heidelberg (2006)

7. de Dios-Castro, J., López-Fraguas, F.: Extra Variables Can Be Eliminated from Functional Logic Programs. In: Proc. of the 6th Spanish Conf. on Programming and Languages (PROLE 2006), ENTCS, vol. 188, pp. 3–19 (2007)
8. Dershowitz, N.: Termination of Rewriting. Journal of Symbolic Computation 3(1,2), 69–115 (1987)
9. Geser, A.: Relative termination. Dissertation, Fakultät für Mathematik und Informatik, Universität Passau, Germany (1990)
10. Giesl, J., Swiderski, S., Schneider-Kamp, P., Thiemann, R.: Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 297–312. Springer, Heidelberg (2006)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
12. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and Improving Dependency Pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
13. Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. Journal of Logic Programming 19,20, 583–628 (1994)
14. Hirokawa, N., Middeldorp, A.: Dependency pairs revisited. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 249–268. Springer, Heidelberg (2004)
15. Iborra, J., Nishida, N., Vidal, G.: Goal-directed Dependency Pairs and its Application to Proving the Termination of Narrowing (2010), http://users.dsic.upv.es/~gvidal/german/papers.html
16. Kusakari, K., Nakamura, M., Toyama, Y.: Argument Filtering Transformation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 48–62. Springer, Heidelberg (1999)
17. Marche, C., Zantema, H.: The termination competition. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 303–313. Springer, Heidelberg (2007)
18. Nishida, N., Vidal, G.: Termination of Narrowing via Termination of Rewriting, Submitted for publication (2009)
19. Ohlebusch, E.: Advanced topics in term rewriting. Springer-Verlag, UK (2002)
20. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 177–193. Springer, Heidelberg (2007)
21. Schneider-Kamp, P., Thiemann, R., Annov, E., Codish, M., Giesl, J.: Proving termination using recursive path orders and sat solving. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 267–282. Springer, Heidelberg (2007)
22. Slagle, J.R.: Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. Journal of the ACM 21(4), 622–642 (1974)
23. Steinbach, J.: Simplification Orderings: Histrory of Results. Fundamenta Informaticae 24(1/2), 47–87 (1995)
24. Urbain, X.: Modular & incremental automated termination proofs. Int. Journal of Approx. Reasoning 32(4), 315–355 (2004)
25. Vidal, G.: Termination of Narrowing in Left-Linear Constructor Systems. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 113–129. Springer, Heidelberg (2008)

LP with Flexible Grouping and Aggregates Using Modes

Marcin Czenko[1] and Sandro Etalle[2]

[1] Department of Computer Science
University of Twente, The Netherlands
marcin.czenko@utwente.nl
[2] Eindhoven University of Technology and University of Twente, The Netherlands
s.etalle@tue.nl

Abstract. We propose a new grouping operator for logic programs based on the *bagof* predicate. The novelty of our proposal lies in the use of modes, which allows us to prove properties regarding groundness of computed answer substitutions and termination. Moreover, modes allow us to define a somewhat declarative semantics for it and to relax some rather unpractical constraints on variable occurrences while retaining a straightforward semantics.

Keywords: Grouping in Logic Programs, Moded Logic Programming, Stratified Logic Programs, Termination of Logic Programs.

1 Introduction

In a system designed to answer queries (be it a database or a logic program), an aggregate function is designed to be carried out on the set of answers to a given query rather than on a single answer. For example, in a Datalog program containing one entry per employee, one needs aggregate functions to compute data such as the average age or salary of the employee, the number of employees etc.

Grouping and aggregation are useful in practice, and paramount in database systems. In fact, the reason why we address the problem here is of a practical nature: we are developing a language for trust management [5,7,18] called TuLiP [8,9,10]. TuLiP is based on (partially function-free) *moded* logic programming, in which a logic program is augmented with an indication of which are the input and the output positions of each predicate. Modes allow to prove program properties such as groundness of answers and termination for those programs which respect them (also called *well-moded* programs) [2]. The problem we faced is the following: in order to write reputation-based rules within TuLiP, we must extend it in such a way that it allows statements such as "employee X will be granted access to confidential document Y provided that the majority of senior executives recommends him", which require the use of grouping and aggregation.

To realise aggregates in logic programming, there are two possible approaches. In the first approach, grouping and aggregation is implemented as one atomic operation. This is equivalent to having aggregates as built ins. In the second one, one first calls a *grouping* query (like bagof), and then computes the aggregate on the result of the grouping. We prefer this second approach for a number of reasons: first, grouping queries are

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 67–81, 2010.

interesting on their own, especially in Trust Management where sometimes we need to query a specific subset of entities without performing any aggregate operation; secondly, by separating grouping from aggregation one can use the same data set for different aggregate operations.

So, basically, what we need then is something similar to the well-known *bagof* predicate, which, however, is not suitable for our purposes for two reasons: first, it is not moded and – being a higher-order predicate – there is no straightforward way to associate a mode to it; secondly, it imposes a somewhat restrictive condition on variable occurrences which can be circumvented, but at the cost of using an ugly construction.

The basic contribution of this paper is the definition and the study of the properties of a new grouping predicate *moded_bagof*, which can be seen as a moded counterpart of *bagof*. We show that – in presence of well-moded programs – *moded_bagof* enjoys the usual properties of moded predicates, namely groundness of c.a. substitutions and (under additional conditions) termination. Moreover, modes allow to lift the restrictive condition on variable sharing we mentioned before. As we will see – assigning modes to *moded_bagof* is not trivial, as it depends on the mode of the subgoal it contains.

We define the semantics of *moded_bagof* in terms of computed answer substitutions. We tried to be precise while avoiding to resort to higher order theories. We succeeded but only to some extent: a disadvantage of having grouping and aggregation as separate operations is that in order to be able to define fully declarative semantics for grouping, one needs to extend the language with set-based primitives like *set membership* ($\in$) or set-equation ($=$). This is a not trivial task and significant work in this area has been carried out (see Section Related Work). Alternatively, one can use a more practical approach and use a list as a representation of a multiset. Because a list is not a multiset (two lists with different order of the elements are two different lists), the declarative semantics cannot be precise in this case.

The paper is structured as follows. In Section 2 we present the preliminaries on Logic Programming and notational conventions used in this paper. In Section 3 we state the basic facts about well-moded logic programs. In Section 4 we show how to do grouping in Prolog and we define our own grouping atom *moded_bagof*. In Section 5 we show an operational semantics of *moded_bagof* by defining the computed answer substitutions for programs that do not contain grouping subgoals. In Section 6 we show how to use *moded_bagof* in programs containing grouping subgoals. Here we generalise the notion of well-moded logic programs to those including grouping subgoals. In Section 7 we discuss the properties of the well-moded programs containing grouping atoms. In particular, we prove two important properties: groundness of computed answer substitutions and termination. The paper finishes with Related Work in Section 8 and Conclusions in Section 9.

2 Preliminaries on Logic Programming (without Grouping)

In what follows we study definite logic programs executed by means of LD-resolution, which consists of the SLD-resolution combined with the leftmost selection rule. The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1]. We use boldface to denote sequences of objects; therefore $\mathbf{t}$ denotes a sequence of terms while $\mathbf{B}$ is a sequence of atoms (i.e. a query). We

denote atoms by $A, B, H, \ldots$, queries by $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P. For any atom A, we denote by $Pred(A)$ the predicate symbol of A. For example, if $A = p(a, X)$, then $Pred(A) = p$. The empty query is denoted by $\Box$ and the set of clauses defining a predicate is called a *procedure*.

For any syntactic object (e.g., atom, clause, query) o, we denote by $Var(o)$ the set of variables occurring in o. Given a *substitution* $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, ..., t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If, $t_1, ..., t_n$ is a permutation of $x_1, ..., x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that an syntactic object (e.g., an atom) o is an *instance* of o' iff for some σ, $o = o'\sigma$, further o is called a *variant* of o', written $o \approx o'$ iff o and o' are instances of each other. A substitution θ is a *unifier* of objects o and o' iff $o\theta = o'\theta$. We denote by $mgu(o, o')$ any *most general unifier* (*mgu*, in short) of o and o'.

(LD) Computations are sequences of LD derivation steps. The non-empty query $q : B, \mathbf{C}$ and the clause $c : H \leftarrow \mathbf{B}$ (renamed apart wrt q) yield the resolvent $(\mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $B, \mathbf{C} \stackrel{\theta}{\Longrightarrow}_c (\mathbf{B}, \mathbf{C})\theta$. c is called its *input clause*. A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := \mathbf{B}_0 \stackrel{\theta_1}{\Longrightarrow}_{c_1} \mathbf{B}_1 \stackrel{\theta_2}{\Longrightarrow}_{c_2} \cdots \mathbf{B}_n \stackrel{\theta_{n+1}}{\Longrightarrow}_{c_{n+1}} \mathbf{B}_{n+1} \cdots$ of derivation steps is called an *LD derivation of* $P \cup \{\mathbf{B}_0\}$ provided that for every step the standardisation apart condition holds, i.e., the input clause employed at each step is variable disjoint from the initial query $\mathbf{B}_0$ and from the substitutions and the input clauses used at earlier steps. If the program P is clear from the context and the clauses $c_1, \ldots, c_{n+1}, \ldots$ are irrelevant, then we drop the reference to them. If δ is maximal and ends with the empty query ($\mathbf{B}_n = \Box$) then the restriction of θ to the variables of $\mathbf{B}$ is called its *computed answer substitution* (*c.a.s.*, for short). The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ.

A *multiset* is a collection of elements that are not necessarily distinct [19]. The number of occurrences of an element x in a multiset M is its *multiplicity* in the multiset, and is denoted by $mult(x, M)$. When describing multisets we use the notation that is similar to that of the sets, but instead of $\{$ and $\}$ we use $[\![$ and $]\!]$ respectively.

3 Well-Moded Logic Programs

Informally speaking, a *mode* indicates how the arguments of a relation should be used, i.e. which are the input and which are the output positions of each atom, and allow one to derive properties such as absence of run-time errors for Prolog built-ins, or absence of floundering for programs with negation [2].

Definition 1 (Mode). *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.*

If $m_p(i) = In$ (resp. *Out*), we say that i is an *input* (resp. *output*) *position of* p (with respect to m_p). We assume that each predicate symbol has a *unique mode* associated to it; multiple modes may be obtained by simply renaming the predicates. We use the notation $(X_1, \ldots, X_n)$ to indicate the mode m in which $m(i) = X_i$. For instance, (In, Out)

indicates the mode in which the first (resp. second) position is an input (resp. output) position. To benefit from the advantage of modes, programs are required to be *well-moded* [2], which means that they have to respect some correctness conditions relating the input arguments to the output arguments. We denote by $In(A)$ (resp. $Out(A)$) the sequence of terms filling in the input (resp. output) positions of A, and by $VarIn(A)$ (resp. $VarOut(A)$) the set of variables occupying the input (resp. output) positions of A.

Definition 2 (Well-Moded). *A clause* $H \leftarrow B_1, \ldots, B_n$ *is* well-moded *if for all* $i \in [1, n]$

$$VarIn(B_i) \subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and}$$
$$VarOut(H) \subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H).$$

A query $\mathbf{A}$ *is* well-moded *iff the clause* $H \leftarrow \mathbf{A}$ *is well-moded, where* H *is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following lemma, due to [2], shows the "persistence" of the notion of well-modedness.

Lemma 1. *An LD-resolvent of a well-moded query and a well-moded clause that is variable-disjoint with it, is well-moded.* □

As a consequence of Lemma 1 we have the following well-known properties. For the proof we refer to [4].

1. Let P be a well-moded program and $\mathbf{A}$ be a well-moded query. Then for every computed answer σ of $\mathbf{A}$ in P, $\mathbf{A}\sigma$ is ground.
2. Let $H \leftarrow B_1, \ldots, B_n$ be a clause in a well-moded program P. If A is a well-moded atom such that $\gamma_0 = mgu(A, H)$ and for every $i \in [1, j], j \in [1, n-1]$ there exists a successful LD derivation $B_i\gamma_0, \ldots, \gamma_{i-1} \xrightarrow{\gamma_i}_P \square$ then $B_{j+1}\gamma_0, \ldots, \gamma_j$ is a well-moded atom.

4 Grouping in Prolog

Prolog already provides some grouping facilities in terms of the built-in predicate *bagof*. The *bagof* predicate has the following form:

$$bagof(Term, Goal, List).$$

Term is a prolog term (usually a variable), *Goal* is a callable Prolog goal, and *List* is a variable or a Prolog list. The intuitive meaning of *bagof* is the following: unify *List* with the list (unordered, duplicates retained) of all instances of *Term* such that *Goal* is satisfied. The variables appearing in *Term* are *local* to the *bagof* predicate and must not appear elsewhere in a clause or a query containing *bagof*[1]. If there are free variables

[1] This is the condition on variable sharing we mentioned in the introduction; it is not problematic as it can be circumvented as follows: consider the goal $bagof(p(X, Y), q(X, Y, Z), W)$, if X occurs elsewhere in the clause or the query containing this goal then one should rewrite it as *bagof* (T,(T=p(X,Y),q(X,Y,Z)),W).

in *Goal* not appearing in *Term*, *bagof* can be re-satisfied generating alternative values for *List* corresponding to different instantiations of the free variables in *Goal* that do not occur in *Term*. The free variables in *Goal* not appearing in *Term* become therefore grouping variables. By using existential quantification, one can force a variable in *Goal* that does not appear in *Term* to be treated as local.

Let us look at some examples of grouping using the *bagof* predicate.

Example 1. Consider program P consisting of the following four ground atoms: `p(a,1)`, `p(a,2)`, `p(b,3)`, `p(b,4)`. Now, query Q = `bagof(Y,p(Z,Y),X)` receives the following two answers: (1) {`X/[1,2],Z/a`} and (2) {`X/[3,4],Z/b`}. Here, because `Z` is an uninstantiated free variable, `bagof` treats `Z` as a grouping variable and `Y` as a local variable. Thus, for each ground instance of `Z`, such that there exists a value of `Y` such that `p(Z,Y)` holds, `bagof` returns a list `X` containing all instances of `Y`. In this case `bagof` returns two lists: the first containing all instances of `Y` such that `p(a,Y)` holds, the second containing all instances of `Y` such that `p(b,Y)` holds. In the query above `Y` is a local variable. If we also want to make `Z` local, then we have to explicitly use existential quantification for `Z`. The query becomes Q = `bagof(Y,Z^p(Z,Y),X)` and there is only one answer {`X/[1,2,3,4]`}. Now both `Y` and `Z` are local: `Y` because it appears in *Term*, `Z` because it is explicitly existentially quantified.

In TuLiP, we use modes to guide the credential distribution and discovery and to guarantee groundness of the computed answer substitutions for the queries. Because we want to state the groundness and termination results also for the programs containing grouping atoms, we need a moded version of *bagof*. Therefore we introduce *moded_bagof*, which a syntactical variant of *bagof* and is moded. We decided to use a slightly different syntax for *moded_bagof* comparing to that of the original *bagof* built-in. First of all we want to make grouping variables explicit in the notation. Secondly, we want to eliminate the need of using the existential quantification for making some of the variables local in the grouping atom. By using different notation we can simplify the definition of local variables in the grouping atom which makes the presentation easier to follow.

Definition 3. *A* grouping atom *moded_bagof is an atom of the form:*

$$A = \mathit{moded\_bagof}(t, gl, \mathit{Goal}, x)$$

where t is a term, gl is a list of distinct variables each of which appears in Goal, Goal is an atomic query (but not a grouping atom itself), and x is a free variable.

The *moded_bagof* grouping atom has similar semantics to that of *bagof*, with one exception: the original *bagof* fails if *Goal* has no solution while *moded_bagof* returns an empty list (in other words *moded_bagof* never fails).

Definition 3 requires that *Goal* is atomic. This simplifies the treatment (in particular the treatment of modes) and is not a real restriction, as one can always define new predicates to break down a nested grouping atom into a number of grouping atoms that satisfy Definition 3.

Example 2. Consider again the program from Example 1. The *moded_bagof* equivalent for the query `bagof(Y,p(Z,Y),X)` is `moded_bagof(Y,[Z],p(Z,Y),X)` and for the query `bagof(Y,Z^p(Z,Y),X)` it is `moded_bagof(Y,[],p(Z,Y),X)`.

5 Semantics of Atomic *moded_bagof* Queries

Before investigating the use of *moded_bagof* atoms as subgoals in programs, in this section we first look more closely at *moded_bagof* atomic queries in combination with programs in which *moded_bagof* atoms themselves do not occur. This way we can focus on the semantics of *moded_bagof* without being immediately distracted with the problems related to the termination of logic programs containing *moded_bagof* atoms as subgoals.

A subtle difficulty in providing a reasonable semantics for *moded_bagof* is due to the fact that we have to take into consideration the multiplicity of answers. In a typical situation, *moded_bagof* will be used to compute e.g. averages, as in the query `moded_bagof(W,[Y],p(Y,W),X), average(X,Z)`. To this end, `X` should actually be instantiated to a *multiset* of terms corresponding to the answers of the query `p(Y,W)`. A number of researchers investigated the problem of incorporating sets into a logic programming language (see Related Work for an overview). Here, we follow a more practical approach and we represent a multiset with a Prolog list. The disadvantage of using a list is that it is order-dependent: by permuting the elements of a list one can obtain a different list. In the (natural) implementation, given the query *moded_bagof*$(t, gl, Goal, x)$, the c.a.s. will instantiate x to a list of elements, the order of which is dependent on the order with which the computed answer substitutions to the query *Goal* are computed. This depends in turn on the order of the clauses in the program. This means that we cannot provide the declarative semantics for our *moded_bagof* construct unless we introduce multisets as first-class citizens of the language.

The fact that we are unable to give fully declarative semantics of *moded_bagof* does not prevent us from proving important properties of groundness of the computed answer substitutions and termination of programs containing grouping atoms. Below, we define the computed answer substitution to *moded_bagof* for two cases: in the first case we assume that multisets of terms are part of the universe of discourse and that a multiset operator $[\![\ ]\!]$ is available, while in the second case we resort to ordinary Prolog lists. The disadvantage of using lists is that they are order-dependent, and that if a multiset contains two or more different elements, then there exists more than one list "representing" it. Here we simply accept this shortcoming and tolerate the fact that, in real Prolog programs, the aggregating variable x will be instantiated to one of the possible lists representing the multiset of answers.

Definition 4 (c.a.s. to *moded_bagof* (Using Multisets and Prolog Lists)). *Let P be a program, and $A = moded\_bagof(t, gl, Goal, x)$ be a query. The multiset $[\![\alpha_1, \ldots, \alpha_k]\!]$ of computed answer substitutions of $P \cup A$ is defined as follows:*

1. *Let $\Sigma = [\![\sigma_1, \ldots, \sigma_n]\!]$ be the multiset of c.a.s. of $P \cup Goal$.*
2. *Let $\Sigma_1, \ldots \Sigma_k$ be a partitioning of Σ such that two answers σ_i and σ_j belong to the same partition iff $gl\sigma_i = gl\sigma_j$,*

3. ***(Multisets)*** *For each Σ_i, let ts_i be the multiset of terms obtained by instantiating t with the substitutions σ_i in Σ_i, i.e. $ts_i = [\![t\sigma_i \mid \sigma_i \in \Sigma_i]\!]$, and let $gl_i = gl\sigma$ where σ is any substitution from Σ_i.*
3. ***(Prolog Lists)*** *For each $i \in [1,k]$, let Δ_i be an ordering on Σ_i, i.e. a list of substitutions containing the same elements of Σ_i, counting multiplicities. Then, for each $\Delta_i = [\sigma_{i_1}, \ldots, \sigma_{i_m}]$, let ts_i be the* list *of terms obtained by instantiating t with the substitutions in Δ_i, i.e. $ts_i = [t\sigma_{i_1}, \ldots, t\sigma_{i_m}]$, and let $gl_i = gl\sigma$ where σ is any substitution from Δ_i.*
4. *For $i \in [1,k]$, α_i is the substitution $\{gl/gl_i, x/ts_i\}$.*

Example 3. Let P be a program containing the following facts: `p(a,c,1)`, `p(a,d,1)`,`p(a,e,3)`,`p(b,c,2)`,`p(b,d,2)`,`p(b,e,4)`.
Let $A =$ `moded_bagof(Z,[Y],p(Y,W,Z),X)`. Then $P \cup A$ yields the following two *c.a.s.*: $\alpha_1 = \{$`Y/a,X/[[1,1,3]]`$\}$ and $\alpha_2 = \{$`Y/b,X/[[2,2,4]]`$\}$. If, instead of multisets, we use Prolog lists we simply have: $\alpha_1 = \{$`Y/a,X/[1,1,3]`$\}$ and $\alpha_2 = \{$`Y/b,X/[2,2,4]`$\}$.

Since Prolog does not support multisets directly, in the sequel we use lists. In order to bring Definition 4 into practice, i.e. to really compute the answer to a query *moded_bagof*$(t, gl, Goal, x)$, we have to require that $P \cup Goal$ terminates.

6 Using *moded_bagof* in Queries and Programs

Because we want to use grouping in our trust management system TuLiP [10,9], we want to be able to use grouping not only in queries but also as subgoals in programs. In this section we discuss the use of *moded_bagof* in programs. In particular, we show how to use modes and the program stratification to guarantee groundness of computed answer substitutions and termination. Termination is of the key importance in any trust management system, especially when the credentials are distributed. In TuLiP, we use modes to guide credential storage and discovery and to prove the soundness and the completeness of TuLiP's Lookup and Inference AlgoRithm (LIAR).

We begin with the definition of a mode of the *moded_bagof* atom.

Modes. The mode of a query *moded_bagof*$(t, gl, Goal, x)$ depends on the mode of the *Goal*, so it is not fixed *a priori*. In addition, we introduce the concept of a *local variable*.

Definition 5. *Let $A =$ moded_bagof$(t, gl, Goal, x)$. We define the following sets of* input, output *and* local *variables for A:*

- $VarIn(A) = VarIn(Goal)$,
- $VarOut(A) = (Var(gl) \setminus VarIn(A)) \cup \{x\}$,
- $VarLocal(A) = Var(A) \setminus (VarIn(A) \cup VarOut(A))$,

For example, let $A =$ `moded_bagof(q(W,Y,Z),[Y],p(W,Y,Z),X)` be an aggregate atom, and assume that the original mode of `p` is (In, Out, Out). Then, $VarIn(A) =$ {`W`}, $VarOut(A) =$ {`X`,`Y`}, and $VarLocal(A) =$ {`Z`}.

Now, we can extend the definition of well-moded programs to take into consideration *moded_bagof* atoms; the only extra care we have to take is that local variables should not appear elsewhere in the clause (or query).

Definition 6 (Well-Moded-Extended). *We say that the clause* $H \leftarrow B_1, \ldots, B_n$ *is* well-moded *if for all* $i \in [1, n]$

$$VarIn(B_i) \subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and}$$
$$VarOut(H) \subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H).$$

and $\forall B_i \in \{B_1, \ldots, B_n\}$

$$VarLocal(B_i) \cap \left(\bigcup_{j \in \{1,\ldots,i-1,i+1,\ldots,n\}} Var(B_j) \cup Var(H) \right) = \emptyset.$$

A query $\mathbf{A}$ *is* well-moded *iff the clause* $H \leftarrow \mathbf{A}$ *is well-moded, where* H *is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

LD Derivations with Grouping. We extend the definition of LD-resolution to queries containing *moded_bagof* atoms.

Definition 7 (LD-Resolvent with Grouping). *Let* P *be a program. Let* $\rho : B, \mathbf{C}$ *be a query. We distinguish two cases:*

1. *if* B *is a moded_bagof atom and* α *is a c.a.s. for* B *in* P *then we say that* $B, \mathbf{C}$ *and* P *yield the resolvent* $\mathbf{C}\alpha$*. The corresponding derivation step is denoted by* $B, \mathbf{C} \overset{\alpha}{\Longrightarrow}_P \mathbf{C}\alpha$*.*
2. *if* B *is a regular atom and* $c : H \leftarrow \mathbf{B}$ *is a clause in* P *renamed apart wrt* ρ *such that* H *and* B *unify with* $mgu\ \theta$*, then we say that* ρ *and* c *yield resolvent* $(\mathbf{B}, \mathbf{C})\theta$*. The corresponding derivation step is denoted by* $B, \mathbf{C} \overset{\theta}{\Longrightarrow}_c (\mathbf{B}, \mathbf{C})\theta$*.*

As usual, a maximal sequence of derivation steps starting from query $\mathbf{B}$ *is called an* LD derivation of $P \cup \{\mathbf{B}\}$ *provided that for every step the standardisation apart condition holds.* □

Example 4. In a company, there is a policy that a confidential project document can be read by any employee recommended by majority of senior executives of one of the project partners. When using *moded_bagof*, such a policy can be modeled by the following two rules:

```
read_document(company,X) :- partner(company,P),
      moded_bagof(Y1,[],senior(P,Y1),Z1),
      moded_bagof(Y2,[X],senior_recommends(P,Y2,X),Z2),
      length(Z1,L1), length(Z2,L2), L2 > L1/2.
senior_recommends(P,X,Y) :- senior(P,X),recommends(X,Y).
```

In TuLiP, the first rule is called a credential, the second rule is a user-defined constraint [8]. Assume that there exist the following credentials:

```
partner(company,company).          senior(partnerA,sandro).
partner(company,partnerA).         senior(partnerA,mark).
partner(company,partnerB).         senior(partnerA,pieter).
partner(company,partnerC).         senior(partnerA,john).
recommends(sandro,marcin).         recommends(pieter,marcin).
recommends(john,marcin).
```

Now, given the query `read_document(company,X)`, one expects to receive {`X/marcin`} as the only c.a.s. Indeed, the answers for the two `moded_bagof(...)` subgoals are {`Z1/[sandro,mark,pieter,john]`} for the first one and {`X/marcin, Z2/[sandro,pieter,john]`} for the second.

Notice the importance of the correct discovery of the credentials. For instance, if one of the `recommends(...)` credentials is not found, the query would fail, which means that `marcin` would not be able to read the document even though he has sufficient permissions. One of the things we try to handle in TuLiP [8,9,10] is where to store the credentials so that they can be found later during the credential discovery. If we assume that *mode*(`read_document`) = *mode*(`partner`) = *mode*(`senior`) = *mode*($recommends$) = (In, Out) and *mode*(`senior_recommends`) = (In, Out, Out) then, by the credential storage principles of TuLiP, all the credentials and the user-defined constraint will be stored by their issuers (indicated by the first argument of a credential atom). For this storage configuration, TuLiP's Lookup and Inference AlgoRithm (LIAR) is guaranteed to find all relevant credentials.

7 Properties

There are two main properties we can prove for programs containing grouping atoms: groundness of computed answer substitutions and – under additional constraints – termination.

Groundness. Well-moded *moded_bagof* atoms enjoy the same features as regular well-moded atoms. The following lemma is a natural consequence of Lemma 1.

Lemma 2. *Let P be a well-moded program and A = moded_bagof$(t, gl, Goal, x)$ be a grouping atom in which gl is a list of variables. Take any ground σ such that $Dom(\sigma) = VarIn(A)$. Then each c.a.s. θ of $P \cup A\sigma$ is ground on A's output variables, i.e. $Dom(\theta) = VarOut(A)$ and $Ran(\theta) = \emptyset$.*

Proof. By noticing that $VarIn(A) = VarIn(Goal)$ and that each variable in the grouping list gl appears in *Goal*, the proof is a straightforward consequence of Lemma 1. □

Termination. Termination is particularly important in the context of grouping queries, because if *Goal* does not terminate (i.e. if some LD derivation starting in *Goal* is infinite) then the grouping atom *moded_bagof*$(t, gl, Goal, x)$ does not return any answer (it loops).

A concept we need in the sequel is that of *terminating* program; since we are dealing with well-moded programs, the natural definition we refer to is that of *well-terminating* programs.

Definition 8. *A well-moded program is called* well-terminating *iff all its LD-derivations starting in a well-moded query are finite.*

Termination of (well-moded) logic programs has been exhaustively studied (see for example [3,15]). Here we follow the approach of Etalle, Bossi, and Cocco [15].

If the grouping atom is only in the top-level query and there are no grouping atoms in the bodies of the program clauses then, to ensure termination, it is sufficient to require that P be well-terminating in the way described by Etalle et al. [15]: i.e. that for every well-moded non-grouping atom A, all LD derivations of $P \cup A$ are finite. If this condition is satisfied then all LD derivations of $P \cup \mathit{Goal}$ are finite and then the query $\mathit{moded\_bagof}(t, gl, \mathit{Goal}, x)$ terminates (provided it is well-moded).

On the other hand, if we allow grouping atoms in the body of the clauses, then we have to make sure that the program does not include recursion through a grouping atom. The following example shows what can go wrong here.

Example 5. Consider the following program:

```
(1) p(X,Z) :- moded_bagof(Y,[X],q(X,Y),Z).
(2) q(X,Z) :- moded_bagof(Y,[X],p(X,Y),Z).
(3) q(a,1).  (4) q(a,2).  (5) q(b,3).  (6) q(b,4).
```

Here `p` and `q` are defined in terms of each other through the grouping operation. Therefore `p(X,Z)` cannot terminate until `q(X,Y)` terminates (clause 1). Computation of `q(X,Y)` in turn depends on the termination of the grouping operation on `p(X,Y)` (clause 2). Intuitively, one would expect that the model of this program contains `q(a,1)`, `q(a,2)`, `q(b,3)`, and `q(b,4)`. However, if we apply the extended LD resolvent (Definition 7) to compute the c.a.s. of `p(X,Y)` we see that the computation loops.

In order to prevent this kind of problems, to guarantee termination we require programs to be *aggregate stratified* [17]. *Aggregate stratification* is similar to the concept of *stratified negation* [1], and puts syntactical restrictions on the aggregate programs so that recursion through *moded_bagof* does not occur. For the notation, we follow Apt et al. in [1]. Before we proceed to the definition of aggregate stratified programs we need to formalise the following notions. Given a program P and a clause $H \leftarrow \ldots, B, \ldots. \in P$:

- if B is a grouping atom $\mathit{moded\_bagof}(t, gl, \mathit{Goal}, x)$ then we say that $\mathit{Pred}(H)$ *refers to* $\mathit{Pred}(\mathit{Goal})$;
- otherwise, we say that $\mathit{Pred}(H)$ *refers to* $\mathit{Pred}(B)$.

We say that relation symbol p *depends on* relation symbol q in P, denoted $p \sqsupseteq q$, iff (p, q) is in the reflexive and transitive closure of the relation *refers to*. Given a non-grouping atom B, the definition of B is the subset of P consisting of *all* clauses with a formula on the left side whose relation symbol is $\mathit{Pred}(B)$. Finally, $p \simeq q \equiv p \sqsubseteq q \wedge p \sqsupseteq q$ means that p and q are mutually recursive, and $p \sqsupset q \equiv p \sqsupseteq q \wedge p \not\simeq q$ means that p calls q as a subprogram. Notice that $\sqsupset$ is a well-founded ordering.

Definition 9. *A program P is called* aggregate stratified *if for every clause $H \leftarrow B_1, \ldots, B_m$, in it, and every B_j in its body if B_j is a grouping atom $B_j = \mathit{moded\_bagof}(t, gl, \mathit{Goal}, x)$ then $\mathit{Pred}(\mathit{Goal}) \not\simeq \mathit{Pred}(H)$.*

Given the finiteness of programs it is easy to show that a program P is aggregate stratified iff there exists a partition of it $P = P_1 \cup \cdots \cup P_n$ such that for every $i \in [1, \ldots, n]$, and every clause $cl = H \leftarrow B_1 \ldots, B_m \in P_i$, and every B_j in its body, the following conditions hold:

1. if $B_j = \mathit{moded\_bagof}(\ldots, \ldots, \mathit{Goal}, \ldots)$ then the definition of $\mathit{Pred}(\mathit{Goal})$ is contained within $\bigcup_{j<i} P_j$,
2. otherwise the definition of $\mathit{Pred}(B)$ is contained within $\bigcup_{j\leq i} P_j$.

Stratification alone does not guarantee termination. The following (obvious) example demonstrates this.

Example 6. Take the following program:

```
q(X,Y) :- r(X,Y).
r(X,Y) :- q(X,Y).
p(Y,X) :- moded_bagof(Z,[Y],q(Y,Z),X).
```

Notice that `q` $\simeq$ `r`. This program is aggregate stratified, but the query `p(Y,X)` will not terminate.

In order to handle the problem of Example 6 we need to modify slightly the classical definition of termination. The following definition relies on the fact that the programs we are referring to are aggregate stratified.

Definition 10 (Termination of Aggregate Stratified Programs). *Let P be an aggregate stratified program. We say that P is well-terminating if for every well-moded atom A the following conditions hold:*

1. *All LD derivations of $P \cup A$ are finite,*
2. *For each LD derivation δ of $P \cup A$, for each grouping atom $\mathit{moded\_bagof}(t, gl, \mathit{Goal}, x)$ selected in δ, $P \cup \mathit{Goal}$ terminates.*

The classical definition of termination considers only point (1). Here however, we have grouping atoms which actually trigger a side goal which is not taken into account by (1) alone. This is the reason why we need (2) as well. Notice that the notion is well-defined thanks to the fact that programs are aggregate stratified.

To guarantee termination, we can combine the notion of aggregate stratified program above with the notion of well-acceptable program introduced by Etalle, Bossi, and Cocco in [15] (other approaches are also possible). We now show how.

Definition 11. *Let P be a program and let $\mathbf{B}_P$ be the corresponding Herbrand base. A function $|\ |$ is a* moded level mapping *iff*

1. *it is a* level mapping *for P, namely it is a function $|\ | : \mathbf{B}_P \rightarrow \mathbb{N}$, from ground atoms to natural numbers;*
2. *if $p(\mathbf{t})$ and $p(\mathbf{s})$ coincide in the input positions then $|p(\mathbf{t})| = |p(\mathbf{s})|$.*

For $A \in \mathbf{B}_P$, $|A|$ is called the level *of A.* □

Condition (2) above states that the level of an atom is independent from the terms filling in its output positions. Finally, we can report the key concept we use in order to prove well-termination.

Definition 12. (Weakly- and Well-Acceptable [15]) *Let P be a program, $|\ |$ be a level mapping and M a model of P.*

– *A clause of P is called* weakly acceptable *(wrt $|\ |$ and M) iff for every ground instance of it, $H \leftarrow \mathbf{A}, B, \mathbf{C}$,*

$$\text{if} \quad M \models \mathbf{A} \quad \text{and} \quad Pred(H) \simeq Pred(B) \quad \text{then} \quad |H| > |B|.$$

P is called weakly acceptable with respect to $|\ |$ and M *iff all its clauses are.*
– *A program P is called* well-acceptable *wrt $|\ |$ and M iff $|\ |$ is a* moded *level mapping, M is a model of P and P is weakly acceptable wrt them.* □

Notice that a fact is always both weakly acceptable and well-acceptable; furthermore if M_P is the least Herbrand model of P, and P is well-acceptable wrt $|\ |$ and some model I then, by the minimality of M_P, P is well-acceptable wrt $|\ |$ and M_P as well. Given a program P and a clause $H \leftarrow \ldots, B, \ldots$ in P, we say that B is *relevant* iff $Pred(H) \simeq Pred(B)$. For the weakly and well-acceptable programs the norm has to be checked only for the relevant atoms, because only the relevant atoms might provide recursion. Notice then that, because we additionally require that programs are aggregate stratified, grouping atoms in a clause are not relevant (called as subprograms).
We can now state the main result of this section.

Theorem 1. *Let P be a well-moded aggregate stratified program.*

– *If P is well-acceptable then P is well-terminating.*

Proof. (Sketch). Given a well-moded atom A, we have to prove that (a) all LD derivations starting in A are finite and that (b) for each LD derivation δ of $P \cup A$, for each grouping atom *moded_bagof*$(t, gl, Goal, x)$ selected in δ, $P \cup Goal$ terminates.

To prove (a) one can proceed exactly as done in [15], where the authors use the same notions of well-acceptable program: the fact that here we use a modified version of LD-derivation has no influence on this point: since grouping atoms are resolved by removing them, they cannot add anything to the length of an LD derivation.

On the other hand, to prove (b) one proceeds by induction on the strata of P. Notice that at the moment that the grouping atom is selected, *Goal* is well-moded (i.e., ground in its input position). Now, for the base case if *Goal* is defined in P_1, then, by (a) we have that all LD-derivations starting in *Goal* are finite, and since we are in stratum P_1 (where clause bodies cannot contain grouping atoms) no grouping atom is ever selected in an LD derivation starting in *Goal*. So $P \cup Goal$ terminates.

The inductive case is similar: if *Goal* is defined in P_{i+1}, then, by (a) we have that all LD-derivations starting in *Goal* are finite, and since we are in stratum P_{i+1} if a grouping atom *moded_bagof*$(t', gl', Goal', x')$ is selected in an LD derivation starting in *Goal*, we have that $Goal'$ must be defined in $P_1 \cup \cdots \cup P_i$, so that – by inductive hypothesis – we know that $P \cup Goal'$ terminates. Hence the thesis. □

8 Related Work

Aggregate and grouping operations are given lots of attention in the logic programming community. In the resulting work we can distinguish two approaches: (1) in which the grouping and aggregation is performed at the same time, and (2) – which is closer to

our approach – in which grouping is performed first returning a multiset and then an aggregation function is applied to this multiset.

In the first approach an *aggregate subgoal* is given by $group\_by(p(\mathbf{x},\mathbf{z}),[\mathbf{x}],y = \mathbb{F}(E(\mathbf{x},\mathbf{z})))$, which is equivalent to $y = \mathbb{F}([\![\, E(\mathbf{x},\mathbf{z}) : \exists(\mathbf{z})p(\mathbf{x},\mathbf{z}) \,]\!])$. Here $\mathbf{x}$ are the grouping variables, $p(\mathbf{x},\mathbf{z})$ is a so called aggregation predicate, and $E(\mathbf{x},\mathbf{z})$ is a tuple of terms involving some subset of the variables $\mathbf{x} \cup \mathbf{z}$. $\mathbb{F}$ is an aggregate function that maps a multiset to a single value. The variables $\mathbf{x}$ and y are free in the subgoal while $\mathbf{z}$ are local and cannot appear outside the aggregate subgoal. In other words, except for output variable y, if a variable does not appear on the grouping list, this variable is local. The early declarative semantics for $group\_by$ was given by Mumick et al. [19]. In this work, aggregate stratification is used to prevent recursion through aggregates. Later, Kemp and Stuckey [17] provide the declarative semantics for $group\_by$ in terms of well-founded and stable semantics. They also examine different classes of aggregate programs: aggregate stratified, group stratified, magical stratified, and also monotonic and semi-ring programs. From a more recent work, Faber et al. [16] also rely on aggregate stratification and they define a declarative semantics for disjunctive programs with aggregates. They use the intensional set definition notation to specify the multiset for the aggregate function. Denecker et al. [12] point out that requiring the programs to be aggregate stratified might be too restrictive in some cases and they propose a stronger extension of the well-founded and stable model semantics for logic programs with aggregates (called *ultimate* well-founded and stable semantics). In their approach, Denecker et al. use the Approximation Theory [11]. The work of Denecker et al. is continued and further extended by Pelov et al. [20].

In the second approach, where the grouping is separated from aggregation (as in our approach), the grouping operation is represented by an intensional set definition. This approach uses an (intensional) set construction operator returning a multiset of answers which is then passed as an argument of an aggregate function: $m = [\![\, E(\mathbf{x},\mathbf{z}) : \exists(\mathbf{z})p(\mathbf{x},\mathbf{z}) \,]\!], y = \mathbb{F}(m)$. To be handled correctly (with a well defined declarative semantics), this approach requires multisets to be introduced as first-class citizens of the language. Dovier, Pontelli, and Rossi [14] introduce intensionally defined sets into the constraint logic programming language $CLP(\{\mathcal{D}\})$ where $\mathcal{D}$ can be for instance $\mathbb{FD}$ for finite domains or $\mathbb{R}$ for real numbers. In their work, Dovier et al. concentrate on the set-based operations and so, they do not consider multisets directly. Interestingly, they treat the intensional set definition as a special case of an aggregate subgoal in which $\mathbb{F}$ is a function which given a multiset m as an argument returns the set of all elements in m – i.e. $\mathbb{F}$ removes duplicates from m.

Introducing (multi)sets to a pure logic programming language (i.e. not relying on a CLP scheme) is also a well-researched area. From the most prominent proposals, Dovier et al. [13] propose an extended logic programming language called {log} (read "set-log") in which sets are first-class citizens. The authors introduce the basic set operations like set membership $\in$ and set equality $=$ along with their negative counterparts $\notin$ and $\neq$.

Concerning multisets directly, Ciancarini et al. [6] show how to extend a logic programming language with multisets. They strictly follow the approach of Dovier et al. [13]. Important to notice here, is that these earlier works of Dovier et al. and

Ciancarini et al. (as well as most of other related work on embedding sets in a logic programming language – see Dovier et al. [14,13] for examples) focus on the so called *extensional set construction* – which basically means that a set is constructed by enumerating the elements of the set. This is not suitable for our work as this does not enable us to perform grouping.

Moded Logic Programming is well-researched area [2,21]. However, modes have been never applied to aggregates. We also extend the standard definition of a mode to include the notion of *local variables*. By incorporating the mode system we are able to state the groundness and termination results for the *bagof*-like operations.

9 Conclusions

In this paper we study the grouping operations in Prolog using the standard Prolog built-in predicate *bagof*. Grouping is needed if we want to perform aggregation, and we need aggregation in TuLiP to be able to model reputation systems. In order to make the grouping operations easier to integrate with TuLiP, we add modes to *bagof* (we call the moded version *moded_bagof*). We extend the definition of a mode by allowing some variables in a grouping atom to be *local*. Finally, we show that for the class of well-terminating aggregate stratified programs the basic properties of well-modedness and well-termination also hold for programs with grouping.

Future Work. At the University of Twente we develop a new Trust Management language TuLiP. TuLiP is a function-free first-order language that uses modes to support distributed credential discovery. In Trust Management, the need of having support for aggregate operations is widely accepted. This would allow one to bridge two related yet different worlds of certificate based and reputation based trust management. At the moment TuLiP does not support aggregate operations. We are planning to incorporate the $moded\_bagof$ operator introduced in this paper in TuLiP and investigate its applicability in the Distributed Trust Management.

Acknowledgements. This work was carried out within the Freeband I-Share project.

References

1. Apt, K.R.: Introduction to Logic Programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B, pp. 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge (1990)
2. Apt, K.R., Marchiori, E.: Reasoning about Prolog programs: from Modes through Types to Assertions. Formal Aspects of Computing 6(6A), 743–765 (1994)
3. Apt, K.R., Pedreschi, D.: Reasoning about termination of pure Prolog programs. Information and Computation 106(1), 109–157 (1993)
4. Apt, K.R., Pellegrini, A.: On the occur-check free Prolog programs. ACM Toplas 16(3), 687–726 (1994)
5. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized Trust Management. In: Proc. 17th IEEE Symposium on Security and Privacy, May 1996, pp. 164–173. IEEE Computer Society Press, Los Alamitos (1996)

6. Ciancarini, P., Fogli, D., Gaspari, M.: A Logic Language based on GAMMA-like Multiset Rewriting. In: Herre, H., Dyckhoff, R., Schroeder-Heister, P. (eds.) ELP 1996. LNCS, vol. 1050, pp. 83–101. Springer, Heidelberg (1996)
7. Clarke, D., Elien, J.E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.L.: Certificate Chain Discovery in SPKI/SDSI. Journal of Computer Security 9(4), 285–322 (2001)
8. Czenko, M.R.: TuLiP: Reshaping Trust Management. PhD thesis, University of Twente, Enschede (June 2009)
9. Czenko, M.R., Doumen, J.M., Etalle, S.: Trust Management in P2P Systems Using Standard TuLiP. In: Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security, IFIP International Federation for Information Processing, Trondheim, Norway, May 2008, vol. 263, pp. 1–16. Springer, Boston (2008)
10. Czenko, M.R., Etalle, S.: Core TuLiP - Logic Programming for Trust Management. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 380–394. Springer, Heidelberg (2007)
11. Denecker, M., Marek, V., Truszczyński, M.: Approximations, Stable Operators, Well-Founded Operators, Fixpoints and Applications in Nonmonotonic Reasoning. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, ch. 6. The Springer International Series in Engineering and Computer Science, vol. 597, pp. 127–144. Springer, Heidelberg (2001)
12. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 212–226. Springer, Heidelberg (2001)
13. Dovier, A., Omodeo, E.G., Pontelli, E., Rossi, G.: {log}: A logic programming language with finite sets. In: ICLP, pp. 111–124. MIT Press, Cambridge (1991)
14. Dovier, A., Pontelli, E., Rossi, G.: Intensional Sets in CLP. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 284–299. Springer, Heidelberg (2003)
15. Etalle, S., Bossi, A., Cocco, N.: Termination of well-moded programs. J. Log. Program 38(2), 243–257 (1999)
16. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
17. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: ISLP, pp. 387–401. MIT Press, Cambridge (1991)
18. Li, N., Mitchell, J., Winsborough, W.: Design of a Role-based Trust-management Framework. In: Proc. IEEE Symposium on Security and Privacy, pp. 114–130. IEEE Computer Society Press, Los Alamitos (2002)
19. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The Magic of Duplicates and Aggregates. In: Proc. 16th International Conference on Very Large Databases, pp. 264–277. Morgan Kaufmann Publishers Inc, San Francisco (1990)
20. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory and Practice of Logic Programming (TPLP) 7(3), 301–353 (2007)
21. Somogyi, Z., Henderson, F., Conway, T.: Mercury: an efficient purely declarative logic programming language. In: Australian Computer Science Conference (1995), http://www.cs.mu.oz.au/mercury/papers.html

On Inductive and Coinductive Proofs via Unfold/Fold Transformations*

Hirohisa Seki

Dept. of Computer Science, Nagoya Inst. of Technology,
Showa-ku, Nagoya, 466-8555 Japan
seki@nitech.ac.jp

Abstract. We consider a new application condition of negative unfolding, which guarantees its safe use in unfold/fold transformation of stratified logic programs. The new condition of negative unfolding is a natural one, since it is considered as a special case of replacement rule. The correctness of our unfold/fold transformation system in the sense of the perfect model semantics is proved. We then consider the coinductive proof rules proposed by Jaffar et al. We show that our unfold/fold transformation system, when used together with Lloyd-Topor transformation, can prove a proof problem which is provable by the coinductive proof rules by Jaffar et al. To this end, we propose a new replacement rule, called *sound* replacement, which is not necessarily equivalence-preserving, but is essential to perform a reasoning step corresponding to coinduction.

Keywords: Preservation of equivalence, negative unfolding, coinduction, unfold/fold transformation.

1 Introduction

Since the pioneering paper by Tamaki and Sato [12], a number of unfold/fold transformation rules for logic programs have been reported (see an excellent survey [7] 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 usual (positive) unfold/fold rules and replacement rules, negative unfolding is shown to play an important role in program transformation, construction (e.g., [5], [3]) and verification (e.g., [8], [10]). One of the motivations of this paper is to re-examine negative unfolding proposed in the literature.

The framework for program synthesis by Kanamori-Horiuchi [5] is one of the earliest works in which negative unfolding is introduced. Pettorossi and Proietti (resp., Fioravanti, Pettorossi and Proietti) have proposed transformation rules for locally stratified logic programs [8] (resp., locally stratified constraint logic programs [3]), including negative unfolding (PP-negative unfolding for short).

* This work was partially supported by JSPS Grant-in-Aid for Scientific Research (C) 21500136.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 82–96, 2010.

Unlike positive unfolding, however, PP-negative unfolding does not always preserve the semantics of a given program in general, when used with unfold/fold rules. We give such a counterexample in Sect. 2.2, which shows that, when used together with unfolding and folding, negative unfolding requires a careful treatment. In this paper, we therefore reconsider the application condition of negative unfolding, and propose a new framework for unfold/fold transformation of stratified programs which contains a replacement rule as well. We show that our proposed framework preserves the perfect model semantics. The new condition of negative unfolding given in this paper is a natural one, since it can be considered as a special case of the application condition of replacement.

Our motivation behind the proposed transformation system is its applicability to proving properties of the perfect model of a (locally) stratified program. The relationship between unfold/fold transformation and theorem proving has been recognized; an unfolding rule corresponds to a resolution step, while a folding operation corresponds to an application of inductive hypotheses. In fact, several approaches of using unfold/fold transformation to proving program properties have been reported, among others, Pettorossi and Proietti [8], Fioravanti et al. [3] and Roychoudhury et al. [10] These are precursors of the present paper. In this paper, we consider the coinductive proof rules proposed by Jaffar et al. [4] We show that our unfold/fold transformation system, when used together with Lloyd-Topor transformation [6], can prove a proof problem which is provable by the coinductive proof rules by Jaffar et al. Our proof method based on unfold/fold transformation has therefore at least the same power as that of Jaffar et al. To this end, we propose a new replacement rule, called *sound* replacement, which is not necessarily equivalence-preserving, but plays an important role to perform a reasoning step corresponding to coinduction.

The organization of this paper is as follows. In Section 2, we describe a framework for unfold/fold transformation of stratified programs and give the new condition for the safe use of negative unfolding. In Section 3, we explain the coinductive proof rules by Jaffar et al. [4], and discuss an application of our framework for unfold/fold transformation to proving properties of constraint logic programs. Finally, we 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 [6,1].

2 A Framework for Unfold/Fold Transformation

In this section, we propose a framework for unfold/fold transformation of stratified programs which includes negative unfolding as well as replacement rule. Although we confine the framework to *stratified* programs here for simplicity, it is possible to extend it to *locally stratified constraint* programs as in [3].

The frameworks proposed by Pettorossi and Proietti [8] and by Fioravanti et al. [3] are based on the original framework by Tamaki-Sato [12] for definite

[1] Due to space constraints, we omit most proofs and some details, which will appear in the full paper.

programs, while our framework given below is based on the generalized one by Tamaki-Sato [13]. Roychoudhury et al. [10] proposed a general framework for unfold/fold transformation which extended the one by Tamaki-Sato [13]. Their systems [10], [9] have a powerful folding rule (*disjunctive* folding), whereas they did not consider negative unfolding.

2.1 Transformation Rules

We first explain our transformation rules here, and then prepare some conditions imposed on the transformation rules and show the correctness of transformation in Sect. 2.2.

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 definition 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. In the following, 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.

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 stratification of predicate symbol of A. We assume that σ satisfies the following: For every primitive atom A, $\sigma(A) = 0$. For a positive literal A, $\sigma(\neg A) = \sigma(A) + 1$ if A is non-primitive, and $\sigma(\neg A) = 0$ otherwise. 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.

For a stratified program P, we denote its perfect model by $M(P)$.

In our framework, we assume that an initial program, from which an unfold/fold transformation sequence starts, has the structure specified in the following definition.[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 satisfies the *initial program condition*, if the following conditions hold:

1. No non-primitive predicate appears in P_{pr}.

[2] In [8], primitive (non-primitive) predicates are called as *basic* (*non-basic*), respectively.

[3] When we say "an initial program P_0" hereafter, P_0 is assumed to satisfy the initial program condition in Def. 1.

2. P_{np} is a *stratified* program, with a stratification σ, called the *initial stratification*. Moreover, σ is defined as follows: For every non-primitive predicate symbol p, $\sigma(p) = \max(1, m)$, where $m := \max\{\sigma(bd(C)) \mid C \in Def(p, P_0)\}$.
3. Each predicate symbol p in P_0 is assigned a non-negative integer i ($0 \leq i \leq I$), called the *level* of the predicate symbol, denoted by $level(p)$, where I is 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 define 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, every predicate symbol of a *positive* literal in the body of a clause in P_0 has a level not greater than the level of the clause. □

Remark 1. The above definition follows the generalized framework in [13], thereby eliminating some restrictions in the original one [12]. In [12], the number of levels is two; each predicate in an initial program is classified as either *old* or *new*. The definition of a new predicate consists of a single clause whose body contains positive literals with old predicates only, thus a recursive definition of a new predicate is not allowed. Moreover, it has no primitive predicates. □

We now give the definitions of our transformation rules. First, *positive* unfolding is defined as usual.

Definition 2. Positive Unfolding
Let C be a renamed apart clause in a stratified 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 unifiable with $hd(D_1), \ldots, hd(D_k)$, with most general unifiers (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 $i = 1, \ldots, k$. □

The following definition of negative unfolding rule is due to Pettorossi and Proietti (*PP-negative unfolding*, for short) [8].

Definition 3. Negative Unfolding
Let C be a renamed apart clause in a stratified 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 unifiable with $hd(D_1), \ldots, hd(D_k)$, with most general unifiers $\theta_1, \ldots, \theta_k$, respectively. Assume that:

1. $A = hd(D_1)\theta_1 = \ldots = hd(D_k)\theta_k$, that is, for each i ($1 \leq i \leq k$), A is an instance of $hd(D_i)$,
2. for each i ($1 \leq i \leq k$), D_i has no existential variables, and
3. 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 first 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 $i = 1, \ldots, r$. □

Next, we recall the definition of folding in [13]. The notion of a *molecule* [13] is useful for clearly stating folding, replacement rule and other related terminology.

Definition 4. Molecule, Identity of Molecules [13]
An existentially quantified 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* and $A_1, \ldots, A_n$ are literals. The set of other variables in M are called *free variables*, denoted by $Vf(M)$.

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 two molecules are involved, they are assumed to have disjoint sets of variables, unless otherwise stated.

A molecule without free variables is said to be *closed*. A molecule without free variables nor existential variables is said to be *ground*. A molecule M is called an *existential instance* of a molecule N, if M is obtained from N by eliminating some existential variables by substituting some terms for them.[4] □

Definition 5. Folding, Reversible Folding
Let P be a 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 $D : A' \leftarrow M'$ in P and a substitution θ of free variables of A' such that $A'\theta = A$ and $M'\theta = M$.

Let C be a clause of the form: $B \leftarrow \exists X_1 \ldots X_n(M, N)$, 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 clause D), the result of *folding* C w.r.t. M *by* P is the clause: $B \leftarrow \exists X_1 \ldots X_n(A, N)$. 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 definition.[5] □

To state conditions on replacement, we need the following definition.

Definition 6. Proof of an Atom (a Molecule)
Let P be a stratified program and A be a ground atom true in $M(P)$. A finite successful ground SLS-derivation T with its root $\leftarrow A$ is called a *proof* of A by P.

The definition of proof is extended from a ground atom to a conjunction of ground literals, i.e., a ground molecule, in a straightforward way. Let L be a ground molecule and T be a proof of L by P. Then, we say that L has a proof T by P. L is also said to be *provable* if L has some proof by P.

For a *closed* molecule M, a proof of any ground existential instance of M is said to be a *proof of* M by P. □

[4] The variables in the substituted terms, if any, becomes free variables of M.
[5] The terminology of *reversible* folding was used in a totally different sense in the literature (see [7]).

Definition 7. 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: $A \leftarrow \exists X_1 \ldots X_n(M_2\theta, N)$.

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 iff $M_2\theta$ has a proof by P_0. □

We can now define a transformation sequence as follows:

Definition 8. 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_n $(n \geq 1)$ is obtained from P_{n-1} by applying to a *non-primitive* clause in P_{n-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.

2.2 Correctness of Unfold/Fold Transformation

To preserve the perfect model semantics of a program in transformation, we need some conditions on the transformation rules. The conditions we impose on the transformation rules are intended for the rules to satisfy the following two properties: one is for the preservation of the initial stratification σ of an initial program P_0, and the other is for preserving an invariant of the size (according to a suitable measure μ) of the proofs of an atom true in P_0. Table 1 summarizes the conditions imposed on the transformation rules in our framework, and they will be explained in this section.

The following definition of the well-founded measure μ is a natural extension of that in [13], where μ is defined in terms of an SLD-derivation.

Definition 9. Weight-Tuple, Well-founded Measure μ, i-th truncation of 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 the initial program 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 $\langle w_1, \ldots, w_I \rangle$.

We define the *well-founded measure* $\mu(A)$ as follows:

$$\mu(A) := \min\{w \mid w \text{ is the weight-tuple of a proof of } A\}$$

Table 1. Conditions Imposed on the Transformations Rules

transformation rule	preservation of σ	preservation of μ-completeness
definition (init. program P_0) (Def. 1)	$\exists$ init. stratification 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 (Def. 15)
folding	σ-consistent (Def. 10)	TS-folding condition (Def. 12)
replacement $M_1 \Rightarrow M_2$	σ-consistent (Def. 14) $\sigma(M_1) \geq \sigma(M_2)$	μ-consistent (Def. 14) $\mu(M_1) \geq \mu(M_2)$

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 defined similarly. For a *closed* molecule M, $\mu(M) := \min\{w \mid w$ is the weight-tuple of a proof of M', where M' is a ground existential instance of $M\}$. The i-th *truncation* of $\mu(M)$, denoted by $\mu_i(M)$, is defined by replacing w by $\langle w_1, \ldots, w_i \rangle$ in the definition of $\mu(M)$. □

Note that the above defined measure μ is *well-founded* over the set of ground molecules which have proofs by P_0. By definition, for a ground primitive atom A true in $M(P_0)$, $\mu(A) = \langle 0, \ldots, 0 \rangle = \mathbf{0}$ (I-tuple).

Conditions on Folding. We first give the conditions imposed on folding. The following is to preserve the initial stratification σ of initial program P_0, when folding is applied.

Definition 10. σ-consistent folding
Let P_0 be an initial program with the initial stratification σ. 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., $\sigma(hd(C)) \geq \sigma(hd(D))$. □

The following gives a sufficient condition for folding to be σ-consistent.

Proposition 1. Let P_0 be an initial program with the initial stratification σ. Suppose further that the definition of clause D in P_0 consists of a single clause, that is, $Def(p, P_0)$ is a singleton, where p is the predicate symbol of $hd(D)$. Then, every application of reversible folding with folding clause D is σ-consistent. □

We note that, when $Def(p, P_0)$ is a singleton, $\sigma(p) = \max(1, \sigma(bd(D)))$ (see Def. 1). Then, the above proposition is obvious. The framework by Pettorossi-Proietti [8] satisfies this condition, since the head predicate of D is supposed to be a new predicate which does not appear elsewhere.

[6] We use the inequality signs $>, \leq$ to represent this lexicographic ordering.

Next, we explain another condition on folding for the preservation of μ, which is due to Tamaki-Sato [13] for definite programs.

Definition 11. Descent Level of a Clause
Let C be a clause appearing in a transformation sequence starting from an initial program P_0 with $I+1$ layers. The *descent level* of C, denoted by $dl(C)$, is defined inductively as follows:

1. If C is in P_0, $dl(C) := level(C)$, where $level(C)$ is the level of C in P_0.
2. If C is first 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 $dl(C) := dl(C')$, if A is primitive. If A is non-primitive, then $dl(C) := \min\{dl(C'), level(A)\}$.
3. If C is first introduced as the result of applying negative unfolding to some clause C', then $dl(C) := dl(C')$.
4. If C is first introduced as the result of folding, or applying some replacement rule to some submolecule of the body of some clause C', then $dl(C) := dl(C')$.

□

The difference between the above definition and that of the original one in [13] is Condition 3 for negative unfolding, while the other conditions remain unchanged.

Definition 12. TS-Folding Condition
In the transformation sequence, suppose that a clause C is folded using a clause D as the folding clause, where C and D are the same as those in Definition 5. Then, the application of folding is said to satisfy the *TS-folding condition*, if the descent level of C is *smaller* than the level of D. □

We are now in a position to give the definition of *μ-completeness* due to [13], which is crucial in the correctness proof; it is a sufficient condition for the preservation of equivalence. A ground instance of some clause C in P is called an *inference by* P, and C is called the *source clause* of the inference.

Definition 13. μ-inference, μ-complete
Let $P_0, \ldots, P_n$ be a transformation sequence, and μ be the well-founded measure for P_0. An inference $A \leftarrow L$ by P_n is called a *μ-inference*, if $\mu_i(A) > \mu_i(L)$ holds, where i is the descent level of the source clause of the inference.
P_n is said to be *μ-complete* if for every ground atom A true in $M(P_0)$, there is a μ-inference $A \leftarrow L$ by P_n such that $M(P_0) \models L$. □

Conditions on Replacement Rules. Next, we state our conditions imposed on replacement rules.

Definition 14. Replacement Rules Consistent with σ and μ
Let P_0 be an initial program with the initial stratification σ, and 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 *consistent* with σ (or σ-consistent) if $\sigma(M_1) \geq \sigma(M_2)$, and it is said to be *consistent* with the well-founded measure μ (or 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 replacement rule in Pettorossi-Proietti [8] satisfies the above conditions, since literals appearing in their replacement rules are primitive. Then, σ-consistency is trivial, since $\sigma(M_1) = \sigma(M_2) = 0$ by definition, while μ-consistency is due to the fact that the weight-tuple of a proof of $M_i\theta$ is $\mathbf{0}$ (I-tuple) for $i = 1, 2$.

The following proposition shows that the initial stratification is preserved in a transformation sequence.

Proposition 2. Preservation of the Initial Stratification σ
Let P_0 be an initial program with the initial stratification σ, and $P_0, \ldots, P_n$ ($n \geq 1$) be a transformation sequence, where every application of folding as well as replacement rule is consistent with σ. Then, P_n is stratified w.r.t. σ. □

The New Condition on Negative Unfolding and the Correctness of Transformation. We are now in a position to give an example which shows that negative unfolding does not always preserve the semantics of a given program.

Example 1. Let P_0 be the stratified program consisting of the following clauses:

$$P_0 = \{\ D: \begin{array}{l} f \leftarrow m, \neg e \\ m \leftarrow \\ e \leftarrow e \\ e \leftarrow \neg m. \end{array} \}$$

$$\begin{array}{lll} (1): & f \leftarrow \neg e & (\text{pos. unfolding } D) \\ (2): & f \leftarrow \neg e, m & (\text{neg. unfolding } (1)) \\ (3): & f \leftarrow f & (\text{folding } (2)) \end{array}$$

We note that $M(P_0) \models m \wedge \neg e$, thus $M(P_0) \models f$. Assume that the predicate symbol of m in P_0 is non-primitive. By applying positive unfolding to clause D w.r.t. m, we derive clause (1). Then, applying negative unfolding to clause (1) w.r.t. $\neg e$ results in clause (2), noting that $\neg(e \vee \neg m) \equiv \neg e \wedge m$. Since positive unfolding is applied to clause D w.r.t. a non-primitive atom m, the folding condition in [8] allows us to fold clause (2) w.r.t. $\neg e, m$ using folder clause D, obtaining clause (3). Now, we note that clause (3) is self-recursive. Let P' be the result of the program transformation starting from P_0, i.e., $P' = P_0 \setminus \{D\} \cup \{(3)\}$. Then, $M(P_0) \neq M(P')$, because $M(P') \models \neg f$. □

The application of negative unfolding always preserves the initial stratification σ, while it does not preserve the well-founded measure μ in general. In fact, applying negative unfolding to (1) w.r.t. $\neg e$ in Example 1 replaces it by $\neg e, m$, obtaining clause (2). We note that $\sigma(\neg e) = \sigma(\neg e, m)$, while it is not always true that $\mu(\neg e) \geq \mu(\neg e, m)$. To avoid the above anomaly, we therefore impose the following condition on negative unfolding.

Definition 15. Negative Unfolding Consistent with μ
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 Def. 3, every positive literal (if any) in Q_i is *primitive*, for $i = 1, \ldots, r$. □

In Example 1, μ-consistency of negative unfolding when applied to clause (1), requires that m be primitive. Then, this prohibits the subsequent folding operation in clause (3) from TS-folding condition (Def. 12). On the other hand, when m is non-primitive, the application of negative unfolding to clause (1) is not allowed, since it is not μ-consistent.

One way to view μ-consistent negative unfolding is that it is a special case of replacement rule R of the form $\neg A \Rightarrow Q_i$, where Q_i is given in Def. 3. When $M(P_0) \models \neg A$, it holds that $\mu(\neg A) = \mathbf{0}$. The μ-consistency of R requires that $\mu(Q_i) = \mathbf{0}$, which means that every positive literal (if any) in Q_i is primitive.

The following shows the correctness of our transformation system.

Proposition 3. Correctness of Transformation
Let P_0 be an initial program with the initial stratification 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 folding is σ-consistent and satisfies TS-folding condition, and (ii) every application of replacement rule is consistent with σ and μ. Moreover, suppose that every application of negative unfolding is μ-consistent. Then, $M(P_n) = M(P_0)$. □

3 Coinductive Proofs via Unfold/Fold Transformations

In this section, we consider the applicability of our transformation system to proving properties of the perfect model of a stratified program. Jaffar et al. [4] consider proof obligations of the form $\mathcal{G} \models \mathcal{H}$, where $\mathcal{G}, \mathcal{H}$ are conjunctions of either an atom or a constraint, and $var(\mathcal{H}) \subseteq var(\mathcal{G})$. The validity of this entailment means that $M(P) \models \forall \tilde{X}(\mathcal{G} \rightarrow \mathcal{H})$[7], where P is a (constraint) definite program which defines predicates, called *assertion* predicates, occurring in $\mathcal{G}$ and $\mathcal{H}$, $\forall \tilde{X}$ is an abbreviation for $\forall X_1 \ldots \forall X_j$ $(j \geq 0)$ s.t. $X_j \in var(\mathcal{G})$. As we noted earlier, although our unfold/fold transformation system is given for *stratified* programs in Sect. 2 for simplicity of explanation, it is possible to extend it to locally stratified *constraint* programs as in [3]. While Jaffar et al. [4] consider a constraint logic program as P, we hereafter consider only equality (=, and $\neq$ for negation of an equation) constraints for the sake of simplicity, and assume the axioms of Clark's equality theory (CET) [6].

The proof rules by Jaffar et al. [4] are given in Fig. 1. A *proof obligation* is of the form $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$, where $\tilde{A}$ is a set of *assumption* goals.

When a proof obligation $\mathcal{G} \models \mathcal{H}$ is given, a proof will start with $\Pi = \{\emptyset \vdash \mathcal{G} \models \mathcal{H}\}$, and proceed by repeatedly applying the rules in Fig. 1 to it. In the figure, the symbol $\uplus$ represents the disjoint union of two sets, and UNFOLD$(\mathcal{G})$ is defined to be $\{\mathcal{G}' \mid \exists C \in P : \mathcal{G}' = reduct(\mathcal{G}, C)\}$, where $\mathcal{G} = B_1, \ldots, B_n$ is a goal (i.e., a conjunction of either constraints or literals), C is a clause in a given program P, and a *reduct* of $\mathcal{G} = B_1, \ldots, B_n$ using a clause C, denoted by $reduct(\mathcal{G}, C)$, is defined to be of the form: $B_1, \ldots, B_{i-1}, bd(C), B_i = hd(C), B_{i+1}, \ldots, B_n$. Note that a constraint $B_i = hd(C)$ gives an m.g.u. of B_i and $hd(C)$.

[7] As noted in [4], the use of the term *coinduction* here has no relationship with the *greatest* fixed point of a program.

$$\text{(LU+I)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}}\quad \text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$$

$$\text{(RU)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}}\quad \mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$$

$$\text{(CO)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\emptyset \vdash \mathcal{H}'\theta \models \mathcal{H}\}}\quad \mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}'\theta$$

$$\text{(CP)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}}\qquad \text{(DP)}\ \frac{\Pi \uplus \{\mathcal{G} \models \mathcal{H}\}}{\Pi}\quad \mathcal{G} \models \mathcal{H} \text{ holds by constraint solving}$$

Fig. 1. Coinductive Proof Rules by Jaffar et al. [4]

The *left unfold with new induction hypothesis* (LU+I) (or simply "left-unfold") rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as an assumption to every newly produced proof obligation.

On the other hand, the *right unfold* (RU) rule performs an unfold on the rhs of a proof obligation. The (RU) rule does not necessarily obtain all the reducts.

The rule *coinduction application* (CO) transforms an obligation by using an assumption which can be created only by the (LU+I) rule, thereby realizing the coinduction principle. The underlying principle behind the (CO) rule is that a "similar" assertion $\mathcal{G}' \models \mathcal{H}'$ has been previously encountered in the proof process, and assumed to be true.

The rule *constraint proof* (CP) removes one occurrence of a predicate $p(\tilde{y})$ appearing in the rhs of a proof obligation. Applying the CP rules repeatedly will reduce a proof obligation to the form which contains no assertion predicates in the rhs and consists only of constraints. Then, the *direct proof* (DP) rule may be attempted by simply removing any predicates in the corresponding lhs and by applying the underlying constraint solver assumed in the language we use.

Jaffar et al. show the soundness of the proof rules in Fig. 1 [4].

Theorem 1 (Soundness). [4] *A proof obligation $\mathcal{G} \models \mathcal{H}$ holds in $M(P)$ for a given definite constraint program P, if, starting with the proof obligation $\emptyset \vdash \mathcal{G} \models \mathcal{H}$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{A} \vdash \mathcal{G}' \models \mathcal{H}'$ such that* (a) *$\mathcal{H}'$ contains only constraints, and* (b) *$\mathcal{G}' \models \mathcal{H}'$ can be discharged by the constraint solver.* □

The following is an example of a coinductive proof in [4], and we show how the corresponding proof is done via unfold/fold transformations. To this end, we first state the notion of *useless* predicates, which is originally due to Pettorossi and Proietti [8], but we use it with a slight modification as follows.

Definition 16. 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 (i) either a positive literal whose predicate is in U, or (ii) a constraint which is unsatisfiable. □

It is easy to see that, if p is a useless predicate of P, then $M(P) \models \neg A$, where A is a ground atom with predicate symbol p.

Example 2. A Coinductive Proof without Base Case [4]
Let P be a program consisting of the following clauses:

$$p(X) \leftarrow q(X)$$
$$q(X) \leftarrow q(X)$$
$$r(X) \leftarrow$$

Suppose that the proof obligation is to prove that $p(X) \models r(X)$, calling this assertion A_1. The proof process is shown in Fig. 2 (left). We first apply rule (LU+I) to A_1, obtaining another assertion A_2: $q(X) \models r(X)$. Again, we apply rule (LU+I) to A_2, deriving another assertion A_3, which is equivalent to A_2. This time, we can apply the coinduction rule (CO) to A_3, and obtain a new assertion $r(X) \models r(X)$. This assertion is then proved simply by applying rules (CP) and (DP).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\emptyset \vdash p(X) \models r(X)}{\{A_1\} \vdash q(X) \models r(X)}\ (\mathsf{LU{+}I})}{\{A_1, A_2\} \vdash q(X) \models r(X)}\ (\mathsf{LU{+}I})}{\emptyset \vdash r(X) \models r(X)}\ (\mathsf{CO})}{\models X = X}\ (\mathsf{CP})}{true}\ (\mathsf{DP})$$

C_f :	$f \leftarrow \neg nf_1$	
C_{nf_1} :	$nf_1 \leftarrow p(X), \neg r(X)$	
C_{nf_2} :	$nf_2(X) \leftarrow q(X), \neg r(X)$	
(1) :	$nf_1 \leftarrow q(X), \neg r(X)$	(pos. unfolding C_{nf_1})
(2) :	$nf_1 \leftarrow nf_2(X)$	(folding (1))
(3) :	$nf_2(X) \leftarrow q(X), \neg r(X)$	(pos. unfolding C_{nf_2})
(4) :	$nf_2(X) \leftarrow nf_2(X)$	(folding (3))

Fig. 2. A Coinductive Proof of Example 2 and the Corresponding Proof via Unfold/fold Transformations

Fig. 2 (right) shows the corresponding proof via unfold/fold transformations. We first consider the clause C_0 corresponding to the initial proof obligation A_1, i.e., $C_0 : \ f \leftarrow \forall X(p(X) \rightarrow r(X))$. Then, we apply Lloyd-Topor transformation to C_0, obtaining the clauses $\{C_f, C_{nf_1}\}$, where predicate nf_1 is a new predicate introduced by Lloyd-Topor transformation.

We assume that assertion predicates (p and q in this example) are non-primitive. By applying positive unfolding to C_{nf_1} w.r.t. $p(X)$, we have clause (1). From this, we consider a new clause C_{nf_2} whose body is the same as that of (1) and assume that C_{nf_2} is in initial program P_0 from scratch. Therefore, let $P_0 = P \cup \{C_f, C_{nf_1}, C_{nf_2}\}$. We then apply folding to clause (1), obtaining clause (2).

On the other hand, applying positive unfolding to C_{nf_2} w.r.t. $q(X)$ results in clause (3), which is then folded by using folder clause C_{nf_2}, giving a self-recursive

clause (4). Let $P_4 = P_0 \setminus \{C_{nf_1}, C_{nf_2}\} \cup \{(2), (4)\}$. Since the above transformation sequence preserves the perfect model semantics, it holds that $M(P_0) = M(P_4)$. Note that nf_2 and nf_1 are useless predicates of P_4. We thus have that $M(P_4) \models \forall X(\neg nf_2(X)) \wedge \neg nf_1$, which means that $M(P_4) \models f$. Therefore, it follows that $M(P_0) \models f$, which is to be proved. □

Next, we consider how to realize the reasoning step corresponding to the coinduction rule in our transformation system. The coinduction rule (CO) in Fig. 1 requires to check whether there exists some substitution θ s.t. $\mathcal{G} \models \mathcal{G}'\theta$ and $\mathcal{H}'\theta \models \mathcal{H}$, which means that $M(P_0) \models (\mathcal{G} \wedge \neg\mathcal{H}) \rightarrow (\mathcal{G}' \wedge \neg\mathcal{H}')\theta$, where P_0 is a program defining assertion predicates. In this case, if $(\mathcal{G} \wedge \neg\mathcal{H})$ has a proof by P_0, then $(\mathcal{G}' \wedge \neg\mathcal{H}')\theta$ has a proof by P_0, but not vice versa. We therefore propose a new form of the replacement rule, which is, unlike the replacement rule in Def. 7, not necessarily equivalence-preserving.

Definition 17. Sound Replacement Rule
Let P_0 be an initial program with the initial stratification σ, and R be a replacement rule of the form $M_1 \Rightarrow M_2$, which is consistent with σ and μ. Then, R is said to be *sound* w.r.t. P_0, if, for every ground substitution θ of free variables in M_1 and M_2, it holds that, *if* $M_1\theta$ has a proof by P_0, then so does $M_2\theta$. □

When we use the sound replacement rules in unfold/fold transformation, we can show the following proposition in place of Proposition 3.

Proposition 4. Soundness of Transformation
Let $P_0, \ldots, P_n$ $(n \geq 0)$ be a transformation sequence under the same assumptions in Prop. 3, except that, in the transformation sequence, some sound replacement rules are applied, with a proviso that, if a sound replacement rule is applied to clause C in P_k for some k $(0 \leq k \leq n)$ and it is the first time a sound replacement rule is applied in the transformation sequence, then every application of a sound replacement rule, if any, is applied to a clause C' in P_i $(k < i \leq n)$ with $\sigma(hd(C')) = \sigma(hd(C))$ for the rest of the transformation sequence, where σ is the initial stratification of P_0.

Then, it holds that (i) $M(P_0) \mid_{<j} = M(P_n) \mid_{<j}$ for all j s.t. $0 \leq j < \sigma(hd(C))$, and (ii) $M(P_0) \mid_{\sigma(hd(C))} \subseteq M(P_n) \mid_{\sigma(hd(C))}$, where $M \mid_{<i}$ $(M \mid_i)$ is the *restriction* of a perfect model M to the set of atoms whose strata are less than i (equal to i), respectively. □

We can now show that our proof via unfold/fold transformations including the sound replacement rule has at least the same power as that of the coinductive proof rules by Jaffar et al. [4], assuming that our transformation system is extended to deal with a constraint logic program with a suitable constraint language and the constraint theory corresponding to the underlying constraint solver. To show that, we find it convenient to use an expression, called an *extended* negative literal, which is defined as follows:

Definition 18. Extended Negative Literal

An *extended negative literal* is an expression of the form: $\forall \tilde{X}(\mathcal{H} \rightarrow \bot)$, where $\mathcal{H}$ is a conjunction of either atoms or constraints, $\tilde{X}$ are some free variables in $\mathcal{H}$, and $\bot$ means false. □

In particular, when an extended negative literal $\mathcal{N}$ is of the form: $h \rightarrow \bot$ and h is an atom, $\mathcal{N}$ is simply a negative literal $\neg h$. When an extended negative literal $\forall \tilde{X}(\mathcal{H} \rightarrow \bot)$ occurs in the body of a clause, we regard it a notational convention of $\neg newp(\tilde{Y})$, where *newp* is a new predicate symbol not appearing elsewhere and is defined by clause D of the form: $newp(\tilde{Y}) \leftarrow \mathcal{H}(\tilde{X}, \tilde{Y})$, where $\mathcal{H}(\tilde{X}, \tilde{Y})$ means that $\tilde{X}$ are the existential variables in D. Therefore, although we use an expression allowing an extended negative literal, our framework still remains in (constraint) stratified programs.

Proposition 5. Coinductive Proofs via Unfold/fold Transformations
Let P be a given (constraint) definite program. Suppose that a proof obligation $M(P) \models \forall \tilde{X}(\mathcal{G} \rightarrow \mathcal{H})$ can be proved by the coinductive proof rules in Fig 1. Suppose further that $P_0 = P \cup \{C_f, C_{nf}\}$, where $C_f = f \leftarrow \neg nf(\tilde{X})$ and $C_{nf} = nf(\tilde{X}) \leftarrow \mathcal{G}, (\mathcal{H} \rightarrow \bot)$.

Then, there exists a transformation sequence $P_0, \ldots, P_n$ $(n \geq 0)$ satisfying the same assumptions in Prop. 4, such that nf is a useless predicate of P_n. □

From the above proposition, our proof scheme via unfold/fold transformations with sound replacement will be as follows: If we obtain by transformation P_n such that nf is useless in P_n, then it follows from Prop. 5 that $M(P_n) \models \forall \tilde{X} \neg nf(\tilde{X})$, thus $M(P_0) \models f$, which is to be proved.

4 Conclusion

We have considered the new application condition of negative unfolding, which guarantees its safe use in an unfold/fold transformation system for stratified programs. We showed that the new application condition imposed on negative unfolding is a natural one, since it can be considered as a special case of the replacement rule. We proved that our unfold/fold transformation system preserves the perfect model semantics.

We then considered the coinductive proof rules proposed by Jaffar et al. [4] We showed that our unfold/fold transformation system, when used together with Lloyd-Topor transformation, can prove a proof problem which is provable by the coinductive proof rules by Jaffar et al. To this end, we proposed a new replacement rule, called *sound* replacement, which is not necessarily equivalence-preserving, but is essential to perform a reasoning step corresponding to coinduction.

In [11], a framework for unfold/fold transformation of locally stratified programs is proposed, where another well-founded ordering is introduced for the correctness proof, thus it is non-comparable with the current work. The transformation system by Roychoudhury et al. [10] used a very general measure for proving the correctness, and they considered a *disjunctive* folding. On the other hand, their systems [10,9] have no negative unfolding. In fact, the correctness

proof in [9] depends on the preservation of the semantic kernel [2], which is not preserved in general when negative unfolding is applied. We leave it for future research to investigate the difference of disjunctive folding and negative unfolding in application areas such as verification.

One of the motivations of this work is to understand the close relationship between program transformation and inductive theorem proving. We hope that our results reported in this paper will be a contribution to promote further cross-fertilization between the two fields.

Acknowledgement. The author would like to thank anonymous reviewers for their constructive and useful comments on the previous version of the paper.

References

1. Apt, K.R.: Introduction to Logic Programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 493–576. Elsevier, Amsterdam (1990)
2. Aravindan, C., Dung, P.M.: On the Correctness of Unfold/fold Transformation of Normal and Extended Logic Programs. J. of Logic Programming 24(3), 295–322 (1995)
3. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation Rules for Locally Stratified Constraint Logic Programs. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 291–339. Springer, Heidelberg (2004)
4. Jaffar, J., Santosa, A., Voicu, R.: A Coinduction Rule for Entailment of Recursively Defined Properties. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 493–508. Springer, Heidelberg (2008)
5. Kanamori, T., Horiuchi, K.: Construction of Logic Programs Based on Generalized Unfold/Fold Rules. In: Proc. the 4th Intl. Conf. on Logic Programming, pp. 744–768 (1987)
6. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
7. Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. J. of Logic Programming 19/20, 261–320 (1994)
8. Pettorossi, A., Proietti, M.: Perfect Model Checking via Unfold/Fold Transformations. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 613–628. Springer, Heidelberg (2000)
9. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C.R., Ramakrishnan, I.V.: Beyond Tamaki-Sato Style Unfold/fold Transformations for Normal Logic Programs. Int. Journal on Foundations of Computer Science 13(3), 387–403 (2002)
10. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C.R., Ramakrishnan, I.V.: An Unfold/fold Transformation Framework for Definite 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. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 168–184. Springer, Heidelberg (2009)
12. Tamaki, H., Sato, T.: Unfold/Fold Transformation of Logic Programs. In: Proc. 2nd Int. Conf. on Logic Programming, pp. 127–138 (1984)
13. Tamaki, H., Sato, T.: A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, Technical Report, No. 86-4, Ibaraki Univ., Japan (1986)

Coinductive Logic Programming with Negation

Richard Min and Gopal Gupta

Department of Computer Science,
The University of Texas, Dallas,
Richardson, TX 75080, USA

Abstract. We introduce negation into coinductive logic programming (co-LP) via what we term *Coinductive SLDNF (co-SLDNF)* resolution. We present declarative and operational semantics of co-SLDNF resolution and present their equivalence under the restriction of rationality. Co-LP with co-SLDNF resolution provides a powerful, practical and efficient operational semantics for Fitting's Kripke-Kleene three-valued logic with restriction of rationality. Further, applications of co-SLDNF resolution are also discussed and illustrated where Co-SLDNF resolution allows one to develop elegant implementations of modal logics. Moreover it provides the capability of non-monotonic inference (e.g., predicate Answer Set Programming) that can be used to develop novel and effective first-order modal non-monotonic inference engines.

Keywords: Coinductive Logic Programming; Negation as Failure; Program Completion; Kripke-Kleene three-valued logic.

1 Introduction

Coinduction is a powerful technique for reasoning about unfounded sets, unbounded structures, and interactive computations. Coinduction allows one to reason about infinite objects and infinite processes [2, 6]. Coinduction has been recently introduced into logic programming (termed *coinductive logic programming*, or *co-LP* for brevity) by Simon *et al* [17] and an operational semantics (termed *co-SLD resolution*) defined for it. Practical applications of co-LP include goal-directed execution of answer set programs [7], reasoning about properties of infinite processes and objects, model checking and verification and planning [16]. Negation is important in logic programming. Without negation, many of the interesting applications of co-LP, to planning, goal-directed execution of answer set programs, etc. are not possible. In this paper we extend Simon *et al*'s work on co-LP with negation as failure [3]. Our work can also be viewed as adding coinduction to SLDNF resolution [10], thus we term the operational semantics of co-LP extended with negation as failure as *coinductive SLDNF resolution* or *co-SLDNF resolution*. Co-SLDNF resolution and its correctness result constitute the main contribution of this paper. Co-LP with co-SLDNF resolution provides a powerful, practical and efficient operational semantics for Fitting's Kripke-Kleene three-valued logic [6] with restriction of rationality. The resulting language efficiently handles many challenging problems and applications dealing with rational infinite objects and streams, modal operators, nonmonotonic inference, etc. [3].

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 97–112, 2010.

2 Preliminaries

Coinduction is the dual of induction. Induction corresponds to well-founded structures that start from a basis which serves as the foundation for building more complex structures. For example, natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality. Thus, the inductive definition of a list of numbers is as follows: (i) [ ] (an empty list) is a list (initiality); (ii) [H | T] is a list if T is a list and H is some number (iteration); and, (iii) the set of lists is the minimal set of such lists (minimality). Minimality implies that infinite-length lists of numbers are not members of the inductively defined set of lists of numbers. Induction corresponds to least fixed point interpretation of recursive definitions. In contrast, coinduction eliminates the initiality condition and replaces the minimality condition with maximality. Thus, the coinductive definition of a list of numbers is: (i) [H | T] is a list if T is a list and H is some number (iteration); and, (ii) the set of lists is the maximal set of such lists (maximality). There is no need for the base case in coinductive definitions, and while this may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point (gfp) interpretation of recursive definitions (recursive definitions for which gfp interpretation is intended are termed corecursive definitions).

The basic concepts of co-LP are based on rational, *coinductive proof* [17], that are themselves based on the concepts of *rational tree* and *rational solved form* of Colmerauer [4]. A tree is *rational* if the cardinality of the set of all its subtrees is finite. An object such as a term, an atom, or a (proof or derivation) tree is said to be rational if it is modeled (or expressed) as a rational tree. A *rational proof* of a rational tree is its *rational solved form* computed by *rational solved form algorithm* [4], following the account of [11]. The reader is referred to [4, 11] for details. Some of the noteworthy results for rational trees and its algebra are: (i) the rational solved form algorithm always terminates, (ii) the conjunction of equations E is solvable iff E has a rational solved form, and (iii) the algebra of rational trees and the algebra of infinite trees are elementarily equivalent. For co-LP, there are three further extensions to the rational solved form. First, we extend the concept of rational proof of rational trees of terms to atoms with terms (predicates). Second, as we recall [8, 10, 11] the equality theory for the algebra of rational trees, requires one modification to the axioms of the equality theory of the algebra of finite trees for co-LP over the rational domain, namely, (i) $t(x) \neq x$, for all x and t for each "finite" term t(x) containing x that is different from x, and (ii) if $t(x) = x$ then $x = t(t(t(...)))$ for all x and t for each "rational" term. Note that this modified axiomatization of the equality theory is required for rational trees and we will elaborate with a few examples with co-LP. Third, negation is added.

A *coinductive proof* of a rational (derivation) tree of program P is a *rational solved form* (tree-solution) of the rational (derivation) tree. One worthy note is that irrational atoms are generally not found in practical logic programs. Further, any irrational atom that has an infinite derivation should have a *rational cover*, as noted in [9], which could be characterized by the (interim) rational atom observed in each step of the derivation. This observation will be used later to assure some of the results of infinite LP also applicable to rational LP.

The *Coinductive hypothesis rule* (CHR) states that during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C encountered earlier,

then the call C' succeeds; the new resolvent is R'θ where θ = mgu(C, C') and R' is obtained by deleting C' from R. With this rational feature, co-LP allows programmers to manipulate rational (finite and rational) structures in a decidable manner as noted earlier. To achieve this feature of rationality, unification has to be necessarily extended, to have "occurs-check" removed [4]. SLD resolution extended with the coinductive hypothesis rule is called co-SLD resolution [16, 17]. Co-SLD resolution is very similar to SLD resolution except that goals with rational proofs are permitted. In SLD-resolution, given a call during execution of a logic program, the candidate clauses are tried one by one via backtracking. Under co-SLD resolution, however, the candidate clauses are extended with yet more alternatives: applying the coinductive hypothesis rule to check if the current call will unify with any of the earlier calls. That is, coinductive hypothesis rule computes whether current node (an atom) in a derivation tree can be unified with an earlier node (an atom) or not. Therefore, if there is a cycle in the path of the execution, it will be detected by co-SLD and infinite traversal of this cycle stopped. Thus by applying co-SLD resolution throughout the rational tree (of derivation) of atoms, one may end up with a rational solved form (a coinductive proof) of rational derivation tree. Thus, given the coinductive logic program:

stream([H | T]):- number(H), stream(T).

the goal **?-stream(X)** will bind **X** to infinite (rational) streams of numbers. Solutions such as X = [1 | X], X = [1, 2 | X], etc., will be produced by the co-LP system using CHR.

The Infinitary Herbrand Universe of a logic program P, HU(P), is the set of all ground terms formed out of the constants and function symbols appearing in P. Note that HU contains infinite terms also (e.g., f(f(f(….)))). Herbrand Base of P, HB(P), is the set of all ground atoms formed by using predicate symbols in P with ground terms from HU(P), and similarly Herbrand Ground, HG(P), for all the ground clauses of P. We denote the subset of Herbrand Universe restricted to rational terms by $HU^R(P)$; $HB^R(P)$ and $HG^R(P)$ are similarly defined. Further, we say *rational Herbrand space* of program P, denoted $HS^R(P)$, to mean the 3-tuple of ($HU^R(P)$, $HB^R(P)$, $HG^R(P)$).

3 Coinductive SLDNF Resolution

Negation causes many problems in logic programming (e.g., nonmonotonicity). For example, one can write programs whose meaning is hard to interpret, e.g., p :- not (p). and whose *completion* [3] is inconsistent. We use the notation nt(A) to denote negation as failure (*naf*) for a coinductive atom A; nt(A) is termed a naf-literal. Also note that without occurs-check, the unification equation X=f(X) means X is bound to f(f(f(....))) (an infinite rational term). From this point, we take all logic programs to be normal logic programs (that is, a logic program with zero or more negative literals in the body of a clause, and zero or one atom in the head) and finite (finite set of clauses with a finite set of alphabets).

Definition 3.1. (Syntax of co-LP with negation as failure): A *coinductive* logic program P is syntactically identical to a traditional (that is, *inductive*) logic program. However, predicates executed with co-SLD resolution (gfp semantics restricted to rational proofs) are declared as coinductive; all other predicates are assumed to be

inductive (i.e., lfp semantics is assumed). The syntax of declaring a clause for a coinductive predicate A of arity n is as follows:

coinductive (A/n).
A :- L_1, …, L_m.

where $m \geq 0$ and A is an atom (of arity n) of a general program P, L_i, $(0 \leq i \leq m)$, is a positive or naf-literal. □

The major considerations for incorporating negation into co-LP are: (i) *negation as failure*: infer nt(p) if p fails and *vice versa*, i.e., nt(p) fails if p succeeds, (ii) *negative coinductive hypothesis rule*: infer nt(p) if nt(p) is encountered again in the process of establishing nt(p), and (iii) *consistency in negation*, infer p from double negation, i.e., nt(nt(p)) = p. Next we extend co-SLD resolution so that naf-goals can also be executed. The extended operational semantics is termed *co-SLDNF resolution*. Co-SLDNF resolution further extends co-SLD resolution with negation. Essentially, it augments co-SLD with the *negative coinductive hypothesis rule,* which states that if a negated call nt(p) is encountered during resolution, and another call to nt(p) has been seen before in the same computation, then nt(p) coinductively succeeds. To implement co-SLDNF, the set of positive and negative (ancestor) calls has to be maintained in the *positive hypothesis table* (denoted χ+) and *negative hypothesis table* (denoted χ-) respectively. The operational semantics of co-LP with negation as failure is defined as an interleaving of co-SLD and negation as failure under the co-Herbrand model. Extending co-SLD to co-SLDNF, the goal {nt(A)} succeeds (or has a successful derivation) if {A} fails; likewise, the goal {nt(A)} fails (or has a failure derivation) if the goal {A} succeeds. We restrict $P \cup \{A\}$ to be *allowed* [10] (p.89) to prevent floundering and thus to ensure soundness. We also restrict ourselves to the rational Herbrand space. Since naf-literals may be nested, one must keep track of the context of each predicate occurring in the body of a clause, i.e., whether it is in the scope of odd or even number of negations. If a predicate is under the scope of even number of negations, it is said to occur in positive context, else it occurs in negative context. In co-SLDNF resolution, negated goals that are encountered should be remembered since negated goals can also succeed coinductively. Thus, the state is represented as (G, E, χ+, χ-) where G is the subgoal list (containing positive or negated goals), E is a system of term equations, χ+ is the set of ancestor calls occurring in positive context (i.e., in the scope of zero or an even number of negations). χ- is the set of ancestor calls occurring in negative context (i.e., in the scope of an odd number of negations). Further, we need a few more requisite concepts for the definition of co-SLDNF.

Given a co-LP P and an atom A in a query goal G, the set of all clauses with the same predicate symbol A in the head is called the *definition* of A. Further the *unifiable-definition* of A is the set of all clauses of $C_i = \{ H_i :- B_i \}$ (where $1 \leq i \leq n$) where A is unifiable with H_i. Each C_i of the unifiable-definition of A is called a *candidate clause* for A. Each candidate clause C_i of the form $\{H_i(\mathbf{t}_i) :- B_i.\}$ is *modified* to $\{H_i(\mathbf{x}_i) :- \mathbf{x}_i = \mathbf{t}_i, B_i.\}$, where $(\mathbf{x}_i = \mathbf{t}_i, B_i)$ refers to the *extended* body of the candidate clause, $\mathbf{t}_i$ is an n-tuple representing the arguments of the head of the clause C_i, B_i is a conjunction of goals, and $\mathbf{x}_i$ is an n-tuple of fresh unbound variables (that is, standardized apart). Let S_i be the extended body of the candidate clause C_i (that is, S_i is

($\mathbf{x}_i = \mathbf{t}_i$, B_i), for each i where $1 \leq i \leq n$). Then an *extension* G_i of G for A in negative context w.r.t. S_i is obtained by replacing A with $S_i\theta_i$ where $\theta_i = mgu(A, H_i)$. The *complete-extension* G' of G for A in negative context is obtained by the conjunction of the extension G_i for each S_i where $1 \leq i \leq n$. If there is no definition for A in P, then the *complete-extension* G' of G for A in negative context is obtained by replacing A with *false*. For example, given $G = nt(D_1, A, D_2)$ with the n-candidate clauses for A where its extended body is $S_i(\mathbf{x}_i)$ where $1 \leq i \leq n$. Then the complete-extension G' of G for A will be: $G' = (nt(D_1, S_1(\mathbf{x}_1)\theta_1, D_2), \ldots, nt(D_1, S_n(\mathbf{x}_n)\theta_n, D_2))$. Intuitively, the concept of the complete-extension captures the idea of *negation as failure* that the proof of A in negative context (that is, a negative subgoal, $\neg A$) requires the failure of all the possibilities of A. That is, $\neg A \leftrightarrow \neg(H_1 \vee \ldots \vee H_n) \leftrightarrow (\neg H_1 \wedge \ldots \wedge \neg H_n)$ where H_i is a candidate clause of A. Thus the complete-extension embraces naturally the dual concepts of (i) the negation of the disjunctive subgoals (the disjunction in negative context) with (ii) the conjunction of the negated subgoals. For example, $nt(D_1, (S_1(\mathbf{x}_1)\theta_1 \vee \ldots \vee S_n(\mathbf{x}_n)\theta_n), D_2)$ is equivalent to $(nt(D_1, S_1(\mathbf{x}_1)\theta_1, D_2), \ldots, nt(D_1, S_n(\mathbf{x}_n)\theta_n, D_2)))$. Co-SLDNF resolution is defined as follows.

Definition 3.2. Co-SLDNF Resolution: Suppose we are in the state (G, E, χ+, χ-) where G is a list of goals containing an atom A, and E is a set of substitutions (environment).

(1) If A occurs in positive context, and A' $\in \chi$+ such that $\theta = mgu(A,A')$, then the next state is (G', Eθ, χ+, χ-), where G' is obtained by replacing A with $\square$.
(2) If A occurs in negative context, and A' $\in \chi$- such that $\theta = mgu(A,A')$, then the next state is (G', Eθ, χ+, χ-), where G' is obtained by replacing A with false.
(3) If A occurs in positive context, and A' $\in \chi$- such that $\theta = mgu(A,A')$, then the next state is (G', E, χ+, χ-), where G' is obtained by replacing A with false.
(4) If A occurs in negative context, and A' $\in \chi$+ such that $\theta = mgu(A,A')$, then the next state is (G', E, χ+, χ-), where G' is obtained by replacing A with $\square$.
(5) If A occurs in positive context and there is no A' $\in (\chi+ \cup \chi-)$ that unifies with A, then the next state is (G', E', $\{A\} \cup \chi+$, χ-) where G' is obtained by expanding A in G via normal call expansion using a (nondeterministically chosen) clause C_i (where $1 \leq i \leq n$) whose head atom is unifiable with A with E' as the new system of equations obtained.
(6) If A occurs in negative context, and there is no A' $\in (\chi+ \cup \chi-)$ that unifies with A, then the next state is (G', E', χ+, $\{A\} \cup \chi-$) where G' is obtained by the complete-extension of G for A.
(7) If A occurs in positive or negative context and there are no matching clauses for A, and there is no A' $\in (\chi+ \cup \chi-)$ such that A and A' are unifiable, then the next state is (G', E, χ+, $\{A\} \cup \chi-$), where G' is obtained by replacing A with false.
(8) (a) nt(…, false, …) reduces to $\square$, (b) nt(A, $\square$, B) reduces to nt(A, B) where A and B represent conjunction of subgoals, and (c) nt($\square$) reduces to false.

Note (i) that the result of expanding a subgoal with a unit clause in step (5) and (6) is an empty clause ($\square$). (ii) When an initial query goal reduces to an empty clause ($\square$), it denotes a success (denoted by [success]) with the corresponding E as the solution, and (ii) when an initial query goal reduces to false, it denotes a fail (denoted by [fail]). $\square$

Definition 3.3. (Co-SLDNF derivation): Co-SLDNF derivation of the goal G of program P is a sequence of co-SLDNF resolution steps (of **Definition 3.2**) with a selected subgoal A, consisting of (1) a sequence (G_i, E_i, χ_i+, χ_i-) of state ($i \geq 0$), of (a) a sequence G_0, G_1, ... of goal, (b) a sequence E_0, E_1, ... of mgu's, (c) a sequence χ_0+, χ_1+, ... of the positive hypothesis table, (d) χ_0-, χ_1-, ... of the negative hypothesis table, where (G_0, E_0, χ_0+, χ_0-) = (G, ∅, ∅, ∅) as the initial state, and (2) for step (5) or step (6) of **Definition 3.2**, a sequence C_1, C_2, ... of variants of program clauses of P where G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} where $E_{i+1} = E_i\theta_{i+1}$ and (χ_{i+1}+, χ_{i+1}-) as its resulting positive and negative hypothesis tables. (3) If a co-SLDNF derivation from G results in an empty clause of query □, that is, the final state of (□, E_i, χ_i+, χ_i-), then it is a successful co-SLDNF derivation, and a derivation fails if a state is reached in the subgoal-list which is non-empty and no transitions are possible from this state (as defined in **Definition 3.2**). □

Note that there could be more than one derivation from a node if there is more than one step available for the selected subgoal (e.g., many clauses are applicable for the expansion rules of step (5) or step (6) in **Definition 3.2**). A co-SLDNF resolution step may involve expanding with a program clause for **Definition 3.2** (5) or (6) with the initial goal $G = G_0$, and the initial state of (G_0, E_0, χ_0+, χ_0-) = (G, ∅, ∅, ∅), and $E_{i+1} = E_i\theta_{i+1}$ (and so on) may look as follows:

$$(G_0, E_0, \chi_0+, \chi_0-) \xrightarrow{C_1,\theta_1} (G_1, E_1, \chi_1+, \chi_1-) \xrightarrow{C_2,\theta_2} (G_2, E_2, \chi_2+, \chi_2-) \xrightarrow{C_3,\theta_3} \ ...$$

Further, for sake of the notational simplicity, we use the disjunctive form for step (6) of **Definition 3.2** instead of the conjunctive form for our examples. For example, nt(D_1, ($S_1(\mathbf{x}_1)\theta_1$; … ; $S_n(\mathbf{x}_n)\theta_n$), D_2) is used for (nt(D_1, $S_1(\mathbf{x}_1)\theta_1$, D_2), …, nt(D_1, $S_n(\mathbf{x}_n)\theta_n$, D_2))) where "∨" is denoted by ";" as we adapt the conventional Prolog disjunctive operator for convenience. Next restricting ourselves to the rational Herbrand space, the *success set* and *finitely-failed* set of co-SLDNF are defined. Let [SS] be the (coinductive) Success Set and let [FF] be the (coinductive) Finite-Failure Set. We assume that a query is a subset of the signed atoms from the given program P.

Definition 3.4. (Success Set and Finite-Failure Set of co-LP with negation) Let P be a normal co-LP program with its rational Herbrand space. Then:

(1) [SS]={ A | A ∈ HB^R(P), the goal { A } →* □ }
(2) [FF]={ A | A ∈ HB^R(P), the goal { nt(A) } →* □ },

where →* denotes a co-SLDNF derivation of length 0 or more, and □ denotes an empty clause { }. □

Note that the third possibility is an irrational (infinite) derivation, considered to be *undefined* in the rational space.

4 Illustrative Examples

Next we consider a few illustrative examples for co-SLDNF resolution. With the example programs and queries, we also consider their model (fixed point) and

program completion [3]. Note that we show co-SLDNF derivation in the left column and the annotation in the right column with co-SLDNF step numbers from Def. 3.2.

Example 4.1. Consider the following program NP1:

NP1: p :- nt(q).
q :- nt(p).

First, consider the query Q1 = **?- p** generating the following derivation:

	({p}, {}, {}, {})	by (5)
→	({nt(q)}, {}, {p},{})	by (6)
→	({nt(nt(p))}, {}, {p}, {q})	by (1)
→	({}, {}, {p}, {q})	[success]

Second, consider the query Q2 = **?- nt(p)** generating the following derivation:

	({nt(p)}, {}, {}, {})	by (6)
→	({nt(nt(q))}, {}, {},{p})	by (5)
→	({nt(nt(nt(p)))}, {}, {q}, {p})	by (2)
→	({}, {}, {q}, {p})	[success]

Third, the query Q3 = **?- p, nt(p)** will generate the following derivation:

	({p, nt(p)}, {}, {}, {})	by (5)
→	({nt(q), nt(p)}, {}, {p},{})	by (6)
→	({nt(nt(p)), nt(p)}, {}, {p}, {q})	by (1)
→	({nt(p)}, {}, {p}, {q})	[success] for p; nt(p) by (4)
→	({nt(□)}, {}, {p}, {q})	by 8(c)
→	({false}, {}, {p}, {q})	[fail]

Finally the query Q3 = **?- p, q.** will generate the following derivation:

	({p, q}, {}, {}, {})	by (5)
→	({nt(q), q}, {}, {p},{})	by (6)
→	({nt(nt(p)), q}, {}, {p}, {q})	by (1)
→	({q}, {}, {p}, {q})	[success] for p; q by (3)
→	({false}, {}, {q}, {p})	[fail]

Note that the queries Q1 and Q2 succeed whereas the queries Q3 and Q4 fail. We should note that the above program NP1 has two fixed points (two models, M1A and M1B where M1A={p}, M1B={q}, M1A∩M1B=∅), that are not consistent with each other. As we noted, the query **?- nt(p)** is true with M1B={q}, while the query **?- p** is true with M1A={p}. Thus, computing with (maximal) fixed point semantics in presence of negation can be troublesome and seemingly lead to contradictions; one has to be careful that given a query, different parts of the query are not computed w.r.t. different fixed points. Moreover, the query **?-p, nt(p)** will never succeed if we are aware of the context (of a particular fixed point being used). However, if the subgoals **p** and **nt(p)** are evaluated separately and the results conjoined without enforcing their consistency, then it will wrongly succeed. To ensure consistency of the partial interpretation, the sets χ+ and χ- are employed in our operational semantics; they in effect keep track of the particular fixed point(s) under use.

Example 4.2. Consider the following program NP2:

NP2: p :- p.

First, consider the query Q1 = **?- p** generating the following derivation:

	({p}, {}, {}, {})	by (5)
→	({p}, {}, {p},{})	by (1)
→	({}, {}, {p}, {})	[success]

Second, consider the query Q2 = **?- nt(p)** generating the following derivation:

	({nt(p)}, {}, {}, {})	by (6)
→	({nt(p)}, {}, {},{p})	by (2)
→	({nt(false)}, {}, {}, {p})	by (8a)
→	({}, {}, {}, {p})	[success]

Third, consider the query Q3 = **?- p, nt(p)** generating the following derivation:

	({p, nt(p)}, {}, {}, {})	by (5)
→	({p, nt(p)}, {}, {p},{})	by (1)
→	({nt(p)}, {}, {p}, {})	[success] for p; nt(p) by (4)
→	({nt(□)}, {}, {p}, {})	by (8c)
→	({false}, {}, {p}, {})	[fail]

Both queries Q1 and Q2 succeed with NP2. The program NP2 has two fixed points (two models M2A and M2B where M2A={p} and M2B={}). Further M2A∩M2B = ∅ and M2B ⊆ M2A. M2A is the greatest fixed point and M2B is the least fixed point of NP2. As we noted, the query **?- nt(p)** is true and the query **?- p** is false with M2B = {}, while the query **?- p** is true with M2A={p}. This type of the behavior of co-LP with co-SLDNF seems to be confusing and counter-intuitive. However, as we noted earlier with NP1, this type of behavior is indeed advantageous as we extend traditional LP into the realm of modal reasoning. Clearly, the addition of a clause like { **p :- p.** } to a program extends each of its initial models into two models where one includes **p** and the other does not include **p**. Further, co-SLDNF enforces the consistency of the query result causing the query **?- p, nt(p)** to fail. However, the query Q4 = **?- (p; nt(p))** will then generate the following derivation with program NP2 and succeed (in fact, there are two distinct success derivations one for p and another for nt(p)):

	({p ; nt(p)}, {}, {}, {})	by (5)
→	({p ; nt(p)}, {}, {p},{})	by (1)
→	({□ ; nt(p)}, {}, {p}, {})	[success] for p; nt(p) by (4)
→	({}, {}, {p}, {})	[success]

Example 4.3. Consider the following program NP3:

NP3: p :- nt(p).

First, consider the query Q1 = **?- p** generating the following derivation:

	({p}, {}, {}, {})	by (5)
→	({nt(p)}, {}, {p},{})	by (4)

→ ({nt(□)}, {}, {p}, {}) by (8c)
→ ({false}, {}, {p}, {}) [fail]

Second, consider the query Q2 = **?- nt(p)** generating the following derivation:

({nt(p)}, {}, {}, {}) by (6)
→ ({nt(nt(p))}, {}, {},{p}) by (3)
→ ({nt(nt(false))}, {}, {}, {p}) by (8a)
→ ({nt(□)}, {}, {}, {p}) by (8c)
→ ({false}, {}, {}, {p}) [fail]

Third, consider the query Q3 = **?- (p; nt(p))** generating the following derivation:

({p ; nt(p)}, {}, {}, {}) by (5)
→ ({nt(p); nt(p)}, {}, {p},{}) by (4)
→ ({nt(□); nt(p)}, {}, {p}, {}) by (8c)
→ ({false; nt(p)}, {}, {p}, {}) [fail] for subgoal p;
→ ({nt(p)}, {}, {p}, {}) by (4)
→ ({nt(□)}, {}, {p}, {}) by (8c)
→ ({false}, {}, {p}, {}) [fail]

The program NP3 has no fixed point (no model), in contrast to the program NP2 which has two fixed points {} and {p}. Further, the query ?- **(p ; nt(p))** provides a validation test for NP3 w.r.t. **p** whether NP3 is consistent or not. Consider the program completion CP2 (of NP2) which is { p ≡ p }. In contrast, there is no consistent completion of program for NP3 where its completion of program CP3 of NP3 is { p ≡ ¬p }, a contradiction.

Example 4.4. Consider the following program NP4:

NP4: p :- nt(q).

and reconsider program NP1:

NP1: p :- nt(q).
q :- nt(p).

NP4 has a model MP4 = {p} whereas NP1 has two models MP1A = {p} and MP1B = {q} as we noted earlier. Further the completion of the program CP4 for NP4 is: { p ≡ ¬q. q ≡ false. }, and the completion of the program CP1 for NP1 is: { p ≡ ¬q. q ≡ ¬p. }. With co-SLDNF semantics, the query **?- p** succeeds with NP1 and NP4 whereas the query **?- q** succeeds with NP1 but not with NP4. This is consistent with the semantics of the program completion of these two programs. After discussing the correctness of co-SLDNF resolution, we will show the equivalence of a logic program under co-SLDNF semantics and the semantics of program completion w.r.t. the result of a successful co-SLDNF derivation, as we noted for this example.

Example 4.5. Consider the following program NP5 with three clauses:

NP5: p :- q.
p :- r.
r.

NP5 has one fixed point (model), which is the least fixed point, MP5 = { p, r }. The query **?- nt(p)** will generate the following transition sequence:

	({nt(p)}, {}, {}, {})	by (6)
→	({nt(q), nt(r)}, {}, {},{p})	by (7)
→	({nt(false), nt(r)}, {}, {}, {p,q})	by (8a)
→	({nt(r)}, {}, {}, {p,q})	by (6)
→	({nt(□)}, {}, {}, {p,q,r})	by (8c)
→	({false}, {}, {}, {p,q,r})	[fail]

We used propositional logic programs in the examples above, but these examples could just as easily be illustrated with predicate logic programs. Note that co-SLDNF resolution allows one to develop elegant implementations of modal logics[12]). In addition, co-SLDNF resolution provides the capability of non-monotonic inference (e.g., predicate Answer Set Programming [12]) that can be used to develop novel and effective first-order modal non-monotonic inference engines.

5 Correctness of co-SLDNF Resolution

The declarative semantics of a co-inductive logic program with negation as failure (co-SLDNF) is an extension of a stratified-interleaving (of coinductive and inductive predicates) of the minimal Herbrand model and the maximal Herbrand model semantics with the restriction of rational trees. This allows the universe of terms to contain rational (that is, rationally infinite) terms, in addition to the traditional finite terms. As we noted earlier with program **NP3** in **Example 4.3**, negation in logic program with coinduction may generate nonmonotonicity and thus there exists no consistent co-Herbrand model. For a declarative semantics to co-LP with negation as failure, we rely on the work of Fitting [6] (Kripke-Kleene semantics with three-valued logic), extended by Fages [5] for stable models with completion of a program. Their framework, which maintains a pair of sets (corresponding to a partial interpretation of success set and failure set, resulting in a partial model) provides a sound theoretical basis for the declarative semantics of co-SLDNF. As we noted earlier, we restrict Fitting's and Fages's results within the scope of rational LP over the rational space. We summarize this framework next.

Definition 5.1. (Pair-set and pair-mapping): Let P be a normal logic program, with its rational Herbrand Space $HS^R(P)$, and let $(M, N) \in 2^{HB}x2^{HB}$ (where HB is the rational Herbrand base $HB^R(P)$) be a partial interpretation. Then the pair-mapping (T_P^+, T_P^-) for defining the pair-set (M, N) are as follows:

$$T_P^+(M,N) = \{\text{head}(R) \mid R \in HG^R(P), \text{pos}(R) \subseteq M, \text{neg}(R) \subseteq N\},$$
$$T_P^-(M,N) = \{A \mid \forall R \in HG^R(P), \text{head}(R)=A \rightarrow \text{pos}(R) \cap N \neq \emptyset \vee \text{neg}(R) \cap M \neq \emptyset\}$$

where head(R) is the head atom of a clause R, pos(R) is the set of positive atoms in the body of R, and neg(R) is the set of atoms under negation. □

It is noteworthy that the T_P^+ operator w.r.t. M of the pair set (M, N) is identical to the *immediate consequence operator* T_P [10] where $T_P(I) = \{ \text{head}(R) \mid R \in HG^R(P), I \models \text{body}(R) \}$ where body(R) is the set of positive and negative literals occurring in the body of a clause R. We recall [10] (also noted by Fages [5] in Proposition 4.1) that a Herbrand interpretation I (that is, $I \subseteq HG(P)$) is a model of comp(P) iff I is a fixed

point of T_p. Intuitively, the outcome of the operator T_P^+ is to compute a success set. In contrast, the outcome of T_P^- is to compute the set of atoms guaranteed to fail. Thus the pair-mapping (T_P^+, T_P^-) specifies essentially a consistent pair of a success set and a finite-failure set. Further the pair-set (M, N) of the pair-mapping (T_P^+, T_P^-) enjoys monotonicity and gives Herbrand models (fixed points) under certain conditions as follows.

Theorem 5.1. (Fages [5], Proposition 4.2, 4.3, 4.4, 4.5). Let P be an infinite LP. Then:

(1) If $M \cap N = \varnothing$ then $T_P^+(M, N) \cap T_P^-(M, N) = \varnothing$.
(2) $< T_P^+, T_P^- >$ is monotonic in the lattice $2^{HB}x2^{HB}$ (where HB is the Herbrand Base) ordered by pair inclusion $\subseteq$, that is, $(M1, N1) \subseteq (M2, N2)$ implies that $< T_P^+, T_P^- > (M1, N1) \subseteq < T_P^+, T_P^- > (M2, N2)$.
(3) If $M \cap N = \varnothing$ and $(M, N) \subseteq < T_P^+, T_P^- > (M, N)$ then there exists a fixed point (M', N') of $< T_P^+, T_P^- >$ such that $(M, N) \subseteq (M', N')$ and $M' \cap N' = \varnothing$.
(4) If (M, N) is a fixed point of $< T_P^+, T_P^- >$, $M \cap N = \varnothing$ and $M \cup N = HB$, then M is a Herbrand Model (HM) of comp(P). □

Note that the pair mapping $<T_P^+, T_P^->$ and the pair-set (M, N) are the declarative counterparts of co-SLDNF resolution; the set (M, N) corresponds to (χ+, χ-) of **Definition 5.1**. Thus Fages's theorem above captures the declarative semantics of co-SLDNF resolution of general infinite LP; and we need to see whether **Theorem 5.1** for a set of finite rational logic programs (which is a subset of infinite logic programs) over the rational space. The proofs for **Theorem 5.1 (1-2)** are straightforward. For **Theorem 5.1 (3)** with $HG^R(P)$, the immediate consequence, that is, the pair-set (M,N) by the pair-mapping $< T_P^+, T_P^- >$ applied each time is rational, and this is true for any finite n steps where $n \geq 0$. This is due to the earlier observations: (i) that the algebra of rational trees and the algebra of infinite trees are elementarily equivalent, (ii) that there is no isolated irrational atom as result of the pair-mapping and pair-set for rational LP over rational space, and (iii) that any irrational atom as result of an infinite derivation in this context should have a *rational cover*, as noted in [9], which could be characterized by the (interim) rational atom observed in each step of the derivation. For **Theorem 5.1 (4)**, there are two cases to consider for each atom resulting in a fixed point: rational or irrational. For the rational case, it is straightforward to see that it will be eventually derived by the pair-mapping as there is a rational cover that eventually converges to the rational atom, and the rational model contains the fixed point. For the irrational case, it does not exist in the program's rational space but there is a rational cover converging into the irrational fixed point over infinity. That is, there is a fixed point but its irrational atom is not in the rational model. In this case, co-SLDNF derivation (tree) will be irrational, and will be labeled *undefined* (even though it will succeed or fail after an infinite number of steps [infinite success or infinite failure]). Thus we have the following corollary.

Corollary 5.2. (Fages's Theorem for Rational Models): Let P be a normal coinductive logic program. Let (T_P^+, T_P^-) be the corresponding pair mappings [**Definition 5.1**]. Given a pair set $(M, N) \in 2^{HB}x2^{HB}$ [where HB is the rational Herbrand base $HB^R(P)$] with $M \cap N = \varnothing$ and $(M, N) \subseteq < T_P^+, T_P^- > (M, N)$ then there exists a fixed

point (M', N') of $< T_P^+, T_P^- >$ such that (M, N) $\subseteq$ (M', N') and M'$\cap$N'=$\emptyset$. If (M', N') is a fixed point of $< T_P^+, T_P^- >$, M'$\cap$N'=$\emptyset$ and M'$\cup$N'=$HB^R(P)$, then M' is a (Rational) Herbrand model of P (denoted $HM^R(P)$). □

Moreover we can establish that a model of P w.r.t. a successful co-SLDNF derivation is also a model of comp(P). Later we show that under a successful co-SLDNF resolution, a program P and its completion, comp(P), coincide. As we noted earlier, the pair mapping $<T_P^+, T_P^->$ and the pair-set (M, N) are the declarative counterparts of co-SLDNF resolution; the set (M, N) corresponds to (χ+, χ-) of **Definition 3.2**. Further we note that there may be more than one fixed points (which are possibly inconsistent with each other). Note that $HM^R(P)$ is also a model of comp(P) since comp(P) coincides with P under co-SLDNF, as we show later. As noted earlier, the condition of mutual exclusion (that is, M$\cap$N = $\emptyset$) keeps the pair-set (M,N) monotonic and consistent under the pair-mapping. The pair-mapping with the pair-set maintains the consistency of truth value assigned to an atom **p**. Thus, cases where both **p** and **nt(p)** are assigned true, or both are assigned false, are rejected. Next we show that P coincides with comp(P) under co-SLDNF. First we recall the work of Apt, Blair and Walker [1] for supported interpretation and supported model.

Definition 5.2. (Supported Interpretation [1]). An interpretation I of a general program P is *supported* if for each A$\in$I there exists a clause $A_1 \leftarrow L_1,...,L_n$ in P and a substitution θ such that $I \models L_1\theta, ..., L_n\theta$, $A=A_1\theta$, and each $L_i\theta$ is ground. Thus I is supported iff for each A$\in$I there exists a clause in HG(P) with head A whose body is true in I. □

Theorem 5.3 (Apt, Blair, and Walker [1], Shepherdson [15]). Let P be a general program. Then: (1) I is a model of P iff $T_P(I) \subseteq I$. (2) I is supported iff $T_P(I) \supseteq I$. (3) I is a supported model of P iff it is a fixed point of T_P, i.e., $T_P(I) = I$. □

We use these results to show that comp(P) and P coincide under co-SLDNF resolution. The positive and negative coinductive hypothesis tables (χ+ and χ-) of co-SLDNF are equivalent to the pair-set under the pair-mapping and thus enjoy (a) monotonicity, (b) mutual exclusion (disjoint), (c) consistency. First (1), it is straightforward to see that in a successful co-SLDNF derivation the coinductive hypothesis tables χ+ and χ- serve as a partial model (that is, if the body of a selected clause is true in χ+ and χ- then its head is also true ($A \leftarrow L_1,...,L_n$)). Second (2), it is also straightforward to see that a successful co-SLDNF derivation constrains the coinductive hypothesis tables χ+ and χ- at each step to stay supported (that is, if the head is true then the body of the clause is true: ($A \rightarrow L_1,...,L_n$)). By co-inductive hypothesis rule, the selected query subgoal (say, A) is placed first in χ+ (resp. χ-) depending on its positive (resp. negative) context. The rest of the derivation is to find a right selection of clauses ($A \rightarrow L_1,...,L_n$) whose head-atom is unifiable with A, and whose body is true using normal logic programming expansion or via negative or positive coinductive hypothesis rule. Thus, it follows from above that a coinductive logic program ($A \leftarrow L_1,...,L_n$) is equivalent to its completed program ($A \leftrightarrow L_1,...,L_n$) under co-SLDNF resolution. Next, correctness of co-SLDNF is proved by equating the operational and declarative semantics, as follows.

Theorem 5.4. (Soundness and Completeness of co-SLDNF). Let P be a general program over its rational Herbrand Space.

(1) (Soundness of co-SLDNF): (a) If a goal {A} has a successful derivation in program P with co-SLDNF, then A is true, i.e., there is a model $HM^R(P)$ where $A \in HM^R(P)$. (b) Similarly, if a goal { nt(A) } has a successful derivation in program P, then nt(A) is true in program P, i.e., there is a model $HM^R(P)$ such that $A \in HB^R(P) \backslash HM^R(P)$.
(2) (Completeness of co-SLDNF): (a) If $A \in HM^R(P)$, then A has a successful co-SLDNF derivation or an irrational derivation. Further (b) if $A \in HB^R(P) \backslash HM^R(P)$, then nt(A) has a successful co-SLDNF derivation or an irrational derivation. □

Note that the coincidence of P and comp(P) under co-SLDNF is important. If comp(P) is not consistent, say w.r.t. an atom **p**, then there is no successful rational derivation of **p** or **nt(p)**.

Example 5.1. Consider the following program IP1 = { **q :- p(a). p(X) :- p(f(X).** }. This is an example of an irrational derivation (irrational proof tree) since for query **?- q** the derivation (q → p(a) → p(f(a)) → p(f(f(a))) → …) is non-terminating. Similarly the negated query **?- nt(q)** is also non-terminating (i.e., nt(q) → nt(p(a)) → nt(p(f(a))) → nt(p(f(f(a)))) → ...). But it is clear that both **q** and **p(a)** are in the rational Herbrand base $HB^R(IP1)$. Moreover, **q** and **p(a)** are not in $HM^R(IP1)$ but in $HB^R(IP1) \backslash HM^R(IP1)$ as there is no rational derivation tree for **q** and **p(a)**. Further, for the second clause { **p(X) :- p(f(X).** }, there is only one ground (rational) atom **p(X′),** where X′=f(X′)=f(f(f(...))), which satisfies the clause and makes **p(X)** true; all other finite or rational atoms **p(Y)** are false. Thus the ground atom **p(f(f(f(...))))** (**p(X)** where **X=f(X)**) is in $HM^R(P)$, and all other finite and rational atoms **p(Y)** where **Y ≠ f(Y)** should be in $HB^R(P) \backslash HM^R(P)$ as one would expect. The derivation of query **?- q** which is irrational hence will not terminate. Thus if there is no rational coinductive proof for an atom **G**, then the query **?- G** will have an irrational infinite derivation.

Example 5.2. Consider the program NP3A as follows:

NP3A: p :- nt(p), q.

The queries Q1 – Q3 of NP3 in **Example 4.3** will generate the same result for NP3A. Its program completion NP3B (denoted comp(NP3A)) is then defined [12] as follows:

NP3B: p :- nt(p), q.
nt(p) :- nt(nt(p), q).
nt(q).

The query Q1 = **?- p** will fail with NP3B while the query Q2 = **?- nt(p)** will succeed with the following derivation:

	({nt(p)}, {}, {}, {})	by (6)
→	({nt(nt(p), q)}, {}, {},{p})	by (3)
→	({nt(nt(false), q)}, {}, {}, {p})	by (8a)
→	({nt(□**, q)**}, {}, {q}, {p})	by (8c)
→	({nt(**q**)}, {}, {q}, {p})	by (6)
→	({}, {}, {q}, {p})	[success]

Recall that NP3 has no fixed point (no model) as its program completion is inconsistent. In contrast, NP3A has a model MP3A (={}) where its program completion CP3A is {p ≡ (¬p∧q) ≡ ¬(p∨¬q). q ≡ false.}, to illustrate the nonmonotonic capability. In summary, co-LP with co-SLDNF provides a powerful, effective and practical operational semantics for Fitting's Kripke-Kleene three-valued logic [6] with restriction of rationality with modal and nonmonotonic capability.

6 Applications of co-LP with co-SLDNF

Some examples of exploratory applications of co-LP with co-SLDNF can be found in [12], for predicate Answer Set Programming (ASP) solver, Boolean SAT solver, model checking and verification, and modal nonmonotonic inference. One major application of co-SLDNF is the top-down goal-directed predicate ASP solver [13]. Here we present an example of Boolean SAT solver to show how one can quickly and elegantly program Boolean SAT solver [14] using co-SLDNF resolution.

Example 6.1. Consider two programs BP1 and BP2 where each is a "naïve" *coinductive SAT solver* (co-SAT Solver) for propositional Boolean formulas:

```
BP1:   pos(X) :- nt(neg(X)).
       neg(X) :- nt(pos(X)).
BP2:   t(X) :- t(X).
```

Note that with a minor variation, BP1 is a predicate version of NP1 = { p :- nt(q). q :- nt(p). }, and BP2 of NP2 = { p :- p. }. With BP1, the rules assert that the predicates pos(X) and neg(X) have mutually exclusive values, i.e., a propositional symbol X cannot be set simultaneously both to true and false. Next, any well-formed propositional Boolean formula constructed from a set of propositional symbols and logical connectives { ∧, ∨, ¬} is now translated into a query that is executed under co-SLDNF resolution. First (1), each positive propositional symbol p will be transformed into pos(p), and each negated propositional symbol into neg(p). The Boolean operator AND ("∧")will be translated into "," (Prolog's AND-operator), while the OR (or "∨") operator will be translated to ";" (Prolog's OR-operator). Thus, the Boolean expression (p1 ∨ p2) ∧ (p1 ∨ ¬ p3) ∧ (¬ p2 ∨ ¬ p4) will be translated into the query: ?- (pos(p1); pos(p2)), (pos(p1); neg(p3)), (neg(p2); neg(p4)). This query can be executed under co-SLDNF resolution to get a consistent assignment for propositional variables p1 through p4. The assignments will be recorded in the positive and negative coinductive hypothesis tables (if one were to build an actual SAT solver, then a primitive will be needed that should be called after the query to print the contents of the two hypotheses tables). Indeed a meta-interpreter for co-SLDNF resolution has been prototyped by us and used to implement the naïve SAT solver algorithm. For the query above our system will print as one of the answers:

```
positive_hypo ==> [pos(p1), neg(p2)]
negative_hypo ==> [neg(p1), pos(p2)]
```

which outputs the solution p1=true and p2=false. More solutions can be obtained by backtracking. Similarly for BP2, Boolean formula is transformed into co-LP query as follows: (1) a positive literal p1 as **t(p1)**, (2) a negative literal ¬p1 as **nt(t(p1))**, (3)

(p1∧p2) as **(t(p1), t(p2))**, (4) (p1∨p2) as **(t(p1); t(p2))**, (5) (p1 ∨ p2) ∧ (p1 ∨ ¬ p3) ∧ (¬ p2 ∨ ¬ p4) as **((t(p1);t(p2)), (t(p1); nt(t(p3)), (nt(t(p2); nt(t(p4)))**, and so on. The derivation of BP2 is very similar to that of NP2.

7 Conclusion and Future Work

Coinductive logic programming realized via co-SLD resolution has many practical applications. It is natural to consider extending coinductive logic programming with negation as failure since negation is required for almost all practical applications of logic programming. In this paper we presented co-SLDNF resolution, which extends Simon *et al*'s co-SLD resolution with negation as failure. Co-LP with co-SLDNF resolution provides a powerful, practical and efficient operational semantics for Fitting's Kripke-Kleene three-valued logic with restriction of rationality. Co-SLDNF resolution has many practical applications, most notably to realizing goal-directed execution strategies for answer set programming extended with predicates.

Acknowledgments. We thank Peter Stuckey for many helpful discussions.

References

[1] Apt, K., Blair, H., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann Publishers, San Francisco (1988)
[2] Barwise, J., Moss, L.: Vicious Circles: On the Mathematics of Non-Wellfounded Phenom-ena. CSLI Publications (1996)
[3] Clark, K.L.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Prenum Press, New York (1978)
[4] Colmerauer, A.: Prolog and Infinite Trees. In: Clark, K.L., Tarnlund, S.-A. (eds.) Logic Programming, pp. 293–322. Prenum Press, New York (1978)
[5] Fages, F.: Consistency of Clark's Completion and Existence of Stable Models. Journal of Methods of Logic in Computer Science 1, 51–60 (1994)
[6] Fitting, M.: A Kripke-Kleene Semantics for Logic Programs. Journal of Logic Programming 2, 295–312 (1985)
[7] Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications (Tutorial Paper). In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007)
[8] Jaffar, J., Lassez, J.-L.: Maher, M. J.: Prolog-II as an Instance of the Logic Programming Language Scheme. In: Wirsing, M. (ed.) Formal Descriptions of Programming Concepts III, pp. 275–299. North-Holland, Amsterdam (1986)
[9] Jaffar, J., Stuckey, P.: Semantics of Infinite Tree Logic Programming. Theoretical Computer Science 46(2-3), 141–158 (1986)
[10] Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987)
[11] Maher, M.J.: Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In: Proc. 3rd Logic in Computer Science Conf., Edinburgh, UK, pp. 348–357 (1988)
[12] Min, R.: Predicate Answer Set Programming with Coinduction. Ph.D. Dissertation, Department of Computer Science. The University of Texas at Dallas (2009), `http://www.utdallas.edu/~rkm010300/research/Min2009Thesis.pdf`

[13] Min, R., Bansal, A., Gupta, G.: Towards Predicate Answer Set Programming Via Coinductive Logic Programming. In: AIAI 2009 (2009)
[14] Min, R., Gupta, G.: Coinductive Logic Programming and Its Application to Boolean Sat. In: Flairs 2009 (2009)
[15] Shepherdson, J.: Negation in Logic Programming. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 19–88. Morgan Kaufmann Pub., San Francisco (1988)
[16] Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007)
[17] Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive Logic Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006)

Refining Exceptions in Four-Valued Logic

Susumu Nishimura

Dept. of Mathematics, Graduate School of Science, Kyoto University
Sakyo-ku, Kyoto 606-8502, Japan
susumu@math.kyoto-u.ac.jp

Abstract. This paper discusses refinement of programs that may raise and catch exceptions. We show that exceptions are expressed by a class of predicate transformers built on Arieli and Avron's four-valued logic and develop a refinement framework for the four-valued predicate transformers. The resulting framework enjoys several refinement laws that are useful for stepwise refinement of programs involving exception handling and partial predicates. We demonstrate some typical usages of the refinement laws in the proposed framework by a few examples of program transformation.

1 Introduction

Program refinement has been intensively studied in the framework of refinement calculus [3,16]. Refinement calculus identifies each program with a predicate transformer and formally justifies refinement of programs by means of the so-called refinement relation that is induced from the logical entailment. Although refinement calculus is successfully applied to a certain extension of Dijkstra's guarded command language [6], fundamental difficulties arise when we try to extend the language with *exceptions*.

First, since exceptional termination is not discriminated from non-termination in the predicate transformer semantics, a construct that catches exceptions would also catch non-termination, which is counter-intuitive from the operational point of view. Second, exceptions are not only raised explicitly by a command but also implicitly by a failure of computation (e.g., division by zero). In this paper, we argue the latter type of exceptions that are raised by *partial predicates*, whose truth value may not be defined. Partiality poses a foundational issue in developing the theory of refinement based on the classical logic, in which partiality is ruled out. For example, in Dijkstra's predicate transformer semantics, the weakest pre-condition of the conditional statement **if** p **then** S **else** T is specified by a formula $(p \Rightarrow S(\varphi)) \wedge (\neg p \Rightarrow T(\varphi))$ for any post-condition φ, but this formula is nonsensical in the classical logic when p is undefined.

King and Morgan [14] proposed a solution to the first problem by developing an extension to the traditional predicate transformer semantics for a language in which exceptions are explicitly raised by the command **exit** and are caught by the exception block construct **try** S **catch** T[1]. They specified the input of each predicate transformer by a pair of post-conditions $\langle \varphi_n, \varphi_e \rangle$, rather than by a single post-condition, where they write $wp(S, \varphi_n, \varphi_e)$ for the weakest pre-condition that guarantees the program S either

[1] This extends the exception block construct proposed in [14] with exception handling.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 113–127, 2010.

to normally terminate establishing φ_n or to exceptionally terminate establishing φ_e. The weakest pre-conditions for **exit** and the exception block are given as below:

$$wp(\mathbf{exit}, \varphi_n, \varphi_e) = \varphi_e, \qquad wp(\mathbf{try}\ S\ \mathbf{catch}\ T, \varphi_n, \varphi_e) = wp(S, \varphi_n, wp(T, \varphi_n, \varphi_e)).$$

The intuition behind these specifications are explained as follows. The **exit** command immediately causes an exceptional termination. Thus, the command is guaranteed to terminate (exceptionally) establishing the pre-condition φ_e. The exception block **try** S **catch** T executes S and terminates normally, if no exception is raised; If an exception is ever raised, the raised exception is caught and then processed by T to resume normal execution. Therefore, for the exception block to terminate establishing the pair $\langle \varphi_n, \varphi_e \rangle$ of post-conditions, S is either to normally terminate establishing φ_n or to exceptionally terminate establishing $wp(T, \varphi_n, \varphi_e)$, which guarantees T to terminate establishing the pair of conditions $\langle \varphi_n, \varphi_e \rangle$.

In this paper, we propose a refinement calculus for a language that may raise and catch exceptions, where exceptions can be raised not only by the **exit** command explicitly but also by the evaluation of partial predicates implicitly. For this, we develop our theory of program refinement in a predicate transformer semantics based on Arieli and Avron's four-valued logic [1,2].

The four-valued predicate transformer semantics can be easily derived from King and Morgan's, in the following way. First, we identify each statement S by a predicate transformer that maps a pair of (classical) predicates $\langle \varphi_n, \varphi_e \rangle$ to another pair of predicates $\langle \varphi'_n, \varphi_e \rangle$, where φ'_n is the weakest pre-condition computed by King and Morgan's predicate transformer *wp*. This definition is intended to guarantee the program S either to normally terminate establishing φ_n or to exceptionally terminate establishing φ_e, whenever the preceding statement normally terminates establishing φ'_n or exceptionally terminates establishing φ_e. Notice that the condition φ_e for exceptional termination is left unchanged by the transformer because no statement can cancel exceptional termination caused by the preceding statements.

Next, let us designate a classical predicate by a total function from the set of states to $\{0,1\}$, where 0 and 1 designates the two classical truth values (i.e., *false* and *true*, respectively). Then we identify each pair of predicates $\langle \varphi_n, \varphi_e \rangle$ by a single *four-valued predicate* φ such that $\varphi(\sigma) = \langle \varphi_n(\sigma), \varphi_e(\sigma) \rangle$ for every state σ. The range of the four-valued predicate is $\{\langle 1,0\rangle, \langle 0,1\rangle, \langle 0,0\rangle, \langle 1,1\rangle\}$, which we designate by **t**, **f**, $\bot$, and $\top$, respectively. This structure with four truth values gives rise to the so called Belnap's four-valued logic [4], which has been studied by Ginsberg in the generalized setting of bilattices [8] and was further examined by Fitting [7]. Arieli and Avron [1,2] introduced the notion of logical bilattices and developed the corresponding proof system.

The four-valued logic provides a firm logical basis for refining exceptions, as the original refinement calculus does for refining the guarded command language. The constructs for exceptions and others as well are concisely specified by the formulas of four-valued logic. The conditional control via partial predicates can be translated into a predicate transformer, where the undefinedness of partial predicates is denoted by the truth value $\bot$. The refinement relation is induced from the logical entailment (in the sense of four-valued logic), i.e., $S \sqsubseteq T$ iff $S(\varphi)$ entails $T(\varphi)$ for any post-condition φ.

We emphasize that we use the four-valued logic in two different ways. In the predicate transformer semantics, it is used for discriminating the possible termination behaviors (either, both, or none of normal termination and exceptional termination), while in modelling partial predicates, it is used as a many-valued logic that allows undefinedness. Although a three-valued logic would be sufficient for the latter purpose, we stick to the four-valued logic in developing the theory of refinement in order to achieve a smooth translation of conditional controls via partial predicates into four-valued predicate transformers. For a more neat characterization of partial predicates that adheres to the operational intuition, we also consider partial predicates in a three-valued sublogic, whose truth values are confined to **f**, **t**, and $\bot$. In later sections we exploit the properties of partial predicates in this three-valued sublogic.

Related work. It seems that there has been no attempt to formulate a predicate transformer semantics that gives a unifying account for both exceptions and partial predicates. The exception mechanism was formulated in terms of predicate transformers in King and Morgan's refinement calculus [14], which was further elaborated in [21]. Partial predicates are out of their concern, however. (If partial predicates are ignored at all, the refinement calculus of theirs and that of ours are essentially the same.)

Partial predicates in program logic have been intensively studied in the context of three-valued logic. For instance, the VDM specification language deals with undefinedness in a logic called LPF [12,13]; Bono et al. [5] formulated a Hoare logic with a third truth value denoting 'crash' of execution. Many other variants of three-valued logic have been proposed for the sake of a better treatment of partiality [19,13,17]. The three-valued logic, however, is not suitable for describing a predicate transformer semantics for exceptions, because the underlying predicate logic must be able to discriminate the four different status of termination. Hähnle [10] discussed that partiality should be dealt by underspecification, rather than by a value representing undefinedness in a many-valued logic. His argument is, however, about predicates in specification statements and does not consider exception catching.

Huisman and Jacobs [11] extended Hoare logic to deal with abrupt (exceptional) termination in Java programming language. They also formulated the mechanism of catching exceptions in their program logic by representing several different modes of exceptional termination by different forms of Hoare triple. In contrast to theirs, ours simply supports a single mode of exceptional termination. This does not imply ours are less expressive than theirs. Ours can simulate different modes of exceptional termination by introducing a special variable indicating the mode of termination.

Outline. The rest of the paper is organized as follows. Section 2 introduces the notion of bilattices and the four-valued logic. Section 3 specifies a set of program statements as four-valued predicate transformers and we identify the class of predicate transformers. The statements involve **exit**, exceptions blocks, and conditional controls via partial predicates. The logical connectives for partial predicates are also discussed. In Section 4, we investigate a set of refinement laws that hold for these statements and logical connectives. In Section 5, we apply the refinement laws to carry out some program transformations. Finally, Section 6 concludes the paper.

2 The Bilattice *FOUR* and the Four-Valued Logic

2.1 The Bilattice *FOUR* of Four Truth Values

Let *TWO* be the lattice of classical truth values of 0, 1 with the trivial order $0 < 1$. The bilattice *FOUR* is a structure obtained by a product construction $TWO \odot TWO$: it consists of four elements $\langle 1,0\rangle$, $\langle 0,1\rangle$, $\langle 0,0\rangle$, and $\langle 1,1\rangle$, which are alternatively written **t**, **f**, $\bot$, and $\top$, respectively. The bilattice has two lattice structures simultaneously (see the double Hasse diagram of Figure 1), each characterized by the partial orders $\leq_t$ and $\leq_k$ defined below.[2]

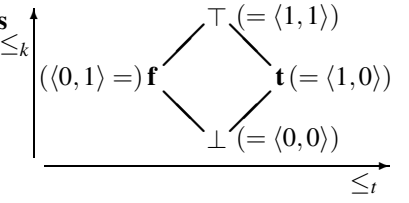


Fig. 1. The bilattice of four truth values

$$\langle x_1,y_1\rangle \leq_t \langle x_2,y_2\rangle \text{ iff } x_1 \leq x_2 \text{ and } y_2 \leq y_1,$$
$$\langle x_1,y_1\rangle \leq_k \langle x_2,y_2\rangle \text{ iff } x_1 \leq x_2 \text{ and } y_1 \leq y_2.$$

The $\leq_t$ order (resp. $\leq_k$ order) induces the meet $\wedge$ and join $\vee$ operators (resp. meet $\otimes$ and join $\oplus$ operators) The definitions are given below, where $\sqcap$ and $\sqcup$ stand for the meet and join in *TWO*, respectively.

$$\langle x_1,y_1\rangle \wedge \langle x_2,y_2\rangle = \langle x_1 \sqcap x_2, y_1 \sqcup y_2\rangle, \qquad \langle x_1,y_1\rangle \vee \langle x_2,y_2\rangle = \langle x_1 \sqcup x_2, y_1 \sqcap y_2\rangle,$$
$$\langle x_1,y_1\rangle \otimes \langle x_2,y_2\rangle = \langle x_1 \sqcap x_2, y_1 \sqcap y_2\rangle, \qquad \langle x_1,y_1\rangle \oplus \langle x_2,y_2\rangle = \langle x_1 \sqcup x_2, y_1 \sqcup y_2\rangle.$$

In addition, negation $\neg$ is defined by $\neg\langle x,y\rangle = \langle y,x\rangle$ as an operator that inverts the $\leq_t$ order but keeps the $\leq_k$ order.

In *FOUR*, the operations $\vee$ and $\wedge$ are De Morgan dual of each other, i.e., $\neg(x \vee y) = \neg x \wedge \neg y$ and $\neg(x \wedge y) = \neg x \vee \neg y$, while $\oplus$ and $\otimes$ are De Morgan self-dual, i.e., $\neg(x \oplus y) = \neg x \oplus \neg y$ and $\neg(x \otimes y) = \neg x \otimes \neg y$. The four values are related with each other by means of $\vee$, $\wedge$, $\oplus$, and $\otimes$, e.g., $\bot \vee \mathbf{f} = \bot$, $\mathbf{t} \oplus \mathbf{f} = \top$, $x \vee \bot = x \otimes \mathbf{t}$.

The bilattice *FOUR* is *distributive*, i.e., the four lattice operations $\wedge$, $\vee$, $\otimes$, and $\oplus$ distribute over each other, e.g., $x \oplus (y \wedge z) = (x \oplus y) \wedge (x \oplus z)$. A distributive bilattice is also *interlaced*, that is, each of the four lattice operations is monotonic with respect to both $\leq_t$ and $\leq_k$, e.g., $y \leq_t z$ implies $x \otimes y \leq_t x \otimes z$.

The bilattice structure can be made into a *logical bilattice* that provides suitable notions of implications in four-valued logic [1]. With $\mathcal{D} = \{\mathbf{t}, \top\}$ being the set of *designated* truth values, which are the values recognized as (at least) known to be true, the bilattice *FOUR* is made into a logical bilattice with two implication connectives, called *weak implication* $\supset$ and *strong implication* $\rightarrow$, which are defined as below:

$$x \supset y \triangleq \begin{cases} \mathbf{t} & (x \notin \mathcal{D}) \\ y & (\text{otherwise}), \end{cases} \qquad x \rightarrow y \triangleq (x \supset y) \wedge (\neg y \supset \neg x).$$

Using strong implication, we define the equivalence $x \leftrightarrow y$ by $(x \rightarrow y) \wedge (y \rightarrow x)$.

[2] In the literature, $\leq_t$ is often regarded as the *degree of truth* and $\leq_k$ as the *amount of information*. Given a product $\langle x,y\rangle$ of classical truth values, x represents the amount of evidence *for* an assertion, while y represents the amount of evidence *against* it. However, one should refrain from sticking to this particular interpretation, when the four-valued logic is used for discriminating the possible termination behaviors in the predicate transformer semantics.

2.2 The Four-Valued Predicate Logic

We give a four-valued first-order predicate logic, based on the Arieli and Avron's four-valued propositional system. (Extension to the predicate logic is straightforward, as mentioned in [1].) We assume the set Value of program values (integers, etc.) and the set Var of program variables. Let us define State to be the set of total functions from Var to Value. Given $\sigma \in \mathsf{State}$ and $X \in \mathsf{Var}$, $\sigma(X)$ denotes the value that is assigned to the program variable X in the state σ.

Four-valued predicates, denoted by p, q, etc., are total functions from State to the four truth values in *FOUR*. The four-valued predicates form a bilattice, where the two partial orders $\leq_t$ and $\leq_k$ and logical connectives $\wedge$, $\vee$, $\otimes$, $\oplus$, $\neg$, $\supset$, $\rightarrow$, $\leftrightarrow$ are accordingly defined in the pointwise way. That is, for every state σ, $p \leq_t q$ (resp. $p \leq_k q$) holds iff $p(\sigma) \leq_t q(\sigma)$ (resp. $p(\sigma) \leq_k q(\sigma)$), and also logical connectives are defined by $(p \vee q)(\sigma) \triangleq p(\sigma) \vee q(\sigma)$, $(\neg p)(\sigma) \triangleq \neg p(\sigma)$, etc. In abuse of notations, we will also denote a constant predicate by the constant itself. That is, we write $\mathbf{t}$ for a predicate p such that $p(\sigma) = \mathbf{t}$ for every state σ; Similarly for $\mathbf{f}$, $\bot$, and $\top$.

It is easy to verify that the bilattice of the four-valued predicates is distributive, interlaced, bounded, and complete. (A bilattice is *complete*, if the two lattices induced by the partial orders $\leq_t$ and $\leq_k$ are both complete.) The completeness indicates that we may also define quantification by means of the infinite join or meet. Given a family of predicates $\{p(i) \mid i \in \mathsf{Value}\}$, we define the universal quantification (resp. existential quantification) over i of predicate $p(i)$ by $\forall i.p(i) \triangleq \bigwedge_i p(i)$ (resp. $\exists i.p(i) \triangleq \bigvee_i p(i)$)

The above mentioned structure of logical bilattice induces a four-valued predicate logic [1], which has a Gentzen-style proof system for sequents of the form $p_1, \cdots, p_n \vdash q_1, \cdots, q_m$ $(n, m \geq 0)$. The sequent corresponds to the consequence relation $p_1, \cdots, p_n \models q_1, \cdots, q_m$, which means, for any state σ, if $p_i(\sigma) \in \mathcal{D}$ for all i, then $q_j(\sigma) \in \mathcal{D}$ for some j. We say a predicate p is *valid* iff $\models p$ holds (i.e., $p(\sigma) \in \mathcal{D}$ for any state σ).

Notice that the four-valued logic is a non-classical logic. In particular it is paraconsistent and does not admit the law of the excluded middle, that is, we have neither $\vdash p \vee \neg p$ nor $p \wedge \neg p \vdash q$. The connectives $\supset$, $\rightarrow$, and $\leftrightarrow$ are a logical implication or an equivalence in the following sense: $\models p \supset q$ iff $p \models q$; $\models p \rightarrow q$ iff $p \leq_t q$; $\models p \leftrightarrow q$ iff $p = q$. Furthermore the logical equivalence $\leftrightarrow$ is a congruence: $\models p \leftrightarrow q$ implies $\models \Theta(p) \leftrightarrow \Theta(q)$ for any formula scheme Θ. For further details of the proof system and logical properties of the four-valued logic, see [1,2].

Throughout the paper, we follow the convention that the negation and quantifications bind most tightly, while implications do least tightly and associate to right. We do not impose any particular precedence between $\vee$, $\wedge$, $\oplus$, and $\otimes$.

Finally, let us introduce some notations that are related to states. A *program expression* e is a total function from State to Value. We write $\sigma[X \backslash v]$ for the state obtained by updating the value assigned to the program variable X in the state σ by the value v. Similarly, we write $\sigma[X \backslash e]$ for an update of variable X with the value of expression e, that is, $\sigma[X \backslash e(\sigma)]$. Given a four-valued predicate p, we also write $p[X \backslash v]$ (resp. $p[X \backslash e]$) for the predicate q such that $q(\sigma) = p(\sigma[X \backslash v])$ (resp. $q(\sigma) = p(\sigma[X \backslash e])$) In particular, a predicate $p[X \backslash v]$ can be recognized as a predicate indexed by v ranging over Value. In abuse of notations, we may often confuse a program variable X with an expression e such that $e(\sigma) = \sigma(X)$. More generally, we may confuse numerical expressions and

predicates with their pointwise extensions. For example, when we write $X+1 \geq Y$, it denotes a predicate q such that $q(\sigma) = \big(\sigma(X)+1 > \sigma(Y)\big)$, where $+$ is the binary integer addition and $\geq$ is the binary predicate such that $(v \geq v') = \mathbf{t}$ if v is greater than or equal to v' but $(v \geq v') = \mathbf{f}$ otherwise.

3 Predicate Transformers and Refinement

3.1 The Lattice of Predicate Transformers

As we have argued earlier, a predicate transformer should be a function that maps a pair of predicates $\langle \varphi_n, \varphi_e \rangle$ to another pair $\langle \varphi'_n, \varphi_e \rangle$. We also require every predicate transformer to be *monotonic*.

Definition 3.1. *A pair of four-valued predicates p and p' is called an* exception matching pair *if $\mathbf{t} \oplus p = \mathbf{t} \oplus p'$ holds.*

A predicate transformer S over four-valued predicates is monotonic *if $S(\varphi) \leq_k S(\varphi')$ holds for every exception matching pair φ and φ' such that $\varphi \leq_k \varphi'$. S is* exception stable *if φ and $S(\varphi)$ are an exception matching pair, for every φ.*

Let PTran be the set of predicate transformers of four-valued predicates that are monotonic and exception stable. Then PTran is made into a bounded complete lattice as follows.

Theorem 3.1. *Let* PTran *be lattice induced by the partial order $\sqsubseteq$ by:*

$$S \sqsubseteq T \quad \textit{iff} \quad S(\varphi) \leq_k T(\varphi) \textit{ for any } \varphi,$$

where the join $\oplus$ and meet $\otimes$ operators are a pointwise extension of the corresponding logical connectives, i.e., $(S \oplus T)(\varphi) = S(\varphi) \oplus T(\varphi)$ and $(S \otimes T)(\varphi) = S(\varphi) \otimes T(\varphi)$. Then PTran *is a bounded complete lattice.*

The class PTran of predicate transformers are also closed under function composition, where we write $S;T$ to mean $(S;T)(\varphi) = S(T(\varphi))$ and intend a sequential execution of S followed by T. The meet $S \otimes T$ and join $S \oplus T$ in PTran, called *demonic choice* and *angelic choice*, respectively, are intended a non-deterministic choice between S and T: The demonic choice represents the least possible non-deterministic execution that the two statements agree, while the angelic choice represents the greatest possible one.

In order to verify that a refinement relation $S \sqsubseteq T$ holds, we need to show $S(\varphi) \leq_k T(\varphi)$ holds for every φ. There are several different ways to verify this.

Proposition 3.1. *For any $S, T \in \mathsf{PTran}$ and any four-valued predicate φ, $S(\varphi) \leq_k T(\varphi)$ iff $S(\varphi) \leq_t T(\varphi)$ iff $\models S(\varphi) \rightarrow T(\varphi)$ iff $S(\varphi) \models T(\varphi)$ iff $S(\varphi) \vdash T(\varphi)$.*

Thus we may verify $S \sqsubseteq T$ by checking the validity of $S(\varphi) \rightarrow T(\varphi)$ in the model of bilattice, which will be effective for the propositional cases. In case quantifiers are involved, we may resort to a formal proof deriving the sequent of the form $S(\varphi) \vdash T(\varphi)$. For further discussions on these alternative ways for validating refinement laws, see the full paper [18].

$$
\begin{array}{rcll}
\mathbf{skip}(\varphi) & \triangleq & \varphi & \text{(skip)} \\
\big(X := e\big)(\varphi) & \triangleq & (\mathbf{f} \oplus \varphi[X \backslash e]) \otimes (\mathbf{t} \oplus \varphi) & \text{(assignment)} \\
\mathbf{abort}(\varphi) & \triangleq & \mathbf{f} \otimes \varphi & \text{(non-termination)} \\
\mathbf{magic}(\varphi) & \triangleq & \mathbf{t} \oplus \varphi & \text{(miracle)} \\
\mathbf{exit}(\varphi) & \triangleq & (\mathbf{t} \oplus \varphi) \otimes \neg(\mathbf{t} \oplus \varphi) & \text{(exit)} \\
\mathbf{try}\ S\ \mathbf{catch}\ T & \triangleq & \big(\mathbf{f} \oplus S\big((\mathbf{f} \oplus \varphi) \otimes \neg(\mathbf{f} \oplus T(\varphi))\big)\big) \otimes (\mathbf{t} \oplus \varphi) & \text{(exception handling)} \\
\big\{p\big\}(\varphi) & \triangleq & \neg(p \supset \top) \otimes \varphi & \text{(assertion)} \\
\big[p\big](\varphi) & \triangleq & (p \supset \bot) \oplus \varphi & \text{(assumption)} \\
\big\langle p\big\rangle(\varphi) & \triangleq & ((p \supset \bot) \oplus \varphi) \otimes \neg((p \supset \top) \oplus \varphi) & \text{(conditional exit)}
\end{array}
$$

Fig. 2. Four-valued predicate transformers for program statements

3.2 Predicate Transformers for Basic Statements

Let us write $\langle \varphi_n, \varphi_e \rangle$ for the pair of predicates that a four-valued predicate φ encodes as we have argued in the introduction. When we define a predicate transformer in PTran, we often need to operate on each component of the pair separately. This can be easily expressed by the four-valued logic formulas. For example, given four-valued predicates p and q, we can express the pair $\langle p_n, q_e \rangle$ by the formula $(\mathbf{f} \oplus p) \otimes (\mathbf{t} \oplus q)$.[3] A simple calculation verifies this as follows:

$$(\mathbf{f} \oplus p) \otimes (\mathbf{t} \oplus q) = (\langle 0,1 \rangle \oplus \langle p_n, p_e \rangle) \otimes (\langle 1,0 \rangle \oplus \langle q_n, q_e \rangle) = \langle p_n, 1 \rangle \otimes \langle 1, q_e \rangle = \langle p_n, q_e \rangle.$$

In a similar way, we can verify that $(\mathbf{t} \oplus p) \otimes \neg(\mathbf{t} \oplus p)$ calculates $\langle p_e, p_e \rangle$ and $(\mathbf{f} \oplus p) \otimes \neg(\mathbf{f} \oplus p)$ does $\langle p_n, p_n \rangle$.

In Figure 2, we give the definitions of four-valued predicate transformers for a set of basic statements. (It is easy to verify that all of them are a member of PTran.)

- **skip** is the idle statement. It is an identity function and hence is a neutral element for the sequential composition, i.e., $\mathbf{skip}; S = S; \mathbf{skip} = S$.
- $X := e$ is the assignment statement. Given a post-condition $\langle \varphi_n, \varphi_e \rangle$, it calculates the weakest pre-condition $\varphi_n[X \backslash e]$ for normal termination and keeps the condition φ_e for exceptional termination unchanged. Note that this assignment is total and deterministic, that is, it always successfully assigns a unique value to the program variable. We will discuss partial assignments in Section 5.2.
- **abort** and **magic** are extremal elements, that is, the least and greatest elements of PTran, respectively. **abort**[4] represents a statement that is not guaranteed to terminate normally. On the other hand, **magic** represents a miraculous statement that always terminates normally, establishing any required post-condition (even falsity).

[3] There are different ways of expressing the same operation, e.g., $(\mathbf{t} \otimes p) \oplus (\mathbf{f} \otimes q)$.

[4] The name 'abort' is historical and is not necessarily adequate in the context of this paper, but we keep using it for compatibility.

They are a left-zero element of sequential composition, that is, **abort**; $S =$ **abort** and **magic**; $S =$ **magic**.
- **exit** is the statement that raises an exception. As we discussed earlier, it is characterized by a function that transforms every post-condition $\langle \varphi_n, \varphi_e \rangle$ into $\langle \varphi_e, \varphi_e \rangle$. Again **exit** is a left-zero element, i.e., **exit**; $S =$ **exit**.
- **try** S **catch** T is the exception handling statement. The statement calculates the weakest post-condition for normal termination given by King and Morgan's *wp* function and combines it with the condition for exceptional termination, using the formulas discussed above.
- $\{p\}$, $[p]$, and $\langle p \rangle$, which are called *assertion*, *assumption*, and *conditional exit*, respectively, are primitive forms of conditional controls, which decide how to continue the execution, depending on the value of the four-valued predicate p, which is called a *guard predicate*. They are all equivalent to **skip**, if the predicate p has a designated truth value (i.e., either **t** or $\top$); otherwise, $\{p\}$, $[p]$, $\langle p \rangle$ are equivalent to **abort**, **magic**, **exit**, respectively.[5]

The basic statements above can be combined to form a more complicated statement. A conditional statement **if** p **then** S **else** T, which may raise an exception when a partial predicate p evaluates to $\bot$, can be defined as follows:

$$\textbf{if } p \textbf{ then } S \textbf{ else } T \triangleq \langle p \vee \neg p \rangle; ((\lceil p \rceil; S) \otimes (\lceil p \supset \bot \rceil; T)).$$

The partiality of predicate p is first tested by the prepended $\langle p \vee \neg p \rangle$, which acts like **exit** if p has the value $\bot$ but like **skip** otherwise. Then, a demonic choice is made between the two branches, each prepended by an assumption statement. (The assumption statement in the unselected branch becomes **magic**, which is dismissed by the outer demonic choice.)

3.3 Logical Connectives for Partial Predicates

In the above definition of conditional statements, we interpret $\top$ as an indication of true on the ground that $\top$ is a designated value in the four-valued logic, but this sometimes leads to a result that runs counter to the operational intuition. (For example, some of the laws given in Section 5.1 do not hold for arbitrary four-valued guard predicates.)

In order to obtain a more precise modelling of partial predicates that adheres to the operational intuition, let us consider *consistent* predicates [7]: A four-valued predicate p is called consistent if $p(\sigma) \in \{\mathbf{t}, \mathbf{f} \bot\}$ for any σ. The class of consistent predicates forms a three-valued sublogic, whose logical operators $\wedge$ and $\vee$, a.k.a. strong Kleene connectives, are non-strict operators that avoid $\bot$ whenever possible. (For instance, both $\mathbf{f} \wedge \bot$ and $\bot \wedge \mathbf{f}$ are interpreted **f** rather than $\bot$.) Non-strictness implies that the strong Kleene connectives cannot be implemented in real programming languages.

[5] Some programming languages provide a feature called 'assertion', which is used for exceptionally terminating the execution when some critical violation of condition is detected. Note the difference from the assertion $\{p\}$, which is non-terminating when the test on p is false. The name 'assertion' is thus somewhat confusing but we keep using it for historical reason.

We can define logical operators that are found in practical programming languages in the three-valued sublogic as follows. Following [7], let us write $p : q$ for $((p \otimes t) \oplus \neg(p \otimes t)) \otimes q$. This derived formula $p : q$ has $\bot$ if p has **f** or $\bot$; otherwise, it has the value of q.

We can define a 'sequential' disjunction $\vee$ and conjunction $\wedge$ for any pair of consistent predicates p and q, as follows.

$$p \wedge q \triangleq p \wedge (p : q) \qquad\qquad p \vee q \triangleq p \vee (\neg p : q)$$

These operators are strict and evaluated sequentially from left to right: it becomes $\bot$ as soon as the left subformula p evaluates to $\bot$.

We can also define the weak Kleene connectives $\vee^w$ and $\wedge^w$ as the consensus of the corresponding two sequential connectives of opposite directions.

$$p \wedge^w q \triangleq (p \wedge q) \otimes (q \wedge p) \qquad\qquad p \vee^w q \triangleq (p \vee q) \otimes (q \vee p)$$

In contrast to the strong Kleene connectives, the value of these connectives is defined only if both of the subformulas are defined.

The strong Kleene connectives $\wedge$ and $\vee$, the sequential connectives $\wedge$ and $\vee$, and also the weak Kleene connectives $\wedge^w$ and $\vee^w$ are all De Morgan dual for each.

4 Refinement Laws for Statements

In the rest of this paper, we assume that guard predicates occurring in control statements are four-valued, unless explicitly stated otherwise. We will indicate wherever a guard predicate is required to be consistent. We further assume that, unless it is explicitly stated otherwise, numerical predicates (which we mentioned in the last paragraph of Section 2.2) are classical, that is, $p(\sigma) \in \{\mathbf{t}, \mathbf{f}\}$ for any σ. The class of classical predicates in the four-valued logic forms a classical sublogic, where the connectives $\vee$, $\wedge$, and $\neg$ substitute for the classical connectives of disjunction, conjunction, and negation, respectively, and implications $\supset$ and $\rightarrow$ substitute for the material implication. We may resort to the standard classical logical reasoning in this sublogic.

Let us first examine some basic refinement laws. From the distributivity of logical connectives, we can derive several distribution laws for demonic choice. The sequencing operator admits the left distribution law, i.e., $(S_1 \otimes S_2); T = (S_1; T) \otimes (S_2; T)$. (The right distribution law does not hold in general, though.) The exception handling statement also admits a distribution law **try** $S_1 \otimes S_2$ **catch** T = (**try** S_1 **catch** T) $\otimes$ (**try** S_2 **catch** T).

By the interlaced property of logical connectives, all the statements introduced in the previous section are monotonic with respect to refinement of its substatements. For instance, $S_1 \otimes T_1 \sqsubseteq S_2 \otimes T_2$ holds if $S_1 \sqsubseteq S_2$ and $T_1 \sqsubseteq T_2$.

4.1 Refinement of Conditional Controls

The statement **skip** and the three conditional control statements are ordered by $\sqsubseteq$ as below.

$$\{p\} \sqsubseteq \textbf{skip} \sqsubseteq [p] \qquad (4.1) \qquad\qquad \{p\} \sqsubseteq \langle p \rangle \sqsubseteq [p] \qquad (4.2)$$

Further, the assertion (resp. the assumption) is monotonic (resp. anti-monotonic) with respect to the $\leq_t$ order over guard predicates. That is, if $p \rightarrow q$ is valid (or equivalently, $p \models q$), we have:

$$\{p\} \sqsubseteq \{q\} \quad (4.3) \qquad [q] \sqsubseteq [p] \quad (4.4)$$

In contrast, the conditional exit has no such particular (anti-)monotonicity property.

Provided that $p \rightarrow q$ is valid, we have:

$$\{p\} = \{p\};\{q\} = \{p\};[q] = \{p\};\langle q\rangle \quad (4.5)$$
$$[p] = [p];\{q\} = [p];[q] = [p];\langle q\rangle \quad (4.6)$$
$$\langle p\rangle = \langle p\rangle;\{q\} = \langle p\rangle;[q] = \langle p\rangle;\langle q\rangle \quad (4.7)$$

The following laws indicate that successive conditional control statements of the same kind can be substituted with a single control statement which combines the guard formulas in the original statements by either $\wedge$, $\wedge$, or $\wedge^w$.

$$\{p\};\{q\} = \{q\};\{p\} = \{p \wedge q\} = \{p \wedge q\} = \{p \wedge^w q\} \quad (4.8)$$
$$[p];[q] = [q];[p] = [p \wedge q] = [p \wedge q] = [p \wedge^w q] \quad (4.9)$$
$$\langle p\rangle;\langle q\rangle = \langle q\rangle;\langle p\rangle = \langle p \wedge q\rangle = \langle p \wedge q\rangle = \langle p \wedge^w q\rangle \quad (4.10)$$

Combining the laws (4.5) through (4.10), we can propagate a copy of a conditional control statement past one or more successive control statements (of possibly different kinds), e.g., $[p];\{q\};\langle r\rangle = [p];\{q\};\langle r\rangle;[p]$.

The disjunction in the guard of an assertion or an assumption can be substituted with an appropriate non-deterministic choice.

$$\{p \vee q\} = \{p\} \oplus \{q\} \quad (4.11) \qquad [p \vee q] = [p] \otimes [q] \quad (4.12)$$

From the fact that exactly one of the formulas p and $p \supset \bot$ can have a designated truth value at once, we obtain the following laws.

$$[p] \otimes [p \supset \bot] = \textbf{skip} \quad (4.13) \qquad \langle p\rangle;\langle p \supset \bot\rangle = \textbf{exit} \quad (4.15)$$
$$[p];[p \supset \bot] = \textbf{magic} \quad (4.14)$$

Recall that we have used $\langle p \vee \neg p\rangle$ for testing partiality of the predicate p in the definition of conditional branch statement in Section 3.2. We will later make use of the following rules in order to exploit the implicit control structure indicated by sequential and weak Kleene connectives occurring in the test predicate.

$$\langle (p \wedge q) \vee \neg(p \wedge q)\rangle = \langle p \vee \neg p\rangle;\langle \neg p \vee q \vee \neg q\rangle \quad (4.16)$$
$$\langle (p \vee q) \vee \neg(p \vee q)\rangle = \langle p \vee \neg p\rangle;\langle p \vee q \vee \neg q\rangle \quad (4.17)$$
$$\langle (p \wedge^w q) \vee \neg(p \wedge^w q)\rangle = \langle (p \vee^w q) \vee \neg(p \vee^w q)\rangle = \langle p \vee \neg p\rangle;\langle q \vee \neg q\rangle \quad (4.18)$$

When the predicate p is classical, the following laws hold.

$$\{p \supset \bot\} = \{\neg p\} \quad (4.19) \qquad [p \supset \bot] = [\neg p] \quad (4.20) \qquad \langle p \vee \neg p \rangle = \mathbf{skip} \quad (4.21)$$

4.2 Refinement of Exceptions

The following refinement laws hold for exception statements.

$$\mathbf{exit}; S = \mathbf{exit} \quad (4.22) \qquad \mathbf{try}\ S; \langle p \rangle\ \mathbf{catch\ skip} = \mathbf{try}\ S\ \mathbf{catch\ skip} \quad (4.23)$$

An interesting subclass of PTran is the one that never raise exceptions. We would say that a transformer S never raises exceptions under any program context, if $\psi_n = \psi'_n$ holds whenever $\langle \psi_n, \varphi_e \rangle = S(\langle \varphi_n, \varphi_e \rangle)$ and $\langle \psi'_n, \varphi'_e \rangle = S(\langle \varphi_n, \varphi'_e \rangle)$. This is formally specified in terms of four-valued logic as follows.

Definition 4.1. *A predicate transformer* $S \in \mathsf{PTran}$ *is called* non-exceptional, *if* $S(\varphi) = S(\varphi')$ *holds whenever* $\mathbf{f} \oplus \varphi = \mathbf{f} \oplus \varphi'$.

It is easy to verify that all the statements introduced in Section 3, except for **exit** and $\langle p \rangle$, are non-exceptional if so are their substatements.

For any non-exceptional statement S, the following laws hold.

$$\mathbf{try}\ S\ \mathbf{catch}\ T = S \quad (4.24) \qquad \mathbf{try}\ S; \mathbf{exit}\ \mathbf{catch}\ T = S; T \quad (4.25)$$

5 Examples of Program Transformation by Stepwise Refinement

We will apply the refinement laws developed in the previous section to transformation of programs that involve exceptions and partial predicates.

5.1 Translating Conjunctions and Disjunctions into Explicit Controls

Programs often contain implicit controls by partial predicates. For example, a single conditional statement **if** $p \wedge q$ **then** S **else** T contains several implicit information for control: The predicate $p \wedge q$ evaluates from left to right; As soon as p evaluates to **f**, the **else** clause is selected; exception is raised as soon as p evaluates to $\bot$; q is examined only if p evaluates to **t**.

We justify this operational intuition via refinement by showing that the above conditional statement is equivalent to the nested conditional statement **if** p **then** (**if** q **then** S **else** T) **else** T. Let us first give a few subsidiary refinement laws.

$$[p \wedge q \supset \bot] = [p \wedge q \supset \bot] = [p \supset \bot] \otimes [q \supset \bot]. \quad (5.1)$$

$$[p]; \langle \neg p \vee q \rangle = [p]; \langle q \rangle \quad \text{if } p \text{ is consistent} \quad (5.2)$$

$$\langle p \vee \neg p \rangle; [p \supset \bot] = \langle p \vee \neg p \rangle; [p \supset \bot]; \langle \neg p \vee q \vee \neg q \rangle \quad \text{if } p \text{ is consistent} \quad (5.3)$$

Then we can carry out the following derivation, provided p is consistent.

$$
\begin{aligned}
&\textbf{if } p\wedge q \textbf{ then } S \textbf{ else } T = \langle (p\wedge q)\vee\neg(p\wedge q)\rangle;\big([p\wedge q];S\otimes[p\wedge q\supset\bot];T\big)\\
&= \langle p\vee\neg p\rangle;\langle\neg p\vee q\vee\neg q\rangle;\big([p];[q];S\otimes[p\supset\bot];T\otimes[q\supset\bot];T\big)\\
&\qquad\qquad \text{— by (4.16), (4.9), (5.1), and distributivity}\\
&= \langle p\vee\neg p\rangle;[p];\langle\neg p\vee q\vee\neg q\rangle;\big([p];[q];S\otimes[p\supset\bot];T\otimes[q\supset\bot];T\big)\\
&\quad\otimes\langle p\vee\neg p\rangle;[p\supset\bot];\langle\neg p\vee q\vee\neg q\rangle;\big([p];[q];S\otimes[p\supset\bot];T\otimes[q\supset\bot];T\big)\\
&\qquad\qquad \text{— by (4.13) and distributivity}\\
&= \langle p\vee\neg p\rangle;\Big([p];\langle q\vee\neg q\rangle;\big([q];S\otimes[q\supset\bot];T\big)\otimes[p\supset\bot];T\otimes[p\supset\bot];[q\supset\bot];T\Big)\\
&\qquad\qquad \text{— by (5.2), (5.3), (4.6), (4.7), (4.9), (4.10), (4.14), and distributivity}\\
&= \langle p\vee\neg p\rangle;\Big([p];\langle q\vee\neg q\rangle;\big([q];S\otimes[q\supset\bot];T\big)\otimes[p\supset\bot];T\Big)\quad \text{— by (4.9), (4.4)}\\
&= \textbf{if } p \textbf{ then } (\textbf{if } q \textbf{ then } S \textbf{ else } T) \textbf{ else } T.
\end{aligned}
$$

We can also derive a law for the sequential disjunction:

$$\textbf{if } p\vee q \textbf{ then } S \textbf{ else } T = \textbf{if } p \textbf{ then } S \textbf{ else } (\textbf{if } q \textbf{ then } S \textbf{ else } T),$$

where p is consistent. For the weak Kleene connectives, we have similar laws:

$$\textbf{if } p\wedge^{w} q \textbf{ then } S \textbf{ else } T = \textbf{if } p \textbf{ then } (\textbf{if } q \textbf{ then } S \textbf{ else } T) \textbf{ else } \langle q\vee\neg q\rangle;T \quad\text{and}$$
$$\textbf{if } p\vee^{w} q \textbf{ then } S \textbf{ else } T = \textbf{if } p \textbf{ then } \langle q\vee\neg q\rangle;S \textbf{ else } (\textbf{if } q \textbf{ then } S \textbf{ else } T),$$

where p need not be consistent.

5.2 Refining Exception Handling

Let us apply our refinement laws to a larger program. In the development, we will make use of the technique that propagates context information via the assertion statement [15, 9]. Below we list several non-trivial laws for propagating context information.

$$\{p\};X := e \sqsubseteq X := e;\{\exists v.(p[X\backslash v]\wedge X = e[X\backslash v])\} \tag{5.4}$$
$$\{p\};[q] \sqsubseteq [q];\{p\wedge q\} \tag{5.5}$$
$$\{p\};\langle q\rangle \sqsubseteq \langle q\rangle;\{p\wedge q\} \tag{5.6}$$
$$\{p\};\textbf{if } q \textbf{ then } S \textbf{ else } T \sqsubseteq \textbf{if } q \textbf{ then } (\{p\wedge q\};S) \textbf{ else } (\{p\wedge(q\supset\bot)\};T) \tag{5.7}$$
$$\textbf{if } q \textbf{ then } (S;\{p\}) \textbf{ else } (T;\{q\}) \sqsubseteq (\textbf{if } q \textbf{ then } S \textbf{ else } T);\{p\vee q\} \tag{5.8}$$
$$\{p\};\textbf{try } S \textbf{ catch } T \sqsubseteq \textbf{try } \{p\};S \textbf{ catch } T \tag{5.9}$$
$$\textbf{try } S;\{p\} \textbf{ catch } (T;\{q\}) \sqsubseteq (\textbf{try } S \textbf{ catch } T);\{p\vee q\}; \tag{5.10}$$

Let us consider the following program S_0 that implements a numerical algorithm.

$$S_0 \triangleq X := N;\textbf{try repeat } Y := X;X := (Y\times Y+N)\div(2\times Y) \textbf{ until } X\geq Y \textbf{ catch skip}.$$

This program computes the integral value of $\sqrt{N}$ for non-negative integer N, based on the Newton-Raphson method [20], and assigns the answer to the variable Y. In the **repeat** $\cdots$ **until** loop, the integer division operator $\div$ may raise an exception due to division-by-zero, in which case, however, the exception is caught and the execution normally terminates with a correct answer.

Since PTran is a bounded complete lattice, each loop statement is specified by the least fixpoint $\mu.\mathcal{F}$ of a function $\mathcal{F} \in \mathsf{PTran} \to \mathsf{PTran}$ that is monotonic w.r.t. refinement order $\sqsubseteq$ [3]. The loop statement in S_0 is given by the least fixpoint of the function:

$$\mathcal{F}(T) \triangleq Y := X; X := (Y \times Y + N) \div (2 \times Y); \textbf{if } X \geq Y \textbf{ then skip else } T.$$

In order to express the partial assignment $X := (Y \times Y + N) \div (2 \times Y)$, which may raise exception due to division-by-zero, we interpret it by the compound statement $\langle \neg(Y = 0) \rangle; X := (Y \times Y + N) \div' (2 \times Y)$, where $\div'$ is a total extension of $\div$ such that division by zero yields a fixed constant value (say, 0) instead of being undefined.

In the following derivation, we refine the original program S_0, with the assumption $N \geq 0$, into a program that makes no uses of exceptional statements.

$$\begin{aligned}
\{N \geq 0\}; S_0 &\sqsubseteq X := N; \{X = N \wedge N \geq 0\}; \textbf{try } \mu.\mathcal{F} \textbf{ catch skip} && \text{— by (5.4)} \\
&\sqsubseteq X := N; \big((\textbf{try } [\neg(X = 0)]; \{0 < X \leq N\}; \mu.\mathcal{F} \textbf{ catch skip}) \otimes \\
&\qquad (\textbf{try } [\neg(X = 0) \supset \bot]; \{\neg(X = 0) \supset \bot\}; \mu.\mathcal{F} \textbf{ catch skip})\big) \\
&\qquad \text{— by (5.9), (4.13), (4.3), (4.1), (4.6), and distributivity.}
\end{aligned}$$

In order to show the refinement of the left substatement of the demonic choice, we need some lemmas.

Lemma 5.1.

$$\begin{aligned}
&\{0 < X \leq N\}; \textbf{repeat } Y := X; X := (Y \times Y + N) \div (2 \times Y) \textbf{ until } X \geq Y \\
\sqsubseteq\ &\textbf{repeat } \{0 < X \leq N\}; Y := X; X := (Y \times Y + N) \div' (2 \times Y) \textbf{ until } X \geq Y
\end{aligned}$$

Lemma 5.2. *Suppose* $\mathcal{F} \in \mathsf{PTran} \to \mathsf{PTran}$ *is a monotonic function. Then,* $\mu.\mathcal{F}$ *is non-exceptional, if* $\mathcal{F}(S)$ *is so for every non-exceptional* S.

Lemma 5.1 indicates that $0 < X \leq N$ is a loop invariant and lemma 5.2 says that the fixpoint operator on PTran preserves non-exceptionality. Proofs of these lemmas can be found in the full paper [18].

$$\begin{aligned}
&\textbf{try } [\neg(X = 0)]; \{0 < X \leq N\}; \mu.\mathcal{F} \textbf{ catch skip} \\
\sqsubseteq\ &\textbf{try } [\neg(X = 0)]; \textbf{repeat } \{0 < X \leq N\}; Y := X; X := (Y \times Y + N) \div' (2 \times Y) \\
&\qquad \textbf{until } X \geq Y \textbf{ catch skip} && \text{— by lemma 5.1} \\
=\ &[\neg(X = 0)]; \textbf{repeat } \{0 < X \leq N\}; Y := X; X := (Y \times Y + N) \div' (2 \times Y) \\
&\qquad \textbf{until } X \geq Y \textbf{ catch skip} && \text{— by (4.24) and non-exceptionality from lemma 5.2} \\
=\ &[\neg(X = 0)]; \textbf{repeat } Y := X; \{\neg(Y = 0)\}; \langle \neg(Y = 0) \rangle; X := (Y \times Y + N) \div' (2 \times Y) \\
&\qquad \textbf{until } X \geq Y && \text{— by (5.4), (4.3), and (4.5)} \\
=\ &[\neg(X = 0)]; \textbf{repeat } Y := X; X := (Y \times Y + N) \div (2 \times Y) \textbf{ until } X \geq Y && \text{— by (4.1)}
\end{aligned}$$

For the other substatement of the choice, we derive:

$$
\begin{array}{ll}
 & \textbf{try } \lceil \neg(X=0) \supset \bot \rceil ; \{\neg(X=0) \supset \bot\}; \mu.\mathcal{F} \textbf{ catch skip} \\
\sqsubseteq & \textbf{try } \big(\lceil \neg(X=0) \supset \bot \rceil ; Y := X; \{\neg(Y=0) \supset \bot\}; \langle \neg(Y=0) \rangle; \\
 & \qquad X := (Y \times Y + N) \div' (2 \times Y); \textbf{if } X \geq Y \textbf{ then skip else } \mu.\mathcal{F}\big) \textbf{ catch skip} \\
 & \qquad\qquad \text{— fixpoint; by (5.4)} \\
= & \textbf{try } \lceil \neg(X=0) \supset \bot \rceil ; Y := X; \{\neg(Y=0) \supset \bot\}; \textbf{exit catch skip} \\
 & \qquad\qquad \text{— by (4.6), (4.10), and (4.15)} \\
\sqsubseteq & \lceil \neg(X=0) \supset \bot \rceil ; Y := X \qquad\qquad \text{— by (4.25) and (4.1).}
\end{array}
$$

Therefore the derivation ends up with:

$$
\begin{array}{ll}
S_0 \sqsubseteq & X := N; \langle \neg(X=0) \vee \neg\neg(X=0) \rangle; (\lceil \neg(X=0) \rceil ; \mu.\mathcal{F} \otimes \lceil \neg(X=0) \supset \bot \rceil ; Y := X) \\
 & \qquad\qquad \text{— by (4.21)} \\
= & X := N; \textbf{if } \neg(X=0) \textbf{ then repeat } Y := X; X := (Y \times Y + N) \div (2 \times Y) \textbf{ until } X \geq Y \\
 & \qquad\qquad \textbf{else } Y := X.
\end{array}
$$

6 Conclusion and Future Work

We proposed a refinement calculus for refining exceptions in programs. In order to model the normal termination as well as the exceptional termination in a single unified platform, we developed a four-valued predicate transformer semantics, which is based on Arieli and Avron's four-valued logic [1]. The programming constructs for raising and catching exceptions can be concisely expressed by the formulas of four-valued logic in this framework. In particular, we allow partial predicates in the conditional control statements in order to model exceptions that are raised implicitly when the predicate in a conditional statement is undefined. The four-valued logic provides a fruitful field for justifying refinement of programs that involve both explicit and implicit controls by exceptions.

This paper, with a few deviations, dealt with concrete program statements such as assignment, (conditional) exit, etc. Future research will concern abstract statements such as non-deterministic (possibly partial) assignment and general specification statement (which allows the uses of partial pre- and post-conditions) and also the methodology for deriving concrete programs from these abstract statements.

Acknowledgment. I thank anonymous reviewers for their suggestions and comments. This work was supported by JSPS KAKENHI(20500011).

References

1. Arieli, O., Avron, A.: Reasoning with logical bilattices. Journal of Logic, Language, and Information 5(1), 25–63 (1996)
2. Arieli, O., Avron, A.: The value of four values. Artificial Intelligence 102(1), 97–141 (1998)
3. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer, Heidelberg (1998)

4. Belnap, N.D.: A useful four-valued logic. In: Epstein, G., Dunn, J.M. (eds.) Modern Uses of Multiple-Valued Logic, pp. 7–37. Reidel Publishing Company, Dordrechtz (1977)
5. Bono, V., Kerber, M.: Extending Hoare calculus to deal with crash. Technical Report CSR-06-08, School of Computer Science, The University of Birmingham (2006)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
7. Fitting, M.: Kleene's three-valued logics and their children. Fundamenta Informaticae 20(1/2/3), 113–131 (1994)
8. Ginsberg, M.L.: Multivalued logics: A uniform approach to reasoning in artificial intelligence. Computational Intelligence 4, 265–316 (1988)
9. Groves, L.J.: Evolutionary Software Development in the Refinement Calculus. PhD thesis, Victoria University of Wellington (2000)
10. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Logic Journal of the IGPL 13(4), 415–433 (2005)
11. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 284–303. Springer, Heidelberg (2000)
12. Jones, C.B.: Systematic Software Development Using VDM. International Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1986)
13. Jones, C.B., Middelburg, C.A.: A typed logic of partial functions reconstructed classically. Acta Informatica 31(5), 399–430 (1994)
14. King, S., Morgan, C.: Exits in the refinement calculus. Formal Aspects of Computing 7(1), 54–76 (1995)
15. Laibinis, L., von Wright, J.: Context handling in the refinement calculus framework. Technical Report 118, TUCS Technical Report (1997)
16. Morgan, C.: Programming from specifications. 2nd edn. Prentice-Hall International Series in Computer Science. Prentice-Hall International (1994)
17. Morris, J.M., Bunkenburg, A.: E3: A logic for reasoning equationally in the presence of partiality. Science of Computer Programming 34(2), 141–158 (1999)
18. Nishimura, S.: Refining exceptions in four-valued logic, http://www.math.kyoto-u.ac.jp/ susumu/papers/lopstr09-full.pdf
19. Owe, O.: Partial logics reconsidered: A conservative approach. Formal Aspects of Computing 5(3), 208–223 (1993)
20. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing, 3rd edn. Cambridge University Press, Cambridge (2007)
21. Watson, G.: Refining exceptions using King and Morgan's exit construct. In: 9th Asia-Pacific Software Engineering Conference (APSEC 2002), pp. 43–51. IEEE Computer Society, Los Alamitos (2002)

Towards a Framework for Constraint-Based Test Case Generation

François Degrave[1,⋆], Tom Schrijvers[2,⋆⋆], and Wim Vanhoof[1]

[1] Faculty of Computer Science,
University of Namur
[2] Department of Computer Science,
Katholieke Universiteit Leuven

Abstract. In this paper, we propose an approach for automated test case generation based on techniques from constraint programming (CP). We advocate the use of standard CP search strategies in order to express preferences on the generated test cases and to obtain the desired degree of coverage. We develop our framework in the concrete context of an imperative language and show that the technique is sufficiently powerful to deal with arbitrary pointer-based data-structures allocated on the heap.

1 Introduction

It is a well-known fact that a substantial part of a software development budget is spent on the act of correcting errors in the software under development. Arguably the most commonly applied strategy for finding errors and thus producing (more) reliable software is testing: running a software component with respect to a well-chosen set of inputs and comparing the outputs that are produced with the expected results in order to find errors. One approach, so called *whitebox* or *structural* testing consists in selecting a set of test inputs that together *cover* a substantially large part of the program's source code, according to some adequacy or coverage criterion. According to [22], an adequacy criterion is considered to be a stopping rule that determines whether sufficient testing has been done. In practice, the most used criteria are different *coverage* criteria, such as *statement*, *branch* or *path coverage* criteria.

In the current paper, we present a technique for automatically generating test inputs for programs written in an imperative language dealing with pointer-based data structures. This is especially challenging, as a test input for a procedure comprises not only a set of atomic values for the procedure's arguments but may also contain data structures build on the heap. The use of Constraint Programming (CP) and its inherent mechanisms facilitate dealing with of a number of important issues. First, representing the heap and environment of the program

⋆ Supported by a grant FRIA - Belgium.
⋆⋆ Post-doctoral researcher of the Fund for Scientific Research - Flanders.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 128–142, 2010.

by means of a symbolic data structure provides a convenient way to describe constraints on those structures. More importantly, we can use the search strategies of CP in order to tackle two essential issues: the first one comprises collecting a finite set of execution paths of the program which satisfies some given adequacy criteria. The second one is the generation, for each such path, of concrete values (a test input) such that when the program is executed with respect to those values, its execution will follow the corresponding path. Therefore, our technique can be seen as parametrised with respect to a coverage criterion or a desired degree of coverage. In order to illustrate the usefulness of this property, let us take an example of a small procedure written in a C-like programming language, supporting pointer-based dynamic data structures. This procedure manipulates a pointer `queu` to a linked list – whose structure is examinated in further details afterwards –, an element `el` of type T (this type has no importance in this example), and two integers `prioD` and `n`.

```
void insert (queu,el,prioD,n) {
        ptr = *queu.next ;
        q = queu ;
        c = 1
        while(ptr.prio >= prioD && c<n){
                ptr = *ptr.next ; c++ }
        r = new(el,max(prioD,*ptr.prio),ptr)
        q.next = r}
```

This procedure basically inserts an element `el` into a priority queue `queu` – represented as a linked list – with respect to a given priority `prioD`. The element is inserted just after the last element having a higher priority than `prioD` if the number of such elements is less than `n`; otherwise, the element is inserted at the $\mathtt{n}^{th}$ position, and its priority is changed to that of the $\mathtt{n}-\mathtt{1}^{th}$ element of the queue.

A test case for a procedure consists of an environment and a heap as they could be at the moment of the procedure's call. For example, a test case for the `insert` procedure could be an environment in which the variables `prioD` and `n` both map to the value 3, `el` maps to an arbitrary value depending on its type, and `queu` maps to a reference, pointing into the heap to the first cell of a linked list, whose cells consist of three fields: 1) a content (whose type and value have no importance in the current example, and is represented as a small shape in Figure 1()), 2) a priority (an integer value) and 3) a reference to the next cell in the list. Two examples of such test cases are depicted in Figure 1()).

One major advantage of our framework is that it is parametrised with respect to a given coverage criterion. Different coverage criteria can lead, of course, to different generated test cases. For example, if statement coverage is used as the coverage criterion, our technique might produce the test case (a) depicted in Figure 1 as the only testcase. Indeed, execution of the procedure with respect

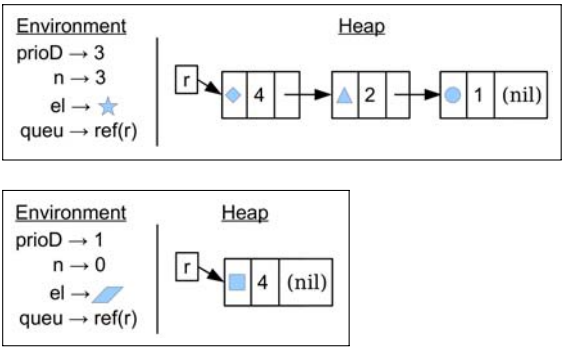


Fig. 1. Examples of test cases for the `insert` procedure

to this data will guarantee that every statement in the procedure's body gets executed. However, if condition coverage is used as the coverage criterion, the test case (a) in itself is not sufficient as the test suite must guarantee that every boolean sub-expression is evaluated both to true and false during testing while with the test case (a), only the subexpression `ptr.prio >= prioD` is evaluated both to true and false. Therefore, instantiated with condition coverage the technique will produce at least one additional testcase, for example the one depicted in Figure 1 (b) in which the subexpression `c<n` is guaranteed to be eventually evaluated to false.

Our specific contributions are:

- We show how to extend the semantics of an imperative language to deal with unknown pointer-based input values. (Section 2.2)
- We show how concrete test cases satisfying adequacy criteria can be generated by using a suitable CP search strategy. (Section 2.5)
- We present a visualization tool and a regression test generator based on our approach. (Section 3)

In order to focus on the essence of constraint-based test generation for imperative languages, we define a small imperative language supporting dynamic pointer-based data structures and show that our approach is able to generate test cases dealing with in-place updates of variables, pointers and a variety of potentially cyclic data structures – for convenience, we refer to this language as ImpL in what follows. As the definition below shows, we only consider integer values and data structures constructed from simple "cons" cells having two fields that we will name *head* and *tail*. We indicate in Section 2.6 how our technique for test case generation can easily be extended to deal with a more involved language having primitive values other than integers and full `struct`-like data structures.

integers	n	
variables	x	
expressions	e ::=	$x \mid n \mid$ `nil` $\mid$ `new cons`$(e_1, e_2) \mid e$`.head` $\mid e$`.tail`
		$\mid e_1$ `==` $e_2 \mid e_1$ `/=` $e_2 \mid e_1$`+`e_2
statements	s ::=	`skip` $\mid l$ `:=` $e \mid s_1$`;`$s_2 \mid$ `if` e `then` s_1 `else` s_2
		$\mid$ `while` e `{` s `}`
left-hand sides	l ::=	$x \mid l$`.head` $\mid l$`.tail`

As usual, expressions are used to syntactically represent values within the source code of a program. Among the possible expressions are program variables, integers, the null-pointer `nil`, a reference to a newly heap-allocated cons cell `new cons(`e_1`,`e_2`)`, the selection of the head ($e.head$), respectively tail ($e.tail$) field of the cons cell referenced by e, equality and inequality tests (`==` and `/=`), and the arithmetic operator for addition `+`.[1] We will assume that ImpL is simply typed and it only allows comparison of two values belonging to the same type (either integers or references).[2] Moreover, arithmetic is only allowed on integer values; the language does not support pointer arithmetics.

A program in ImpL is a single statement or a sequence of statements, where a statement is either a no-op (`skip`), an assignment, another sequence, a selection or a while-loop. The left-hand side of an assignment is either a variable or a reference to one of the fields in a cons cell. Consider, for example, the following simple program:

```
while (x.tail.head /= x.head) {
        x := x.tail };
x.tail := nil
```

The above program basically manipulates a simply linked list `x` whose cells consist of two fields: a *head* containing an integer and a *tail* containing a pointer to the following cell or `nil`. It scans the list for two successive identical elements, and severs the list after the first such occurrence. For example, using the notation `[1,2,3]` for the nil-terminated linked list with successive elements 1,2 and 3, the effect of running this program with `x` the list `[1,2,3,3,4]`, is that, after the statement `x.tail := nil`, the list will have the value `[1,2,3]`.

2 Generating Test Inputs

2.1 Overview

The execution of an imperative program manipulates an environment E and a heap H. An environment is a finite mapping from variables to *values*, where a value is either an integer, `nil` or a reference to a cons cell represented by `ptr`(r) with r a unique value denoting the address of the cons cell on the heap.

[1] Other arithmetic operators are omitted in order to keep the formal definition of the semantics small, but they can be added at will.

[2] Integers are also used as booleans: 0 denotes false and all other integers denote true.

Likewise, a heap is a finite mapping from such references r to cons cells of the form `cons(`v_h`,`v_e`)` with v_h and v_e values (possibly including references to other cons cells). For the example given above (with `x` initially the list `[1,2,3,3,4]`), the environment and heap before and after running the program would look as follows:

$E : x \mapsto$ `ptr(`r_1`)`	$E : x \mapsto$ `ptr(`r_3`)`
$H : r_1 \mapsto$ `cons(1,ptr(`r_2`))` $r_4 \mapsto$ `cons(3,ptr(`r_5`))`	$H : r_1 \mapsto$ `cons(1,ptr(`r_2`))`
$r_2 \mapsto$ `cons(2,ptr(`r_3`))` $r_5 \mapsto$ `cons(4,nil)`	$r_2 \mapsto$ `cons(2,ptr(`r_3`))`
$r_3 \mapsto$ `cons(3,ptr(`r_4`))`	$r_3 \mapsto$ `cons(3,nil)`

Now, in order to generate test inputs for a program, the idea is to *symbolically* execute the program, replacing unknown values by *constraint variables*. During such a symbolic execution, each test in the program (i.e. the *if-then-else* and *while* conditions) represents a choice; the sequence of choices made determines the execution path followed. There are many possible execution paths through the program. Each one of them can be represented by constraints on the introduced variables and on the environment and heap.

Returning to our example, we would replace the concrete value for `x` by a constraint variable, say `V`, representing an unknown value. Among the infinite number of possible execution paths, a particular path would execute the *while* condition three times, and the loop body twice. This would imply that the value represented by `V` is a list of at least 4 elements, and the third and fourth element are identical, whereas the first differs from the second and the second from the third. This information would be represented by constraints on `V` and the heap collected along the execution. Solving these constraints could get us for instance the concrete input `[1,2,3,3,4]` proposed above. However, there are many other concrete inputs that satisfy these constraints: `[1,2,3,3]`, `[0,1,0,0]`, or even the cyclic list that starts with `[1,2,1]` and then points back the first element.

Using our constraint-based approach, we can both capture the many paths and the many solutions for a single path as non-determinism in our constraint-based modelling of test case generation. This allows us to use the search strategies of CP to deal with *both* of them. For instance, we can find all paths up to length 6 using a simple depth-bounded search.

2.2 Constraint Generation

In order to represent unknown input data we add logical (or constraint) variables to the semantic domain of values and represent the environment and heap by logical variables as well. In order to model symbolic execution of our language, we introduce a semantics in which program state is represented by a triple $\langle E, H, C\rangle$ where E and H are constraint variables symbolically representing, respectively, the environment and heap, and C is a set of constraints over E and H. Constraints are conjunctions of primitive constraints that take the following form:

$$\text{(Var)}\ \frac{(x \mapsto v) \in E}{\langle E, H\rangle\ x \rightsquigarrow v \langle E, H\rangle}$$

$$\text{(Int)}\ \frac{n \in \mathbb{Z}}{\langle E, H\rangle\ n \rightsquigarrow n \langle E, H\rangle}$$

$$\text{(Nil)}\ \langle E, H\rangle\ \texttt{nil} \rightsquigarrow \texttt{nil} \langle E, H\rangle$$

$$\text{(Cons)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2 \langle E, H_3\rangle \qquad r \text{ fresh}}{\langle E, H_1\rangle\ \texttt{new cons}(e_1, e_2) \rightsquigarrow \texttt{ptr}(r) \langle E, H_3 \uplus \{r \mapsto \texttt{cons}(v_1, v_2)\}\rangle}$$

$$\text{(Head)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow \texttt{ptr}(r) \langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ e.\texttt{head} \rightsquigarrow v_h \langle E, H_2\rangle}$$

$$\text{(Tail)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow \texttt{ptr}(r) \langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ e.\texttt{tail} \rightsquigarrow v_t \langle E, H_2\rangle}$$

$$\text{(EqualT)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2 \langle E, H_3\rangle \qquad v_1 \equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 1 \langle E, H_3\rangle}$$

$$\text{(EqualF)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2 \langle E, H_3\rangle \qquad v_1 \not\equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 0 \langle E, H_3\rangle}$$

$$\text{(NEqualT)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2 \langle E, H_3\rangle \qquad v_1 \not\equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 1 \langle E, H_3\rangle}$$

$$\text{(NEqualF)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2 \langle E, H_3\rangle \qquad v_1 \equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 0 \langle E, H_3\rangle}$$

Fig. 2. Semantics of expressions in ImpL

$$\text{(Skip)}\ \langle E, H\rangle\ \texttt{skip}\ \langle E, H\rangle$$

$$\text{(VarAss)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v \langle E, H_2\rangle}{\langle E, H_1\rangle\ \}x\ \texttt{:=}\ e\ \langle E \uplus \{x \mapsto v\}, H_2\rangle}$$

$$\text{(HeadAss)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ l \rightsquigarrow \texttt{ptr}(r) \langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ l.\texttt{head}\ \texttt{:=}\ e\ \langle E, H_2 \uplus \{r \mapsto \texttt{cons}(v, v_t)\}\rangle}$$

$$\text{(TailAss)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v \langle E, H_2\rangle \qquad \langle E, H_2\rangle\ l \rightsquigarrow \texttt{ptr}(r) \langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ l.\texttt{tail}\ \texttt{:=}\ e\ \langle E, H_2 \uplus \{r \mapsto \texttt{cons}(v_h, v)\}\rangle}$$

$$\text{(Seq)}\ \frac{\langle E_1, H_1\rangle\ s_1\ \langle E_2, H_2\rangle \qquad \langle E_2, H_2\rangle\ s_2\ \langle E_3, H_3\rangle}{\langle E_1, H_1\rangle\ s_1 ; s_2\ \langle E_3, H_3\rangle}$$

$$\text{(IfThen)}\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n \langle E_1, H_2\rangle \qquad n \not\equiv 0 \qquad \langle E_1, H_2\rangle\ s_1\ \langle E_2, H_3\rangle}{\langle E_1, H_1\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_2, H_4\rangle}$$

$$\text{(IfElse)}\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n \langle E_1, H_2\rangle \qquad n \equiv 0 \qquad \langle E_1, H_2\rangle\ s_2\ \langle E_2, H_3\rangle}{\langle E_1, H_1\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_2, H_4\rangle}$$

$$\text{(WhileT)}\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n \langle E_1, H_2\rangle \qquad n \not\equiv 0 \qquad \langle E_1, H_2\rangle\ s\ \langle E_2, H_3\rangle \qquad \langle E_2, H_3\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_3, H_4\rangle}{\langle E_1, H_1\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_3, H_4\rangle}$$

$$\text{(WhileF)}\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n \langle E_1, H_2\rangle \qquad n \equiv 0}{\langle E_1, H_1\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2\rangle}$$

Fig. 3. Semantics of statements in ImpL

- $o_1 = o_2$, equality of two syntactic objects,
- $o_1 \neq o_2$, inequality of two syntactic objects,
- $(o_1 \mapsto o_2) \in M$, membership of a mapping M, and
- $M_1 \uplus \{o_1 \mapsto o_2\} = M_2$, update of a mapping M_1.

where a mapping M denotes a constraint variable representing an environment or a heap. Constraint solvers for these constraints are defined in Section 2.4.

The symbolic semantics is depicted in Figures 4 and 5. In these figures and in the remainder of the text, we use uppercase characters to syntactically distinguish constraint variables from ordinary program variables (represented by lowercase characters). A judgement of the form $\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle$ denotes that given a program state $\langle E_0, H_0, C_0\rangle$, the expression e evaluates to value v and transforms the program state into a state represented by $\langle E_0, H_1, C_1\rangle$. Note that H_1 is a *fresh* constraint variable that represents the possibly modified heap whose content is defined by the constraints in C_1. Likewise, a judgement of the form $\langle E_0, H_0, C_0\rangle s\langle E_1, H_1, C_1\rangle$ denotes the fact that a statement s transforms a program state represented by $\langle E_0, H_0, C_0\rangle$ into the one represented by $\langle E_1, H_1, C_1\rangle$. Since a newly added constraint can introduce inconsistencies in the set of collected constraints, we define the *conditional evaluation* of an expression and a statement as follows: judgements of the form $\{E, H_0, C_0\}\ e \rightsquigarrow v\langle E, H_1, C_1\rangle$ and $\{E_0, H_0, C_0\}s\langle E_1, H_1, C_1\rangle$ denote, respectively, $\langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle$ and $\langle E_0, H_0, C_0\rangle s\langle E_1, H_1, C_1\rangle$ under the condition that C_0 is *consistent* (represented by $\mathcal{T} \models C_0$, where $\mathcal{T}$ is the constraint theory).[3] Formally:

$$\text{(Cond-e)}\ \frac{\mathcal{T} \models C_0 \qquad \langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle}{\{E, H_0, C_0\}\ e \rightsquigarrow\ v\langle E, H_1, C_1\rangle}$$

$$\text{(Cond-s)}\ \frac{\mathcal{T} \models C_0 \qquad \langle E_0, H_0, C_0\rangle\ s\ \langle E_1, H_1, C_1\rangle}{\{E_0, H_0, C_0\}\ s\ \langle E_1, H_1, C_1\rangle}$$

The use of conditional evaluation avoids adding further constraints to an already inconsistent set. This implies that search strategies (see Section 2.5) will only explore execution paths that can model a real execution.

2.3 Properties

Given environments E, E' and heaps H, H', we use $\langle E, H\rangle \cong \langle E', H'\rangle$ to denote the fact that E and E' define the same program variables and that each such variable either has the same primitive value (integer or `nil`) in both environments or points to identical data structures in both heaps. More formally, this means that there must exist a bijective mapping σ between (a subset of) the references used in H and (a subset of) those used in H' such that $\forall x \in dom(E) = dom(E') : E(x) =_\sigma E'(x)$ where $=_\sigma$ is defined as follows: $\texttt{nil}=_\sigma\texttt{nil}$, $n =_\sigma n$ for all integer constants n and $ptr(r) =_\sigma ptr(r')$ if $r' = \sigma(r)$, $H(r) = cons(v_1, v_2)$ and $H'(r') = cons(v_1', v_2')$ and $v_1 =_\sigma v_1'$ and $v_2 =_\sigma v_2'$.

Theorem 1 (Completeness)
Let E and H be an environment and a heap, and s a statement manipulating the variables in E. If $\langle E, H\rangle\ s\ \langle E', H'\rangle$ then there exists a satisfiable set of constraints C such that $\langle E_v, H_v, true\rangle\ s\ \langle E_v', H_v', C\rangle$ with ρ a solution for C such that

$$\langle E, H\rangle \cong \langle \rho(E_v), \rho(H_v)\rangle$$
$$\langle E', H'\rangle \cong \langle \rho(E_v'), \rho(H_v')\rangle$$

[3] In practice, the consistency check may be incomplete. Then unreachable execution paths may be explored.

$$\text{(Var)}\ \frac{V \text{ fresh}}{\langle E, H, C\rangle\ x \rightsquigarrow V \langle E, H, C \wedge \{x \mapsto V\} \in E\rangle}$$

$$\text{(Int)}\ \frac{n \in \mathbb{Z}}{\langle E, H, C\rangle\ n \rightsquigarrow n \langle E, H, C\rangle} \qquad \text{(Nil)}\ \langle E, H, C\rangle\ \texttt{nil} \rightsquigarrow \texttt{nil}\langle E, H, C\rangle$$

$$\text{(Cons)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle \qquad H_3, r \text{ fresh}}{\langle E, H_0, C_0\rangle\ \texttt{new cons}(e_1, e_2) \rightsquigarrow \texttt{ptr}(r) \langle E, H_3, C_2 \wedge H_3 = H_2 \uplus \{r \mapsto \texttt{cons}(v_1, v_2)\}\rangle}$$

$$\text{(Head)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v \langle E, H_1, C_1\rangle \qquad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0\rangle\ e.\texttt{head} \rightsquigarrow V_h \langle E, H_1, C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1\rangle}$$

$$\text{(Tail)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v \langle E, H_1, C_1\rangle \qquad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0\rangle\ e.\texttt{tail} \rightsquigarrow V_t \langle E, H_1, C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1\rangle}$$

$$\text{(EqualT)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 1 \langle E, H_2, C_2 \wedge v_1 = v_2\rangle}$$

$$\text{(EqualF)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 0 \langle E, H_2, C_2 \wedge v_1 \neq v_2\rangle}$$

$$\text{(NEqualT)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 1 \langle E, H_2, C_2 \wedge v_1 \neq v_2\rangle}$$

$$\text{(NEqualF)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 0 \langle E, H_2, C_2 \wedge v_1 = v_2\rangle}$$

$$\text{(Add)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1 \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2 \langle E, H_2, C_2\rangle \qquad v \text{ fresh}}{\langle E, H_0, C_0\rangle\ e_1 + e_2 \rightsquigarrow v \langle E, H_2, C_2 \wedge v = v_1 + v_2\rangle}$$

Fig. 4. Symbolic evaluation of expressions

$$\text{(Skip)}\ \langle E, H, C\rangle\ \texttt{skip}\ \langle E, H, C\rangle$$

$$\text{(VarAss)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v \langle E_0, H_1, C_1\rangle \qquad E_1 \text{ fresh}}{\langle E_0, H_0, C_0\rangle\ x\ \texttt{:=}\ e\ \langle E_1, H_1, C_1 \wedge E_1 = E_0 \uplus \{x \mapsto v\}\rangle}$$

$$\text{(HeadAss)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r \langle E, H_1, C_2\rangle \qquad R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0\rangle\ l.\texttt{head}\ \texttt{:=}\ e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(v, V_t)\}\rangle}$$

$$\text{(TailAss)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v \langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r \langle E, H_1, C_2\rangle \qquad R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0\rangle\ l.\texttt{tail}\ \texttt{:=}\ e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(V_h, v)\}\rangle}$$

$$\text{(Seq)}\ \frac{\langle E_0, H_0, C_0\rangle\ s_1\ \langle E_1, H_1, C_1\rangle \qquad \{E_1, H_1, C_1\}\ s_2\ \{E_2, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ s_1 ; s_2\ \langle E_2, H_2, C_2\rangle}$$

$$\text{(IfThen)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v \langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s_1\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(IfElse)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v \langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v = 0\}\ s_2\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(WhileT)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v \langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s ; \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2, C_2\rangle}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(WhileF)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v \langle E, H_1, C_1\rangle}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_0, H_1, C_1 \wedge v = 0\rangle}$$

Fig. 5. Symbolic execution of statements

The completeness property states that any concrete execution of a program s with respect to an initial environment E and heap H is modeled by some abstract derivation represented by a set of constraints C such that there exists a solution to C that models both the initial and final environment and heap. In other words, our method is able to capture *all* executions of a program fragment s. In addition, the soundness property given below states the inverse, namely that our method does not model spurious executions.

Theorem 2 (Soundness)
Let s be a statement. If $\langle E_v, H_v, true\rangle\ s\ \langle E'_v, H'_v, C\rangle$ and if there exists a solution ρ for the set of constraints C then $\langle \rho(E_v), \rho(H_v)\rangle\ s\ \langle E, H\rangle$ such that

$$\langle E, H\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle.$$

2.4 Constraint Propagation

Among the four types of primitive constraints (Section 2.2), the equality and inequality constraints are easily defined as Herbrand equality and inequality, and appropriate implementations can be found in Prolog systems as, respectively, unification and the `dif/2` inequality constraint. The constraints on the environment and heap (membership and update of a mapping) on the other hand are specific to our purpose. We define them in terms of the following propagation rules, that allow us to infer additional constraints:

$$(o \mapsto o_1) \in M \wedge (o \mapsto o_2) \in M \Longrightarrow o_1 = o_2$$
$$M_1 \uplus \{o \mapsto o_1\} = M_2 \Longrightarrow (o \mapsto o_1) \in M_2$$
$$o \neq o' \wedge M_1 \uplus \{o \mapsto o_1\} = M_2 \wedge (o' \mapsto o_2) \in M_2 \Longrightarrow (o' \mapsto o_2) \in M_1$$

The above rules are easily implemented as Constraint Handling Rules (CHR) [10].

2.5 Search

In order to obtain concrete test cases, our constraint solver has to overcome two forms of non-determinism: 1) the non-determinism inherent to the extended operational semantics, and 2) the non-determinism associated to the selection of concrete values for the program's input. Traditionally, in Constraint Programming a problem with non-deterministic choices is viewed as a (possibly infinite) tree, where each choice is represented as a fork in the tree. Each path from the root of the tree to a leaf represents a particular set of choices, and has zero or one solution. In our context, a solution is of course a concrete test case. As the tree does not imply a particular order on the solutions, we are free to choose any *search strategy*, which specifies how the tree is navigated in search of the solutions. Moreover, since the problem tree can be infinite, we may select an incomplete search strategy, i.e. one that only visits a finite part of the tree. Let us have a more detailed look at these two forms of non-determinism and how they can be handled by a solver.

Non-Deterministic semantics. Several of the language constructs have multiple overlapping rules in the definition of the symbolic semantics. In particular those for if-then-else ((IfThen) and (IfElse)) and while ((WhileT) and (WhileF)) constructs imply alternate execution paths through the program. Also, observe that the while-construct is a possible source of infinity in the problem tree as the latter must in general contain a branch for each possible number of iterations of the loop body. This means that a solver is usually forced to use an *incomplete*

search strategy; for example a *depth-bounded* search strategy which does not explore the tree beyond a given depth.

Recall the example in Section 2.1 where the while-loop may iterate an arbitrary number of times. A depth-bounded search only considers test cases that involve iterations up to a given bound.

Non-Deterministic Values. As the following example shows, even a single execution path can introduce non-determinism in the solving process. Consider the program `y := x.tail`, which has only one execution path. This execution path merely restricts the initial environment and heap to $E_0 = \{\mathtt{x} \rightsquigarrow \mathtt{ptr}(A), \mathtt{y} \rightsquigarrow V_y\}$ and $(A \rightsquigarrow \mathtt{cons}(V_h, V_t)) \in H_0$. There are an infinite number of concrete test cases that satisfy these restrictions. Here are just a few:

E_0	H_0
$\{\mathtt{x} \rightsquigarrow \mathtt{ptr}(\mathtt{a1}), \mathtt{y} \rightsquigarrow \mathtt{nil}\}$	$\{\mathtt{a1} \rightsquigarrow \mathtt{cons}(0, \mathtt{nil})\}$
$\{\mathtt{x} \rightsquigarrow \mathtt{ptr}(\mathtt{a1}), \mathtt{y} \rightsquigarrow \mathtt{nil}\}$	$\{\mathtt{a1} \rightsquigarrow \mathtt{cons}(0, \mathtt{a1})\}$
$\{\mathtt{x} \rightsquigarrow \mathtt{ptr}(\mathtt{a1}), \mathtt{y} \rightsquigarrow \mathtt{nil}\}$	$\{\mathtt{a1} \rightsquigarrow \mathtt{cons}(1, \mathtt{nil})\}$
$\{\mathtt{x} \rightsquigarrow \mathtt{ptr}(\mathtt{a1}), \mathtt{y} \rightsquigarrow \mathtt{ptr}(\mathtt{a1})\}$	$\{\mathtt{a1} \rightsquigarrow \mathtt{cons}(0, \mathtt{nil})\}$
$\{\mathtt{x} \rightsquigarrow \mathtt{ptr}(\mathtt{a1}), \mathtt{y} \rightsquigarrow \mathtt{nil}\}$	$\{\mathtt{a1} \rightsquigarrow \mathtt{cons}(0, \mathtt{ptr}(\mathtt{a2})), \mathtt{a2} \rightsquigarrow \mathtt{cons}(0, \mathtt{nil})\}$

There are two kinds of unknown values: unknown integer V_i and unknown references V_r. Integers are easy: non-deterministically assign any natural number to an unknown integer: $\bigvee_{n \in \mathbb{N}} V_i = n$.

For the references the story is more involved. Assume that R is the set of references created so far, r' is a fresh reference, and V_i' and V_r' are fresh unknown integer and reference values. Then there are three assignments for an unknown reference V_r: 1) `nil`, 2) one of the previous references R, or 3) a new reference r'. In the last case, the heap must contain an additional cell with fresh unknown components.

$$V_r = \mathtt{nil} \vee (\bigvee_{r \in R} V_r = \mathtt{ptr}(r)) \vee (V_r = \mathtt{ptr}(r') \wedge (r' \mapsto \mathtt{cons}(V_i', V_r')) \in H_0)$$

In practice, we must again restrict ourselves to a finite number of alternatives. We may be interested in only a single solution: an arbitrary one, one that satisfies additional constraints or one that is minimal according to some criterion. Alternatively, multiple solutions may be desired, each of which *differs sufficiently* from the others based on some measure. All of these preferences can be expressed in terms of suitable search strategies. For instance, the minimality criterion is captured by a branch-and-bound optimization strategy.

2.6 Generalized Data Structures

So far we have only considered data structures composed of simple `cons` cells, for the sake of simplicity and concision in the definitions. However, our constraint-based approach can easily be extended to cope with arbitrary structures. Consider for instance this C-like struct for binary trees:

```
struct tree { int value;
              tree left;
              tree right; }
```

In order to deal with the `tree` type defined above , it suffices to extend both the concrete and the constraint semantics of ImpL with 1) a new `tree` constructor representing a triple and 2) three field selectors (e.g. `value`, `left`, and `right`) similar to the `cons` constructor and the `head` and `tail` selectors. In addition, the search process employed by the solver needs to be adjusted in order to generate arbitrary tree values. An unknown tree value V_t is assigned as follows:

$$V_t = \texttt{nil} \vee (\bigvee_{r \in R_t} V_t = \texttt{ptr}(r)) \vee (V_r = r' \wedge (r' \mapsto \texttt{tree}(V_i', V_l', V_r') \in H_0)$$

where R_t is the set of previously created tree references, r' is a fresh tree reference, and V_i', V_l' and V_r' are respectively a fresh unknown integer value and fresh unknown tree values. It should be clear to the reader that the above approach is easily generalized to arbitrary structures in a datatype-generic manner.

Also, other primitive types such as reals and booleans are easily supported by integrating additional off-the-shelf constraint solvers for them.

Moreover, note that invariants on the data structures, such as acyclicness, *can* be imposed on the unknown input in terms of additional constraints, e.g. provided by the programmer. This allows to seamlessly incorporate specification-level constraints into our method – similarly to [21,20].

3 Applications

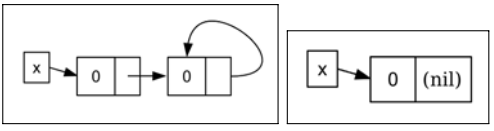
In this section we propose two applications of our method for test case generation. The first one consists in providing the programmer with (a visualization of) input/output pairs for the program under test satisfying a certain coverage criterion. We have developed a tool that allows to visualise such input/output pairs involving heap-allocated data structures based on Graphviz.[4] This allows the programmer to visually inspect them and verify that the program behaves as expected. For example, Fig. 6 depicts an input/output pair for the example program of Section 2.1.

A second application is the automatic creation of a test suite that can be repeatedly evaluated during regression testing, for example after certain parts of the code have been refactored. The main problem is to translate the data structures originating from a solution to a constraint set into executable code that 1) creates the data structures that are input to the program, and that 2) verifies whether the data structures output by the code correspond to the expected output. Hence, a concrete test case for a program P looks like

$$\mathit{Setup}\,;P\,;\mathit{Check}$$

where *Setup* sets up the initial environment and heap, and *Check* inspects the final environment and heap.

[4] http://www.graphviz.org/



(a) Input (b) Output

Fig. 6. Visualization of an input/output pair for the example program

```
  // setup phase
x := new cons(7,nil);
x.tail := x;
  // program under test
x := nil;
  // check phase
if (x == nil) then {
  accept := 1
} else {
  accept := 0
}
```

```
  // setup phase
x := new cons(7,nil);
x.tail := x;
  // program under test
if (x == nil) then {
  foundnil := 1;
  x := nil
} else {
  foundnil := 0;
  x := nil
}
  // check phase
if (x == nil) then {
  accept := 1
} else {
  accept := 0
}
```

Fig. 7. Testcase for the original (left) and refactored (right) code of Example 1

Example 1. Consider the simple program `x := nil`. One test configuration consists of an initial environment $E_0 = \{\texttt{x} \rightsquigarrow \texttt{ptr(r1)}\}$ and an initial heap $H_0 = \{\texttt{r1} \rightsquigarrow \texttt{cons}(7, \texttt{ptr(r1)})\}$. The final environment is $E_1 = \{\texttt{x} \rightsquigarrow \texttt{nil}\}$ and the final heap $H_1 = H_0$. The concrete test case for this test configuration looks like the code represented on the left of Figure 7. After running this test case, the variable `accept` contains `1` iff the test succeeds; otherwise it contains `0`.

If the program is changed, e.g. due to refactoring, the existing test case can be used to test the *modified* source code (regression testing). If we replace the program of Example 1 above by the refactored version `if (x == nil) { foundnil := 1; x := nil } else { foundnil := 0; x := nil }`, the above test case looks as the code on the right of Figure 7. Observe that we consider neither *garbage*, i.e. the parts of the heap H_1 that are unreachable from the environment E_1, nor newly introduced variables such as `foundnil` in the example.

Setup Phase. The inference rules depicted in Figure 8 explain how to construct the setup code of the test case from an initial environment E and heap H. The judgement $H, \emptyset \vdash_s E : s$ expresses that s is the setup code for environment E and heap H. The set of rules basically defines an algorithm that constructs the setup code by generating code for one element of the environment at a time. Note the role of the set A containing the references generated so far.

$$\text{(S-Done)}\ \frac{}{H, A \vdash_s \emptyset : \texttt{skip}}$$

$$\text{(S-Int)}\ \frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto n\} \cup E : l\ \texttt{:=}\ n\texttt{;}s}$$

$$\text{(S-Nil)}\ \frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \texttt{nil}\} \cup E : l\ \texttt{:=}\ \texttt{nil;}s}$$

$$\text{(S-NRef)}\ \frac{a \notin \mathit{domain}(A) \qquad (a \mapsto \texttt{cons}(v_h, v_t,)) \in H \qquad H, A \cup \{a \mapsto l\} \vdash_s \{l.\texttt{tail} : v_t\} \cup E : s}{H, A \vdash_s \{l \mapsto \texttt{ptr}(a)\} \cup E : l\ \texttt{:=}\ \texttt{new cons}(v_h, \texttt{nil})\texttt{;}s}$$

$$\text{(S-ORef)}\ \frac{(a \mapsto l') \in A \qquad H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \texttt{ptr}(a)\} \cup E : l\ \texttt{:=}\ l'\texttt{;}s}$$

Fig. 8. Setup Phase Algorithm

$$\text{(C-Done)}\ \frac{}{H, A \vdash_c \emptyset : \texttt{accept := 1}}$$

$$\text{(C-Int)}\ \frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto n\} \cup E : \texttt{if}\ l\ \texttt{==}\ n\ \texttt{then}\ s\ \texttt{else accept := 0}}$$

$$\text{(C-Nil)}\ \frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \texttt{nil}\} \cup E : \texttt{if}\ l\ \texttt{==}\ \texttt{nil then}\ s\ \texttt{else accept := 0}}$$

$$\text{(C-NRef)}\ \frac{a \notin \mathit{domain}(A) \qquad (a \mapsto \texttt{cons}(v_h, v_t,)) \in H \qquad H, A \cup \{a \mapsto l\} \vdash_c \{l.\texttt{head} : v_h, l.\texttt{tail} : v_t\} \cup E : s_1 \qquad l, s_1 \vdash_n \mathit{range}(A) : s_2}{H, A \vdash_c \{l \mapsto \texttt{ptr}(a)\} \cup E : \texttt{if}\ l\ \texttt{/=}\ \texttt{nil then}\ s_2\ \texttt{else accept := 0}}$$

$$\text{(C-ORef)}\ \frac{(a \mapsto l') \in A \qquad H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \texttt{ptr}(a)\} \cup E : \texttt{if}\ l\ \texttt{==}\ l'\ \texttt{then}\ s\ \texttt{else accept := 0}}$$

$$\text{(N-Base)}\ l, s \vdash_n \emptyset : s$$

$$\text{(N-Rec)}\ \frac{l, s \vdash_n R : s'}{l, s \vdash_n \{l'\} \cup R : \texttt{if}\ l\ \texttt{/=}\ l'\ \texttt{then}\ s'\ \texttt{else accept := 0}}$$

Fig. 9. Check Phase Algorithm

Check Phase. Likewise, the algorithm given by the inference rules in Figure 9 explains how to construct the check code of the test case from the final environment E and heap H. The judgement $H, \emptyset \vdash_c E : s$ expresses that s is the setup code for environment E and heap H.

4 Related Work and Conclusion

A large amount of work exists in the field of automatic test case generation for imperative programs. The arguably simplest method is *random generation* of test data [3,8]. In *symbolic evaluation* techniques (e.g. [5,15,18]), the input parameters are replaced by symbolic values, in order to derive a symbolic expression representing the values of a program's variables. This approach is notably used in the ATGen tool for structural coverage of Spark ADA programs [17]. In so-called *dynamic* approaches, the program is actually executed on input data that is arbitrarily chosen from a given domain. The input data is then iteratively refined to obtain a final test input such that the execution follows a chosen path,

or reaches a chosen statement [14,9]. Another approach is the generation of test cases based on a formal specification of the program, written for example in the B language [16,2].

Constraint-based test data generation was originally introduced in [7] in the context of mutation testing [6] and aims at transforming the automatic test data generation problem into a CLP problem over finite domains. This approach has been used in many works, including [12,13,1]. It is also used in two different testing tools, Godzilla [19] and InKA [11]. The latter notably generates test data satisfying different criteria such as statement coverage, branches coverage and MC/DC[5]. The main advantage of our technique over this work is that it can be parametrised by any coverage criterion, instead of proposing a limited set of built-in coverage criteria.

In this paper, we have presented a constraint-based approach for generating white-box test cases for a small but representative imperative programming language. Our technique is able to generate complex heap-allocated and pointer-based data structures without any user intervention. The selection of both execution paths and concrete test inputs are modelled uniformly as a single constraint problem. An interesting advantage of our technique over most existing work is that we use parametrizable CP search strategies within the solver in order to express (coverage) criteria the generated test suites must satisfy. Our framework would therefore be able to generate test cases accordingly to any (coverage) criterion, instead of proposing a predefined set of built-in criteria.

We have proposed two applications of this approach; the first one is to provide the programmer a visualization of input-output pairs for his program. The second one is the creation of a concrete test case for checking a refactored version of the original program. Our prototype and example programs are available at `http://www.cs.kuleuven.be/~toms/Testing/`. In future work, we will investigate the exact relation between the use of particular search strategies for constraint solving and the generation of *interesting* sets of test cases according to different adequacy criteria. Other topics of further work include extending the ImpL language towards more involved imperative and object-oriented languages.

References

1. Albert, E., Gómez-Zamalloa, M., Puebla, G.: Test data generation of bytecode by clp partial evaluation, pp. 4–23 (2009)
2. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Utting, M., Vacelet, N.: BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In: Proceedings of FATES 2002, pp. 105–120, August 2002, Technical Report, INRIA (2002)
3. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Syst. J. 22(3), 229–245 (1983)
4. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 9(5), 193–200 (1994)

[5] Modified condition/decision coverage [4].

5. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. 2(3), 215–222 (1976)
6. DeMillo, R.A.: Test adequacy and program mutation. In: 11th International Conference on Software Engineering, May 1989, pp. 355–356 (1989)
7. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
8. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Trans. Software Eng. 10(4), 438–444 (1984)
9. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. 5(1), 63–86 (1996)
10. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Logic Programming, Special Issue on Constraint Logic Programming 37(1-3), 95–138 (1998)
11. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. SIGSOFT Softw. Eng. Notes 23(2), 53–62 (1998)
12. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
13. Gotlieb, A., Denmat, T., Botella, B.: Goal-oriented test data generation for programs with pointer variables. Computer Software and Applications Conference, Annual International 1, 449–454 (2005)
14. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. SIGSOFT Softw. Eng. Notes 23(6), 231–244 (1998)
15. King, J.C.: Symbolic execution and program testing. ACM Commun. 19(7), 385–394 (1976)
16. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 21–40. Springer, Heidelberg (2002)
17. Meudec, C.: Atgen: automatic test data generation using constraint logic programming and symbolic executionatgen. Software Testing Verification and Reliability 11(2), 81–96 (2001)
18. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
19. Offutt, A.J., Jin, Z., Pan, J.: The dynamic domain reduction procedure for test data generation: Design and algorithms. Software - Practice and Experience 29, 167–193 (1994)
20. Offutt, A.J., Liu, S.: Generating test data from sofl specifications. The Journal of Systems and Software 49, 49–62 (1999)
21. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java pathfinder. SIGSOFT Softw. Eng. Notes 29(4), 97–107 (2004)
22. Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4) (1997)

Using Rewrite Strategies for Testing BUpL Agents

Lăcrămioara Aştefănoaei[1], Frank S. de Boer[1,2], and M. Birna van Riemsdijk[3]

[1] Centrum voor Wiskunde en Informatica (CWI), P. O. Box 94079,
1090 GB Amsterdam, The Netherlands
Tel.: +31 (0)20 592 4368
L.Astefanoaei@cwi.nl
[2] LIACS - Leiden University, The Netherlands
[3] TU, Delft, The Netherlands

Abstract. In this paper we focus on the problem of testing agent programs written in BUpL, an executable, high-level modelling agent language. Our approach consists of two main steps. We first define a formal language for the specification of test cases with respect to BUpL. We then implement test cases written in the formal language by means of a general method based on rewrite strategies. Testing an agent program with respect to a given test case corresponds to strategically executing the rewrite theory associated to the agent with respect to the strategy implementing the test case.

Keywords: Agent Languages, Testing, Rewriting, Strategies.

1 Introduction

An agent is commonly seen as an encapsulated computer system that is situated in some environment and that is capable of flexible, *autonomous action* in that environment in order to meet its design objectives [12], or *dynamic goals*. An important line of research in the agent systems field is the design of agent languages [3] with emphasis on the use of formal methods. The guiding idea is that agent-specific concepts such as beliefs (representing the environment and possibly other data the agent has to store), goals (representing the desired state of the environment), and plans (specifying which sequences of actions and possibly compositions of other plans to execute in order to reach the goals) facilitate the programming of agents. Along these lines, we take as case of study in this paper a simple variant of 3APL [7], the agent language BUpL, which is introduced in [1]. There the authors advocate the use of the Maude language [4] and its supporting tools for *prototyping*, *executing*, and *verifying* BUpL agents. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Namely, being a rewrite-based framework, it makes it is easy to prototype modelling languages with an operational semantics by means of rewrite theories [8], and it provides mechanisms for verifying programs and language definitions by means of LTL model-checking [6]. Furthermore, the inherent reflective feature of rewriting logic (and of Maude, in particular) offers an alternative to model-checking by means of rewrite strategies.

In this paper, we extend the results from [1]. More precisely, we investigate the problem whether a BUpL agent is conformant with respect to a given specification, however,

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 143–157, 2010.

from a different perspective. We understand conformance as the refinement relation in [1], that is, it holds when the set of traces of a BUpL agent is included in the set of traces of the specification. In a straight-forward approach, one solution is to look at each execution trace of the agent and to check whether it is also a trace of the specification. However, this is often practically unfeasible due to large (possibly infinite) sets of agent executions. A more clever way is to consider the trace inclusion problem in the opposite direction, that is, to look first at the traces of the specification and to check whether these are also traces of the agent. Usually, "check" is achieved by model-checking or inductive verification. However, both approaches have their disadvantages: with model-checking one might run into the state explosion problem, while inductive verification is not automatic. An orthogonal technique is to use *testing*.

In the literature, the very basic idea behind testing is that it aims at showing that the intended and the actual behaviour of a system differ by generating and checking individual executions. Testing object-oriented software has been extensively researched and there are many pointers in the literature with respect to manual and automated, partition and random testing, test case generation, criteria for test selection (please see [9] for an overview). In an agent-oriented setup, there are less references. A few pointers are [13,10] for developing test units from different agent methodologies, however the direction is orthogonal to the one we consider.

Our testing methodology consists of the following steps. We see the traces of the specification as the basic constructions for *test cases*. Since specifications are meant to be "small", generating test cases is a much simpler task than exhaustively exploring possible agent executions. Either represented by regular expressions or by finite transition systems, specifications can be used to generate test cases by model-checking, for example. Traces are *deterministic*, and since we build test cases on top of traces, also test cases are deterministic, in contrast to specifications. This is an important feature which makes testing an efficient approach. We define test cases as pairs of tests on actions and tests on facts. The tests on actions are finite sequences of pairs (a, R) where a is the action to be executed and R is the set of actions which are allowed to be executed at a given state. Whenever the agent *cannot execute* the action specified by the test on actions, or whenever the agent *can execute* a *forbidden* action, the corresponding trace represents a nonconformant execution. Tests on facts are temporal formulae that are checked on the traces generated with respect to tests on actions. They can be further used to detect "bad" executions.

Given that we define a formal language for expressing *what* a test case is, we then describe *how* to implement test cases. Namely, we provide a strategy-based mechanism to define *test drivers*. In a rewrite-based framework, strategies are meant to control nondeterministic executions by instrumenting the rewrite rules at a meta-level. Usually, in concrete implementations the nondeterminism is reduced by means of scheduling policies. While testing a concrete implementation, e.g., a multi-threaded Java application, there is no obvious distinction between testing the program itself and testing the default scheduling mechanism of the threads. We emphasise that the language we consider, BUpL, is a modelling language, where the *nondeterminism* in choices among plans, exception handling mechanisms and internal actions is a main aspect we deal with.

Strategies give a great degree of flexibility which becomes important when the interest is in verification. For example, in our case, in order to analyse or experiment with a new testing formalism one only needs to change the *strategy* instead of changing *the semantics of the agent language* or *the agent program* itself.

Though test cases are deterministic, test drivers need to search all intermediary states that can be reached by nondeterministically executing internal BUpL computations. Defining test drivers by means of strategies is an elegant solution to the implicit nondeterminism in BUpL. However, it does not directly solve the problem of possibly divergent executions of internal steps. To avoid some divergent computations, we need to impose restrictions on the application of the strategies. This makes it less intuitive that test drivers are faithfully implementing test cases, and thus the last issue we focus upon is the correctness of our mapping between test cases and test drivers.

2 BUpL Agents by Example

In this section, we briefly present the syntax and semantics of BUpL for ease of reference and completeness. A BUpL agent has an initial belief base and an initial plan. A belief base is a collection of ground (first-order) atomic formulae which we refer to as beliefs. The agent is supposed to execute its initial plan, which is a sequential composition and/or a nondeterministic choice of actions or composed plans. The semantics of actions is defined using pre and post conditions. An action can be executed if the precondition of the action matches the belief base. The belief base is then updated by adding or removing the elements specified in the postcondition. When, on the contrary, the precondition does not match we say the execution of the action (or the plan of which it is a part) fails. In such a case repair rules are applied (if any), and this results in replacing the plan that failed.

Syntactically, a BUpL agent is a tuple ($\mathcal{B}_0$, p_0, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$), where $\mathcal{B}_0$ is the initial belief base, p_0 is the initial plan, $\mathcal{A}$ is the set of internal and observable actions, $\mathcal{P}$ are the plans, and $\mathcal{R}$ are the repair rules. The initial belief base and plan form the initial mental state of the agent. To illustrate the syntax, we take as an example a BUpL agent that solves the Hanoi towers problem. We represent blocks by natural numbers. We assume that the initial configuration is of three blocks arranged on a table as follows: blocks 1 and 2 are on the table (0), and 3 is on top of 1. The agent has to rearrange them such that they form the tower 321 (1 is on 0, 2 on top of 1 and 3 on top of 2). The only action the agent can execute is $move(x, y, z)$ to move block x from block y onto z, if x and z are clear. Blocks can always be moved to the table, i.e., the table is always clear.

$\mathcal{B}_0$ = { $on(3,1)$, $on(1,0)$, $on(2,0)$, $clear(2)$, $clear(3)$, $clear(0)$ }
p_0 = $build$
$\mathcal{A}$ = { $move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \{on(x,z), \neg on(x,y), \neg clear(z))\}$ }
$\mathcal{P}$ = { $build = move(2,0,1); move(3,0,2)$ }
$\mathcal{R}$ = { $on(x,y) \leftarrow move(x,y,0); build$ }

Fig. 1. A BUpL Toy Agent

The BUpL agent from Figure 1 is modelled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally writing a plan to move block 2 onto 1. This is not possible, since block 3 is already on top of 1. Similar scenarios can easily arise in multi-agent systems: imagine that initially 3 is on the table, and the agent decides to move 2 onto 1; imagine also that another agent comes and moves 3 on top of 1, thus moving 2 onto 1 will fail. The failure is handled by the repair rule $on(x,y) \leftarrow move(x,y,0); build$. Choosing $[x/3][y/1]$ as a matcher enables the agent to move block 3 onto the table and then the initial plan can be restarted.

We shortly describe (please see [1] for more details) the BUpL operational semantics. The states of BUpL agents are pairs of belief bases and plans, symbolically denoted by $(\mathcal{B}, p)$. These BUpL states change with respect to the transition rules in Figure 2.

$$\frac{p = (a; p') \qquad a = (\psi, \xi) \in \mathcal{A} \qquad \theta \in Sols(\mathcal{B} \models \psi)}{(\mathcal{B}, p) \xrightarrow{(\tau/a\theta)} (\mathcal{B} \uplus \xi\theta, p'\theta)} \; ((i/o)\text{-}act)$$

$$\frac{}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\tau} (\mathcal{B}, p_i)} \; (sum_i, i \in \{1, 2\})$$

$$\frac{(\mathcal{B}, a; p) \not\xrightarrow{a} \qquad \phi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B} \models \phi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \; (fail\text{-}act)$$

$$\frac{\pi(x_1, \ldots, x_n) := p}{(\mathcal{B}, \pi(t_1, \ldots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \ldots, t_n))} \; (\pi)$$

Fig. 2. BUpL Rules

The rules $(i\text{-}act)$ and $(o\text{-}act)$[1] capture the effects of performing action a (either internal or observable), which is the head of the current plan. These rules basically say that for a given as a pair of a precondition (i.e., a first order formula) ψ and a post-condition (i.e., a set of literals) ξ, if θ is a solution (i.e., a substitution) such that ψ matches[2] $\mathcal{B}$ (i.e., $\mathcal{B} \models \psi\theta$), then the current mental state changes to a new one, where the belief base is updated by adding/removing the positive/negative literals from ξ. It is also the case that the current plan becomes $p'\theta$, that is, the "tail" of the previous plan p instantiated with respect to θ. The transition rule $(fail\text{-}act)$ handles exceptions. If the head of the current plan is an action that cannot be executed (the set of solutions for the matching problem is empty) and if there is a repair rule $\phi \leftarrow p'$ such that the new matching problem $\mathcal{B} \models \phi$ has a solution θ then the plan is replaced by $p'\theta$. The transition rule (π) implements "plan calls". If the abstract plan $\pi(x_1, \ldots, x_n)$ defined as $p(x_1, \ldots, x_n)$ is instantiated with the terms $t_1, \ldots, t_n$ then the current plan becomes $p(t_1, \ldots, t_n)$ which stands for $p[x_1/t_1] \ldots [x_n/t_n]$. The transition rule (sum_i) replaces a choice between two plans by either one of them.

[1] For simplicity, they are denoted by the same transition $((i/o)\text{-}act)$. Syntactically, the only difference between them is that the label for *i-act* is τ.

[2] Belief bases are sets of *ground* positive literals, thus we solve a generalisation of the *matching* and not unification problem.

2.1 Prototyping BUpL Agents as Rewrite Theories

In [1] it is shown how the operational semantics of BUpL can be implemented and executed as a *rewrite theory* in Maude. The main advantage of using Maude for this is that the translation of operational semantics into Maude is direct [11], ensuring a *faithful implementation*. Thanks to this, it is relatively easy to experiment with different kinds of semantics, making Maude suitable for rapid *prototyping*.

We do not explain here the way BUpL is prototyped in Maude but we briefly illustrate at a more generic level how BUpL transition rules map into rewrite rules. A rewriting logic specification or rewrite theory is a tuple $\langle \Sigma, E, R \rangle$, where Σ is a signature consisting of sorts (types) and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the *terms* that form the state of the system. These terms can be rewritten using equations and rewrite rules. Rewrite rules are used to model the dynamics of the system, i.e., they describe transitions between states. Equations form the functional part of a rewrite theory, and are used to reduce terms to their "normal form" before they are rewritten using rewrite rules. The application of rewrite rules is intrinsically nondeterministic, which makes rewriting logic a good candidate for modelling concurrency.

In our case, the signature (the set of terms) maps the mental states of the agents and the rewrite rules map BUpL transitions, thus they describe how BUpL mental states change. There is a natural encoding of transition rules as *conditional rewrite rules*. The general mathematical format of a conditional rewrite rule is as follows:

$$l : t \longrightarrow t' \; if \; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

It basically says that l is the label of the rewrite rule $t \longrightarrow t'$ which is used to "rewrite" the term t to t' when the conditions on t are satisfied. Such conditions can be either equations like $u_i = v_i$, memberships like $w_j : s_j$ (that is, w_j is of type s_j) or other rewrites like $p_k \longrightarrow q_k$. For example, the corresponding rewrite rule for transition (act) in the case of observable actions is:

$$\textit{o-act} : (\mathcal{B}, p) \longrightarrow (update(\mathcal{B}, \xi\theta), p'\theta) \; \textit{if} \; p = \textit{o-a}; p' \wedge \textit{o-a} = (\psi, \xi) \wedge \\ \theta = match(\mathcal{B}, \psi) \wedge \textit{o-a} : A^o$$

where A^o denotes the sort of observable actions. As it will be clear in the next sections, we need the distinction between internal and observable actions for testing, in order to have a more expressive framework.

All other transition rules are encoded as rewrite rules in a similar manner and we do not further explain them. In what follows, we only need to remember that each transition has a corresponding rewrite rule labelled with the same name.

2.2 Meta-controlling BUpL Agents with Rewrite Strategies

In this section we make a short overview of the strategy language presented in [5] with illustrations of how strategies can be used to control the execution of BUpL agents. We denote the rewrite theory that implements the operational semantics of BUpL by

T. Given a BUpL agent, we denote by ms terms corresponding to BUpL mental states $(\mathcal{B}, p)$. These terms can be rewritten by the rewrite rules from T. We further denote by S the strategy language from [5]. The strategy language S can be viewed as a transformation of the rewrite theory T into $S(T)$ such that the latter represents the execution of T in a controlled way. Given a strategy expression E in the strategy language S, the application of E to ms is denoted by $E@ms$. The semantics of $E@ms$ is the set of successors which result by rewriting ms using the rewrite rules from $S(T)$.

The simplest strategies we can define in the strategy language S are the constants *idle* and *fail*: *idle* $@\, ms = \{ms\}$, *fail* $@\, ms = \emptyset$. Another basic strategy consists of applying to a BUpL agent state ms a rule identified by one of the labels: *i-act*, *o-act*, *fail-act*, or *sum*, possibly with instantiating some variables appearing in the rule. The semantics of $l@ms$, where l is one of the above rule labels, is the set of all terms to which ms rewrites in one step using the rule labelled l. For example, applying the strategy *o-act* to the initial state $(\mathcal{B}_0, build)$ of the BUpL builder from Figure 1 has as result $\emptyset$ because initially the only possible observable action $move(2, 0, 1)$ fails. However, applying the strategy *fail-act* has as result the set $\{(\mathcal{B}_0, (move(3, 1, 0); build)), (\mathcal{B}_0, (move(1, 0, 0); build)), (\mathcal{B}_0, (move(2, 0, 0); build))\}$, thus the set of all possible states reflecting a solution to the matching problem $\mathcal{B}_0 \models on(x, y)$. Of course, some of these resulting states are meaningless in the sense that there is no point in moving a block from the table to the table. A much more adequate strategy is *fail-act*$[\theta \leftarrow [x/3][y/1]]$, that is, to explicitly give the value we are interested in to the variable θ which appears in the rewrite rule *fail-act*. This results in a set containing only the state $(\mathcal{B}_0, (move(3, 1, 0); build))$.

Since matching is one of the basic steps that take place when applying a rule, another strategy one can define is *match* T *s.t.* C. When applied to a given state term ms, the result of this strategy is $\{ms\}$ if ms matches the pattern T and the condition C is satisfied with the substitutions for the variables obtained in the matching, otherwise $\emptyset$. For example, applying *match* $(\mathcal{B}, p)$ *s.t.* $on(2, 1) \in \mathcal{B}$ to $(\mathcal{B}_0, build)$ has as result $\emptyset$ because $on(2, 1)$ is not in $\mathcal{B}_0$. The language S allows further strategies definitions by combining them under the usual regular expression constructions like concatenation (";"), union ("|") and iteration ("*", "+"). Thus, given E, E' as already defined strategies, we have that $(E; E')@ms = E'@(E@ms)$, meaning that E' is applied to the result of applying E to ms. The strategy $(E \mid E')@ms$ defined as $(E@ms) \cup (E'@ms)$ means that both E and E' are applied to ms. The strategy $E^+@ms$ is defined as $\bigcup_{i \geq 1} (E^i@ms)$ with $E^1 = E$ and $E^n = E^{n-1}; E$, $E^* = idle \mid E^+$, thus it recursively re-applies itself. It is also possible to define *if-then-else* combinators. The strategy $E \ ? \ E' : E''$ defined as (if $(E@ms) = \emptyset$ then $E'@(E@ms)$ else $E''@ms$ fi) has the meaning that if, when evaluated in a given state term, the strategy E is successful then the strategy E' is evaluated in the resulting states, otherwise E'' is evaluated in the *initial* state. The *if-then-else* combinator is further used to define the following strategies. The strategy *not*$(E) = E$? *fail* : *idle* which reverses the result of applying E. The strategy *try*$(E) = E$? *idle* : *idle* changes the state term if the evaluation of E is successful, and if not, returns the initial state. The strategy *test*$(E) =$ *not*(E) ? *fail* : *idle* checks the success (resp. the failure) result of E but it does not change the initial state. The strategy $E! = E^*$; *not*(E) "repeats until the end", that is, it applies E until no longer possible.

3 Formalising Test Cases

Our test case format is based on two main concepts: observable actions and facts as appearing in belief bases. Our test case format is a kind of black box testing, aimed at testing the observable behaviour of agents. For this reason, we have made a distinction between internal and observable actions. The idea is that the execution of observable actions is visible from outside the agent. Observable actions can be actions the agent executes in the environment in which it operates. In the sequel, we will sometimes omit the adjective "observable" if it is clear from the context.

We introduce a general test case format that allows to express that certain sequences of observable actions are executed, and that the belief bases of the corresponding trace satisfy certain properties. That is, we consider that a test case $\mathcal{T}$ is a pair consisting of a test on actions $\mathcal{T}_a$ and a test on facts $\mathcal{T}_f$. Tests on actions are finite sequences of pairs $(a_0, R_0); \ldots; (a_n, R_n)$. Each pair (a_i, R_i) consists of a ground observable action a_i to be executed and a set of actions R_i which are allowed to be executed from the current state. The idea is that a test on actions controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. Furthermore, the sets R can be used to identify "bad" traces. If, at a certain state of execution, the agent can perform a *forbidden* action, i.e., which is not allowed by the test case, then the corresponding trace is seen as a counter-example. If no restriction is imposed on the enabled actions we simply use the notation a instead of the pair (a, R). It is then the case that a counter-example can be generated when the agent cannot execute the action indicated by the test. Tests on actions can be derived from a given specification by means of model-checking, for example. We stress that though the specification may be nondeterministic, tests on actions should be deterministic. This is crucial for reducing the state space and makes this approach essentially different from search techniques since it is more efficient. Tests on facts are specified like LTL formulae. For ease of presentation, we work only with a subset of basic formulae:

$$\mathcal{T}_f ::= true \mid fact \mid \neg fact \mid \Box(\neg \bigcirc true \rightarrow fact) \mid fact \wedge fact \mid \Box fact \mid \Diamond fact$$

with $fact$ being a ground atomic formula. Observe that the syntax allows also test cases consisting of tests on actions only, $(\mathcal{T}_a, true)$ which we write shortly as $\mathcal{T}_a$. The LTL formula $\Box(\neg \bigcirc true \rightarrow fact)$ can be used to check if $fact$ holds in the last states, that is, in the states reachable after executing the test on actions. Tests on facts are meant to provide additional counter-examples besides those reflecting forbidden actions. While tests on actions can be automatically derived from the specification (where the tester needs only to choose adequate test cases), using tests on facts requires more effort and intuition from the tester. For illustration purposes, we provide an example of an adequate test on facts by the end of the paper.

To define formally when a BUpL agent satisfies a test we use induction on the structure of test cases. We denote the application of a test $\mathcal{T}$ on an initial configuration (an initial BUpL mental state) ms_0 as $\mathcal{T}@ms_0$. The (set) semantics is defined such that it yields the set of final states reachable through executing the agent restricted by the test, i.e., only those actions are executed that comply with the test. This means that an agent

with initial mental state ms_0 satisfies a test $\mathcal{T}$ if $\mathcal{T}@ms_0 \neq \emptyset$, in which case we say that a test $\mathcal{T}$ is *successful*.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms \mid ms_0 \stackrel{a}{\Rightarrow} ms\}, & \mathcal{T} = (a, R) \wedge R(ms_0) \subseteq R \\ \emptyset, & \mathcal{T} = (a, R) \wedge R(ms_0) \not\subseteq R \\ \mathcal{T}_a^2@(\mathcal{T}_a^1@ms_0), & \mathcal{T} = \mathcal{T}_a^1; \mathcal{T}_a^2 \\ \{ms \mid ms \in \mathcal{T}_a@ms_0 \wedge \Pi_{ms_0}^{\mathcal{T}_a}(ms) \models \mathcal{T}_f\}, & \mathcal{T} = (\mathcal{T}_a, \mathcal{T}_f) \end{cases}$$

The arrow $\stackrel{a}{\Rightarrow}$ stands for $\Rightarrow\stackrel{a}{\rightarrow}\Rightarrow$, where $\Rightarrow$ denotes the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$, and $R(ms)$ denotes the set of actions ready to be executed from ms, i.e., $R(ms) = \{a \mid \exists ms' \text{ s.t. } ms \stackrel{a}{\Rightarrow} ms'\}$. The idea behind the definition of the semantics of $(a, R)@ms_0$ is that the test should be successful for ms_0 if action a can be executed in ms_0, while the enabled actions from the states reached by doing a should be a subset of R (defined by $R(ms) \subseteq R$). The result is then the set of mental states resulting from the execution of a, as defined by $\{ms \mid ms_0 \stackrel{a}{\Rightarrow} ms\}$. We need to keep those mental states to allow a compositional definition of the semantics. In particular, when defining the semantics of $\mathcal{T}_a^1; \mathcal{T}_a^2$ we need the mental states resulting from applying the test $\mathcal{T}_a^1$, since those are the mental states in which we then apply the test $\mathcal{T}_a^2$, as defined by $\mathcal{T}_a^2@(T_a^1@ms_0)$. In the definition of the semantics of $(\mathcal{T}_a, \mathcal{T}_f)$, by abuse of notation, we use $\Pi_{ms_0}^{\mathcal{T}_a}(ms)$ to denote the paths from ms_0 to ms which are taken while executing $\mathcal{T}_a$. These paths are with respect to observable actions, that is, we abstract from intermediary states reached by doing τ steps. More specifically, each state in a path is reached from the previous by executing an observable action and then executing a number of τ steps until an observable action is again about to be executed (or no transitions are possible). In the initial state, first τ steps can be executed before the first observable action is executed. Tests on facts are thus checked in states resulting from the execution of an observable action and as many τ steps as possible. We call these states *stable*. The definition says that the result of applying the test $(\mathcal{T}_a, \mathcal{T}_f)$ is a subset of $\mathcal{T}_a@ms_0$, namely, those states ms which are reachable after executing $\mathcal{T}_a$ and the corresponding path LTL satisfies $\mathcal{T}_f$.

Our language is such that tests on facts can be omitted. By design, they are meant to provide more expressivity and to give more freedom to the tester. One might raise the issue that inspecting facts classifies our method as white-box testing. However, since facts can be deduced from the effects of actions, our method lies at the boundary between black-box and gray-box testing. In order to define test cases, there is no need to understand the way BUpL agents work (i.e., the internal mechanism for updating states or the structure of repair rules and plans), but only to look at basic actions, which we see as the interface of BUpL agents.

4 Using Rewrite Strategies to Define Test Drivers

In this section we describe how to define *test drivers* for test cases by means of the strategy language S. To give some intuition and motivation, we consider the way one would implement the basic test case a. By definition, the application of this test case to a BUpL mental state ms is the set of all mental states which can be reached from ms by

executing the observable action a *after eventually executing* τ *steps corresponding to internal actions, applying repair rules or making choices*, i.e., after computing closure sets of particular types of rewrite rules. It thus represents a *strategic* rewriting of ms. We are only interested in those rewritings which finally make it possible to execute a. To achieve this at the object-level means to have a procedure implementing the computation of the closure sets. However, the semantics of the application of the test a is independent of the computation of closure sets. Following [5], we promote the design principle that automated deduction methods (e.g., closure sets of τ steps) should be specified *declaratively* as nondeterministic sets of inference rules and not *procedurally*. Depending on the application, specific algorithms for implementing the specifications should be given as *strategies* to apply the inference rules. This has the implication that there is a clear separation between *execution* (by rewriting) at the object-level and *control* (of rewriting) at the meta-level.

In what follows, for ease of reference, we denote by $\mathbb{S}$ (resp. $\mathbb{T}$) the set of strategies (tests) and by s the mapping from tests to test drivers, i.e., $s : \mathbb{T} \rightarrow \mathbb{S}$. Since the definition of tests is inductive, so is the definition of s. We first consider the test drivers for tests on actions:

$$s(\mathcal{T}) = \begin{cases} allow(R)\,;do(a), & \mathcal{T} = (a, R) \\ s(\mathcal{T}_1)\,;s(\mathcal{T}_2), & \mathcal{T} = \mathcal{T}_1\,;\,\mathcal{T}_2 \end{cases}$$

thus sequences of tests map to sequences of strategies. We describe the basic test driver $do(a)$ in more detail. Observe that though tests on actions are deterministic, there are still possibly many executions due to internal actions, choices in plans and repair rules. Thus the test driver must search "all" possible intermediary states which can be reached by doing τ steps. By means of strategies, this is an easy process. By definition, the transitive closure of τ steps, $\Rightarrow$, is $\xrightarrow{\tau}{}^{*}$, with τ being one of the label *sum*, *i-act*, or *fail-act* and the corresponding being maximal, in the sense that no τ steps are possible from the last state. Thus, in a naive approach, we could simply consider the following test driver:

$$tauClosure = (\textit{sum} \mid \textit{i-act} \mid \textit{fail-act})!$$

which is clearly implementing $\Rightarrow$. However, though the order of application of the τ steps does not matter when the computation paths are finite, this is no longer the case when considering infinite paths. Consider an extraneous agent program with a plan $p = \textit{i-a} + \textit{i-b}$ where *i-a* is always enabled and *i-b*, on the contrary, is never enabled and a repair rule $(true \leftarrow \textit{i-b})$ which says that whenever there is a failure repair it by executing *i-b*. Applying $tauClosure$ as defined above we obtain two solutions corresponding to a finite path reflecting the choice for executing *i-a* and a divergent path reflecting the choice for executing *i-b* then failing all the time. As long as we are only interested in the "first" solution, then $tauClosure$ is fine, however, if we want to generate also the "next" solution then the computation will not terminate. From this we conclude that we may lose termination if any application order is allowed while we may be able to achieve it if we impose a certain order. Since one source of non-termination is mainly in a sort of "unfairness" with regard to enabled internal actions, a much more adequate test driver is implemented if we enforce the execution of internal actions after eventually applying the sequence (*sum*; *fail-act*). That is, $tauClosure$ becomes:

$$tauClosure = (try(sum); try(fail\text{-}act); i\text{-}act)!; try(sum); try(fail\text{-}act)$$

We make a few observations with respect to the new definition of $tauClosure$. First, since one might expect multiple sum and fail applications before an internal action is executed, it is no longer immediately clear that $tauClosure$ faithfully implements $\Rightarrow$. We present a correctness proof by the end of the section. Second, because we use the sequential strategy, we need to surround both *sum* and *fail-act* by *try* blocks. Otherwise, if either one of them were not applicable, i.e., the current plan is not a sum and the "head" action is enabled, then the strategy (*sum* ; *fail-act*; *i-act*) fails which is not what we want. By means of the parametrised strategy *try* the initial state is preserved in the case that *sum* or *fail-act* fails. Third, we order *fail-act* after *sum* because if we were to use the strategy (*try*(*sum* | *fail-act*) ; *i-act*) and the current plan is a sum of two failing plans, then the whole strategy fails though there might have been possible to replace the failing plans with a "good" plan by applying *fail-act*. Fourth, we require that repair rules are of a particular format, that is $\phi \leftarrow p$ with p not containing the sum operator. This is in order to avoid situations where the application of *fail-act* entails the application of *sum* which entails the application of *fail-act* and so forth (that is, non-terminating strategies (*sum* ; *fail-act*)!). Such format does not result in the loss of expressivity since having one repair rule $\phi \leftarrow p_1 + p_2$ is equivalent to having two repair rules $\phi \leftarrow p_i$, with $i \in \{1, 2\}$. Fifth, the use of strategies can be tricky. Though one might be tempted to use the strategy *try*(*sum ; fail-act*) instead of *try*(*sum*) ; *try*(*fail-act*), the first one is "wrong", meaning that if *fail-act* is not applicable after *sum* then the original state is returned instead of the one reached by applying *sum*. The last observation is with respect to the normalisation strategy. Since "!" returns the state previous to the one that failed, we need to apply again *try*(*sum*); *try*(*fail-act*) to make sure that from the resulting state no τ steps can be taken.

By means of $tauClosure$, the definition of $do(a)$ is straight-forward:

$$do(a) = tauClosure; o\text{-}act[o\text{-}a \leftarrow a]; tauClosure$$

which corresponds to the definition of $\stackrel{a}{\Rightarrow}$. We note that $tauClosure$ is no longer applicable when *i-act* fails after *sum* and *fail-act* have been applied. This means that the only possible scenario is that the head of the current plan is an observable action. If this action is in fact a, then *o-act*[*o-a* $\leftarrow$ a] is successful, otherwise it fails.

The definition of the strategy $allow(R)$ makes use of the *match* construction:

$$allow(R) = \textit{match } ms \textit{ s.t. } ready(ms) \subseteq R$$

which means that $allow(R)$ succeeds if the current mental state satisfies the condition $ready(ms) \subseteq R$, where *ready* is a function defined on BUpL mental states. This function is implemented such that it returns the set of actions ready to be executed. For simplicity, we do not detail its implementation but briefly describe it. Recall that BUpL mental states are pairs of belief bases and plans. The function *ready* reasons on possible cases. If the current plan is a sum of plans then *ready* is called recursively. Otherwise, depending on the action a in the head of the plan, either a is enabled and so the function *ready* returns a, or a fails and the function *ready* recursively considers all the plans that can substitute the current one, that is, it recursively analyses the active repair rules.

So far, we have focused on tests on actions $\mathcal{T}_a$. We focus now on the general test cases$(\mathcal{T}_a, \mathcal{T}_f)$. We begin by first considering the test driver implementing the test case for checking whether $fact$ is in the last states reachable by executing $\mathcal{T}_a$, i.e., $s((\mathcal{T}_a, \Box(\neg \bigcirc true \rightarrow fact)))$. For this, we consider an auxiliary strategy $check(fact)$:

$$check(fact) = match(\mathcal{B}, p) \text{ s.t. } fact \in \mathcal{B}$$

which is successful if $fact$ is in the belief base from the current state. With this strategy we can define $s((\mathcal{T}_a, \Box(\neg \bigcirc true \rightarrow fact)))$ simply as $s(\mathcal{T}_a); check(fact)$. We can further use $check(fact)$ for defining test drivers working with $\neg fact$ as $not(check(fact))$ and with $fact_1 \wedge fact_2$ as $check(fact_1); check(fact_2)$. The cases with respect to the temporal formulae are defined by case analysis. We present only the implementation of the non-trivial ones:

$$s(((a, R); \mathcal{T}_a, \Diamond fact)) = check(fact) \ ? \ s((a, R); \mathcal{T}_a) : s((a, R)) \ ; \ s((\mathcal{T}_a, \Diamond fact))$$
$$s(((a, R); \mathcal{T}_a, \Box fact)) = check(fact) \ ; \ s((a, R)) \ ; \ s((\mathcal{T}_a, \Box fact))$$

which illustrates that the main difference between them is that for $\Diamond fact$ we stop checking $fact$ as soon as we reached a state where $fact$ is in the belief base; from this state we continue with only executing the test on actions. However, for $\Box fact$ we check until the end.

Observe that the semantics of the testing language was defined such that we have a separation between implementing test drivers and *reporting* the results. This is important since running a test driver should be orthogonal to the interpretation and the analysis of the possible output. One plausible and intuitive interpretation is the following one. When the test driver is successful the tester has the confirmation that the test case corresponds to a "good" trace in the agent program. When the test driver fails, the tester can further define new strategies to obtain more information. Consider, as an example, a strategy returning the states previous to the failure. More sophisticated implementations like gathering information about traces instead of states are left to the imagination of the reader. These traces correspond to the shortest counter-examples. This follows from the semantics of the testing language. At each action execution a check is performed whether forbidden actions are possible. If this is the case, then the test fails.

Assuming that we fix an interpretation of the results as above, we proceed by showing that test drivers are partially correct and complete with respect to the definition of test cases.

Definition 1. *Given a test case $\mathcal{T}$ and the corresponding test driver $s(\mathcal{T})$, we say that the application of $s(\mathcal{T})$ is correct, if, on the one hand, successful executions of the test driver are successful applications of the test case, and if, on the other hand, the test driver fails then test case also fails. Similarly, s is complete if (un)successful applications of the test case $\mathcal{T}$ are (un)successful executions of the test driver $s(\mathcal{T})$.*

Before stating the main result, we show two helpful lemmas. Recall that, at each repetition step, the strategy $tauClosure$ tries to apply *sum* and *fail-act* only once. Intuitively, this is sufficient for the following reason. Let us first consider *fail-act*: if, on the one hand, after the application of *fail-act* no action can take place then applying *fail-act* again can do no good, since nothing changed; if, on the other hand, after applying once

fail-act the first action of the new plan can be executed then we are done, the faulty plan has been repaired. From this, we have the following lemma:

Lemma 1. *The strategy try(fail-act) is idempotent, i.e., for any* ms *try(fail-act)*^{2} $@ms$ = *try(fail-act)* $@ms$.

Proof. Let Res = *try*(*fail-act*) $@ms$. Any $ms' \in Res$ different from ms is the result of applying the rewrite rule *fail-act* so it has the form $(\mathcal{B}, p\theta)$, where $\phi \leftarrow p \in \mathcal{R}$ (the set of repair rules) and $\theta \in Sols(\mathcal{B} \models \phi)$. If *fail-act* were again applicable for such ms', the resulting term ms'' is also of the same form since $\mathcal{R}$ is fixed and $\mathcal{B}$ does not change. Thus, any ms'' is already an element of Res and so *try*(*fail-act*) $@Res = Res$. □

An analogous reasoning works also for *sum*. Taking into account that the "+" operator is commutative and associative and that the ";" operator is associative, a *normal form* (i.e., sum of plans with only sequence operators) always exists. Since *sum* is applied to states where the plans are reduced to their normal form we have that states with *basic* plans will always be in the result of trying to apply *sum* more than once.

Lemma 2. *Given a mental state* ms *we have that sum*! $@ms \subseteq$ *try(sum)* $@ms$.

Proof. We only consider the interesting case where *sum* is applicable, that is, when *try*(*sum*) $@ms$ = *sum* $@ms$. Let $ms = (\mathcal{B}, p)$ where p has been reduced to the form $\sum_{i=1}^{n} p_i$ and p_i are basic plans (composed by only the ";" operator). Since *sum* is commutative, we have that *sum* $@ms = \{(B, \sum_{j=1}^{k} p_{i_j}) \mid \forall k, i_j \in \{1, \ldots, n\}\}$, i.e., any possible combination of p_i. On the other hand, *sum*! $@ms = \{(B, p_i) \mid i \in \{1, \ldots, n\}\}$ which is clearly included in *sum* $@ms$. □

Theorem 1 (Partial Correctness & Completeness). *Given* ms *a mental state,* $\mathcal{T}$ *a test case we have that* $s(\mathcal{T})@ms = \mathcal{T}@ms$.

Proof. We consider only the strategy do. The proof for the compositions follows from the definitions of the strategies. We proceed, by showing, as usually, a double inclusion. "$\subseteq$": By the definition of $do(a)$ we have that the result of applying it on ms is:

$$Res = tauClosure\ @\ \underbrace{(\textit{o-act}[\textit{o-a} \leftarrow a]\ @\ \underbrace{tauClosure\ @\ ms}_{Res'})}_{Res''}$$

If the normalisation strategy "!" from the definition of $tauClosure$ terminates, then by definition, there exists an $i \geq 0$ s.t.:

$$Res_i = \textit{i-act}\ @\ (\textit{try}(\textit{fail-act})\ @\ (\textit{try}(\textit{sum})@Res_{i-1}))$$

and for any $ms_i \in Res_i$ we have that *i-act* @ (*try*(*fail-act*) @ (*try*(*sum*) $@ms_i$)) is empty (1). Thus, we can construct the computation:

$$ms_0 \xrightarrow{\tau}{}^{*}\ ms_1 \xrightarrow{\tau}{}^{*}\ \ldots \xrightarrow{\tau}{}^{*}\ ms_{i-1} \xrightarrow{\tau}{}^{*}\ ms_i$$

where we take $ms_j \in Res_j$ with $j \leq i$, ms_0 as ms and * denotes at most 3 τ steps, corresponding to the 3 possible rule labels for τ steps. By the definition of $tauClosure$, Res' is the union of *try*(*fail-act*) @ (*try*(*sum*)@Res_i). This implies that any $ms' \in Res'$ is obtained from a ms_i after eventually applying *sum* and *fail-act*. From (1) we have that from ms' it is not possible to apply *i-act*. Furthermore, by the lemmas, whatever state can be reached from ms' by *sum* and *fail-act* is already in Res'. Thus, $ms \Rightarrow ms'$. By definition, Res'' is empty iff *o-act*[*o-a* $\leftarrow a$] @ms' fails for any element $ms' \in Res'$. That is, if Res'' is empty then $ms \stackrel{a}{\not\Rightarrow} ms'$ and thus $a@ms$ returns the empty set.

If Res'' were not empty, then for any element ms'' contained in it we have that $ms' \stackrel{a}{\rightarrow} ms''$, thus $ms \Rightarrow \stackrel{a}{\rightarrow} ms''$. Similarly, for any element $ms_f \in Res$ we have $ms'' \Rightarrow ms_f$ and from this we can conclude that $ms \stackrel{a}{\Rightarrow} ms_f$, thus ms_f is also an element of $a@ms$.

"$\supseteq$": By the definition of $\Rightarrow$ we have that, if no τ divergence, then there exists a $k \geq 0$ s.t. $ms \stackrel{\tau^k}{\rightarrow} ms_1$ and $ms_1 \not\rightarrow$. The trace τ^k can be divided in m packages of the form:

$$\sigma_m = (\textit{sum}^{i_m}; \textit{fail-act}^{j_m}; \textit{i-act}^{l_m})^m,$$

with $\sum_m (i_m + j_m + l_m) * m = k$. By the lemmas we have that $\textit{sum}^{i_m}; \textit{fail-act}^{j_m}; \textit{i-act}$ is obtained by applying the strategy *try*(*sum*); *try*(*fail-act*); *i-act* (2). As for $\textit{i-act}^{l_m - 1}$, it is obtained by (*try*(*sum*); *try*(*fail-act*); *i-act*)$^{l_m - 1}$ (3). If successive applications of *i-act* are possible then neither *fail-act* nor *sum* is applicable (at most one of *i-act*, *fail-act*, *sum* is enabled at a time) thus trying to applying them is harmless, i.e., does not change the state. Repeating m times the same argument from (2) + (3) and taking into account that we have that sequences σ_m where l_m is 0 are mapped to *try*(*sum*); *try*(*fail-act*) we can derive that $ms_1 \in tauClosure@ms$ (4).

If $ms_1 \stackrel{a}{\rightarrow} ms'$, then $ms' \in$ *o-act*[*o-a* $\leftarrow a$] @ms_1. Applying a similar reasoning for ms' we obtain (4'): $ms_2 \in tauClosure@ms'$. In consequence, we have that if $ms \stackrel{a}{\Rightarrow} ms_2$ then also $ms_2 \in do(a)@ms$.

If $ms_1 \stackrel{a}{\not\rightarrow} ms'$, then *o-act*[*o-a* $\leftarrow a$] @ms_1 fails, thus this is also the case for $do(a)$. □

Observe that in our proof we consider only finite computations. Thus, infinite computations do not violate the result. Since τ divergence is undecidable for BUpL agents, we cannot provide conditions such that test drivers terminate for all test cases. The most we can do, with respect to divergent computations, is to state the following proposition as a consequence of the above result:

Corollary 1 (Divergence). *If the application of $s(\mathcal{T})$ diverges then so does $\mathcal{T}$.*

5 A Running Example

The BUpL builder described in Figure 1 has a small number of states. Thus, verification by model-checking is feasible. We provide now an illustration of the utility of testing. Consider the agent from Figure 3[3]. It is meant to implement the specification "the agent should always construct towers, the order of the blocks is not relevant, however each

[3] The code presents only the constructions which are additional to the ones from Figure 1.

$$
\begin{aligned}
\mathcal{A} = \{\ & incLength(x) = (length(x), \{\ \neg length(x), length(x+1)\ \}),\\
& addBlock(x) = (\neg on(x,0), \{\ on(x,0), clear(x)\ \}),\\
& setMax(x,y) = (max(y), \{\ \neg max(y), max(x)\ \}),\\
& finish(x,y) = (\neg done(x) \wedge done(y), \{\ \neg(done(y)), done(x)\ \})\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P} = \{\ & build(n,c) = move(c-n,0,c-n-1); incLength(c-n-1); build(n-1,c)\\
& generate(x,y) = addBlock(x); generate(x-1,y),\\
& p_0(x,y) = setMax(x,y); generate(x,y))\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R} = \{\ & length(x) \wedge max(y) \wedge (x \leq y) \leftarrow build(y, y+x-1),\\
& length(x) \wedge max(x) \wedge done(y) \wedge (x \geq y) \leftarrow finish(x,y); \bot,\\
& max(x) \wedge done(x) \leftarrow setMax(x+2,x), generate(x+2,x)\ \}
\end{aligned}
$$

Fig. 3. A BUpL Builder with Infinite State Space

tower should use more blocks than the previous, and additionally, the length of the towers must be an even number"[4] (for example, 21, 4321 are "well-formed" towers).

The agent is designed such that it always builds a higher tower. The example can be understood as a typical agent with *maintenance* goals. Since the number of its mental states continuously increases, instead of model-checking, we test it. For illustration purposes, the implementation of the agent is on purpose faulty: assuming a correct initialisation, the agent program does not perform a sanity check with respect to the parity of X before adding the fact $done(X)$ to signal that it constructed a tower X.

Thanks to the fact that the strategy language S has been incorporated into the Maude system, it was relatively easy to extend the implementation from [1]. In this way, we provide a testing framework as alternative to the model-checking facility. We have experimented with different test cases which we applied to the Maude prototype of the BUpL builder. For example, we have considered the test whether $done(2)$ appears in the belief base after executing $move(2,0,1)$. To implement it, we only needed to apply the strategy $do(move(2,0,1)); check(done(2))$. The application of the strategy failed, meaning that the agent is not conformant with the test case. On the contrary, the application succeeded when the correct agent program is tested. We have run our tests on a Fedora 10 system (Kernel linux 2.6.27.12-170.2.5.fc10.x86_64) with an AMD Athlon(tm) 64 Processor 3500+ and 1 GB memory. The process of executing the BUpL builder with respect to the test case $do(move(2,0,1)); check(done(2))$ took 1876ms and generated 35745 rewrites. The number of rewrites is high mainly because the strategy language is implemented at the meta-level and because computations at the meta-level involve many rewrite steps. For the correct agent, the output generated by Maude illustrates that the strategy has succeeded and that the resulting state reflects that $done(2)$ has been updated to the belief base and that the current tower is 21. By means of the Maude command `next` we can further see that there are no more solutions (corresponding to faulty executions). More examples and the actual Maude code (also including more test case implementations) can be downloaded from our website `http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php`

[4] Since it is just meant to be an illustration, the notion of specification is merely informal.

6 Conclusions and Future Work

In this paper, we focused on two aspects. First, we have provided a formalisation for testing BUpL agents. Second, we have introduced rewrite strategies to define test drivers that implement test cases. For simplicity, we have considered testing individual agents. Generalising our current results to multi-agent systems should be easy in a *particular framework* as the one proposed in [2]. There, the interaction between agents is achieved not by means of communication but by *action-based coordination* mechanisms. The advantage of this approach is that the framework is compositional and thanks to this, the verification problem (by model-checking) of the whole system can be reduced to the verification of individual agents. Using the compositionality result we can obtain the same reduction when we consider *testing* instead of *model-checking*.

References

1. Astefanoaei, L., de Boer, F.S.: Model-checking agent refinement. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 705–712. IFAAMAS (2008)
2. Astefanoaei, L., de Boer, F.S., Dastani, M.: The refinement of choreographed multi-agent systems. In: Baldoni, M., van Riemsdijk, M.B. (eds.) DALT 2009. LNCS, vol. 5948, pp. 20–34. Springer, Heidelberg (2010)
3. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Programming: Languages, Platforms and Applications (Multiagent Systems, Artificial Societies, and Simulated Organizations), vol. 15. Springer, Heidelberg (2005)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. Electronic Notes in Theoretical Computer Science (ENTCS) 174(11), 3–25 (2007)
6. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proceedings of the 4th Workshop on Rewriting Logic and its Applications (WRLA). ENTCS, vol. 71. Elsevier, Amsterdam (2002)
7. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems (AAMAS) 2(4), 357–401 (1999)
8. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Meseguer, J. (ed.) Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier, Amsterdam (2000)
9. Meyer, B.: Seven Principles of Software Testing. IEEE Computer 41(8), 99–101 (2008)
10. Nguyen, D.C., Perini, A., Tonella, P.: A Goal-Oriented Software Testing Methodology. In: Agent Oriented Software Engineering (AOSE), pp. 58–72 (2007)
11. Serbanuta, T.-F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics (extended abstract). Electronic Notes in Theoretical Computer Science (ENTCS) 192(1), 125–141 (2007)
12. Wooldridge, M.: Agent-based software engineering. IEEE Proceedings Software Engineering 144(1), 26–37 (1997)
13. Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing intelligent agents in PDT. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1673–1674. IFAAMAS (2008)

Towards Just-In-Time Partial Evaluation of Prolog

Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
cfbolz@gmx.de, leuschel@cs.uni-duesseldorf.de, arigo@tunes.org

Abstract. We introduce a just-in-time specializer for Prolog. Just-in-time specialization attempts to unify of the concepts and benefits of partial evaluation (PE) and just-in-time (JIT) compilation. It is a variant of PE that occurs purely at runtime, which lazily generates residual code and is constantly driven by runtime feedback.

Our prototype is an on-line just-in-time partial evaluator. A major focus of our work is to remove the overhead incurred when executing an interpreter written in Prolog. It improves over classical offline PE by requiring almost no heuristics nor hints from the author of the interpreter; it also avoids most termination issues due to interleaving execution and specialization. We evaluate the performance of our prototype on a small number of benchmarks.

1 Introduction

Just-in-time compilers have been hugely successful in recent years, often providing significant benefits over traditional (ahead-of-time) compilers.[1] Indeed, much more information is available at runtime, some of which can be very expensive or impossible to obtain ahead-of-time by traditional static analysis. The biggest success story is possibly the Java HotSpot [23] just-in-time compiler, which now often matches or beats classical C++ compilers in terms of speed.

Dynamic languages have seen a recent surge in activity and industrial applications. Dynamic languages, due to their very nature, make traditional static analysis and compilation nigh impossible. Hence, a lot of hope is put into just-in-time compilation. Many techniques have been proposed; one of the main recent successes is the Psyco just-in-time specializer [25] for Python. In the best cases it can remove all the overhead incurred by the dynamic nature of the language. Its successor, the JIT compiler generator developed in the PyPy framework [27], is one of the bases for the present work, where we are interested in applying similar techniques to Prolog in general and partial evaluation of Prolog programs in particular.

[1] Even though there is of course room for both. Some applications do require static compilation techniques and validation, in the form of static analysis or type checking, which provides benefits over runtime validation.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 158–172, 2010.

Partial evaluation [17] is a technology that has been very popular for improving the performance of Prolog programs. Indeed, for Prolog, partial evaluation is more tractable than for imperative or object-oriented languages, such as C or Python. Especially for interpreters (one of the typical Prolog applications), speedups of several orders of magnitude are possible [2]. However, while some isolated successul applications exist, there is no widespread usage of partial evaluation technology. One problem is that the static input needs to be known ahead of time, whereas quite often the input that enables optimisations is only available at runtime. Also, one faces problems such as code explosion, as the specialized program sometimes needs to anticipate all possible runtime combinations in order not to loose static information. We argue that these problems can be solved by incorporating and adapting ideas from just-in-time compilation.

In this paper we present the technique of just-in-time partial evaluation along with an first prototype implementation for Prolog. The key contributions of our work are:

1. Just-in-time specialization allows us to decide which information is relevant for good optimisation; we can decide at runtime what is static and dynamic.
2. The specializer can inspect a runtime value at any point in time, and use it as a static value in order to partially evaluate the code that follows. We call this concept *promotion*.
3. Partial evaluation is done lazily; only parts really required are specialized, and compilation and execution are tightly interleaved.

Our paper is structured as follows. We discuss the problems that trouble classical partial evaluation in more detail in Sect. 2. The main mechanism of just-in-time partial evaluation is explained in Sect. 3. These goals are achieved with the use of "lazy choice points", which are the basic concept of this work. The control of our partial evaluator is discussed in Sect. 4. In Sect. 5 we examine the behaviour of our specializer for some examples. Related work and conclusion are presented in Sect. 6 and 7 respectively.

2 Problems of Classical Partial Evaluation

Partial evaluation [17] is a well-known source-to-source program transformation technique. It specialises programs by pre-computing those parts of the program which depend only on statically known input. The so-obtained transformed programs are less general than the original but can be much more efficient. In the context of logic programming, partial evaluation proceeds mostly by unfolding [20,18] and is sometimes referred to as *partial deduction*.

Partial evaluation has a number of problems that have prevented it from being widely used, despite its considerable promise. One of the hardest problems of partial evaluation is the balance between under- and over-specialization. Over-specialization occurs when the partial evaluator generates code that is too specialized. This usually leads to too much code being generated and can lead

to "code explosion", where a huge amount of code is generated, without significantly improving the speed of the code.

The opposite effect is that of under-specialization. When it occurs, the residual code is too general. This happens either if the partial evaluator does not have enough static information to make better code, or if the partial evaluator erroneously decides that some of the information it has is actually not useful and it then discards it.

The partial evaluator has to face difficult choices between over- and under-specialization. To prevent under-specialization it must keep as much information as possible, since once some information is lost, it cannot be regained. However, keeping too much information is also not desirable, since it can lead to too much residual code being produced, without producing any real benefit.

Figure 1 shows an example where ECCE (a partial evaluator for pure Prolog [19]) produces bad code when doing partial evaluation. The code in the figure is a simple Prolog meta-interpreter which stores the outstanding goals in a list (the point of the `jit_merge_point` predicate is explained in Sect. 4.2. ECCE just ignores it). The interpreter works on object-level representations of append, naive reverse and a predicate replacing the leaves of a tree. When ECCE is asked to residualize a call to the meta-interpreter interpreting the `replaceleaves` predicate, it loses the information that the list of goals can only consist of `replaceleaves` terms. Thus eventually the residual code must be able to deal with arbitrary goals in the list of goals, which causes the full original program to be included in the residual code that ECCE produces (see predicates `solve__5`, `my_clause__6` and `append__7` in the residual code). This is a case of under-specialization (the code could be more specific and thus faster) and also of code explosion (the full interpreter is contained again, not only the parts that are needed for `replaceleaves`). We will come back to this example in Section 5.

A related problem is Prolog builtins. Many Prolog partial evaluators do not handle Prolog builtins very well. For example ECCE only supports purely logical builtins (which are builtins which could in theory be implemented by writing down a potentially infinite set of facts). Some builtins are just hard to support in principle, e.g., a partial evaluator cannot assume anything about the result of `read(X)`.

The fact that many classical Prolog partial evaluators do not support builtins, means that quite often user programs have to be rewritten in non-trivial ways – a time-consuming task.

3 Basics of Just-In-Time Specialization

3.1 Basic Setting

We propose to solve the problems described in the previous section by *just-in-time partial evaluation*. The basic idea is that the partial evaluator is executed at runtime rather than ahead of time, interleaved with the execution of the specialized code. This allows it to observe the runtime behaviour of the program,

Original code:

```
solve([]).
solve([A|T]) :-
    jit_merge_point,
    my_clause(A,B), append(B,T,C), solve(C).

append([], T, T).
append([H|T1], T2, [H|T3]) :-
    append(T1, T2, T3).

my_clause(app([],L,L),[]).
my_clause(app([H|X],Y,[H|Z]),[app(X,Y,Z)]).
my_clause(replaceleaves(leaf, NewLeaf, NewLeaf),[]).
my_clause(replaceleaves(node(Left, Right), NewLeaf,
                        node(NewLeft, NewRight)),
          [replaceleaves(Left, NewLeaf, NewLeft),
           replaceleaves(Right, NewLeaf, NewRight)]).
my_clause(nrev([],[]), []).
my_clause(nrev([H|T], Z), [nrev(T, T1), app(T1, [H], Z)]).
```

Residual code for `solve([replaceleaves(A, B, C)])` by Ecce :

```
solve([replaceleaves(A, B, C)]) :- solve__2(A, B, C).
solve__2(leaf,A,A).
solve__2(node(A,B),C,node(D,E)) :- solve__3(A,C,D,B,E,[]).
solve__3(leaf,A,A,B,C,D) :- solve__4(B,A,C,D).
solve__3(node(A,B),C,node(D,E),F,G,H) :-
    solve__3(A,C,D,B,E,[replaceleaves(F,C,G)|H]).
solve__4(leaf,A,A,B) :- solve__5(B).
solve__4(node(A,B),C,node(D,E),F) :- solve__3(A,C,D,B,E,F).

solve__5([]).
solve__5([A|B]) :-
    my_clause__6(A,C),
    append__7(C,B,D),
    solve__5(D).
my_clause__6(app([],A,A),[]).
my_clause__6(app([A|B],C,[A|D]),[app(B,C,D)]).
my_clause__6(replaceleaves(leaf,A,A),[]).
my_clause__6(replaceleaves(node(A,B),C,node(D,E)),
             [replaceleaves(A,C,D),replaceleaves(B,C,E)]).
my_clause__6(nrev([],[]),[]).
my_clause__6(nrev([A|B],C),[nrev(B,D),app(D,[A],C)]).
append__7([],A,A).
append__7([A|B],C,[A|D]) :-
    append__7(B,C,D).
```

Fig. 1. Under-Specialization in Ecce for a Meta-Interpreter

giving it more information than a static specializer to base its decisions on. The approach we take is that the specializer produces some residual code upon demand, uses `assert` to put it into the Prolog database and then immediately runs the asserted code.[2] More residual code is produced later, if that becomes necessary. The details of when this process is started and stopped are described below.

The specialization process itself proceeds by interpretation of the Prolog source code. If a deterministic call to a user-predicate is interpreted, it is unfolded; otherwise specialization stops as described in the following section. If a call to a builtin is encountered, in the general case the call is skipped, i.e. put in the residual code; but a number of common builtins have corresponding custom specialization rules and produce specialized residual code (or no code at all).

3.2 Promotion: Lazy Choice Points

The fundamental building block for the partial evaluator to make use of the just-in-time setting are *lazy choice points*. When reaching a choice point in the original program, the partial evaluator does not know which choice would be taken at runtime. Compiling all cases is undesirable, since that can lead to code explosion. Therefore it inserts a *callback* to the specializer into the residual code and stops the partial evaluation to let the residual code run. When the callback is reached, the specializer is invoked again and specializes exactly the switch case that is needed by the running code. After specialization has finished, this new code is generated.

Another usage of lazy choice points by the partial evaluator is to get information about terms which are required to obtain good specialization but are not available statically. When the actual runtime value (or some partial info about the value, like the functor and arity) of an unknown term is needed by the partial evaluator during specialization, specialization stops and a callback is inserted. Then the residual code generated so far is executed until the callback point is reached. When this happens, the value of the formerly unknown term *is* available (there are no unknown terms at runtime of course). At this point the specializer is invoked with the now known term and more code can be produced. We call this process *promotion*: it promotes a dynamic, unknown value to a static value available to the specializer.

Our approach is best illustrated by an example. Assume we have the following predicate:

```
negation(true(A), false(A)).
negation(false(A), true(A)).
```

First, our specializer rewrites this predicate in a pre-processing phase into the following form, which makes the choice point and first-argument indexing visible:

[2] On some Prolog systems, dynamically asserted code runs slower than static code. We can sometimes use workarounds, like `compile_predicates` in SWI-Prolog.

```
negation(X, Y) :- switch_functor(X, [
                    case(true/1,  (X = true(Z), Y = false(Z))),
                    case(false/1, (X = false(Z), Y = true(Z)))]).
```

The predicate `switch_functor` performs a switch on the functor of its first argument, the possible cases are described by the second argument.[3] It could be implemented as a Prolog-predicate like this:

```
switch_functor(X, [case(F/Arity, Body)|_]) :-
    functor(X, F, Arity), call(Body).
switch_functor(X, [_|MoreCases]) :-  switch_functor(X, MoreCases).
```

If the specializer encounters the call `negation(X, Y)` it cannot know whether the functor of `X` will be `true` or `false` (if it would know the functor of `X` it could continue unfolding with the correct case immediately). Therefore the specialization process stops. At this point the following code has been generated and put into the clause database:

```
'$negation1'(X, Y) :- '$case1(X),  '$promotion1'(X, Y).
'$case1(true(_)).
'$case1(false(_)).
'$promotion1'(X, Y) :-  functor(X, F, N),
    callback_pe(F/N, '$promotion1', ...),  '$promotion1'(X, Y).
```

The predicate `'$negation1'` is the entry-point of the specialized version of `negation`. The `'$case1'` predicate ensures that `X` is bound when `'$promotion1'` is called and that solutions are generated in the right order. The `'$promotion1'` predicate is the lazy choice point. At this point this predicate has only one clause, which is for invoking the specializer again. More clauses will be added later. If it is executed, partial evaluation will be resumed by calling `callback_pe`, passing in the functor and the arity of the argument as information for specializing more code. Thus, one concrete clause of the choice point will be generated. After this is done, the promotion predicate is called again, which will execute the newly generated case.

The `callback_pe` gets the functor and arity as its first argument. The second argument is the name of the predicate that should get a new clause added. The further arguments (shown only as `...` in the code above) contain the `Cases` in the `switch_functor` call, the continuation of what the partial evaluator still has to evaluate after the choice point. When `callback_pe` is called, it will use its first argument to decide which of the cases it should partially evaluate further.

Let us assume that `'$negation1'` is first called with `false(A)` as an argument. Then `'$promotion1'` will be executed, calling `callback_pe(false/1, '$promotion1', ...)`. This will resume the partial evaluator which then generates residual code only for the case where `X` is of the form `false(_)`. The residual code looks as follows:

[3] Calls to `switch_functor` are generated at specialisation time, i.e., not by a static mode analysis. The transformation takes into account the actual call pattern, and will generate different versions of a predicate for different call patterns.

```
'$promotion1'(false(Z), Y) :-  !, Y = true(Z).
```

This code will be asserted using `asserta`, which means that it will be tried before the clause of `'$promotion1'` shown above. This has the effect that the next time `'$negation1'` is called with `false(X)` as an argument, this code will be used and no specialization will be performed. The cut is necessary to prevent the backtracking into the clause calling back into the specializer.

If the `'$negation1'` predicate is never actually called with an argument of the form `true(X)`, then the other case of the switch will never be specialized, saving time and memory. This might not matter for such a trivial case as the one above, but it strongly reduces specialization time and size of the residual code for more realistic cases (e.g. consider what happens if the body of `negation` contains calls to many predicates). If the other case will be specialized eventually, the residual code would look like this:

```
'$promotion1'(true(Z), Y) :- !, Y = false(Z).
```

This code will again be inserted into the database using `asserta` so that it too will be tried before the specialization case.

3.3 Other Uses of Lazy Switches

The `switch_functor` primitive has some other uses apart from the obvious ones that it was designed for. These other uses also exploit the laziness of `switch_functor`, less so the switching part. One of them is to implement a lazy version of disjunction (the "`;`" builtin). In this form it is also used if several clauses of a predicate are not mutually exclusive.

Another use of `switch_functor` is to support the `call(X)` builtin (which very few partial evaluators for Prolog do efficiently). This can be considered to be a switch of `X` over all the predicates in the program. Since `switch_functor` is lazy, only those predicates that are actually called at runtime need to be specialized. An example for this can be found in Sect. 4.3.

4 Control and Ensuring Termination

4.1 Code Generation and Local Control

So far we have not explained exactly how we generate the specialized code (apart from the lazy switches). Basically, we use the well-known partial evaluation framework as presented in [18] (which builds upon the original work in [20]). The control of partial evaluation for logic programs is often separated into local and global control [22], where the global control decides which calls are specialized and the local control performs the unfolding of those calls. In the simple setting described so far, we can simply view the local control of our just-in-time specializer as performing unfolding until a choice point is reached. At this point, the specializer stops and generates a resultant clause with a callback into

the specializer (as explained in the last section). More precisely, the unfolding rule will recursively process the leftmost literal in a goal that has not yet been examined, with the following options:

1. If it is a `switch_functor` which is sufficiently instantiated, the proper case will be chosen.
2. If it is a `switch_functor` which is *not* sufficiently instantiated, unfolding stops and a call back into the specializer is inserted into the resultant, using a lazy switch, as explained in the previous section.
3. If it is a builtin, then the builtin is specialized, yielding a single computed answer along with a specialized version of the builtin to be put into the residual code. For non-deterministic builtins, the computed answer is general enough to cover all solutions. Failure can also sometimes be detected, in which case the branch is pruned [24].
4. If the leftmost literal is a user-predicate, it will be simply unfolded. Observe that this is deterministic, as all choice points are encoded via the `switch_functor` primitive.

To ensure that the semantics are preserved in the presence of impure builtins or predicates, we do not always left-propagate bindings (in case we do not select the leftmost literal). Bindings are left-propagated only until impure builtins are met, using techniques from [24].

As our just-in-time specializer interleaves ordinary execution with code generation, the overall procedure cannot always terminate (namely when the user query under consideration does not terminate). However, we would like to ensure that if the unspecialized program itself terminates (existentially or universally respectively) then the just-in-time specializer process should also terminate (existentially or universally respectively). The above process does not fully guarantee this, as our just-in-time specializer may not detect that a call to a builtin in point 3 actually fails. This means that the just-in-time specializer would proceed specialization on a computation path which does not occur at runtime, which is a problem if this path is infinite.

One pragmatic solution is to ensure that the just-in-time specializer will maximally perform N specialization steps before executing residual code again. Every time the residual code is executed, the computation progresses. Therefore the presence of the just-in-time specializer can only lead to a linear slowdown, which means in particular that it preserves termination behaviour.

4.2 Global Control

In some cases the specialization technique described so far can be sufficient. However, it does not reuse any of the generated residual code (i.e., the specializer produces a tree of predicates); what we want is to eventually obtain a jump to an already-specialized predicate, typically closing a loop. Instead of a tree, the final result should be an arbitrary graph of residual predicates.

In the current prototype, the specializer never tries to reuse existing residual code on its own. To trigger global control, the specialized program needs to request the attempt to reuse existing residual code by inserting a call to a special

predicate called `jit_merge_point`. This predicate does nothing if executed normally, but is dealt with by the partial evaluator in a special way. For an example usage, see Figure 1.

The need for this sort of explicit hint is clearly not ideal, but we felt that it simplified implementation enough to still be a good choice, given that most programs with an interpretative nature need to contain only one call or a small number of calls to this predicate. We plan to find ways of automatically placing this call in the future.

At the places where a call to `jit_merge_point` is seen, the partial evaluator tries to reuse an already existing residual predicate. It does this by comparing the list of goals that the partial evaluator currently has with those it had at earlier calls to `jit_merge_point`. If two such lists of goals are similar enough the partial evaluator inserts a call to the residual predicate produced earlier and stops the partial evaluation process. The exact conditions when this is possible are outside the scope of this paper and are fully explained in [3]. In summary, the procedure remembers which parts of the term have been used to resolve choice points; parts which did not contribute in any way to improve the specialisation are thrown away.[4] The fact that partial evaluation happens at runtime allows us to discard information more aggressively because it is possible to regain information at a later point with the help of lazy choice points, if this becomes necessary.

In the next subsection we present a simple example which illustrates this aspect of our system, and also highlights the potential of our just-in-time specialization compared to traditional partial evaluation.

4.3 A Worked Out Example: Read-Eval-Print Loop

As a showcase example we wrote a minimal read-eval-print loop for Prolog, which can be seen in Fig. 2. Most classical partial evaluators have a hard-time producing good code for `read_eval_print_loop`, because after `read(X)` the value of `X` is unknown, which makes it impossible to figure out which predicate `call(X)` will ultimately call.

For our prototype this represents no real problem. The functor of `X` can be promoted, thus observing at runtime which predicate is to be called. Subsequently, this predicate can be specialized. Fig. 2 also shows an example session as well as the residual code that our prototype generated for this session (note that the clauses for `'$callpromotion1'` are shown in the order in which they are in the database, which is the reverse order in which they have been generated).

5 Experimental Results

To get some idea about the performance of our dynamic partial evaluation system, we ran a number of benchmarks. We compared the results with those of

[4] In some sense this can be seen as an evolution of the generalisation operator from [13] to a just-in-time specialisation setting.

Code of the read-eval-print loop and some example predicates:

```
read_eval_print_loop :- jit_merge_point,
    read(X), call(X), print(X), nl, read_eval_print_loop.

% example predicates
f(a). f(b). f(c).
g(X) :- h(Y, X), f(Y).
h(c, d).
k(_, _, _) :- g(X), g(X).
```

Example session:

```
|: f(c).
f(c)
|: g(X).
g(d)
|: fail.
No
```

Produced residual code (promotion specialization cases not shown):

```
'$entrypoint1' :-  read(A), '$callpromotion1'(A).
'$callpromotion1'(fail) :- !,  fail.
'$callpromotion1'(g(A)) :- !, A=d,print(g(d)), nl,  '$entrypoint1'.
'$callpromotion1'(f(A)) :- !, '$case1'(A), '$promotion1'(A).
'$case1'(a). '$case1'(b). '$case1'(c).
'$promotion1'(c) :- !, print(f(c)), nl,  '$entrypoint1'.
```

Fig. 2. A Simple read-eval-print-loop for Prolog

Ecce [19], an automatic online program specializer for pure Prolog. The experiments were run on a machine with a 1.4 GHz Pentium M processor and 1GiB RAM, using Linux 2.6.24. For running our prototype and the original and specialized programs we used SWI-Prolog Version 5.6.47 (Multi-threaded, 32 bits). Ecce was used both in "classic mode" which uses normal partial evaluation and in "conjunctive mode" (which uses conjunctive partial deduction with characteristic trees and homeomorphic embedding; see [10]). Conjunctive partial evaluation is considerably more powerful, but also much more complex.

Figure 3 presents five benchmarks. The first three are examples for a typical logic programming interpreter with one and also with two levels of interpretation. The fourth example is a higher-order example, using the meta-predicates `=..` and `call`. Finally, the fifth is a small interpreter for a dynamic language. Note that "spec" refers to the specialization time and "run" to the runtime of the specialized code. The second number for the just-in-time partial evaluator is derived by running the same goal a second time, which will not trigger more partial evaluation. For Ecce the specialization time was not measured.

	Experiment	Inferences	CPU Time	Speedup
A vanilla meta-interpreter [15, 21] running append with a list of 100000 elements. The interpreter can be seen in Figure 1.	Vanilla - Append			
	original	500008	0.35 s	1.0
	JIT PE, spec+run	281842	0.13 s	2.69
	JIT PE, run	200016	0.11 s	3.18
	ecce classic	100003	0.03 s	11.67
	ecce conjunctive	100003	0.03 s	11.67
The vanilla interpreter running itself running append with a list of 100000 elements.	Vanilla - Vanilla - Append			
	original	2000023	1.42 s	1.0
	JIT PE, spec+run	1577228	0.66 s	2.15
	JIT PE, run	700020	0.32 s	4.44
	ecce classic	100003	0.04 s	35.5
	ecce conjunctive	100003	0.04 s	35.5
The vanilla interpreter running `replaceleaves`, see Figure 1. Input was a full tree of depth 18.	Vanilla - Replace Leaves			
	original	2621438	2.76 s	1.0
	JIT PE, spec+run	2493636	1.77 s	1.56
	JIT PE, run	2097162	1.58 s	1.75
	ecce classic	2097074	2.64 s	1.05
	ecce conjunctive	589825	0.78 s	3.54
A higher order example: `reduce` in Prolog using `=..` and `call`. This is summing a list of 100000 integers, knowing statically the functor that is used for the summation.	Reduce - Add			
	original	1492586	16.73 s	1.0
	JIT PE, spec+run	5082861	3.53 s	4.74
	JIT PE, run	5000014	3.24 s	5.16
	ecce classic	1134504	8.5 s	1.97
	ecce conjunctive	2000001	1.85 s	9.04
An interpreter (∼100 lines of Prolog) for a small stack-based dynamic language. The benchmark is running an empty loop of 100000 iterations.	Stack Interpreter			
	original	2100010	3.13 s	1.0
	JIT PE, spec+run	5699992	1.46 s	2.14
	JIT PE, run	200019	0.08 s	39.13
	ecce classic	100003	0.05 s	62.6
	ecce conjunctive	100003	0.04 s	78.25

Fig. 3. Experimental Results

Our prototype is in all cases faster than the original code, but also in all cases slower (by a factor between 2 and 8) than Ecce in conjunctive mode. On the other hand, our prototype is faster than Ecce in classical mode in two cases. These are not bad results, considering the relative complexity of the two projects. Our prototype is rather straightforward. It was written from scratch over the course of some months and consists of about 1500 lines of Prolog code. On the other hand, Ecce is a mature system that employs serious theoretical machinery and consists of about 25000 lines of Prolog code.

As we have also seen in Section 2 the third benchmark is one where Ecce in classical mode produces rather bad code. This can be seen in the benchmark results as well, there is nearly no speedup when compared to the original code. Our prototype has the same problem, it also loses the information that all the goals in the goal list are `replaceleaves` calls. However, in our case this is not a problem, since that information can be regained with a promotion, thus preventing code explosion and under-specialization.

The examples above are chosen such that the loops involved run for a large number of iterations, thus making the overhead of just-in-time partial evaluation worthwhile. If the loop is running only a small amount of times, the overhead of partial evaluation can sometimes not be regained. This can be solved with the help of a hybrid interpreter/partial evaluation system, that first interprets everything and does profiling to find commonly executed parts of the system, and then partially evaluates only those.

6 More Related Work

Promotion is a concept that we have already explored in other contexts. Psyco is a run-time specializer for Python that uses promotion (called "unlift" in [25]). Similarly, the PyPy project [26,5,4], in which all three authors are also involved, contains a just-in-time specialization system built on promotion [27].

Greg Sullivan describes a runtime partial evaluator for a small dynamic language based on lambda calculus [28]. Sullivan [28] further distinguishes two cases (quoting): *"Runtime partial evaluation [...] defers some of the partial evaluation process until actual data is available at runtime. However the scope and actions related to partial evaluation are largely decided at compile time. Dynamic partial evaluation goes further, deferring all partial evaluation activity to runtime."* Using this terminology, our system does dynamic partial evaluation.

One of the earliest works on runtime specialization is Tempo for C [9,8]. However, it is essentially an offline specializer "packaged as a library"; decisions about what can be specialized and how are pre-determined.

Another work in this direction is DyC [14], another runtime specializer for C. Specialization decisions are also pre-determined, i.e. dynamic partial evaluation is not attempted, but "polyvariant program-point specialization" gives a coarse-grained equivalent of our promotion. Targeting the C language makes higher-level specialization difficult, though (e.g. `malloc` is not optimized).

Polymorphic inline caches (PIC) [16] are very closely related to promotion. They are used by JIT compilers of object-oriented language and also insert a growable switch directly into the generated machine code. This switch examines the receiver types for a message for a particular call site. From that angle, promotion is an extension of PICs, since promotions can be used to switch on arbitrary values, not just receiver types.

The recent work on trace-based JITs [12] (originating from Dynamo [1]) shares many characteristics of our work. Trace-based JITs concentrate on generating good code for loops, and generate code by observing the runtime behaviour of the user program. They also only generate code for code paths that are actually followed by the program at runtime. The generated code typically contains guards; in recent research [11] on Java, these guards' behavior is extended to be similar to our promotion. This has been used by several implementations to implement a dynamic language (JavaScript) [6,7].

7 Conclusion and Future Work

In this paper we drew explicit parallels between partial evaluation and just-in-time compilers. We showed with a Prolog prototype of a just-in-time partial evaluator that these two domains might benefit a lot from a synergy. In particular, inspired by Polymorphic Inline Caches, we have developed the notion of *promotion* for partial evaluation. We hope that our approach can help address several fundamental issues that so far prevent classical partial evaluation to reach its fullest potential: code explosion, termination, full Prolog support, and scalability to large programs.

Due to the use of promotion our just-in-time partial evaluator works reasonably well for interpreters of dynamic languages and generally in situations where information that the partial evaluator needs is only available at runtime. This is an advantage that a classical partial evaluator can never possess for fundamental reasons. We have not tried our prototype on really large programs yet, so it remains to be seen whether it works well for these.

There are some downsides to our approach. In particular promotion needs a Prolog system that supports `assert` well, since the whole approach depends on that in a crucial manner. We have not yet evaluated our work on any Prolog system other than SWI-Prolog (which supports `assert` rather well). In the future we would like to support other Prolog platforms like Ciao Prolog or Sicstus Prolog as well.

Global control is another area that still needs further work. We plan to explore ways of inserting the `jit_merge_points` automatically. Furthermore, the global control strategy needs further evaluation and possible refinement.

Finally we need to take a look at the speed of the partial evaluator itself, which we so far disregarded completely. Since partial evaluation happens at runtime it is necessary for the partial evaluator to not have too bad performance. Also we plan to incorporate profiling techniques to only apply the partial evaluator to those parts of the program that are executed often.

References

1. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. ACM SIGPLAN Notices 35, 1–12 (2000)
2. Barker, S., Leuschel, M., Varea, M.: Efficient and flexible access control via logic program specialisation. In: Proceedings PEPM 2004, pp. 190–199. ACM Press, New York (2004)
3. Bolz, C.F.: Automatic JIT Compiler Generation with Runtime Partial Evaluation. Master thesis, Heinrich-Heine-Universität Düsseldorf (2008), http://www.stups.uni-duesseldorf.de/thesis_detail.php?id=14
4. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler. In: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, Genova, Italy, pp. 18–25. ACM, New York (2009)
5. Bolz, C.F., Rigo, A.: How to *not* write a virtual machine. In: Proceedings of 3rd Workshop on Dynamic Languages and Applications, DYLA 2007 (2007)

6. Chang, M., Bebenita, M., Yermolovich, A., Gal, A., Franz, M.: Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine (2007)
7. Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., Franz, M.: Tracing for Web 3.0: Trace compilation for the next generation web applications. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Washington, DC, USA, pp. 71–80. ACM Press, New York (2009)
8. Consel, C., Hornof, L., Noël, F., Noyé, J., Volansche, N.: A uniform approach for compile-time and run-time specialization. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 54–72. Springer, Heidelberg (1996)
9. Consel, C., Noël, F.: A general approach for run-time specialization and its application to C. In: POPL, pp. 145–156 (1996)
10. De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., Sørensen, M.H.: Conjunctive partial deduction: Foundations, control, algorithms and experiments. The Journal of Logic Programming 41(2,3), 231–277 (1999)
11. Gal, A., Franz, M.: Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine (November 2006)
12. Gal, A., Probst, C.W., Franz, M.: HotpathVM: an effective JIT compiler for resource-constrained devices. In: Proceedings of the 2nd international conference on Virtual execution environments, Ottawa, Ontario, Canada, pp. 144–153. ACM, New York (2006)
13. Gallagher, J., Bruynooghe, M.: The derivation of an algorithm for program specialisation. New Generation Computing 9(3,4), 305–333 (1991)
14. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: an expressive annotation-directed dynamic compiler for C. Theoretical Computer Science 248, 147–199 (2000)
15. Hill, P., Gallagher, J.: Meta-programming in logic programming. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 421–497. Oxford Science Publications, Oxford University Press (1998)
16. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 21–38. Springer, Heidelberg (1991)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
18. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: Control issues. Theory and Practice of Logic Programming 2(4,5), 461–515 (2002)
19. Leuschel, M., Martens, B., De Schreye, D.: Controlling generalisation and polyvariance in partial deduction of normal logic programs. ACM Transactions on Programming Languages and Systems 20(1), 208–258 (1998)
20. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming 11(3,4), 217–242 (1991)
21. Martens, B., De Schreye, D.: Two semantics for definite meta-programs, using the non-ground representation. In: Apt, K.R., Turini, F. (eds.) Meta-logics and Logic Programming, pp. 57–82. MIT Press, Cambridge (1995)

22. Martens, B., Gallagher, J.: Ensuring global termination of partial deduction while allowing flexible polyvariance. In: Sterling, L. (ed.) Proceedings ICLP 1995, Kanagawa, Japan, June 1995, pp. 597–613. MIT Press, Cambridge (1995)
23. Paleczny, M., Vick, C., Click, C.: The Java HotSpot server compiler. In: Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium, Monterey, California, vol. 1. USENIX Association (2001)
24. Prestwich, S.: An unfold rule for full Prolog. In: Lau, K.-K., Clement, T. (eds.) Logic Program Synthesis and Transformation. Proceedings of LOPSTR 1992, Workshops in Computing. University of Manchester, pp. 199–213. Springer, Heidelberg (1992)
25. Rigo, A.: Representation-based just-in-time specialization and the Psyco prototype for Python. In: Heintze, N., Sestoft, P. (eds.) PEPM, pp. 15–26. ACM Press, New York (2004)
26. Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, pp. 944–953. ACM Press, New York (2006)
27. Rigo, A., Pedroni, S.: JIT compiler architecture. Technical Report D08.2, PyPy Consortium (2007), http://codespeak.net/pypy/dist/pypy/doc/index-report.html
28. Sullivan, G.T.: Dynamic partial evaluation. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 238–256. Springer, Heidelberg (2001)

Program Parallelization Using Synchronized Pipelining

Leonardo Scandolo[1], César Kunz[1], and Manuel Hermenegildo[1,2]

[1] IMDEA Software
[2] Technical U. of Madrid, Spain
Firstname.Lastname@imdea.org

Abstract. While there are well-understood methods for detecting loops whose iterations are independent and parallelizing them, there are comparatively fewer proposals that support parallel execution of a sequence of loops or nested loops in the case where such loops have dependencies among them. This paper introduces a refined notion of independence, called *eventual independence*, that in its simplest form considers two loops, say loop_1 and loop_2, and captures the idea that for every i there exists k such that the $i+1$-th iteration of loop_2 is independent from the j-th iteration of loop_1, for all $j \geq k$. Eventual independence provides the foundation of a semantics-preserving program transformation, called *synchronized pipelining*, that makes execution of consecutive or nested loops parallel, relying on a minimal number of synchronization events to ensure semantics preservation. The practical benefits of synchronized pipelining are demonstrated through experimental results on common algorithms such as sorting and Fourier transforms.

1 Introduction

Multi-core processors are becoming ubiquitous: most laptops currently on the market contain at least two execution units, whereas servers commonly use eight or more cores. Since the number of on-chip cores is expected to double with each processor generation, there is a pressing challenge to develop programming methodologies which exploit the power of multi-core processors without compromising correctness and reliability. One prominent approach is to let programmers write sequential programs and to build compilers that parallelize these programs automatically.

Most parallelization techniques rely on some notion of independence, which ensures that certain fragments of the program only access distinct regions of memory, and thus execution of one such code fragment has no effect on the execution of the others. For example, code fragments written in a simple imperative language are guaranteed to be independent if their reads and writes are *disjoint*, in which case their sequential composition can be parallelized without modifying the overall semantics of the program. More refined notions of independence include the classical notions of absence of flow dependence, anti-dependence, or output dependence [7].

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 173–187, 2010.

Well-understood methods exist for detecting loops whose iterations are independent (i.e., they do not contain *loop-carried dependencies*) and parallelizing them. These techniques have been used to achieve automated/correct parallelization of a number of algorithms for scientific computing such as, e.g., image processing, data mining, DNA analysis, or cosmological simulation. However, these parallelization methods do not provide significant speedups for other algorithms which contain sequences or nesting of loops whose iterations are partially dependent and/or irregular. Examples of such loops appear, for example, in sorting algorithms or Fourier transformations. On the other hand, such algorithms can be parallelized efficiently by the technique that we propose, *synchronized pipelining*, which allows loops with dependencies to be executed in parallel by making sparse use of synchronization events to ensure that the ahead-of-time execution of loop iterations does not alter the original semantics.

Our proposal is illustrated in Section 2 with a mergesort algorithm. As a warm-up to Section 2, let us first consider synchronized pipelining in its simplest form, when it deals with two consecutive loops manipulating an array structure:

$$\texttt{while } b_1 \texttt{ do } c_1 ; \texttt{while } b_2 \texttt{ do } c_2$$

For simplicity, assume that the data dependence between c_1 and c_2 is restricted to the contents of the array structure. The aim is to return code that may start the execution of some iterations of c_2 before completion of the loop with body c_1. To justify such a transformation, we rely on *eventual independence*, a generalization of independence which accounts for the possibility of executing the $m+1$-th iteration of a loop ahead of time. Informally, c_2 is eventually independent from c_1 iff for every n_2, there exists n_1 such that after n_1 iterations of c_1 and n_2 iterations of c_2, c_1 and c_2 are independent. Once eventual independence between the two loops is established, it is possible to define a semantics-preserving transformation that outputs a program:

$$\texttt{while } b_1 \texttt{ do } c_1' \parallel \texttt{while } b_2 \texttt{ do } c_2'$$

where c_1' is obtained from c_1 by adding event announcements to indicate that part of the computation of c_2 can be performed, and c_2' is obtained from c_2 by inserting blocking statements that control the gradual and early computation of c_2; in both cases, the transformation of c_i into c_i' is guided by the eventual independence relation.

In the course of the paper, we develop the notions of eventual independence and synchronized pipelining, starting from the simple case discussed above and then dealing with sequences of loops and nested loops. In addition, we illustrate the benefits of our approach, drawing experimental results from common cases such as the above mentioned sorting algorithms and Fourier transforms. We also outline the necessary procedures and tools to automatically generate this transformation for the case in which we deal with simple data structures (arrays), and outline future lines of research to extend this approach to more general problems. In summary, the main contributions of this paper are the formal definition of eventual independence (Section 4), eventual independence

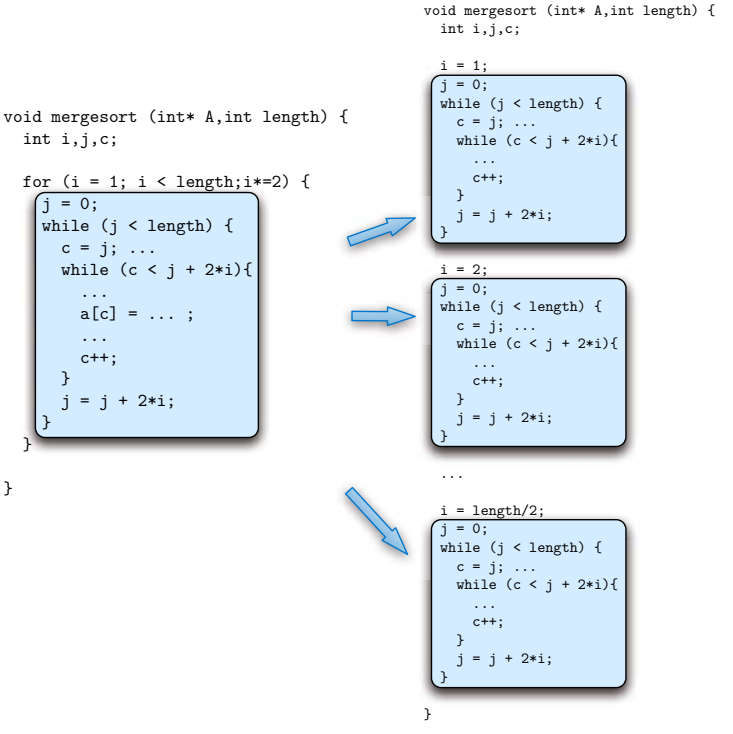


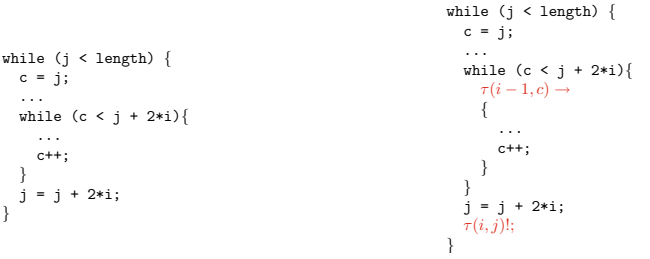
Fig. 1. Iterative mergesort algorithm

criteria for the particular case of array manipulating loops, and an experimental evaluation of the benefits of synchronized pipelining (Section 6). Although many of the concepts and results of the paper only make minimal assumptions on the programming language, we carry our development in the setting of a parallel imperative language with events, introduced in Section 3.

2 Motivating Example: Mergesort

Figure 1 presents the structure of an iterative mergesort algorithm. After unrolling some of the `for` loop iterations from the fragment shown on the right of the figure, we have a sequence of iterations of the inner loop `while(j <` `length)`$\{\ldots\}$ accessing and modifying the array intervals $[0, 1]$, $[2, 3]$, .., [`length`-1, `length`] in the first iteration, the intervals $[0, 3]$, $[4, 7]$, ..., [`length`-3, `length`] in the second iteration, and so on until the last iteration in which the intervals $[0,$ `length`$/2]$ and [`length`$/2+1$, `length`] are accessed.

One can clearly see that the first and second unrolled iteration cannot be executed in parallel (without changes) since they read and/or modify overlapping regions of the array. However, after partial completion of the first iteration, the



(a) Original **(b)** Pipelined

Fig. 2. Illustrative example of pipelined code

second iteration can advance without waiting for the first iteration to finish. For instance, the second iteration can safely start processing the array interval $[0, 3]$, right after the first iteration has finished processing the array intervals $[0, 1]$ and $[2, 3]$. The parallelization technique we propose allows the second loop iteration to gradually progress in parallel with the first one (and successive ones), introducing synchronization primitives in order to preserve the original semantics. To this end, we rely on a heuristic oracle Ω, defined in terms of the number of steps already executed by the first and second loop, that determines at which point of the first loop it is safe to enable a partial execution of the second one.

Figure 2 gives a brief but illustrative scheme of how the code in Figure 1 is to be annotated with parallelization primitives. In this case we use τ to denote such device, using a question mark to signify a wait event on a certain subscript (or set of subscripts) that the current loop is waiting to use, and an exclamation mark to denote a signaling event which allows other threads to continue execution.

3 Setting

The target language for synchronized pipelining is a simple imperative language with arrays, extended with parallel composition and synchronization primitives.

The extension includes an empty statement nil, a standard parallel composition $\|$, and event-based synchronization primitives. We assume given a set of events $\mathcal{S}$ used for synchronization. Let $\tau \in \mathcal{S}$ and $S \subseteq \mathcal{S}$ represent a synchronization event and a synchronization event set, respectively. The statement $S!$ is a non-blocking announcement of the events in S, whereas the statement $\tau \rightarrow c$ waits for the event τ to be announced before proceeding with the execution of c.

Let $Stmt$ be the set of program statements, Σ be the set of mappings from program variables to integer values, and $\mathcal{S}^\star$ be the powerset of $\mathcal{S}$. The program semantics is given by a transition relation between configurations, where a configuration is either an exceptional configuration abort, resulting, e.g., from and array-out-of-bound access, or a normal configuration, i.e., an element of $Stmt \times \Sigma \times \mathcal{S}^\star$. Formally, the semantics is given by a small-step relation: $\rightsquigarrow \subseteq (Stmt \times \Sigma \times \mathcal{S}^\star) \times ((Stmt \times \Sigma \times \mathcal{S}^\star) + \{\mathsf{abort}\})$.

$$\frac{}{\langle S!, \sigma, \epsilon\rangle \rightsquigarrow \langle \mathsf{nil}, \sigma, \epsilon \cup S\rangle} \qquad \frac{\tau \in \epsilon}{\langle \tau \rightarrow c, \sigma, \epsilon\rangle \rightsquigarrow \langle c, \sigma, \epsilon\rangle}$$

$$\frac{c \equiv d \quad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \langle d', \sigma', \epsilon'\rangle \quad d' \equiv c'}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle} \qquad \frac{c \equiv d \quad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$\frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \langle c' \parallel d, \sigma', \epsilon'\rangle} \qquad \frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$i \parallel \mathsf{nil} \equiv i \qquad i \parallel j \equiv j \parallel i \qquad i \parallel (j \parallel k) \equiv (i \parallel j) \parallel k$$

Fig. 3. Operational semantics (excerpts)

The transition rules for synchronization and parallel execution are given in Figure 3, together with the definition of the congruence relation $\equiv \subseteq \mathit{Stmt} \times \mathit{Stmt}$; all other rules are standard. Note that event announcement is asynchronous and that event identifiers are never removed from ϵ. Thus, once an event has been announced, and until the end of the program execution, every process waiting for that event is ready to proceed.

Example 1. Consider for example the statement $(\mathtt{x} := 5; \tau!) \parallel \tau \rightarrow \mathtt{x} := 1$. Starting from a state where τ has not been announced, the execution terminates with the variable x holding the value 1, since $\mathtt{x} := 1$ cannot proceed before the event τ has been announced.

As usual, we can derive from the small-step semantics an evaluation semantics $\Downarrow \subseteq (\mathit{Stmt} \times \Sigma \times \mathcal{S}^\star) \times (\Sigma + \mathsf{abort})$, by setting:

$$\begin{array}{lll} \langle c, \sigma, \epsilon\rangle \Downarrow \sigma' & \text{iff} & \exists \epsilon'.\ \langle c, \sigma, \epsilon\rangle \rightsquigarrow^\star \langle \mathsf{nil}, \sigma', \epsilon'\rangle \\ \langle c, \sigma, \epsilon\rangle \Downarrow \mathsf{abort} & \text{iff} & \langle c, \sigma, \epsilon\rangle \rightsquigarrow^\star \mathsf{abort} \end{array}$$

where $\rightsquigarrow^\star$ denotes the reflexive and transitive closure of $\rightsquigarrow$. In turn, the evaluation semantics can be used to define a notion of semantic equivalence.

Definition 1 (Semantic Equivalence). *Let $c_1, c_2 \in \mathit{Stmt}$ be two statements, $\sigma \in \Sigma$ be a state and $\epsilon \subseteq \mathcal{S}$ be a set of synchronization events. We say that c_2 simulates c_1 w.r.t. σ and ϵ, written $[\![c_1]\!] \leq_{(\sigma,\epsilon)} [\![c_2]\!]$, iff for every $\sigma' \in (\Sigma + \mathsf{abort})$, we have $\langle c_1, \sigma, \epsilon\rangle \Downarrow \sigma' \Rightarrow \langle c_2, \sigma, \epsilon\rangle \Downarrow \sigma'$. We say that c_1 and c_2 are semantically equivalent w.r.t. σ and ϵ, written $[\![c_1]\!] \equiv_{(\sigma,\epsilon)} [\![c_2]\!]$, iff $[\![c_1]\!] \leq_{(\sigma,\epsilon)} [\![c_2]\!]$ and $[\![c_2]\!] \leq_{(\sigma,\epsilon)} [\![c_1]\!]$.*

4 Eventual Independence

The purpose of this section is to introduce the notion of eventual independence, and to discuss how eventual independence relations may be inferred. For the sake of completeness, we start by recalling the semantic notion of independence between two statements.

Definition 2 (Independent Statements). *Two statements $c_1, c_2 \in Stmt$ are* independent *iff* $[\![c_1; c_2]\!] \equiv [\![c_1 \parallel c_2]\!]$.

Eventual independence aims to capture a relation between iterations of two loop bodies c_1 and c_2, and thus would be naturally formalized as a relation between natural numbers. For the clarity of the technical development, it is however preferable to view eventual independence as a relation between natural numbers and events, and assume given a function $\lambda : \mathbb{N} \to \mathcal{S}$ that assigns to each natural number m of loop_2 the event $\lambda(m)$ that will release the m-th iteration of loop_2.

Definition 3 (Eventual Independence Relation). *Statements $c_1, c_2 \in Stmt$ are eventually independent w.r.t. a relation $\Omega \subseteq \mathbb{N} \times \mathcal{S}$ iff for all $m, n \in \mathbb{N}, \epsilon \subseteq \mathcal{S}$ s.t. $(n, \lambda(m)) \in \Omega$, $\sigma \in \Sigma$ and no synchronization variables in ϵ appear in c_1 or c_2:*

$$[\![c_1^n; c_2^{m-1}; c_1^k; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^n; c_2^{m-1}; (c_1^k \parallel c_2)]\!]$$

for all $k \in \mathbb{N}$. The expression c^i stands for the sequential composition of i instances of the statement c. Given Ω and $n \in \mathbb{N}$, we let $\omega(n) = \{s \mid (n, s) \in \Omega\}$.

Example 2. Consider the following program:

$$\texttt{while b}_\texttt{i}\ \texttt{do}\ \{\texttt{a[i]} := \texttt{a[i]} + 1; \texttt{i} := \texttt{i} * 2\};\ \texttt{while b}_\texttt{j}\ \texttt{do}\ \{\texttt{a[j]} := \texttt{a[j]} + 1; \texttt{j} := \texttt{j} + 1\}$$

The two loop statements are not necessarily independent, but one can define an eventual independence relation over the loop bodies in order to parallelize their iterations. In this case, the loop statements are eventually independent with respect to a relation Ω, if $(n, \lambda(m)) \in \Omega$ implies $i^\star * 2^n < j^\star + m$, where $i^\star$ is the initial value of variable i and $j^\star$ is the initial value of variable j.

In practice, when considering sequential code, it is sufficient to state the semantics equivalence in terms of the event set $\epsilon = \emptyset$. From the definition of eventual independence, if $\lambda(m) = s$, then the m^{th} execution of c_2 shall wait for the event s to execute. Assuming $(n, s) \in \Omega$ then it is safe to signal the event s after executing n times the statement c_1, allowing the m^{th} execution of statement c_2 to take place. Indeed, by definition of Ω, it follows from $(n, s) \in \Omega$ that after n iterations of c_1, any and all subsequent executions of c_1 do not modify a piece of memory on which the m^{th} iteration of c_2 depends.

The main reason for defining the Ω relation is to link the iterations of the loop bodies that are safe to execute in parallel. If we take $m = 1$ in the definition, then we see that n is simply the number of iterations of c_1 that we need to execute before we can execute the first iteration of c_2 (in parallel with the remaining iterations of the first loop) without altering the semantics of the original program. Higher values of m are in relation through Ω with the values n after which it is safe to execute the m^{th} iteration of the second loop, provided that the $m - 1$ previous iterations where executed following the guidelines that Ω defines. This is the basis for the transformation we are aiming at and it is formalized in the next section.

The set $\omega(n)$, which is defined in terms of Ω, is the set of all the events that are safe to announce after n executions of the statement c_1. Since the purpose

of this definition is to have a construct that will allow us to denote the set of events the first loop can safely announce after each iteration has ended, we will mainly use ω when defining our transformation.

4.1 Inferring Eventual Independence

The eventual independence relation Ω and the function λ are essential ingredients of synchronized pipelining, as they will be used to guide the insertion of synchronization statements in the original program. Therefore, it is important to be able to infer Ω and λ for a large class of code fragments. We have been able to infer this data efficiently for the algorithms under consideration, that manipulate array structures of significant size. Consider the case in which both c_1 and c_2 read and modify data from a single array `a`, iterating over the induction variables h_1 and h_2 respectively. By simple code inspection, one can easily collect the sets of syntactic expressions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ used to read or update the array `a` inside the loop body. These array accesses are not always expressed in terms of the induction variables h_1 and h_2. However, in general, we have found that they are expressed in terms of induction variables h'_1 and h'_2 *derived* from h_1 and h_2. In those cases, induction variable analysis [4] allows one to rewrite the derived induction variables h'_1 and h'_2 in terms of the induction variables h_1 and h_2, i.e. $h'_1 = f_1(h_1)$ and $h'_2 = f_2(h_2)$ for some function expressions f_1 and f_2.

Most frequently, when h'_i is an induction variable derived from h_i, then f_i is a linear function on h_i. More complex cases may arise, for instance when f_i is defined as a polynomial or geometric function on h_i. In those cases, the expressions $\boldsymbol{e}_1(h'_1)$ and $\boldsymbol{e}_2(h'_1)$ are easily rewritten in terms of the inductive variables, i.e., as $\boldsymbol{e}_1(f_1(h_1))$ and $\boldsymbol{e}_2(f_2(h_2))$. By static interval analysis, we can approximate the regions of data that are read and modified by c_1 and c_2, in terms of the induction variables h_1 and h_2, and the expressions $\boldsymbol{e}_1(f_1(h_1))$ and $\boldsymbol{e}_2(f_2(h_2))$.

Assume $[l_1^{rw}, u_1^{rw}]$ represents the interval of the array `a` that is written or read by c_1, where l_1^{rw}, u_1^{rw} are integer expressions that depend on h_1 (and similarly with c_2). Since $\boldsymbol{e}(f_1(h_1))$ and $\boldsymbol{e}(f_2(h_2))$ are linear (or polynomial) functions on h_1 and h_2, one can determine whether they are monotonic (or determine the points from which they are monotonic). If the l and u expressions are increasing as the h variables grow (the decreasing case is symmetrical) one can propose an eventual independence relation Ω. For instance when l_1^{rw} and u_2^{rw} are increasing functions, we determine the pairs (a,b) of values for h_1 and h_2 such that $u_2^{rw} < l_1^{rw}$, and then, since the b^{th} iteration of c_2 is independent of the a^{th} iteration of c_1, we can have $(a, \lambda(b)) \in \Omega$.

Example 3. We show in this paragraph how to determine an eventual independence relation for this simple pair of loop statements

$$\texttt{while } b_1 \texttt{ do } c_1; \texttt{while } b_2 \texttt{ do } c_2$$

where c_1 and c_2 are defined as

$$\begin{array}{lcl} c_1 & \doteq & \mathtt{a[x]} := 1; \mathtt{x} := \mathtt{x} + 1 \\ c_2 & \doteq & \mathtt{y} := \mathtt{y} + \mathtt{a[z]}; \mathtt{z} := \mathtt{z} + 1 \end{array}$$

First of all, notice that statements c_1 and c_2 access the array $\mathtt{a}$, so they are not independent. By examining statements c_1 and c_2, it is immediate that the indexes of the array accesses are monotonically increasing and the relation between the initial values of program variables (denoted $x^\star$ for a variable x) define the eventual independence relation. In this case, a simple induction variable analysis will define $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, and thus l_1^{rw},l_2^{rw}, u_1^{rw} and u_2^{rw}, as a linear function of the induction variables: $l_1^{rw}(h_1) = u_1^{rw}(h_1) = h_1 + x^\star$ and $l_2^{rw}(h_2) = u_2^{rw}(h_2) = h_2 + z^\star$. Thus, the procedure's requirements translate into: $h_2 + z^\star < h_1 + x^\star$. The argument above allows us to propose an eventual independence relation Ω.

$$\begin{array}{l} (z^\star - x^\star + 1, \lambda(1)) \in \Omega_{c_1,c_2} \\ \forall x.\ x \leq z^\star - x^\star + 1 \Rightarrow (x, \lambda(1)) \notin \Omega_{c_1,c_2} \end{array}$$

This Ω relation formalizes the intuition that c_1 and c_2 can be executed in parallel as long as every iteration k of c_2 executes after the iteration number $z^\star - x^\star + k$ of c_1. Furthermore, since the size of the array $\mathtt{a}$ ($|\mathtt{a}|$) is bounded, if c_1 is executed more than $|\mathtt{a}| - x^\star$ times, we end up at an exceptional state abort, in which case any execution of c_2 is independent. In conclusion, the following relation Ω determines the eventual independence between c_1 and c_2:

$$\begin{array}{l} x + x^\star \leq |\mathtt{a}| \wedge y \leq x + z^\star - x^\star - 1 \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \\ x + x^\star > |\mathtt{a}| \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \end{array}$$

5 Synchronized Pipelining

We now define synchronized pipelining, starting from two consecutive loops, and then extending the transformation to sequences of loops and nested loops.

Consider a program c of the form $\mathtt{while}\ b_1\ \mathtt{do}\ c_1; \mathtt{while}\ b_2\ \mathtt{do}\ c_2$, where c_1 and c_2 are compound statements that access an array. We assume that the boolean conditions b_1 and b_2 are not affected by the execution of c_2 and c_1, respectively. Further, we let h_1 and h_2 be program counters that determine the number of iterations already performed for the first and second loop respectively. Our aim is to transform the program so that it executes both loops in parallel. To preserve the program semantics, the transformation must insert code that ensures a correct synchronization between the two loops, so the resulting program will be of the form $\mathtt{while}\ b_1\ \mathtt{do}\ c_1' \parallel \mathtt{while}\ b_2\ \mathtt{do}\ c_2'$, where c_1' is derived from c_1 by adding event announcements and c_2' is derived from c_2 by adding synchronization guards. Both transformations are guided by a relation Ω of eventual independence and by a function λ that are given as input to the transformation.

Definition 4. *The* synchronized pipelining *of c is statement $\bar{\bar{c}}$ defined as:*

$$\bar{\bar{c}} = (\texttt{while } b_1 \texttt{ do } c_1'); S! \parallel \texttt{while } b_2 \texttt{ do } c_2'$$

where $c_1' = c_1; \omega(h_1)!$, $c_2' = \lambda(h_2) \rightarrow c_2$, and S is the set of all events on which statement c_2' can wait.

Statement $S!$ is introduced after the execution of c_1' to ensure that all events are indeed announced, and thus the progress of the original program is preserved. In order to accomplish that, statement $S!$ simply announces all events, in any order. Since all events in which statement c_2' is waiting are eventually announced by $S!$, statement c_2' cannot block indefinitely. For the same reason, $c \leq \bar{\bar{c}}$. Notice that the set of events announced by c_1' and $S!$ may be redundant. In practice, one can reduce program size and synchronization overhead by statically removing duplicated events. Similarly, c_2 may be simplified by removing synchronization primitives that wait on the same event. We assume, however, the definition given above for notational simplicity.

The eventual independence condition determined by Ω is enough to show that the semantics is preserved. That is, every execution state reached by the final program is also reachable by the original one.

Proposition 1 (Semantics Preservation). *For every initial state $\sigma \in \Sigma$ and every event set ϵ disjoint from the fresh synchronization variables introduced by the transformation, we have that $[\![c]\!] \equiv_{(\sigma,\epsilon)} [\![\bar{\bar{c}}]\!]$.*

5.1 Extensions

We first analyze the case of a sequence of loops. Then, we explain how we proceed in the presence of nested loops.

Loop Sequences. Now suppose the original program is of the form:

$$\texttt{while } b_1 \texttt{ do } c_1; \ldots; \texttt{while } b_n \texttt{ do } c_n$$

The idea is to parallelize the whole program by progressively applying the basic transformation to each pair of interfering loops. Therefore, we must provide for all i, j such that $i < j$ an eventual independence relation $\Omega_{i,j}$ and a function $\lambda_{i,j} : \mathbb{N} \rightarrow \mathcal{S}$. By definition of eventual independence, we must have for every $(n, \lambda_{i,j}(m)) \in \Omega_{i,j}$ and for all state σ and event set ϵ:

$$[\![c_i^n; c_j^{m-1}; c_i^k; c_j]\!] \equiv_{(\sigma,\epsilon)} [\![c_i^n; c_j^{m-1}; (c_i^k \parallel c_j)]\!]$$

Since the parallel execution of the i^{th} loop may interfere not only with its immediately preceding loop, but with every preceding one, we synchronize each pair of non-independent loops. Thus, the i^{th} loop of the final program becomes:

$$\texttt{while } b_i \texttt{ do } \bigcup_{1 \leq j < i} \lambda_{i,j}(h) \rightarrow \left(c_i; \bigcup_{i < j \leq n} \omega_{i,j}(h)! \right); \forall_{i<j\leq n} S_{i,j}!$$

where $S_{i,j}$ stands for all the synchronization events used to synchronize execution between `while` b_i `do` c_i and `while` b_j `do` c_j, for every $i < j$. From the expression above, it may seem that excessive synchronization overhead is introduced. However, the actual number of synchronization primitives depends on the definition of λ and ω, and on the removal of duplicated synchronization events.

Nested Loops. We now turn our attention to a different but more common program structure: nested loops. Consider the following program as the target of the parallelization: `while` a `do` $(c_1; \texttt{while}\ b\ \texttt{do}\ c; c_2)$. In order to be able to apply our transformation we take the following assumptions:

1. We assume that the number of iterations of the outer loop (or an overapproximation) can be computed at runtime. In the rest of this section we let β stand for the number of iterations that may be computed at runtime and, for simplicity, we assume that the boolean condition a is of the form $l \leq \beta$, where l is the induction variable of the outer loop, incremented with step 1 from the initial value 1. In practice, the exact form of a may differ from this assumption, but we assume that it is possible to evaluate the number of iterations at runtime based on the current memory state. Intuitively, if we can determine the exact number of iterations of the outer loop, we can unroll it and parallelize the resulting program by applying the transformation on sequences of loops as explained above. However, assuming that we can statically determine the exact number of iterations is an unnecessary and too strong assumption.
2. We assume also that there is no interference between the scalar variables read and modified in c_1 and c. We can reduce the interference between loop iterations by vectorizing each scalar variable v into an array $\hat{v}$, with the cost of extra memory usage. For every statement c and boolean condition b, we denote $\hat{c}[l]$ and $\hat{b}[l]$ the result of vectorizing scalar variables in c and b, respectively. The value of the variable l determines which position of the vectorized variables is in use. At the end of the transformed program, a `sync` operation takes each vectorized variable $\hat{v}$, and transforms it back into the original scalar variable v, i.e., executes $v = \hat{v}[\beta]$. The reason for this vectorization is to avoid clashes between the values that are accessed by the fragments `while` $\hat{b}[i]$ `do` $\hat{c}[i]$, for different values of i.
3. The last hypothesis we make is that the scalar variables initialized by the statement c_1 are not modified by c or c_2 after vectorization. This is a reasonable assumption to make, since data structure accesses are in most cases confined to the inner loop. This allows us to ignore dependencies between these instructions and the rest of the loop.

As before, for every $i, j \in \mathbb{N}$ s.t. $i < j \leq \beta$ we need a function $\lambda_{i,j} : \mathbb{N} \to \mathcal{S}$ mapping iterations to synchronization events. In this case, the parametric relation $\Omega_{i,j}$ takes into account the last instructions of the outer loop. We require, if $(n, \lambda_{i,j}(m)) \in \Omega_{i,j}$ and for every $\epsilon \subseteq \mathcal{S}$ and $\sigma \in \Sigma$, that:

$$[\![\hat{c}[i]^n; \hat{c}[j]^{m-1}; \hat{c}[i]^k; \hat{c_2}[i]; \hat{c}[j]]\!] \equiv_{(\sigma,\epsilon)} [\![\hat{c}[i]^n; \hat{c}[j]^{m-1}; (\hat{c}[i]^k; \hat{c_2}[i] \parallel \hat{c}[j])]\!]$$

The transformation is similar to the one performed for sequences of loops. Since inner loops are syntactically equal, the value of induction variable l corresponding to the outer loop is used to distinguish between different iterations. The transformation follows, thus, the scheme:

$$\begin{array}{l}\texttt{while } a \texttt{ do} \quad \tau_{c,l-1} \rightarrow (\hat{c_1}[l]; \tau_{c_1,l}!)\,; \\ \quad \texttt{while } \hat{b}[l] \texttt{ do } \left(\bigcup_{1\leq j<l} \lambda_{l,j}(h) \rightarrow \left(\hat{c}[l]; \bigcup_{l<j\leq\beta} \omega_{l,j}(h)!; \hat{c_2}[h']\right)\right)\,; \\ \texttt{sync}\end{array}$$

Notice that the order in which the instances of $\hat{c}[l]$ are executed is preserved.

5.2 Motivating Example Revisited

Our motivating example, `mergesort`, was annotated with synchronization statements that follow the guidelines described in our transformation. If we take two consecutive iterations of the main loop of the program, we can sketch the constructs we have presented in our theoretical model.

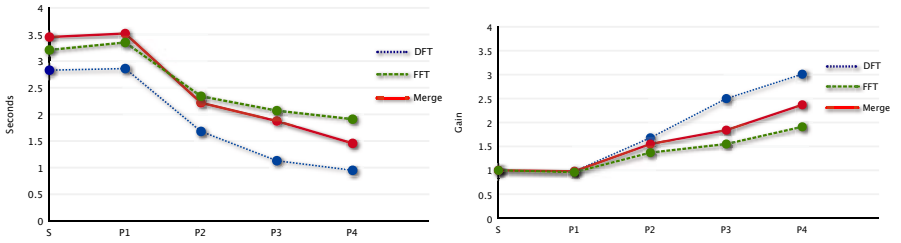
Starting from the original code, we need first to vectorize the variables that parameterize our inner loop. In our example this is variable `i`. Since we need to spawn a new procedure in order to launch (possibly) a new thread, we encapsulate the inner loop in a function call, which receives `i` as a parameter. Then, the stack allocation scheme automatically vectorizes variable `i` for us, since now each iteration will possess its own copy of `i`, independent from the others, and initialized to the value which each iteration would see in a sequential execution. The only problem here consists in working with a language which allows function calls to be made to run in parallel. Later we will explain how we deal with this issue in practice.

In the original program, the variable `c` is the expression used for writing in the array, and furthermore it is the lowest variable which is read or written in the array. On the other side, the variable `r` is the highest variable which is read, this is a consequence of the initial state of the inner loop and is preserved in the loop body. We can analyze the loop and determine that `c` is monotonically increasing. It follows that if we have two consecutive iterations,i and $i+1$, of the loop, the latter cannot proceed unless it can assure that the value of[1] $\texttt{c}^{\texttt{i}}$ is bigger than that of $\texttt{r}^{\texttt{i+1}}$.

Thus, the following piece of code is added to the original code:

```
...
while (j < length){
      while (c-j<2*i){
        event_wait(r);
        fromQueue = last(Q);
        if (l-j > i){
            ...
            A[c] = dequeue(Q);
        }
        event_announce(c);
        c++;
      }
...
```

[1] We use superscripts to denote which loop variables belong to and subscripts to refer to the value of the variable at a given iteration of its loop.



(a) Computation Time **(b)** Speedup

Fig. 4. Experimental Results

Our function λ essentially maps $m \rightarrow \mathtt{r}_{\mathtt{m}}$. It becomes apparent now that our Ω relation must relate every tuple $(n,\lambda(m))$ where $\mathtt{c}_{\mathtt{n}}^{\mathtt{i}+1}$ is larger than $\mathtt{r}_{\mathtt{m}}^{\mathtt{i}}$.

We now need to determine λ and Ω for every other possible combination of iterations. But since the same loop is repeated, with the same properties, we require the same condition to advance, namely $\mathtt{r}_{\mathtt{m}}^{\mathtt{i}} < \mathtt{c}_{\mathtt{n}}^{\mathtt{j}}$, and thus $\Omega_{i,j}$ again contain pairs $(n,\lambda_{i,j}(m))$ which meet that condition.

6 Experimental Results

We have experimented with the parallelizing transformation taking as input a program written in a subset of *C* and returning a *Cilk* [5] program. *Cilk* is an extension of *C* for multithreaded parallel programming, that provides a light-weight thread model based on job stealing.

We proceed by annotating the source program with *Cilk* statements for thread creation and synchronization, using *Cilk locks* and *spawn* procedures to implement event signaling and efficient variable synchronization. We encapsulate inner loops in *spawned* procedures, and use the *C* stack allocation scheme to efficiently allocate memory for vectorization.

The proposed transformation has been applied to well-known algorithms that traverse arrays to obtain information as to the applicability and the efficiency of our approach. In all cases, the transformation yields good results unless the input size is tiny enough to make the synchronization overhead relatively significant.

For our tests we have used a 64bit Intel(R) Core(TM)2 Quad CPU at 2.4 GHz clock speed, 1GB of DIMM 800 MHz memory, running GNU/Linux.

In all cases we have labeled the graphics with `S` for the sequential (unmodified) algorithm, running on a single processor, and we have labeled `Pn` for our modified, pipelined algorithm with `n` processors.

Figure 4 shows the computing time and the relative performance gain of the DFT[2], FFT[3], and MergeSort algorithms run under the different conditions we have explained. The pipelined version of our DFT program is slightly slower while

[2] Discrete Fourier Transform.
[3] Fast Fourier Transform.

running with only one processor, due to the overhead of synchronization variable allocation and signaling. Once we augment the number of available processors the amount of time spent computing starts to decrease as the several runs on the array on which we are working start to (safely) overlap. The efficiency gain is almost linear, but of course the overhead of signaling and also the thread creation and manipulation overhead add some extra work to the computation. The algorithm used is well suited for our transformation since it copies the input array and then modifies one element at a time incrementally, allowing several elements to be modified at the same time without interference.

Our experiments with an FFT algorithm also yield good results, though not as good as with the DFT algorithms. The reason for this is that unlike DFT, FFT traverses the input array heavily and performs the computation in-place, so it slowly gives up resources and thus the overlapping of different traversals is smaller. Nevertheless, some performance gain is indeed achieved in our pipelined version of the algorithm, roughly a 50% gain with 4 processors. The pipelined version is still outperformed by the sequential one in the case we have a single processor available, again due to synchronization overheads.

The last benchmark we present is that of our motivating example, namely mergesort. This algorithm also traverses an array several times incrementally, which allows us to obtain greater benefits from our transformation. The benchmarks were made sorting an array of one million elements. The results show that our transformation yields a 240% efficiency increase by overlapping the merging steps that are otherwise run sequentially, for a 4 processor machine.

7 Related Work

Ottoni et al. [13] proposed a technique called Decoupled Software Pipelining (DSWP) to extract the fine-grained parallelism hidden in most applications. The process is automatic, and general, since it considers non-scientific applications in which the loop iterations have heavy data dependencies. It provides a transformation that is slightly different to typical loop parallelization, in which each iteration is assigned alternately to each core, with an appropriate synchronization to prevent data races. As a result, no complete iteration is executed simultaneously with another one, since every iteration has a data dependence with every other one. Instead of alternating each complete loop iteration on each core, DSWP splits each loop body before distributing them among the available cores. This technique improves the locality of reference of standard parallelization techniques, and thus reduces the communication latency. It is effective in a more general set of loop bodies, but it does not take advantage of the eventual data independence hidden in scientific algorithms.

A recent experimental study [10] analyzes particular cases in which standard automatic parallelization fails to introduce significant improvements. This is the case of applications that manipulate complex and mutable data structures, such as Delauney mesh refinement and agglomerative clustering. The authors propose a practical framework, the *Galois* system, that relies on syntactic constructs to

enable programmers to hint to the compiler on parallelization opportunities and an optimistic parallelization run-time to exploit them. Due to the unpredictability of irregular operations on mutable and complex data structures, the *Galois* framework is mostly based on runtime decisions and backtracking, and does not exploit statically inferred data dependence.

Data Parallel Haskell [14] (DPH) provides nested data parallelism to the existing functional language compiler GHC. Flat parallelism is restricted to the concurrent execution of sequential operations. Nested parallelism generalizes flat parallelism by considering the concurrent execution of functions that may be executed in parallel, and thus provides a more general and flexible approach, suitable for irregular problems. DPH extends Haskell with parallel primitives, such as *parallel arrays* and a set of *parallel operations* on arrays. The compiler compiles these parallel constructions by desugaring them into the GHC Core language, followed by a sequence of Core-to-Core transformations. DPH is a notable framework for the specification of concurrent programs, but the compiler is not intended to automatically discover parallel evaluations.

In a different line of work, the Manticore project is developing a parallel programming language for heterogeneous multi-core processor systems [3]. A main feature of the language is the support for both implicit and explicit threading. Nevertheless, as a design choice, it avoids implicit parallelism (i.e., it requires the programmer to hint parallelism by providing annotations) since they claim implicit parallelism to be only effective for dense regular parallel computations.

The goal of the Paraglide project at IBM is to assist the construction of highly-concurrent algorithms. The Paraglider tool [17] is a linearization-based framework to systematically construct complex algorithms manipulating concurrent data structures, from a sequential implementation. This approach combines manual guidance with automatic assistance, focusing mainly on fine-grained synchronization.

8 Conclusion

Synchronized pipelining is a parallelization technique that relies on eventual independence, a new refinement of the established notion of independence, to successfully transform programs with nested loops. This paper has set the theoretical foundations of the transformation, and showed its practical benefits on representative examples.

Future work includes applying this transformation to general recursive procedures, which is a possibility if the program is first transformed into an iterative version of itself. This is a widely studied optimization problem [11] which can significantly improve performance. Other lines of research include applying the transformation to languages that manipulate the heap. Many concepts developed in this paper are largely independent of the underlying programming language, and the main issue is rather to find an analysis to detect independence. Recent work on the use of shape analysis and separation logic for detecting data dependence and for parallelization provide a good starting point (e.g., [15,16,8,6,12]).

References

1. Allan, V., Jones, R., Lee, R., Allan, S.J.: Software pipelining. ACM Computing Surveys 27(3), 367–432 (1995)
2. Fahringer, T., Scholz, B.: A unified symbolic evaluation framework for parallelizing compilers. IEEE Transactions on Parallel and Distributed Systems PDS-11(11), 1105–1125 (2000)
3. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly-threaded parallelism in manticore. In: Hook, J., Thiemann, P. (eds.) ICFP, pp. 119–130. ACM, New York (2008)
4. Gerlek, M., Stoltz, E., Wolfe, M.: Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. ACM Transactions on Programming Languages and Systems 17(1), 85–122 (1995)
5. Supercomputing Technologies Group: Cilk 5.4.6 reference manual (1998)
6. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)
7. Hennessy, J., Patterson, D.: Computer Architecture: a quantitative approach. Morgan Kaufman, San Francisco (2003)
8. Hummel, J., Hendren, L., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: PLDI, pp. 218–229 (1994)
9. Joyner, M., Budimlic, Z., Sarkar, V.: Optimizing array accesses in high productivity languages. In: Perrott, R.H., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC 2007. LNCS, vol. 4782, pp. 432–445. Springer, Heidelberg (2007)
10. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: Ferrante, J., McKinley, K. (eds.) PLDI, pp. 211–222. ACM, New York (2007)
11. Liu, Y., Stoller, S.: From recursion to iteration: What are the optimizations? In: PEPM, pp. 73–82 (2000)
12. Marron, M., Kapur, D., Stefanovic, D., Hermenegildo, M.: Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 94–108. Springer, Heidelberg (2008)
13. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic thread extraction with decoupled software pipelining. In: MICRO, pp. 105–118. IEEE Computer Society, Los Alamitos (2005)
14. Peyton Jones, S.: Harnessing the multicores: Nested data parallelism in haskell. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, p. 138. Springer, Heidelberg (2008)
15. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009)
16. Rugina, R., Rinard, M.: Automatic parallelization of divide and conquer algorithms. In: PPOPP, pp. 72–83 (1999)
17. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Gupta, R., Amarasinghe, S. (eds.) PLDI, pp. 125–135. ACM, New York (2008)

Defining Datalog in Rewriting Logic*

M. Alpuente, M.A. Feliú, C. Joubert, and A. Villanueva

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,mfeliu,joubert,villanue}@dsic.upv.es

Abstract. In recent work, the effectiveness of using declarative languages has been demonstrated for many problems in program analysis. Using a simple relational query language, like Datalog, complex interprocedural analyses involving dynamically created objects can be expressed in just a few lines. By exploiting the power of the Rewriting Logic language Maude, we aim at transforming Datalog programs into efficient rewrite systems that compute the same answers. A prototype has been implemented and applied to some real-world Datalog-based analyses. Experimental results show that the performance of solving Datalog queries in rewriting logic is comparable to state-of-the-art Datalog solvers.

1 Introduction

Datalog [17] is a simple relational query language that allows complex interprocedural program analyses involving dynamically created objects to be described in an intuitive way. The main advantage of formulating data-flow analyses in Datalog is that analyses that traditionally take hundreds of lines of code can be expressed in a few lines [19]. In real-world problems, the Datalog rules that encode a particular analysis must be solved generally under the huge set of Datalog facts that are automatically extracted from the analyzed program. In this context, all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results.

The aim of this paper is to provide efficient Datalog query answering in Rewriting Logic [14], which is a very general *logical* and *semantical framework* that is efficiently implemented in the high-level programming language Maude [7]. Our motivation for using Rewriting Logic is to overcome the difficulty of handling metaprogramming features such as reflection in traditional analysis frameworks [11]. Tracking reflective methods invocations requires not just tracking object references through variables but actually tracking method values and method name strings. Unless reflective calls are interpreted during the computation, analysis tools run the danger of incorrectness and incompleteness, and

* This work has been partially supported by the EU (FEDER), the Spanish MEC/MICINN under grant TIN 2007-68093-C02, the Generalitat Valenciana under grant Emergentes GV/2009/024, and the Universidad Politécnica de Valencia under grant PAID-06-07. M. A. Feliú was partially supported by the Spanish MEC FPU grant AP2008-00608.

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 188–204, 2010.

we consider it a challenge to investigate the interaction of static analysis with metaprogramming frameworks [7]. An additional goal of this work is to determine whether MAUDE is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of JAVA programs.

In the related literature, the solution for a DATALOG query is classically constructed following a bottom-up approach, thus the information in the query is not taken advantage of until the model has been built [9]. In contrast, the typical top-down, logic programming interpreter would produce the output by reasoning backwards from the query. Between these two extremes, there is a whole spectrum of evaluation strategies [5,6,18], but in this work, we have considered a top-down approach for developing our transformation, since it is closer to MAUDE's evaluation principle that is based on (non-deterministic) rewriting.

Logic and functional programming are both instances of rule-based, declarative programming; hence, it is not surprising that the relationship between them has been studied. However, the operational principle differs: logic programming is based on *resolution* whereas functional programs are executed by *term rewriting*. There exist many proposals for transforming logic programs into rewriting theories [12,15,16]. These transformations aim at reusing the infrastructure of term rewriting systems to run the (transformed) logic program while preserving the intended observable behavior (e.g., termination, success set, computed answers, etc). Traditionally, translations of logic programs into functional programs are based on imposing an input/output relation (mode) among the parameters of the original program [15]. However, one distinguished feature of DATALOG programs that burdens the transformation is that predicate arguments are not *moded*, meaning that they can be used both as input or output parameters.

One recent transformation that does not impose modes among parameters was presented in [16]. The authors defined a transformation from definite logic programs into (infinitary) term rewriting for the termination analysis of logic programs. Contrary to our approach, the transformation of [16] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [16] does not tackle the problem of efficiently encoding logic (DATALOG) programs containing a huge amount of facts in a rewriting-based infrastructure such as MAUDE. After exploring the impact of different implementation choices (equations *vs* rules, etc.) in our working scenario, i.e., sets of hundreds of facts and a few clauses that encode the analysis, in this work, we present an equation-based transformation that leads to efficient MAUDE-programs.

In previous work [3], we developed a DATALOG query solving technique based on *Boolean Equation Systems* (BESS) [4]. In this paper, we work at a higher level, transforming a high-level DATALOG program into another high-level MAUDE program. Our goal is to take advantage of the flexibility and versatility of MAUDE in order to achieve scalability and meta-programming capabilities without losing the declarative nature of specifying program analyses in DATALOG.

In Section 2, we present our running example: a program analysis expressed as a DATALOG program that we will use to illustrate the general transformation from DATALOG programs into MAUDE programs. In Section 3, we describe such transformation. Section 4 formalizes the general process and establishes its correctness and completeness. Section 5 shows experimental results obtained with realistic examples and compares our MAUDE implementation to state-of-the-art DATALOG solvers. We conclude and discuss future work in Section 6. More details and missing proofs can be found in [1].

2 A Program Analysis Written as a DATALOG Program

DATALOG is a relational language that uses declarative *clauses* to both describe and query a deductive database. A definite DATALOG clause is a function-free Horn clause over a finite alphabet of *predicate* symbols (e.g., relation names or arithmetic predicates, such as $<$) whose *arguments* are either variables or constant symbols. A DATALOG program $\mathcal{R}$ is a finite set of DATALOG clauses [9].

Definition 1 (Syntax of Rules). *Let $\mathcal{P}$ be a set of* predicate *symbols, $\mathcal{V}$ be a finite set of* variable *symbols, and $\mathcal{C}$ a set of* constant *symbols. A* DATALOG clause *r, defined over a finite alphabet $P \subseteq \mathcal{P}$ and arguments from $V \cup C$, $V \subseteq \mathcal{V}$ and $C \subseteq \mathcal{C}$, has the following syntax:*

$$p_0(a_{0,1}, \ldots, a_{0,n_0}) \ :- \ p_1(a_{1,1}, \ldots, a_{1,n_1}), \ \ldots, \ p_m(a_{m,1}, \ldots, a_{m,n_m}).$$

where $m \geq 0$, and each p_i is a predicate symbol of arity n_i with arguments $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$), where p_0 is not arithmetic.

The atom $p_0(a_{0,1}, \ldots, a_{0,n_0})$ in the left-hand side of the clause is the clause's *head*. The finite conjunction of *subgoals* in the right-hand side of the clause is the clause's *body*, *i.e.*, a sequence of atoms that contain all variables appearing in the head. A clause with empty body ($m = 0$) is called a *fact*. A clause with empty head and $m > 0$ is called a *query*, and $\Box$ denotes the empty clause. A syntactic object (argument, atom, or clause) that contains no variables is called *ground*. Moreover, an *existentially quantified variable* is a variable that appears in the body of a clause and does not occur in its head.[1]

Given a DATALOG program $\mathcal{R}$ and a query q, we follow a top-down approach and use SLD-resolution to compute the set of answers of q in $\mathcal{R}$. Given the successful derivation $\mathcal{D} \equiv q \Rightarrow^{\theta_1}_{SLD} q_1 \Rightarrow^{\theta_2}_{SLD} \ldots \Rightarrow^{\theta_n}_{SLD} \Box$, the answer computed by $\mathcal{D}$ is $\theta_1\theta_2 \ldots \theta_n$ restricted to the variables occurring in q.

Let us now introduce the running DATALOG program example that we use throughout the paper. This program defines a simple context-insensitive inclusion-based pointer analysis for an object-oriented language such as JAVA. This analysis is defined by the following predicate `vP/2` representing the fact that a program variable points directly (via `vP0/2`) or indirectly (via `a/2`) to a

[1] In the rest of the paper, DATALOG programs are considered to be as defined here.

given position in the heap. The second clause states that `Var1` points to `Heap` if `Var2` points to `Heap` and `Var2` is assigned to `Var1`:

```
vP(Var,Heap)  :- vP0(Var,Heap).
vP(Var1,Heap) :- a(Var1,Var2),vP(Var2,Heap).
```

The predicates `a/2` and `vP0/2` are defined extensionally by a number of facts that are automatically extracted from the original program being statically analyzed. The intuition is that `a/2` represents a direct assignment from a program variable to another variable, whereas `vP0/2` represents newly created pointers within the analyzed (object-oriented) program from a program variable to the heap. The following code excerpt contains some DATALOG facts complementing the above pointer analysis description for an object-oriented example program.

```
a(v1,v2).     a(v1,v3).     vP0(v3,h4).     vP0(v2,h5).
```

In the considered DATALOG analysis program, a query typically consists in computing the objects in the heap pointed by a specific variable. We write such a query as `?- vP(v1,Heap).`. The expected outcome of this query is the set of all possible answers, i.e., the set of substitutions mapping the variable `Heap` to constants satisfying the query. In the example, the set of computed answers for the considered query is {{`Heap/h4`},{`Heap/h5`}}. Another possible query is `?- vP(Var,h5).`, where `h5` stands for a heap object.

Similarly to [16], our goal is to define a *mode*-independent transformation for (DATALOG) logic programs in order to keep the possibility of running both kinds of queries. Since variables in rewriting logic are input-only parameters, we cannot use them to encode logic variables of DATALOG. We follow the standard approach based on defining a ground representation for logic variables [7,8].

3 From DATALOG to MAUDE

As explained above, we are interested in computing all answers for a given query by term rewriting. A naïve approach is to translate DATALOG clauses into MAUDE rules, and then use the `search`[2] command of MAUDE in order to mimic all possible executions of the original DATALOG program. However, in the context of program analysis with a huge number of facts, this approach results in poor performance [2]. This is because *rules* are handled non-deterministically in MAUDE whereas *equations* are applied deterministically [7].

In this section, we first formulate a suitable representation in MAUDE of the DATALOG computed answers. Then, we informally introduce our equation-based transformation by means of the running example.

[2] Intuitively, `search` $t \rightarrow t'$ explores the whole rewriting space from the term t to any other terms that match t' [7].

3.1 Answer Representation

Let us first introduce our representation of variables and constants of a DATALOG program as *ground terms* of a given sort in MAUDE. We define the sorts `Variable` and `Constant` to specifically represent the variables and constants of the original DATALOG program in MAUDE, whereas the sort `Term` (resp. `TermList`) represents DATALOG terms (resp. lists of terms, built by simple juxtaposition):

```
sorts Variable Constant Term TermList .
subsort Variable Constant < Term .
subsort Term < TermList .
op __ : TermList TermList -> TermList [assoc] .
op nil : -> TermList .
```

For instance, `T1 T2` represents the list of terms `T1` and `T2`. In order to construct the elements of the `Variable` and `Constant` sorts, we introduce two constructor symbols: DATALOG constants are represented as MAUDE *Quoted Identifiers* (`Qid`s), whereas logical variables are encoded in MAUDE by means of the constructor symbol `v`. These constructor symbols are specified in MAUDE as follows:

```
subsort Qid < Constant .
op v : Qid -> Variable [ctor] .
op v : Term Term -> Variable [ctor] .
```

The last line of the above code excerpt allows us to build variable terms of the form `v(T1,T2)` where both `T1` and `T2` are `Term`s. This is used to ensure that the ground representation in MAUDE for existentially quantified variables that appear in the body of DATALOG clauses is unique to the whole MAUDE program.

With ground terms representing variables, we still lack a way to collect the answers for the variables in the query. In our formulation, answers are stored within the term representing the ongoing partial computation of the MAUDE program. Thus, we represent a (partial) answer for the original DATALOG query as a sequence of equations (called answer constraint) that represents the substitution of (logical) variables by (logical) constants computed during the program execution. We define the sort `Constraint` representing a single answer for a DATALOG query, but we also define a hierarchy of subsorts (e.g., the sort `FConstraint` at the bottom of the hierarchy represents inconsistent solutions) that allows us to identify the inconsistent as well as the *trivial* constraints (`Cst = Cst`) whenever possible. This hierarchy[3] allows us to simplify constraints as soon as possible and to improve performance. The resulting MAUDE program is as follows:

[3] The prefix `e` (resp. `Ne`) is used to form an abbreviated name of sorts for empty (resp. non-empty) constraints and constraint sets.

```
sorts Constraint eConstraint NeConstraint TConstraint FConstraint .
subsort eConstraint NeConstraint < Constraint .
subsort TConstraint FConstraint < eConstraint .

op _=_ : Term Constant -> NeConstraint .
op T : -> TConstraint .
op F : -> FConstraint .
op _,_ : Constraint Constraint -> Constraint  [assoc comm id: T] .
op _,_ : FConstraint Constraint -> FConstraint           [ditto] .
op _,_ : TConstraint TConstraint -> TConstraint          [ditto] .
op _,_ : NeConstraint TConstraint -> NeConstraint        [ditto] .
op _,_ : NeConstraint NeConstraint -> NeConstraint       [ditto] .

var Cst Cst1 Cst2 : Constant . var NEC : NeConstraint .
var V : Variable .
eq (Cst = Cst) = T .                    --- Simplification
eq (Cst1 = Cst2) = F [owise] .          --- Unsatisfiability
eq NEC,NEC = NEC .                      --- Idempotence
eq F,NEC = F .                          --- Zero element
eq F,F = F .                            --- Simplification
eq (V = Cst1),(V = Cst2) = F [owise] . --- Unsatisfiability
```

Note that the conjunction operator `_,_` has identity element `T` and obeys the laws of associativity and commutativity. We express the idempotence property of the operator by a specific equation on variables from the `NeConstraint` subsort `NEC`. A query reduced to `T` represents a successful computation.

Since equations in Maude are run deterministically, all the non-determinism of the original Datalog program has to be embedded into the carried constraints themselves. This means that we need to carry on all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and we implement a new sort called `ConstraintSet`:

```
sorts ConstraintSet eConstraintSet NeConstraintSet .
subsort eConstraintSet NeConstraintSet < ConstraintSet .
subsort NeConstraint TConstraint < NeConstraintSet .
subsort FConstraint < eConstraintSet .
op _;_ : ConstraintSet   ConstraintSet -> ConstraintSet [assoc comm id: F] .
op _;_ : NeConstraintSet ConstraintSet -> NeConstraintSet          [ditto] .
var NECS : NeConstraintSet .
eq NECS ; NECS = NECS .      --- Idempotence
```

It is easy to grasp the intuition behind the different sorts and the subsort relations in the above fragment of Maude code. The operator `_;_` represents the disjunction of constraints. The properties of associativity, commutativity and identity element of `_;_` can be easily expressed by using `ACU` attributes in Maude, thus simplifying the equational specification and achieving better efficiency.

In order to incrementally add new constraints throughout the program execution, we define the composition operator `x` as follows:

```
op _x_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc] .

var NEC NEC1 NEC2 : NeConstraint .
var CS : ConstraintSet . var NECS1 NECS2 : NeConstraintSet .
eq F x CS = F .                                                    --- L-Zro el.
eq CS x F = F .                                                    --- R-Zro el.
eq NEC1 x (NEC2 ; CS) = (NEC1 , NEC2) ; (NEC1 x CS) .              --- L-Dist.
eq (NEC ; NECS1) x NECS2 = (NEC x NECS2) ; (NECS1 x NECS2) .       --- R-Dist.
```

Note that, in order to keep information consistent, automatically trivial constraints are simplified whereas inconsistent ones collapse into an `F` value.

3.2 A Glimpse of the Transformation

In order to mimic the execution order of the subgoals in the body of the DATALOG clauses, the first naïve idea is trying to translate each DATALOG clause into a conditional equation. The execution of these kinds of equations suffers an important penalty within the rewriting machinery of MAUDE that dramatically slows down the overall performance of the computation. In order to obtain better performance, we disregard conditional equations in favor of non-conditional ones and impose an evaluation order by means of some auxiliary *unraveling* [13] functions that stepwisely evaluate each call and propagate the (partially) computed information. We rely on pattern matching to ensure that a call is executed only when the previous one has been solved.

For each DATALOG predicate, we introduce a single equation that represents the disjunction of the possible answers delivered by all the clauses defining that predicate. In the case of predicates defined by facts, each fact can be represented as a `Constraint` term in our setting. Thus, we transform the set of facts defining a particular predicate as a single equation whose *rhs* consists of the disjunction of `Constraint` terms representing each particular DATALOG fact. Considering the running example, facts are transformed to:

```
eq a(T1,T2)   = ((T1 = 'v1) , (T2 = 'v2)) ; ((T1 = 'v1) , (T2 = 'v3)) .
eq vP0(T1,T2) = ((T1 = 'v2) , (T2 = 'h5)) ; ((T1 = 'v3) , (T2 = 'h4)) .
```

In the case of predicates defined by clauses with non-empty body, we generate as many auxiliary functions as different clauses define the DATALOG predicate. For instance, the answers for `vP/2` in the example are the disjunction of the answers of functions `vPc1` and `vPc2`, representing the calls to the first and second DATALOG clauses of the running example, respectively:

```
eq vP(T1,T2) = vPc1(T1,T2) ; vPc2(T1,T2) .
```

The specification for the first clause is given by the function `vPc1`:

```
eq vPc1(T1,T2) = vP0(T1,T2) .
```

The transformation of the second clause into the function `vPc2` is a bit more elaborated since it contains more than one subgoal. Thus, we need an auxiliary

function to impose the execution order. Moreover, it contains an existentially quantified variable which carries information from one subgoal to the other.

```
eq vPc2(T1,T2) = vPc2s2(a(T1,v(T1,T2)), T1 T2) .
eq vPc2s2(((v(T1,T2) = Cst) , C) ; CS, T1 T2) =
         (vP(Cst,T1 T2) x ((v(T1,T2) = Cst) , C)) ; vPc2s2(CS,T1 T2) .
eq vPc2s2(F,T1 T2) = F .
```

As can be observed, `vPc2` calls to `vPc2s2`, whose first argument represents the execution of the first subgoal and the second argument is the list of parameters in the head of the original clause. The pattern in the first argument in the *lhs* of the equation for `vPc2s2` forces the computation of the (partial) answers resulting from the resolution of `a(T1,v(T1,T2))` first in order to proceed. The use of the term `v(T1,T2)`, which represents the existentially quantified variable `Var2` of the original DATALOG program, in the pattern of the equation `vPc2s2` is the key for carrying the computed information from one subgoal to the subsequent subgoals where the variable occurs. The idea is that `vPc2s2` is defined to receive the value of the shared variable on the pattern `((V = Cst) , C) ; CS)`. The recursion over `vPc2s2` is needed because its first argument represents all the possible answers computed by `a(T1,v(T1,T2))`; thus, we recursively compute each solution and use the constraints composition operator to combine them.

In order to execute a query in the transformed program, we call the MAUDE `reduce` command. The query that computes all positions to which each variable can point to can be written in MAUDE as follows:

```
reduce vP(v('variable),v('heap)) .
```

The answers to this query are shown below. The first sentence specifies the term that has been reduced. The second sentence shows the number of rewrites and the execution time that MAUDE invested to perform the reduction. The last sentence, which is written in several lines for the sake of readability, shows the result of the reduction together with its sort.

```
reduce in ANALYSIS : vP(v('v), v('h)) .
rewrites: 39 in 0ms cpu5 (0ms real) (~ rewrites/second)
result NeConstraintSet:
  ((v('h) = 'h4),v('v) = 'v3) ; ((v('h) = 'h5),v('v) = 'v2) ;
  ((v('h) = 'h4),(v('v) = 'v1),v(v('v), v('h)) = 'v3) ;
  (v('h) = 'h5),(v('v) = 'v1),v(v('v), v('h)) = 'v2
```

As expected, four answers were returned: the first two were obtained by the function `vPc1`, whereas the other two were computed by the function `vPc2`.

4 Formal Definition of the Transformation

In this section, we first give a formal description of the new transformation from a DATALOG program into a MAUDE program that delivers the same answers. The correctness and completeness of the transformation is given in Section 4.2.

4.1 The Transformation

Let P be a DATALOG program defining predicate symbols $p_1 \dots p_n$. Before describing the transformation process, we introduce some auxiliary notations. $|p_i|$ is the number of facts or clauses defining the predicate symbol p_i. Following the DATALOG standard, we assume without loss of generality that a predicate p_i is defined only by facts, or only by clauses [9]. The arity of p_i is ar_i.

Let us start by describing the case when predicates are defined by facts. We transform the whole set of facts defining a given predicate symbol p_i into a single equation by means of a disjunction of answer constraints. Formally, for each p_i with $1 \leq i \leq n$ that is defined in the DATALOG program only by facts, we write the following snippet of MAUDE code, where the symbol $c_{i,j,k}$ is the k-th argument of the j-th fact defining the predicate symbol p_i:

$$\begin{array}{l}\texttt{var } \mathtt{T}_{i,1} \dots \mathtt{T}_{i,ar_i} \texttt{ : Term .} \\ \texttt{eq } \mathtt{p}_i(\mathtt{T}_{i,1},\dots,\mathtt{T}_{i,ar_i}) = (\mathtt{T}_{i,1} = c_{i,1,1},\dots,\mathtt{T}_{i,ar_i} = c_{i,1,ar_i}) \texttt{ ; } \dots \\ \qquad\qquad\qquad\qquad \texttt{; } (\mathtt{T}_{i,1} = c_{i,|p_i|,1},\dots,\mathtt{T}_{i,ar_i} = c_{i,|p_i|,ar_i}) \texttt{ .}\end{array}$$

Similarly, our transformation for DATALOG clauses with non-empty body combines in a single equation the disjunction of the calls to all functions representing the different clauses for the considered predicate symbol p_i. For each p_i with $1 \leq i \leq n$ with non empty body, we have the following piece of code:

$$\begin{array}{l}\texttt{var } \mathtt{T}_{i,1} \dots \mathtt{T}_{i,ar_i} \texttt{ : Term .} \\ \texttt{eq } \mathtt{p}_i(\mathtt{T}_{i,1},\dots,\mathtt{T}_{i,ar_i}) = \mathtt{p}_{i,1}(\mathtt{T}_{i,1},\dots,\mathtt{T}_{i,ar_i}) \texttt{ ; } \dots \texttt{ ; } \mathtt{p}_{i,|p_i|}(\mathtt{T}_{i,1},\dots,\mathtt{T}_{i,ar_i}) \texttt{ .}\end{array}$$

Each call $\mathtt{p}_{i,j}$ with $1 \leq j \leq |p_i|$ produces the answers computed by the j-th clause of the predicate symbol. Now we need to define how each of these clauses is transformed. Notation $\tau^a_{i,j,s,k}$ denotes the name of the variable or constant symbol appearing in the k-th argument of the s-th subgoal in the j-th clause defining the i-th predicate of the original DATALOG program. When $s = 0$, then the function refers to the arguments in the head of the clause.

Let us start by considering the case of just one subgoal in the body. We define the function $\tau^p_{i,j,s}$, which returns the predicate symbol that appears in the s-th subgoal of the j-th clause that defines the i-th predicate in the DATALOG program. For each one-subgoal clause, we get the following transformation:

$$\texttt{eq } \mathtt{p}_{i,j}(\tau^a_{i,j,0,1},\dots,\tau^a_{i,j,0,ar_i}) = \tau^p_{i,j,1}(\tau^a_{i,j,1,1},\dots,\tau^a_{i,j,1,r}) \texttt{ .}$$

In the equation, r is the arity of the predicate $\tau^p_{i,j,1}$.

In the case where more than one subgoal appears in the body of a clause, we want to impose a left-to-right evaluation strategy. We use auxiliary functions defined by patterns to force such an execution order. Specifically, we set that a subgoal cannot be invoked until the variables in its arguments that also occur in previous subgoals have been instantiated. We call these variables *linked* variables.

Definition 2 (Linked variable). *A variable is called* linked variable iff *it does not occur in the head of a* DATALOG *clause, and occurs in two or more subgoals.*

Definition 3 (Function linked). *Let C be a* DATALOG *clause. Then the function* linked*(C) is the function that returns the list of pairs containing a linked variable in the first component, and the list of positions where such a variable occurs in the body of the clause in the second component*[4]*.*

For example, given the DATALOG clause

$$C = \texttt{p(X1,X2) :- p1(X1,X3), p2(X3,X4), p3(X4,X2).}$$

we have that linked(C) = `[(X3,[1.2,2.1]),(X4,[2.2,3.1])]`

Now we define the notion of *relevant* linked variables for a given subgoal, namely the linked variables of a subgoal that also appear in a previous subgoal.

Definition 4 (Relevant linked variables). *Given a clause C and an integer number n, we define the function* relevant *that returns the variables that are common for the n-th subgoal and some previous subgoal:*

$$\mathit{relevant}(n, C) = \{X \mid (X, LX) \in \mathsf{linked}(C), \text{ and there exists } m < n, \exists j \text{ s.t. } m.j \in LX\}$$

Note that, similarly to [16], we are not marking the input/output positions of predicates, as required in more traditional transformations. We are just identifying the variables whose values must be propagated in order to evaluate the subsequent subgoals following the evaluation strategy.

Now we are ready to address the problem of transforming a clause with more than one subgoal (and maybe existentially quantified variables) into a set of equations. Intuitively, the main function initially calls to an auxiliary function that undertakes the execution of the first subgoal. We have as many auxiliary functions as subgoals in the original clause. Also, in the *rhs* of the auxiliary functions, the execution order of the successive subgoals is controlled by passing the results of each subgoal as a parameter to the subsequent function call.

Let the function $\mathtt{p}_{i,j}$ generate the solutions calculated by the j-th clause of the predicate symbol p_i. We state that $\mathtt{ps}_{i,j,s}$ represents the auxiliary function corresponding to the s-th subgoal of the j-th clause defining the predicate p_i. Then, for each clause, we have the following translation, where the variables $\mathtt{X}_1 \ldots \mathtt{X}_N$ of each equation are calculated by the function relevant$(s,$linked$(\mathit{clause}(i,j)))$[5] and transformed into the corresponding MAUDE terms. In the equations below, N stands for the number of relevant variables of the subgoal being transformed.

The equation for $\mathtt{p}_{i,j}$ below reduces the considered DATALOG predicate to a call to the first auxiliary function that calculates the (partial) answers for the second subgoal by first computing the answers from the first subgoal $\tau^p_{i,j,1}$ in its first argument. The second argument of the equations represents the list of terms in the initial predicate call that, together with the information retrieved from Definitions 3 and 4, allow us to correctly build the patterns and function calls during the transformation.

$$\texttt{eq } \mathtt{p}_{i,j}(\tau^a_{i,j,0,1}, \ldots, \tau^a_{i,j,0,ar_i}) = \mathtt{ps}_{i,j,2}(\tau^p_{i,j,1}(\tau^a_{i,j,1,1}, \ldots, \tau^a_{i,j,1,r}),\ \tau^a_{i,j,0,1} \ldots \tau^a_{i,j,0,ar_i}) \ .$$

[4] Positions extend to goals in the natural way.

[5] $\mathit{clause}(i,j)$ represents the j-th DATALOG clause defining the predicate symbol p_i.

Then, for each auxiliary (unraveling) function, we declare as many constants as there are relevant variables in the corresponding subgoal. The left hand side of the equation for this auxiliary function is defined with patterns that adjust the relevant variables to the values already computed by the execution of a previous subgoal. Note that we may have more assignments in the constraint, which is represented by C, and that we may have more possible solutions in CS. The auxiliary equation $\texttt{ps'}_{i,j,s}$ takes each possible (partial) solution and combines it with the solutions given by the s-th subgoal in the clause (whose predicate symbol is $\tau^p_{i,j,s}$). Note that we propagate the instantiation of the relevant variables by means of a substitution.

$\texttt{var C}_1 \ldots \texttt{C}_N \texttt{ : Constant .}$
$\texttt{var NECS : NeConstraintSet .}$

$\texttt{eq ps}_{i,j,s}\texttt{(NECS , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) = ps}_{i,j,s+1}\texttt{(ps'}_{i,j,s}\texttt{(NECS , T}_1 \ldots \texttt{T}_{ar_i}\texttt{), T}_1 \ldots \texttt{T}_{ar_i}\texttt{) .}$
$\texttt{eq ps}_{i,j,s}\texttt{(F , LL) = F .}$
$\texttt{eq ps'}_{i,j,s}\texttt{(((X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C) ; CS) , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) =}$
$\quad \texttt{((}\tau^p_{i,j,s}\texttt{(}\tau^a_{i,j,s,1}\texttt{,...,}\tau^a_{i,j,s,r}\texttt{)[X}_1\texttt{\C}_1\texttt{,...,X}_N\texttt{\C}_N\texttt{]) x (X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C)) ;}$
$\quad \texttt{ps'}_{i,j,s}\texttt{(CS , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) .}$
$\texttt{eq ps'}_{i,j,s}\texttt{((T ; CS), T}_1 \ldots \texttt{T}_{ar_i}\texttt{) = }\tau^p_{i,j,s}\texttt{(}\tau^a_{i,j,s,1}\texttt{,...,}\tau^a_{i,j,s,r}\texttt{) ; ps'}_{i,j,s}\texttt{(CS,T}_1 \ldots \texttt{T}_{ar_i}\texttt{) .}$
$\texttt{eq ps'}_{i,j,s}\texttt{(F , LL) = F .}$

The equation for the last subgoal in the clause is slightly different, since we need not invoke the following auxiliary function. Assuming that g denotes the number of subgoals in a clause, we define

$\texttt{eq ps}_{i,j,g}\texttt{(((X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C) ; CS) , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) =}$
$\quad \texttt{((}\tau^p_{i,j,g}\texttt{(}\tau^a_{i,j,g,1}\texttt{,...,}\tau^a_{i,j,g,r}\texttt{)[X}_1\texttt{\C}_1\texttt{,...,X}_N\texttt{\C}_N\texttt{]) x (X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C)) ;}$
$\quad \texttt{ps}_{i,j,g}\texttt{(CS , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) .}$
$\texttt{eq ps}_{i,j,g}\texttt{((T ; CS) , T}_1 \ldots \texttt{T}_{ar_i}\texttt{) = }\tau^p_{i,j,g}\texttt{(}\tau^a_{i,j,g,1}\texttt{,...,}\tau^a_{i,j,g,r}\texttt{) ; ps}_{i,j,g}\texttt{(CS,T}_1 \ldots \texttt{T}_{ar_i}\texttt{) .}$
$\texttt{eq ps}_{i,j,g}\texttt{(F , LL) = F .}$

Finally, we define the transformation for the DATALOG query $q(X_1, \ldots, X_n)$ (where X_i, $1 \leq i \leq n$ are DATALOG variables or constants) as the MAUDE code $\texttt{q(}\tau^q_1\texttt{,...,}\tau^q_n\texttt{)}$, where τ^q_i, $1 \leq i \leq n$ is the transformation of the corresponding X_i.

4.2 Correctness of the Transformation

We have defined a transformation from DATALOG programs into MAUDE programs in such a way that the normal form computed for a term of the ConstraintSet sort represents the set of computed answers for a query of the original DATALOG program. In this section, we show that the transformation is sound and complete w.r.t. the observable computed answers.

We first introduce some notation. Let CS be a ConstraintSet of the form $\texttt{C}_1$; $\texttt{C}_2$; ... ; $\texttt{C}_n$ where each $\texttt{C}_i$, $i \geq 1$ is a Constraint in normal form ($\texttt{C}_1$ = $\texttt{Cst}_1$, ... , $\texttt{C}_m$ = $\texttt{Cst}_m$), and let V be a list of variables. The restriction of the constraint $\texttt{C}_i$ to the variables in V is written as $\texttt{C}_i|_V$. We extend the notion to sets of constraints in the natural way, and denote it as $\texttt{CS}|_V$. Given two terms

t and t', we write $t \rightarrow_S^* t'$ when there exists a rewriting sequence from t to t' in the MAUDE program S. Also, $var(t)$ is the set of variables occurring in t.

Now we define a suitable notion of *(rewriting) answer constraint*:

Definition 5 (Answer Constraint Set). *Given a* MAUDE *program S as described in this work and an input term t, we say that the* answer constraint set *computed by $t \rightarrow_S^* \texttt{CS}$ is $\texttt{CS}|_{var(t)}$.*

There is a natural isomorphism between the equational constraint C and an idempotent substitution $\theta = \{X_1/C_1, X_2/C_2, \ldots, X_n/C_n\}$, which is given by the following: $\texttt{C}$ is equivalent to θ *iff* $(\texttt{C} \Leftrightarrow \hat{\theta})$, where $\hat{\theta}$ is the equational representation of θ. Abusing notation, given a disjunction $\texttt{CS}$ of equational constraints and a set of idempotent substitutions $(\Theta = \cup_{i=1}^{n} \theta_i)$, we define $\Theta \equiv \texttt{CS}$ *iff* $\texttt{CS} \Leftrightarrow \bigvee_{i=1}^{n} \hat{\theta}_i$

Next, we prove that, for a given query and DATALOG program, each answer constraint set computed for the corresponding input term in the transformed MAUDE program is equivalent to the set of computed answers of the original DATALOG program. The proof of this result is given in [1].

Theorem 1 (Correctness and completeness). *Consider a* DATALOG *program P together with the query q. Let $\mathcal{T}(P)$ be the corresponding, transformed* MAUDE *program, and let $\mathcal{T}_g(q)$ be the corresponding, transformed input term. Let Θ be the set of computed answers of P for the query q, and let $\texttt{CS}|_{var(\mathcal{T}_g(q))}$ be the answer constraint set computed by $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^* \texttt{CS}$. Then, $\Theta \equiv \texttt{CS}|_{var(\mathcal{T}_g(q))}$.*

5 Experimental Results

This section reports on the performance of our prototype, called DATALAUDE[6], implementing the transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [2]; then, we evaluate the performance of our prototype by comparing it to three state-of-the-art DATALOG solvers. All the experiments were conducted using JAVA JRE 1.6.0, JOEQ version 20030812, on a Mobile AMD Athlon XP2000+ (1.66GHz) with 700 Megabytes of RAM, running Ubuntu Linux 8.04.

5.1 Comparison w.r.t. a Previous Rewriting-Based Implementation

We implemented several transformations from DATALOG programs to MAUDE programs before developing the one presented in this paper [2]. The first attempt consisted of a one-to-one mapping from DATALOG rules into MAUDE conditional rules. Then, in order to get rid of all the non-determinism caused by conditional equations and rules in MAUDE, we restricted our transformation to produce only unconditional equations. In the following, we briefly present the results obtained by using the rule-based approach, the equational-based approach, and the equational-based approach improved by using the memoization capability of

[6] `http://www.dsic.upv.es/users/elp/datalaude`

MAUDE [7]. MAUDE is able to store each call to a given function (in the running example `vP(X,Y)`) together with its normal form. Thus, when MAUDE finds a memoized call it doesn't reduce it but it just replaces it with its normal form, saving a great number of rewrites.

Table 1 shows the resolution times of the three selected versions. The sets of initial DATALOG facts (`a/2` and `vP0/2`) are extracted by the JOEQ compiler from a JAVA program (with 374 lines of code) implementing a tree visitor. The DATALOG clauses are those of our running example: a simple context-insensitive inclusion-based pointer analysis. The evaluated query is `?- vP(Var,Heap).`, i.e., all possible answers that satisfy the predicate `vP/2`.

Table 1. Number of initial facts (`a/2` and `vP0/2`) and computed answers (`vP/2`), and resolution time (in seconds) for the three implementations

a/2	vP0/2	VP/2	rule-based	equational	equational+memo
100	100	144	6.00	0.67	0.02
150	150	222	20.59	2.23	0.04
200	200	297	48.48	6.11	0.10
403	399	602	382.16	77.33	0.47
807	1669	2042	4715.77	1098.64	3.52

The results obtained with the equational implementation are an order of magnitude better than those obtained by the naïve transformation based on rules. These results are due to the fact that the backtracking associated to the nondeterministic evaluation penalizes the naïve version. It can also be observed that using memoization allows us to gain another order of magnitude in execution time with respect to the basic equational implementation. These results confirm that the equational implementation fits our program analysis purposes better, and provides a versatile and competitive DATALOG solver as compared to other implementations of DATALOG.

5.2 Comparison w.r.t. Other State-of-the-Art DATALOG Solvers

The same sets of initial facts were used to compare our prototype (the equational-based version with memoization) with three state-of-the-art DATALOG solvers, namely XSB 3.2, DATALOG 1.4, and IRIS 0.58 [7]. Average resolution times of three runs for each solver are shown in Figure 1.

In order to evaluate the performance of our implementation with respect to the other DATALOG solvers, only resolution times are presented in Figure 1 since the compared implementations are quite different in nature. This means that initialization operations, like loading and compilation, are not taken into account in the results. Our experiments conclude that DATALAUDE performs similarly

[7] `http://xsb.sourceforge.net`, `http://datalog.sourceforge.net` and `http://iris-reasoner.sourceforge.net`

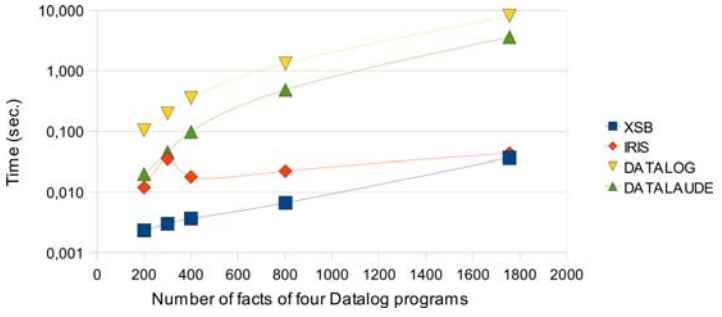


Fig. 1. Average resolution times of four DATALOG solvers (logarithmic time)

to optimized deductive database systems like DATALOG 1.4, which is implemented in C, although it is slower than XSB or IRIS. Therefore, under a suitable transformation scheme such as the equational implementation extended with memoization, MAUDE is able to process a large number of equations extracted from statically analyzed, real JAVA programs. However, our purpose is not to produce the faster DATALOG solver ever, but to provide a tool that supports sophisticated analyses with reasonable performance in a clean way.

5.3 Analyzing Java Programs with Reflection

Addressing reflection is considered a difficult problem in the static analysis of JAVA programs, which is generally handled in an unsound or ad-hoc manner [11]. Reflection in JAVA is a powerful technique that is used when a program needs to examine or modify the runtime behavior of applications running in the JAVA virtual machine. For example, by using reflection, it is possible to write to object fields and invoke methods that are not known at compile time. JAVA provides a set of methods to handle reflection. These methods are found in java.lang.reflect.

In Figure 2 we show a simple example. We define a class `PO` with two fields: `c1` and `c2`. In the `Main` class, an object `u` of class `PO` is created by using the constructor method `new`, which assigns the empty string to the two fields of `u`. Then, `r` is defined as a field of a class, specifically, as the field `c1` of an object of class `PO` since `v` stores the value `"c1"`. The sentence `r.set(u, w)` states that `r` is the field object `c1` of `u`, and its value is that of `w`, i.e., `"c2"`. Finally, the last instruction sets the new value of `v` to the value of `u.c1`, i.e., `"c2"`.

A pointer flow-insensitive analysis of this program would tell us that `r` may point not only to the field object `u.c1`, but also `u.c2` since `v` in the argument of the reflective method `getField` may be assigned both to string `"c1"` and `"c2"`.

The key point for the reflective analysis is the fact that we don't have all the basic information for the points-to analysis at the beginning of the computation. In fact, the variables that occur in the methods handling reflection may generate new basic information. A sound proposal for handling JAVA reflection in DATALOG analyses is proposed in [11]. It essentially consists in first

```
class PO {
      PO (String c1, String c2) {
         this.c1 = c1;
         this.c2 = c2;
      }
      public String c1;
      public String c2;
}
```

```
public class Main {
  public static void main(String[] args) {
  PO u = new PO("","");
  String v = "c1";
  String w = "c2";
  java.lang.reflect.Field r = PO.class.getField(v);
  r.set(u, w);
  v = u.c1;
  } }
```

Fig. 2. JAVA reflection example

annotating the DATALOG program and subsequently transforming it by means of an external (to DATALOG) engine. As in [11], we assume we know the name of the methods and objects that may be used in the invocations. In our approach, DATALOG rules are transformed into MAUDE rules. Then, the MAUDE reflection capability is used during the analysis to automatically generate the rules that represent new deduced information without resorting to any ad-hoc notation or external artifact.

Rewriting logic is reflective in a precise mathematical way: there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent (as data) any finitely presented rewrite theory $\mathcal{R}$ in $\mathcal{U}$ (including $\mathcal{U}$ itself), and then mimic the behavior of $\mathcal{R}$ in $\mathcal{U}$. The fact that rewriting logic is a reflective logic and the fact that MAUDE effectively supports reflective rewriting logic computation make reflective design (in which theories become data at the metalevel) ideally suited for manipulation tasks in MAUDE.

MAUDE's reflection is systematically exploited in our tool. On one hand, we can easily define new rules to be included in the specification by manipulating term meta-representations of rules and modules. On the other hand, by virtue of our reflective design, our metatheory of program analysis (which includes a common fixpoint infrastructure) is made accessible to the user who writes a particular analysis in a clear and principled way.

We have endowed our prototype implementation with the capability to carrying on reflection analysis for JAVA. The extension essentially consists of a module at the MAUDE meta-level that implements a generic infrastructure to deal with reflection. Figure 3 shows the structure of a typical reflection analysis to be run in our tool.

The static analysis is specified in two object-level modules, a *basic module* and a *reflective module*, that can be written in either MAUDE or DATALOG, since DATALOG analyses are automatically compiled into MAUDE code. The *basic program analysis (PA)* module contains the rules for the classical analysis (that neglects reflection) whereas the *reflective program analysis* module contains the part of the analysis dealing with the reflective components of the JAVA program. At the meta-level, the *solver* module consists of a generic fixpoint algorithm that feds the reflective module with the information inferred by the basic analysis. Then, MAUDE rules encoding the new inferred information are synthesized by the reflective analysis and added to the basic module in order to infer new

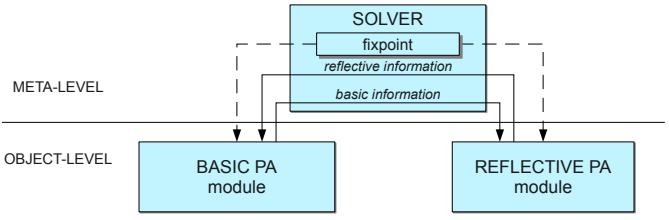


Fig. 3. The structure of the reflective analysis

information, until a fixpoint is reached. For the points-to analysis with reflection, the reflective and basic modules contain 11 rules each, whereas the generic solver is written in just 50 rules. More details can be found in [1].

6 Conclusions

In this work, we have defined and implemented a transformation from definite Datalog programs into Maude programs in the context of Datalog-based static analysis. We have formalized and proved the correctness of the transformation, and we have compared our implementation to standard Datalog solvers. We confirmed that Maude is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of Java programs. By taking advantage of Maude's reflective design, we have also demonstrated how it is possible to perform efficient reflection analyses of Java programs in Maude that are, at one time, declarative, accurate, and sound.

As future work, we plan to extend our transformation to stratified Datalog programs in order to support richer analyses. We also plan to explore the impact of adapting to our context more sophisticated Datalog optimizations [10].

Acknowledgements. We are grateful to Adam Kepa for his valuable contribution to the experiments.

References

1. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Defining Datalog in Rewriting Logic. Tech. Rep. DSIC-II/07/09, DSIC, Technical University of Valencia (2009)
2. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Implementing Datalog in Maude. In: Proc. of I Taller de Programación Funcional (TPF 2009), 15–22 (2009)
3. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Using Datalog and Boolean Equation Systems for Program Analysis. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 215–231. Springer, Heidelberg (2009)
4. Andersen, H.R.: Model checking and boolean graphs. Theoretical Computer Science 126(1), 3–30 (1994)
5. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986), pp. 1–15. ACM Press, New York (1986)

6. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer, Heidelberg (1990)
7. Clavel, M., Durán, F., Ejer, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Hill, P.M., Lloyd, J.W.: Analysis of Meta-Programs. In: Proc. of the First International Workshop on Meta-Programming in Logic (META 1988), pp. 23–51 (1988)
9. Leeuwen, J. (ed.): Formal Models and Semantics, vol. B. Elsevier, The MIT Press (1990)
10. Liu, Y., Stoller, S.: From Datalog Rules to Efficient Programs with Time and Space Guarantees. ACM Transactions on Programming Languages and Systems 31(6) (2009)
11. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 139–160. Springer, Heidelberg (2005)
12. Marchiori, M.: Logic Programs as Term Rewriting Systems. In: Rodríguez-Artalejo, M., Levi, G. (eds.) ALP 1994. LNCS, vol. 850, pp. 223–241. Springer, Heidelberg (1994)
13. Marchiori, M.: Unravelings and ultra-properties. In: Hanus, M., Rodríguez-Artalejo, M. (eds.) ALP 1996. LNCS, vol. 1139, pp. 107–121. Springer, Heidelberg (1996)
14. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
15. Reddy, U.: Transformation of Logic Programs into Functional Programs. In: Proc. of the Symposium on Logic Programming (SLP 1984), pp. 187–197. IEEE Computer Society Press, Los Alamitos (1984)
16. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 177–193. Springer, Heidelberg (2007)
17. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. In: The New Technologies, vol. I, II. Computer Science Press, Rockville (1989)
18. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Proc. of the 1st International Conference on Expert Database Systems (EDS 1986), pp. 253–267 (1986)
19. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)

Author Index