TECHNICAL REPORT

TR 2009/04

ISSN 0874-338X

Preliminary Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation LOPSTR 2009. 9-11 September 2009 Coimbra, Portugal

Editores: Danny De Schreye, Pedro Quaresma

CISUC

Centro de Informática e Sistemas da Universidade de Coimbra

TR 2009/04 **ISSN 0874-338X**

Preliminary Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation LOPSTR 2009.
9-11 September 2009
Coimbra, Portugal

Editores: Danny De Schreye, Pedro Quaresma

Coimbra, Portugal 2009

Preliminary Proceedings of the 19th International Symposium on
Logic-Based Program Synthesis and Transformation
LOPSTR 2009
9-11 September 2009
Coimbra, Portugal

Preface

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. LOPSTR is open to contributions in logic-based program development in any language paradigm. LOPSTR has a reputation for being a lively, friendly forum for presenting and discussing work in progress.

The book contains the preliminary proceedings of the LOPSTR09 symposium. Formal proceedings are produced only after the symposium so that authors can incorporate feedback in the published papers. The proceedings of LOPSTR09 will be published in the Lecture Notes in Computer Science series of Springer-Verlag.

LOPSTR09 is held in Coimbra, Portugal. It is co-located with PPDP 2009 (International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming) and CSL 2009 (EACSL Annual Conference on Computer Science Logic). Previous symposia were held in Valencia, Lyngby, Venice, London, Verona, Uppsala, Madrid, Paphos, London, Venice, Manchester, Leuven, Stockholm, Utrecht, Pisa, Louvain-la-Neuve, and Manchester (two years in row).

From the submitted papers, 16 high quality papers were selected for presentation at the conference. These appear in this book. In addition, there is an invited talk by German Vidal. The abstract of this talk also appears in this proceedings.

I want to thank the program committee members, Slim Abdennadher, Maria Alpuente Frasnedo, Roberto Bagnara, John Gallagher, Robert Glueck, Michael Hanus, Reinhard Kahle, Andy King, Michael Leuschel, Fabio Martinelli, Fred Mesnard, Mario Ornaghi, German Puebla, Sabina Rossi, Josep Silva, Peter Schneider-Kamp, Tom Schrijvers, Petr Stepanek and Wim Vanhoof for their thorough work in evaluating the submitted papers.

I want to thank the organizers,Ana Almeida, Pedro Quaresma and Reinhard Kahle for their great job in preparing the conference.

I wish all participants a very fruitful and pleasant meeting.
Danny De Schreye
Program Chair

Content.

Rewriting:
- Bernd Brassel and Rudolf Berghammer: Functional (Logic) Programs as Equations over Order-Sorted Algebras
- M. Alpuente, M. A. Feliu, C. Joubert and A. Villanueva: Defining Datalog in Rewriting Logic

Towards Scalable Partial Evaluation of Declarative Programs*

Germán Vidal

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

Introduction

Partial evaluation is a well-known technique for program specialization [3]. 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 and by the *Generalitat Valenciana* under grant GVPRE/2008/001.

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. Our recent work [1, 4–7] shows that this drawback can be overcome by using instead a *strong* termination analysis [2], 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.

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, e.g., by using the information from a termination analysis for a particular selection or evaluation strategy—which is only run once—, by means of user provided annotations, by adding *online* annotations, etc. Therefore, although there is ample room for improving accuracy, we consider our approach a promising framework for developing scalable partial evaluators for declarative programs. 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 term rewrite systems, (first-order) functional programs, logic programs, and functional logic programs.

References

1. G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Logic-Based Program Synthesis and Transformation. Revised and Selected Papers from LOPSTR'06*, pages 60–76. Springer LNCS 4407, 2007.
2. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
3. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
4. M. Leuschel, S. Tamarit, and and G. Vidal. Fast and Accurate Strong Termination Analysis with an Application to Partial Evaluation. In *Proc. of the 18th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'09)*, to appear in Springer LNCS, 2009.
5. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Logic-based Program Synthesis and Transformation. Revised and selected papers from LOPSTR'08*, pages 119–134. Springer LNCS 5438, 2009.
6. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'05)*, pages 228–239. ACM Press, 2005.
7. G. Vidal. 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'07)*, pages 51–60. ACM Press, 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 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 system for monadic ω-programs. This proof system can be encoded as a logic program which always terminates if the evaluation is performed by tabled resolution. Our verification algorithm has essentially the same time complexity of the best algorithms known in the literature.

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 [5]. 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 [3] for a survey). Most of the known model checking algorithms for CTL* either combine model checking algorithms for CTL and LTL [3], or use techniques based on translations to automata on infinite trees [8].

In this paper we extend to CTL* a method proposed in [13] for LTL. We encode the satisfaction relation of a CTL* formula φ with respect to a Kripke structure $\mathcal{K}$ by means of a 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 on infinite lists 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* [16] which for ω-programs is defined in terms of a non-Herbrand interpretation for infinite lists.

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 which defines unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules similar to those presented in [7,19,20] according to a strategy which is a variant of the strategy for the elimination of multiple occurrences of variables [14]. Similarly to [7,19], 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 set of proof rules for monadic ω-programs which are sound and complete with respect to the perfect model semantics. Moreover, those rules can be encoded in a straightforward way as a logic program which always terminates if it is evaluated by using tabled resolution [2,18].

Our verification method based on very general transformation techniques, has essentially the same time complexity as the algorithms based on the techniques presented in [3,8].

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 transformation rules and the strategy which allows us to transform an ω-program into a monadic ω-program. In Section 3.2 we present the proof rules for monadic ω-programs and the encoding of those proof rules as clauses of a logic program, and in Section 3.3 we study the computational complexity of our verification algorithm. Finally, in Section 4 we discuss the 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 a formula *Prop* such that φ is true in $\mathcal{K}$, written $\mathcal{K} \models \varphi$, iff *Prop* is true in the perfect model of $P_{\mathcal{K},\varphi}$, written $M(P_{\mathcal{K},\varphi}) \models \mathit{Prop}$. Thus, the problem of checking whether or not $\mathcal{K} \models \varphi$ holds, also called the problem of model checking φ with respect to $\mathcal{K}$, is reduced to the problem of testing whether or not $M(P_{\mathcal{K},\varphi}) \models \mathit{Prop}$ holds.

Now we briefly recall the definition of the temporal logic CTL* (see [3] 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) $\lambda : \Sigma \rightarrow \mathcal{P}(\mathit{Elem})$ 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 *state formula* φ_s or a *path formula* φ_p defined as follows:

$$\begin{array}{ll} (\textit{state formulas}) & \varphi_s ::= d \mid \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \mathsf{E}\,\varphi_p \\ (\textit{path formulas}) & \varphi_p ::= \varphi_s \mid \neg\varphi_p \mid \varphi_p \wedge \varphi_p \mid \mathsf{X}\,\varphi_p \mid \varphi_p\,\mathsf{U}\,\varphi_p \end{array}$$

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

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

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 $\mathcal{K}, \pi \models \varphi$ is inductively defined as follows:

$$\begin{array}{lll} \mathcal{K}, \pi \models d & \text{iff} & \pi = [a_0, a_1, \ldots] \text{ and } d \in \lambda(a_0) \\ \mathcal{K}, \pi \models \neg\,\varphi & \text{iff} & \mathcal{K}, \pi \not\models \varphi \\ \mathcal{K}, \pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \mathcal{K}, \pi \models \varphi_1 \text{ and } \mathcal{K}, \pi \models \varphi_2 \\ \mathcal{K}, \pi \models \mathsf{E}\,\varphi & \text{iff} & \pi = [a_0, a_1, \ldots] \text{ and there exists a computation path } \pi' \\ & & \text{from } a_0 \text{ such that } \mathcal{K}, \pi' \models \varphi \\ \mathcal{K}, \pi \models \mathsf{X}\,\varphi & \text{iff} & \mathcal{K}, \pi_1 \models \varphi \\ \mathcal{K}, \pi \models \varphi_1\,\mathsf{U}\,\varphi_2 & \text{iff} & \text{there exists } i \geq 0 \text{ such that } \mathcal{K}, \pi_i \models \varphi_2 \\ & & \text{and, for all } 0 \leq j < i,\ \mathcal{K}, \pi_j \models \varphi_1. \end{array}$$

Given a *state formula* φ, we say that $\mathcal{K}$ is a *model* of φ, written $\mathcal{K} \models \varphi$, 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 $\mathcal{K}, \pi \models \varphi$ holds.

The above definition of the satisfaction relation for CTL* formulas is a shorter, yet equivalent, version of the usual definition given in the literature [3].

In order to encode the satisfaction relation for CTL* formulas as a logic program, in the next section we will introduce 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 an infinite list whose head is H and whose tail is the infinite list T.

We assume that $\mathcal{L}_\omega$ is a typed language [11] with 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

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 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 *$\omega$-program* is a logic program constructed as usual (see, for instance, [11]) 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'.

If f is a term or a formula, then by $vars(f)$ we denote the set of variables occurring in f. By $\forall(\varphi)$ and $\exists(\varphi)$ we denote, respectively, the *universal closure* and the *existential closure* of the formula φ.

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) to the types `fterm`, `state`, and `ilist`, I assigns the sets HU, Σ, and Σ^ω, respectively, (ii) to the function symbol $[\_|\_]$, I assigns 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) to every n-ary predicate $p \in Pred$ of type $\tau_1 \times \ldots \times \tau_n$, I assigns 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, if 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 $I \vDash \forall(\gamma)$. Similarly to the case of logic programs, we can define (*locally*) *stratified* ω-programs and we have that every (locally) stratified ω-program P has a unique *perfect ω-model* (or *perfect model*, for short) which we denote $M(P)$ [1,16] (in Example 1 below we present the construction of the perfect model of an ω-program).

Definition 3 (Monadic ω-Programs). A *monadic* ω-clause is of the form:

$$p_0([s|X_0]) \leftarrow p_1(X_1) \wedge \ldots \wedge p_k(X_k) \wedge \neg p_{k+1}(X_{k+1}) \wedge \ldots \wedge \neg p_m(X_m)$$

where: (i) $0 \leq k \leq m$, (ii) for $i = 0, \ldots, m$, $p_i \in Pred$, (iii) s is a constant of type `state`, (iv) X_0 is a variable of type `ilist`, and (v) $X_1, \ldots, X_k, X_{k+1}, \ldots, X_m$ are *distinct* variables of type `ilist`. A *monadic ω-program* is a stratified, finite set of monadic ω-clauses.

Note that in Definition 3 the predicate symbols $p_0, p_1, \ldots, p_m$ are *not* necessarily distinct and X_0 may be one of the variables in $\{X_1, \ldots, X_m\}$.

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 $\ell : Pred \rightarrow \mathbb{N}$ such that $\ell(p) = 2$, $\ell(q) = 1$, and $\ell(r) = 0$. The value of the predicate of an atom under ℓ is called the *level* of that atom. The perfect model $M(P)$ is constructed by increasing levels of ground atoms. We start from ground atoms of level 0, that is, 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 ground atoms of level 1, that is, 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 ground atoms of level 2, that is, 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* 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 unary predicate *prop* that encodes the property φ to be verified, in the sense that $prop(\pi)$ holds iff the first state of π 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. $prop(X) \leftarrow sat([s_0|X],\varphi)$
2. $path(X) \leftarrow \neg\, notpath(X)$
3. $notpath([S_1,S_2|X]) \leftarrow \neg\, tr(S_1,S_2)$
4. $notpath([S|X]) \leftarrow notpath(X)$
5. $sat([S|X],F) \leftarrow elem(F,S)$
6. $sat(X,not(F)) \leftarrow \neg\, sat(X,F)$
7. $sat(X,and(F_1,F_2)) \leftarrow sat(X,F_1) \wedge sat(X,F_2)$
8. $sat([S|X],e(F)) \leftarrow path([S|Y]) \wedge sat([S|Y],F)$
9. $sat([S|X],x(F)) \leftarrow sat(X,F)$
10. $sat(X,u(F_1,F_2)) \leftarrow sat(X,F_2)$
11. $sat([S|X],u(F_1,F_2)) \leftarrow sat([S|X],F_1) \wedge sat(X,u(F_1,F_2))$

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

Clause 1 of Definition 4 states that the property φ holds for an infinite list whose first state is s_0. Clauses 2–4 stipulate 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 3 and 4 stipulate that $notpath(X)$ holds iff there exist two

consecutive elements a_i and a_{i+1} on X such that $(a_i, a_{i+1}) \notin \rho$. Clauses 5–11 define the satisfaction relation $sat(X, \varphi)$ for any infinite list X and CTL* formula φ. The definition is given by cases on the structure of φ.

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 (here, for any CTL* formula ψ, we denote by $|\psi|$ the number of occurrences of function symbols in ψ): for all states $a, a_1, a_2 \in \Sigma$, for all infinite lists $\pi \in \Sigma^\omega$, for all elementary properties $d \in Elem$, and for all CTL* formulas ψ, (i) $\sigma(prop(\pi)) = |\varphi| + 1$, (ii) $\sigma(path(\pi)) = 2$, (iii) $\sigma(notpath(\pi)) = 1$, (iv) $\sigma(tr(a_1, a_2)) = \sigma(elem(d, a)) = 0$, (v) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, and (vi) for every ground atom A, $\sigma(\neg A) = \sigma(A) + 1$.

Example 2. Let us consider the set $Elem = \{a, b, tt\}$ of elementary properties, where tt is the elementary property which holds in all states, and 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 $\varphi = \mathsf{E} \neg (tt\, \mathsf{U} \neg a) \wedge \mathsf{E} \neg (tt\, \mathsf{U} \neg (tt\, \mathsf{U}\, b))$, which is usually abbreviated as $\mathsf{EG}\, a \wedge \mathsf{EGF}\, b$, where: (i) $\mathsf{F}\, \psi$ (*eventually* ψ) stands for $tt\, \mathsf{U}\, \psi$, and (ii) $\mathsf{G}\, \psi$ (*always* ψ) stands for $\neg \mathsf{F} \neg \psi$. The encoding program $P_{\mathcal{K},\varphi}$ is as follows:

$prop(X) \leftarrow sat([s_0|X], and(e(not(u(tt, not(a)))), e(not(u(tt, not(u(tt, b)))))))$
$path(X) \leftarrow \neg\, notpath(X)$
$notpath([S_1, S_2|X]) \leftarrow \neg\, tr(S_1, S_2)$
$notpath([S|X]) \leftarrow notpath(X)$
$tr(s_0, s_0) \leftarrow$ $tr(s_0, s_1) \leftarrow$ $tr(s_1, s_1) \leftarrow$ $tr(s_1, s_2) \leftarrow$ $tr(s_2, s_1) \leftarrow$
$elem(a, s_0) \leftarrow$ $elem(b, s_1) \leftarrow$ $elem(a, s_2) \leftarrow$ $elem(tt, S) \leftarrow$

together with clauses 5–11 of Definition 4 defining the predicate *sat*.

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) $M(P_{\mathcal{K},\varphi}) \models \exists X\, prop(X)$. Now, if we denote the statement $\exists X\, prop(X)$ by *Prop*, the correctness of $P_{\mathcal{K},\varphi}$ is stated by 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 set of proof rules 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*, *instantiation*, *positive* and *negative unfolding*, *clause deletion*, *positive* and *negative folding* (see, for instance, [7,19,20]). These rules are applied according to a transformation strategy which is a variant of the one for eliminating multiple occurrences of variables [14].

Our strategy starts off from the clause γ_1: $prop(X) \leftarrow sat([s_0|X], \varphi)$ in $P_{\mathcal{K},\varphi}$ (see clause 1 in Definition 4) and iteratively applies two procedures: (i) the *unfold* procedure, and (ii) the *define-fold* procedure. At each iteration, the set *InDefs* of clauses, which is initialized to $\{\gamma_1\}$, is transformed into a set *Es* 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. Our strategy terminates when no new auxiliary clauses are introduced.

The Transformation 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 $M(P_{\mathcal{K},\varphi}) \models Prop$ iff $M(T) \models Prop$.

$T := \emptyset$; $\quad Defs := \{prop(X) \leftarrow sat([s_0|X], \varphi)\}$; $\quad InDefs := Defs$;
while $InDefs \neq \emptyset$ **do**
  $unfold(InDefs, Ds)$;
  $define\text{-}fold(Ds, Defs, NewDefs, Es)$;
  $T := T \cup Es$; $\quad Defs := Defs \cup NewDefs$; $\quad InDefs := NewDefs$
od;

Let us now introduce some notions needed for presenting the *unfold* procedure and the *define-fold* procedure. A conjunction of literals $L_1 \wedge \ldots \wedge L_m$, with $m \geq 1$, is called a *block* if, for some variable X of type `ilist`, $vars(L_1) = \cdots = vars(L_m) = \{X\}$. A block is said to be *positive* if it contains at least one positive literal. Otherwise, it is said to be *negative*. Two blocks B_1 and B_2 are *disjoint* if $vars(B_1) \cap vars(B_2) = \emptyset$

A *definition clause* is a (non-monadic) ω-clause of the form $p(X) \leftarrow B$, where B is a positive block and $vars(B) = \{X\}$. A *quasi-monadic* clause is a clause of the form $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$, where $l \geq 0$ and $B_1, \ldots, B_l$ are pairwise disjoint blocks. Thus, a monadic ω-clause is a quasi-monadic clause where every block consists of exactly one literal.

The *unfold* procedure takes as input a set *InDefs* of definition clauses and returns as output a set *Ds* of quasi-monadic clauses. (Note that, in particular, clause $prop(X) \leftarrow sat([s_0|X], \varphi)$ is a definition clause.) The *unfold* procedure starts off by instantiating every variable of type `ilist` occurring in *InDefs* by a term of the form $[s|Y]$, where s is a state in Σ and Y is a new variable of type `ilist`, thereby deriving a new set *Cs* of clauses. Then, the *unfold* procedure repeatedly applies as long as possible the positive and negative unfolding rules starting from the set *Cs* of instantiated clauses. Finally, the *unfold* procedure

deletes clauses that are subsumed by other clauses. By reasoning on the structure of the program $P_{\mathcal{K},\varphi}$ one can prove that the output of the *unfold* procedure is a set *Ds* of quasi-monadic clauses.

The *unfold* Procedure.
Input: A set *InDefs* of definition clauses. *Output*: A set *Ds* of quasi-monadic clauses.

(*Instantiation*)
let Y be a new variable of type `ilist` and let $s_0, \ldots, s_h$ be the states of $\mathcal{K}$
$Cs := \{\, C\{X/[s_0|Y]\}, \ldots, C\{X/[s_h|Y] \mid C \in \mathit{InDefs} \text{ and } \mathit{vars}(C) = \{X\}\}$
(*Unfolding*)
while there exists a clause C in *Cs* **do**
(*Case* 1. *Positive Unfolding*)
if (i) C is of the form $p([s|Y]) \leftarrow G_1 \wedge A \wedge G_2$, where A is a positive literal, and $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ 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
(ii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i)
then $Cs := (Cs - \{C\}) \cup \{p([s|Y]) \leftarrow G_1 \wedge B_1\vartheta_1 \wedge G_2, \ldots, p([s|Y]) \leftarrow G_1 \wedge B_m\vartheta_m \wedge G_2\}$
(*Case* 2. *Negative Unfolding*)
elseif (i) C is of the form $p([s|Y]) \leftarrow G_1 \wedge \neg A \wedge G_2$ and $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$,
(ii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i), and
(iii) for $i = 1, \ldots, m$, $\mathit{vars}(B_i) \subseteq \mathit{vars}(K_i)$
then from $G_1 \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_2$ 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;
$Cs := (Cs - \{C\}) \cup \{p([s|Y]) \leftarrow Q_1, \ldots, p([s|Y]) \leftarrow Q_r\}$
(*Case* 3. *No Unfolding*)
else $Cs := (Cs - \{C\})$; $Ds := Ds \cup \{C\}$
od;
(*Subsumption*)
while there exists a clause C_1 in *Ds* of the form $p([s|Y]) \leftarrow G_1$ and a variant of a clause C_2 in $Ds - \{C_1\}$ of the form $p([s|Y]) \leftarrow G_1 \wedge G_2$ **do** $Ds := Ds - \{C_2\}$ **od**

The *define-fold* procedure transforms every quasi-monadic clause $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$ in *Ds* into a monadic ω-clause by applying, for $i = 1, \ldots, l$, to each block B_i the positive or negative folding rule as follows. If B_i is a positive block, then the *define-unfold* procedure introduces a new definition clause N of the form $q_i(Y_i) \leftarrow B_i$, where q_i is a fresh new predicate symbol and $\mathit{vars}(B_i) = \{Y_i\}$, unless that clause N already belongs to *Defs* (modulo the name of the head predicate). Clause N is added to the set *Defs* of definition clauses which can be used for subsequent folding steps. Clause N is also added to the set *InDefs*

of definition clauses. (*InDefs* will be processed in a subsequent execution of the body of the while loop of the strategy.)

Otherwise, if B_i is a negative block of the form $\neg A_1 \wedge \ldots \wedge \neg A_m$, the *define-fold* procedure introduces m new definition clauses N_1: $r_i(Y_i) \leftarrow A_1$, ..., N_m: $r_i(Y_i) \leftarrow A_m$, where r_i is a fresh new predicate symbol and $vars(B_i) = \{Y_i\}$, unless those clauses already belong to *Defs* (modulo the name of the head predicate). The definition clauses $N_1, \ldots, N_m$ are added to *Defs* and to *InDefs*.

Finally, we obtain a monadic ω-clause $p([s|X]) \leftarrow M_1 \wedge \ldots \wedge M_l$ as follows. For $i = 1, \ldots, l$, if B_i is a positive block then the we apply the positive folding rule to B_i using clause N and, thus, the literal M_i is $q_i(Y_i)$. Otherwise, if B_i is a negative block, we apply the negative folding rule to B_i using clauses $N_1, \ldots, N_m$ and, thus, the literal M_i is $\neg\, r_i(Y_i)$. (Note that X and Y_i may be identical.)

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

NewDefs := $\emptyset$; *Es* := $\emptyset$;
for each clause $D \in Ds$ **do**
if D is a monadic ω-clause **then** $Es := Es \cup \{D\}$ **else**
  let D be of the form $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$, where $B_1, \ldots, B_l$ are pairwise disjoint blocks.
  for $i = 1, \ldots, l$ **do**
    let $B_i = L_1 \wedge \ldots \wedge L_m$ and $vars(L_1) = \ldots = vars(L_m) = \{Y_i\}$
  (*Case* 1. *Positive Define-Fold*)
    if for some $j \in \{1, \ldots, m\}$, L_j is a *positive* literal
    then if there exists a clause N of the form $q_i(Y_i) \leftarrow L_1 \wedge \ldots \wedge L_m$ such that $N \in Defs \cup NewDefs$ and the predicate symbol q_i does not occur in $(Defs \cup NewDefs) - \{N\}$
      then $M_i := q_i(Y_i)$
      else *NewDefs* := *NewDefs* $\cup \{r_i(Y_i) \leftarrow L_1 \wedge \ldots \wedge L_m\}$, where r_i is a fresh new predicate symbol;
        $M_i := r_i(Y_i)$
  (*Case* 2. *Negative Define-Fold*)
    else for $j = 1, \ldots, m$, let L_j be a *negative* literal $\neg A_j$
      if there exists a set *Ns* of clauses of the form $\{q_i(Y_i) \leftarrow A_1, \ldots, q_i(Y_i) \leftarrow A_m\}$ such that $Ns \subseteq Defs \cup NewDefs$ and q_i does not occur in $(Defs \cup NewDefs) - Ns$
      then $M_i := \neg\, q_i(Y_i)$
      else *NewDefs* := *NewDefs* $\cup \{r_i(Y_i) \leftarrow A_1, \ldots, r_i(Y_i) \leftarrow A_m\}$, where r_i is a fresh new predicate symbol;
        $M_i := \neg\, r_i(Y_i)$
  od;
  $Es := Es \cup \{H \leftarrow M_1 \wedge \ldots \wedge M_l\}$
od

The transformation 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 $M(P_{\mathcal{K},\varphi}) \models \mathit{Prop}$ iff $M(T) \models \mathit{Prop}$. This correctness result can be proved similarly to [7,19]. Note that the instantiation rule we use in the *unfold* procedure is not present in [7,19], but its application can be viewed as an unfolding of an additional atom $\mathit{ilist}(X)$ defined by the clauses: $\mathit{ilist}([s_0|Y]) \leftarrow, \ldots, \mathit{ilist}([s_h|Y]) \leftarrow$.

Our transformation strategy terminates for every input program $P_{\mathcal{K},\varphi}$. The proof of termination of the *unfold* procedure is based on 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 3) or the term occurring in the body is a proper subterm of the term occurring in the head (see clause 4). (4) For each clause in $P_{\mathcal{K},\varphi}$ whose head is of the form $\mathit{sat}(l_1, \psi_1)$ and for each literal of the form $\mathit{sat}(l_2, \psi_2)$ 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) The applicability conditions given in the *unfold* procedure (see Point (ii) of Case 1 and Case 2) do not allow the unfolding of a clause C if this unfolding instantiates a variable in C.

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

Finally, the proof of termination of the while loop of the transformation strategy follows from the fact that only a finite number of new clauses can be introduced by the *define-fold* procedure. Indeed, every new clause is of the form $\mathit{newp}(X) \leftarrow L_1 \wedge \ldots \wedge L_m$, where for $i = 1, \ldots, m$, either (a) L_i is an atom A_i or a negated atom $\neg A_i$, where A_i belongs to the set $\{\mathit{notpath}(X)\} \cup \{\mathit{notpath}([s|X]) \mid s \in \Sigma\} \cup \{\mathit{sat}(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$, or (b) L_i belongs to the set $\{\neg\, \mathit{sat}([s|X], e(\psi)) \mid s \in \Sigma \text{ and } \psi \text{ is a subformula of } \varphi\}$.

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

Example 3. Let us consider program $P_{\mathcal{K},\varphi}$ of Example 2. Our transformation strategy starts off from the sets $T = \emptyset$ and $\mathit{Defs} = \mathit{InDefs} = \{\gamma_1\}$, where γ_1 is the following definition clause:

$$\mathit{prop}(X) \leftarrow \mathit{sat}([s_0|X], \mathit{and}(e(\mathit{not}(u(tt, \mathit{not}(a)))), e(\mathit{not}(u(tt, \mathit{not}(u(tt, b))))))))$$

In the first execution of the loop body of our strategy we apply the *unfold* procedure to the set *InDefs*. We get the set $Ds = \{\gamma_2, \gamma_3, \gamma_4\}$ of quasi-monadic clauses, where:

$$\begin{aligned}
\gamma_2\colon\ & \mathit{prop}([s_0|X]) \leftarrow \neg\, \mathit{notpath}([s_0|Y]) \wedge \neg\, \mathit{sat}(Y, u(tt, \mathit{not}(a))) \wedge \\
& \qquad \neg\, \mathit{notpath}([s_0|Z]) \wedge \mathit{sat}(Z, u(tt, b)) \wedge \neg\, \mathit{sat}(Z, u(tt, \mathit{not}(u(tt, b))))) \\
\gamma_3\colon\ & \mathit{prop}([s_1|X]) \leftarrow \neg\, \mathit{notpath}([s_0|Y]) \wedge \neg\, \mathit{sat}(Y, u(tt, \mathit{not}(a))) \wedge \\
& \qquad \neg\, \mathit{notpath}([s_0|Z]) \wedge \mathit{sat}(Z, u(tt, b)) \wedge \neg\, \mathit{sat}(Z, u(tt, \mathit{not}(u(tt, b)))))
\end{aligned}$$

γ_4: $prop([s_2|X]) \leftarrow \neg\, notpath([s_0|Y]) \wedge \neg\, sat(Y, u(tt, not(a))) \wedge$
$\neg\, notpath([s_0|Z]) \wedge sat(Z, u(tt, b)) \wedge \neg\, sat(Z, u(tt, not(u(tt, b))))$

Then, by applying the *define-fold* procedure, we get the set $NewDefs = \{\gamma_5, \gamma_6, \gamma_7\}$ of definition clauses and the set $Es = \{\gamma'_2, \gamma'_3, \gamma'_4\}$ of monadic ω-clauses (note that the body of each clause in Ds is partitioned into two blocks), where:

γ_5: $p_1(X) \leftarrow notpath([s_0|X])$
γ_6: $p_1(X) \leftarrow sat(X, u(tt, not(a)))$
γ_7: $p_2(X) \leftarrow \neg\, notpath([s_0|X]) \wedge sat(X, u(tt, b)) \wedge$
$\neg\, sat(X, u(tt, not(u(tt, b))))$
γ'_2: $prop([s_0|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
γ'_3: $prop([s_1|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
γ'_4: $prop([s_2|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$

Thus, at the end of the first execution of the body of the while loop of our strategy, we get: $T = \{\gamma'_2, \gamma'_3, \gamma'_4\}$, $Defs = \{\gamma_1\} \cup \{\gamma_5, \gamma_6, \gamma_7\}$, and $InDefs = \{\gamma_5, \gamma_6, \gamma_7\}$. Since $InDefs \neq \emptyset$ we execute again the body the while loop. After a few more executions we get the following monadic ω-program T:

$prop([s_0|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$prop([s_1|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$prop([s_2|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$p_1([s_0|X]) \leftarrow p_{10}(X)$
$p_1([s_0|X]) \leftarrow p_{11}(X)$
$p_1([s_1|X]) \leftarrow$
$p_1([s_2|X]) \leftarrow$
$p_2([s_0|X]) \leftarrow p_2(X)$
$p_2([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_2([s_1|X]) \leftarrow p_4(X)$
$p_3([s_0|X]) \leftarrow$
$p_3([s_1|X]) \leftarrow p_9(X)$
$p_3([s_1|X]) \leftarrow p_7(X)$
$p_3([s_2|X]) \leftarrow p_5(X)$
$p_3([s_2|X]) \leftarrow \neg\, p_6(X)$
$p_3([s_2|X]) \leftarrow p_7(X)$
$p_4([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_4([s_1|X]) \leftarrow p_4(X)$
$p_4([s_2|X]) \leftarrow p_8(X)$
$p_5([s_0|X]) \leftarrow$
$p_5([s_1|X]) \leftarrow p_9(X)$
$p_5([s_2|X]) \leftarrow$
$p_6([s_0|X]) \leftarrow p_6(X)$
$p_6([s_1|X]) \leftarrow$
$p_6([s_2|X]) \leftarrow p_6(X)$
$p_7([s_0|X]) \leftarrow \neg\, p_6(X)$
$p_7([s_0|X]) \leftarrow p_7(X)$
$p_7([s_1|X]) \leftarrow p_7(X)$
$p_7([s_2|X]) \leftarrow \neg\, p_6(X)$
$p_7([s_2|X]) \leftarrow p_7(X)$
$p_8([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_8([s_1|X]) \leftarrow p_4(X)$
$p_9([s_0|X]) \leftarrow$
$p_9([s_1|X]) \leftarrow p_9(X)$
$p_9([s_2|X]) \leftarrow p_5(X)$
$p_{10}([s_0|X]) \leftarrow p_{10}(X)$
$p_{10}([s_1|X]) \leftarrow p_9(X)$
$p_{10}([s_2|X]) \leftarrow$
$p_{11}([s_0|X]) \leftarrow p_{11}(X)$
$p_{11}([s_1|X]) \leftarrow$
$p_{11}([s_2|X]) \leftarrow p_{11}(X)$

3.2 A Proof System for Monadic ω-Programs.

Now we present a proof system for checking whether or not $M(P) \models F$ holds for any monadic ω-program P and any quantified literal F. Thus, in particular, we will be able to check whether or not $M(T) \models Prop$ holds for the monadic ω-program T derived by the transformation strategy presented in Section 3.1 and the quantified literal $Prop$ which encodes the state formula to be verified (recall that $Prop$ stands for $\exists X\, prop(X)$).

S1. $\dfrac{}{P \vdash \mathit{true}}$ S2. $\dfrac{P \nvdash F}{P \vdash \neg F}$

S3. $\dfrac{P \vdash F}{P \vdash \exists(F)}$ if $\mathit{closed\_literal}(F)$

S4. $\dfrac{P \vdash \exists(F)}{P \vdash \exists(p(X))}$ if there exists $p([s|Y]) \leftarrow F \in P$ for some $s \in \{s_0, \ldots, s_h\}$

S5. $\dfrac{P \vdash \neg\forall(p(X))}{P \vdash \exists(\neg p(X))}$ S6. $\dfrac{P \vdash \exists(F_1) \quad P \vdash \exists(F_2)}{P \vdash \exists(F_1 \wedge F_2)}$

S7. $\dfrac{P \vdash F}{P \vdash \forall(F)}$ if $\mathit{closed\_literal}(F)$

S8. $\dfrac{P \vdash \forall(F_0) \quad \ldots \quad P \vdash \forall(F_h)}{P \vdash \forall(p(X))}$ if $\{p([s_0|Y]) \leftarrow F_0, \ldots, p([s_h|Y]) \leftarrow F_h\} \subseteq P$

S9. $\dfrac{P \vdash \neg\exists(p(X))}{P \vdash \forall(\neg p(X))}$ S10. $\dfrac{P \vdash \forall(F_1) \quad P \vdash \forall(F_2)}{P \vdash \forall(F_1 \wedge F_2)}$

Fig. 1. Proof system for monadic ω-programs. $\Sigma = \{s_0, \ldots, s_h\}$ is the set of states of the Kripke structure $\mathcal{K}$. For any formula F, $\mathit{closed\_literal}(F)$ holds iff F is either the formula *true* or the formula $\exists(L)$, where L is a literal.

Every monadic ω-program P used by the proof system is first rewritten as follows. (i) Every clause of the form $H \leftarrow$ is rewritten as $H \leftarrow \mathit{true}$ (so that no clause in P has an empty body), and (ii) for every clause of the form $H \leftarrow G$, each literal L occurring in G such that $\mathit{vars}(H) \cap \mathit{vars}(L) = \emptyset$ is replaced by its existential closure $\exists(L)$. (Recall that in the body of a monadic ω-clause two distinct literals do not share any variable.) For instance, the clause $p([s|X]) \leftarrow q(X) \wedge p(Y)$ is rewritten as $p([s|X]) \leftarrow q(X) \wedge \exists Y\, p(Y)$.

The proof rules presented in Figure 1 define a relation $\vdash$ such that $P \vdash F$ iff $M(P) \models F$, for every monadic ω-program P and closed formula F, which is of one of the following forms: true, $\forall(F_1 \wedge \ldots \wedge F_k)$, $\exists(F_1 \wedge \ldots \wedge F_k)$, where $k \geq 1$ and for $i = 1, \ldots k$, F_i is either a literal or the existential closure of a literal.

Note that the proof rule S2 is the *negation as failure* rule, that is, the negated judgement '$P \nvdash F$' should be interpreted as '$P \vdash F$ cannot be proved by using the proof rules S1–S10'. This interpretation of $P \nvdash F$ as (finite or infinite) failure of $P \vdash F$ is meaningful because P is stratified and, thus, also the instances of the proof rules are stratified. The stratification of these instances is induced by the existence of a mapping μ from formulas to natural numbers such that, for every rule instance with conclusion $P \vdash F_1$, (i) if $P \vdash F_2$ occurs as a premise, then $\mu(F_1) \geq \mu(F_2)$, and (ii) if $P \nvdash F_3$ occurs as a premise, then $\mu(F_1) > \mu(F_3)$. Thus, when constructing a proof of $P \vdash F$ it is never required to show that $P \nvdash F$, that is, it is never required to show that no proof of $P \vdash F$ can be constructed.

Note also that the proof rule S6 is sound because, as already mentioned, the literals in the body of a monadic ω-clause do not share any variable and, therefore, the existential quantifier distributes over the conjunction of those literals.

By induction on the strata of the monadic ω-program P we can show that the proof rules of Figure 1 are sound and complete for proving that a quantified literal F is true in the perfect model of P, as stated by the following theorem.

Theorem 3. *Let P be a monadic ω-program and F a formula of the form $\exists(L)$ or $\forall(L)$, where L is a literal. Then, $P \vdash F$ iff $M(P) \models F$.*

The proof system for monadic ω-programs given in Figure 1 can be encoded as a logic program, called *Demo*. Given a monadic ω-program P and a closed literal F, the program *Demo* uses the (ground) representations $\lceil P \rceil$ and $\lceil F \rceil$ of P and F, respectively, which are constructed as follows. Let v be a new constant symbol. (i) For any variable X, $\lceil X \rceil$ is v, (ii) for any list $[s|X]$, where s is a state and X is a variable, $\lceil [s|X] \rceil$ is $[s|v]$, (iii) for any unary predicate q and term t, $\lceil q(t) \rceil$ is $(q, \lceil t \rceil)$, (iv) for any atom A, $\lceil \neg A \rceil$ is $not(\lceil A \rceil)$, (v) for any conjunction $F_1 \wedge F_2$, $\lceil F_1 \wedge F_2 \rceil$ is $and(\lceil F_1 \rceil, \lceil F_2 \rceil)$, (vi) for any F which is either a literal or a conjunction, $\lceil \exists(F) \rceil$ is $exists(\lceil F \rceil)$ and $\lceil \forall(F) \rceil$ is $all(\lceil F \rceil)$, (vii) for any clause C of the form $H \leftarrow$, $\lceil C \rceil$ is the unit clause $clause(\lceil H \rceil, true) \leftarrow$, (viii) for any clause C of the form $H \leftarrow F$, $\lceil C \rceil$ is the unit clause $clause(\lceil H \rceil, \lceil F \rceil) \leftarrow$, and, finally, (ix) for any monadic ω-program $P = \{C_1, \ldots, C_n\}$, $\lceil P \rceil$ is the set of ground unit clauses $\{\lceil C_1 \rceil, \ldots, \lceil C_n \rceil\}$.

D1. $demo(true) \leftarrow$
D2. $demo(not(F)) \leftarrow \neg\, demo(F)$
D3. $demo(exists(F)) \leftarrow closed\_literal(F) \wedge demo(F)$
D4. $demo(exists((R, v))) \leftarrow clause((R, [S|v]),\ F) \wedge demo(exists(F))$
D5. $demo(exists(not((R, v)))) \leftarrow demo(not(all((R, v))))$
D6. $demo(exists(and(F_1, F_2))) \leftarrow demo(exists(F_1)) \wedge demo(exists(F_2))$
D7. $demo(all(F)) \leftarrow closed\_literal(F) \wedge demo(F)$
D8. $demo(all((R, v))) \leftarrow clause((R, [s_0|v]),\ F_0) \wedge demo(all(F_0)) \wedge \ldots \wedge$
$clause((R, [s_h|v]),\ F_h) \wedge demo(all(F_h))$
D9. $demo(all(not((R, v)))) \leftarrow demo(not(exists((R, v))))$
D10. $demo(all(and(F_1, F_2))) \leftarrow demo(all(F_1)) \wedge demo(all(F_2))$

Since P is a stratified program, $Demo \cup \lceil P \rceil$ is a *weakly stratified* program [15] and, hence, it has a unique perfect model $M(Demo \cup \lceil P \rceil)$.

Theorem 4. *Let P be a monadic ω-program and F be a formula of the form $\exists(L)$ or $\forall(L)$, where L is a literal. Then, $P \vdash F$ iff $M(Demo \cup \lceil P \rceil) \models demo(\lceil F \rceil)$.*

Thus, by Theorems 3 and 4 for any monadic ω-program T derived from $P_{\mathcal{K},\varphi}$ using the transformation strategy described in Section 3.1, we can check whether or not $M(T) \models Prop$ holds (and, thus, whether or not $\mathcal{K} \models \varphi$ holds) by using any logic programming system which computes the perfect model of $Demo \cup \lceil T \rceil$. One can use, for instance, a system based on *tabled resolution* [2,18] which guarantees the termination of the evaluation of the query $demo(\lceil Prop \rceil)$

and returns 'yes' iff $M(Demo \cup \lceil T \rceil) \models demo(\lceil Prop \rceil)$. Indeed, starting from $demo(\lceil Prop \rceil)$, we can only derive a finite set of queries of the form $demo(\lceil F \rceil)$, and the tabling mechanism ensures that each query is evaluated at most once.

Example 4. Let T be the monadic ω-program obtained in Example 3 as the output of our transformation strategy. In order to prove that $T \vdash \exists X\, prop(X)$, we compute the encoding $\lceil T \rceil$ of the program T and the encoding $exists((prop, v))$ of the property $\exists X\, prop(X)$. Then, we evaluate the query $demo(exists((prop, v)))$ w.r.t. the program $Demo \cup \lceil T \rceil$ by using a system based on tabled resolution and we obtain the answer 'yes'. Since $T \vdash \exists X\, prop(X)$, we have that $M(P_{\mathcal{K},\varphi}) \models Prop$ and, thus, we get that the formula φ holds in the Kripke structure $\mathcal{K}$ (see Example 2).

3.3 Complexity of the Verification Technique

We will measure the time complexity of our verification technique as: (i) the number of applications of transformation rules in Step 1 and (ii) the number of closed literals that are evaluated when executing the *Demo* program in Step 2.

Let us first consider Step 1 and let us measure the number of new predicate symbols, that is, the number of distinct blocks that can be generated during the *define-fold* procedure. Let $B = L_1 \wedge \ldots \wedge L_m$ be a block. Since the literals in B cannot be further unfolded, we have that, for $i = 1, \ldots, m$, either (a) L_i is the atom A_i or the negated atom $\neg A_i$, where A_i belongs to the set $\{notpath(X)\} \cup \{notpath([s|X]) \mid s \in \Sigma\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$, or (b) L_i belongs to the set $\{\neg\, sat([s|X], e(\psi)) \mid s \in \Sigma \text{ and } \psi \text{ is a subformula of } \varphi\}$. We have also the following properties of B: (i) there is at most one atom in B of the form $notpath([s|X])$, and (ii) if in B there are two occurrences of terms of the form $[s_1|X]$ and $[s_2|X]$, then $s_1 = s_2$. By these properties the number of distinct blocks and, thus, the number of new predicate symbols is $\mathcal{O}(|\Sigma| \cdot 2^{|\varphi|})$. Moreover, the size of each block is $\mathcal{O}(|\varphi|)$, which is also the number of clauses introduced in the set *InDefs* for each new predicate symbol.

For each clause in *InDefs*, our transformation strategy performs one execution of the loop body, which starts off by applying the *unfold* procedure. This procedure applies the instantiation rule which generates $|\Sigma|$ clauses and, then, makes $\mathcal{O}(|\varphi|)$ unfolding steps for each instantiated clause. The total number of unfolding steps for each new predicate is, thus, $\mathcal{O}(|\Sigma| \cdot |\varphi|^2)$, which is also the number of the derived clauses. Since there are $\mathcal{O}(|\varphi|^2)$ clauses with same head, the number of subsumption steps is $\mathcal{O}(|\Sigma| \cdot |\varphi|^4)$ for each new predicate symbol. Thus, the total number of transformation rule applications in the *unfold* procedure is $\mathcal{O}(|\Sigma| \cdot |\varphi|^4)$. If we consider all new predicates symbols, we get $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^4 \cdot 2^{|\varphi|})$ applications of transformation rules.

The *define-fold* procedure performs at most one folding and one definition introduction for each block occurring in the body of a clause. The number of blocks in the body of a clause is $\mathcal{O}(|\varphi|)$ because each block has been introduced by unfolding an atom of the form $sat(X, e(\psi))$, where $e(\psi)$ is a subformula of φ.

The number of folding steps is, thus, $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^5 \cdot 2^{|\varphi|})$, while the number of definition introductions is equal to the number of predicate symbols. Therefore, the total number of applications of transformation rules in Step 1 of our verification technique is $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^5 \cdot 2^{|\varphi|})$.

In Step 2, a logic programming system which uses tabled resolution evaluates every closed literal at most once. The proof of a closed literal requires $\mathcal{O}(|\Sigma|^2 \cdot 2^{|\varphi|})$ applications of proof rules. Therefore, we may conclude that the time complexity of our verification algorithm is $\mathcal{O}(|\Sigma|^2) \cdot 2^{\mathcal{O}(|\varphi|)}$.

In [8] an algorithm for CTL* model checking is provided whose time complexity is $\mathcal{O}(|\Sigma| + |\rho|) \cdot 2^{\mathcal{O}(|\varphi|)}$. Note that, since ρ is a total binary relation, we have $|\Sigma| \leq |\rho| \leq |\Sigma|^2$. Thus, in the case where $|\rho| = |\Sigma|^2$, the time complexity of our algorithm is the same of the best known algorithm for CTL* model checking. The case where the Kripke structure $\mathcal{K}$ represents a deterministic transition system and, thus, $|\rho| = |\Sigma|$ is quite unfrequent in practice.

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 [17]. 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, [4,6,10,12]). In this paper we have extended to CTL* model checking the transformational approach which was proposed for LTL model checking in [13].

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, 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 truth in the perfect model is decidable. This subclass of programs properly extends the class of *linear recursive* ω-programs introduced in [13] and also the proof system presented here extends the one in [13]. (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. The transformation strategy presented in this paper is more elaborated than the one considered in [13] for the case of LTL properties. However, the worst case time complexity of the two strategies is the same.

Our transformation strategy can be understood 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 [9] for a brief survey). Indeed, our transformation strategy performs folding 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 fi-

nite state system and a CTL* formula are translated into alternating tree automata [8]. 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 features. (1) The specification of properties of reactive systems, the transformation of specifications into a monadic ω-programs, and the proofs of properties of monadic ω-programs are all expressed within the single framework of logic programming, while in the automata-theoretic approach one has to translate the temporal logic formalism into the automata-theoretic formalism. (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 easily be extended to the case of infinite state systems.

Issues that can be investigated in the future include: (i) the relationship between monadic ω-programs and alternating tree automata, (ii) the applicability of our transformational approach to other logics, such as the monadic second order logic of successors, and (iii) the experimental evaluation of the efficiency of our transformational approach by considering various test cases and comparing its performance in practice w.r.t. that of other model checking techniques.

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
2. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1), 1996.
3. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
4. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
5. E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.
6. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01*, Technical Report DSSE-TR-2001-3, pages 85–96. Univ. Southampton, UK, 2001.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, pp. 292–340. Springer-Verlag, 2004.
8. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.
9. M. Leuschel. Logic program specialisation. In J. Hatcliff and P. Thiemann (Eds.) T. Mogensen, editors, *Partial Evaluation - Practice and Theory*, LNCS 1706, pages 155–188. Springer, 1998.
10. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proc. of LOPSTR '99*, LNCS 1817, pages 63–82. Springer, 2000.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

12. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. of CL 2000*, LNAI 1861, pages 384–398. Springer, 2000.
13. A. Pettorossi, M. Proietti, and V. Senni. Transformational verification of linear temporal logic. *24th Italian Conference on Computational Logic June 24-26, 2009, Ferrara, Italy (CILC '09)*. http://www.ing.unife.it/eventi/cilc09.
14. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
15. H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.
16. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
17. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *Proceedings of CAV '97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
18. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB System, Version 2.2., 2000.
19. H. Seki, Unfold/Fold transformation of stratified programs. *Theoretical Computer Science*, 86, 1:107–139, 1991.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of ICLP'84*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.

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 be used to prove any property 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 optimizations [9], more precise termination conditions [4, 7] and more refined interpretations for proving termination [1].

In Logic Programming (LP), many tools are available for performing such analyses [2, 5, 10]. 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 set of constraints. This policy prevents the infinite application of propagation rules, that add constraints to the store without removing any. The approach in [8] has a problem with this.

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

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, several direct approaches were developed [3, 9], mainly based on approaches developed for LP. To the best of our knowledge, no transformational approaches have been attempted. However, the transformation discussed here is founded on the termination preserving transformation discussed in [8].

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, executed under the abstract CHR semantics, to Prolog. Section 4, discusses the application of our transformation to type analysis of CHR, using PolyTypes 1.3 (based on [2]) on the transformed programs. Finally in Section 5, 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 CHR and built-in constraints. Finally, *simpagation rules* replace CHR constraints by new CHR and built-in 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, $R \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B, C$. 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.

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

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 of constraints, called the *constraint store*. In it, there may be multiple identical constraints.

Definition 1 (CHR state). *A CHR state S is a conjunction 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.

3 Transforming CHR 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 (commutative) conjunction of constraints. To represent it in Prolog, we fix some order and represent it as a list, called the *storelist*. The code that handles 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 in CHR). The following CHR program is executed with a query, $a \wedge b$.

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

The program may or may not terminate for the initial query, depending on which rules are applied. If the first rule is applied, then the program terminates immediately. If the second and third rule are applied 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:

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

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 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 $rule([p(X_1, \ldots, X_n)|R], R) \ \texttt{:-} \ p(X_1, \ldots, X_n)$.

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.

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

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 $goal([mergesort(L)])$, where L is a list of natural numbers as in Example 1. □

3.4 Transformation Summary

To transform CHR states to Prolog queries, we introduce a mapping, $\alpha : S \rightarrow Q$, 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:
  $rule([H_1, \ldots, H_n|R], [C_1, \ldots, C_m, B_1, \ldots, B_l|R])$:- $G_1, \ldots, G_k$.
- *The Prolog program $\wp$ contains for every built-in predicate p/n in P, a clause:*
  $rule([p(X_1, \ldots, X_n)|R], R)$:- $p(X_1, \ldots, X_n)$. □

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', that 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.

Theorem 1 (Connecting $\wp$ and P). *There exist the following relation between a transformed CHR program, $\wp = C2P(P)$, and the original CHR program, P:*

$$\begin{array}{ccc} Q & \xrightarrow[\wp]{} & Q' \\ \uparrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \gamma} \\ S & \xrightarrow[P]{} & S' \end{array}$$

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 a CHR transition in P and a derivation in the transformed program $\wp$. □

This relation establishes that the analysis of properties of constraints, part of the CHR constraint store, can take place on its transformed program $C2P(P)$ as well. 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 $goal/1$.

4 Application of the transformation to type analysis

We apply our transformation to type analysis of merge-sort from Example 1. First, we represent all rules by simplification rules, as was done in Example 3. Then, we transform the program to a Prolog program according to Definition 3:

```
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(_).

leq(s(X),s(Y)) :- leq(X,Y).          leq(0,X).

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]) :- leq(L1,L2).
rule([a(L1,L2),a(L1,L3)|T],[a(L1,L2),a(L2,L3)|T]) :- leq(L2,L3).
```

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:	**Signatures:**
$t5 \rightarrow msort(t26); a(t34, t34); r(t28, t34)$	$goal(t35)$
$t26 \rightarrow []; [t34 \mid t26]$	$perm(t35, t35)$
$t28 \rightarrow 0; s(t28)$	$rule(t35, t35)$
$t34 \rightarrow 0; s(t34)$	$del(t5, t35, t35)$
$t35 \rightarrow []; [t5 \mid t35]$	$leq(t34, t34)$

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, the storelist, is $t35$. That is, a list of elements of type $t5$. Thus, terms of the form $msort(t26)$, $a(t34, t34)$ or $r(t28, t34)$. Hence PolyTypes 1.3 correctly derives the types of the constraints that can occur in the storelist and thus in the constraint store.

One could also 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 $t26$, as the argument of $msort$ in the query uses different list constructors than those in the $msort$-rules. Actually, the obtained type is then a grave overestimation of the actual content 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 has the type $t35$ and specifying an initial type for $t5$, e.g. $t5 \rightarrow msort(t26)$. Translating these types into input for PolyTypes, the tool will then extend the types and obtain the same types as the ones shown above.

The transformation to Prolog has been implemented[1] according to Definition 3. Although the transformation has been shown here in the context of a type analysis, it can also be used to derive other properties of the elements of the CHR constraint store, e.g. groundness information, mode informations, call types, etc.

5 Conclusion

We have presented a transformation from CHR 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 and obtained accurate type descriptions for the constraints of a CHR program.

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

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

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 a 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 our transformation to groundness and call type analysis. Together with type analysis, these are the main sources of information for termination analysis.

References

1. Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination Analysis through Combination of Type Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):10, 2007.
2. Maurice Bruynooghe, John P. Gallagher, and Wouter Van Humbeeck. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *SAS '05: Proceedings of 12th International Statical Analysis Symposium*, pages 35–51, 2005.
3. Emmanuel Coquery and François Fages. A Type System for CHR. In *CHR '05: Proceedings of the 2nd International Workshop on Constraint Handling Rules*, pages 19–33, 2005.
4. Danny De Schreye and Stefaan Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.
5. Gerda Janssens and Maurice Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):199–260, 1992.
6. Manh Thang Nguyen. *Termination Analysis: Crossing Paradigm Borders.* PhD thesis, Katholieke Universiteit Leuven - Departement Computer Wetenschappen, Belgium, 2009.
7. Paolo Pilozzi and Danny De Schreye. Termination analysis of CHR revisited. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 501–515, 2008.
8. Paolo Pilozzi, Tom Schrijvers, and Danny De Schreye. Proving termination of CHR in Prolog: A transformational approach. In *WST '07: Proceedings of the 9th International Workshop on Termination*, 2007.
9. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules.* PhD thesis, Katholieke Universiteit Leuven, Departement Computer Wetenschappen, Belgium, 2005.
10. Tom Schrijvers, Maurice Bruynooghe, and John P. Gallagher. From Monomorphic to Polymorphic Well-Typings and Beyond. In *LOPSTR '08: Pre-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation*, pages 152–167, 2008.

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*,[4] 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*.

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

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 $\Box$.

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*".*

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

For example, for suitable substitutions δ_0 and δ_1 we have $\mathsf{goal}(\mathsf{s}(0), \mathsf{s}(0), Msu) \vdash_{\delta_0,\mathcal{P}} \mathsf{s2m}(\mathsf{s}(0), \mathsf{s}(0), M), \mathsf{subs\_mat}(M, Msu) \vdash^8_{\delta_1,\mathcal{P}} \square$. *So* $Answer(\mathcal{P}, \mathsf{goal}(\mathsf{s}(0), \mathsf{s}(0), Msu))$ *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.[5]

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 $DT(\mathcal{P}) = \{H \leftarrow I, B \mid H \leftarrow I, B, \ldots \in \mathcal{P}\}$.*

[5] 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 $B_i\theta_i\sigma_i \in \mathcal{C}$.*[6]

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 $\theta_0 = \{X/\mathsf{s}(0), Y/\mathsf{s}(0)\}$, $\sigma_0 = \{M/[[C]]\}$, and $\theta_1 = \{R/[C], Rs/[\,], Msu/[SR, SRs]\}$. Then, for $A = \mathsf{goal}(\mathsf{s}(0), \mathsf{s}(0), Msu) \in \mathcal{S}$, we have $H_0\theta_0 = \mathsf{goal}(X, Y, Msu)\theta_0 = A\theta_0$. Furthermore, we have $\sigma_0 \in Answer(\mathcal{P}, \mathsf{s2m}(X, Y, M)\theta_0) = Answer(\mathcal{P}, \mathsf{s2m}(\mathsf{s}(0), \mathsf{s}(0), M))$ 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 $A'_{j_1} =$

[6] 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 $Q \in \mathcal{S}$. $\square$

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

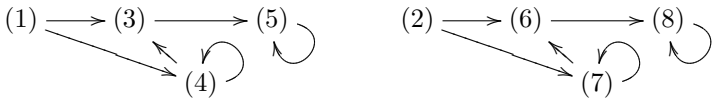
In Sect. 4.1 and 4.2, we adapt two of the most important DP processors from term rewriting [2, 6, 8, 9] to the LP setting. In Sect. 4.3 we present a new DT processor to convert DT problems to DP problems.

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].[7] 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 $\mathsf{p} \leftarrow \mathsf{q}(\mathsf{a}), \mathsf{p}$ and $\mathsf{q}(\mathsf{b})$. We consider the set of queries $\mathcal{S}' = \{\mathsf{p}\}$ and obtain $Call(\mathcal{P}', \mathcal{S}') = \{\mathsf{p}, \mathsf{q}(\mathsf{a})\}$. There are two DTs $\mathsf{p} \leftarrow \mathsf{q}(\mathsf{a})$ and $\mathsf{p} \leftarrow \mathsf{q}(\mathsf{a}), \mathsf{p}$. 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 $\mathsf{q}(\mathsf{a})$.

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

[7] 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 $t_1 \succ t_3$).[8]

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*[9]

[8] 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.

[9] So for terms t_1, t_2 we define $t_1 \mathrel{(\succsim)} t_2$ iff $\|t_1\| \mathrel{(\geq)} \|t_2\|$ and for atoms A_1, A_2 we define $A_1 \mathrel{(\succsim)} A_2$ iff $|A_1| \mathrel{(\geq)} |A_2|$.

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.[10] 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* $\sigma(Rs) \in List$, *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.

[10] 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,[11] 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.[12] In addition, one could easily adapt more techniques from

[11] 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].

[12] 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:

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

  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*,[13] 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

[13] 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 $\pi(u^\sharp_1) = \pi(\mathsf{tr}_{out}) = \{1, 2\}$. To see this, consider the substitution $\sigma = \{X/\mathsf{n}(\mathsf{n}(\mathsf{a})), Z/\mathsf{a}\}$. Now $\mathsf{u}^\sharp_1(\mathsf{tr}_{in}(X, Z), X, Y)\sigma$ reduces in one step to $\mathsf{u}^\sharp_1(\mathsf{tr}_{out}(X, Z), X, Y)\sigma$ 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

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

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

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

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

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

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].[14] The *International Termination Competition* showed that direct termination tools like Polytool and

[14] 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. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.
2. T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1,2):133–178, 2000.
3. A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Th. Comp. Sc.*, 124(2):297–328, 1994.
4. M. Codish and C. Taboch. A Semantic Basis for Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
5. N. Dershowitz. Termination of Rewriting. *J. Symb. Comp.*, 3(1,2):69–116, 1987.
6. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proc. LPAR '04*, LNAI 3452, pp. 301–331, 2005.
7. J. Giesl, P. Schneider-Kamp, R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the DP Framework. In *Proc. IJCAR '06*, LNAI 4130, pp. 281–286, 2006.
8. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
9. N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.
10. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2,3):205–258, 1992.
11. M. Jurdzinski. LP Course Notes. `http://www.dcs.warwick.ac.uk/~mju/CS205/`.
12. F. Mesnard and R. Bagnara. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1, 2):243–257, 2005.
13. M. T. Nguyen and D. De Schreye. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. *Proc. ICLP '05*, LNCS 3668, 311–325, 2005.
14. M. T. Nguyen and D. De Schreye. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In *Proc. LOPSTR '06*, LNCS 4407, pp. 210–218, 2007.
15. M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *Proc. LOPSTR '07*, LNCS 4915, pp. 8–22, 2008.
16. E. Ohlebusch, C. Claves, and C. Marché. TALP: A Tool for the Termination Analysis of Logic Programs. In *Proc. RTA '00*, LNCS 1833, pp. 270–273, 2000.
17. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 2009. To appear. Short version appeared in *Proc. LOPSTR '06*, LNCS 4407, pp. 177–193, 2007.

Improving the Termination Analysis of Narrowing in Left-Linear Constructor Systems*

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

1 Motivation

Narrowing [11] extends rewriting in order to deal with terms containing logic variables by replacing pattern-matching with unification. It has been widely used in different contexts, ranging from theorem proving (e.g., protocol verification) to language design (e.g., it forms the basis of functional logic languages).

Recently, [9, 12] introduced a termination analysis for narrowing which is roughly based on the following process:[3] First, following [1, 4], 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. Then, an *argument filtering* is used to filter away occurrences of gen in the considered computations so that the termination of narrowing reduces to a problem of termination of rewriting. Finally, termination is analyzed using the *dependency pair* framework [7] for proving the termination of rewriting over the filtered terms.

Intuitively speaking, argument filterings map every function symbol to a subset of its argument positions, i.e., given a function symbol f of arity n, $\pi(\mathsf{f})$ returns a subset of $\{1, \ldots, n\}$ so that the arguments whose position is not in the set are filtered away. Consider, for instance, the following rewrite system $\mathcal{R}$:

$$\begin{array}{rl} \mathsf{leq}(\mathsf{zero}, y) \to \mathsf{true} & \qquad \mathsf{leq}(\mathsf{succ}(x), \mathsf{succ}(y)) \to \mathsf{leq}(x, y) \\ \mathsf{leq}(\mathsf{succ}(x), \mathsf{zero}) \to \mathsf{false} & \end{array}$$

which defines the less-or-equal relation on natural numbers built from zero and succ. The set of *dependency pairs* $DP(\mathcal{R})$ of a rewrite system $\mathcal{R}$ contains a rule $\mathsf{f}^\sharp(s_1, \ldots, s_n) \to \mathsf{g}^\sharp(t_1, \ldots, t_m)$ for every rule $\mathsf{f}(s_1, \ldots, s_n) \to r \in \mathcal{R}$ such that $\mathsf{g}(t_1, \ldots, t_m)$ is a subterm of r rooted by a defined function and $\mathsf{f}^\sharp, \mathsf{g}^\sharp$ are (fresh) tuple symbols associated to the defined functions f and g, usually denoted by capital letters in the examples. In our case, the associated set of dependency pairs, $DP(\mathcal{R})$, contains the following single rule: $\mathsf{LEQ}(\mathsf{succ}(x), \mathsf{succ}(y)) \to \mathsf{LEQ}(x, y)$.

Assume now that we want to analyze the termination of narrowing for the terms of the form $\mathsf{leq}(t, x)$, where t is a ground constructor term of the form

* This work has been partially supported by the *MICINN* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant GVPRE/2008/001.

[3] The termination analysis of logic programs of [10] follows a similar pattern but logic variables are replaced with *infinite* terms (the net effect, though, is similar).

$\mathsf{succ}(\ldots(\mathsf{zero})\ldots)$ and x is a logic variable. For this purpose, one should first define an argument filtering π that filters away the second argument of leq, e.g., $\pi(\mathsf{leq}) = \{1\}$, since it will be replaced by an occurrence of gen. Then, one can analyze the termination of the standard [7] *DP problem* $(\pi(DP(\mathcal{R})), \pi(\mathcal{R}))$, in order to analyze the termination of narrowing for $\mathsf{leq}(t, x)$, where

$$\pi(DP(\mathcal{R})) = \left\{ \mathsf{LEQ}(\mathsf{succ}(x)) \to \mathsf{LEQ}(x) \right\} \qquad \pi(\mathcal{R}) = \left\{ \begin{array}{c} \mathsf{leq}(\mathsf{zero}) \to \mathsf{true} \\ \mathsf{leq}(\mathsf{succ}(x)) \to \mathsf{false} \\ \mathsf{leq}(\mathsf{succ}(x)) \to \mathsf{leq}(x) \end{array} \right\}$$

Trivially, this DP problem is finite because the only possible source of non-termination is the dependency pair $\mathsf{LEQ}(\mathsf{succ}(x)) \to \mathsf{LEQ}(x)$, but it is straightforward to find an order ">" such that $\mathsf{LEQ}(\mathsf{succ}(x)) > \mathsf{LEQ}(x)$.

In general, however, $\pi(DP(\mathcal{R}) \cup \mathcal{R})$ may contain *extra variables*, i.e., variables that appear in the right-hand side of a rule but not in its left-hand side. Consider, e.g., the following rewrite system $\mathcal{R}^{\mathsf{add}}$ and termination of $\mathsf{add}(t, x)$:

$$\mathsf{add}(\mathsf{zero}, y) \to y \qquad \mathsf{add}(\mathsf{succ}(x), y) \to \mathsf{succ}(\mathsf{add}(x, y))$$

Given an argument filtering with $\pi(\mathsf{add}) = \pi(\mathsf{succ}) = \{1\}$ and $\pi(\mathsf{zero}) = \varnothing$, we produce the following filtered DP problem $(\pi(DP(\mathcal{R}^{\mathsf{add}})), \pi(\mathcal{R}^{\mathsf{add}}))$ (here, we assume the same argument filtering for the defined function add and its associated tuple symbol ADD):

$$\left(\left\{ \mathsf{ADD}(\mathsf{succ}(x)) \to \mathsf{ADD}(x) \right\}, \left\{ \begin{array}{c} \mathsf{add}(\mathsf{zero}) \to y \\ \mathsf{add}(\mathsf{succ}(x)) \to \mathsf{succ}(\mathsf{add}(x)) \end{array} \right\} \right)$$

In this case, termination cannot be proved since rules containing extra variables (like $\mathsf{add}(\mathsf{zero}) \to y \in \pi(\mathcal{R}^{\mathsf{add}})$) are not terminating by definition.

However, this is unnecessarily restrictive in our context since narrowing can only instantiate extra variables to *constructor* terms (when unifying a function call with some left-hand side) and, thus, only instantiation of these extra variables to infinite terms can introduce an infinite computation.[4]

In order to overcome this problem, we can find two different approaches in the literature. On the one hand, [10] requires $\pi(DP(\mathcal{R}) \cup \mathcal{R})$ to be free of extra variables. For this purpose, argument filterings are *refined* until the condition holds. Basically, an argument filtering π' is a refinement of another argument filtering π if it filters away the same or more arguments, i.e., $\pi'(\mathsf{f}) \subseteq \pi(\mathsf{f})$ for every symbol f. The main drawback of this approach, besides losing some accuracy due to the refinement of the argument filtering, is that it cannot be generally applied to arbitrary rewrite systems. 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 of π will be able to eliminate it.[5]

[4] A similar situation occurs in [10] where extra variables coming from the translation of a logic program can only be bound to constructor terms.

[5] This is not a limitation of [10] since the considered rewrite systems that are produced from the translation of logic programs never have collapsing rules.

On the other hand, [12] considers an alternative approach that is based on replacing all extra variables by a fresh constant symbol, $\bot$, and then requiring the argument filtering to fulfill some additional conditions that basically amount to saying that the occurrences of $\bot$ play no role in the considered derivations.

In this work, we go one step further and consider that $\bot$ might be a *defined* function. Therefore, we get stronger results (compared to [9, 12]) in some cases, namely when the extra variables of the filtered system should take some value for the narrowing derivation to proceed. Loosely speaking, our aim is to consider $\bot$ as a *terminating* restriction of gen and identify the conditions under which this restriction is safe. For this purpose, we introduce a transformation of the initial rewrite system that allows us to introduce, in some cases, an appropriate definition for $\bot$.

2 Improving the Termination Analysis of Narrowing

We assume familiarity with basic concepts of term rewriting. The notations not defined in the paper can be found in [3] and [8]. 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 $\mathcal{F} \cap \mathcal{V} = \varnothing$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. 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 $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. The root symbol of a term t is denoted by $\mathsf{root}(t)$. Given a set S, $\mathcal{P}os_S(t)$ denotes the set of positions of a term t that are rooted by function symbols or variables in S. $\mathcal{V}\mathsf{ar}(t)$ denotes the set of variables appearing in t. A term t is *ground* if $\mathcal{V}\mathsf{ar}(t) = \varnothing$. We write $\mathcal{T}(\mathcal{F})$ as a shorthand for the set of ground terms $\mathcal{T}(\mathcal{F}, \varnothing)$.

The *narrowing* principle [11] mainly extends term rewriting by replacing pattern matching with unification, so that terms containing logic variables can also be reduced by non-deterministically instantiating these variables. Formally, 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 $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(s)}$ (i.e., we label the narrowing step only with the bindings for the narrowed term). 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 $\sigma = id$).

Following [12], we are not interested in the termination of narrowing for arbitrary terms (this is similar to the case of logic programming, where a logic program seldom terminates for all possible initial goals). Rather, we are interested in proving the termination of narrowing for a particular class of initial terms, which is specified using the notion of *abstract term* (inspired by the mode declarations of logic programming [5]). Intuitively, an *abstract term* has the form $\mathsf{f}(m_1, \ldots, m_n)$, where $\mathsf{f} \in \mathcal{D}$ and m_i is either g (definitely **g**round) or v (possibly **v**ariable), for all $i = 1, \ldots, n$. Any abstract term t^α, implicitly induces a (possibly infinite) set of terms, $\gamma(t^\alpha)$, by replacing the g arguments of t^α with ground constructor terms and the v arguments with arbitrary terms.

In the following, given a set of terms T and a binary relation $\propto$ on terms, we say that T is $\propto$-terminating if there is no term $t_1 \in T$ such that an infinite sequence of the form $t_1 \propto t_2 \propto t_3 \propto \ldots$ exists. For instance, given a TRS $\mathcal{R}$ and a set T, we will consider whether T is $\rightsquigarrow_{\mathcal{R}}$-terminating (for narrowing) or $\rightarrow_{\mathcal{R}}$-terminating (for rewriting).

2.1 Termination of narrowing and extra variables

As mentioned in Sect. 1, the challenge in this paper is dealing with the termination of narrowing in TRSs whose filterings introduce extra variables.

Definition 1 (argument filtering, π). *An argument filtering over a signature $\mathcal{F}$ is a function π such that, for every function or constructor symbol $\mathsf{f}/n \in \mathcal{F}$, we have $\pi(\mathsf{f}) \subseteq \{1, \ldots, n\}$. Argument filterings are extended to terms as follows:*[6]

- $\pi(x) = x$ *for all* $x \in \mathcal{V}$,
- $\pi(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\pi(t_{i_1}), \ldots, \pi(t_{i_m}))$ *for all* $\mathsf{f}/n \in \mathcal{F}$, $n \geq 0$, *where* $\pi(\mathsf{f}) = \{i_1, \ldots, i_m\}$ *and* $1 \leq i_1 < \ldots < i_m \leq n$.

Let us consider the following TRS $\mathcal{R}^c$:

$$\begin{array}{rr} \mathsf{f}(\mathsf{succ}(\mathsf{zero}), y) \to \mathsf{inc}(\mathsf{dec}(\mathsf{succ}(\mathsf{zero})), \mathsf{min}, y) & \mathsf{dec}(\mathsf{succ}(y)) \to y \\ \mathsf{min} \to \mathsf{zero} & \mathsf{inc}(\mathsf{zero}, y, z) \to \mathsf{f}(y, z) \end{array}$$

together with the argument filtering

$$\pi(\mathsf{f}) = \pi(\mathsf{succ}) = \{1\} \qquad \pi(\mathsf{inc}) = \{1, 2\} \qquad \pi(\mathsf{dec}) = \pi(\mathsf{min}) = \pi(\mathsf{zero}) = \varnothing$$

Now, if we *filter* the TRS, i.e., if we replace every rule $l \to r$ in this TRS by $\pi(l) \to \pi(r)$, we get the following filtered TRS $\pi(\mathcal{R}^c)$:

$$\begin{array}{rr} \mathsf{f}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{inc}(\mathsf{dec}, \mathsf{min}) & \mathsf{dec} \to y \\ \mathsf{min} \to \mathsf{zero} & \mathsf{inc}(\mathsf{zero}, y) \to \mathsf{f}(y) \end{array}$$

This filtered TRS is not useful because of the extra variable in the third rule. Therefore, we propose to replace this variable by a fresh function $\bot$, so that the third rule becomes $\mathsf{dec} \to \bot$. In general, though, $\bot$ should be reducible to any ground constructor term to keep the correctness of this replacement (i.e., $\bot$ is nothing but a variant of gen), which makes it useless since any filtered TRS extended with such a definition for $\bot$ would be non-terminating by definition.

Luckily, in some cases, we may have a terminating definition for $\bot$. For instance, if the set $\{\pi(l) \mid l \to r \in \mathcal{R}\}$ is *shallow*, i.e., if all terms in this set have the form $\mathsf{f}(t_1, \ldots, t_n)$, where t_i is either a variable or a ground (constructor) term for all $i = 1, \ldots, n$. A TRS is called *left-shallow* if the set of its left-hand sides is shallow. In this case, it suffices to consider a (finite) definition of $\bot$ that reduces to any ground constructor term t_i in the above set. This is safe from the point of view of termination since, under the above conditions, $\bot$ will be reducible to any

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

(non-variable) argument in the left-hand side of the filtered rules and, thus, no potentially infinite derivation might be broken by the introduction of the fresh function $\bot$. E.g., for the example above, we have

$$\{\pi(l) \mid l \to r \in \mathcal{R}^c\} = \{\mathsf{f}(\mathsf{succ}(\mathsf{zero})),\ \mathsf{min},\ \mathsf{dec},\ \mathsf{inc}(\mathsf{zero}, y)\}$$

which is shallow. Hence the following definition for $\bot$ would suffice: $\bot \to \mathsf{zero}$, $\bot \to \mathsf{succ}(\mathsf{zero})$. The net effect is that we produced an *equivalent* DP problem

$$\left(\left\{ \begin{array}{l} \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{INC}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{DEC} \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{MIN} \\ \mathsf{INC}(\mathsf{zero}, y) \to \mathsf{F}(y) \end{array} \right\}, \left\{ \begin{array}{rl} \mathsf{f}(\mathsf{succ}(\mathsf{zero})) & \to \mathsf{inc}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{min} & \to \mathsf{zero} \\ \mathsf{dec} & \to \bot \\ \mathsf{inc}(\mathsf{zero}, y) & \to \mathsf{f}(y) \\ \bot & \to \mathsf{zero} \\ \bot & \to \mathsf{succ}(\mathsf{zero}) \end{array} \right\} \right)$$

that can be solved using standard techniques for the termination of rewriting (e.g., AProVE [6]). Such an example could not be proved terminating using the previous techniques because of the extra variable in the filtered rules.

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

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

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

where constants $\mathsf{c}()$ are simply denoted by c. We also denote by $\mathcal{R}_{\mathsf{gen}}$ a TRS over $\mathcal{F} \uplus \{\mathsf{gen}\}$ resulting from augmenting $\mathcal{R}$ with $\mathsf{GEN}(\mathcal{R})$, in symbols $\mathcal{R}_{\mathsf{gen}} = \mathcal{R} \cup \mathsf{GEN}(\mathcal{R})$.

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 $\sigma = \{x \mapsto \mathsf{gen} \mid x \in \mathcal{V}\mathsf{ar}(t)\}$. Note that $\widehat{t}$ is ground for any term t since all variables occurring in t are replaced by the function gen.

We say that the set of *reachable calls* from a given term are the subterms rooted by a defined function symbol (thus the name *call*) that occur in the rewrite derivations issuing from this term. Formally, given a TRS $\mathcal{R}$ and a term t, we define the set of reachable calls $\mathit{calls}_{\mathcal{R}}(t)$ from t in $\mathcal{R}$ as follows: $\mathit{calls}_{\mathcal{R}}(t) = \{ \ s|_p \ \mid \ t \to^*_{\mathcal{R}} s, \text{ with } \mathsf{root}(s|_p) \in \mathcal{D} \text{ for some position } p \ \}$. The following definition formalizes the notion of *narrowing chain*, a slight extension of the standard notion of chain in term rewriting in order to consider an initial set of terms (as specified by an abstract term) and an argument filtering:

Definition 2 (chain). *Let $\mathcal{R}$ and $\mathcal{P}$ be TRSs over the signatures $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ and $\mathcal{F}^\sharp = \mathcal{F} \cup \{f^\sharp \mid f/n \in \mathcal{D}\}$, respectively. Let π be an argument filtering over $\mathcal{F}$ that is extended over tuple symbols so that $\pi(f^\sharp) = \pi(f)$ for all $f \in \mathcal{D}$ and let t^α be an abstract term. A (possibly infinite) sequence of pairs $s_1 \to t_1$, $s_2 \to t_2$, ... from $\mathcal{P}$ is a $(t^\alpha, \mathcal{P}, \mathcal{R}, \pi)$-chain if there is a substitution $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that the following conditions hold:*[7]

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

– *there is a term* $s \in calls_{\mathcal{R}_{\mathsf{gen}}}(\widehat{t})$ *for some* $t \in \gamma(t^\alpha)$ *such that* $s^\sharp = \widehat{s_1\sigma}$ *and*
– $\widehat{t_i\sigma} \to^*_{\mathcal{R}_{\mathsf{gen}}} \widehat{s_{i+1}\sigma}$ *for every two consecutive pairs in the sequence and, moreover, we have* $\pi(\widehat{s_i\sigma}), \pi(\widehat{t_i\sigma}) \in \mathcal{T}(\mathcal{F}^\sharp)$ *for all* $i > 0$ *(i.e.,* π *filters away all occurrences of* gen*).*

Unfortunately, not all argument filterings are useful in our context. In the following, we focus on what we call *safe* argument filterings.

Definition 3 (safe argument filtering). *Let* $\mathcal{R}$ *be a TRS over a signature* $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$*. Let* π *be an argument filtering over* $\mathcal{F}$ *that is extended over tuple symbols so that* $\pi(f^\sharp) = \pi(f)$ *for all* $f \in \mathcal{D}$ *and let* $t^\alpha = \mathsf{f}(m_1, \ldots, m_n)$ *be an abstract term. We say that* π *is safe for* t^α *in* $\mathcal{R}$ *if the following conditions hold:*[8]

(1) $m_i = g$ *for all* $i \in \pi(\mathsf{f})$,
(2) $\mathcal{V}\mathsf{ar}(\pi(t)) \subseteq \mathcal{V}\mathsf{ar}(\pi(s))$ *for all rules* $s \to t \in \mathcal{R} \cup DP(\mathcal{R})$.

Observe that condition (2) above is equivalent to the variable condition in [6]. We argue however that it is too restrictive for narrowing and in the next section we introduce a transformation on the TRS $\mathcal{R}$ which allows for more freedom when constructing a safe argument filtering.

Once a safe argument filtering has been found, it is possible to transform the narrowing problem into a rewriting DP problem, as shown in [9, 12], and to solve it using one of the many solvers available nowadays.

2.2 The $\bot$-transformation

In the following, given a TRS $\mathcal{R}$ and an argument filtering π, we denote by $[\mathcal{R}]^\pi_\bot$ the TRS that results from $\mathcal{R}$ by replacing in $\mathcal{R}$ every extra variable of $\pi(\mathcal{R})$ with $\bot$. Formally, $[\mathcal{R}]^\pi_\bot = \{l \to r\sigma \mid l \to r \in \mathcal{R}$ with $\sigma(x) = \bot$ if $x \in \mathcal{V}\mathsf{ar}(\pi(r)) \setminus \mathcal{V}\mathsf{ar}(\pi(l))$ and $\sigma(x) = x$ otherwise $\}$.

As in [12], given an argument filtering π and a TRS $\mathcal{R}$, we replace the extra variables of $\pi(\mathcal{R})$ (if any) by the fresh symbol $\bot$, i.e., we replace $\pi(\mathcal{R})$ by $\pi([\mathcal{R}]^\pi_\bot)$. The novelty, then, is that we will also add a *safe* definition for $\bot$. What is a safe definition of $\bot$? The following condition characterizes this notion. In what follows, we denote by $s \to_{>p,\mathcal{R}} t$ any rewrite step $s \to_{q,\mathcal{R}} t$ with $s|_q$ a subterm of $s|_p$ (i.e., any rewrite step where a subterm below position p has been reduced). This notation is extended to rewrite derivations in the natural way.

Definition 4. *Let* $\mathcal{R}$ *be a TRS,* π *an argument filtering, and* t^α *an abstract term. We say that a set of rules* $\mathcal{R}^\bot$ *defining* $\bot$ *are safe for* t^α *if* $s \to^*_{>\epsilon,\mathcal{R}_{\mathsf{gen}}} s'$ *implies* $\pi(s) \to^*_{>\epsilon,\pi([\mathcal{R}]^\pi_\bot \cup \mathcal{R}^\bot)} \pi(s')$ *for all* $s \in calls_{\mathcal{R}_{\mathsf{gen}}}(\widehat{t})$ *with* $t \in \gamma(t^\alpha)$.[9]

The condition above is essential to guarantee that the connection between the dependency pairs of the original TRS is not lost. Of course, a definition of $\mathcal{R}^\bot$ which is analogous to that of gen is *always* safe. This definition, however, contains

[8] This is similar to the definition introduced in [12] and is slightly extended in [9].
[9] This is a strict extension of the notion of safeness in [9].

size-increasing rules as soon as the signature of $\mathcal{R}$ contains constructors of non-zero arity. There are cases, aside from the trivial case of no extra variables in $\pi(\mathcal{R})$, in which we can include a less aggressive definition of $\bot$ and still be able to ensure safeness. We are ready now to give a complete definition of our transformation, which always produces a safe definition of $\bot$.

Definition 5 ($\bot$-transformation). *Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$, π an argument filtering, t^α an abstract term, and $\bot$ a fresh symbol. Then we have*

$$\bot(\mathcal{R}, \pi, t^\alpha) = [\mathcal{R}]^\pi_\bot \cup \mathcal{R}^\bot$$

where $\mathcal{R}^\bot$ is defined as follows:

1. *If $\pi(\mathcal{R})$ does not contain extra variables, or it contains extra variables but do not play any role in the rewrite derivations connecting dependency pairs (see [9, Lemma 11] for a precise definition), then $\mathcal{R}^\bot = \{\ \}$.*
2. *Otherwise, if $\pi(\mathcal{R})$ is left-shallow, then $\mathcal{R}^\bot = \{l \to l|_p \mid l \to r \in \pi(\mathcal{R}), p \in \mathcal{P}os_\mathcal{F}(l),\ \text{and } p \neq \epsilon\ \}$.*
3. *In any other case, we have $\mathcal{R}^\bot = \{\ \bot \to \mathsf{c}(\overbrace{\bot, \ldots, \bot}^{n\ times})\ \mid\ \mathsf{c}/n \in \mathcal{C},\ n \geqslant 0\ \}$.*

The next lemma clarifies our interest in the above transformation:

Lemma 1. *Let $\mathcal{R}$ be a TRS, π an argument filtering, t^α an abstract term, and $\bot$ a fresh symbol. Then $\bot(\mathcal{R}, \pi, t^\alpha)$ enjoys the following properties:*

- *$\bot(\mathcal{R}, \pi, t^\alpha)$ contains no extra variables;*
- *the definition $\mathcal{R}^\bot$ of $\bot$ in $\bot(\mathcal{R}, \pi, t^\alpha)$ is safe according to Def. 4.*

If we look at the definition of $\bot(\mathcal{R}, \pi, t^\alpha)$ in detail, cases (1) and (2) are advantageous. Obviously (3) is the least desirable case, being $\mathcal{R}^\bot$ generally a non-terminating system (which does not necessarily imply an infinite narrowing problem, since we consider derivations starting from an initial term).

The following result states the soundness of our approach:

Theorem 1. *Let $\mathcal{R}$ be a left-linear constructor TRS over a signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ and let t^α be an abstract term. Let π be a safe argument filtering for t^α in $\bot(\mathcal{R}, \pi, t^\alpha)$ that is extended over tuple symbols so that $\pi(f^\sharp) = \pi(f)$ for all $f \in \mathcal{D}$. If there exists no infinite $(t^\alpha, DP(\mathcal{R}), \bot(\mathcal{R}, \pi, t^\alpha), \pi)$-chain, then $\gamma(t^\alpha)$ is $\leadsto_\mathcal{R}$-terminating.*

Now, we derive a sufficient condition for the termination of narrowing in terms of a standard dependency pair problem:

Theorem 2. *Let $\mathcal{R}$ be a left-linear constructor TRS over a signature $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$ and let t^α be an abstract term. Let π be a safe argument filtering for t^α in $\bot(\mathcal{R}, \pi, t^\alpha)$ that is extended over tuple symbols so that $\pi(f^\sharp) = \pi(f)$ for all $f \in \mathcal{D}$. If $(\pi(DP(\mathcal{R})), \pi(\bot(\mathcal{R}, \pi, t^\alpha)))$ is a finite DP problem, then there exist no infinite $(t^\alpha, DP(\mathcal{R}), \bot(\mathcal{R}, \pi, t^\alpha), \pi)$-chains.*

2.3 Mechanizing the approach

The problem of termination of narrowing boils down to computing a safe argument filtering and transforming the problem into a rewriting DP problem. The argument filtering employed determines the form of the resulting rewriting DP problem, and if too restrictive it effectively destroys the chances of succeeding in obtaining a termination proof.

The new component of our approach consists in replacing the initial TRS $\mathcal{R}$ with a new one, $\perp(\mathcal{R}, \pi, t^\alpha)$, which relaxes the second condition in the definition of a safe argument filtering, increasing the choice of safe argument filterings. However, $\perp(\mathcal{R}, \pi, t^\alpha)$ can include a set of rules $\mathcal{R}^\perp$ which may introduce increasing rules into the picture when case (3) of Def. 5 is considered. Hence it is only advisable to apply the transformation when the TRS considered falls into one of the *nice* cases (1) or (2).

The problem of computing a safe argument filtering is approached as a search problem. We propose the following strategy:

1. The starting point is a filtering π_0 which enforces the first condition in Def. 3.
2. Then, we proceed by progressively refining π_0 until a filtering π_1 is obtained which enforces the following condition: $\mathcal{V}\mathsf{ar}(\pi_1(t)) \subseteq \mathcal{V}\mathsf{ar}(\pi_1(s))$ for all dependency pairs $s \to t \in DP(\mathcal{R})$. Note that this is always possible, although there are multiple choices.
3. At this point we proceed depending on $\mathcal{R}$ and π_1:
   - If we are in case (1) or (2) of Def. 5, then perform the $\perp$-transformation and apply Theorem 2 (note that this is always possible as the transformation replaces all the extra variables).
   - Otherwise, then there is at least one extra variable in $\pi_1(\mathcal{R})$ and $\pi_1(\mathcal{R})$ is not left-shallow:
     - If there is an extra variable and there are one or more ways to refine π_1 to filter it away, then do so enforcing the condition in Step 2, and continue repeating Step 3.
     - If there is an extra variable left which cannot be filtered away, perform the transformation and apply Theorem 2.

In our running example, we have that $(\pi_0(DP(R)), \pi_0(R))$ is

$$\left(\left\{ \begin{array}{l} \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{INC}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{DEC} \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{MIN} \\ \mathsf{INC}(\mathsf{zero}, y) \to \mathsf{F}(y) \end{array} \right\}, \left\{ \begin{array}{r} \mathsf{f}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{inc}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{min} \to \mathsf{zero} \\ \mathsf{dec} \to \mathbf{x} \\ \mathsf{inc}(\mathsf{zero}, y) \to \mathsf{f}(y) \end{array} \right\} \right)$$

At this point there is an extra variable $\mathbf{x}$ left, which cannot be filtered away. We apply the transformation and since $\pi_0(\mathcal{R})$ is left-shallow, we get

$$\left(\left\{ \begin{array}{l} \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{INC}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{DEC} \\ \mathsf{F}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{MIN} \\ \mathsf{INC}(\mathsf{zero}, y) \to \mathsf{F}(y) \end{array} \right\}, \left\{ \begin{array}{r} \mathsf{f}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{inc}(\mathsf{dec}, \mathsf{min}) \\ \mathsf{min} \to \mathsf{zero} \\ \mathsf{dec} \to \perp \\ \mathsf{inc}(\mathsf{zero}, y) \to \mathsf{f}(y) \\ \perp \to \mathsf{zero} \\ \perp \to \mathsf{succ}(\mathsf{zero}) \end{array} \right\} \right)$$

which can easily be proved terminating (using, e.g., AProVE [6]).

3 Discussion

In this work, we improve a previous approach to the termination analysis of narrowing. Basically, our main achievement is relaxing the so-called variable condition, allowing for extra variables in cases where they are not harmful. This includes the case when the variables are not used in between pairs in a chain, and the case when the filtered TRS is left-shallow. This is a significant improvement over [12] and can also be applied to remove some extra variables within the termination analysis for logic programs of [10].

As for future work, we plan to formalize the notion of termination from an initial goal *in general*, which should pave the way for an adequate formalization of the approach presented in this abstract. Moreover we consider the use of static analysis on the TRS in order to compute an approximation of the possible values that an extra variable may take in a given narrowing computation. When this set of possible values is finite, even if the filtered TRS is not left-shallow, a more accurate definition for $\bot$ can be constructed.

References

1. S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proc. of ICLP'06*, pages 87–101. Springer LNCS 4079, 2006.
2. T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. J. de Dios-Castro and F. López-Fraguas. Extra Variables Can Be Eliminated from Functional Logic Programs. In *Proc. of the 6th Spanish Conf. on Programming and Languages (PROLE'06)*, pages 3–19. ENTCS 188, 2007.
5. S. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming*, 5:207–230, 1988.
6. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. of IJCAR'06*, pages 281–286. Springer LNCS 4130, 2006.
7. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proc. of LPAR'04*, pages 301–331. Springer LNCS 3452, 2005.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
9. N. Nishida and G. Vidal. Termination of Narrowing via Termination of Rewriting, 2009. Submitted for publication. Available from `http://users.dsic.upv.es/~gvidal/german/papers.html`.
10. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting, 2008. Submitted for publication.
11. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
12. G. Vidal. Termination of Narrowing in Left-Linear Constructor Systems. In J. Garrigue and M. Hermenegildo, editors, *Proc. of FLOPS 2008*, pages 113–129. Springer LNCS 4989, 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.

Key words: 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 [5,7,17] called TuLiP [9,8]. 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

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; there-

fore $\mathbf{t}$ 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 [18]. 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 $B_i\gamma_0, \ldots, \gamma_{i-1} \xrightarrow{\gamma_i}_P \Box$ 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*[3]. If there are free variables 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:*

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

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).

[3] 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).

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 focus on the semantics of *moded_bagof* when used 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 by the problems related to the termination of logic programs containing *moded_bagof* atoms as subgoals (we extend the use of *moded_bagof* to programs in the Section 6).

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

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.

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, $VarIn(A) = \{\texttt{W}\}$, $VarOut(A) = \{\texttt{X}, \texttt{Y}\}$, and $VarLocal(A) = \{\texttt{Z}\}$.

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 $B, \mathbf{C} \stackrel{\alpha}{\Longrightarrow}_P \mathbf{C}\alpha$.*
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 θ, then we say that ρ and c yield resolvent $(\mathbf{B}, \mathbf{C})\theta$. The corresponding derivation step is denoted by $B, \mathbf{C} \stackrel{\theta}{\Longrightarrow}_c (\mathbf{B}, \mathbf{C})\theta$.*

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. Such a policy can be modelled by the following rule (a credential in TuLiP):

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

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).
```

One of the things we try to handle in TuLiP is where to store the credentials so that they can be found later during the credential discovery. If we assume that *mode*$(read\_doc) =$ *mode*$(partner) =$ *mode*$(senior) =$ *mode*$(recommends) = (In, Out)$ then by the credential storage principles of TuLiP, all the credentials will be stored by their issuers (indicated by the first argument in an atom). TuLiP's Lookup and Inference AlgoRithm (LIAR) is guaranteed to find all the relevant credentials. Now, given the query `read_doc(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 access the document even though he has sufficient permissions.

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 [3,14]). Here we follow the approach of Etalle, Bossi, and Cocco [14].

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. [14]: 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* [16]. *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 $Pred(\mathit{Goal}) \not\simeq Pred(H)$.*

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 $Pred(\mathit{Goal})$ is contained within $\bigcup_{j<i} P_j$,
2. otherwise the definition of $Pred(B)$ 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 $\mathit{moded\_bagof}(t, gl, Goal, x)$ selected in δ, $P \cup Goal$ 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 [14] (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 [14]) *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 [14], 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 $y = \mathbb{F}(\llbracket\, E(\mathbf{x},\mathbf{z}) : \exists(\mathbf{z})p(\mathbf{x},\mathbf{z})\,\rrbracket)$. 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. [18]. In this work, aggregate stratification is used to prevent recursion through aggregates. Later, Kemp and Stuckey [16] 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. [15] 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. [11] 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 [10]. The work of Denecker et al. is continued and further extended by Pelov et al. [19].

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: $m = \llbracket\, E(\mathbf{x},\mathbf{z}) : \exists(\mathbf{z})p(\mathbf{x},\mathbf{z})\,\rrbracket, y = \mathbb{F}(m)$. 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 [13] 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. [12] 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. [12]. 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. [13,12] 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 [2,20]. 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. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.
3. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

4. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
5. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. 17th IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.
6. P. Ciancarini, D. Fogli, and M. Gaspari. A Logic Language based on GAMMA-like Multiset Rewriting. In *Extensions of Logic Programming (ELP)*, volume 1050 of *LNCS*, pages 83–101. Springer, March 1996.
7. D. Clarke, J.E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.
8. M. R. Czenko, J. M. Doumen, and S. Etalle. Trust Management in P2P Systems Using Standard TuLiP. In *Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security, Trondheim, Norway*, volume 263/2008 of *IFIP International Federation for Information Processing*, pages 1–16, Boston, May 2008. Springer.
9. M. R. Czenko and S. Etalle. Core TuLiP - Logic Programming for Trust Management. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal*, volume 4670 of *Lecture Notes in Computer Science*, pages 380–394, Berlin, October 2007. Springer Verlag.
10. M. Denecker, V. Marek, and M. Truszczyński. *Approximations, Stable Operators, Well-Founded Operators, Fixpoints and Applications in Nonmonotonic Reasoning*, volume 597 of *The Springer International Series in Engineering and Computer Science*, chapter 6, pages 127–144. Springer, 2001.
11. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates. In *ICLP*, volume 2237 of *LNCS*, pages 212–226. Springer, 2001.
12. A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A logic programming language with finite sets. In *ICLP*, pages 111–124. MIT Press, 1991.
13. A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. In *Logic Programming*, volume 2916 of *LNCS*, pages 284–299, Berlin, 2003. Springer.
14. S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *J. Log. Program.*, 38(2):243–257, 1999.
15. W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Logics in Artificial Intelligence (JELIA)*, volume 3229 of *LNCS*, pages 200–212. Springer, 2004.
16. D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *ISLP*, pages 387–401. MIT Press, 1991.
17. N. Li, J. Mitchell, and W. Winsborough. Design of a Role-based Trust-management Framework. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.
18. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proc. 16th International Conference on Very Large Databases*, pages 264–277. Morgan Kaufmann Publishers Inc., 1990.
19. N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.
20. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, 1995. available at 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.
Key Words: 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).

Unlike positive unfolding, however, PP-negative unfolding does not always preserve the semantics of a given program in general, when used with unfold/fold

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

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 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

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

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*). We assume that every primitive clause in an initial program remains *untransformed* at any step in the transformation sequence (Definition 8).

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.

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. 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 $m := \max\{\sigma(bd(C)) \mid C \in Def(p, P_0)\}$.

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

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. In the original framework [12], each predicate in an initial program[3] is classified as either *old* or *new*, thus the number of levels is two. The definition of a new predicate consists of a single clause whose body contains positive literals with old predicates only. Therefore, a recursive definition of a new predicate is not allowed. Moreover, it has no primitive predicates. The above definition follows the generalized framework in [13], thus eliminating such limitations. □

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.

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

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) of the proofs of an atom true in P_0. The following definition of the well-founded measure μ is a natural extension of that in [13] for definite programs, where μ is defined in terms of an SLD-derivation.

Definition 9. Weight-Tuple, Well-founded Measure μ

Let P_0 be an initial program with the maximum level I and A be a ground atom true in $M(P_0)$. Let T be a proof of A by 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:

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

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 $M\}$. □

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

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 $M(P_0)$, $\mu(A) = \langle 0, \ldots, 0 \rangle$ (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. folding consistent with the initial stratification

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, 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 consistent with the initial stratification.

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 consistent with σ. □

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.

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. 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 *folding condition*, if the descent level of C is *smaller* than the level of D. □

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

Definition 13. Replacement Rules Consistent with σ and μ

Let R be a replacement rule of the form $M_1 \Rightarrow M_2$, which is correct w.r.t. initial program P_0. Then, R is said to be *consistent* with the initial stratification σ if $\sigma(M_1) \geq \sigma(M_2)$, and it is said to be *consistent* with the well-founded measure μ 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. In their case, literals appearing in the replacement rule are primitive. Then, the consistency with σ is trivial, since $\sigma(M_1) = \sigma(M_2) = 0$ by definition, and the consistency with μ is due to the fact that the weight-tuple of a proof of $M_i\theta$ is $\langle 0, \ldots, 0\rangle$ (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:

$$P_0 = \{\ D: \begin{array}{l} f \leftarrow m, \neg e \\ m \leftarrow \\ e \leftarrow e \\ e \leftarrow \neg m.\ \} \end{array} \qquad \begin{array}{lll} (1): & f \leftarrow \neg e & (\text{pos. unfolding } D) \\ (2): & f \leftarrow \neg e, m & (\text{neg. unfolding } (1)) \\ (3): & f \leftarrow f & (\text{folding } (2)) \end{array}$$

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 14. Negative Unfolding Consistent with μ

The application of negative unfolding is said to be *consistent* with μ, 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, the consistency of negative unfolding with μ when applied to clause (1), requires that m be primitive. Then, this prohibits the subsequent folding operation in clause (3) from the folding condition (Def. 12).

One way to view our condition on negative unfolding is that negative unfolding is a special case of replacement, i.e., a replacement rule R of the form $\neg A \Rightarrow Q_i$, where Q_i is given in Def. 3. Note that, when $M(P_0) \models \neg A$, it holds that $\mu(\neg A) = \langle 0, \ldots, 0 \rangle$. Therefore, $\mu(Q_i)$ should be $\langle 0, \ldots, 0 \rangle$ in order for R to be consistent w.r.t. μ. This means that every positive literal (if any) in Q_i is primitive, which is exactly what Def. 14 requires.

Remark 2. We note that, unlike negative unfolding, folding is not an instance of replacement in our framework. In Def. 5, the application of reversible folding replaces a molecule M in the body of the folded clause by an atom A, where M is a P-expansion of A by some folding clause D in P_0. Then, we have from the definition of μ that $\mu(M\theta) < \mu(A\theta)$ for any ground substitution θ such that $M\theta$ and $A\theta$ are provable by P_0. [7] As this replacement of M by A is not consistent with μ, it does not satisfy our conditions imposed on replacement rules (Def. 13). This is the reason why we need the extra condition on folding in Def. 12. □

The following shows the correctness of our transformation system.

Proposition 3. Correctness of Transformation

Let P_0 be an initial program 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 consistent with σ and satisfies the folding condition (Def. 12), and (ii) every application of replacement rule is consistent with σ and μ. Moreover, suppose that every application of negative unfolding is consistent with μ. Then, $M(P_n) = M(P_0)$. □

[7] We assume from the folding condition (Def. 12) that A is non-primitive.

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 $M(P) \models \forall \tilde{X}(\mathcal{G} \rightarrow \mathcal{H})$[8], 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 $hd(C)$.

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].

[8] As noted in [4], the use of the term *coinduction* here has no relationship with the *greatest* fixed point of a program.

Fig. 1. Coinductive Proof Rules by Jaffar et al. [4]

$$\text{(LU+I)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}} \quad \text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$$

$$\text{(RU)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}} \quad \mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$$

$$\text{(CO)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\emptyset \vdash \mathcal{H}'\theta \models \mathcal{H}\}} \quad \mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}'\theta$$

$$\text{(CP)}\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}} \qquad \text{(DP)}\ \frac{\Pi \uplus \{\mathcal{G} \models \mathcal{H}\}}{\Pi}\ \mathcal{G} \models \mathcal{H} \text{ holds by constraint solving}$$

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 15. 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).

Fig. 2. A Coinductive Proof of Example 2 and the Corresponding Proof via Unfold/fold Transformations

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\emptyset \vdash p(X) \models r(X)}{\{A_1\} \vdash q(X) \models r(X)}\ (\mathsf{LU{+}I})}{\{A_1, A_2\} \vdash q(X) \models r(X)}\ (\mathsf{LU{+}I})}{\emptyset \vdash r(X) \models r(X)}\ (\mathsf{CO})}{\models X = X}\ (\mathsf{CP})}{true}\ (\mathsf{DP})$$

$$\begin{array}{rll}
C_f: & f \leftarrow \neg nf_1 & \\
C_{nf_1}: & nf_1 \leftarrow p(X), \neg r(X) & \\
C_{nf_2}: & nf_2(X) \leftarrow q(X), \neg r(X) & \\
\hline
(1): & nf_1 \leftarrow q(X), \neg r(X) & (\text{pos. unfolding } C_{nf_1}) \\
(2): & nf_1 \leftarrow nf_2(X) & (\text{folding } (1)) \\
(3): & nf_2(X) \leftarrow q(X), \neg r(X) & (\text{pos. unfolding } C_{nf_2}) \\
(4): & nf_2(X) \leftarrow nf_2(X) & (\text{folding } (3))
\end{array}$$

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

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 17. 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, thus their correctness proof in [9] will be unavailable in the presence of negative unfolding. 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 J. van Leeuwen, (Eds.) *Handbook of Theoretical Computer Science*, pp. 493-576, Elsevier, 1990.
2. Aravindan, C. and 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. and Proietti, M., Transformation Rules for Locally Stratified Constraint Logic Programs, Maurice Bruynooghe et al. (Eds.) *Program Development in Computational Logic*, LNCS 3049, Springer, pp. 291-339, 2004.
4. Jaffar, J., Santosa, A. and Voicu, R., A Coinduction Rule for Entailment of Recursively Defined Properties, *14th Int'l. Conf. on Principles and Practice of Constraint Programming*, LNCS 5202, Springer, pp. 493-508, 2008.
5. Kanamori, T. and K. Horiuchi, Construction of Logic Programs Based on Generalized Unfold/Fold Rules, *Proc. the 4th Intl. Conf. on Logic Programming*, pp. 744-768, 1987.
6. Lloyd, J. W. , *Foundations of Logic Programming*, Springer, 1987, Second edition.
7. Pettorossi, A. and Proietti, M., Transformation of Logic Programs: Foundations and Techniques, *J. of Logic Programming*, 19/20:261-320, 1994.
8. Pettorossi, A. and Proietti, M., Perfect Model Checking via Unfold/Fold Transformations, J. W. Lloyd et al. (Eds.) *Proc. 1st Int. Conf. on Computational Logic*, LNAI 1861, Springer, pp. 613-628, 2000.
9. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C. R. and Ramakrishnan, I. V., Beyond Tamaki-Sato Style Unfold/fold Transformations for Normal Logic Programs, *Int. Journal on Foundations of Computer Science* 13(3), 387-403, 2002.
10. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C. R. and Ramakrishnan, I. V., An Unfold/fold Transformation Framework for 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, M. Hanus (Ed.) *Proc. 18th Int. Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, LNCS 5438, Springer, pp. 168-184, 2008.
12. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, *Proc. 2nd Int. Conf. on Logic Programming*, 127-138, 1984.
13. Tamaki, H. and Sato, T., A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, *Technical Report*, No. 86-4, Ibaraki Univ., 1986.

Coinductive Logic Programming with Negation

Richard Min and Gopal Gupta

Department of Computer Science
The University of Texas at 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.

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].

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* 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 □.
(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 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 □.
(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 □, (b) nt(A, □, B) reduces to nt(A, B) where A and B represent conjunction of subgoals, and (c) nt(□) 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 (□). (ii) When an initial query goal reduces to an empty clause (□), 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]). □

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, $nt(D_1, (S_1(\mathbf{x}_1)\theta_1; \ \ldots \ ; S_n(\mathbf{x}_n)\theta_n), D_2)$ is used for $(nt(D_1, S_1(\mathbf{x}_1)\theta_1, D_2), \ldots, nt(D_1, S_n(\mathbf{x}_n)\theta_n, D_2)))$ 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 next 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) $[SS]=\{ A \mid A \in HB^R(P)$, the goal $\{ A \} \rightarrow^* \square \}$

(2) $[FF]=\{ A \mid A \in HB^R(P)$, the goal $\{ nt(A) \} \rightarrow^* \square \}$,

where $\rightarrow^*$ 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 Some Illustrating 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. 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 $\chi+$ and $\chi-$ 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 MP2A={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:

$T_P^+(M,N) = \{head(R) \mid R \in HG^R(P), pos(R) \subseteq M, neg(R) \subseteq N\}$,

$T_P^-(M,N) = \{A \mid \forall R \in HG^R(P), head(R)=A \rightarrow pos(R) \cap N \neq \varnothing \vee neg(R) \cap M \neq \varnothing\}$

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 $T_P(I) = \{ head(R) \mid R \in HG^R(P), I \models body(R) \}$ 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 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, to be labeled *undefined* (even though it is meant for 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 $M' \cap N' = \varnothing$. If (M', N') is a fixed point of $< T_P^+, T_P^- >$, $M' \cap N' = \varnothing$ 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, $M \cap N = \varnothing$) 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 ($q \rightarrow p(a) \rightarrow p(f(a)) \rightarrow p(f(f(a))) \rightarrow \ldots$) 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)\$HM^R$(IP1) as there is no rational derivation tree for **q** and **p(a)**. Further, for the second clause { **p(X) :- p(f(X).** }, there is only one ground (rational) atom **p(X′),** where X′=f(X′)=f(f(f(...))), which satisfies the clause and makes **p(X)** true; all other finite or rational atoms **p(Y)** are false. Thus the ground atom **p(f(f(f(...))))** (**p(X)** where **X=f(X)**) is in HM^R(P), and all other finite and rational atoms **p(Y)** where **Y ≠ f(Y)** should be in HB^R(P)\$HM^R$(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 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 of the exploratory and exemplary 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 { $\wedge$, $\vee$, $\neg$} 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 ("$\wedge$")will be translated into "," (Prolog's AND-operator), while the OR (or "$\vee$") operator will be translated to ";" (Prolog's OR-operator). Thus, the Boolean expression (p1 $\vee$ p2) $\wedge$ (p1 $\vee$ $\neg$ p3) $\wedge$ ($\neg$ p2 $\vee$ $\neg$ 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 $\neg$p1 as **nt(t(p1))**, (3) (p1$\wedge$p2) as **(t(p1), t(p2))**, (4) (p1$\vee$p2) as **(t(p1); t(p2))**, (5) (p1 $\vee$ p2) $\wedge$ (p1 $\vee$ $\neg$ p3) $\wedge$ ($\neg$ p2 $\vee$ $\neg$ 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 (1988)
[2] Barwise, J., Moss, L.: Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena. 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, pp. 51-60 (1994)
[6] Fitting, M.: A Kripke-Kleene Semantics for Logic Programs. Journal of Logic Programming 2, pp. 295-312 (1985)
[7] Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications. (Tutorial Paper). In, Proc. of ICLP07, 27-44. (2007)
[8] Jaffar, J., J-L. Lassez, 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 (1986)
[9] Jaffar, J., Stuckey, P.: Semantics of Infinite Tree Logic Programming. Theoretical Computer Science 46(2-3), pp. 141-158 (1986)
[10] Lloyd, J.W.: Foundations of Logic Programming. Springer (1987)
[11] Maher, M.J.: Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In: Proc. 3rd Logic in Computer Science Conf., 348-357. Edinburgh, UK (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'09 (2009)
[14] Min, R., Gupta, G.: Coinductive Logic Programming and Its Application to Boolean Sat. In: Flairs'09 (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. (1988)
[16] Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming. In, Proc. ICALP'07, 472-483. (2007)
[17] Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive Logic Programming. In: ICLP'06. LNCS 4079, 330-344. (2006)

New Development in Extracting Tail Recursive Programs from Proofs

Luca Chiarabini[1] and Philippe Audebaud[2]

[1] Mathematisches Institut, Ludwig-Maximilians-Universität München
Theresienstr. 39, D-80333 München, Germany
(chiarabi@mathematik.uni-muenchen.de)
[2] LIP - ENS Lyon
46 alle d'Italie, 69437 Lyon, France
(Philippe.Audebaud@ens-lyon.fr)

Abstract. Transforming a recursive procedure into a tail recursive one brings many computational benefits; in particular in each recursive call there is no context information to store. In this paper we consider the particularly simple induction schema over natural numbers, and we propose two methods to automatically turn it into another proof with tail recursive content: one *continuation* and one *accumulator* based.

Key words: Functional Programming, Program extraction from constructive proofs, Program development by proof transformation, CPS-Transformation, Defunctionalization

1 Introduction

Let M be a proof by induction over n (natural number) of the property $\forall n.\varphi(n)$, and let, by the *Proofs-as-Program* paradigm, $[\![M]\!]$ be the (recursive) content of M. In this paper we will try to answer the following question: *How to turn M automatically into another proof N with tail recursive content for the same statement?* Penny Anderson in her Ph.D. thesis [1] used Frank Pfenning's *Insertion Lemma* [16] proof transformation, in order to extract *tail recursive* programs from proofs. This method, although particularly interesting, is *user dependent*. What we will do here is to present and develop in a formal setting an idea first roughly introduced in [4] (originated from an informal chat the first author had with Andrej Bauer in the spring of 2004, reported in Bauer's mathematical blog [3]) in order to extract tail recursive programs from proofs but in a completely automatic fashion.

Let us consider the following program, written in an ML-like syntax:

```
let rec FACT n = if n = 0 then 1 else n * FACT (n - 1)
```

FACT computes the factorial of n, for any positive integer n. But this implementation is not *tail* recursive because in each step of the computation the compiler has to store (on a stack) the context $(n * [])$, evaluate `FACT (n-1)` $\mapsto v$, and

return $(n * v)$. It is well known that `FACT` can be turned into a simpler function where it is not necessary to stack any context information:

```
let rec FACT' n =
  let rec FACT'' n m y =
    if n = 0 then y else FACT'' (n - 1) (m + 1) ((m + 1) * y)
  in FACT'' n 0 1
```

Now assume `FACT` to be the computational content of the proof by induction M, with end formula $\forall n \varphi(n)$, that states that for each natural n there exists $n!$. From which proof is it possible to extract `FACT'`? Both programs `FACT` and `FACT'` compute the factorial function, so `FACT'` should be the content of an appropriate proof of $\forall n \varphi(n)$ as well. So the problem has shifted to understanding which property satisfies `FACT''`. Given a natural n, (`FACT''`n) is a function that takes the natural m, the witness y for $\varphi(m)$ and returns a witness for $\varphi(n+m)$. Hence given n, (`FACT''`$n\,0\,1$) is the witness for $\varphi(n)$ as expected. Intuitively, we expect `FACT''` to be the computational content of some proof of the formula $\forall n, m(\varphi(m) \rightarrow \varphi(n+m))$ In this article we will show that this is the right intuition to follow for the automatic generation of tail recursive programs.

The paper is organized as follows: section 2 is a short introduction to the logical foundation of program extraction and to Gödel T, in Section 3 we address two proofs transformation in order to extract *continuation* and *accumulator* based tail recursive programs, in Section 4 we show that there exists a formal connection between the two proof transformations presented in Section 3 and finally, in Section 5 we apply our methods to a well known problem in bioinformatics, the *Maximal Scoring Subsequence Problem*. All the proofs presented in the paper are developed with the MINLOG proof assistant [18].

2 Logical Foundations

2.1 Program Extraction in MINLOG

MINLOG (`www.minlog-system.de`) is a proof assistant intended to reason about computable functions of finite type using minimal logic. A major aim of the Minlog project is the development of practically useful tools for the machine-extraction of realistic programs from proofs.

The method of program extraction implemented in MINLOG is based on modified realizability as introduced by Kreisel [14]. In short, from every constructive proof M of a non-Harrop formula A (in natural deduction proof calculus) one extracts a program $[\![M]\!]$ "realizing" A, essentially, by removing computationally irrelevant parts from the proof (proofs of Harrop formulas have no computational content). The extracted program (Gödel T with types) has some simple type $\tau(A)$ which depends on the logical shape of the proved formula A only.

Besides the usual quantifiers, $\forall$ and $\exists$, MINLOG has so-called non-computational quantifiers, $\forall^{\mathsf{nc}}$ and $\exists^{\mathsf{nc}}$ which allow for the extraction of simpler programs. Intuitively, a proof of $\forall^{\mathsf{nc}} A(x)$ ($A(x)$ non-Harrop) represents a procedure that

assigns to every x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make "computational use" of x, i.e. the extracted program $[\![M(x)]\!]$ does not depend on x. Dually, a proof of $\exists^{\mathsf{nc}}A(x)$ is proof of $M(x)$ for some x where the witness x is "hidden", that is, not available for computational use.

For a complete documentation on the extraction of programs in MINLOG please refer to [20].

2.2 Gödel's T with Types

Types are built from base types $\mathbf{N}$ (Naturals) , $\mathbf{L}(\rho)$ (lists with elements of type ρ) and $\mathbf{B}$ (booleans) by function ($\to$) and pair ($\times$) formation. The *Terms* of Gödel's T [21] are simply typed λ-calculus terms with pairs, projections (π_i) and constants (constructors and recursive operators for the basic types)

$$\begin{aligned} \mathit{Types}\ \rho,\sigma &::= \mathbf{N} \mid \mathbf{B} \mid \mathbf{L}(\rho) \mid \rho \to \sigma \mid \rho \times \sigma \\ \mathit{Const}\ c &::= 0^{\mathbf{N}} \mid \mathsf{Succ}^{\mathbf{N}\to\mathbf{N}} \mid \mathsf{tt}^{\mathbf{B}} \mid \mathsf{ff}^{\mathbf{B}} \mid (:)^{\mathbf{L}(\rho)} \mid ::^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)} \mid \mathcal{R}^{\sigma}_{\mathbf{N}} \mid \mathcal{R}^{\sigma}_{\mathbf{L}(\rho)} \mid \mathcal{R}^{\sigma}_{\mathbf{B}} \\ \mathit{Terms}\ r,s,t &::= c \mid x^{\rho} \mid (\lambda x^{\rho} r^{\sigma})^{\rho\to\sigma} \mid (r^{\rho\to\sigma} s^{\rho})^{\sigma} \mid (\pi_0 t^{\rho\times\sigma})^{\rho} \mid (\pi_1 t^{\rho\times\sigma})^{\sigma} \mid (r^{\rho}, s^{\sigma})^{\rho\times\sigma} \end{aligned}$$

The expression (:) represents the empty list, and $(a_0 :: \ldots :: a_n :)$ a list with $n+1$ elements. We equip this calculus with the usual conversion rules for the recursive operators, applications and projections.

The η-reduction relation $\lambda x(r\ x) \longrightarrow_{\eta} r$ is defined for $x \notin \mathrm{FV}(r)$. By $\longrightarrow_{\mathcal{R}\eta\beta}$ we indicate the union of $\longrightarrow_{\beta}$, $\longmapsto$, and $\longrightarrow_{\eta}$. By $\longrightarrow^{+}_{\mathcal{R}\eta\beta}$ we indicate the transitive closure of $\longrightarrow_{\mathcal{R}\eta\beta}$. Finally we define the *extensional equality* relation $=_{\mathcal{R}\eta\beta}$, as the least equivalence relation that contain $\longrightarrow_{\mathcal{R}\eta\beta}$. The *extensional equality* relation captures the idea that two functions should be considered equal if they yield equal results whenever applied to equal arguments.

We already emphasised that realizability is extensively used in MINLOG. In this proof assistant, extracted programs are presented in a textual style, that we briefly describe now along with the correspondence with the above mathematical notations: in programs produced by MINLOG, tt and ff are typeset `#tt` and `#ff` respectively; $\lambda x.t$ written `([x]t)`, $(\mathcal{R}^{\sigma}_{\mathbf{N}/\mathbf{B}/\mathbf{L}(\rho)}\, b\, s)$ as `(Rec (nat/bool/list rho => sigma) b s)` and $(\pi_{0/1} e)$ as `(left/right e)`. Finally the term $(\mathcal{R}^{\sigma}_{\mathbf{B}}\, r\, s)t$ is printed as `(if t r s)`.

3 Proof Manipulation

This section is devoted to exposing the proof transformations we have in mind in order to generate (by extraction) more efficient programs starting with a given inductive proof on natural numbers. How the techniques can be extended to other data types is discussed in the conclusion.

Definition 3.1 (Tail Expressions [12]) *The tail expressions of $t \in \mathit{Terms}$, are defined inductively as follows:*

1. *If $t \equiv (\lambda x.e)$ then e is a tail expression.*

2. *If* $t \equiv (\mathsf{if}\, t\, r\, s)$ *is a tail expression, then both* r *and* s *are tail expressions.*
3. *If* $t \equiv (\mathcal{R}_\iota\, r\, s)$ *is a tail expression, then* r *and* s *are tail expressions.*
4. *Nothing else is a tail expression,*

where $\iota \in \{\mathbf{N}, \boldsymbol{L}(\rho)\}$.

Definition 3.2 *A tail call is a tail expression that is a procedure call.*

Definition 3.3 (Tail Recursion [13]) *A recursive procedure is called tail recursive when its tail calls itself or calls itself indirectly through a series of tail calls.*

Now, let us consider F be the following proof by induction over $\mathbf{N}$:

$$\frac{\begin{array}{cc} |M & |N \\ \varphi(0) & \forall n(\varphi(n) \to \varphi(n+1)) \end{array}}{\forall n \varphi(n)}$$

The content of F is $(\mathcal{R}^\sigma_{\mathbf{N}}\, b\, f)$ with b and f base and step case of the recursion operator, content of the proofs M and N.

3.1 *Continuation* based Tail Recursion

Given the procedure $(\mathcal{R}^\sigma_{\mathbf{N}}\, b\, f)$ defined in the previous section, let Λ be the term:

$$(\mathcal{R}^{(\sigma\to\sigma')\to\sigma'}_{\mathbf{N}}\, \lambda k(kb) \quad \lambda n, p, k(\, p\, \lambda u(k(f\, n\, u))))$$

The procedure Λ, by Definition 3.3, is tail recursive. The first input argument of Λ, which has type $(\sigma \to \sigma')$, is called a *continuation*; Λ is a function with just one tail recursive call and a functional accumulator parameter k with the following property: for each n, at the i-th $(0 < i \leq n)$ step of the computation of $(\Lambda\, n\, (\lambda x.x))$ the continuation has the form $\lambda u(f\, (n-1)\, (\ldots (f\, (n-i)\, u)\ldots))$. At the n-th step the continuation $\lambda u(f\, (n-1)\, (\ldots (f\, 0\, u)\ldots))$ is applied to the term b and returns. We see that such returned value corresponds to $(\mathcal{R}^\sigma_{\mathbf{N}}\, b\, f)n$. This fact is stated formally in the following,

Theorem 3.1 *For each natural* n*:*

$$\Lambda\, n =_{\mathcal{R}\eta\beta} \lambda k^{\sigma\to\sigma'} k((\mathcal{R}^\sigma_{\mathbf{N}}\, b\, f)n)$$

Proof. By induction over n:

As expected, when applied to the *identity continuation* $(\lambda x\, x)$ we get another program in the same equivalence class:

Corollary 3.1 $\lambda n(\Lambda\, n\, (\lambda x\, x)) =_{\mathcal{R}\eta\beta} (\mathcal{R}^\sigma_{\mathbf{N}}\, b\, f)$

Now we show how $\lambda n(\Lambda\, n\, (\lambda x\, x))$ can be synthesized, in an automatic way, from another proof of the same given statement $\forall n \varphi(n)$. More formally, assume we are given some proof term F, with extraction $[\![F]\!] = (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$, is it possible to find *another* proof F' of the same statement, which leads to the other program: $[\![F']\!] = \lambda n(\Lambda\, n\, (\lambda x\, x))$? This is the challenge we give a positive answer hereafter.

The key point is to understand the logical role of the continuation parameter in Λ: given a natural n, at each step $i : n, \ldots, 0$ in computing $(\Lambda\, n\, (\lambda x.x))$, the continuation is a function that takes the witness for $\varphi(i)$ and returns the witness for $\varphi(i+m)$, for m such that $i+m = n$. So we expect Λ to be the computational content of a proof with end formula:

$$\forall n \forall^{\mathsf{nc}} m((\varphi(n) \to \varphi(n+m)) \to \varphi(n+m)) \tag{1}$$

We observe that the counter m is introduced to count how much n is *decreasing* during the computation. So, as such, it plays a "logical" role (or commentary role if one prefers); in other words, it is irrelevant at the programming level, and should be marked to be dropped out. To this end, we explicitly underline the "hidden" role of m quantifying over it by the special *non-computational* quantifier $\forall^{\mathsf{nc}}$[20, page 47]. Let us prove the above statement (1), under the assumptions we have proofs for both $\varphi(0)$ and $\forall n(\varphi(n) \to \varphi(n+1))$ statements.

Proposition 3.1 $\varphi(0) \to \forall n(\varphi(n) \to \varphi(n+1)) \to \forall n \forall^{\mathsf{nc}} m((\varphi(n) \to \varphi(n+m)) \to \varphi(n+m))$

Proof. Assume $b : \varphi(0)$ and $f : \forall n(\varphi(n) \to \varphi(n+1))$. By induction on n.

$n = 0$ We have to prove

$$\forall^{\mathsf{nc}} m((\varphi(0) \to \varphi(m)) \to \varphi(m))$$

So assume m and $k : (\varphi(0) \to \varphi(m))$. Apply k to $b : \varphi(0)$.

$n+1$ Assume n, the recursive call $p : \forall^{\mathsf{nc}} m((\varphi(n) \to \varphi(n+m)) \to \varphi(n+m))$, m, and the continuation $k : \varphi(n+1) \to \varphi(n+m+1)$. We have to prove:

$$\varphi(n+m+1)$$

Apply p to $(m+1)$ obtaining $(p\,(m+1)) : (\varphi(n) \to \varphi(n+m+1)) \to \varphi(n+m+1)$. So, if we are able to prove the formula $\varphi(n) \to \varphi(n+m+1)$, by some proof t, we can just apply $(p\,(m+1))$ to t and we are done.
So let us prove

$$\varphi(n) \to \varphi(n+m+1)$$

Assume $v : \varphi(n)$. We apply k to $(f\, n\, v)$.

Corollary 3.2 $\varphi(0) \to \forall n((\varphi(n) \to \varphi(n+1))) \to \forall n \varphi(n)$.

Proof. Assume $b : \varphi(0)$, $f : \forall n(\varphi(n) \to \varphi(n+1))$ Given n, to prove $\varphi(n)$, we instantiate the formula proved in Proposition 3.1 on b, f, n, 0 and $\varphi(n) \to \varphi(n)$.

Extracted program 1 Ind_CONT

```
[b,f,n](Rec nat => (sigma => sigma') => sigma'
        [k](k b)
        [n,p,k]p ([u] k(f n u))) n ([x]x)
```

We name the corresponding extracted term Ind_CONT (Extracted program 1).

Notice that, although the functional parameter in Λ is a *continuation*, Λ is not of the kind provided alongside a CPS-transformation[10] of the recursion over naturals schemata. In fact f and b are not altered in our transformation and they could contain *bad* expressions, like not tail calls.

The formula (1) could be substituted by the more general $\forall n((\varphi(n) \to \bot) \to \bot)$. However, we offer a simpler formulation for the logical property the continuation parameter is supposed to satisfy. In addition, this approach represents a non trivial usage of the *non* computational quantifiers $\forall^{\mathsf{nc}}$.

3.2 *Accumulator* based tail recursion

Here we present the essence of Bauer's [3] original idea. Given the procedure $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$ defined in the last section, let $\mathit{\Pi}$ be the term:

$$(\mathcal{R}^{\mathbf{N}\to\sigma\to\sigma}_{\mathbf{N}}\ \lambda m, y(y) \quad \lambda n, p, m, y(p\,(m+1)\,(f\,m\,y)))$$

In $\mathit{\Pi}$ there are two accumulator parameters: a natural and parameter of type σ where intermediate results are stored. For each natural n, at the i-th $(0 < i \leq n)$ step of the computation of $(\mathit{\Pi}\, n\, 0\, b)$ the accumulator of the partial results will be equal to the expression $(f\,(i-1)\,(\ldots(f\,0\,b)\ldots))$. At the n-th step (base case of $\mathit{\Pi}$) the accumulator of the partial results is returned and it corresponds to $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n$. This fact is stated in theorem 3.2 below.

Definition 3.4 *For all n,m, let* $\overline{f}^{\mathbf{N}\to\mathbf{N}\to\sigma\to\sigma}$ *be a function such that:*

$$\overline{f}_m\, n = f\,(n+m)$$

Proposition 3.2 *For all naturals* n *and* m*:*

$$(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\, 0\, b)\,\overline{f}_{m+1})\, n =_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\,\overline{f}_m)\,(n+1)$$

Proof. By induction on n.

Theorem 3.2 *For all natural* n*,*

$$\mathit{\Pi}\, n =_{\mathcal{R}\eta\beta} \lambda m, y((\mathcal{R}^{\sigma}_{\mathbf{N}}\, y\,\overline{f}_m)\, n)$$

Proof. By induction on n.

Now, compared with the previous step, we have to provide an *initial value* to Π in order to get an equivalent program. According to the accumulator-based approach, arguments $0, b$ roughly take the place of the continuation (function). See section 4 for more development on this remark.

Corollary 3.3 $\lambda n(\Pi\, n\, 0\, b) =_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$

Again, we still have to address the question, whether being given a proof F such that

$$[\![F]\!] = (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$$

it is possible to find out F' such that:

$$[\![F']\!] = \lambda n(\Pi\, n\, 0\, b)?$$

Functions are very powerful tools, so it is not a surprise that going along without them has a cost. Actually, we can still achieve our goal, but the answer is now a little bit more elaborate.

Given two natural indices i , j, with $i+j = n$, $(\Pi\, i\, j)$ is a function that takes the witness for $\varphi(j)$ and returns the witness for $\varphi(i + j)$. So we expect Π to be the computational content of a proof with end formula:

$$\forall n, m(\varphi(m) \to \varphi(n + m))$$

that uses the proofs terms $M^{\varphi(0)}$ and $N^{\forall n(\varphi(n)\to\varphi(n+1))}$ as assumptions. Let us prove this claim.

Proposition 3.3 $\varphi(0) \to \forall n(\varphi(n) \to \varphi(n + 1)) \to \forall n, m(\varphi(m) \to \varphi(n + m))$

Proof. Assume $b : \varphi(0)$ and $f : \forall n(\varphi(n) \to \varphi(n + 1))$. By induction on n:

$n = 0$ We have to prove

$$\forall m(\varphi(m) \to \varphi(m))$$

this is trivially proved by $\lambda\, m, u(u)$.

$n + 1$ Let us assume n, the recursive call $p : \forall m(\varphi(m) \to \varphi(n + m))$, m and the accumulator $y : \varphi(m)$. We have to prove

$$\varphi(n + m + 1)$$

Apply f to m and y obtaining $(f\, m\, y) : \varphi(m + 1)$. Now apply p to $(m + 1)$ and $(f\, m\, y)$.

The accumulator-based program transformation provides us with a new proof of the induction principle over natural numbers:

Corollary 3.4 $\varphi(0) \to \forall n(\varphi(n) \to \varphi(n + 1)) \to \forall n\varphi(n)$.

Proof. Assume $b : \varphi(0)$, $f : \forall n(\varphi(n) \to \varphi(n + 1))$ and n. To prove $\varphi(n)$: instantiate the formula proved in Proposition 3.3 on n, 0 and $b : \varphi(0)$

We are done: the corresponding extracted program, named Ind_ACC, is presented in the Extracted program 2.

Extracted program 2 Ind_ACC

```
[b,f,n](Rec nat => nat => sigma => sigma
        [m,y]y
        [n,p,m,y] p (m+1) (f m y) n 0 b)
```

4 From Higher Order to First Order Computation

In this section, we answer positively to the question of the existence of some formal connection between Ind_CONT and Ind_ACC. The link between the two of them relies on *Defunctionalization*. This program transformation, first introduced by Reynolds in the early 1970's [17] and later on extensively studied by Danvy [8], is a whole program transformation to turn higher-order into first-order functional programs, that is to transform programs where functions may be anonymous, given as arguments to other functions and returned as results, into programs where none of the functions involved accept arguments or produce results that are functions. Let us consider the following simple example taken from [8]:

```
(* aux : (nat -> nat) -> nat *)
let aux f = (f 1) + (f 10)

(* main : nat * nat * bool -> nat *)
let main x y b = aux (fun z -> x + z) *
                 aux (fun z -> if b then y + z else y * z)
```

The above function `aux` calls the higher order function f twice: on 1 and 10 and returns the sum as its result. Also, the `main` function calls `aux` twice and returns the product of these calls. There are only two function abstractions and they occur in `main`.

Defunctionalizing this program amounts to defining a data type with two constructors, one for each function abstraction, and its associated apply function. The first function abstraction contains one free variable (`x`, of type `nat`), and therefore the first data-type constructor requires a natural. The second function abstraction contains two free variables (`y`, of type `nat`, and `b` of type `bool`), and therefore the second data-type constructor requires an integer and a boolean.

In main, the first abstraction is thus introduced with the first constructor and the value of `x`, and the second abstraction with the second constructor and the values of `y` and `b`.

To the functional argument used in `aux`, corresponds a pattern matching done by the following `apply` function:

```
type lam = LAM1 of nat | LAM2 of nat * bool

(* apply : lam * nat -> nat *)
let apply l z =
```

```
  match l with
  | LAM1 x -> x + z
  | LAM2 y b -> if b then y + z else y - z

(* aux_def : lam -> nat *)
let aux_def f = apply f 1 + apply f 10

(* main_def : nat * int * bool -> nat *)
let main_def x y b = aux_def (LAM1 x) * aux_def (LAM2 y b)
```

Now let us apply defunctionalization to Ind_CONT. We introduce the algebra path_nat (below) to represent the *initial* continuation $\lambda x\, x$ and the *intermediate* continuation $\lambda u(k(f\, n\, u))$.

```
type path_nat = TOP | UP of path_nat * nat
```

Each constructor has as much parameters as free variables occurring in the corresponding continuation function. Finally the call `(k b)` in Ind_CONT is replaced by the *apply* function (here is anonymous) that dispatches over the path_nat constructors. We named the defunctionalization of Ind_CONT Insd_Def_CONT (Extracted program 3).

Extracted program 3 Ind_Def_CONT

```
[n](Rec nat => path_nat => sigma
     [q] (Rec path_nat => sigma => sigma
          [y] y
          [m,q',p,y] p (f m y)) q b
     [n,p,q] p (UP q n)) n TOP
```

Now the question is: from which proof is it possible to extract Ind_Def_CONT? Given q of type `path_nat` and y "of type" $\varphi(n)$ the inner procedure would be expected to return an element of type $\varphi(n)$ when $q =$ `TOP` and an element of type $\varphi(n+m+1)$ when $q =$ `(UP (...(UP TOP` $n+m$`)...) n)`. But q does not depend explicitly on n, so given y and p alone one cannot guess anything about the type of the returned value. In order to state this link between the above two inputs we need to quantify *non computationally* over an additional parameter as showed in the theorem below. In order to do that, let us before introduce the following notation.

Definition 4.1 *Given p and q of type `path_nat` the "degree" of q with respect to p is defined by the following partial function:*

$$\sharp_p(q) = \begin{cases} \sharp_p(p) = 0 \\ \sharp_p(\mathit{TOP}) = \mathit{Undef} & \mathit{if} p \neq \mathit{TOP} \\ \sharp_p(\,(\mathit{UP}\, q\, n)) = 1 + \sharp(q) \end{cases}$$

Definition 4.2 *Given x and p of type* `path_nat` *and a natural n, we say that x has a "good shape" with respect to p at level n when*

$$\mathsf{GoodShape}(x,p,n) \Longleftrightarrow \begin{cases} p = x \\ p \neq x = (\mathtt{UP}\, q\, l) \wedge (l = n) \wedge \mathsf{GoodShape}(q,p,n+1) \end{cases}$$

In the following we adopt the following notation: by $\mathcal{C}[t]$ we indicate a `path_nat` term that contain an occurrence of the term t. So for example if $\mathcal{C}[t]$=`(UP(UP TOP` j `)`i`)`, for some naturals i and j, then t it could be `TOP`, `(UP TOP` j `)` or $\mathcal{C}[t]$ it self.

Theorem 4.1 $\varphi(0) \to \forall n(\varphi(n) \to \varphi(n{+}1)) \to \forall x \forall^{\mathsf{nc}} n(\mathsf{GoodShape}(x, \mathtt{TOP}, n) \to \varphi(n) \to \varphi(n + \sharp_{\mathtt{TOP}}(x)))$

Proof. By induction over x.

$x = \mathtt{TOP}$ Assume n, $u : \varphi(n)$ and $\mathsf{GoodShape}(\mathtt{TOP}, \mathtt{TOP}, n)$. The thesis follows by u.

$x = (\mathtt{UP}\, q\, l)$ Assume $\mathsf{p} : \forall^{\mathsf{nc}} n(\mathsf{GoodShape}(q, \mathtt{TOP}, n) \to \varphi(n) \to \varphi(n + \sharp(q)))$, n, $\mathsf{gs} : \mathsf{GoodShape}((\mathtt{UP}\, q\, l), \mathtt{TOP}, n)$ and $y : \varphi(n)$. By gs and definition 4.1 follows $l = n$ and $\mathsf{gs}' : \mathsf{GoodShape}(q, \mathtt{TOP}, n+1)$. Instantiate f on l and $\varphi(n)$ (l is equal to n) obtaining $(f\, l\, y) : \varphi(n+1)$. To prove the thesis, it remains to instantiate p on $n+1$, gs' and $(f\, l\, y)$.

The program extracted from theorem 4.1 is Ind_Def_CONT (see the Extracted program 3). But we are not done yet: the theorem below shows as Ind_Def_CONT needs some additional simplification. In the following lines we will favor the presentation $\lambda n(\mathcal{P}\, n\, \mathsf{TOP})$ in place of Ind_Def_CONT.

Proposition 4.1 *For all $n, p^{\texttt{path\_nat}}, ACC^{\texttt{path\_nat}}$, if*

$$\lambda n((\mathcal{P}\, n\, p)\, (n+1)) =_{\mathcal{R}\eta\beta} \mathcal{P}\, 0\, ACC$$

then $\mathsf{GoodShape}(ACC, p, 0)$ *and* $\sharp_p ACC = n+1$.

Proof. By induction on n.

As a corollary of theorem 4.1, we have that, for $p = \mathtt{TOP}$ the expression $\lambda n(\mathcal{P}\, n\, \mathtt{TOP})(n+1)$, that is $\mathsf{Ind\_Def\_CONT}(n+1)$, rewrites to $(\mathcal{P}\, 0\, \mathsf{ACC})$ with $\mathsf{GoodShape}(\mathsf{ACC}, \mathtt{TOP}, 0)$ and $\sharp_{\mathtt{TOP}}(\mathsf{ACC}) = n+1$. A data structure like `path_nat` is too complex to store this particular simple data. So we replace `path_nat` by $\mathbf{N}$ in Ind_Def_CPS according to the informal correspondence:

$$\begin{aligned} \mathsf{TOP} &\leftrightsquigarrow 0 \\ (\mathtt{UP}\, \mathsf{TOP}\, n) &\leftrightsquigarrow 1 \\ \vdots \quad & \quad \vdots \\ (\mathtt{UP}(\ldots(\mathtt{UP}\, \mathsf{TOP}\, n)\ldots)0) &\leftrightsquigarrow n+1 \end{aligned}$$

Extracted program 4 Ind_Intermediate_ACC

```
[n] (Rec nat => nat => sigma
     [q] (Rec nat => nat => sigma => sigma
          [m,y] y
          [q',p,m,y] p (m+1) (f m y)) q 0 b
     [n,p,q] p (q+1)) n 0
```

obtaining the code Ind_Intermediate_ACC (Extracted program 4). This procedure still performs some redundant computations: the outer recursion runs over n, so the accumulator parameter q ranges from 0 to n. At this point the inner routine (that will return the final result) is called on q, now equal to n. This is equivalent to calling directly the subroutine over n, which corresponds to Ind_ACC as expected.

5 Case Study

Let us consider now a more elaborated example taken from Bioinformatics. This is an area where the *correctness* and the *efficiency* of programs plays a crucial role: *efficiency* because DNA sequences are really huge and getting a lower complexity class is essential, *correctness* because we need to trust programs and we cannot check their results by hand. An important line of research is "Sequence Analysis", which is concerned with locating biologically meaningful segments in DNA sequences. In this context, we will treat the so-called "Maximal Scoring Subsequence" (MSS) Problem. For a sequence of real numbers, we are looking for a contiguous sub-sequence such that the sum of its elements is maximal over all sub-sequences. Several authors have investigated that problem or a variation thereof, see, e.g., [9, 6, 11, 15, 22]

5.1 The MSS Problem

The MSS problem, in its most general presentation, can be explained as follows:

MSS Problem : Given a list l of real numbers, find an interval (i, k) (with $i \leq k \leq |l| - 1$) such that

$$\sum_{j=i'}^{k'} l\,[j] \leq \sum_{j=i}^{k} l\,[j]$$

for every (i', k') (with $i' \leq k' \leq |l| - 1$). The problem doesn't admit solutions for all the inputs, in fact on the empty list there is no solution.

Here we report on a variant of the MSS problem first proposed in [2, 19].

MSS Problem Instance :Given the function $seg : \mathbb{N} \times \mathbb{N} \to \mathbb{X}$ defined on $[0, \ldots, n] \times [0, \ldots, n]$, find an interval (i, k),(with $i \leq k \leq n$) such that

$$seg\,[i', j'] \leq_{\mathbb{X}} seg\,[i, j]$$

for every (i', k'), (with $i' \leq k' \leq n$). This time the problem admits a solution on each natural input n. Here $\mathbb{X}$ is a set on which we can define a total order relation $\leq_{\mathbb{X}}$. Moreover we require seg to have the following property:

$$\mathsf{AX} = \forall n, i, j(seg[i, n] \leq_{\mathbb{X}} seg[j, n] \to seg[i, (\mathsf{Succ}\ n)] \leq_{\mathbb{X}} seg[j, (\mathsf{Succ}\ n)])$$

Theorem 5.1 *For all n*

$$\exists i, k((i \leq k \leq n) \wedge \forall i', k'(i' \leq k' \leq n) \to seg[i', k'] \leq_{\mathbb{X}} seg[i, k]) \tag{2}$$

$$\exists j((j \leq n) \wedge \forall j'(j' \leq n) \to seg[j', n] \leq_{\mathbb{X}} seg[j, n]) \tag{3}$$

Proof. By induction on n.

$\underline{n = 0}$ We set $i = k = j = 0$.

$\underline{n + 1}$ Assume (2) and (3) hold for n (hypothesis IH^1_n,IH^2_n). Let (i_n, k_n) and j_n be the segment and the value that satisfy IH^1_n and IH^2_n respectively (see picture in Figure 1) By IH^2_n, for an arbitrary $j' \leq n$

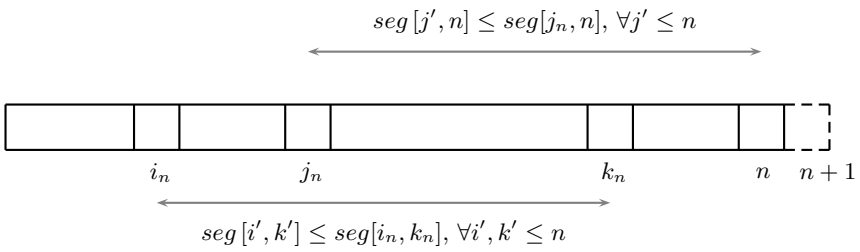


Fig. 1. The witnesses i_n,j_n and k_n at step n of the induction

$$seg[j', n] \leq_{\mathbb{X}} seg[j_n, n] \tag{4}$$

Instantiating Ax on n,j',j_n and (4),

$$seg[j', n + 1] \leq_{\mathbb{X}} seg[j_n, n + 1]$$

The witness for IH^2_{n+1} is given by:

$$j_{n+1} = \begin{cases} j_n & seg[n + 1, n + 1] \leq_{\mathbb{X}} seg[j_n, n + 1] \\ (n + 1) & seg[n + 1, n + 1] \not\leq_{\mathbb{X}} seg[j_n, n + 1] \end{cases}$$

We have to prove that j_{n+1} satisfies,

$$\forall j'.(j' \leq (n+1)) \rightarrow seg[j', (n+1)] \leq_{\mathbb{X}} seg[j_{n+1}, (n+1)])$$

This has to be proved both for $j' \leq n$ and $j' = (n+1)$. Both cases follow straightforwardly from IH_n^2 and the construction of j_{n+1}. The new maximal segment, is given by:

$$(i_{n+1}, j_{n+1}) = \begin{cases} (i_n, k_n) & seg[j_{n+1}, n+1] \leq_{\mathbb{X}} seg[i_n, k_n] \\ (j_{n+1}, n+1) & seg[j_{n+1}, n+1] \not\leq_{\mathbb{X}} seg[i_n, k_n] \end{cases}$$

Again, we have to prove that (i_{n+1}, k_{n+1}) satisfies,

$$\forall i', k'(i' \leq k' \leq (n+1)) \rightarrow seg[i', k'] \leq_{\mathbb{X}} seg[i_{n+1}, k_{n+1}]$$

This property has to be proved both for $(i' \leq k' \leq n)$ and $(i' \leq k' = n+1)$. Both cases follows from IH_n^1, IH_n^2, and the construction of (i_{n+1}, k_{n+1})

The program extracted from the previous proof is MSS (Extracted program 5). The above algorithm makes use of the expression (`LET r IN s`). This is actually

Extracted program 5 MSS

```
[seg] (Rec nat => sigma
       (0,0,0)
       [n,(i,j,k)]
         LET m = if (seg[n+1, n+1] <= seg[j, n+1]) j (n+1)
         IN if (seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))
```

syntactic sugar: although it does not belong to our term language, MINLOG allows the user to make use of it. This is irrelevant in the context of the paper, and the reader is referred to [5] for a more detailed development on that issue.

By the following extension of the definition 3.1:

$3'$. *if* $t \equiv (\mathsf{LET}\ r\ \mathsf{IN}\ s)$ *then* s *is a tail expression.*

and accordingly to definition 3.3, the program MSS is not *tail* recursive.

5.2 Generation of a Continuation/Accumulator based MSS-Program

We apply the transformations proposed in section 3.1 and 3.2 to the proof of the theorem 5.1 in order to extract respectively a continuation and an accumulator based version of the MSS program. We named the extracted code of these two transformations respectively MSS_CONT (Extracted program 6) and MSS_ACC (Extracted program 7). Both MSS_ACC and MSS_ACC are tail recursive, as the result of automatic transformation from the proof of the theorem 5.1. This way, we have ensured they are still correct implementations of the abstract algorithm while being more efficient at the same time.

Extracted program 6 MSS_CONT

```
([seg,n]
  (Rec nat => (sigma => sigma) => sigma
   [k] k (0,0,0)
   [n,p,k] p ([(i,j,k)]
               LET m = if (seg[n+1, n+1] <= seg[j, n+1]) j (n+1)
               IN if (seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))))
 n [x]x
```

Extracted program 7 MSS_ACC

```
([seg,n]
  (Rec nat => nat => sigma => sigma
   [m,y] y
   [n,p,m,(i,j,k)]
         p (m+1) LET m = if (seg[n+1, n+1] <= seg[j, n+1]) j (n+1)
                 IN if (seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))))
 n 0 (0,0,0)
```

6 Conclusions and future work

The expression Π introduced in section 3.2 represents a way to mimic a *let* expression just by a pure lambda calculus expression (original goal of Bauer in designing it). The terms Λ and Π compute, applied to appropriate input parameters, the same function $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$. But we note that Λ is in some way more *general* than Π. The modification of Λ in order to make it work on lists (let us name it $\Lambda_{\mathbf{L}(\rho)}$) instead of naturals is easy; more importantly, the proof from which $\Lambda_{\mathbf{L}(\rho)}$ can be extracted is obtained by a slight modification of the proof from which Λ is extracted. In the case of lists the end formula to prove should be: $\forall l^{L(\rho)}(P(l) \to \bot) \to \bot$. Unfortunately we can not extend in the same way Π and its proof: Π looks intrinsically dependent on the algebra of natural numbers.

A final remarks on the formal transformation of Ind_CONT into Ind_ACC presented in section 4. It could be interesting to study if, and how, to perform the inverse operation, that is to go from Ind_ACC to Ind_CONT. We argue that it could be done by the *Refunctionalization* technique [7], but this aspect needs a deeper investigation.

References

1. Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1993.
2. Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

3. Andrej Bauer. http://math.andrej.com/2005/09/16/proof-hacking/.
4. Luca Chiarabini. Extraction of Efficient Programs from Proofs: The case of Structural Induction over Natural Numbers. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, 2008.
5. Luca Chiarabini. A new adaptation of the pruning technique for the extraction of efficient program from proofs, 2008.
6. K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34:373–387, 2004.
7. Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science Of Computer Programming*, 2008.
8. Olivier Danvy and Lasse R.Nielsen. Defunctionalization at work. In editor Harald Søndergaard, editor, *Proceedings of the Third International Conference of Principles and Practice of Declarative Programming*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.
9. P. Fariselli, M. Finelli, D. Marchignoli, P.L. Martelli, I. Rossi, and R. Casadio. Maxsubseq: An algorithm for segment-length optimization. the case study of the transmembrane spanning segments. *Bioinformatics*, 19:500–505, 2003.
10. Matthias Felleisen and Amr Sabry. Continuations in programming practice: Introduction and survey. Draft of August 26, 1999.
11. M.H. Goldwasser, M.-Y. Kao, and H.-I. Lu. Linear-time algorithms for computing maximum-density sequence. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.
12. R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August, 1998.
13. K. Kent Dybvig. *The Scheme Programming Language*. Mit Press, 1996.
14. G. Kreisel. Interpretation of Analysis by means of Functionals of Finite Type. In A. Heyting, editor, *Constructivity in Mathematics*, 1959.
15. Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis. *Journal of Computer and System Sciences*, 65:570–586, 2002.
16. Frank Pfenning. Program development through proof transformation. In *Contemporary Mathematics*, volume 106, pages 251–262, 1990.
17. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).
18. Helmut Schwichtenberg. http://www.minlog-system.de/.
19. Helmut Schwichtenberg. Programmentwicklung durch beweistransformation: Das maximalsegmentproblem. In *Bayer. Akad.*, 1996.
20. Helmut Schwichtenberg. Minimal Logic for Computable Functionals. http://www.mathematik.uni-muenchen.de/~chiarabi/mlcf.pdf, December 2008.
21. M.H. Sørensen and P.Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.
22. M. Tompa W.L. Ruzzo. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology, ISMB'99*, pages 234–241, 1999.

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 naturally 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 [BvW98, Mor94]. 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 [Dij76], 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. In [KM95], King and Morgan proposed a solution to this problem and developed a refinement calculus that adds the **exit** command and the exception block construct **try** S **catch** T[1] to the language. Their solution was to specify each predicate transformer by a pair of post-conditions $\langle \varphi_n, \varphi_e \rangle$ as the input, rather than by a single post-condition: They write $wp\ S, \varphi_n, \varphi_e$ for the weakest pre-condition that guarantees the program S either 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 by:

$$wp\ \textbf{exit}, \varphi_n, \varphi_e \quad \varphi_e, \qquad wp\ \textbf{try}\ S\ \textbf{catch}\ T, \varphi_n, \varphi_e \quad wp\ S, \varphi_n, wp\ T, \varphi_n, \varphi_e \quad .$$

The intuition behind these specifications are explained as follows. The **exit** command immediately causes an exceptional termination. Thus, in order for the **exit** command to exceptionally terminate establishing φ_e, the weakest pre-condition must be φ_e. The exception block **try** S **catch** T executes S and terminates normally, if no exception is

[1] This exception block construct is our own extension of the one proposed in [KM95].

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 $wp\ T, \varphi_n, \varphi_e$, guaranteeing that the execution of T is to terminate in a condition as required by the pair $\langle \varphi_n, \varphi_e \rangle$.

Another problem in refining exceptions is that they are not only raised explicitly by the command **exit** 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*, i.e., predicates whose truth value may be undefined in some states. 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.

In this paper, we propose a refinement calculus for refining programs that may raise and catch exceptions, where exceptions can be raised explicitly by the **exit** command and implicitly by the evaluation of partial predicates. For this, we develop our theory of program refinement in a predicate transformer semantics based on Arieli and Avron's four-valued logic [AA96, AA98].

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 $\varphi\ \sigma \quad \langle \varphi_n\ \sigma\ , \varphi_e\ \sigma\ \rangle$ 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 [Bel77], which has been studied by Ginsberg in the generalized setting of bilattices [Gin88] and was further examined by Fitting [Fit94]. Arieli and Avron [AA96, AA98] 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 limited to **f**, **t**, and $\bot$. In later sections we exploit the properties of partial predicates in this three-valued sublogic.

Related work. Formal treatment of exceptions and partial predicates has been studied rather separately. The exception mechanism was formulated in the refinement calculus by King and Morgan [KM95] and further examined in [Wat02], but partial predicates are out of their concern. (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 [Jon86, JM94]; Bono et al. [BK06] 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 [Owe93, JM94, MB99]. 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 [Häh05] 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 [HJ00] 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

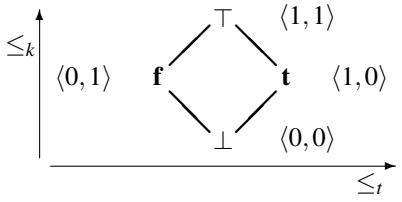


Fig. 1. The bilattice of four truth values

Let *TWO* be the lattice of classical truth values 0 and 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]

$$\langle x_1,y_1\rangle \leq_t \langle x_2,y_2\rangle \text{ iff } x_1 \leq x_2 \text{ and } y_2 \leq y_1,$$
$$\langle x_1,y_1\rangle \leq_k \langle x_2,y_2\rangle \text{ iff } x_1 \leq x_2 \text{ and } y_1 \leq y_2.$$

The $\leq_t$ order ($\leq_k$ order, resp.) induces the meet $\wedge$ and join $\vee$ operators (meet $\otimes$ and join $\oplus$ operators, resp.) The definitions are given below, where $\sqcap$ and $\sqcup$ stand for the meet and join in *TWO*, respectively.

$$\langle x_1,y_1\rangle \wedge \langle x_2,y_2\rangle \quad \langle x_1 \sqcap x_2, y_1 \sqcup y_2\rangle, \qquad \langle x_1,y_1\rangle \vee \langle x_2,y_2\rangle \quad \langle x_1 \sqcup x_2, y_1 \sqcap y_2\rangle,$$
$$\langle x_1,y_1\rangle \otimes \langle x_2,y_2\rangle \quad \langle x_1 \sqcap x_2, y_1 \sqcap y_2\rangle, \qquad \langle x_1,y_1\rangle \oplus \langle x_2,y_2\rangle \quad \langle x_1 \sqcup x_2, y_1 \sqcup y_2\rangle.$$

In addition, negation $\neg$ is defined by $\neg\langle x,y\rangle \quad \langle y,x\rangle$ 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., $\neg\ x \vee y \quad \neg x \wedge \neg y$ and $\neg\ x \wedge y \quad \neg x \vee \neg y$, while $\oplus$ and $\otimes$ are De Morgan self-dual, i.e., $\neg\ x \oplus y \quad \neg x \oplus \neg y$ and $\neg\ x \otimes y \quad \neg x \otimes \neg y$. The four values are related with each other by means of $\vee$, $\wedge$, $\oplus$, and $\otimes$, e.g., $\bot \vee \mathbf{f} \quad \bot$, $\mathbf{t} \oplus \mathbf{f} \quad \top$, $x \vee \bot \quad x \otimes \mathbf{t}$.

The bilattice *FOUR* is *distributive*, i.e., the four lattice operations $\wedge$, $\vee$, $\otimes$, and $\oplus$ distribute over each other, e.g., $x \oplus \ y \wedge z \quad x \oplus y \ \wedge \ x \oplus z$. 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 [AA96]. With $\mathcal{D} \quad \{\mathbf{t}, \top\}$ 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, since 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 [AA96].) We assume the set Value of program values (e.g., integers) 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, $p \leq_t q$ ($p \leq_k q$, resp.) holds iff $p\ \sigma\ \leq_t q\ \sigma$ ($p\ \sigma\ \leq_k q\ \sigma$, resp.), 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 (existential quantification, resp.) over i of predicate $p\ i$ by $\forall i.p\ i \triangleq \bigwedge_i p\ i$ ($\exists i.p\ i \triangleq \bigvee_i p\ i$, resp.)

The above mentioned structure of logical bilattice induces a four-valued predicate logic [AA96], 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 $p_1, \cdots, p_n \mid q_1, \cdots, q_m$, 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 $\mid p$ holds (in other words, $p\ \sigma \in \mathcal{D}$ 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, neither $\vdash p \vee \neg p$ nor $p \wedge \neg p \vdash q$ hold. The connectives $\supset$, $\rightarrow$, and $\leftrightarrow$ are a logical implication or an equivalence in the following sense: $\mid p \supset q$ iff $p \mid q$; $\mid p \rightarrow q$ iff $p \leq_t q$; $\mid p \leftrightarrow q$ iff $p = q$. Furthermore the logical equivalence $\leftrightarrow$ is a congruence: $\mid p \leftrightarrow q$ implies $\mid \Theta\ p \leftrightarrow \Theta\ q$ for any formula scheme Θ. For further details of the proof system and logical properties of the four-valued logic, see [AA96, AA98].

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 $p\ X\backslash v$ ($p\ X\backslash e$, resp.) for the predicate q such that $q\ \sigma = p\ \sigma\ X\backslash v$ ($q\ \sigma = p\ \sigma\ X\backslash e$, resp.) 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 $e\ \sigma \quad \sigma\ X$. More generally, we may confuse numerical expressions and predicates with their pointwise extensions. For example, when we write $X \quad 1 \geq Y$, it denotes a predicate q such that $q\ \sigma \quad (\sigma\ X \quad 1 > \sigma\ Y\ )$, where $\quad$ is the binary integer addition and $\geq$ is the binary predicate such that $v \geq v' \quad \mathbf{t}$ if v is greater than or equal to v' but $v \geq v' \quad \mathbf{f}$ 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 $\mathbf{t} \oplus p \quad \mathbf{t} \oplus p'$ 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:*

$$S \sqsubseteq T \quad \textit{iff} \quad S\ \varphi\ \leq_k T\ \varphi \quad \textit{for any } \varphi,$$

where the join $\oplus$ and meet $\otimes$ operators are a pointwise extension of the corresponding logical connectives:

$$S \oplus T \quad \varphi \quad S\ \varphi\ \oplus T\ \varphi \quad \textit{and} \quad S \otimes T \quad \varphi \quad S\ \varphi\ \otimes T\ \varphi\ .$$

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 $S;T \quad \varphi \quad S\ T\ \varphi$ 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 $S\ \varphi\ \leq_k T\ \varphi$ for every φ. There are several different ways to verify this.

Proposition 3.1. *For any $S, T \in$ PTran and any four-valued predicate φ, $S\ \varphi\ \leq_k T\ \varphi$ iff $S\ \varphi\ \leq_t T\ \varphi$ iff $\mid \ S\ \varphi \rightarrow T\ \varphi$ iff $S\ \varphi \mid \ T\ \varphi$ 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 $S\ \varphi \vdash T\ \varphi$. Further discussions on these alternative ways for validating refinement laws are found in the full paper [Nis].

$$
\begin{array}{rcll}
\mathbf{skip}\ \varphi & \triangleq & \varphi & \text{(skip)} \\
(X :=\ e)\ \varphi & \triangleq & \mathbf{f} \oplus \varphi\, X \backslash e\ \otimes\ \mathbf{t} \oplus \varphi & \text{(assignment)} \\
\mathbf{abort}\ \varphi & \triangleq & \mathbf{f} \otimes \varphi & \text{(non-termination)} \\
\mathbf{magic}\ \varphi & \triangleq & \mathbf{t} \oplus \varphi & \text{(miracle)} \\
\mathbf{exit}\ \varphi & \triangleq & \mathbf{t} \oplus \varphi\ \otimes \neg\ \mathbf{t} \oplus \varphi & \text{(exit)} \\
\mathbf{try}\ S\ \mathbf{catch}\ T & \triangleq & \big(\mathbf{f} \oplus S(\ \mathbf{f} \oplus \varphi\ \otimes \neg\ \mathbf{f} \oplus T\ \varphi\ )\big) \otimes\ \mathbf{t} \oplus \varphi & \text{(exception handling)} \\
\{p\}\ \varphi & \triangleq & \neg\ p \supset \top\ \otimes \varphi & \text{(assertion)} \\
[p]\ \varphi & \triangleq & p \supset \bot\ \oplus \varphi & \text{(assumption)} \\
\langle p \rangle\ \varphi & \triangleq & p \supset \bot\ \oplus \varphi\ \otimes \neg\ p \supset \top\ \oplus \varphi & \text{(conditional exit)}
\end{array}
$$

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 the two classical predicates in the pair separately. For example, given four-valued predicates p and q, a four-valued predicate that encodes $\langle p_n, q_e \rangle$ can be expressed by the formula $\mathbf{f} \oplus p\ \otimes\ \mathbf{t} \oplus q$. This is verified by a simple calculation:

$$
\mathbf{f} \oplus p\ \otimes\ \mathbf{t} \oplus q \qquad \langle 0,1 \rangle \oplus \langle p_n, p_e \rangle\ \otimes\ \langle 1,0 \rangle \oplus \langle q_n, q_e \rangle \qquad \langle p_n, 1 \rangle \otimes \langle 1, q_e \rangle \qquad \langle p_n, q_e \rangle .
$$

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., $\mathbf{skip}; S \quad S; \mathbf{skip} \quad S$.
- $X :=\ e$ 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**[3] 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).

[2] There are different ways of expressing the same operation, e.g., $\mathbf{t} \otimes p\ \oplus\ \mathbf{f} \otimes q$; Similarly for other formulas.

[3] 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, $\mathbf{abort};S \quad \mathbf{abort}$ and $\mathbf{magic};S \quad \mathbf{magic}$.

- **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., $\mathbf{exit};S \quad \mathbf{exit}$.
- **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 formula 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.[4]

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:

$$\mathbf{if}\ p\ \mathbf{then}\ S\ \mathbf{else}\ T \triangleq \langle p \vee \neg p \rangle; \quad [p];S \ \otimes \ [p \supset \bot];T \quad .$$

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 run 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: 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 native conjunction operator $\wedge$ and disjunction operator $\vee$, 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 $\bot$.)

Though the strong Kleene connectives have no corresponding operations in real programming languages, but we can define logical operators that are found in practical

[4] 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.

programming languages in the three-valued sublogic as follows. Let us write, following [Fit94], $p : q$ 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 $\vec{\vee}$ and conjunction $\vec{\wedge}$ for any pair of consistent predicates p and q, as follows.

$$p \vec{\wedge} q \triangleq p \wedge p : q \qquad\qquad p \vec{\vee} q \triangleq p \vee \neg p : q$$

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.

$$p \wedge^w q \triangleq p \vec{\wedge} q \otimes q \vec{\wedge} p \qquad\qquad p \vee^w q \triangleq p \vec{\vee} q \otimes q \vec{\vee} p$$

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 $\vec{\wedge}$ and $\vec{\vee}$, 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., $S_1 \otimes S_2 \,;T \quad S_1;T \otimes S_2;T$. (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** $T \quad$ **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 in PTran as below.

$$\{p\} \sqsubseteq \mathbf{skip} \sqsubseteq [p] \qquad (4.1) \qquad\qquad \{p\} \sqsubseteq \langle p \rangle \sqsubseteq [p] \qquad (4.2)$$

Further, the assertion (the assumption, resp.) is monotonic (anti-monotonic, resp.) with respect to the $\leq_t$ order over guard predicates. That is, if $p \rightarrow q$ is valid (or equivalently, $p \mid \quad q$), we have:

$$\{p\} \sqsubseteq \{q\} \qquad (4.3) \qquad\qquad [q] \sqsubseteq [p] \qquad (4.4)$$

In contrast, the conditional exit has no such particular (anti-)monotonicity property.

Provided that $p \rightarrow q$ is valid, we have:

$$\{p\} \quad \{p\};\{q\} \quad \{p\};[q] \quad \{p\};\langle q \rangle \qquad (4.5)$$
$$[p] \quad [p];\{q\} \quad [p];[q] \quad [p];\langle q \rangle \qquad (4.6)$$
$$\langle p \rangle \quad \langle p \rangle;\{q\} \quad \langle p \rangle;[q] \quad \langle p \rangle;\langle q \rangle \qquad (4.7)$$

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 $\wedge$, $\vec{\wedge}$, or $\wedge^w$.

$$\{p\};\{q\} \quad \{q\};\{p\} \quad \{p \wedge q\} \quad \{p \vec{\wedge} q\} \quad \{p \wedge^w q\} \qquad (4.8)$$
$$[p];[q] \quad [q];[p] \quad [p \wedge q] \quad [p \vec{\wedge} q] \quad [p \wedge^w q] \qquad (4.9)$$
$$\langle p \rangle;\langle q \rangle \quad \langle q \rangle;\langle p \rangle \quad \langle p \wedge q \rangle \quad \langle p \vec{\wedge} q \rangle \quad \langle p \wedge^w q \rangle \qquad (4.10)$$

Combining the laws (4.5) through (4.10), we can propagate a copy of a statement of conditional control past one or more successive control statements (of possibly different kinds), e.g., $[p];\{q\};\langle r \rangle \quad [p];\{q\};\langle r \rangle;[p]$.

The disjunction in the guard of an assertion or an assumption can be substituted with an appropriate non-deterministic choice.

$$\{p \vee q\} \quad \{p\} \oplus \{q\} \qquad (4.11) \qquad\qquad [p \vee q] \quad [p] \otimes [q] \qquad (4.12)$$

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.

$$[p] \otimes [p \supset \bot] \quad \mathbf{skip} \qquad (4.13) \qquad\qquad \langle p \rangle;\langle p \supset \bot \rangle \quad \mathbf{exit} \qquad (4.15)$$
$$[p];[p \supset \bot] \quad \mathbf{magic} \qquad (4.14)$$

Some interesting laws hold for the conditional exit of the form $\langle p \vee \neg p \rangle$, when p is a formula of a sequential or a weak Kleene connective.

$$\langle\ p \vec{\wedge} q\ \vee \neg\ p \vec{\wedge} q\ \rangle \quad \langle p \vee \neg p \rangle;\langle \neg p \vee q \vee \neg q \rangle \qquad (4.16)$$
$$\langle\ p \vec{\vee} q\ \vee \neg\ p \vec{\vee} q\ \rangle \quad \langle p \vee \neg p \rangle;\langle p \vee q \vee \neg q \rangle \qquad (4.17)$$
$$\langle\ p \wedge^w q\ \vee \neg\ p \wedge^w q\ \rangle \quad \langle\ p \vee^w q\ \vee \neg\ p \vee^w q\ \rangle \quad \langle p \vee \neg p \rangle;\langle q \vee \neg q \rangle \qquad (4.18)$$

When the predicate p is classical, we have:

$$\{p \supset \bot\} \quad \{\neg p\} \quad (4.19) \qquad [p \supset \bot] \quad [\neg p] \quad (4.20) \qquad \langle p \vee \neg p \rangle \quad \textbf{skip} \quad (4.21)$$

4.2 Refinement of exceptions

The following refinement laws hold for exception statements.

$$\textbf{exit}; S \quad \textbf{exit} \quad (4.22) \qquad \textbf{try } S; \langle p \rangle \textbf{ catch skip} \quad \textbf{try } S \textbf{ catch skip} \quad (4.23)$$

Some useful laws hold for a subclass of PTran, called *non-exceptional* predicate transformers. A transformer S is called non-exceptional, if it has no chance of exceptional termination, that is, $\psi_n \quad \psi'_n$ holds whenever $\langle \psi_n, \varphi_e \rangle \quad S \ \langle \varphi_n, \varphi_e \rangle$ 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 $\mathbf{f} \oplus \varphi \quad \mathbf{f} \oplus \varphi'$.*

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.

$$\textbf{try } S \textbf{ catch } T \quad S \quad (4.24) \qquad \textbf{try } S; \textbf{exit catch } T \quad S; T \quad (4.25)$$

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** $p \vec{\wedge} q$ **then** S **else** T contains several implicit information for control: The predicate $p \vec{\wedge} q$ evaluates from left to right; As soon as p evaluates to $\mathbf{f}$, the **else** clause is selected; As soon as p evaluates to $\bot$, an exception is raised; If p evaluates to $\mathbf{t}$, q is examined.

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.

$$[p \vec{\wedge} q \supset \bot] \quad [p \wedge q \supset \bot] \quad [p \supset \bot] \otimes [q \supset \bot]. \qquad (5.1)$$

$$[p]; \langle \neg p \vee q \rangle \quad [p]; \langle q \rangle \qquad \text{if } p \text{ is consistent} \qquad (5.2)$$

$$\langle p \vee \neg p \rangle; [p \supset \bot] \quad \langle p \vee \neg p \rangle; [p \supset \bot]; \langle \neg p \vee q \vee \neg q \rangle \qquad \text{if } p \text{ is consistent} \qquad (5.3)$$

Then we can carry out the following derivation, provided p is consistent.

$$\begin{array}{l}
\textbf{if } p \vec{\wedge} q \textbf{ then } S \textbf{ else } T \quad \langle\ p \vec{\wedge} q\ \vee \neg\ p \vec{\wedge} q\ \rangle; \big([p \vec{\wedge} q]; S \otimes [p \vec{\wedge} q \supset \bot]; T\big) \\
\langle p \vee \neg p\rangle; \langle \neg p \vee q \vee \neg q\rangle; \big([p]; [q]; S \otimes [p \supset \bot]; T \otimes [q \supset \bot]; T\ \big) \\
\quad \text{— by (4.16), (4.9), (5.1), and distributivity} \\
\langle p \vee \neg p\rangle; [p]; \langle \neg p \vee q \vee \neg q\rangle; \big([p]; [q]; S \otimes [p \supset \bot]; T \otimes [q \supset \bot]; T\big) \\
\quad \otimes \langle p \vee \neg p\rangle; [p \supset \bot]; \langle \neg p \vee q \vee \neg q\rangle; \big([p]; [q]; S \otimes [p \supset \bot]; T \otimes [q \supset \bot]; T\big) \\
\quad \text{— by (4.13) and distributivity} \\
\langle p \vee \neg p\rangle; \Big([p]; \langle q \vee \neg q\rangle; \big([q]; S \otimes [q \supset \bot]; T\big) \otimes [p \supset \bot]; T \otimes [p \supset \bot]; [q \supset \bot]; T\Big) \\
\quad \text{— by (5.2), (5.3), (4.6), (4.7), (4.9), (4.10), (4.14), and distributivity} \\
\langle p \vee \neg p\rangle; \Big([p]; \langle q \vee \neg q\rangle; \big([q]; S \otimes [q \supset \bot]; T\big) \otimes [p \supset \bot]; T\Big) \qquad \text{— by (4.9), (4.4)} \\
\textbf{if } p \textbf{ then } \ \textbf{if } q \textbf{ then } S \textbf{ else } T \ \ \textbf{else } T
\end{array}$$

We can also derive a law for the sequential disjunction:

$$\textbf{if } p \vec{\vee} q \textbf{ then } S \textbf{ else } T \quad \textbf{if } p \textbf{ then } S \textbf{ else } \ \textbf{if } q \textbf{ then } S \textbf{ else } T\ ,$$

where p is consistent. For the weak Kleene connectives, we have similar laws:

$$\begin{array}{l}
\textbf{if } p \wedge^w q \textbf{ then } S \textbf{ else } T \quad \textbf{if } p \textbf{ then } \ \textbf{if } q \textbf{ then } S \textbf{ else } T \ \ \textbf{else } \langle q \vee \neg q\rangle; T \ \text{ and} \\
\textbf{if } p \vee^w q \textbf{ then } S \textbf{ else } T \quad \textbf{if } p \textbf{ then } \langle q \vee \neg q\rangle; S \textbf{ else } \ \textbf{if } q \textbf{ then } S \textbf{ else } T\ ,
\end{array}$$

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 [LvW97, Gro00]. Below we list several non-trivial laws for propagating context information.

$$\{p\}; X : \quad e \sqsubseteq X : \quad e; \{\exists v.\ p\ X\backslash v\ \wedge X \quad e\ X\backslash v\ \} \tag{5.4}$$

$$\{p\}; [q] \sqsubseteq [q]; \{p \wedge q\} \tag{5.5}$$

$$\{p\}; \langle q\rangle \sqsubseteq \langle q\rangle; \{p \wedge q\} \tag{5.6}$$

$$\{p\}; \textbf{if } q \textbf{ then } S \textbf{ else } T \sqsubseteq \textbf{if } q \textbf{ then } \ \{p \wedge q\}; S \ \ \textbf{else } \ \{p \wedge\ q \supset \bot\ \}; T \tag{5.7}$$

$$\textbf{if } q \textbf{ then } \ S; \{p\} \ \ \textbf{else } \ T; \{q\} \ \sqsubseteq \ \textbf{if } q \textbf{ then } S \textbf{ else } T\ ; \{p \vee q\} \tag{5.8}$$

$$\{p\}; \textbf{try } S \textbf{ catch } T \sqsubseteq \textbf{try } \{p\}; S \textbf{ catch } T \tag{5.9}$$

$$\textbf{try } S; \{p\} \textbf{ catch } \ \ T; \{q\} \ \sqsubseteq \ \textbf{try } S \textbf{ catch } T\ ; \{p \vee q\}; \tag{5.10}$$

Let us consider a program that implements a numerical algorithm:

$$S_0 \triangleq X : \quad N; \textbf{try repeat } Y : \quad X; X : \quad Y \times Y \quad N \ \div\ 2 \times Y \ \ \textbf{until } X \geq Y \textbf{ catch skip}.$$

This program computes the integral value of $\sqrt{N}$ for non-negative integer N, based on the Newton-Raphson method [PTVF07], 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 $\sqsubseteq$ [BvW98]. The loop statement in S_0 is given by the least fixpoint of the function:

$$\mathcal{F}\ T \triangleq Y :\quad X;X :\quad Y \times Y \quad N \ \div\ 2 \times Y\ ;\textbf{if } X \geq Y \textbf{ then skip else } T.$$

In order to express the possible partiality caused by division-by-zero, we interpret the partial assignment statement $X :\quad Y \times Y \quad N \ \div\ 2 \times Y$ by $\langle \neg\ Y \quad 0\ \rangle; X :\quad Y \times Y \quad N \ \div'\ 2 \times Y$, where $\div'$ is a total extension of $\div$ such that division by 0 yields a fixed constant value (say, 0), instead of being undefined.

In the following derivation, we refine the original program S_0 by a program that makes no uses of exceptional statements.

$$\begin{array}{ll} S_0 & X :\quad N; \textbf{try } [\neg\ X \quad 0\ ]; \mu.\mathcal{F} \otimes [\neg\ X \quad 0\ \supset \bot]; \mu.\mathcal{F} \textbf{ catch skip} \\ & \quad \text{— by (4.13) and distributivity} \\ & X :\quad N; (\ \textbf{try } [\neg\ X \quad 0\ ]; \{0 < X \leq N\}; \mu.\mathcal{F} \textbf{ catch skip} \otimes \\ & \qquad\qquad \textbf{try } [\neg\ X \quad 0\ \supset \bot]; \{\neg\ X \quad 0\ \supset \bot\}; \mu.\mathcal{F} \textbf{ catch skip}\ ) \\ & \quad \text{— by distributivity and (4.6)} \end{array}$$

In order to show the refinement of the left substatement of the demonic choice, we need some lemmas.

Lemma 5.1.

$$\begin{array}{rl} & \{0 < X \leq N\}; \textbf{repeat } Y :\quad X;X :\quad Y \times Y \quad N \ \div\ 2 \times Y \ \textbf{ until } X \geq Y \\ \sqsubseteq & \textbf{repeat } \{0 < X \leq N\}; Y :\quad X;X :\quad Y \times Y \quad N \ \div'\ 2 \times Y \ \textbf{ until } X \geq Y \end{array}$$

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 [Nis].

$$\begin{array}{rl} & \textbf{try } [\neg\ X \quad 0\ ]; \{0 < X \leq N\}; \mu.\mathcal{F} \textbf{ catch skip} \\ \sqsubseteq & \textbf{try } [\neg\ X \quad 0\ ]; \textbf{repeat } \{0 < X \leq N\}; Y :\quad X;X :\quad Y \times Y \quad N \ \div'\ 2 \times Y \\ & \qquad \textbf{until } X \geq Y \textbf{ catch skip} \qquad \text{— by lemma 5.1} \end{array}$$

$\sqsubseteq$ **try** $\lceil \neg X \quad 0 \rceil$;**repeat** $Y: \quad X; \{\neg Y \quad 0\}; X: \quad Y \times Y \quad N \div' 2 \times Y$
until $X \geq Y$ **catch skip** — by (5.4) and (4.3)

$\lceil \neg X \quad 0 \rceil$;**repeat** $Y: \quad X; \{\neg Y \quad 0\}; X: \quad Y \times Y \quad N \div' 2 \times Y$
until $X \geq Y$ — by (4.24) and non-exceptionality from lemma 5.2

$\lceil \neg X \quad 0 \rceil$;**repeat** $Y: \quad X; X: \quad Y \times Y \quad N \div 2 \times Y$ **until** $X \geq Y$
— by (4.5) and (4.1)

For the other substatement, we derive:

try $\lceil \neg X \quad 0 \supset \bot \rceil; \{\neg X \quad 0 \supset \bot\}; \mu.\mathcal{F}$ **catch skip**
$\sqsubseteq$ **try** $\big(\lceil \neg X \quad 0 \supset \bot \rceil; Y: \quad X; \{\neg Y \quad 0 \supset \bot\}; \langle \neg Y \quad 0 \rangle;$
$X: \quad Y \times Y \quad N \div' 2 \times Y$;**if** $X \geq Y$ **then skip else** $\mu.\mathcal{F}\big)$ **catch skip**
— fixpoint; by (5.4)
try $\lceil \neg X \quad 0 \supset \bot \rceil; Y: \quad X; \{\neg Y \quad 0 \supset \bot\};$**exit catch skip**
— by (4.6) and (4.15)
$\sqsubseteq$ $\lceil \neg X \quad 0 \supset \bot \rceil; Y: \quad X.$ — by (4.25) and (4.1)

Therefore the derivation ends up with:

$S_0 \sqsubseteq X: \quad N; \ \lceil \neg X \quad 0 \rceil; \mu.\mathcal{F} \otimes \lceil \neg X \quad 0 \supset \bot \rceil; Y: \quad X$
$X: \quad N;$**if** $\neg X \quad 0$ **then repeat** $Y: \quad X; X: \quad Y \times Y \quad N \div 2 \times Y$ **until** $X \geq Y$
else $Y: \quad X.$

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 logical platform, we developed a four-valued predicate transformer semantics, which is based on Arieli and Avron's four-valued logic [AA96]. 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

[AA96] Ofer Arieli and Arnon Avron. Reasoning with logical bilattices. *Journal of Logic, Language, and Information*, 5(1):25–63, 1996.

[AA98] Ofer Arieli and Arnon Avron. The value of four values. *Artificial Intelligence*, 102(1):97–141, 1998.

[Bel77] N. D. Belnap. A useful four-valued logic. In G. Epstein and J. M. Dunn, editors, *Modern Uses of Multiple-Valued Logic*, pages 7–37. Reidel Publishing Company, 1977.

[BK06] Viviana Bono and Manfred Kerber. Extending Hoare calculus to deal with crash. Technical Report CSR-06-08, School of Computer Science, The University of Birmingham, 2006.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Fit94] Melvin Fitting. Kleene's three-valued logics and their children. *Fundamenta Informaticae*, 20(1/2/3):113–131, 1994.

[Gin88] Matthew L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[Gro00] Lindsay John Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[Häh05] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303, 2000.

[JM94] Cliff B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1986.

[KM95] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54–76, 1995.

[LvW97] Linas Laibinis and Joakim von Wright. Context handling in the refinement calculus framework. Technical Report 118, TUCS Technical Report, 1997.

[MB99] Joseph M. Morris and Alexander Bunkenburg. E3: A logic for reasoning equationally in the presence of partiality. *Science of Computer Programming*, 34(2):141–158, 1999.

[Mor94] Carroll Morgan. *Programming from specifications*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 2nd edition, 1994.

[Nis] Susumu Nishimura. Refining exceptions in four-valued logic. available online at `http://www.math.kyoto-u.ac.jp/~susumu/papers/lopstr09-full.pdf`.

[Owe93] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.

[Wat02] Geoffrey Watson. Refining exceptions using King and Morgan's exit construct. In *9th Asia-Pacific Software Engineering Conference (APSEC 2002)*, pages 43–51. IEEE Computer Society, 2002.

Towards a framework for constraint-based test case generation

Tom Schrijvers*[1], François Degrave**[2], and Wim Vanhoof[2]

[1] Department of Computer Science
Katholieke Universiteit Leuven
[2] Faculty of Computer Science
University of Namur

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. In previous work [4], we have developed a technique that allows to automatically generate a set of test inputs for a given program written in the logic programming language Mercury. The basic idea of that technique, itself inspired by [20], is as follows: first, a path through the control-flow graph of a predicate is transformed into a set of constraints on the (input) variables of the predicate such that when the predicate is called with input values satisfying these constraints, its execution is guaranteed to follow the given path. Next, a dedicated constraint solver is used to compute such concrete input values. By repeating the process for a carefully constructed finite subset of all execution paths one can guarantee that the set of generated input values *cover* a substantial part of the source code and can as such be used to perform so-called *structural* or *whitebox* testing.

In the current paper, which is a report on work in progress, we generalize and extend the approach of [4], reformulating it completely within the framework of constraint programming (CP). This presents several advantages over [4] and similar proposals:

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.
** Supported by a grant FRIA - Belgium.

– The whole process – from the construction of an execution path to the generation of concrete test inputs – is now modeled as a single constraint problem, rather than as a pair of complementary processes. This presents a cleaner and more uniform formalisation of the technique.
– More importantly, since the selection of an execution path is modeled within the constraint problem, one can obtain different degrees of coverage, or even coverage with respect to different criteria, by using different search strategies within the solver. In other words, the technique can – at least to a certain extent – be seen as parametrised with respect to a coverage criterion or a desired degree of coverage.
– The resulting technique is also more efficient, since it avoids the construction of constraint sets representing paths that do not correspond to a real execution.

In order to show the power of our generalized approach, we cast it in the setting of a (small) deterministic imperative language with 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. 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 2.5 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	$e ::= x \mid n \mid \texttt{nil} \mid \texttt{new cons}(e_1, e_2) \mid e\texttt{.head} \mid e\texttt{.tail}$
	$\mid e_1 \texttt{ == } e_2 \mid e_1 \texttt{ /= } e_2 \mid e_1\texttt{+}e_2$
statements	$s ::= \texttt{skip} \mid l \texttt{ := } e \mid s_1\texttt{;}s_2 \mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2$
	$\mid \texttt{while } e \texttt{ \{ } s \texttt{ \}}$
left-hand sides	$l ::= x \mid l\texttt{.head} \mid l\texttt{.tail}$

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 `+`.[3] We will assume that ImpL is simply typed and only allows comparison of two values belonging to the same type (either integers or references).[4] 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

[3] Other arithmetic operators are omitted in order to keep the formal definition of the semantics small, but they can be added at will.
[4] Integers are also used as booleans: 0 denotes false and all other integers denote true.

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 while loop, the list will have the value `[1,2,3]`.

2 Generating test inputs

2.1 Overview

As usual, 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 $\texttt{ptr}(r)$ with r a unique value denoting the address of the cons cell on the heap. Likewise, a heap is a finite mapping from such references r to cons cells of the form $\texttt{cons}(v_h,v_e)$ 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:

$E : x \mapsto \texttt{ptr}(r_1)$	$E : x \mapsto \texttt{ptr}(r_1)$
$H : r_1 \mapsto \texttt{cons(1,ptr}(r_2)\texttt{)}$ $r_4 \mapsto \texttt{cons(3,ptr}(r_5)\texttt{)}$	$H : r_1 \mapsto \texttt{cons(1,ptr}(r_2)\texttt{)}$
$r_2 \mapsto \texttt{cons(2,ptr}(r_3)\texttt{)}$ $r_5 \mapsto \texttt{cons(4,nil)}$	$r_2 \mapsto \texttt{cons(2,ptr}(r_3)\texttt{)}$
$r_3 \mapsto \texttt{cons(3,ptr}(r_4)\texttt{)}$	$r_3 \mapsto \texttt{cons(3,nil)}$

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 form 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. Figure 1 illustrates the search tree for the example.

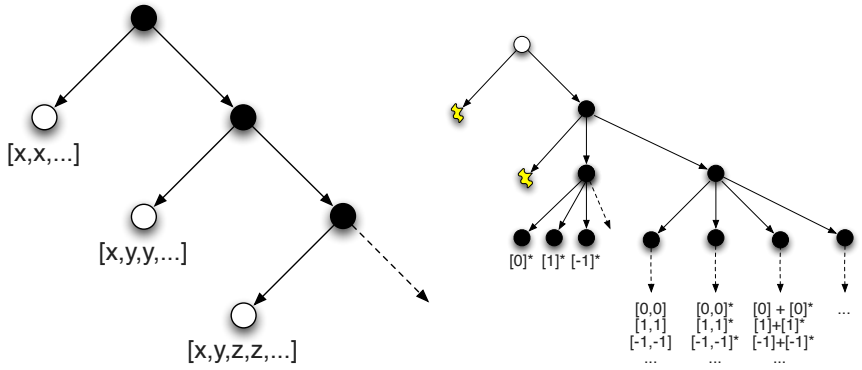


Fig. 1. Tree of paths (left) and tree of value assignments for left-most path (right).

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:

- $o_1 = o_2$, 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 2.3.

The symbolic semantics is depicted in Figures 2 and 3. 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 $\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1$ 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 $\{E_0, H_0, C_0\}\ e \rightsquigarrow v; H_1; C_1$ and $\{E_0, H_0, C_0\} s \langle E_1, H_1, C_1\rangle$ denote, respectively, $\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1$ 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).[5] Formally:

$$\text{(Cond-e)}\ \frac{\mathcal{T} \models C_0 \qquad \langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1}{\{E_0, H_0, C_0\}\ e \rightsquigarrow\ v; H_1; C_1}$$

$$\text{(Cond-s)}\ \frac{\mathcal{T} \models C_0 \qquad \langle E_0, H_0, C_0\rangle\ s\ \langle E_1, H_1, C_1\rangle}{\{E_0, H_0, C_0\}\ s\ \langle E_1, H_1, C_1\rangle}$$

The use of conditional evaluation avoids adding further constraints to an already inconsistent set. This implies that search strategies (see Section 2.4) will only explore execution paths that can model a real execution.

2.3 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) [9].

[5] In practice, the consistency check may be incomplete. Then unreachable execution paths may be explored.

$$\text{(Var)}\ \frac{V \text{ fresh}}{\langle E, H, C\rangle\ x \rightsquigarrow V; H; C \wedge \{x \mapsto V\} \in E} \qquad \text{(Int)}\ \frac{n \in \mathbb{Z}}{\langle E, H, C\rangle\ n \rightsquigarrow n; H; C}$$

$$\text{(Nil)}\ \langle E, H, C\rangle\ \texttt{nil} \rightsquigarrow \texttt{nil}; H; C$$

$$\text{(Cons)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2 \qquad H_3, r \text{ fresh}}{\langle E, H_0, C_0\rangle\ \texttt{new cons}(e_1, e_2) \rightsquigarrow \texttt{ptr}(r); H_3; C_2 \wedge H_3 = H_2 \uplus \{r \mapsto \texttt{cons}(v_1, v_2)\}}$$

$$\text{(Head)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0\rangle\ e\texttt{.head} \rightsquigarrow V_h; H_1; C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}$$

$$\text{(Tail)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0\rangle\ e\texttt{.tail} \rightsquigarrow V_t; H_1; C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}$$

$$\text{(EqualT)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 1; H_2; C_2 \wedge v_1 = v_2}$$

$$\text{(EqualF)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 0; H_2; C_2 \wedge v_1 \neq v_2}$$

$$\text{(NEqualT)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 1; H_2; C_2 \wedge v_1 \neq v_2}$$

$$\text{(NEqualF)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 0; H_2; C_2 \wedge v_1 = v_2}$$

$$\text{(Add)}\ \frac{\langle E, H_0, C_0\rangle\ e_1 \rightsquigarrow v_1; H_1; C_1 \qquad \{E, H_1, C_1\}\ e_2 \rightsquigarrow v_2; H_2; C_2 \qquad v \text{ fresh}}{\langle E, H_0, C_0\rangle\ e_1 + e_2 \rightsquigarrow v; H_2; C_2 \wedge v = v_1 + v_2}$$

Fig. 2. Symbolic evaluation of expressions.

$$\text{(Skip)}\ \langle E, H, C\rangle\ \texttt{skip}\ \langle E, H, C\rangle$$

$$\text{(VarAss)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad E_1 \text{ fresh}}{\langle E_0, H_0, C_0\rangle\ x\ \texttt{:=}\ e\ \langle E_1, H_1, C_1 \wedge E_1 = E_0 \uplus \{x \mapsto v\}\rangle}$$

$$\text{(HeadAss)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r; H_1; C_2 \qquad R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0\rangle\ l\texttt{.head :=}\ e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(v, V_t)\}\rangle}$$

$$\text{(TailAss)}\ \frac{\langle E, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r; H_1; C_2 \qquad R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0\rangle\ l\texttt{.tail :=}\ e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(V_h, v)\}\rangle}$$

$$\text{(Seq)}\ \frac{\langle E_0, H_0, C_0\rangle\ s_1\ \langle E_1, H_1, C_1\rangle \qquad \{E_1, H_1, C_1\}\ s_2\ \{E_2, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ s_1\texttt{;}s_2\ \langle E_2, H_2, C_2\rangle}$$

$$\text{(IfThen)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s_1\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(IfElse)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad \{E_0, H_1, C_1 \wedge v = 0\}\ s_2\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(WhileT)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1 \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s\texttt{;while}\ e\ \{\ s\ \}\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2, C_2\rangle}$$

$$\text{(WhileF)}\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v; H_1; C_1}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_0, H_1, C_1 \wedge v = 0\rangle}$$

Fig. 3. Symbolic execution of statements.

2.4 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.

$$V_r = \texttt{nil} \vee (\bigvee_{r \in R} V_r = \texttt{ptr}(r)) \vee (V_r = \texttt{ptr}(r') \wedge (r' \mapsto \texttt{cons}(V_i', V_r')) \in H_0)$$

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.

2.5 Generalized Data Structures

So far we have only considered data structures composed of simple `cons` cells. 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, 18].

3 Ongoing Work

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 [1, 7]. In *symbolic evaluation* techniques (e.g. [3, 14, 16]), 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 [15]. 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 [13, 8].

Constraint-based test data generation was originally introduced in [6] in the context of mutation testing [5] 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 [11, 12]. It is also used in two different testing tools, Godzilla [17] and InKA [10]. The latter notably generates test data satisfying different criteria such as statement coverage, branches coverage and MC/DC[6].

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 developed an initial prototype in SWI-Prolog, which is available at `http://www.cs.kuleuven.be/~toms/Testing/` together with example programs. As we've already said, Prolog's advantage is that it provides most of the required constraints for free, and the missing ones are easily implemented in CHR. However, Prolog's disadvantage is that depth-first search is built in, which makes experimenting with alternative search strategies hard – if not impossible.

We are currently working on a new implementation of our approach within the Monadic Constraint Programming framework [19], which is an open framework for building complex search strategies as a composition of simpler ones. This system will enable us to 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.

[6] Modified condition/decision coverage [2]

References

1. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.
2. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sep 1994.
3. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.
4. F. Degrave, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for Mercury. In *Logic-Based Program Synthesis and Transformation*. Springer, 2008.
5. R.A. DeMillo. Test adequacy and program mutation. In *Software Engineering, 1989. 11th International Conference on*, pages 355–356, May 1989.
6. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, Sep 1991.
7. J. W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.
8. Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
9. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
10. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.
11. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A clp framework for computing structural test data. In *Proceedings of the First International Conference on Computational Logic*, pages 399–413, London, UK, 2000. Springer-Verlag.
12. Arnaud Gotlieb, Tristan Denmat, and Bernard Botella. Goal-oriented test data generation for programs with pointer variables. *Computer Software and Applications Conference, Annual International*, 1:449–454, 2005.
13. Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Softw. Eng. Notes*, 23(6):231–244, 1998.
14. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
15. C. Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic executionatgen. *Software Testing Verification and Reliability*, 11(2):81–96, 2001.
16. Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.
17. A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation: Design and algorithms. *Software - Practice and Experience*, 29:167–193, 1994.
18. A. Jefferson Offutt and Shaoying Liu. Generating test data from sofl specifications. *The Journal of Systems and Software*, 49:49–62, 1999.
19. Tom Schrijvers, Peter Stuckey, and Phil Wadler. Monadic Constraint Programming, 2009. `http://www.cs.kuleuven.be/~toms/Haskell/`.
20. Nguyen Tran Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *Proceedings of ASE 01*, 2001.
21. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

Using Rewrite Strategies for Testing BUpL Agents

Lăcrămioara Aştefănoaei*[1], Frank S. de Boer[1], M. Birna van Riemsdijk[2]

[1] CWI, Amsterdam, The Netherlands
[2] TU, Delft, The Netherlands

Abstract. In this paper we focus on the problem of testing agent programs written in BUpL, an executable high-level agent modelling 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 [15]. An important line of research in the agent systems field is the design of agent languages [2] with emphasis on the use of formal methods. The guiding idea is that agent-specic 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 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 [8], the agent language BUpL, which is introduced in [1]. There the authors advocate the use of the Maude language [5] and its supporting tools for both *prototyping* 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. Maude is a high-performance reflective language and system supporting equational and rewriting logic specification and programming. The language has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework, through which programming languages with an operational semantics can be implemented in a rigorous way [11]. Not only is it possible to prototype and execute operational semantics in Maude, but also to verify temporal properties of the prototyped programs using the Maude LTL model-checker [6]. Furthermore, Maude facilitates the specification of strategies for controlling the application of rewrite rules.

* **Email**: L.Astefanoaei@cwi.nl; **Address**: Centrum voor Wiskunde en Informatica (CWI),P.O. Box 94079,1090 GB Amsterdam, The Netherlands; **Tel.**: +31 (0)20 592 4368

This paper extends the results from [1] by investigating the problem of agent correctness, however for infinite state agents. In [1], verification was achieved by means of LTL model-checking. Model-checking works only for finite state systems. In fact, sometimes model-checking fails even in the finite case. This is because of the state explosion problem. In such conditions, when directly proving correctness is no longer possible, new techniques come into view. For example, one can build finite (or small) abstractions which can be model-checked instead, following the counterexample-guided abstraction methodology [4]. Or one can define a logic in the Hoare style [9] and use annotations in order to reason about the properties of the system. We, however, consider yet another possibility which is to "simply" look for bugs (failures) by *testing* the system. This is because, as it will be later clear, test cases, as we define them, have a natural mapping into rewrite strategies. Since we work with the Maude system and since a strategy language has already been integrated into Maude [7], extending the previous implementation was an easy process which allowed us to experiment with our definitions in a short time.

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, criterias for test selection (please see [12] for an overview). In an agent-oriented setup, there are less references. A few pointers are [16, 13] for developing test units from different agent methodologies, however the direction is orthogonal to the one we consider.

Our intention is to follow the *model-based testing* methodology. Roughly, model-based testing involves 4 stages: (1) build an abstract model of the system under test, (2) validate the model, (3) generate test cases from the model, (4) apply test cases to the system under test and compare the output to the models output. For simplicity, in this paper we do not discuss the first two stages. We only assume that each agent implementation follows a specification which is assumed to be valid by definition. Instead, we are concerned with the last stages. In order to achieve (3), an important step is to have a *systematic method* for specifying *what* we want to test, i.e., test cases. This is one issue we focus upon. We define a formal language for the specification of test cases for individual BUpL agents. Test cases are then defined by regular expressions where the basic elements are observable actions and beliefs. An agent passes a test when it successfully executes certain actions in a certain order with certain results reflected in mental states. We make the short note that we deliberately choose to design in our language tests on beliefs in order to have a more expressive formalism. One might raise the issue that inspecting the mental states of the agents classifies our method as white-box testing. However, since beliefs can be deduced from the effects of the observable actions, in fact, our method lies at the boundary between black-box and gray-box testing. To define test cases, there is no need in understanding 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 observable actions, which we see as the interface of BUpL agents.

The other issue we focus upon relates to (4), and is, namely, *how* to test. Our method to perform *testing* is to use a strategy-based mechanism for implementing test cases. In

a rewriting 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 and exception handling mechanisms is a main aspect we deal with. It is important to note that strategies are a means of clearly separating between the executions of programs (with respect to a given semantics) at *object-level* and the control of executions at *meta-level* (without any change in the semantics). 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 another testing formalism one only needs to change the *strategy* instead of changing *the semantics of the agent language* or *the agent program* itself.

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 non-deterministic 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 by another.

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 *tower of blocks* problem. We represent blocks by natural numbers. We assume that initially there are three 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 an 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.

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

$\mathcal{B}_0$ = { $on(3,1), on(1,0), on(2,0), clear(2), clear(3), clear(0)$ }

$\mathcal{A}$ = { $move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \{ on(x,z), \neg on(x,y), \neg clear(z) \})$ }

$\mathcal{P}$ = { $build = move(2,0,1); move(3,0,2)$ }

$\mathcal{R}$ = { $on(x,y) \leftarrow move(x,y,0); build$ }

Fig. 1: A BUpL Toy Agent

$on(x,y) \leftarrow move(x,y,0); build$. 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 the BUpL semantics. The states of BUpL agents are pairs of belief bases and plans. These change with respect to the transition rules in Figure 2.

$$\frac{p = (a; p') \quad a = (\psi, \xi) \in \mathcal{A} \quad \theta \in Sols(\mathcal{B} \models \psi)}{(\mathcal{B}, p) \xrightarrow{a\theta} (\mathcal{B} \uplus \xi\theta, p'\theta)} \; ((i/o)\text{-}act)$$

$$\frac{}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\tau} (\mathcal{B}, p_i)} \; (sum_i)$$

$$\frac{(\mathcal{B}, a; p) \not\xrightarrow{a} \quad \phi \leftarrow p' \in \mathcal{R} \quad \theta \in Sols(\mathcal{B} \models \phi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \; (fail\text{-}act)$$

$$\frac{\pi(x_1, \ldots, x_n) := p}{(\mathcal{B}, \pi(t_1, \ldots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \ldots, t_n))} \; (\pi)$$

Fig. 2: BUpL Rules

The rule $((i/o)\text{-}act)$ captures the effects of performing action a (either internal or observable), which is the head of the current plan. If θ is a solution to the matching problem[3] $\mathcal{B} \models \psi$, where ψ is the precondition of a, then the current mental state changes to a new one, where the current belief base is updated with the effects of a and the current plan becomes the "tail" of the previous one. 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.

[3] We make the short observation that since belief bases are always ground, we need to solve matching and not unification problems.

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 [14], 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 non-deterministic, which makes rewriting logic a good candidate for modelling concurrency.

In our case, the signature describes the mental states of the agents. The rewrite rules 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 [7] with illustrations of how strategies can be used to control the execution of BUpL agents. We first 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 [7]. 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 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/failure result of E but it does not change the initial state. The strategy $E!$ = E^* ; *not*(E) "repeats until the end".

3 Testing BUpL Agents

Searching can be viewed as an ad hoc way of testing. While it may work for certain cases, it has several drawbacks. As for model-checking, state space explosion may be a problem since the whole state space is considered (if no bound is used on the search). Moreover, it works with invariants expressed over the states of the system, while one may also want to test other properties such as the execution of certain sequences of actions.

3.1 Formalising Test Cases

Our test case format is based on two main BUpL concepts: observable actions and beliefs. 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. Sequences of actions are defined as regular expressions, and properties of belief bases can be specified in a subset of LTL with only $\Box$ (*always*) and $\Diamond$ (*eventually*) as temporal operators. The idea is that the action expression of a test is used to generate execution traces satisfying the action expression. That is, the action expression controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. This is crucial for reducing the state space, and makes this approach essentially different from searching.

The following BNF grammar defines the language $\mathcal{T}$ of test cases, where a denotes a ground observable action, R a set of observable actions, x is a ground atom and bel is a predicate which holds when a certain belief is present in the current belief base.

$$\begin{gathered}\mathcal{T}_a ::= idle \mid (a, R) \mid \mathcal{T}_a; \mathcal{T}_a \mid \mathcal{T}_a + \mathcal{T}_a \mid \mathcal{T}_a^* \\ \mathcal{T}_b ::= bel(x) \mid \neg\mathcal{T}_b \mid \mathcal{T}_b \wedge \mathcal{T}_b \mid \Box\mathcal{T}_b \mid \Diamond\mathcal{T}_b \\ \mathcal{T} ::= \mathcal{T}_a \mid (\mathcal{T}_a, \mathcal{T}_b) \mid \mathcal{T}; \mathcal{T} \mid \mathcal{T} + \mathcal{T}\end{gathered}$$

$\mathcal{T}_a$ and $\mathcal{T}_b$ are the languages of expressions over actions and beliefs, respectively. $\mathcal{T}_a$ defines regular expressions over pairs (a, R), which express that observable action a should be executed while R is a set of observable actions forming a subset of those actions that are ready (enabled) to be executed. This provides additional expressivity in comparison with a variant where one could only say that some action a should be executed. The latter can be expressed in our language by $(a, \emptyset)$ (for convenience denoted simply as a), since $\emptyset$ is trivially a subset of the enabled actions. A test case can be an expression over actions, a pair consisting of an action expression and a belief expression, or a test composed using sequential composition or nondeterministic choice. As suggested above, a test $(\mathcal{T}_a\ , \mathcal{T}_b)$ informally means that the actions of $\mathcal{T}_a$ should be executed and the belief bases of the corresponding trace should satisfy $\mathcal{T}_b$. The sequential composition of two tests is satisfied by a trace if the first part of the trace satisfies the first test, and the second part satisfies the second test, and similarly for non-deterministic choice. We note that $\mathcal{T}_b$ is not a test on its own, since then we would lose the control over the execution provided by the action expression, which is necessary for reducing the state space. If $\mathcal{T}_b$ could be a test on its own, this would come down to model-checking for satisfaction of $\mathcal{T}_b$.

We now define formally when a BUpL agent satisfies a test. We denote the application of a test $\mathcal{T}$ on an initial configuration (an initial BUpL mental state) ms_0 as $\mathcal{T}@ms_0$ and we define its semantics (more precisely, its *set semantics*) inductively, on the structure of tests. The 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*. Since one usually tests for the absence of "bad" execution paths, we say that a BUpL agent with initial mental state ms0 is *safe with respect to a test* $\mathcal{T}$ if the application of the test fails, i.e., $\mathcal{T}@ms_0 = \emptyset$.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms_0\}, & \mathcal{T} = idle \\ \{ms \mid ms_0 \stackrel{a}{\Rightarrow} ms\}, & \mathcal{T} = (a, R) \wedge R \subseteq R(ms) \\ \emptyset, & \mathcal{T} = (a, R) \wedge R \not\subseteq R(ms) \\ \mathcal{T}_1@ms_0 \cup \mathcal{T}_2@ms_0, & \mathcal{T} = \mathcal{T}_1 + \mathcal{T}_2 \\ \mathcal{T}_2@(\mathcal{T}_1@ms_0), & \mathcal{T} = \mathcal{T}_1; \mathcal{T}_2 \\ \{ms_0\} \cup \bigcup_{i \geq 1} \mathcal{T}_1^i@ms_0, & \mathcal{T} = \mathcal{T}_1^* \\ \mathcal{T}_a@ms_0, & \mathcal{T} = (\mathcal{T}_a, \mathcal{T}_b) \wedge (\mathcal{T}_a, ms_0 \models_t \mathcal{T}_b) \\ \emptyset, & \mathcal{T} = (\mathcal{T}_a, \mathcal{T}_b) \wedge (\mathcal{T}_a, ms_0 \not\models_t \mathcal{T}_b) \end{cases}$$

The arrow $\stackrel{a}{\Rightarrow}$ stands for $\Rightarrow\stackrel{a}{\rightarrow}\Rightarrow$, where $\Rightarrow$ denotes the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$. $\mathcal{T}^i@t$ is $\mathcal{T}@\mathcal{T}^{i-1}@t$ 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 satisfiability relation $\models_t$ is defined as an extension of $\models_{LTL}$:

$$\begin{array}{ll} \mathcal{T}_a, ms_0 \models_t bel(x) & \text{if } ms_0 = (\mathcal{B}, p) \wedge x \in \mathcal{B} \\ \mathcal{T}_a, ms_0 \models_t \mathcal{T}_b & \text{if } (\forall \sigma \in Paths_t(\mathcal{T}_a, ms_0))(\sigma \models_{LTL} \mathcal{T}_b) \end{array}$$

with $Paths_t(\mathcal{T}_a, ms_0)$ denoting the paths from ms_0 taken while executing the test $\mathcal{T}_a$.

We explain the semantics of $(a, R)@ms_0$ in some more detail. The idea is that the test should be successful for ms_0 if action a can be executed in ms_0, while R is a subset of the enabled actions (defined by $R \subseteq R(ms)$). 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 $\mathcal{T}_1; \mathcal{T}_2$ we need the mental states resulting from applying the test $\mathcal{T}_1$, since those are the mental states in which we then apply the test $\mathcal{T}_2$, as defined by $\mathcal{T}_2@(T_1@ms_0)$.

3.2 Using Rewrite Strategies to Implement Test Cases

In this section we describe how the strategy language S can be used for implementing test cases. To give some intuition and motivation, we consider the way one would implement the basic test case a. As we have defined it above, 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 [7], we promote the design principle that automated deduction methods (e.g., closure sets of τ steps) should be specified *declaratively* as inference systems and not *procedurally*. Depending on the application, specific algorithms for implementing the inference systems should be specified 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.

While implementing the test a, the closure sets we need to consider are with respect to the rules *act-fail*, *sum* and *i-act*. We make the short note that for a different agent language the closure computations could be different, depending on the semantics of the language. In our case, the implementation we propose is as follows:

$$do(a) = try(sum)\ ;\ i\text{-}act\ !\ ;\ try(fail\text{-}act)\ ;\ test(o\text{-}act)\ ?\ o\text{-}act[o\text{-}a \leftarrow a]\ :\ (\ test(match\ (\mathcal{B}, nil))\ ?\ fail\ :\ do(a)\ )$$

It basically first reduces the current plan to plans containing only the sequence operator. The strategy further computes the closure of internal actions. Executing *i-act* might lead to failures, thus also repair rules are applied. At this point we test whether with the new plans it is possible to execute an observable action. If this is the case then we apply the rewrite rule *o-act* where we substitute the variable *o-a* by the parameter a. The application either succeeds or fails (in the case where the observable action is different from the one we want). If *test(o-act)* fails then the head of the plan is not observable but an internal action, thus we need to repeat the sequence of closure computations for sum, internal actions and repair rules.

The test whether a certain fact is believed (after eventually applying internal actions, repair rules and plan choices) is implemented as follows:

$$bel(x) = (test(match\ (\{x\} \cup \mathcal{B}, p))\ ?\ fail\ :\ i\text{-}act)\ !\ ;\ (test(\ match\ (\{x\} \cup \mathcal{B}, p))\ ?\ idle\ :\ (\ test(match\ (\mathcal{B}, nil))\ ?\ fail\ :\ try(fail\text{-}act)\ ;\ try(sum)\ ;\ not(o\text{-}act)\ ;\ bel(x)\ )\ )$$

The strategy $bel(x)$ checks first if x has been added to the belief base. If this is the case, then we are done. Otherwise, it repeatedly applies internal action until either x has been added or *i-act* fails. In this latter case, either the plan is nil and the strategy finishes with fail or an internal action fails and consequently *fail-act* is applied. After, it tries to apply *sum* reducing all plans to sequences. Any state where the plan has an observable action in the head is discarded (*not*(*o-act*)) since we look for new belief updates while executing only internal actions.

All other test cases are implemented by means of composing the strategies do and bel under the regular expression constructions. We do not detail them, a simple inductive reasoning with respect to the test case definition from Section 3.1 should suffice. We only consider that given a test case $\mathcal{T}$, the strategy implementing it is denoted by $s(\mathcal{T})$.

We now approach the issue of the correctness. Given a test case $\mathcal{T}$ and the corresponding strategy implementing it $s(\mathcal{T})$, in order to prove that the application of $s(\mathcal{T})$

is correct, we need to prove that, on the one hand, any success of the strategy is an error found by testing and on the other hand, if the strategy fails then the agent is safe with respect to the test.

Theorem 1 (Correctness). *Given ms a mental state, $\mathcal{T}$ a test case and $s(\mathcal{T})$ the strategy implementing $\mathcal{T}$, we have that $s(\mathcal{T})@ms \subseteq \mathcal{T}@ms$.*

Proof. We only consider the strategy do. The proof for bel is similar. The correctness of the possible compositions with respect to the regular expression constructions follows from the correctness of the strategy language.
Basically, we need to prove the following implications:

1. $do(a)@ms = Res$ and $Res \neq \emptyset \Rightarrow (\forall ms' \in Res)(ms \stackrel{a}{\Rightarrow} ms')$
2. $do(a)@ms = \emptyset \Rightarrow ms \stackrel{a}{\not\Rightarrow}$

1. From the definition of do:

$$Res = E@(\textit{try}(\textit{fail-act})@(\textit{i-act}!@\underbrace{(\textit{try}(\textit{sum})@ms)}_{Res_1})\underbrace{)}_{Res_2}\underbrace{)}_{Res_3}$$

with E = *test(o-act)* ? *o-act*[*o-a* $\leftarrow a$] : (*test*(*match* ($\mathcal{B}$, *nil*)) ? *fail* : *do*(a)).

Let $ms' \in Res$. We have that there exists $ms_3 \in Res_3$ (1) s.t. $ms_3 \stackrel{a}{\rightarrow} ms'$.
From (1) we have that there exists $ms_2 \in Res_2$ s.t. $ms_2 \stackrel{\tau}{\rightarrow} ms_3$ (2), with ms_2 being identical to ms_3 in the case the strategy *try*(*fail-act*) did not change the state (i.e., the rule *fail-act*) was not applicable).

From (2) we have that there exists $ms_1 \in Res_1$ s.t. $ms_1 \stackrel{\tau^*}{\rightarrow} ms_2$ (3), corresponding to executing internal actions.
From (3) we have that $ms \stackrel{\tau}{\rightarrow} ms_1$ (4), with ms being identical to ms_1 in the case the strategy *try*(*sum*) did not change the state.
From (4) - (1) we have that $ms \stackrel{\tau}{\rightarrow} ms_1 \stackrel{\tau^*}{\rightarrow} ms_2 \stackrel{\tau}{\rightarrow} ms_3 \stackrel{a}{\rightarrow} ms'$ that is $(ms \stackrel{a}{\Rightarrow} ms')$.

2. None of Res_i with $i \in \{1, 2, 3\}$ cannot be empty since by the definition the strategies *try* and "!" return at least the initial state, that is there exists $ms_3 \in Res_3$ such that $ms \stackrel{\tau^*}{\rightarrow} ms_3$, with ms_3 being identical to ms in the case none of the rewrite rules *sum*, *i-act*, *fail-act* were applicable. This means that $do(a)@ms = \emptyset$ only if $E@ms_3$ fails. There are two cases:
(1) *test(o-act)* is successful $\Rightarrow$ from ms_3 it is possible to execute an external action, however different from a, which means that *o-act*[*o-a* $\leftarrow a$] fails;
(2) *test(o-act)* fails $\Rightarrow$ *test*(*match* ($\mathcal{B}$, *nil*)) is successful, which means that the plan in ms_3 has reached its end.
In any case we have that $ms \stackrel{\tau^*}{\rightarrow} ms_3 \stackrel{a}{\not\rightarrow}$. □

This result allows us to conclude that if the application of $s(\mathcal{T})$ is successful then $\mathcal{T}$ is also successful.

We now approach the issue of completeness. We want to show that any error found by a test $\mathcal{T}$ is also found by the strategy $s(\mathcal{T})$ and that if an agent is safe with respect to $\mathcal{T}$ then $s(\mathcal{T})$ fails. Before proving these results, we present two helpful lemmas. We note that in the strategies do and bel we try to apply *fail-act* and *sum* only once. This is enough for the completeness of our test case implementation. Indeed, let us first consider *fail-act*. Intuitively, 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.

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. □

A similar 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.

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). From the definition of the rewrite rule *sum* we have that $(\mathcal{B}, p_i) \in$ *sum* $@ms$ $\forall i \in \{1, \ldots, n\}$. Since these are the only plans which can no longer be simplified by the rule *sum*, we have that they are the only elements of *sum*! $@ms$, thus *sum*! $@ms \subseteq$ *try*(*sum*) $@ms$. □

Theorem 2 (Completeness). *Given* ms *a mental state,* $\mathcal{T}$ *a test case and* $s(\mathcal{T})$ *the strategy implementing* $\mathcal{T}$*, we have that* $\mathcal{T}@ms \subseteq s(\mathcal{T})@ms$*.*

Proof. The proof is by structural induction on the definition of $\mathcal{T}$. Due to space limit, as it was the case when proving correctness, we only consider the basic test a. We need to prove the following implications:

1. $(ms \stackrel{a}{\Rightarrow} ms') \Rightarrow (ms' \in do(a)@ms)$
2. $(ms \stackrel{a}{\not\Rightarrow}) \Rightarrow (do(a)@ms = \emptyset)$

1. Let ms_3 be such that $ms \stackrel{\tau^k}{\rightarrow} ms_3 \stackrel{a}{\rightarrow} ms'$, with k a natural number. We have that $ms' \in$ *o-act*[*o-a* $\leftarrow a$] $@ms_3$ (1).

We now consider the sequence $ms \stackrel{\tau^k}{\rightarrow} ms_3$.

Let $ms = (\mathcal{B}, p)$. If p is a sum of plans then from ms a state $ms_i = (\mathcal{B}, p_i)$ with p_i being a basic plan of p can be reached in a number of τ steps, that is $ms \stackrel{\tau^*}{\Rightarrow} ms_i$. If p is not a sum, then we let p_i and ms_i be identical to p resp. ms. From Lemma 2 we obtain that $ms_i \in$ *try*(*sum*)@ms (2).
For the ease of notation, let $head$ (resp. $tail$) be a function on plans returning the first action of a plan (resp. a plan without its head). Let $ms_1 = \mathcal{B}_1, p_1)$ be the state reachable from ms_i by executing internal actions from p_i (if $head(p_i) \notin \mathcal{A}^i$ then $ms_1 = ms_i$). We have that $ms_1 \in$ *i-act*!@ms_i (3).
If $head(p_1)$ is an internal action which cannot be executed, then we let $ms_2 = (\mathcal{B}_1, p_2)$ be the state with an enabled $head(p_2)$ which results by applying repair rules (otherwise $ms_2 = ms_1$). From Lemma 1 we know that ms_2 (if exists) can be found by applying only once *fail-act*, thus $ms_2 \in$ *try*(*fail-act*)@ms_1 (4).
From ms_2 the same reasoning applies (wrt (2), (3), (4)) until ms_3 is reached (in a finite number of τ steps, smaller than k). Thus $ms_3 \in$ (*try*(*fail-act*) @ *i-act*! @ *try*(*sum*) @ *not*(*o-act*))! @ms (5).
From (1) and (5) we have that $ms' \in do(a)@ms$.

2. Let $ms_1 = (\mathcal{B}, p)$ be such that $ms \stackrel{\tau^k}{\rightarrow} ms_1$ and the τ steps correspond to the application of *sum*, *i-act* and *fail-act* until no longer possible (i.e., $ms_1 \stackrel{\mathcal{T}}{\not\rightarrow}$). This means that either p is nil or that $head(p) \in \mathcal{A}^o$ but different from a. Thus, $ms_1 \in$ (*try*(*fail-act*) @ *i-act*! @ *try*(*sum*) @ *not*(*o-act*))! @ms and *test(o-act)* ? *o-act*[*o-a* $\leftarrow a$] : (*match* ($\mathcal{B}$, *nil*) ; *fail*) @ms_1 fails, thus $do(a)@ms = \emptyset$. □

This result makes it possible to conclude that if the application of $s(\mathcal{T})$ to the initial mental state ms of a well-defined BUpL agent fails then ms is safe with respect to $\mathcal{T}$.

We conclude by making a short discussion on the termination of the application of strategies. It can be the case that the strategies do and bel do not terminate for certain agents. We take, as a trivial example, an agent with a recursive plan $p =$*i-a* $; p$, where $i_a \in \mathcal{A}^i$, e.i., is an internal action. We have that *i-act*! never terminates and thus neither does do. Termination is an important property which we would like to ensure. However, we need to impose certain constraints and this might result in loosing completeness.

4 A Running Example

The BUpL builder described in Figure 1 has a small number of states. Thus, verification by model-checking is feasible. In what follows we modify it so that the resulting state space is infinite, making model-checking infeasible. We consider that the programmer needs to implement a BUpL builder which respects the specification "the agent should always construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number", thus 21, 4321 are "well-formed" towers. Since it is out of the scope of this paper, our notion of specification is merely informal. We only mention that the specification, in model-based testing, plays the role of the *model*. Its main characteristics are that it serves a *specific purpose* (building even length towers) and that it is a *simplification* of the concrete system by omitting and encapsulating details

(e.g., the internal updates). We make the short remark that such specifications could be represented by a transition system. Or by a BUnity agent, as it is considered in [1].

Figure 3 illustrates the code of a BUpL agent implementing the above specification. Due to space limit we do not explain in detail its mechanism. The agent is designed such that it always builds a higher tower[4], thus the number of its mental states continuously increases.

$\mathcal{B}_0$ = { $on(1,0), length(1), max(0), clear(1), clear(0), done(0)$ }

$\mathcal{A}$ = { $move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z)$, { $on(x,z), \neg on(x,y), \neg clear(z)$ },)
$incLength(x)$ = ($length(x)$, { $\neg length(x), length(x+1)$ }),
$addBlock(x)$ = ($\neg on(x,0)$, { $on(x,0), clear(x)$ }),
$setMax(x,y)$ = ($max(y)$, { $\neg max(y), max(x)$ }),
$finish(x,y)$ = ($\neg done(x) \wedge done(y)$, { $\neg(done(y)), done(x)$ }) }

$\mathcal{P}$ = { $build(n,c) = move(c-n, 0, c-n-1); incLength(c-n-1); build(n-1,c)$
$generate(x,y)$ = $addBlock(x); generate(x-1,y)$,
$p_0(x,y) = setMax(x,y); generate(x,y))$ }

$\mathcal{R}$ = { $length(x) \wedge max(y) \wedge (x \leq y) \leftarrow build(y, y+x-1)$,
$length(x) \wedge max(x) \wedge done(y) \wedge (x \geq y) \leftarrow finish(x,y); \bot$,
$max(x) \wedge done(x) \leftarrow setMax(x+2,x), generate(x+2,x)$ }

Fig. 3: A BUpL Builder with Infinite State Space

For such an agent we can no longer verify correctness with respect to the specification by model-checking. Instead, we test it. We recall that our purpose is to test whether "bad" states are reachable from the initial configuration of the BUpL builder and that "bad" means odd length towers, in our case. Thus, if we consider testing whether $done(3)$ appears in the belief base after executing $move(2,0,1)$ followed by $move(3,0,2)$ we only need to apply the strategy $do(move(2,0,1)); do(move(3,0,2)); bel(done(3))$. For illustration purposes, the implementation of the agent is on purpose faulty (the correct agent tests the parity of x in the repair rule corresponding to executing the action $finish$). This means that the application of the strategy should result in a non empty state, meaning that the test case is successful. On the contrary, the application should be unsuccessful when the correct repair rule is used. From this we can conclude that the agent is safe with respect to the test.

We have already mentioned that the strategy language S has been incorporated into the Maude system. This made it possible for us to extend the implementation from [1] such that we can provide a testing framework as alternative to the model-checking facility. We have further experimented with different test cases which we applied to the Maude prototype corresponding to the BUpL builder from Figure 3. 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 $do(move(2,0,1)); do(move(3,0,2)); bel(done(3))$

[4] The example can be understood as a typical agent with *maintenance* goals

took 4ms for the BUpL builder and generated 24040 rewrites. We make the short note that the number of rewrites is high mainly because the strategy language is implemented at the meta-level and thus

```
Maude> (srew builder(3, 0) using
        do(move(2,0,1)); do(move(3,0,2)); bel(done(3)) .)
rewrites: 24040 in 3943ms cpu (4079ms real)
        (6096 rewrites/second)
rewrite with strategy :
result LBpMentalState :
        << iLabel('finish['s_^3['0.Zero],'s_['0.Zero]]),
        clear(0)# clear(3)# done(3)# length(3)# max(3)#
        on(1,0)# on(2,1)# on(3,2),i[bot,empty]>>
Maude> (next .)
rewrites: 1136 in 46ms cpu (81ms real)
        (24173 rewrites/second)
next solution rewriting with strategy :
No more solutions .
```

Fig. 4: One Test Execution for the BUpL Builder

From the Maude output illustrated in Figure 4, we can see that the strategy has succeeded and that the resulting state reflects that $done(3)$ has been updated to the belief base and that the current tower is 321. We can further see that there are no more solutions (corresponding to faulty executions) from the output of the default command `next`. 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-strat.php`.

5 Conclusions and Future Work

We have formalized testing and we have introduced strategies to implement test cases. Generalising our current results to multi-agent systems should be easy in a particular framework where the interaction between agents is achieved not by means of communication but by action-based coordination mechanisms. We already have some results with respect to implementing such coordination mechanisms by strategies and this is why we think the generalisation should be easy.

We did not address the issue of automatic model extraction. In our scenario, it was easy to infer the model from the specification, however, this is not always the case. Furthermore, one usually needs to validate the model with respect to the system under testing. Nor did we address the issue of automatic test case generation. Writing suitable test cases is not a trivial task, which would be alleviated by automatic test case generation. One trivial solution would be to consider all possible paths in the model. However, more refined approaches to reduce the paths to subsets of interesting ones are based on model-checking, theorem-proving or symbolic executions [10]. It is also the case that such techniques can be used in combination to better cover test case generation. These are subjects of future research. Also ongoing work are the techniques we have mentioned in the introduction, abstraction and reasoning about annotations.

References

1. Lacramioara Astefanoaei and Frank S. de Boer. Model-checking agent refinement. In Lin Padgham, David C. Parkes, Jörg Müller, and Simon Parsons, editors, *AAMAS*, pages 705–712. IFAAMAS, 2008.
2. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.
3. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
4. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
5. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
7. Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11):3–25, 2007.
8. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
10. Levi Lucio and Marko Samer. Technology of test-case generation. In Broy et al. [3], pages 323–354.
11. Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.
12. Bertrand Meyer. Seven Principles of Software Testing. *IEEE Computer*, 41(8):99–101, 2008.
13. Duy Cu Nguyen, Anna Perini, and Paolo Tonella. A Goal-Oriented Software Testing Methodology. In Michael Luck and Lin Padgham, editors, *AOSE*, volume 4951 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2007.
14. Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer. A rewriting logic approach to operational semantics (extended abstract). *Electronic Notes in Theoretical Computer Science*, 192(1):125–141, 2007.
15. M. Wooldridge. Agent-based software engineering. *IEEE Proceedings Software Engineering*, 144(1):26–37, 1997.
16. Zhiyong Zhang, John Thangarajah, and Lin Padgham. Automated unit testing intelligent agents in PDT. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1673–1674, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

Towards Just-In-Time Partial Evaluation of Prolog

Carl Friedrich Bolz, Michael Leuschel, 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 [22] 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 [24] 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 [26], 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.

Partial evaluation [16] 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 [16] 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 [19, 17] 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 [18]) 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

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

specialized code. This allows it to observe the runtime behaviour of the program, 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 built-in is encountered, in the general case the call is skipped, i.e. put in the residual code; but a number of common built-ins 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 (X in the figure) 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(X), false(X)).
negation(false(X), true(X)).
```

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. 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)` 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(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`, 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` 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` is called, it will use its first argument to decide which of the cases it should partially evaluate further.

Let us assume that `'$negation1'` is first called with `false(X)` as an argument. Then `'$promotion1'` will be executed, calling `callback(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:

```
'$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).

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 [17] (which builds upon the original work in [19]). The control of partial evaluation for logic programs is often separated into local

and global control [21], 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 built-in, then the built-in is specialized, yielding a single computed answer along with a specialized version of the built-in to be put into the residual code. For non-deterministic built-ins, the computed answer is general enough to cover all solutions. Failure can also sometimes be detected, in which case the branch is pruned.
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 built-ins 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 built-ins are met, using techniques from [23].

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 built-in 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.[3]

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 an 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).

[3] In some sense this can be seen as an evolution of the generalisation operator from [12] 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

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 Ecce [18], 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 [9]). 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.

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 results 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.

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 [24]). Similarly, the PyPy project [25, 4], in which all three authors are also involved, contains a just-in-time specialization system built on promotion [26].

Greg Sullivan describes a runtime partial evaluator for a small dynamic language based on lambda calculus [27]. Sullivan [27] further distinguishes two cases (quoting): *"Runtime partial evaluation [...] defers some of the partial evaluation*

	Experiment	Inferences	CPU Time	Speedup
A vanilla meta-interpreter [14, 20] 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

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 [8, 7]. 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 [13], 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) [15] 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 [11] (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 [10] 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) [5, 6].

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.

References

1. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35:1–12, 2000.
2. S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *Proceedings PEPM'04*, pages 190–199. ACM Press, 2004.
3. C. F. Bolz. *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. C. F. Bolz and A. Rigo. How to *not* write a virtual machine. In *Proceedings of 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.
5. M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. 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.
6. M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. 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 ExecutionEnvironments*, pages 71–80, Washington, DC, USA, 2009. ACM.
7. C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volansche. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 54–72. Springer, 1996.
8. C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *POPL*, pages 145–156, 1996.
9. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
10. A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.
11. A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.
12. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
13. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248:147–199, 2000.
14. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
15. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

16. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
17. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
18. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
19. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
20. B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.
21. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
22. M. Paleczny, C. Vick, and C. Click. The java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, Monterey, California, 2001. USENIX Association.
23. S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, pages 199–213, University of Manchester, 1992. Springer-Verlag.
24. A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In N. Heintze and P. Sestoft, editors, *PEPM*, pages 15–26. ACM, 2004.
25. A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.
26. A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
27. G. T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001.

Program Parallelization using Synchronized Pipelining

Leonardo Scandolo[1], César Kunz[1], Gilles Barthe[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 [11].

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 transforms. 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:

$$\texttt{while}\ b_1\ \texttt{do}\ c_1; \texttt{while}\ b_2\ \texttt{do}\ c_2$$

where c_2 (but not b_2) depends on c_1. The aim is to return a code where (modifications of) the two loops are executed in parallel, so that iterations of c_2 are executed as early as possible. 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. In summary, the main contributions of this paper are:

- the formal definition of eventual independence (Section 4),
- the definition and correctness proof of synchronized pipelining (Section 5), and
- experimental results that validate the benefits of synchronized pipelining (Section 6).

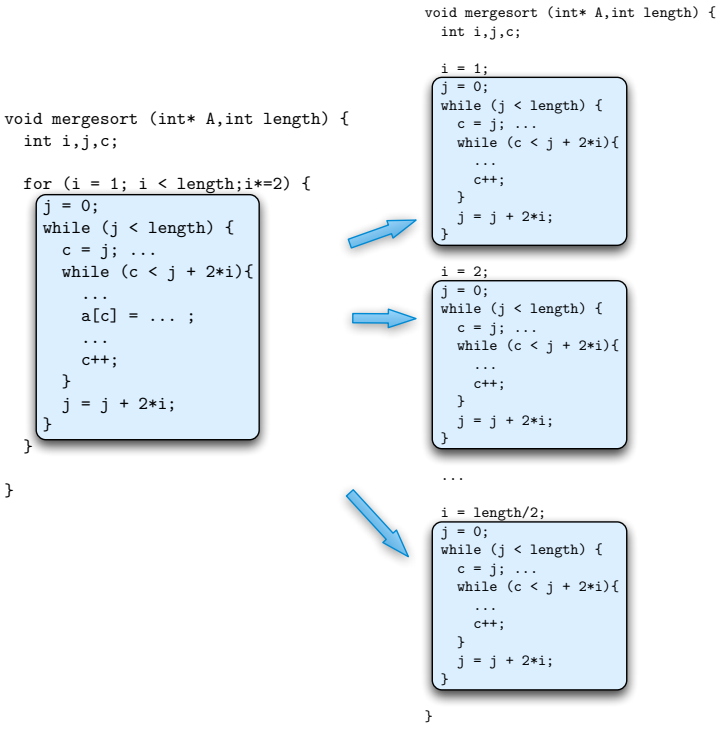


Fig. 1: Iterative mergesort algorithm

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 an iterative merge-sort 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`$(j < \texttt{length})\{\ldots\}$ accessing and modifying the array intervals $[0, 1]$, $[2, 3]$, ..., $[\texttt{length} - 1, \texttt{length}]$ in the first iteration, the intervals $[0, 3]$, $[4, 7]$, ..., $[\texttt{length} - 3, \texttt{length}]$ in the second iteration, and so on until the last iteration in which the intervals $[0, \texttt{length}/2]$ and $[\texttt{length}/2 + 1, \texttt{length}]$ are accessed.

One can clearly see that, from the common notion of data dependence, the first and second unrolled iteration cannot be executed in parallel, since they read and/or modify overlapping regions of the array. However, after partial completion of the first iteration, the second iteration can proceed without waiting for the first iteration to finish. For instance, in the sequential version, the iteration

that process the interval $[0, 3]$ waits for the first iteration to finish processing the interval $[\texttt{length} - 1, \texttt{length}]$. However, 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, 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.

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 $\tau \to c$ waits for the event τ to be announced before proceeding with the execution of c.

The semantics of programs 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 $\mathit{Stmt} \times \Sigma \times \mathcal{S}^\star$, where Stmt is the set of program statements, Σ is the set of states, i.e., mappings from program variables to integer values, and $\mathcal{S}^\star$ is the powerset of $\mathcal{S}$. Formally, the semantics is given by a small-step relation: $\rightsquigarrow \subseteq (\mathit{Stmt} \times \Sigma \times \mathcal{S}^\star) \times ((\mathit{Stmt} \times \Sigma \times \mathcal{S}^\star) + \{\mathsf{abort}\})$.

The transition rules for synchronization and parallel execution are given in Figure 2, 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 $(\texttt{x} := \texttt{5}; \tau!) \parallel \tau \to \texttt{x} := \texttt{1}$. Starting from a state where τ has not been announced, the execution terminates with the variable x holding the value 1, since $\texttt{x} := \texttt{1}$ 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.

$$\frac{}{\langle S!, \sigma, \epsilon\rangle \rightsquigarrow \langle \mathsf{nil}, \sigma, \epsilon \cup S\rangle} \qquad \frac{\tau \in \epsilon}{\langle \tau \rightarrow c, \sigma, \epsilon\rangle \rightsquigarrow \langle c, \sigma, \epsilon\rangle}$$

$$\frac{c \equiv d \qquad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \langle d', \sigma', \epsilon'\rangle \qquad d' \equiv c'}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle} \qquad \frac{c \equiv d \qquad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$\frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \langle c' \parallel d, \sigma', \epsilon'\rangle} \qquad \frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$i \parallel \mathsf{nil} \equiv i \qquad i \parallel j \equiv j \parallel i \qquad i \parallel (j \parallel k) \equiv (i \parallel j) \parallel k$$

Fig. 2: Operational semantics (excerpts)

Definition 1 (Semantical Equivalence). *Let $c_1, c_2 \in Stmt$ 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 loops, 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 $m, n, k \in \mathbb{N}, \epsilon \subseteq \mathcal{S}$ s.t. $(n, \lambda(m)) \in \Omega$, $\sigma \in \Sigma$ and no synchronization variables in ϵ appear in c_1 or c_2: $[\![c_1^n; c_2^{m-1}; c_1^k; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^n; c_2^{m-1}; (c_1^k \parallel c_2)]\!]$ where c^i stands for the sequential composition of i instances of the statement c. Given Ω and $n \in \mathbb{N}$, we let $\omega(n) = \{s \mid (n, s) \in \Omega\}$.*

Notice that when considering sequential code, it is sufficient to state the semantics equivalence in terms of the empty event set. If $\lambda(m) = s$, then the m^{th} iteration 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 c_2 to proceed. Indeed, by definition of Ω, it follows from $(n, s) \in \Omega$ that after n iterations of c_1, all subsequent iterations of c_1 never write again on a piece of memory on which the m^{th} iteration of c_2 depends.

Perhaps surprisingly, independent loops need not be eventually independent. Consider the following program

$$(\texttt{i} := 0;\ \texttt{while i} < 5\ \texttt{do a[i]} := \texttt{a[i]} + 1);\ (\texttt{j} := 5;\ \texttt{while j} < 10\ \texttt{do a[j]} := \texttt{a[j]} + 1)$$

The two statements are independent, but the bodies of the two loops are not eventually independent.

It is possible to resolve the discrepancy by refining eventual independence so that it considers a set of initial states (restricting the scope of σ in Definition 3) and, when they exist, bounds on the number of iterations performed by the loops (restricting the scope of m, n, k in Definition 3). To avoid cluttering the exposition, we stick to our simpler notion of eventual independence; however, the definitions and correctness proof extend readily to these more refined notions.

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 $\vec{e}_1$ and $\vec{e}_2$ 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 [7] 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. If h'_i is derived from h''_i, which is an induction variable derived from h_i, then f_i is a polynomial function. More complex cases may arise, for instance when f_i is defined as a geometric function on h_i.

In most of the algorithms that we have considered as the target of the transformation, f_1 and f_2 are defined as linear functions. One can find, although less frequently, cases in which f_1 and f_2 are polynomial functions. In those cases, the expressions $\vec{e}_1(h'_1)$ and $\vec{e}_2(h'_1)$ are easily rewritten in terms of the inductive variables, i.e., as $\vec{e}_1(f_1(h_1))$ and $\vec{e}_2(f_2(h_2))$. By static interval analysis, we can approximate the regions of data that is read and modified by c_1 and c_2, in terms of the induction variables h_1 and h_2, and the expressions $\vec{e}_1(f_1(h_1))$

and $\vec{e}_2(f_2(h_2))$. Assume $[d_1^{rw}, e_1^{rw}]$ represents the interval of the array $\mathtt{a}$ that is written or read by c_1, where d_1^{rw}, e_1^{rw} are integer expressions that depend on h_1 (and similarly with c_2). Since $\vec{e}(f_1(h_1))$ and $\vec{e}(f_2(h_2))$ 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 d and e expressions are increasing as the h variables grow (the decreasing case is symmetrical) one can propose an eventual independence relation Ω. For instance when d_1^{rw} and e_2^{rw} are increasing functions, we determine the values for h_1 and h_2 such that $e_2^{rw} < d_1^{rw}$, and then, since the h_2-th iteration of c_2 is independent of the h_1-th iteration of c_1, we can have $(h_1, \lambda(h_2)) \in \Omega$.

Example 2. We show in this paragraph how to determine an eventual independence relation for two particular loops statements. Suppose the loop statements are defined, respectively, by the loop bodies c_1 and c_2, defined as

$$\begin{array}{lcl} c_1 & \doteq & \mathtt{a[x]} := \mathtt{1}; \mathtt{x} := \mathtt{x} + \mathtt{1} \\ c_2 & \doteq & \mathtt{y} := \mathtt{y} + \mathtt{a[z]}; \mathtt{z} := \mathtt{z} + \mathtt{1} \end{array}$$

First of all, notice that statements c_1 and c_2 access the array $\mathtt{a}$, so they are not independent. By examining the 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 we have $d_1^{rw}(h_1) = e_1^{rw}(h_1) = h_1 + x^\star$ and $d_2^{rw}(h_2) = e_2^{rw}(h_2) = h_2 + z^\star$ so 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 as long as the statement c_1 has been executed $x^\star - y^\star + 1$ more times than the statement c_2, the next execution of c_2 is independent of any further execution 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:

$$\begin{array}{l} x + x^\star \leq |\mathtt{a}| \wedge y < x - (z^\star - x^\star + 1) \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \\ x + x^\star > |\mathtt{a}| \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \end{array}$$

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. We then briefly discuss how the method applies to recursive procedures.

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 $\texttt{while}\ b_1\ \texttt{do}\ c_1' \parallel \texttt{while}\ b_2\ \texttt{do}\ c_2'$, 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 then 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 preservation. 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. Finally, we consider recursive procedures.

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 $\lambda_{i,j} : \mathbb{N} \to \mathcal{S}$. 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:

$$\texttt{while } b_i \texttt{ do } \bigcup_{1 \le j < i} \lambda_{i,j}(h) \to \left(c_i; \bigcup_{i < j \le n} \omega_{i,j}(h)! \right); \forall_{i<j\le n} S_{i,j}!$$

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` $(c_1;$ `while` b `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 $l \le \beta$, 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 on these instructions to 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:

$$\begin{array}{l} \texttt{while } a \texttt{ do} \quad \tau_{c,l-1} \to (\hat{c_1}[l]; \tau_{c_1,l}!)\,; \\ \quad \texttt{while } \hat{b}[l] \texttt{ do } \left(\bigcup_{1 \leq j < l} \lambda_{l,j}(h) \to \left(\hat{c}[l]; \bigcup_{l < j \leq \beta} \omega_{l,j}(h)!; \hat{c_2}[h']\right)\right); \\ \texttt{sync} \end{array}$$

Notice that the order in which the instances of $\hat{c}[l]$ are executed is preserved.

Recursive Procedures. We can extend the parallelization transformation applied to nested loops to recursively defined functions containing loop statements.

Automatic transformation of recursive functions into iterative loops, i.e., recursion removal, is straightforward if functions are tail recursive. In our programming language, a function is tail recursive if no extra computation is performed after a recursive function call returns. For instance, the mergesort algorithm shown in Section 2, is an iterative variant of the more typical mergesort algorithm defined in terms of mutually recursive procedures.

Extending this automatic transformation to general recursive procedures is challenging, and has been widely studied [16] as a program optimization since it enables to reduce the overhead of call stack manipulation.

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 parametrize 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 *Cilk* procedure, 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.

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[3] $\mathtt{c}^{\mathtt{i}}$ is bigger than that of $\mathtt{r}^{\mathtt{i}+1}$.

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++;
      }
...
```

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 $\mathtt{right}_{\mathtt{m}}^{\mathtt{i}}$.

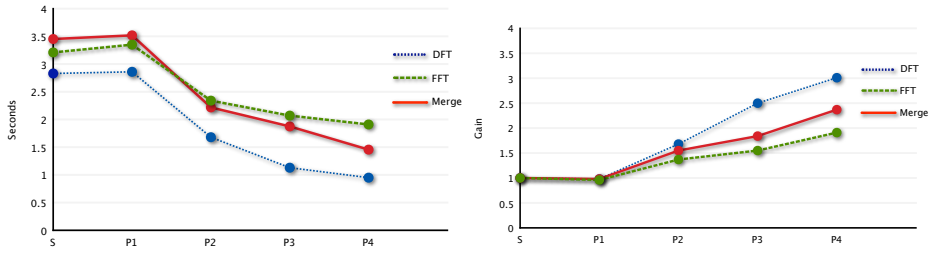
We now need to determine λ and Ω for every other possible combination of iterations. But since the same loop is repeated, with the same properties, we require the same condition to advance, namely $\mathtt{r}_{\mathtt{m}}^{\mathtt{i}} < \mathtt{c}_{\mathtt{n}}^{\mathtt{j}}$, and thus $\Omega_{i,j}$ again 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* [8] program. *Cilk* is an extension of *C* for multithreaded parallel programming, that provides a lightweight 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.

[3] 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. 3: Experimental Results

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 3 shows the computing time and the relative performance gain of the DFT, FFT, and MergeSort algorithms run under the different conditions we have explained. The pipelined version of our DFT program is slightly slower while 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 bench-

marks 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. [19] 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 [15] 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 [21] (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 [6]. 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 [25] 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 extending 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., [22, 23, 12, 9, 18]).

References

1. V. Allan, R. Jones, R. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.
2. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Tex., January 1989.
3. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.
4. T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, PDS-11(11):1105–1125, November 2000.
5. L. Flon and N. Suzuki. Consistent and complete proof rules for the total correctness of parallel programs. In *19th Annual Symposium on Foundations of Computer Science (FOCS '78)*, pages 184–192, Long Beach, Ca., USA, October 1978. IEEE Computer Society Press.
6. M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. In J. Hook and P. Thiemann, editors, *ICFP*, pages 119–130. ACM, 2008.
7. M. Gerlek, E. Stoltz, and M. Wolfe. 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.
8. Supercomputing Technologies Group. Cilk 5.4.6 reference manual, 1998.

9. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
10. M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
11. J. Hennessy and D. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufman, 2003.
12. J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, 1994.
13. K. Inenaga, S. Kusakabe, and M. Amamiya. Producer-consumer pipelining for structured-data in a fine-grain non-strict dataflow language on commodity machines. In *IWIA '99: Proceedings of the 1999 International Workshop on Innovative Architecture*, page 77, Washington, DC, USA, 1999. IEEE Computer Society.
14. M. Joyner, Z. Budimlic, and V. Sarkar. Optimizing array accesses in high productivity languages. In Ronald H. Perrott, Barbara M. Chapman, Jaspal Subhlok, Rodrigo Fernandes de Mello, and Laurence Tianruo Yang, editors, *High Performance Computing and Communications, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782 of *LNCS*, pages 432–445. Springer, 2007.
15. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In J. Ferrante and K. McKinley, editors, *PLDI*, pages 211–222. ACM, 2007.
16. Y. Liu and S. Stoller. From recursion to iteration: What are the optimizations? In *PEPM*, pages 73–82, 2000.
17. G. Lueker. Some techniques for solving recurrences. *CSURV: Computing Surveys*, 1980.
18. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models. In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.
19. Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, pages 105–118. IEEE Computer Society, 2005.
20. P. Paczkowski. Proving total correctness of concurrent programs without auxiliary variables. Technical Report ECS-LFCS-89-100, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, November 1989.
21. Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.
22. M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, 2009. To appear.
23. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, 1999.
24. J. Subhlok and K. Kennedy. Integer programming for array subscript analysis. *IEEE Transactions on Parallel and Distributed Systems*, PDS-6(6):662–668, June 1995.
25. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In R. Gupta and S. Amarasinghe, editors, *PLDI*, pages 125–135. ACM, 2008.

Safety Preserving Transformations for General Answer Set Programs

Pedro Cabalar[1*], David Pearce[2*], and Agustín Valverde[3⋆]

[1] Universidade da Coruña
cabalar@udc.es
[2] Universidad Politécnica de Madrid, Spain
david.pearce@upm.es
[3] Universidad de Málaga, Málaga, Spain
a_valverde@ctima.uma.es

Abstract. Many answer set solvers deal with programs with variables by requiring a safety condition on rules: any variable in a rule must appear in its positive body. This idea of safety has recently been extended to cover more general kinds of rules or first-order formulas that might be accepted by existing or future generation ASP systems [4, 15, 7]. In this paper we continue the study of the generalised safety concept recently proposed in [7]. In particular, we show that safety is preserved under a major subset of the transformations that reduce universal theories to disjunctive rules in ASP.

1 Introduction

This paper is concerned with some program transformations in the declarative programming framework known as *answer set programming* (ASP). Specifically, we examine the extent to which general first order formulas that are *safe* in ASP remain safe when transformed into sets of disjunctive program rules. Our concept of safety is the one recently defined in [7, 8], while program transformations are taken from [5] where it is shown how in ASP any first order formula can be reduced to a strongly equivalent logic program of a general form.[4] Our main result is that for universal formulas safety is preserved under all but one of the program transformations.

1.1 Extensions of answer set semantics

Answer set programming has become established as a vibrant new sub-field of logic programming and knowledge representation. There are now several rival implementations of ASP, many different kinds of language extensions, and a growing catalogue of

⋆ Partially supported by the MEC (now MICINN) projects TIN2006-15455-(C01,C02,C03) and CSD2007-00022, Junta de Andalucia project P6-FQM-02049 and Xunta de Galicia project INCITE08-PXIB105159PR.

[4] Here we use a slight generalisation and improvement of the safety concept that was defined in [7] and in the first version of [8]. This improvement was subsequently incorporated into the final version of [8].

practical applications.[5] While ASP systems continue to eliminate variables from programs by means of a grounding process, there is currently much interest in issues involving first order languages and programs. One important line of work in this direction concerns extending the basic language of disjunctive programs to embrace more general kinds of first order formulas.

Answer set semantics can be defined for general logical formulas by regarding answer sets or stable models as minimal models in a non-classical logic called *here-and-there*. This was shown for propositional theories in [19] and for first order theories in [23–25]. Subsequently, equivalent characterisations of answer sets using alternative logical frameworks were provided by [18, 11] in the propositional case and in [12] for first order logic. However the here-and-there approach to answer set semantics remains in our view the most natural and intuitive one. A key point in its favour is that here-and-there logic precisely captures the robust notion of strong equivalence for theories or programs under answer set semantics [16, 17]. That is to say, under answer set semantics one theory can be replaced by another in any context without loss if and only if the theories are equivalent in the logic of here-and-there. We denote this logic by $\mathbf{HT}$ in the propositional and by $\mathbf{QHT}$ in the quantified, first order case.

Besides ordinary disjunctive rules and general first order formulas, certain intermediate classes of formulas are also of special interest in ASP. Examples are general disjunctive rules where negation '$\neg$' is allowed in the heads as well as the bodies of rules, and rules with nested expressions where the rule body and head can be any compound expression involving $\wedge, \vee, \neg$ [18]. Recently [4] has studied a syntactically restricted subclass of the latter programs, called *normal form nested* or NFN programs.

Following these extensions of answer set semantics to more general syntactic classes of formulas, one further line of research in ASP has been to study program transformations that reduce a program from a more expressive syntactic class to one belonging to a simpler class. [18] already showed how nested programs could be transformed into equivalent general disjunctive programs. [6] later showed that any propositional theory is strongly equivalent to a general disjunctive program in the same vocabulary, while [5] provided a complete set of transformations that effectively carries out this reduction.

In the first order case, the situation is briefly described as follows. As usual a first order sentence is said to be in *prenex* form if it has the following shape, for some $n \geq 0$:

$$Q_1 x_1 \ldots Q_n x_n \psi \tag{1}$$

where Q_i is $\forall$ or $\exists$ and ψ is quantifier-free. A sentence is said to be *universal* if it is in prenex form and all quantifiers are universal. A universal theory is a set of universal sentences. In [24] it is shown that in the logic $\mathbf{QHT}$ every sentence is logically equivalent to a sentence in prenex form. Without loss of generality we can therefore focus on sentences in prenex form. Since the matrix ψ in a prenex form is quantifier-free, we can apply equivalences from propositional logic to convert ψ into a special reduced form using the transformations described in [5]. They allow us to convert ψ into a logically equivalent general disjunctive rule. In this paper we shall focus on universal theories

[5] The recent LPNMR conferences provide a good source of references, eg [3, 2].

so that the transformations are all of a type that reduce (1) to a logic program of this general type.[6]

1.2 Safe formulas

The aim of this paper is to re-examine the transformations described in [5] from the point of view of *safety*. This fundamental concept in ASP is applied to rules of ordinary logic programs in the following way. A rule is said to be *safe* if each variable in it appears in the positive body of the rule. Many ASP implementations impose this condition by accepting only safe rules. There are three main properties of safe rules that we should distinguish. The first is that the answer sets of safe rules do not contain unnamed individuals. This condition is already fulfilled by formulas that we call *semi-safe*. Secondly there is the property usually called *domain independence* which says that grounding a program with respect to any superset of the program's constants will not change the class of answer sets. The third property satisfied by safe formulas is that the collection of their answer sets is first order definable. Like the other properties, this one is relevant for computational purposes, being exploited for instance by the method of loop formulas.

Recently, the concept of safety has been extended to more general formulas, for example to NFN programs in [4] and to arbitrary first-order formulas in [15]. Our own approach also covers arbitrary formulas and is described in [7, 8]. It generalises the safety concept from [15] by re-classifying some kinds of formulas as safe that are unsafe according to [15]. At the same time, our concept still satisfies the three mentioned desiderata for safe formulas.

It is important to notice that safety is defined at the level of single formulas and is an inherently syntactical condition. It is therefore unreasonable to expect that the safety of a formula will be transferred to arbitrary equivalent formulas. In particular safety may be gained or lost by removing or adding some redundant subformulas. For instance, a rule like $p \to q(x)$ is unsafe and, in principle, may easily have different stable models depending on the domain we use for grounding (just add a fact p). However, if it is included in any program containing a constraint like $p \to \bot$, the unsafe rule becomes irrelevant. Similarly, with nested expressions, any safe rule $F \to G$ may become unsafe by a simple addition of a $\mathbf{QHT}$ tautology or inconsistency, as in $F \to G \vee (p(x) \wedge \neg p(x))$.

On the other hand, if we start with a general expression that is safe and apply certain kinds of logical re-writing steps such as those used in [5] to simplify formulas and reduce them to sets of general disjunctive rules, it might be reasonable to expect that an adequate concept of safety should be preserved under the transformations. In other words, while we cannot replace a safe formula by any arbitrary formula logically equivalent to it without losing safety, we can transform it into a possibly simpler expression while still maintaining safety. This is the problem that we shall study in the remainder of the paper.

[6] We postpone to future work the study of transformations that apply to an arbitrary prenex sentence or other kind of sentences involving existential quantifiers.

The main result we establish is that when applied to universal sentences all but one of the transformation rules from [5] preserves safety. This means that a large class of safe first order formulas can be converted into strongly equivalent general disjunctive programs each of whose rules is safe. This collection includes the important class of all programs with nested expressions. While studying this problem we also found a slight generalisation of the safety concept from [7] which we have applied here and in [8]. The usual properties of safe formulas remain true for this revised concept.

2 Logical Background

Usually in quantified equilibrium logic we consider a full first-order language allowing function symbols and we include a second, strong negation operator as occurs in several ASP dialects. However, in this paper we restrict attention to function-free first order languages $\mathcal{L} = \langle C, P\rangle$ built over a set of *constant* symbols, C, and a set of *predicate* symbols, P.[7] We assume a single negation symbol, '$\neg$', together with the usual connectives and quantifiers, $\wedge, \vee, \rightarrow, \exists, \forall$. We shall also assume that $\mathcal{L}$ contains the constants $\top$ and $\bot$ and, where convenient, we regard $\neg\varphi$ as an abbreviation for $\varphi \rightarrow \bot$. In other respects we follow the treatment of [25]. The sets of $\mathcal{L}$-formulas, $\mathcal{L}$-sentences and atomic $\mathcal{L}$-sentences are defined in the usual way. The set of (free) variables of a formula φ will be denoted as $\mathrm{VARS}(\varphi)$.

We work in a non-classical logic called *Quantified Here-and-There Logic with static domains and decidable equality*. For reasons of space we give here just a short summary. A complete axiomatisation and more detailed description of this logic can be found in [17] where the logic is denoted by $\mathbf{SQHT}^{=}$. In terms of satisfiability and validity this logic is equivalent to the logic previously introduced in [24]. To simplify notation we drop the labels for static domains and equality and refer to this logic simply as quantified here-and-there, $\mathbf{QHT}$.

The semantics of $\mathbf{QHT}$ is given in terms of intuitionistic Kripke models, see [10], with two notable exceptions. One concerns equality: we regard equality as decidable and as satisfying the axiom $\forall x \forall y((x = y) \vee \neg(x = y))$. Furthermore, we suppose a logic with constant or static domains; in other words, within a given Kripke model the same set of individuals populates each world. In addition, $\mathbf{QHT}$ is complete for very simple Kripke models, those possessing just two worlds, sometimes labelled h ("here") and t ("there"), ordered by $h \leq t$.

We use the following notation. If D is a non-empty set, we denote by $At(D, P)$ the set of ground atomic sentences in the language $\langle D, P\rangle$. By an $\mathcal{L}$-interpretation I over a set D we mean a subset of $At(D, P)$. A $\mathbf{QHT}(\mathcal{L})$-*structure* can therefore be regarded as a tuple $\mathcal{M} = \langle (D, \sigma), I_h, I_t\rangle$. where I_h, I_t are $\mathcal{L}$-interpretations over D such that $I_h \subseteq I_t$ and $\sigma\colon C \cup D \rightarrow D$ is a mapping, called the *assignment*, such that $\sigma(d) = d$ for all $d \in D$. Evidently, $\langle (D, \sigma), I_h\rangle$ and $\langle (D, \sigma), I_t\rangle$ are classical $\mathcal{L}$-structures. Given an interpretation we let $\sigma|_C$ denote the restriction of the assignment σ to constants in C.

[7] Not that in ASP the restriction to function-free languages is standard. The study of functions in the framework of ASP is recent and still largely theoretical.

Truth of a sentence in a model is defined as follows: $\mathcal{M} \models \varphi$ iff $\mathcal{M}, w \models \varphi$ for each $w \in \{h, t\}$. In a model $\mathcal{M}$ we also use the symbols H and T, possibly with subscripts, to denote the interpretations I_h and I_t, respectively; so, an $\mathcal{L}$-structure may be written in the form $\langle U, H, T\rangle$, where $U = (D, \sigma)$. A structure $\langle U, H, T\rangle$ is called *total* if $H = T$, hence it is equivalent to a classical structure.

An answer set semantics for arbitrary first-order formulas can be defined using the quantified variant of equilibrium logic [19, 20] that we denote by **QEL**. As in the propositional case, this is based on a suitable notion of minimal model as follows.

Definition 1 ([23, 24]). *Let Γ be a set of $\mathcal{L}$-sentences. An* equilibrium *model or* answer set *of Γ is a total model $\mathcal{M} = \langle (D, \sigma), T, T\rangle$ of Γ such that there is no model of Γ of the form $\langle (D, \sigma), H, T\rangle$ where H is a proper subset of T.*

An equivalent characterisation of stable model or answer set for a finite set of first-order formulas is given in [12].

The study of strong equivalence for logic programs and nonmonotonic theories was initiated in [16]. It has since become an important tool in ASP as a basis for program transformation and optimisation. In equilibrium logic we say that two (first-order) theories Π_1 and Π_2 are strongly equivalent if and only if for any theory Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same equilibrium models [17, 25]. Under this definition we have:

Theorem 1 ([17, 25]). *Two (first-order) theories Π_1 and Π_2 are strongly equivalent if and only if they are equivalent in* **QHT**.

Below we shall treat reductions that transform a formula into a logically equivalent set of formulas. These transformations therefore preserve strong equivalence.

3 Review of the Safety Concept

We use the same concept of restricted variable as in [14, 15]. To every quantifier-free formula φ the set $\mathrm{RV}(\varphi)$ of its *restricted variables* is defined as follows:

- If φ is a non-equality atom: $\mathrm{RV}(\varphi) = \mathrm{VARS}(\varphi)$;
- $\mathrm{RV}(\varphi_1 \wedge \varphi_2) = \mathrm{RV}(\varphi_1) \cup \mathrm{RV}(\varphi_2)$;
- $\mathrm{RV}(\varphi_1 \vee \varphi_2) = \mathrm{RV}(\varphi_1) \cap \mathrm{RV}(\varphi_2)$;
- For the rest of cases: $\mathrm{RV}(\bot) = \mathrm{RV}(\varphi_1 \to \varphi_2) = \mathrm{RV}(t_1 = t_2) = \varnothing$.

As the following Definition 2 indicates, the set $\mathrm{RV}(\varphi)$ comes into play when φ is an antecedent of an implication (i.e., a rule body); it collects each variable x whose extent can be directly obtained by examining positive occurrences of predicates containing x.

As in [15], we define a concept of *semi-safety* of a prenex form sentence φ in terms of the *semi-safety* of all its variable occurrences.[8] Formally, this is done by defining an operator NSS that collects the variables that have *non-semi-safe* occurrences in a formula φ.

[8] Notice that while we study the effect of program transformations on *universal* sentences, safety and semi-safety are actually defined for arbitrary prenex sentences, so we give the general definition here.

Definition 2 (NSS **and semi-safety).**

1. *If* φ *is an atom,* $\mathrm{NSS}(\varphi) = \mathrm{VARS}(\varphi)$.
2. $\mathrm{NSS}(\bot) = \varnothing$.
3. $\mathrm{NSS}(\varphi_1 \wedge \varphi_2) = \mathrm{NSS}(\varphi_1 \vee \varphi_2) = \mathrm{NSS}(\varphi_2) \cup \mathrm{NSS}(\varphi_1)$.
4. $\mathrm{NSS}(\varphi_1 \to \varphi_2) = \mathrm{NSS}(\varphi_2) \setminus \mathrm{RV}(\varphi_1)$.

A sentence φ *is said to be* semi-safe *if* $\mathrm{NSS}(\varphi) = \varnothing$.

In other words, a variable x is semi-safe in φ if every occurrence is inside some subformula $\alpha \to \beta$ such that, either x is restricted in α or x is semi-safe in β. Note that any negated formula is semi-safe, because $\mathrm{NSS}(\neg\varphi) = \mathrm{NSS}(\varphi \to \bot) = \varnothing$.

Example 1. Suppose that a process y will ignore any request of an item z from another process x, unless x is a subprocess of y that does not have item z. When the request is ignored, we further want to assert that x becomes unattended. We can represent this behaviour by the following rule with nested expressions:

$$request(x, y, z) \wedge \neg(subproc(x, y) \wedge \neg has(y, z)) \to ignore(y, x) \wedge unatt(x) \quad (2)$$

The formula (2) is semi-safe: all variables x, y and z occur in an implication (the main one) whose variables are restricted in the antecedent, $\mathrm{RV}(request(x, y, z) \wedge \neg(subproc(x, y) \wedge \neg has(y, z))) = \mathrm{RV}(request(x, y, z)) = \{x, y, z\}$.

The following results establish the main property of semi-safe formulas: their equilibrium models only refer to constants in the original language.

Proposition 1 ([7, 8]). *If* φ *is semi-safe, and* $\langle (D, \sigma), T, T\rangle \models \varphi$*, then* $\langle (D, \sigma), T|_C, T\rangle \models \varphi$.

Theorem 2 ([7, 8]). *If* φ *is semi-safe, and* $\langle (D, \sigma), T, T\rangle$ *is an equilibrium model of* φ*, then* $T|_C = T$.

The concept of safety relies on semi-safety plus an additional condition on variable occurrences. As a technical device we can define this condition using Kleene's three-valued logic [13]. Given a three-valued interpretation $\nu\colon Atoms \to \{0, 1/2, 1\}$, we extend it to evaluate arbitrary formulas $\nu(\varphi)$ as follows:

$$\begin{array}{ll} \nu(\varphi \wedge \psi) = \min(\nu(\varphi), \nu(\psi)) & \nu(\bot) = 0 \\ \nu(\varphi \vee \psi) = \max(\nu(\varphi), \nu(\psi)) & \nu(\varphi \to \psi) = \max(1 - \nu(\varphi), \nu(\psi)) \end{array}$$

from which we can derive $\nu(\neg\varphi) = \nu(\varphi \to \bot) = 1 - \nu(\varphi)$ and $\nu(\top) = \nu(\neg\bot) = 1$.

Definition 3 (ν_x **operator).** *Given any quantifier-free formula* φ *and any variable* x*, we define the three-valued interpretation so that for any atom* α*,* $\nu_x(\alpha) = 0$ *if* x *occurs in* α *and* $\nu_x(\alpha) = 1/2$ *otherwise.*

Intuitively, $\nu_x(\varphi)$ fixes all atoms containing the variable x to 0 (falsity) leaving all the rest undefined and then evaluates φ using Kleene's three-valued operators, that is nothing else but exploiting the defined values 1 (true) and 0 (false) as much as possible. For instance, $\nu_x(p(x) \to q(x))$ would informally correspond to $\nu_x(0 \to 0) = \max(1 - 0, 0) = 1$ whereas $\nu_x(p(x) \vee r(y) \to q(x)) = \nu_x(0 \vee 1/2 \to 0) = \max(1 - \max(0, 1/2), 0) = 1/2$.

Definition 4 (Weakly-restricted variable). *An occurrence of a variable x in $Qx\ \varphi$ is* weakly-restricted *if it occurs in a subformula ψ of φ such that:*

- $Q = \forall$, *ψ is positive and* $\nu_x(\psi) = 1$
- $Q = \forall$, *ψ is negative and* $\nu_x(\psi) = 0$
- $Q = \exists$, *ψ is positive and* $\nu_x(\psi) = 0$
- $Q = \exists$, *ψ is negative and* $\nu_x(\psi) = 1$

In all cases, we say additionally that ψ makes the ocurrence weakly restricted in φ.

Definition 5 (safety). *A semi-safe sentence is said to be* safe *if all its positive occurrences of universally quantified variables, and all its negative occurrences of existentially quantified variables are weakly restricted.*

For instance, notice that (2) introduced in Example 1 is safe. All variables are universally quantified and all (positive) occurrences of x, y and z occur in a positive subformula, (2) itself, for which $\nu_x((2)) = \nu_y((2)) = \nu_z((2)) = 1$.

Theorem 3 establishes the main property of safe formulas. The grounding over C of a sentence φ, denoted by $\mathrm{Gr}_C(\varphi)$, is defined recursively: the operator does not modify ground formulas, commutes with propositional connectives and

$$\mathrm{Gr}_C(\forall x\varphi(x)) = \bigwedge_{c\in C} \mathrm{Gr}_C\varphi(c) \qquad \mathrm{Gr}_C(\exists x\varphi(x)) = \bigvee_{c\in C} \mathrm{Gr}_C\varphi(c)$$

Theorem 3 ([7, 8]). *Let φ be a safe prenex formula, then: $\langle (D,\sigma), T, T\rangle$ is an equilibrium model of φ if and only if it is an equilibrium model of* $\mathrm{Gr}_C(\varphi)$.

Notice that, although in [7] Theorems 2 and 3 were established under a slightly different safety concept, it is easy to see that they continue to hold for the revised concept used here.

4 Negation Normal Form

The transformations introduced in [5] are top-down processes that rely on the successive application of several rewriting rules that operate on sets (conjunctions) of implications. A rewriting takes place whenever one of those implications does not yet have the form of a (non-nested) program rule.

Two sets of transformations are described next. A formula is said to be in *negation normal form* (NNF) when negation is only applied to literals. As a first step, we describe a set of rules that move negations inwards until a NNF is obtained:

$$\neg\top \Longleftrightarrow \bot \quad \text{(N1)} \qquad \neg\bot \Longleftrightarrow \top \quad \text{(N2)}$$
$$\neg\neg\neg\alpha \Longleftrightarrow \neg\alpha \quad \text{(N3)} \qquad \neg(\alpha\wedge\beta) \Longleftrightarrow \neg\alpha\vee\neg\beta \quad \text{(N4)}$$
$$\neg(\alpha\vee\beta) \Longleftrightarrow \neg\alpha\wedge\neg\beta \quad \text{(N5)} \qquad \neg(\alpha\rightarrow\beta) \Longleftrightarrow \neg\neg\alpha\wedge\neg\beta \quad \text{(N6)}$$

Lemma 1. *For any instance of γ and γ' in any pair $\gamma \Longleftrightarrow \gamma'$ in transformations (N1)-(N6) we have that* $\mathrm{NSS}(\gamma) = \mathrm{NSS}(\gamma')$ *and* $\mathrm{RV}(\gamma) = \mathrm{RV}(\gamma')$.

Proof. Both properties are trivial, because for every transformation the application of the operators on both sides returns the empty set. □

By an inductive application of this lemma, we immediatelly conclude that NNF transformations preserve the semi-safe property, as stated below:

Proposition 2. *For any sentence φ and for every pair $\gamma \iff \gamma'$ in transformations (N1)-(N6) we have that* $\mathrm{NSS}(\varphi) = \mathrm{NSS}(\varphi[\gamma/\gamma'])$.

So, if φ' is an NNF formula obtained from φ by the application of the rules (N1)-(N6), then φ' is semi-safe if and only if φ is semi-safe. We prove now that the NNF conversion also preserves safety. To this aim, we first provide a pair of properties.

Observation 1 *For any pair $\gamma \iff \gamma'$ in transformations (N1)-(N6), if ψ is a subformula of α or β, then the sign of ψ in γ is equal to the sign in γ'.* □

Lemma 2. *For any pair $\gamma \iff \gamma'$ in transformations (N1)-(N6) we have $\nu_x(\gamma) = \nu_x(\gamma')$ and thus, $\nu_x(\psi) = \nu_x(\psi[\gamma/\gamma'])$ for any formula ψ.*

Proof. It can be easily checked that, for each pair, $\gamma \iff \gamma'$, formulas γ and γ' are semantically equivalent in Kleene's three-valued logic, that is $\nu(\gamma) = \nu(\gamma')$ for any three-valued interpretation ν. □

Theorem 4. *Consider a semi-safe universal sentence $\forall x_1 \ldots \forall x_n \varphi$ and any pair $\gamma \iff \gamma'$ in transformations (N1)-(N6) such that γ is a subformula of φ. The following hold; (i) if x_i is safe in φ then it is also safe in $\varphi[\gamma/\gamma']$; (ii) therefore, if φ is safe and φ' is an NNF formula obtained from φ by applying the transformations (N1)-(N6), then φ' is also safe.*

Proof. To prove the result, we must analyse every occurrence of every variable in $\varphi[\gamma/\gamma']$ to check if it is made weakly-restricted. It is important to note that each one of these occurrences corresponds in a natural way to a specific occurrence of the same variable in the formula φ, because the transformations do not modify either the number or the relative situation of the variables. Note also that, by Observation 1, the transformation does not modify the sign of the variable occurrences. We proceed with the proof, distinguishing several cases.

Since the formula is universal, let us consider a positive occurrence of x_i and ψ the subformula of φ making the ocurrence weakly-restricted.

- If the occurrence is outside γ and γ is not a subformula of ψ, then ψ directly makes the corresponding ocurrence weakly-restricted in $\varphi[\gamma/\gamma']$.
- If γ is a subformula of ψ, then $\psi[\gamma/\gamma']$ makes the ocurrence weakly-restricted in $\varphi[\gamma/\gamma']$, because $\nu_x(\psi) = \nu_x(\psi[\gamma/\gamma'])$ (by Lemma 2) and the sign of ψ in φ is equal to the sign of $\psi[\gamma/\gamma']$ in $\varphi[\gamma/\gamma']$.
- If ψ is a strict subformula of γ, then we need to analyse every rule. For (N1), (N2) and (N3) the result is trivial. For rules (N4), (N5) and (N6), if ψ is subformula of α or β, then ψ makes the ocurrence weakly restricted in $\varphi[\gamma/\gamma']$, because the rule preserves the sign of subformulas of α and β.

- If $\psi = \alpha \wedge \beta$ then $\psi' = \neg\alpha \vee \neg\beta$ makes the corresponding ocurrence weakly restricted in $\varphi[\gamma/\gamma']$, because the sign of ψ' is the opposite of the sign of ψ and $\nu_x(\neg\alpha \vee \neg\beta) = \nu_x(\neg(\alpha \wedge \beta))$
- If $\psi = \alpha \vee \beta$ then $\psi' = \neg\alpha \wedge \neg\beta$ makes the corresponding ocurrence weakly restricted in $\varphi[\gamma/\gamma']$, because the sign of ψ' is the opposite of the sign of ψ and $\nu_x(\neg\alpha \wedge \neg\beta) = \nu_x(\neg(\alpha \vee \beta))$
- If $\psi = \alpha \rightarrow \beta$ then $\psi' = \neg\neg\alpha \wedge \neg\beta$ makes the corresponding ocurrence weakly restricted in $\varphi[\gamma/\gamma']$, because the sign of ψ' is the opposite of the sign of ψ and $\nu_x(\neg\neg\alpha \wedge \neg\beta) = \nu_x(\neg(\alpha \rightarrow \beta))$. □

Continuing Example 1, after applying the NNF transformations, we obtain the safe rule:

$$request(x, y, z) \wedge (\neg subproc(x, y) \vee \neg\neg has(y, z)) \rightarrow ignore(y, x) \wedge unatt(x) \quad (3)$$

5 Transformations with implications

In the second set of transformations, as in [5] we deal with sets (conjunctions) of implications (the empty conjunction corresponds to $\top$). Each step replaces one of the implications by new implications to be included in the set. If φ is the (matrix of the) original formula, the initial set of implications is the singleton $\{\top \rightarrow \varphi\}$. Without loss of generality, we assume that any implication $\alpha \rightarrow \beta$ to be replaced has been previously transformed into NNF. Furthermore, we always consider that α is a conjunction and β a disjunction (if not, we just take $\alpha \wedge \top$ or $\beta \vee \bot$, respectively), and we implicitly apply commutativity of conjunction and disjunction as needed.

Left side rules:

$$\top \wedge \alpha \rightarrow \beta \iff \{\, \alpha \rightarrow \beta \,\} \quad \text{(L1)}$$

$$\bot \wedge \alpha \rightarrow \beta \iff \varnothing \quad \text{(L2)}$$

$$\neg\neg\varphi \wedge \alpha \rightarrow \beta \iff \{\, \alpha \rightarrow \neg\varphi \vee \beta \,\} \quad \text{(L3)}$$

$$(\varphi \vee \psi) \wedge \alpha \rightarrow \beta \iff \left\{ \begin{array}{l} \varphi \wedge \alpha \rightarrow \beta \\ \psi \wedge \alpha \rightarrow \beta \end{array} \right\} \quad \text{(L4)}$$

$$(\varphi \rightarrow \psi) \wedge \alpha \rightarrow \beta \iff \left\{ \begin{array}{r} \neg\varphi \wedge \alpha \rightarrow \beta \\ \psi \wedge \alpha \rightarrow \beta \\ \alpha \rightarrow \varphi \vee \neg\psi \vee \beta \end{array} \right\} \quad \text{(L5)}$$

Right side rules

$$\alpha \rightarrow \bot \vee \beta \iff \{\, \alpha \rightarrow \beta \,\} \quad \text{(R1)}$$

$$\alpha \rightarrow \top \vee \beta \iff \varnothing \quad \text{(R2)}$$

$$\alpha \rightarrow \neg\neg\varphi \vee \beta \iff \{\, \neg\varphi \wedge \alpha \rightarrow \beta \,\} \quad \text{(R3)}$$

$$\alpha \rightarrow (\varphi \wedge \psi) \vee \beta \iff \left\{ \begin{array}{l} \alpha \rightarrow \varphi \vee \beta \\ \alpha \rightarrow \psi \vee \beta \end{array} \right\} \quad \text{(R4)}$$

$$\alpha \rightarrow (\varphi \rightarrow \psi) \vee \beta \iff \left\{ \begin{array}{c} \varphi \wedge \alpha \rightarrow \psi \vee \beta \\ \neg\psi \wedge \alpha \rightarrow \neg\varphi \vee \beta \end{array} \right\} \quad \text{(R5)}$$

It is perhaps worth emphasising that all these transformations (L1)-(L4), (R1)-(R5) together with those for NNF, (N1)-(N6), are not just the result of an arbitrary choice, but they are justified[9] by (non-redundant) equivalences in the logic **HT**.

Theorem 5. $\mathrm{NSS}(\gamma) = \mathrm{NSS}(\gamma')$ *for any transformation of the form* $\gamma \Longleftrightarrow \gamma'$ *in (L1)-(L4), (R1)-(R5), where* γ' *is the conjunction of the resulting formulas. Therefore, if* γ *is semi-safe, then* γ' *is also semi-safe.*

Proof. We prove case by case:

(L1) Trivially, $\mathrm{RV}(\bot \wedge \alpha) = \mathrm{RV}(\alpha)$ and thus,

$$\mathrm{NSS}(\top \wedge \alpha \to \beta) = \mathrm{NSS}(\beta) \smallsetminus \mathrm{RV}(\top \wedge \alpha) = \mathrm{NSS}(\beta) \smallsetminus \mathrm{RV}(\alpha) = \mathrm{NSS}(\alpha \to \beta)$$

(R1) Trivially, $\mathrm{NSS}(\top \vee \beta) = \mathrm{NSS}(\beta)$ and thus,

$$\mathrm{NSS}(\alpha \to \bot \vee \beta) = \mathrm{NSS}(\bot \vee \beta) \smallsetminus \mathrm{RV}(\alpha) = \mathrm{NSS}(\beta) \smallsetminus \mathrm{RV}(\alpha) = \mathrm{NSS}(\alpha \to \beta)$$

(L3) In the equality $(*)$ of the following sequence, we use that $\mathrm{RV}(\neg\neg\varphi) = \varnothing = \mathrm{NSS}(\neg\varphi)$:

$$\begin{aligned}
\mathrm{NSS}(\neg\neg\varphi \wedge \alpha \to \beta) &= \mathrm{NSS}(\beta) \smallsetminus \mathrm{RV}(\neg\neg\varphi \wedge \alpha) \\
&= \mathrm{NSS}(\beta) \smallsetminus (\mathrm{RV}(\neg\neg\varphi) \cup \mathrm{RV}(\alpha)) \\
&= (\mathrm{NSS}(\neg\varphi) \cup \mathrm{NSS}(\beta)) \smallsetminus \mathrm{RV}(\alpha) \qquad (*) \\
&= (\mathrm{NSS}(\neg\varphi \vee \beta)) \smallsetminus \mathrm{RV}(\alpha) \\
&= \mathrm{NSS}(\alpha \to \neg\varphi \vee \beta)
\end{aligned}$$

(R3) In the equality $(*)$ of the following sequence, we use that $\mathrm{RV}(\neg\neg\varphi) = \varnothing = \mathrm{NSS}(\neg\varphi)$:

$$\begin{aligned}
\mathrm{NSS}(\alpha \to \neg\neg\varphi \vee \beta) &= \mathrm{NSS}(\neg\neg\varphi \vee \beta) \smallsetminus \mathrm{RV}(\alpha) \\
&= (\mathrm{NSS}(\neg\neg\varphi) \cup \mathrm{NSS}(\beta)) \smallsetminus \mathrm{RV}(\alpha) \\
&= \mathrm{NSS}(\beta) \smallsetminus (\mathrm{RV}(\neg\varphi) \cup \mathrm{RV}(\alpha)) \qquad (*) \\
&= \mathrm{NSS}(\beta) \smallsetminus (\mathrm{RV}(\neg\varphi \wedge \alpha)) \\
&= \mathrm{NSS}(\neg\varphi \wedge \alpha \to \beta)
\end{aligned}$$

(L4) The main steps are properties of naive set theory

$$\begin{aligned}
\mathrm{NSS}((\varphi \vee \psi) \wedge \alpha \to \beta) &= \mathrm{NSS}(\beta) \smallsetminus \mathrm{RV}((\varphi \vee \psi) \wedge \alpha) \\
&= \mathrm{NSS}(\beta) \smallsetminus ((\mathrm{RV}(\varphi) \cap \mathrm{RV}(\psi)) \cup \mathrm{RV}(\alpha)) \\
&= \mathrm{NSS}(\beta) \smallsetminus ((\mathrm{RV}(\varphi) \cup \mathrm{RV}(\alpha)) \cap (\mathrm{RV}(\psi) \cup \mathrm{RV}(\alpha))) \\
&= \mathrm{NSS}(\beta) \cap ((\mathrm{RV}(\varphi) \cup \mathrm{RV}(\alpha)) \cap (\mathrm{RV}(\psi) \cup \mathrm{RV}(\alpha)))^{\complement} \\
&= \mathrm{NSS}(\beta) \cap ((\mathrm{RV}(\varphi) \cup \mathrm{RV}(\alpha))^{\complement} \cup (\mathrm{RV}(\psi) \cup \mathrm{RV}(\alpha))^{\complement}) \\
&= (\mathrm{NSS}(\beta) \cap (\mathrm{RV}(\varphi) \cup \mathrm{RV}(\alpha))^{\complement}) \cup (\mathrm{NSS}(\beta) \cap (\mathrm{RV}(\psi) \cup \mathrm{RV}(\alpha))^{\complement}) \\
&= \mathrm{NSS}(\varphi \wedge \alpha) \to \beta) \cup \mathrm{NSS}(\psi \wedge \alpha \to \beta)
\end{aligned}$$

[9] In fact, all the right hand sides of these transformations constitute a *minimal* representation (in the sense of [9]) in **HT** of their corresponding left hand sides as logic program rules.

(R4) The main steps are properties of naive set theory

$$\begin{aligned}
\mathrm{NSS}(\alpha \to (\varphi \wedge \psi) \vee \beta) &= \mathrm{NSS}((\varphi \wedge \psi) \vee \beta) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= (\mathrm{NSS}(\varphi) \cup \mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= (\mathrm{NSS}(\varphi) \cup \mathrm{NSS}(\beta) \cup \mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= ((\mathrm{NSS}(\varphi) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement}) \cup ((\mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement}) \\
&= \mathrm{NSS}(\alpha \to \varphi \vee \beta) \cup \mathrm{NSS}(\alpha \to \psi \vee \beta)
\end{aligned}$$

(R5) In the equality $(*)$ of the following sequence, we use that $\mathrm{RV}(\neg\varphi) = \varnothing = \mathrm{RV}(\neg\psi)$:

$$\begin{aligned}
\mathrm{NSS}(\alpha \to (\varphi \to \psi) \vee \beta) &= (\mathrm{NSS}(\varphi \to \psi) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= ((\mathrm{NSS}(\psi) \cap \mathrm{RV}(\varphi)^{\complement}) \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= (\mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap (\mathrm{RV}(\varphi)^{\complement} \cup \mathrm{NSS}(\beta)) \cap \mathrm{RV}(\alpha)^{\complement} \\
&= (\mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap ((\mathrm{RV}(\varphi)^{\complement} \cap \mathrm{RV}(\alpha)^{\complement}) \cup (\mathrm{NSS}(\beta) \cap \mathrm{RV}(\alpha)^{\complement})) \\
&= ((\mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap (\mathrm{RV}(\varphi)^{\complement} \cap \mathrm{RV}(\alpha)^{\complement})) \\
&\quad \cup ((\mathrm{NSS}(\psi) \cup \mathrm{NSS}(\beta)) \cap (\mathrm{NSS}(\beta) \cap \mathrm{RV}(\alpha)^{\complement})) \\
&= \mathrm{NSS}(\varphi \wedge \alpha \to \psi \vee \beta) \cup (\mathrm{NSS}(\beta) \cap \mathrm{RV}(\alpha)^{\complement}) \\
&= \mathrm{NSS}(\varphi \wedge \alpha \to \psi \vee \beta) \cup ((\mathrm{NSS}(\neg\varphi) \cup \mathrm{NSS}(\beta)) \cap (\mathrm{RV}(\alpha) \cup \mathrm{RV}(\neg\psi))^{\complement}) \\
&= \mathrm{NSS}(\varphi \wedge \alpha \to \psi \vee \beta) \cup \mathrm{NSS}(\neg\psi \wedge \alpha \to \neg\varphi \vee \beta)
\end{aligned}$$

□

It is important to note that (L5) *does not preserve* semi-safety. If $(\varphi \to \psi) \wedge \alpha \to \beta$ is semi-safe then, although we can easily see that the two first rules resulting from (L5) are semi-safe:

$$\begin{aligned}
\varnothing = \mathrm{NSS}((\varphi \to \psi) \wedge \alpha \to \beta) &= \mathrm{NSS}(\beta) \setminus \mathrm{RV}((\varphi \to \psi) \wedge \alpha) \\
&= \mathrm{NSS}(\beta) \setminus \mathrm{RV}(\alpha) \\
&= \mathrm{NSS}(\beta) \setminus (\mathrm{RV}(\neg\varphi) \cup \mathrm{RV}(\alpha)) \\
&= \mathrm{NSS}(\neg\varphi \wedge \alpha \to \beta)
\end{aligned}$$

$$\begin{aligned}
\varnothing = \mathrm{NSS}((\varphi \to \psi) \wedge \alpha \to \beta) &= \mathrm{NSS}(\beta) \setminus \mathrm{RV}(\alpha) \\
&\supseteq \mathrm{NSS}(\beta) \setminus (\mathrm{RV}(\psi) \cup \mathrm{RV}(\alpha)) \\
&= \mathrm{NSS}(\psi \wedge \alpha \to \beta)
\end{aligned}$$

the third rule $\alpha \to \varphi \vee \neg\psi \vee \beta$, in the general case, is not semi-safe. As a counterexample, take the formula

$$(p(x) \to q) \to \neg r(x) \tag{4}$$

This formula is semi-safe and in fact, is safe. However, after applying (L5) to (4), we get the implication $\top \to p(x) \vee \neg q \vee \neg r(x)$, which is not semi-safe (in particular,

the first of occurrence of x is not semi-safe) and thus, is not safe either. As we will see next, our definition of safety is indeed preserved for all transformations involving nested expressions (N1)-(N6), (L1)-(L4), (R1)-(R4), but fails for some cases dealing with nested implications.

Lemma 3. *For any pair $\gamma \iff \gamma'$ in transformations (L1)-(L5), (R1)-(R5) we have $\nu_x(\gamma) = \nu_x(\gamma')$ and thus, $\nu_x(\psi) = \nu_x(\psi[\gamma/\gamma'])$ for any formula ψ.*

Proof. Again, the result follows from semantic equivalences in Kleene's logic. □

Theorem 6. *Consider a semi-safe sentence $\forall x_1 \ldots \forall x_n \gamma$ and a pair $\gamma \iff \gamma'$ in transformations (L1)-(L4), (R1)-(R5): if x_i is safe in γ then it is also safe in γ'.*

Proof. The proof is similar to that for Theorem 4. To prove the result, we must analyse each occurrence of every variable in γ' to check if it is made weakly restricted. Again, each of these occurrences corresponds, in a natural way, to a specific occurrence of the same variable in the formula φ, although an occurrence in γ may correspond to up to two occurrences in γ'. Also, it is easy to check now that the transformation does not modify the sign of the occurrences of the variables and that, in any pair in transformations (L1)-(L5), (R1)-(R5), if δ is a subformula of α, β, φ or ψ, then the sign of δ in γ is equal to the sign of the corresponding occurrence of δ in γ'.

Since the sentence is universal, if δ is a subformula that makes weakly restricted an ocurrence of x_i then we only need to analyse the cases in which δ is a strict subformula of γ, because the proof for the other situations is the same as for Theorem 4. Finally, for (L1), (L2), (L3), (R1), (R2) and (R3) the result is trivial.

(L4) If $\delta = \varphi \vee \psi$, then $\nu_x(\varphi \vee \psi) = \bot$ and thus $\nu_x(\varphi) = \nu_x(\psi) = \bot$; therefore, φ makes weakly restricted the corresponding ocurrence of x_i in $\varphi \wedge \alpha \rightarrow \beta$ and ψ makes weakly restricted the corresponding ocurrence of x_i in $\psi \wedge \alpha \rightarrow \beta$.
If $\delta = (\varphi \vee \psi) \wedge \alpha$, then $\nu_x((\varphi \vee \psi) \wedge \alpha) = \bot$ and either $\nu_x(\varphi \vee \psi) = \bot$ or $\nu_x(\alpha) = \bot$; both cases reduce to some of the previous cases.

(R4) If $\delta = \varphi \wedge \psi$, then $\nu_x(\varphi \wedge \psi) = \top$ and thus $\nu_x(\varphi) = \nu_x(\psi) = \top$; therefore, φ makes weakly restricted the corresponding ocurrence of x_i in $\alpha \rightarrow \varphi \vee \beta$ and ψ makes weakly restricted the corresponding ocurrence of x_i in $\alpha \rightarrow \psi \vee \beta$.
If $\delta = (\varphi \wedge \psi) \vee \beta$, then $\nu_x((\varphi \vee \psi) \wedge \alpha) = \top$ and either $\nu_x(\varphi \wedge \psi) = \top$ or $\nu_x(\alpha) = \top$; both cases reduce to some of the previous cases.

(R5) If $\delta = \varphi \rightarrow \psi$, then $\nu_x(\varphi \rightarrow \psi) = \top$ and either $\nu_x(\varphi) = \bot$ or $\nu_x(\psi) = \top$. In both cases, $\nu_x(\varphi \wedge \alpha \rightarrow \psi \vee \beta) = \nu_x(\neg\psi \wedge \alpha \rightarrow \neg\varphi \vee \beta) = \top$ and the complete formulas make the corresponding occurrences weakly restricted. □

This result shows that transformations (L1)-(L4) and (R1)-(R4) plus (N1)-(N6), which allow unfolding rules with nested expressions, preserve safety. If we apply these transformations to our running example (3) we obtain the four *safe* rules:

$$request(x,y,z) \wedge \neg subproc(x,y) \rightarrow ignore(y,x)$$
$$request(x,y,z) \wedge \neg subproc(x,y) \rightarrow unatt(x)$$
$$request(x,y,z) \rightarrow ignore(y,x) \vee \neg has(y,z)$$
$$request(x,y,z) \rightarrow unatt(x) \vee \neg has(y,z)$$

In the case of nested implications, although for (L5) we do not obtain a positive result, we can still establish a sufficient condition for preserving safety, as follows.[10]

Theorem 7. *Consider a semi-safe sentence $\forall x_1 \ldots \forall x_n \varphi$, the pair $\gamma \Longleftrightarrow \gamma'$ in transformation (L5) and suppose that $\alpha \rightarrow \varphi$ is semi-safe. Then, if x_i is safe in φ then it is also safe in $\varphi[\gamma/\gamma']$.*

Proof. Semi-safety of rules $(\neg\varphi \wedge \alpha \rightarrow \beta)$ and $(\psi \wedge \alpha \rightarrow \beta)$ was proved before. As for $(\alpha \rightarrow \varphi \vee \neg\psi \vee \beta)$, we get:

$$\begin{aligned}\mathrm{NSS}(\alpha \rightarrow \varphi \vee \neg\psi \vee \beta) &= (\mathrm{NSS}(\beta) \cup \mathrm{NSS}(\varphi) \cup \mathrm{NSS}(\neg\psi)) \setminus \mathrm{RV}(\alpha) \\ &= (\mathrm{NSS}(\beta) \cup \mathrm{NSS}(\varphi)) \setminus \mathrm{RV}(\alpha)\end{aligned}$$

but as $\alpha \rightarrow \varphi$ is semi-safe, $\mathrm{NSS}(\varphi) \setminus \mathrm{RV}(\alpha) = \varnothing$ and we obtain:

$$\begin{aligned}&= \mathrm{NSS}(\beta) \setminus \mathrm{RV}(\alpha) \\ &= \mathrm{NSS}((\varphi \rightarrow \psi) \wedge \alpha \rightarrow \beta) = \varnothing\end{aligned}$$

□

To see how this sufficient condition can be applied, let us consider a variation of (4) where we include in the antecedent an additional atom $dom(x)$ (possibly fixing the "domain" of x):

$$(p(x) \rightarrow q) \wedge dom(x) \rightarrow \neg r(x)$$

This formula is still safe and, furthermore, the implication $dom(x) \rightarrow p(x)$ is semi-safe. Thus, the result of applying (L5) yields the three (now safe) rules:

$$\begin{aligned}\neg p(x) \wedge dom(x) &\rightarrow \neg r(x) \qquad & dom(x) &\rightarrow p(x) \vee \neg q \vee \neg r(x) \\ q \wedge dom(x) &\rightarrow \neg r(x) & &\end{aligned}$$

5.1 Discussion

Taken together, the transformations (N1)-(N6), (L1)-(L5), (R1)-R(5), are sufficient to reduce a universal sentence in prenex form with matrix φ into a prenex formula whose matrix, say φ', has the form of a general disjunctive program rule of shape $\alpha \rightarrow \beta$, where α is a conjunction of literals and β is a disjunction of literals. The resulting transformation therefore has the form of a logic program allowing negation in the heads of rules. As we saw, all transformations preserve the property of safety, except (L5), where safety preservation can be ensured, but at the cost of an additional condition. On the other hand, removing nested occurrences of implication in the heads of rules does not affect safety.

Quantifier-free formulas of the form $\alpha \rightarrow \beta$ where α, β do not contain occurrences of implication, other than in the form '$\rightarrow \bot$', are known in ASP as rules with *nested expressions* and a set of such rules is called a *nested program*, [18]. Taken together, therefore, (N1)-(N6), (L1)-(L4), (R1)-R(5), are sufficient to transform any nested program into a fully equivalent general disjunctive program, preserving the safety of formulas in each case.

[10] Notice that rule (L5) cannot be further simplified. In particular, dropping the problematic third formula on the right hand side we would lose strong equivalence. For instance, the formulas $((p \rightarrow q) \wedge r) \rightarrow s$ and $((\neg p \wedge r) \rightarrow s) \wedge ((q \wedge r) \rightarrow s)$ are not strongly equivalent: $\langle\{r\}, \{p, q, r, s\}\rangle$ is a model of the second one but not of the former.

6 Related Work and Conclusions

We have studied the safety condition on first order formulas from [7, 8] and identified a syntactic class of formulas that can be transformed via rewriting rules that preserve strong equivalence and safety. This syntactic class contains eg. all nested logic programs.

While the condition of safety is in general highly relevant for computational purposes, it should be noted that here we have been concerned with logical issues rather than matters of computation and implementation. In fact, while the transformations we have studied do not introduce any new terms into the language, they are also in general not polynomial in size. Polynomial reductions of nested programs have been studied in [21, 22] and polynomial reductions of arbitrary propositional formulas to logic programs are discussed in [5]. These transformations may lend themselves to a more efficient implementation of the reductions, but they introduce new predicates that may cause loss of safety. In [4] a restricted subclass of nested programs is identified, called *normal form nested* or NFN. A concept of safety for NFN rules is proposed in [4] along with a polynomial algorithm for reducing them to disjunctive programs that can be processed by the DLV system. This reduction method introduces new predicates and other auxiliary devices. It does preserve safety, however [4] does not prove the correctness of the reduction, something that in our case of strong-equivalence preserving transformations is easy to establish. Evidently, our safety concept is also much more widely applicable than that of [4].

There remain several directions for further study. One is the search for an improved concept of safety along with a complete set of transformations that preserve this property. Another is the investigation of algorithms for a more efficient reduction of general formulas to logic programs while preserving safety. Another topic is the study of transformations on existential sentences and arbitrary formulas involving existential quantifiers.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP, Cambridge (2002)
2. Baral, C., Brewka, G. and Schlipf, J. (Eds): *Proc. of the 9th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2007)*, Springer LNAI 4483, 2007.
3. Baral, C., Greco, G., Leone, N. and Terracina, G. (Eds): *Proc. of the 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2005)*, Springer LNAI 3662, 2005.
4. Bria, A., Faber, W. and Leone, N.: Normal Form Nested Programs. In *Proc. of JELIA 2008*, Springer LNAI 5293, pp. 76–88, 2008.
5. Cabalar, P., Pearce, D., Valverde, A.: Reducing Propositional Theories in Equilibrium Logic to Logic Programs. In *Proc. of EPIA 05*, Springer LNAI 3808, pp. 4-17, 2005.
6. Cabalar, P. and Ferraris, P.: Propositional Theories are Strongly Equivalent to Logic Programs, *Theory and Practice of Logic Programming* 7 (6), pp. 745-759, 2007.

7. Cabalar, P., Pearce, D., Valverde, A.: A Revised Concept of Safety for General Answer Set Programs (extended version). Technical Report `http://www.ia.urjc.es/~dpearce`.
8. Cabalar, P., Pearce, D., Valverde, A.: A Revised Concept of Safety for General Answer Set Programs. In *Proc. of LPNMR 09*, Springer, (to appear).
9. Cabalar, P., Pearce, D., Valverde, A.: Minimal Logic Programs. In: Dahl, V., Niemelä, I. (eds) ICLP 07, LNCS 4670, pp. 104-118, Springer, Heidelberg (2007).
10. van Dalen, D.: *Logic and Structure*. Springer, 2004.
11. Ferraris, P.: Answer Sets for Propositional Theories In *Proc. of LPNMR 2005*, Springer LNAI 3662, pp. 119-131, 2005.
12. Ferraris, P., Lee, J. and Lifschitz, V.: A New Perspective on Stable Models. In *Proc. of IJCAI 2007*, pp. 372-379.
13. Klenne, S. C.: *On Notation for Ordinal Numbers.* The J. of Symbolic Logic 3(4), 1938.
14. Lee, J., Lifschitz, V., Palla, R.: A Reductive Semantics for Counting and Choice in Answer Set Programming. In *Proc. of AAAI 2008*, pp. 472-479.
15. Lee, J., Lifschitz V. and Palla, R.: Safe formulas in the general theory of stable models. (Preliminary report). In *Proc. of ICLP-08*, Springer LNCS 5366, 2008.
16. Lifschitz, V., Pearce, D., and Valverde, A.: Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
17. Lifschitz, V., Pearce, D.and Valverde, A.: A Characterization of Strong Equivalence for Logic Programs with Variables. In *Proc. of LPNMR 2007*, Springer LNAI 4483, pp. 188-200, 2007.
18. Lifschitz, V., Tang, L. and Turner, H.: Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4): 369-389, 1999.
19. Pearce, D.: A New Logical Characterisation of Stable Models and Answer Sets. In *Proc. of NMELP 96*, 1997.
20. Pearce, D.: Equilibrium logic. *Ann. Math. & and Art. Intelligence*, 47:3-41, 2006.
21. Pearce, D., Tompits, H., and Woltran, S.: Encodings for equilibrium logic and logic programs with nested expressions. In *Proc. of EPIA 2001*, Springer LNAI 2258, pp. 306-320.
22. Pearce, D., Tompits, H., and Woltran, S.: Chatacterising equilibrium logic and nested logic programs: reductions and complexity. Technical Report GIA 2007-01-12, Universidad Rey Juan Carlos, 2007; to appear in *Theory and Practice of Logic programming*.
23. Pearce, D. and Valverde, A.: Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proc. of JELIA 2004*, Springer LNAI 3229, pp. 147-160.
24. Pearce, D. and Valverde, A.: A First-Order Nonmonotonic Extension of Constructive Logic. *Studia Logica* 80:321-346, 2005.
25. Pearce, D. and Valverde, A.: Quantified Equilibrium Logic. *Tech. report, Univ. Rey Juan Carlos*, 2006. `http://www.matap.uma.es/investigacion/tr/ma06_02.pdf`.
26. Pearce, D. and Valverde, A.: Quantified Equilibrium Logic and Foundations for Answer Set Programs. In *Proc. ICLP 08*, Springer, LNCS, 2008.

Functional (Logic) Programs as Equations over Order-Sorted Algebras*

Bernd Braßel and Rudolf Berghammer

Institut für Informatik,
Christian-Albrechts-Universität zu Kiel
{bbr,rub}@informatik.uni-kiel.de

Abstract. Order-sorted algebras provide a well developed and expressive framework for theoretical considerations about programming languages. We show that a number of declarative program paradigms fit well into this setting. For this aim we introduce a simple order, based on which strict functional programs, lazy functional programs and lazy functional logic programs with run-time choice or call-time choice can be modeled. An interesting feature of the presented approach is that the semantic differences between strictness/laziness on one hand and run-time/call-time choice on the other hand are only reflected in the type of variables allowed in the programs. Especially, this feature allows the admission of mixed programs without any change in the underlying theory.

1 Introduction

The presented work introduces an application of order-sorted algebra to functional and functional logic programming languages. We show that a simple order structure suffices to capture four basic semantical notions within a single formal framework. The result is a unified approach entailing the following advantages.

- By connecting functional logic programming to order-sorted algebra standard results from a well developed theory become applicable to these programming languages. This enables for example the use of standard transformations to obtain unsorted conditional rewrite systems and, further, standard unconditional rewrite systems for a given program.
- The presented framework contains an explicit specification of the laws of non-deterministic choice. This allows for a flexibility to examine alternative approaches to non-determinism by providing different specifications. This flexibility could be used, for example, to integrate approaches to encapsulated search [3, 7] or probabilistic programming languages [9] in the future.
- Because of this integration into a uniform framework, the presented approach allows an arbitrary mix of the integrated semantic notions. This could be useful to examine, e.g., the problems involved with approaches to encapsulated search. In the presented framework these problems can be expressed as a clash of intuitions about the sort of program variables.

* This work has been partially supported by the DFG, grant Ha 2457/1-2.

– The high level of abstraction supported by order-sorted algebra makes the presented framework modularly extensible. Further specifications modeling, e.g., typed programs could be added without losing the achieved results.

The remaining introductory subsection will provide minimalistic examples to distinguish the four basic notions of semantics integrated in the presented framework. Section 2 will then give an introduction to order-sorted algebra. Section 3 develops the new representation of functional (logic) programs as equations over an order-sorted signature. Section 4 discusses the semantics of the resulting programs as well as the connection to former approaches. Especially, that section contains the discussion of how the four semantic notions exemplified in Section 1.1 are integrated within the framework. Section 5 concludes.

1.1 Four Basic Semantic Notions

The formal framework presented in this work integrates four different semantic notions by introducing a simple order of sorts. For functional programming languages the basic semantic notions are well known as call-by-value, call-by-name and call-by-need. With the extension to functional *logic* programming call-by-need is to be further distinguished into call-time choice and run-time choice [6]. The differences between the semantics become apparent in non right-linear program rules, i.e., when variables are duplicated on the right hand side of a rule. The following example is therefore a standard minimal one.

Example 1. Consider the following functional program rule.

```
double x = x+x
```

When evaluating the expression `double (0+1)`, call-by-value allows the following derivation only.

```
double (0+1) = double 1 = 1+1 = 2
```

With call-by-name, in contrast, the sub-expression `(0+1)` is copied and may therefore be evaluated more than once as in the following derivation.

```
double (0+1) = (0+1) + (0+1) = 1 + (0+1) = 1 + 1 = 2
```

Like call-by-name, call-by-need allows to apply the rule for `double` before fully evaluating the arguments. In contrast to call-by-name, however, not sub-expressions but *references to sub-expressions* are copied. This requires syntactic extensions to, e.g., graph rewriting or the introduction of `let`-expressions, as in the following derivation.

```
double (0+1) = let x=0+1 in x+x =
               let x=1   in x+x = 1+1 = 2
```

When considering pure functional programs only it is a well known and intuitive fact that call-by-need semantics corresponds to call-by-name. With the extension to non-deterministic choice, this correspondence becomes less tight and there are two possibilities to interpret call-by-need derivations as illustrated in the next example.

Example 2. A minimalistic way to define a non-deterministic operation in functional logic programming languages is the following.

```
coin = 0
coin = 1
```

We consider the evaluation of the expression `double coin` and separate independent non-deterministic possibilities by the symbol "`|`". In a call-by-value derivation of the expression, the possible choices for `coin` are done before applying the rule for `double`. The resulting semantics is called call-time choice.

```
double coin = double 0 = 0+0 = 0
            | double 1 = 1+1 = 2
```

In call-by-name the sub-expression `coin` is copied resulting in more non-deterministic combinations. The corresponding semantics is called run-time choice.

```
double coin = coin + coin = 0 + coin = 0 + 0 = 0
                                     | 0 + 1 = 1
                          | 1 + coin = 1 + 0 = 1
                                     | 1 + 1 = 2
```

Call-by-need is compatible with both variations of choice semantics. A derivation call-by-need with run-time choice could be represented as follows.

```
double coin = let x=coin in x+x
            = let x=0|1  in x+x = 0|1 + 0|1
            = ... = 0 | 1 | 1 | 2
```

And call-by-need with call-time choice is realized in the next derivation.

```
double coin = let x=coin in x+x
            = let x=0 in x+x = 0
            | let x=1 in x+x = 2
```

The four basic semantic notions considered in the following are therefore call-by-value, call-by-name, call-by-need with run-time choice and call-by-need with call-time choice.

2 (Order-)Sorted Algebra

The aim of this section is to give an account of the notions and notations used in the theory of both sorted and order-sorted algebra [4]. We assume the reader to be familiar with the notions of a *relation*, a *partial order*, the *least* and *largest element* of a partial order, an *equivalence class* and the *transitive and symmetric closure* of a relation. We recall that the *connected components* of a partial order $(M, \leq)$ are the equivalence classes of the transitive and symmetric closure of $\leq$. A subset $D \subseteq M$ is called *directed* iff for all $a, b \in D$ there exists an element $c \in D$ with $a \leq c$ and $b \leq c$. A partial order is called *locally directed* iff each connected component is directed. The first central notion is that of sorts, sorted sets and functions/relations on such sets.

Definition 1. *Let S be an arbitrary set and call it a* set of sorts. *An S-*sorted set *A is a family of sets A_s for each sort in S; and we write $\{A_s \mid s \in S\}$. For S-sorted sets A and B an S-*sorted function *$f : A \to B$ is an S-sorted family $f = \{f_s : A_s \to B_s \mid s \in S\}$ and an S-*sorted relation *$(A, \bowtie)$ is an S-sorted family $\bowtie = \{\bowtie_s \subseteq A_s \times A_s \mid s \in S\}$. Let $(S, \leq)$ be a partially ordered set. Then we call S-sorted sets, functions, and relations* order-sorted.

A signature is a collection of symbols which are later interpreted in algebraic structures. For each symbol a signature contains some structural information, in the simplest case a natural number, called the symbol's *arity*. *Sorted* signatures add more structure to this concept and once sorted signatures are established a desirable next step is to add *subsort relations*. This line of thought leads to *order-sorted* signatures.

Definition 2. *A* many-sorted signature *Σ is a pair (S, M) such that M is an $S^* \times S$-sorted family $\{M_{w,s} \mid \langle w, s\rangle \in S^* \times S\}$. We call S the* sort set *of Σ and the elements of M are called the symbols of Σ.*

An order-sorted signature *Σ is a triple $(S, \leq, M)$, such that (S, M) is a many-sorted signature, $(S, \leq)$ is a partial order for which the symbols in Σ satisfy the* monotonicity *condition:*

$$\sigma \in M_{w,s} \cap M_{w',s'} \text{ and } w \leq w' \text{ imply } s \leq s'$$

We write $o : w \to s \in \Sigma$ to state that $o \in M_{w,s}$ and say that o has rank *$\langle w, s\rangle$,* arity *w and* (result-)sort *s. In the special case that $w = \varepsilon$, the empty sequence, we call $o : \varepsilon \to s$ a* constant *(of sort s). We call an order-sorted signature* regular *iff given $o : w \to s \in \Sigma$ and given $w' \leq w$ in S^* there exists a least rank $\langle w_0, s_0\rangle \in S^* \times S$ such that $w' \leq w_0$ and $o : w_0 \to s_0 \in \Sigma$. We call an order-sorted signature* coherent *if it is regular and its sort set is locally directed.*

In the following we assume Σ to denote a many- or order-sorted signature with sort set S.

Definition 2 introduces several concepts to formalize the idea of subsorts. Firstly, an order serves the purpose of defining the inclusion structure. The purpose of the *monotonicity* condition is to minimize redundancy in the algebras. When the signature is monotone it is possible, for example, to employ the same function to interpret + for integers and natural numbers by simply restricting its domain. Naturally, this property is interesting when thinking about implementation and code duplication. Monotonicity is not enough to ensure that terms can be constructed without type information. A sufficient additional condition is *regularity*. Together with monotonicity the regularity condition makes sure that each term has a well defined least sort [4, Fact 2.4, Proposition 2.10]. Finally, *coherence* will get important when we add equations to the setting below.

Naturally, order-sorted algebra features extended versions of notions like algebra, homomorphisms, isomorphism, etc. In this work, however, we can be content with defining the special algebra of terms and the special homomorphisms called substitutions.

Definition 3. *Let X be an S-sorted family of* disjoint *sets $X = \{X_s \mid s \in S\}$ which we call* variable set *disjoint from S.*

Then the set of terms $\mathcal{T}_{\Sigma(X)}$ is a family of sets $\{\mathcal{T}_{\Sigma(X),s} \mid s \in S\}$ defined as the smallest set satisfying:

(1) $X_s \cup \Sigma_{\varepsilon,s} \subseteq \mathcal{T}_{\Sigma(X),s}$ for any $s \in S$
(2) if $o : w \to s \in \Sigma$ and if $t_i \in \mathcal{T}_{\Sigma(X),s_i}$ for $i \in \{1, \ldots, n\}$ where $w = s_1 \ldots s_n \neq \varepsilon$, then $o(t_1, \ldots, t_n)$ is in $\mathcal{T}_{\Sigma(X),s}$

When $\Sigma = (S, \leq, M)$ is an order-sorted signature we additionally have

(3) $\mathcal{T}_{\Sigma(X),s} \subseteq \mathcal{T}_{\Sigma(X),s'}$ if $s \leq s'$

We will often write $\overline{o_n}$ as an abbreviation for arbitrary objects $o_1, \ldots, o_n$, e.g., $o(\overline{t_n})$ for the term above.

An important feature of a given order-sorted Σ-term t is that there is a unique least sort s such that $t \in T_{\Sigma,s}$. We denote this least sort by $LS(t)$. By $\mathcal{V}ar(t)$ we will denote the set of variables occurring in a term t which is defined in the usual inductive way.

Definition 4. *Let X, Y be two S-sorted variable sets. Then a* substitution *is an S-sorted map $\sigma : X \to \mathcal{T}_{\Sigma(Y)}$. This map can be extended canonically to $\hat{\sigma} : \mathcal{T}_{\Sigma(X)} \to \mathcal{T}_{\Sigma(Y)}$ in the usual inductive way. We adopt the convention that this unique many- or order-sorted homomorphism $\hat{\sigma}$ is also denoted as σ.*

Note especially that variable substitutions are by definition sorted. A variable of sort s is always assigned a term of sort s. Hence, substitutions are *sort preserving*. It may happen, however, that the *least sort* $LS(\sigma(t))$ is smaller than $LS(t)$ because, e.g., a variable of the sort representing integers is replaced with a term of the sort representing natural numbers. The feature of sort preservation is important for the correspondence between variable sorts and the semantic notions introduced in Section 1.1.

Functional and functional logic programs will be defined as a set of equations over special order-sorted signatures.

Definition 5. *A Σ-equation is of the form $\forall X.l = r$ where X is a variable set and $l, r \in \mathcal{T}_{\Sigma(X)}$ such that l and r are of the same sort when Σ is many-sorted and $LS(l)$ and $LS(r)$ are connected in the sort order when Σ is order-sorted.*

When the variable set X is clear from the context, e.g., if it contains only the variables of l and r and these variables and their sorts are clear, we may omit the quantifier $\forall X$.

Up to now we have considered regular signatures. Adding equations to the setting requires the signatures to satisfy an additional property introduced in Definition 2: coherence. Coherence means that for any two connected sorts s, s' there is a sort greater or equal than both. In other words the set of sorts is locally directed with respect to the sort order. Coherence is a sufficient requirement to ensure that isomorphic models satisfy the same equations.

Five rules are sufficient to derive all and only those equations holding in any model of a given set of equations [4, Theorem 3.1 and Corollary 3.2].

Definition 6. *Let Γ be a set of Σ-equations, called* derivable. *The following rules allow to derive further equations:*

(1) Reflexivity: Each equation of the form $\forall X.t = t$ is derivable.
(2) Symmetry: If $\forall X.t = t'$ is derivable then so is $\forall X.t' = t$.
(3) Transitivity: If $\forall X.t = t'$, $\forall X.t' = t''$ are derivable then so is $\forall X.t = t''$.
(4) Congruence: If $\sigma, \sigma' : X \to \mathcal{T}_{\Sigma(Y)}$ are substitutions such that for each $x \in X$ $\forall Y.\sigma(x) = \sigma'(x)$ is derivable then so is $\forall Y.\sigma(t) = \sigma'(t)$ for any $t \in \mathcal{T}_{\Sigma(X)}$.
(5) Substitutivity: If $\forall X.t = t'$ is in Γ and if $\sigma : X \to \mathcal{T}_{\Sigma(Y)}$ is a substitution then $\forall Y.\sigma(t) = \sigma(t')$ is derivable.

In comparison to equational deduction rewriting is more operational.

Definition 7. *A Σ-rewrite rule is of the form $l \to r$ where l, r are terms in $\mathcal{T}_{\Sigma(X)}$ such that l and r are of the same sort if Σ is many sorted or $LS(l)$ and $LS(r)$ are connected when Σ is order-sorted, respectively. For a special constant symbol $\Box \notin \Sigma$ and a sort s in the sort set of Σ, a context (of sort s) is a term C over $\Sigma \cup \{\Box_{\varepsilon,s}\}$ such that $\Box$ occurs exactly once in C. For given terms $\overline{t_n} \in \mathcal{T}_{\Sigma,s}$ the notation $C[t]$ denotes the replacement of the hole from t. A reduction step according to the rewrite rule $\rho = l \to r$ is of the form*

$$C[\sigma(l)] \to_\rho C[\sigma(r)]$$

where C is a context of sort s such that $LS(l), LS(r) \leq s$ and σ is a Σ-substitution. A Σ-rewrite system $\mathcal{R}$ is a set of rewrite rules. The one-step rewrite relation of $\mathcal{R}$, denoted by $\to_\mathcal{R}$, is defined by $t \to_\mathcal{R} s'$ iff there is a rule $\rho \in \mathcal{R}$ such that $t \to_\rho t'$ is a reduction step according to ρ. We may omit the subscripts ρ or $\mathcal{R}$ when rule or rewrite system is clear from context or arbitrary. By $\to^*_\mathcal{R}$ we denote the reflexive and transitive closure of $\to_\mathcal{R}$.

Finally, a rewrite system $\mathcal{R}$ is called compatible *iff for any rewrite step $t \to_\mathcal{R} t'$ and any context C, we have that $C[t] \to_\mathcal{R} C[t']$ is also a rewrite step.*

We will only consider compatible rewrite systems as it can be shown that for such systems key results of the theory of unsorted rewriting carry over. We deviate, however, from common restrictions to rewriting that the left hand side of a rule must not be a variable and that $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ should be satisfied without consequences for the purposes of this work.[1]

3 Functional (Logic) Programs

A functional logic program will be defined as a restricted set of equations over an order-sorted signature. Therefore, the first step will be to define the sorts used

[1] Compare also to [2, page 36] "the restrictions on rewrite rules are not a priori necessary and some authors prefer not to have them." and [1, page 61]: "The two restrictions that distinguish a rewrite rule from an identity avoid certain pathological cases and obvious sources of non-termination. Much of term rewriting carries over to arbitrary identities with only minor modifications."

in a functional logic program together with an appropriate order on these sorts. After that we will define what kind of symbols and equations will be used to construct functional logic programs. The sorts for functional logic programs will be defined to distinguish four classes of terms: *(total) values*, *partial values*, *choices (over partial values)* and finally arbitrary *expressions*. The distinction between these classes is at the heart of our discussion of different semantic conceptions, such as call-by-name versus call-by-value and run-time choice versus call-time choice. The notions of order-sorted algebra will allow us to conveniently state that, e.g., values are a subsort of partial values.

Definition 8. *Let* $S_P = \{Val, PVal, Ch, Exp\}$. *The* set of program sorts $(S_P, \leq_P)$ *is the smallest partial order containing the chain* $Val < PVal < Ch < Exp$.

Obviously, the set of program sorts is locally directed. In functional and functional logic programs alike there is a basic partition of the symbols of a program signature into *constructor* and *operation* symbols. In addition to these user defined symbols we will use the constant symbol $\bot$ and the binary symbol $\sqcup$. Constructors are used to build values while the inclusion of $\bot$ yields a partial value. The symbol $\sqcup$ is used to construct terms of sort Ch and, finally, the appearance of an operation symbol in a term leads to its sort being Exp. This is the content of the following definition.

Definition 9. *Let* C *and* O *be two disjoint (standard) signatures with natural numbers as arity such that* $\bot \notin C_0 \cup O_0$ *and* $\sqcup \notin C_2 \cup O_2$. *Then the* program signature of $C \cup O$ *is the triple* $(S_P, \leq_P, C' \cup O' \cup L)$ *where*

$$\begin{aligned} C' := & \bigcup \{\{c : Val^n \to Val, c : PVal^n \to PVal, \\ & \qquad c : Ch^n \to Ch, c : Exp^n \to Exp\} \mid c \in C_n\} \\ O' := & \quad \{o : Exp^n \to Exp \mid c \in O_n\} \\ L := & \quad \{\bot : \varepsilon \to PVal, \sqcup : Ch\,Ch \to Ch, \sqcup : Exp\,Exp \to Exp\} \end{aligned}$$

Here, for $p \in S_P$ *the notation* p^n *denotes a sequence* $w = p \ldots p$ *with* $|w| = n$.

The symbols in the subsets of C *are called constructor symbols and those in the subsets of* O *are called operation symbols, respectively. By* Σ_C *and* Σ_O *we denote the sub-signature of constructor and operation symbols, respectively.*

We will prove that program signatures are order sorted signatures and then give some examples of terms constructed from program signatures and their respective terms.

Proposition 1. *Program signatures are regular and coherent order-sorted signatures. Moreover, any rewrite system over a program signature is compatible.*

Proof. Let $\Sigma = (S, \leq, M)$ be a program signature. Then Σ satisfies the monotonicity condition of Definition 2 since all ranks in Σ are of the form $\langle p^n, p\rangle$ and therefore $o \in M_{w,p} \cap M_{w',p'}$ and $w \leq w'$ always imply $p \leq p'$. Moreover, Σ is regular since for any s and natural number n the ranks

$$\{\langle w, p\rangle \mid s : w \to p \in \Sigma, |w| = n\}$$

term t	$LS(t)$
$\texttt{Cons}(\texttt{True}, \texttt{Nil})$	Val
$\texttt{Cons}(\texttt{True}, \bot)$	$PVal$
$\texttt{Cons}(\texttt{True} \sqcup \texttt{False}, \bot)$	Ch
$\texttt{Cons}(\texttt{True}, \texttt{tail}(\texttt{Nil}))$	Exp

term t	$LS(t)$
$\texttt{Cons}(v, \texttt{Nil})$	Val
$\texttt{Cons}(\texttt{True}, p)$	$PVal$
$\texttt{Cons}(c, \bot)$	Ch
$\texttt{Cons}(\texttt{True}, e)$	Exp

Fig. 1. Different Terms and their Respective Sorts

form a chain. Finally, Σ is coherent by Definition 8. Rewrite systems over Σ are compatible since each n-ary symbol has a rank $\langle Exp^n, Exp \rangle$. Therefore, replacing a sub-term of program sort s with a term of program sort $s' > s$ again yields a well-sorted term in general, regardless of the rewrite rules. □

Example 3. Let $C := C_0 \cup C_2$ where $C_0 := \{\texttt{True}, \texttt{False}, \texttt{Nil}\}$ and $C_2 = \{\texttt{Cons}\}$ and let $O := O_1 := \{\texttt{tail}\}$. Then the left table of Figure 1 shows terms over the program signature of $C \cup O$ together with their least sort. By Definition 3 variables are also sorted. Let X be an according set of variables and $v \in X_{Val}$, $p \in X_{PVal}$, $c \in X_{Ch}$ and $e \in X_{Exp}$. The terms with variables and their least sorts are shown in the right table of Figure 1. Moreover, note that no substitution in the sense of Definition 4 can map the variable v to, e.g., the term $\texttt{tail}(\texttt{Nil})$ or to $\texttt{Nil} \sqcup \texttt{Cons}(\texttt{True}, \texttt{Nil})$.

We are now ready to give our notion of a program.

Definition 10. *Let C be a set of constructor and O a set of operations symbols, and Σ- be the program signature for $C \cup O$. Then a* program over Σ *is a pair (Σ, E) such that the following conditions hold.*

- *E is a set of* program rules*. There are two forms of rules: Σ-equations $l = r$ and Σ-*inequations *$l \sqsupseteq r$. In both forms we call l the left-hand side and r the right-hand side of the rule. For any rule l, r must satisfy the conditions*
  - *l is linear, i.e., each variable in $\mathcal{V}ar(l)$ occurs at most once in l*
  - *l is of the form $f(\overline{t_n})$ where $f \in O_n$ and for each t_i holds $t_i \in \mathcal{T}_{\Sigma_C}(X)$*
- *If $l = r$ is a rule in E there must be no other rule $l' = r'$ or $l' \sqsupseteq r'$ in E such that l, l' are unifiable, i.e., there is no substitution σ with $\sigma(l) = \sigma(l')$.*

4 Semantics

Definition 10 introduces two kinds of program rules: equations and inequations. Definition 6 introduced how new equations can be derived from given ones. However, we still have to define how the inequations contained in a program should be treated in such a derivation. For this we follow the classic way to integrate inequations into an equational setting: by embedding orders into (semi) lattices. The idea is that inequations of the form $l \sqsupseteq r$ will be treated as equations $l = l \sqcup r$. Not incidentally, the symbol $\sqcup$ will serve at the same time as an operator for non-deterministic choice. The benefit we get from the general setting

of arbitrary equations over program signatures introduced in Section 2 is that we can define the laws for $\sqcup$ within the same framework as the programs. Thus, the setting allows us to define not only program rules but also the semantical logic within the same framework. The additional equations fixing semantical properties will be called the *program logic* in the following.

4.1 Semi Lattices, Inequations and Program Logic

We start our semantic considerations by defining the notion of a semi lattice.

Definition 11. *A semi lattice is a structure $(S, \sqcup)$ satisfying the following laws for all elements of S.*

$$x \sqcup x = x \quad \mathsf{(idem)} \qquad x \sqcup y = y \sqcup x \quad \mathsf{(comm)} \qquad (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \quad \mathsf{(assoc)}$$

A structure $(S, \sqcup, \bot)$ is called a semi lattice with identity iff $(S, \sqcup)$ is a semi lattice which additionally satisfies the following law for all elements of S.

$$x = \bot \sqcup x \quad \mathsf{(bot)}$$

Semi lattices induce a partial order $(S, \sqsubseteq)$ defined by $x \sqsubseteq y$ iff $x \sqcup y = y$. Moreover, for a semi lattice with identity, $\bot$ is the least element of $(S, \sqsubseteq)$.

A *semi lattice homomorphism* is a function $h : S \to S'$ where S, S' are semi lattices such that $h(a \sqcup_S b) = h(a) \sqcup_{S'} h(b)$. As we will see below, constructors will be defined to be semi lattice homomorphisms. A *monotone function* can be likewise understood as a homomorphism on orders. For such a function $h : S \to S'$, we have that $a \sqsupseteq_S b$ implies $h(a) \sqsupseteq_{S'} h(b)$. Below we will define that all operations introduced in a program are monotone.

In accordance with Definition 11, inequations are treated as syntactic sugar for special equations. The reason to add inequations at all is to make our programs look similar to those used in existing functional logic languages. The main difference now is that we require the use of $l \sqsupseteq r$ for overlapping rules instead of also using the symbol $=$. Actually, we think that adding such a requirement to existing languages could improve the programming practice. With such a requirement, the program from Example 2

```
coin = 0
coin = 1
```

would be rejected by the compiler, since the left-hand sides of the rules are trivially unifiable. Rather, the program should be written as

```
coin ⊒ 0
coin ⊒ 1
```

and by stating it this way, we would have more confidence that the programmer knows "what he is doing". (In addition we do no longer induce the feeling that `True=False` is derivable via a reasoning along the lines of `True=coin=False`.)

Like a program, a program logic is a set of equations over a program signature. Unlike the program, however, the program logic is not restricted syntactically. This allows us to add the equations fixing the laws of non-determinism.

Definition 12. *Let P be a program over a program signature Σ with operations O and constructors C. The set of Σ-equations $PL(P)$ contains the equations of a semi lattice with identity (Definition 11) together with the following equations, where the variables $x, y, z, \overline{x_n}, \overline{y_m}$ are of sort Exp.*

$$\{f(\overline{x_n}, x \sqcup y, \overline{y_m}) \sqsupseteq f(\overline{x_n}, x, \overline{y_m}) \mid f \in O_{n+m+1}\} \quad \text{(mon)}$$
$$\{c(\overline{x_n}, x \sqcup y, \overline{y_m}) = c(\overline{x_n}, x, \overline{y_m}) \sqcup c(\overline{x_n}, y, \overline{y_m}) \mid c \in C_{n+m+1}\} \quad \text{(hom)}$$

4.2 Program Classification

The definition of program sorts allows us to define different classes of programs. The distinction concerns the variables used within the program. If, for instance, all variables have to be of sort *Val* then every argument of any function has to be a constructor term. Obviously, this corresponds to the arguments being fully evaluated. In the other extreme if variables are of sort *Exp* the resulting program will be call-by-name. The two intermediate possibilities classify the different approaches to non-determinism in programs with call-by-need semantics.

Definition 13. *When all the variables contained in program rules are*

of sort	*the program obeys*
Val	*call-by-value (with call-time choice)*
PVal	*call-by-need with call-time choice*
Ch	*call-by-need with run-time choice*
Exp	*call-by-name (with run-time choice)*

A program containing variables of different sorts will be called mixed. *In addition there is the classification into functional and functional logic programs. A program is called* functional *iff all program rules are of the form $l = r$, do not contain the symbol $\sqcup$ and satisfy the proposition $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A program which is not functional is called a* functional logic program.

In the remainder of this section we will exemplify the different program classes by revisiting the introductory example from Section 1.1. After that we will strengthen our result by showing that functional logic programs with call-by-need and call-time-choice correspond in our framework to the classical modeling of this class [5].

Example 4. Recall the program in Example 1.

```
double x = x+x
```

First we consider `x` to be of sort *Exp*. Then mapping `x` to `0+1` yields a valid substitution and therefore we can derive from the equation `double x = x+x` that `double (0+1) = (0+1) + (0+1)` holds by the substitutivity rule of Definition 6. This yields the validity of the corresponding equational deduction from Example 1 (assuming suitable equations for addition).

```
double (0+1) = (0+1) + (0+1) = 1 + (0+1) = 1+1 = 2
```

When `x` is of sort *Val*, in contrast, `0+1` is not a valid substitution for `x`. Therefore, the above deduction is not possible. The only derivation to yield a constructor value is the call-by-value deduction of Example 1.

```
double (0+1) = double 1 = 1+1 = 2
```

That program variables of sort *PVal* yield a call-by-need semantics is less obvious. Indeed, the only valid deduction to a value in such a program corresponds to the call-by-value one above. Therefore we consider an example for which call-by-value and call-by-need indeed have different semantics. For this we consider `Z` and `S` the symbols to construct Peano numbers.

```
loop = loop
isZero (S x) = True
```

For this program the expression `isZero (S loop)` is not greater or equal than any constructor value when `x` is of sort *Val*. If, however, the sort of `x` is *PVal*, then we may substitute `x` by $\bot$ and get the following deduction.

```
isZero (S loop) = isZero (S (⊥ ⊔ loop))      (bot)
                = isZero (S ⊥ ⊔ S loop)       (hom)
                ⊒ isZero (S ⊥)                (mon)
                = True
```

The above considerations make it is easy to see that programs with variables of sort *Val* feature call-time and those of sort *Exp* feature run-time choice. Therefore, the further classification will consider call-by-need programs, only.

Example 5. Reconsider the program from Example 2 (in the new syntax).

```
coin ⊒ 0
coin ⊒ 1
```

When the variable `x` in the definition of `double` is of sort *Ch*, mapping it to `0⊔1` yields a valid substitution. Therefore we get the following deduction.

```
 double coin = double (0 ⊔ coin)
             = double (0 ⊔ (1 ⊔ coin))
             = double ((0 ⊔ 1) ⊔ coin)      (assoc)
             ⊒ double (0 ⊔ 1)               (mon)    (∗)
             = (0 ⊔ 1) + (0 ⊔ 1)
             ⊒ 0 + (0 ⊔ 1) ⊔ 1 + (0 ⊔ 1)    (idem),(mon)
             ⊒ 0+0 ⊔ 0+1 ⊔ 1+0 ⊔ 1+1         (idem),(mon)
          ... = 0 ⊔ 1 ⊔ 2                    (+),(assoc),(idem)
```

When `x` is of sort *PVal*, we can likewise proceed to the term `double (0⊔1)` in the line marked $(*)$. But then the above deduction is no longer valid as `x` may not be substituted with `0⊔1`. Instead we can only derive values in the following way.

```
double (0⊔1) = double 0 ⊔ double 1 ⊔ double (0⊔1)   (mon)×2
             ⊒ double 0 ⊔ double 1                   def ⊒
             = 0+0 ⊔ 1+1
             = 0 ⊔ 2
```

The discussion above employed the general assumption of functional logic programming that one is only interested in values, i.e., constructor terms which are terms of sort $PVal$ in our setting. In general, the terms considered as values should match the maximal sort of the variables allowed in the program. A program employing variables of sort $PVal$, for instance, should consider the set T_{PVal} for values whereas a call-by-value program should employ T_{Val}. This is the content of the following definition.

Definition 14. *Let P be a Σ-program and s a program sort. Then the s-semantics of a given Σ-term e, denoted by $[\![e]\!]_s$, is the set $[\![e]\!]_s = \{v \in T_s \mid e \sqsupseteq v\}$.*

In order to lift this definition to an algebraic semantics one would need to use standard constructs to obtain a complete lattice from the lattice defined by the Σ-terms. Well known constructions are completion by ideals or Dedekind-MacNeille completion [8, Section 2.2]. We omit the according development due to space constraints. Also due to space constraints we consider the above observations as sufficient to demonstrate the correspondences for call-by-value. As call-by-need with call-time choice is a more subtle concept we show that programs with variable sort $PVal$ indeed obey this semantics. For this we prove correspondence with the classical setting [5] in the next subsection. Regarding the two remaining semantics employing run-time choice we think that our approach is indeed novel as detailed in the conclusion.

4.3 Call-by-Need with Call-Time Choice – a Proof

In this section we compare our notion of call-by-need with call-time choice with the classical setting for functional logic programs by González-Moreno et al [5], called the *CRWL*-calculus.

Definition 15 (*CRWL*). *Let C, O be two disjoint (standard) signatures with natural numbers as arity such that $\bot \notin C_0 \cup O_0$ and $\sqcup \notin C_2 \cup O_2$. Then the set*

1) of CRWL-values CV is defined as the terms over C
2) of CRWL-partial values CP is defined as the terms over $C \cup \{\bot^{(0)}\}$
3) of CRWL-expressions CE is defined as the terms over $C \cup O \cup \{\bot^{(0)}\}$

in the usual inductive way. A substitution θ over the signature $C \cup O \cup \{\bot^{(0)}\}$ is a CRWL-substitution ($\theta \in CS$) iff its range is a subset of CP.

We assume C and O fixed in the following. The terms employed in the *CRWL*-setting correspond very closely to those defined in this work as indicated by the following proposition.

Proposition 2. *Let Σ be the program signature over C and O and for any Σ-term t let $\lfloor t \rfloor$ be the term over the signature $C \cup O \cup \{\bot^{(0)}, \sqcup^{(2)}\}$ derived by simply forgetting about the sorts, which is possible by Definition 9. Likewise, for any expression $e \in CE$ we define $\lceil e \rceil$ to be the Σ-term whith $\lfloor \lceil e \rceil \rfloor = e$ and where all variables are of sort $PVal$. This definitions are naturally extended to Σ-substitutions and constructor substitutions. Then we have*

1) $t \in \mathcal{T}_{\Sigma(X),Val}$ implies $\lfloor t \rfloor \in CV$
2) $t \in \mathcal{T}_{\Sigma(X),PVal}$ implies $\lfloor t \rfloor \in CP$ and $\lceil \lfloor t \rfloor \rceil = t$
3) $e \in CE$ implies $\lceil e \rceil \in \mathcal{T}_{\Sigma(X),PVal}$
4) $\mathcal{V}ar(t) \subseteq \mathcal{T}_{\Sigma(X),PVal}$ implies for any Σ-substitution σ that $\lfloor \sigma \rfloor$ restricted to $\mathcal{V}ar(t)$ is in CS
5) $\rho \in CS$ and $e \in CE$ imply $\lceil \rho \rceil(\lceil e \rceil) = \lceil \rho(e) \rceil$.

Proof. Obvious by construction of the program signature from C and O.

Definition 16. *For any Σ-program P the set of rewrite rules*

$$\{\lfloor l \rfloor \rightarrowtail \lfloor r \rfloor \mid l = r \in P \vee l \sqsupseteq r \in P\} \quad \cup \quad \{x \sqcup y \rightarrowtail x\} \quad \cup \quad \{x \sqcup y \rightarrowtail y\}$$

is called the CRWL-Program $\lfloor P \rfloor$. For any such program the one-step CRWL-reduction $\rightarrowtail$ is defined by

$$\begin{array}{lll} \textsf{(OR)}\ C[f(\overline{\theta(t_n)})] & \rightarrowtail & C[\theta(e)] \text{ if } \theta \in CS \text{ and } f(\overline{t_n}) \rightarrowtail e \in \lfloor P \rfloor \\ \textsf{(B)}\ \ C[e] & \rightarrowtail & C[\bot] \quad \text{if } e \in CE \end{array}$$

where C is a context over signature $C \cup O \cup \{\bot^{(0)}, \sqcup^{(2)}\}$.

Because of the close correspondence between the *CRWL*-notions and those introduced in this paper, we can directly relate *CRWL* derivations and order-sorted rewriting with respect to a Σ-program.

Proposition 3. *Let P be a Σ-program and P' be the set of the following order-sorted rewrite rules, where x, y are of sort $PVal$, z of sort Exp.*

$$\{l \rightarrow r \mid l = r \in P \vee l \sqsupseteq r \in P\} \quad \cup \quad \{x \sqcup y \rightarrow x\} \quad \cup \quad \{x \sqcup y \rightarrow y\} \quad \cup \quad \{z \rightarrow \bot\}$$

Then we have for any Σ-terms t, t' of sort Exp that $t \rightarrow t'$ implies $\lfloor t \rfloor \rightarrowtail \lfloor t' \rfloor$ for CRWL-program $\lfloor P \rfloor$. And for any $e, e' \in CE$ with $e \rightarrowtail e'$ we have $\lceil e \rceil \rightarrow \lceil e' \rceil$.

Proof. Obvious by Definitions 7 and 16 and by Proposition 2.

For the rewrite system of Proposition 3 the semantics of a given term e is:

$$\{\{e\}\} := \{v \in T_{PVal} \mid e \rightarrow^* v\}$$

With this last definition we have translated the *CRWL*-setting into a rewrite system which is comparable within our framework. It remains to be shown how the equational deduction for our programs correspond to this rewrite system. For this, we will show that $[\![e]\!]_{PVal}$ in our setting is equivalent to $\{\{e\}\}$. The following lemma translated from the classical setting will be useful.

Lemma 1. *Let P be a Σ-program, C a Σ-context, σ a Σ-substitution and s, t Σ-terms. Then $\{\{s\}\} = \{\{t\}\}$ implies $\{\{C[s]\}\} = \{\{C[t]\}\}$ and $\{\{\sigma(s)\}\} = \{\{\sigma(t)\}\}$.*

Proof. We proof context stability by induction on the structure of C. The base case $C = \Box$ holds trivially. There are the following inductive cases.
Case 1, $C = e \sqcup \Box$ or $C = \Box \sqcup e$: The only way to eliminate the $\sqcup$-symbol of the context is by a reduction via one of the rules $x \sqcup y \to x$ or $x \sqcup y \to y$. In both cases we have $C[s] \to^* v$ iff $e \to^* v$ or $s \to^* v$ which is equivalent by assumption to $C[t] \to^* v$.
Case 2, $C = c(\overline{e}, \Box, \overline{e'})$: Since c is a constructor symbol we directly have $C[t] \to^* v$ iff $C[t'] \to^* v$.
Case 3, $C = f(\overline{e_i}, \Box, \overline{e'_j})$: By assumption all program variables are of sort $PVal$. Therefore we have $\sigma(f(\overline{p_{i+j+1}})) \to \sigma(r)$ for any rule with left hand side $f(\overline{p_{i+j+1}})$ and right-hand side r iff $\sigma(p_{i+1})$ is a term of sort $PVal$. This in turn means that $C[t]$ has to be reduced to $C[v]$ for some $v \in \{\{t\}\}$ before the rule can be applied. Therefore we have $C[s] \to^* v$ iff $C[t] \to^* v$ as required.

For stability under substitution we show that generally $s \to^* t$ implies $\sigma(s) \to^* \sigma(t)$. This claim follows by a simple induction on the length of the derivation because for any term $C[\theta(t)]$ we have

$$\sigma(C[\theta(t)]) = \sigma(C)[\sigma(\theta(x))] = \sigma(C)[\sigma \circ \theta(x)]$$

by definitions of context and substitution. Thus, any step $C[\theta(l)] \to C[\theta(r)]$ implies by definition the existence of a rewrite step $\sigma(C[\theta(l)]) \to \sigma(C[\theta(r)])$. $\square$

Theorem 1. *Let P be a Σ-Program, $e \in T_{Exp}$ and $v \in T_{PVal}$. Then we have that $\{\{e\}\} = [\![e]\!]_{PVal}$, i.e., $e \to^* v$ iff $e \sqsupseteq v$.*

Proof. ($\Rightarrow$): We first observe that a simple induction on the structure of an arbitrary context C shows that a finite application of the laws (mon) and (hom) derive $C[s \sqcup t] \sqsupseteq C[s]$ for arbitrary terms s, t. With this we show the claim by induction on the length n of the derivation $e \to^* v$.
Base case $n = 0$: By the reflexivity rule (1) of Definition 6 we have $v = v$. Employing a substitution of law (idem) we get $v = v \sqcup v$ which yields $v \sqsupseteq v$.
Inductive case, let the claim hold for $t \to^* v$. We show the claim for $e \to_\rho t$ by distinguishing the rule applied in that step.
Case 1, ρ is $x \sqcup y \to x$ or $x \sqcup y \to y$. Then the step $e \to t$ is of the form $C[e_1 \sqcup e_2] \to C[e_i]$ for suitable C, e_1, e_2 and i. By the above observation we can derive, applying laws (mon), (hom)and maybe (comm) if $i = 2$, that $C[e_1 \sqcup e_2] \sqsupseteq C[e_i]$.
Case 2, ρ is $x \to \bot$. Then $e \to t$ is of the form $C[e'] \to C[\bot]$ for a suitable C,e'. An application of law (bot) yields $C[e'] = C[\bot \sqcup e']$ by congruence and the observation thus entails $e \sqsupseteq C[\bot]$.
Case 3, ρ is derived from a program rule. Then the step $e \to t$ is of the form $C[\sigma(l)] \to C[\sigma(r)]$ and we have either $l \sqsupseteq r$ or $l = r$ in P. In this case the claim holds by substitutivity and congruence.

($\Leftarrow$): We first note that $x \sqcup y \to^* x$ and $x \sqcup y \to^* y$ by $x \sqcup y \to x \sqcup \bot \to x$ and $x \sqcup y \to \bot \sqcup y \to y$. Now let s, t be Σ-terms such that $s = t$. We show that for

any $v \in T_{PVal}$ that $s \to^* v$ iff $t \to^* v$. This implies the claim because, especially, $e = v \sqcup e$ implies $e \to^* v$ since $v \sqcup e \to^* v$. The according proof is by induction on the equational derivation to obtain $s = t$ from $P \cup PL(P)$.
Base cases, $s = t \in P \cup PL(P)$. For (idem), we have $x \sqcup x \to^* x$. For (comm), v must be either x or y and we can derive both from $x \sqcup y$ and $y \sqcup x$. The analogue argument holds for (assoc), with respect to the derivability of x, y, z. For (bot), we have $\bot \sqcup x \to x$. Finally, if $s = t$ is a program rule $l = r$ or $l = r \sqcup l$ we have by construction $l \to^* r$.
Inductive cases, let the claim hold for all equations in a set Γ derivable from $P \cup PL(P)$. Then it also holds for an equation $s = t$ derivable from Γ by applying one of the rules We distinguish the rule applied to obtain the equation.
(1)-(3) Reflexivity, Symmetry, Transitivity: immediate from reflexivity, symmetry and transitivity of iff.
(4) Congruence, there exist substitutions $\sigma, \sigma' : X \to \mathcal{T}_{\Sigma(Y)}$ such that for each $x \in X$ the equation $\forall Y.\sigma(x) = \sigma'(x)$ is in Γ and there is a t' such that $s = \sigma(t')$ and $t = \sigma'(t')$: We show the claim by induction on the number n of variable occurrences in t'. For the base case $n = 0$ the claim is immediate by reflexivity of $\to^*$. For the inductive step let the claim hold for all terms with n or less variables, $|\mathcal{V}ar(t')| = n + 1$ and x a variable in $\mathcal{V}ar(t')$. Then we can construct a context C by replacing one occurrence of x by $\Box$ and get $t = C[\sigma(x)]$. As the induction hypothesis yields $\sigma(x) \to^* v$ iff $\sigma'(x) \to^* v$, the claim holds by Lemma 1.
(5) Substitutivity, there exists a substitution σ and terms u, u' such that $s = \sigma(u), t = \sigma(u')$ and $\forall X.u = u' \in \Gamma$: the claim follows directly from the induction hypothesis and Lemma 1. □

5 Conclusion

We have applied order-sorted algebra to define a semantics for functional and functional-logic programs. It is based on a linearely ordered set of four sorts, the representation of programs as equations over an order-sorted signature and the sorts of variables allowed in a program. As main result we have modeled four import semantic notions in a uniform and succinct manner: call-by-value, call-by-name and call-by-need with run-time choice and call-time choice.

Regarding related work we have shown that our modeling of call-by-need with call-time choice corresponds to the classic setting [5]. That

With regard to run-time choice a recent investigation [10] has shown that in the presence of complex data structures a compositional run-time choice semantics does not correspond to standard (unsorted) rewriting, contrasting general assumption. The illustrative example employs the definition `f (S x) = x+x`. For this the expression `f (S (1⊔0))` rewrites to `1` among other possibilities. However, the expression `f (S 1 ⊔ S 0)` does not yield `1`, in contrast. As both arguments, `S (1⊔0)` and `S 1 ⊔ S 0`, represent the same constructor terms this can be seen as a breech with compositionality [10]. To improve matters Rodríguez-Hortalá proposes a *plural* semantics for run-time choice [10]. In our framework, the equality

`S (1⊔0)= S 1 ⊔ S 0` is established by law (hom). In contrast to plural semantics, however, we do not identify arbitrary products, e.g., the terms `C 0 1 ⊔ C 1 0` and `C (0⊔1) (1⊔0)` for a binary `C`. In the hierarchy containing known approaches to run-time choice [10] our approach is, thus, indeed a novel intermediate step.

Several extensions of the presented framework could be future work. Firstly, an introduction of (non-polymorphic) many-kinded sub-typing to our framework causes no problems. First positive investigations give reason to hope that exchanging partially ordered sort sets with quasi-ordered ones could also incorporate parametric polymorphism. A second extension could be the formulation and investigation of laws for encapsulated search [3, 7]. And, lastly, we plan to generalize the hitherto framework to an algebraic semantics, i.e., in such a way that a semantics is given not only to expressions over the programs but also to the single functions introduced within the scope of a program. Standard approaches to lattice completion seem to be a suitable means to obtain this goal.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems.* Cambridge University Press, 2003.
3. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
4. Joseph A. Goguen and José Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.
5. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artale jo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
6. H. Hußmann. *Nondeterministic Algebraic Specifications and Algebraic Programs.* Birkhäuser Verlag, 1993.
7. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
8. Ralph McKenzie, George McNulty, and Walter Taylor. *Algebras, Lattices, Varieties.* Wadsworth and Brooks, Monterey, California, 1987.
9. Avi Pfeffer. Ibal: A probabilistic rational programming language. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 733–740, 2001.
10. Juan Rodríguez-Hortalá. A hierarchy of semantics for non-deterministic term rewriting systems. In Hariharan, Mukund, and Vinay, editors, *Proc. of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, Leibniz International Proceedings in Informatics, 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. Recently, it has been demonstrated the effective application of logic programming to 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 main features of a term rewriting system like MAUDE, we aim at transforming DATALOG programs into efficient rewrite systems. The transformation is proved to be sound and complete. A prototype has been implemented and applied to some real-world DATALOG-based analyses. Experimental results show that solving a DATALOG query using rewriting logic is comparable to state-of-the-art DATALOG solvers.

1 Introduction

DATALOG [1] is a simple relational query language which permits to describe, in an intuitive way, complex interprocedural program analyses involving dynamically created objects. The main advantage of formulating data-flow analyses as a DATALOG query is that analyses that take hundreds of lines of code in a traditional language can be expressed in a few lines of DATALOG [2]. In real-world problems, the DATALOG rules encoding 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. An important number of optimization techniques for DATALOG has been designed and studied extensively in program analysis, logic programming, and deductive databases [3, 4].

The aim of this paper is to provide efficient DATALOG query answering in Rewriting Logic [5], a very general *logical* and *semantical framework* efficiently implemented in the high-level programming language MAUDE [6]. Our motivation for using Rewriting Logic is to overcome the difficulty to handle metaprogramming features such as reflection in traditional analysis frameworks [7]. Actually, tracking reflective methods invocations requires not just tracking object references through variables but actually tracking method values and method

* 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.

name strings. We consider it a challenge to investigate the interaction of static analysis with metaprogramming frameworks. An additional goal of this work is to evaluate if MAUDE is able to process a sizable number of equations that come from real-life problems, like those from static analysis problems.

In the related literature, the solution for a Datalog query is classically constructed following a bottom-up approach, thus not taking advantage of the information in the query until the model has been constructed [8]. On the contrary, the typical logic programming interpreter would produce the output in a top-down fashion by reasoning backwards from the query. Between these two extremes, there is a whole spectrum of evaluation strategies [4, 9, 10]. While bottom-up computation may be very inefficient, the top-down approach is prone to infinite computations. In this work, we follow a top-down approach equipped with a loop check in order to avoid infinite computations, as in [11].

Logic and functional programming are both instances of rule-based, declarative programming and 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]. 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 among the parameters of the original program [15]. However, one distinguished feature of DATALOG programs burdening 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 an input/output behavior among parameters was presented in [14]. 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 [14] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [14] 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, extra conditions, etc.) in our working scenario, i.e., heavy data load (sets of hundreds of facts) together with relatively few clauses encoding the analysis to perform, in this work we present an equation-based transformation that leads to efficient MAUDE programs.

In previous work [16], we developed a DATALOG query solving technique based on *Boolean Equation Systems* (BESs) [17]. Although the correspondence between answering a DATALOG query and solving a BES can be established naturally, the main limitation of this approach is in the difficulty to combine indexed and linked data structures in order to schedule suitable optimizations which ensure that only useful combination of facts are simultaneously considered. In this paper, we stay at a higher level in the sense that we transform a high-

level DATALOG program into another high-level MAUDE program. The goal is to take advantage of the flexibility and versatility of MAUDE in order to achieve scalability without losing declaratively.

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 a DATALOG program into a MAUDE program. In Section 3, we describe the transformation of the example. Section 4 formalizes the general process and prove 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.

2 A program analysis written as a DATALOG program

DATALOG is a relational language using declarative *clauses* to both describe and query a deductive database. A 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 [8].

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 $V \cup C$, $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, 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 $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$).

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*, where p_0 is not arithmetic. 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. Following logic programming terminology, a clause with empty body ($m = 0$) is called a *fact*. A clause with empty head and $m > 0$ is called a *query*, and $\square$ 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. The variables appearing in a query are called *output* variables.[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 $\mathcal{D} \equiv q \Rightarrow^{\theta_1}_{SLD} q_1 \Rightarrow^{\theta_2}_{SLD} \ldots \Rightarrow^{\theta_n}_{SLD} \square$, the answer computed by $\mathcal{D}$ is $\theta_1\theta_2 \ldots \theta_n$ restricted to the variables occurring in q.

[1] In the sequel of the paper, DATALOG programs are considered to be as defined in this section.

Let us now introduce the running DATALOG program example that we use along 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 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 the `a/2` predicate 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 a number of DATALOG facts complementing the above pointer analysis description for a particular object-oriented example program.

```
a(v1,v2).
a(v1,v3).
vP0(v2,h5).
vP0(v3,h4).
```

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. The solver should compute which are the variables in the analyzed program that can point to the object `h5`.

Similarly to [14], 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 [6, 18].

3 From DATALOG to MAUDE

As explained above, we are interested in computing by rewriting all answers for a given query. A naïve approach is to translate DATALOG clauses into MAUDE rules, similarly to [14], 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, this approach results in poor performance.

[2] Intuitively, `search` $t \rightarrow t'$ explores the whole rewriting space from the term t to any other terms that match t' [6].

In this section, we first formulate a suitable representation in MAUDE of the DATALOG computed answers. Then, we informally introduce the transformation by means of the running example. Section 4 formalizes the translation procedure and proves its correctness.

3.1 Answers 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 in MAUDE the variables and constants of the original DATALOG program, 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 .         --- Every Qid is a Constant
op v : Qid -> Variable [ctor] . --- v(q) is a Variable if q is a Qid
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 appearing in the body of DATALOG clauses is unique to the whole MAUDE program.

Having ground terms representing variables, we still lack a way to collect the answers for an output variable. 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 (`Cte = Cte`) whenever possible. This hierarchy allows us to simplify constraints as soon as possible and to improve performance. The resulting MAUDE program is as follows:

```
sorts Constraint EmptyConstraint NonEmptyConstraint TConstraint FConstraint .
subsort EmptyConstraint NonEmptyConstraint < Constraint .
subsort TConstraint FConstraint < EmptyConstraint .
```

```
op _=_ : Term Constant -> NonEmptyConstraint .
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 _,_ : NonEmptyConstraint TConstraint -> NonEmptyConstraint        [ditto] .
op _,_ : NonEmptyConstraint FConstraint -> FConstraint               [ditto] .
op _,_ : NonEmptyConstraint NonEmptyConstraint -> NonEmptyConstraint [ditto] .

var NEC           : NonEmptyConstraint .
var V             : Variable .
var Cte Cte1 Cte2 : Constant .

eq (Cte = Cte) = T .                 --- Simplification
eq (Cte1 = Cte2) = F [owise] .       --- Unsatisfiability
eq NEC,NEC = NEC .                   --- Idempotence
eq F,NEC = F .                       --- Zero element
eq F,F = F .                         --- Simplification
eq (V = Cte1),(V = Cte2) = 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 `NonEmptyConstraint` 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 not only a single answer, but all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and implement in MAUDE a new sort called `ConstraintSet`:

```
sorts ConstraintSet EmptyConstraintSet NonEmptyConstraintSet .
subsort EmptyConstraintSet NonEmptyConstraintSet < ConstraintSet .
subsort NonEmptyConstraint TConstraint < NonEmptyConstraintSet .
subsort FConstraint < EmptyConstraintSet .

op _;_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc comm id: F] .
op _;_ : NonEmptyConstraintSet ConstraintSet -> NonEmptyConstraintSet [assoc comm id: F] .

var NECS : NonEmptyConstraintSet .

eq NECS ; NECS = NECS .     --- Idempotence
```

It is easy to grasp the intuition behind the different sorts and subsort relations in the above fragment of MAUDE code. The operator `_;_` represents the disjunction of constraints. The associativity, commutativity and (the existence of an) identity element properties of `_;_` can be easily expressed by using `ACU` attributes in MAUDE, thus simplifying the equational specification and achieving better efficiency. We express the idempotence property of the operator `_;_` by a specific equation on variables from the `NonEmptyConstraintSet` subsort.

In order to incrementally add new constraints along the program execution, we define the composition operator `x` as follows:

```
op _x_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc] .

var CS            : ConstraintSet .
var NECS1 NECS2   : NonEmptyConstraintSet .
var NEC NEC1 NEC2 : NonEmptyConstraint .

eq F x CS = F .                                                  --- L-Zero element
eq CS x F = F .                                                  --- R-Zero element
eq F x F = F .                                                   --- Double-Zero
eq NEC1 x (NEC2 ; CS) = (NEC1 , NEC2) ; (NEC1 x CS) .            --- L-Distributive
eq (NEC ; NECS1) x NECS2 = (NEC x NECS2) ; (NECS1 x NECS2) . --- R-Distributive
```

In order to keep information consistent, some simplification equations are automatically applied. These equations make every inconsistent constraint collapse into a `F` value, and trivial constraints are simplified.

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* [19] 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 one equation representing 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`,[3] 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) .
```

[3] The `c` in `vPc1` and `vPc2` stands for *clause*.

The specification for the first clause `vPc1` is given by

```
eq vPc1(T1,T2) = vP0(T1,T2) .
```

The transformation for the second clause of the program, represented by `vPc2`, is a bit more elaborated since, first, it contains more than one subgoal, thus we need an auxiliary function to impose the execution order. Moreover, it contains an existentially quantified variable (not appearing in the head of the clause) that 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) = Cte) , C) ; CS, T1 T2) =
         (vP(Cte,T1 T2) x ((v(T1,T2) = Cte) , C)) ; vPc2s2(CS,T1 T2) .
eq vPc2s2(F,T1 T2) = F .
```

As one can observe, `vPc2` calls to `vPc2s2`, whose first argument represents the execution of the first subgoal and the second one 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 to compute the (partial) answers resulting from the resolution of `a(T1,v(T1,T2))` first to proceed. The use of the term `v(T1,T2)`, representing the existentially quantified variable `Var2` of the original DATALOG program, in the pattern of the specification 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 = Cte) , 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 previously defined 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 which 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, written in several lines for the sake of readability, shows the result of the reduction together with its type.

```
reduce in ANALYSIS : vP(v('v), v('h)) .
rewrites: 39 in 0ms cpu5 (0ms real) (  rewrites/second)
result NonEmptyConstraintSet:
  ((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 it was expected, four answers have been returned: the first two are obtained by the auxiliary function `vPc1`, whereas the other two are 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. Then, in Section 4.2 we prove the correctness and completeness of the transformation.

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 [8]. 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:

```
var T_{i,1} ... T_{i,ar_i} : Term .
eq p_i(T_{i,1}, ... ,T_{i,ar_i}) = (T_{i,1} = c_{i,1,1}, ... , T_{i,ar_i} = c_{i,1,ar_i}) ; ...
                                 ; (T_{i,1} = c_{i,|p_i|,1}, ... , T_{i,ar_i} = c_{i,|p_i|,ar_i}) .
```

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 $1 \leq i \leq m$ with non empty body, we have the following MAUDE piece of code:

```
var T_{i,1} ... T_{i,ar_i} : Term .
eq p_i(T_{i,1}, ... ,T_{i,ar_i}) = p_{i,1}(T_{i,1}, ... ,T_{i,ar_i}) ; ...
                                 ; p_{i,|p_i|}(T_{i,1},...,T_{i,ar_i}) .
```

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 when the body has just one subgoal. We define the function $\tau^p_{i,j,s}$ that returns the predicate symbol appearing in the s-th subgoal of the j-th clause defining the i-th predicate in the DATALOG program. For each clause having just a subgoal, we get the following transformation:

$$\mathtt{eq}\ \mathtt{p}_{i,j}(\tau^a_{i,j,0,1},\dots,\tau^a_{i,j,0,ar_i})\ \mathtt{=}\ \tau^p_{i,j,1}(\tau^a_{i,j,1,1},\dots,\tau^a_{i,j,1,ar_l})\ \mathtt{.}$$

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. In particular, 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.

Let us first introduce some definitions and functions that will be used in our transformation.

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 of the clause's body.*

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 in the first component a linked variable, and in the second component the list of positions where such a variable occurs in the body of the clause*[4].

Example 1. For example, given the DATALOG clause

$$C = \texttt{p(X1,X2) :- p1(X1,X3), p2(X3,X4), p3(X4,X2).}$$

we have that

$$\mathsf{linked}(C) = \texttt{[(X3,[1.2,2.1]),(X4,[2.2,3.1])]}$$

Now we define the notion of *relevant* linked variables for a given subgoal, namely the linked variables of a subgoal appearing also in some 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 [14], we are not marking the input/output positions of predicates, as required in more traditional transformations. We are just identifying which are the variables whose values must be propagated for evaluating 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, and in the rhs's of the auxiliary functions, the execution order of the successive subgoals is implictly controlled by passing the results of each subgoal as a parameter to the subsequent function.

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 $\mathsf{relevant}(k,\mathsf{linked}(\mathit{clause}(i,j)))$[5] and transformed into the corresponding MAUDE terms.

[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.

The first equation reduces the considered DATALOG predicate to a call to the first auxiliary function that calculates the (partial) answers for the second subgoal, 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.

```
eq p_{i,j}(τ^a_{i,j,0,1},...,τ^a_{i,j,0,ar_i}) = ps_{i,j,2}(τ^p_{i,j,1}(τ^a_{i,j,1,1},...,τ^a_{i,j,1,r}), τ^a_{i,j,0,1} ... τ^a_{i,j,0,ar_i}) .
```

where r is the arity of the predicate $\tau^p_{i,j,1}$. Then, for each auxiliary function, first we declare as many constants as relevant variables the given subgoal has. The left hand side of the equation 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, 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.

```
var C_1 ...C_N : Constant .
var NECS : NonEmptyConstraintSet .

eq ps_{i,j,s}(NECS, T_1...T_{ar_i}) = ps_{i,j,s+1}(ps'_{i,j,s}(NECS, T_1...T_{ar_i}), T_1...T_{ar_i}) .
eq ps_{i,j,s}(F , LL) = F .

eq ps'_{i,j,s}(((X_1=C_1,...,X_N=C_N, C) ; CS), T_1...T_{ar_i}) =
        ((τ^p_{i,j,s}(τ^v_{i,j,s,1},...,τ^v_{i,j,s,r})[X_1\C_1,...,X_N\C_N]) x (X_1=C_1,...,X_N=C_N, C)) ;
        ps'_{i,j,s}(CS, T_1...T_{ar_i}) .
eq ps'_{i,j,s}((T ; CS), T_1...T_{ar_i}) =
        τ^p_{i,j,s}(τ^v_{i,j,s,1},...,τ^v_{i,j,s,r}) ; ps'_{i,j,s}(CS, T_1...T_{ar_i}) .
eq ps'_{i,j,s}(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

```
eq ps_{i,j,g}(((X_1=C_1,...,X_N=C_N, C) ; CS) , T_1...T_{ar_i}) =
        ((τ^p_{i,j,g}(τ^v_{i,j,g,1},...,τ^v_{i,j,g,r})[X_1\C_1,...,X_N\C_N]) x (X_1=C_1,...,X_N=C_N, C)) ;
        ps_{i,j,g}(CS , T_1...T_{ar_i}) .
eq ps_{i,j,g}((T ; CS) , T_1...T_{ar_i}) =
        τ^p_{i,j,g}(τ^v_{i,j,g,1},...,τ^v_{i,j,g,r}) ; ps_{i,j,g}(CS , T_1...T_{ar_i}) .
eq ps_{i,j,g}(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. The MAUDE encoding of the query is $\texttt{q}(\tau^q_1, \ldots, \tau^q_n)$ 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 specifying static analyses 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. 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 ($\texttt{C}_1$ `=` $\texttt{Cte}_1$`,...,`$\texttt{C}_m$ `=` $\texttt{Cte}_m$), and V be a list of variables. We write $\texttt{C}_i|_V$ to the restriction of the constraint $\texttt{C}_i$ to the variables in 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 a 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 $\theta = \{X_1/C_1, X_2/C_2, \ldots, X_n/C_n\}$, given by: `C` is equivalent to θ iff ($\texttt{C} \Leftrightarrow \hat{\theta}$), where $\hat{\theta}$ is the equational representation of θ. By abuse, given a disjunction `CS` of equational constraints and a set of idempotent substitutions ($\Theta = \cup_{i=1}^n \theta_i$), we define $\Theta \equiv$ `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 [21].

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 $\mathcal{T}_g(q)$ be the corresponding, transformed input term. Let Θ be the set of computed answers of P for the query q, and $\texttt{CS}|_{var(\mathcal{T}_g(q))}$ 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 implementing the transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [20]; then, we evaluate the performance of our prototype by comparing it to three state-of-the-art DATALOG solvers.

All 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. the previous rewriting-based versions

We implemented several versions of transformation from DATALOG programs to MAUDE programs before the one presented in this paper [20]. The first attempt consisted on a transformation based on a one-to-one correspondence of

DATALOG rules and MAUDE conditional rules. Then we tried to get rid of all the non-determinism introduced by conditional equations and rules. 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 MAUDE [6]. 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 won't reduce it but just replaces it with its normal form, saving a great amount 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	?	6.00	0.67	0.02
150	150	?	20.59	2.23	0.04
200	200	?	48.48	6.11	0.10
403	399	?	382.16	77.33	0.47
807	1669	?	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 backtracking operations associated to the non-determinism of rules and conditional equations and rules penalize the naïve version. We can also observe that using memoization enables 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 stated purpose, namely program analysis, and that it is likely to provide a useful way forward, 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 [6], DATALOG 1.4 [7], and IRIS 0.58 [8]. 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. This

[6] `http://xsb.sourceforge.net`
[7] `http://datalog.sourceforge.net`
[8] `http://iris-reasoner.sourceforge.net`

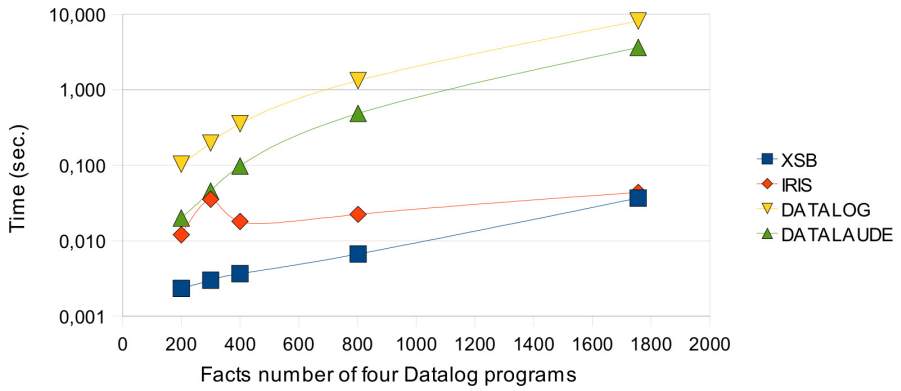


Fig. 1. Average resolution times of four Datalog solvers (logarithmic time).

means that initialization operations, like loading and compilation, are not taken into account in the results. Although being slower than XSB or IRIS, from our figures we can conclude that MAUDE behaves similarly to optimized deductive database systems, like DATALOG 1.4, which is implemented in C. This validates that MAUDE is able to process, under a good transformation such as the equational implementation extended with memoization, a large number of equations extracted from real programs in the context of static program analysis. Our rewrite system could be even more enhanced with the incorporation of efficient BDD representation [22] of the input data.

6 Conclusion

In this work, we have defined and implemented an efficient 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 compared the implementation to standard DATALOG solvers. We evaluated that MAUDE was able to process a sizable number of equations, that come from real-life problems, like those from the static analysis of programs.

As a future work, we plan to use a more compact representation of the facts, such as BDDs, in order to minimize the significant loading time and size of the manipulated term in the rewriting system. We also plan to explore the impact of more sophisticated optimization techniques like tail-recursion or memoization (at the logical level) and other specific DATALOG optimizations [23]. Our final goal is to explore the impact of using the metalevel capabilities of rewriting logic for the analysis of object-oriented programs that include metaprogramming features such as reflection.

References

1. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Vol. I and II, The New Technologies. Computer Science Press (1989)
2. Whaley, J., Avots, D., Carbin, M., Lam, M.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Proc. of 3rd Asian Symp. on Programming Lang. and Systems (APLAS'05). Vol. 3780 of LNCS, Springer-Verlag (2005) 97-118

3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1995)
4. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer-Verlag (1990)
5. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1) (1992) 73–155
6. Clavel, M., Durán, F., Ejer, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Vol. 4350 of LNCS. Springer-Verlag (2007)
7. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Lang. and Systems (APLAS'05). (2005) 139–160
8. Leeuwen, J., ed.: Formal Models and Semantics. Volume B. Elsevier, The MIT Press (1990)
9. 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 Symp. on Principles of Database Systems (PODS'86), ACM Press (1986) 1–15
10. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Proc. of the 1st Int'l Conf. on Expert Database Systems (EDS'86). (1986) 253–267
11. Sagonas, K.F., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: Proc. of the 1994 ACM SIGMOD Int'l Conf. on Management of Data, ACM Press (1994) 442–453
12. Emden, M., Lloyd, J.: A logical reconstruction of Prolog II. Journal on Logic Programming **1** (1984)
13. Marchiori, M.: Logic Programs as Term Rewriting Systems. In: Proc. of the 4th Int'l Conf. on Algebraic and Logic Programming (ALP'94. Vol. 850 of LNCS., Springer-Verlag (1994) 223– 241
14. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Proc. of the 16th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'06). Vol. 4407 of LNCS., Springer-Verlag (2007) 177–193
15. Reddy, U.: Transformation of Logic Programs into Functional Programs. In: Proc. of the Symp. on Logic Programming (SLP'84), IEEE Computer Society Press (1984) 187–197
16. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Using Datalog and Boolean Equation Systems for Program Analysis. In: Proc. of the 13th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS'08). Vol. 5596 of LNCS., Springer-Verlag (2009) 215–231
17. Andersen, H.R.: Model checking and boolean graphs. Theoretical Computer Science **126**(1) (1994) 3–30
18. Hill, P.M., Lloyd, J.W.: Analysis of Meta-Programs. In: Proc. of the First Int'l Workshop on Meta-Programming in Logic (META'88). (1988) 23–51
19. Marchiori, M.: Unravelings and ultra-properties. In: Proc. of the 5th Int'l Conf. on Algebraic and Logic Programming (ALP'96). Vol. 1039 of LNCS., Springer-Verlag (1996) 107–121
20. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Implementing Datalog in Maude. In: Proc. of the IX Jornadas sobre Programación y Lenguajes (PROLE'09) and I Taller de Programación Funcional (TPF'09). (September 2009) To appear.
21. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Defining Datalog in Rewriting Logic. Technical Report DSIC, Universidad Politécnica de Valencia. `http://www.dsic.upv.es/~villanue/pub/AFJV09-techrep.pdf`

22. Zantema, H., Pol, J.: A rewriting approach to binary decision diagrams. Journal of Logic and Algebraic Programming **49** (2001) 61–86
23. Liu, Y., Stoller, S.: From Datalog Rules to Efficient Programs with Time and Space Guarantees. ACM Transactions on Programming Languages and Systems (2009) To appear.