RWTH Aachen

Department of Computer Science
Technical Report

Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

Peter Schneider-Kamp

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-17

RWTH Aachen · Department of Computer Science · December 2008

Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jan Peter Schneider-Kamp
aus
Geldern

Berichter: Prof. Dr. Jürgen Giesl
Prof. Michael Codish

Tag der mündlichen Prüfung: 8. Dezember 2008

Jan Peter Schneider-Kamp
Lehr- und Forschungsgebiet Informatik 2
`psk@informatik.rwth-aachen.de`

Aachener Informatik Bericht AIB-2008-17

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

The most fundamental decision problem in computer science is the *halting problem*, i.e., given a description of a program and an input, decide whether the program terminates after finitely many steps or runs forever on that input. While Turing showed this problem to be undecidable in general, developing static analysis techniques that can automatically prove termination for many pairs of programs and inputs is of great practical interest.

This is true in particular for *logic programming*, as the inherent lack of direction in the computation virtually guarantees that any non-trivial program terminates only for certain classes of inputs. Thus, termination of logic programs is widely studied and significant advances have been made during the last decades. Nowadays, there are fully-automated tools that try to prove termination of a given logic program w.r.t. a given class of inputs. Nevertheless, there still remain many logic programs that cannot be handled by any current termination technique for logic programs that is amenable to automation.

Another area where termination has been studied even more intensively is term rewriting. This basic computation principle underlies the evaluation mechanism of many programming languages. Significant advances towards powerful automatable termination techniques during the last decade have yielded a plethora of powerful tools for proving termination automatically.

In this thesis, we show that techniques developed for proving termination of term rewriting can successfully be adapted and applied to analyze logic programs. The new techniques developed significantly extend the applicability and the power of automated termination analysis for logic programs. The work presented here ranges from adapting techniques to work directly on logic programs to transformations from logic programs to a specialized version of term rewriting. On the logic programming side we also present a new pre-processing approach to handle logic programs with cuts. On the term rewriting side we show how to search for certain popular classes of well-founded orders on terms more efficiently by encoding the search into satisfiability problems of propositional logic.

The contributions developed in this thesis are implemented in tools for automated termination analysis – mostly in our fully automated termination prover AProVE. The significance of our results is demonstrated by the fact that AProVE has reached the highest score both for term rewriting and logic programming at the annual international Termination Competitions in all years since 2004, where the leading automated tools try to analyze termination of programs from different areas of computer science.

Acknowledgments

First and foremost I would like to thank Jürgen Giesl for being a challenging and demanding and at the same time trusting and supportive supervisor, employer, colleague, and mentor. His door was always open, there was always time for discussion, and his optimism always kept me on the track.

I am equally thankful towards my colleague and friend René Thiemann. His intuition and craftsmanship in formal proofs gave rise to countless fruitful discussions, some frustrating, most very productive, but all mutually rewarding. It was my greatest pleasure to work in a team with him and Jürgen.

Many thanks go to Mike Codish for agreeing to be the second supervisor of my thesis and for many very satisfying discussions – ranging from face to face over a beer to long Skype sessions where he made me aware that bird noise is actually nice. Without his initiative in early 2006, there would be no second part of this thesis.

I am grateful to Alexander Serebrenik for introducing me to the world of logic programming. The ideas we scribbled on a paper napkin outside a Valencian café back in 2003 have been the starting point for a fruitful cooperation that has inspired most of the first part of this thesis.

My thanks belong to my office mate Stephan Swiderski for exchanging all those ideas and providing me with his sometimes unconventional but often enlightening perspective, and to my colleague Carsten Fuhs for co-developing the SAT framework and, given a sufficient supply of coffee, finding the most obscure of mistakes.

Special thanks go to Thomas Ströder for great work on the logic programming implementation, and to Carsten Fuhs, Carsten Otto, René Thiemann, Stephan Swiderski, and Erika Ábrahám for proof-reading preliminary versions of this thesis.

Sincere thanks go to all other co-authors of joint papers for fruitful cooperations, to all the other members of the AProVE team for all their contributions and many a dark Guinness, and to the colleagues of the MOVES group for a nice atmosphere before and after 2 pm. I also enjoyed the presence of and the cooperation with our international guests Manh Thang Nguyen, Raúl Gutiérrez, and Beatriz Alarcón.

Last but not least, I am much obliged to my parents and Silke, who had to put up with long and irregular work hours, frequent travels, and incomprehensible research topics, yet still supported me in so many ways.

Peter Schneider-Kamp

Contents

1. Introduction

The most fundamental decision problem in computer science is the *halting problem*, i.e., given a program and an input, decide whether the program terminates after finitely many steps or whether it runs forever on that input. While [Tur36] showed this problem to be undecidable in general, developing static analysis techniques that can automatically prove termination for many pairs of programs and inputs is of great practical interest.

Proving termination is essential for a number of verification techniques. For example, both theorem proving [BM79] and Knuth-Bendix completion [KB70] require frequent termination proofs for their correct operation (see [MV06, WSW06, BKN07] for recent work on applying termination analysis in these areas). In process verification, liveness problems can often be reduced to termination problems [GZ03], and Microsoft uses termination analysis to statically verify their device drivers [CPR05].

Termination analysis is of particular interest in logic programming as the inherent lack of direction in the computation virtually guarantees that any non-trivial program terminates only for certain classes of inputs. Thus, termination of logic programs is widely studied (see [DD94] for an overview and [BCG$^+$07, CLS05, CLSS06, DS02, LMS03, MR03, MS07, ND05, ND07, NGSD08, SD05a, Sma04] for more recent work) and significant advances have been made during the last decades. Nowadays, there are fully-automated tools [LSS97, MB05, TGC02, SD03a, GST06, ND07, OCM00] that try to prove termination of a given logic program w.r.t. a given class of inputs. Nevertheless, there still remain many natural logic programs that cannot be shown terminating by any of these tools.

Another area where termination has been studied even more intensively is term rewriting (see for example [Der87, Zan95, AG00, GTSF06, EWZ06]). This basic computation principle underlies the evaluation mechanism of many programming languages and, indeed, transformations from functional programs and logic programs to term rewriting have yielded powerful termination provers [Ohl01, GSST06, SGST07].

Significant advances towards powerful automatable termination techniques during the last decade [AG00, GTS05a, HM05a, EWZ06] have yielded a plethora of powerful tools for proving termination of term rewriting automatically [HM07, CMU03, End06, Wal04, Zan05, FGK03, BR03, Kop06, Wul06, Kor07, AGIL07, GST06].

The state of the art in automated termination proving is assessed through the annual international Termination Competition [MZ07], where the leading automated tools try to analyze termination of programs from different areas of computer science.

Among these tools, our fully automated termination prover AProVE [GST06] has reached the highest score for term rewriting in the years 2004, 2005, 2006, and 2007.[1] Thanks to the contributions of this thesis and our work on analyzing termination of Haskell programs [GSST06], AProVE has also reached the highest scores for both logic programming and functional programming.

Termination of Logic Programs

In the first part of this thesis, we show that techniques developed for proving termination of term rewriting can successfully be adapted and applied to analyze logic programs. The newly developed techniques significantly extend the applicability and the power of automated termination analysis for logic programs. The work presented here ranges from adapting techniques to work directly on logic programs to transformations from logic programs to a specialized version of term rewriting. On the logic programming side we also present a new pre-processing approach to handle logic programs with cuts.

To be more precise, we make the following contributions:

(i) Transformational approaches transform the logic program into a term rewrite system and then prove termination of the term rewrite system. Existing approaches are rather limited in their applicability and not very powerful.

    We present a novel transformation from logic programs to term rewriting which is applicable for all definite logic programs and show that it is much more powerful than existing transformations.

(ii) Direct approaches work directly on logic programs to prove its termination. They are often efficient, but not very powerful as either they lack modularity or they lack a constraint-based approach. Here, constraint-based means that one does not fix the well-founded order beforehand but instead generates a set of symbolic constraints, which then can be solved by a constraint solver.

    We show how to use two popular techniques from term rewriting (dependency pairs and polynomial interpretations) to obtain a constraint-based modular approach that is more powerful than existing direct approaches.

(iii) Direct and transformational approaches are incomparable w.r.t. efficiency and power. Therefore, the best would be a combination of the two.

    We introduce a modular variant of our transformation from (i) as a modular technique in (ii) to obtain a combined approach that subsumes (i) and (ii) when they are used separately.

[1]The next competition will take place no earlier than November 2008.

(iv) The approaches in (i) – (iii) are all limited to universal termination of definite logic programs. They cannot analyze programs where termination depends on cuts or negation-as-failure and they cannot show existential termination, i.e., that the first solution will be found in finite time.

We present a novel pre-processing based on symbolic execution to handle logic programs with cuts. This allows also for the handling of negation-as-failure and existential termination as these can be simulated using cuts and meta-programming.

Automating Search by Encoding to SAT

Analyzing the termination of programs by techniques from term rewriting presents a serious challenge to the efficiency of the search algorithms for well-founded orders.

Since 2006, several publications [CLS06, CSL$^+$06, HW06, EWZ06, FGM$^+$07, STA$^+$07, ZM07] have illustrated the huge potential in applying SAT solvers for various types of termination problems for term rewrite systems (TRSs).

In the second part of this thesis we show how to search for certain popular classes of well-founded orders on terms more efficiently by encoding the search into satisfiability problems of propositional logic.

More precisely, our contributions to this part are:

(v) Recursive path orders are a popular class of orders used in the termination analysis of term rewriting. They are a useful alternative to polynomial orders (cf. our contributions in [FGM$^+$07]) as they are incomparable w.r.t. power and w.r.t. efficiency. And indeed, as described in Section 7.5, the use of a recursive path order together with Contribution (vi) is essential for some termination proofs of logic programs.

We show how to encode the search for recursive path orders into satisfiability of propositional logic. In particular, we show how to encode the precedence-based comparison of terms common to all syntactic path orders, the multiset extension of the ordering used for comparing arguments, and the lexicographic extension of the order w.r.t. arbitrary permutations of the arguments. This yields an implementation that is orders of magnitudes faster than existing dedicated algorithms.

(vi) Direct termination proofs for term rewriting using well-founded orders (such as recursive path orders) are rather weak in practice. Therefore, virtually all tools for termination analysis of term rewriting combine orders with dependency pairs and, thus, need to search also for so-called argument filters.

We show how to extend the encoding of Contribution (v) to include argument filters. In this way, we obtain a powerful implementation that is again orders of magnitude faster than existing dedicated algorithms.

Implementation & Evaluation

The theoretical contributions developed in this thesis have been implemented in tools for automated termination analysis – mostly in our fully automated termination prover AProVE and partly in Polytool [ND07]. Altogether, the author of this thesis has contributed approx. 110,000 lines of Java code to the AProVE sources. The author has also performed extensive experiments to evaluate these implementations.

More precisely, the practical contributions of this thesis are:

(vii) Contributions (i) and (iv) have been implemented as the logic programming frontend of our termination prover AProVE.

We evaluated our implementation on a set of 296 logic programs (cf. Section 7.4) and compared its performance to that of [ND05, TGC02, MB05, OCM00]. Furthermore, we showed that the heuristics presented in Section 3.4 are successful in practice.

(viii) For Contributions (ii) and (iii), a prototypical implementation of the techniques from [NGSD08] in Polytool by Manh Thang Nguyen shows their potential. This implementation uses our implementation of our results from [FGM+07] in AProVE to search for polynomial orders.

We evaluated this implementation successfully on the same example set that we used for Contribution (vii).

(ix) For [CSL+06, FGM+07] we implemented a framework for SAT encodings in our termination prover AProVE. Building on this framework, we implemented Contributions (v) and (vi) in a modular way such that instead of searching for recursive path orders and argument filters, we can search for restrictions like, for instance, the lexicographic path order without argument filters or the multiset path order with argument filters.

We performed extensive experiments with full recursive path orders with and without argument filters as well as with 14 restricted classes of orders. For each configuration we compared the performance of our new SAT-based implementation to that of existing dedicated algorithms, and we observed a decrease in runtime by orders of magnitude.

The significance of our practical contributions is demonstrated by the fact that AProVE has reached the highest score both for term rewriting and logic programming at the annual international Termination Competitions in the years 2004 – 2007. Also, Polytool has come in at a respectable second place for logic programming in 2007.

Published vs. New Contributions

Preliminary versions of some parts of this thesis have already been published by the author in 20 articles in international journals or conference proceedings [SGST, TZGS08, GTSF06, AEF$^+$08, FNO$^+$08, FGM$^+$08, TGS08, NGSD08, STA$^+$07, GTSS07, FGM$^+$07, CSL$^+$06, GSST06, SGST07, GST06, GTS05b, GTS05a, TGS04, GTSF04, GTSF03].

However, this thesis contains numerous substantial novel contributions that have not been published yet:

- Contribution (ii) is a novel unpublished contribution that extends the technique of [NGSD08]. While [NGSD08] is based on the dependency pair approach [AG00], we show how to adapt it to the more recent dependency pair framework [GTS05a].
- Contribution (iii) is novel and unpublished.
- Contribution (iv) is novel and unpublished.
- Contribution (i) extends the 15-page conference paper [SGST07] by approx. 30 pages containing detailed algorithms and heuristics required for a successful implementation, a formal proof for subsumption of previous transformational approaches, and more extensive and detailed experiments.

  An extended version mostly corresponding to Contribution (i) was submitted to *ACM Transactions on Computational Logic* in February 2008 and has been accepted without revision in July 2008 [SGST] due to very positive reviews which considered this contribution to be *"a milestone in termination analysis of logic programs"*.
- Contributions (v) and (vi) are mostly a restructured version of two published papers [CSL$^+$06, STA$^+$07]. Additionally, the contributions of [CSL$^+$06] have been extended from lexicographic path orders to recursive path orders as used in [STA$^+$07].

Structure of the Thesis

In Chapter 2 we establish some basic notions about terms, logic programming, and term rewriting that are widely used throughout this thesis. Further notions or differences to standard definitions are given in the preliminary sections of the respective chapters.

The first part of this thesis about termination of logic programs is divided into three chapters. In Chapter 3 we present Contribution (i), i.e., the new transformation from logic programs to term rewriting. Contributions (ii) and (iii) regarding our direct approach and the combination of our direct and our transformational approaches are presented in Chapter 4. Contribution (iv), i.e., our new pre-processing approach for logic programs with cuts (and, thus, negation-as-failure, existential termination, and meta-programming), constitutes Chapter 5.

The second part of this thesis about SAT encodings for efficient automation of search problems is divided into two chapters. In Chapter 6 we present Contribution (v), i.e., our SAT encoding for recursive path orders. This encoding is extended to recursive path orders combined with dependency pairs and argument filters in Chapter 7, which contains Contribution (vi).

Details about Contribution (vii) are given in Sections 3.6 and Section 5.6. Some details about Contribution (viii) are found in Section 4.4. The implementation and the experiments for Contribution (ix) are described in Sections 6.3 and 7.4.

Finally, we conclude in Chapter 8, where we also show how the contributions of this thesis can be combined to obtain a powerful termination analyzer for logic programs.

2. Preliminaries

In this chapter we establish the basic definitions for termination, term rewriting, and logic programming used throughout this thesis. Special notations and formalisms relevant to only one chapter are introduced in the preliminaries section of the respective chapter.

Abstract Reductions, Termination

To model computation steps of a program, we use the concept of abstract reductions. An *abstract reduction system* is a pair $(\mathcal{A}, \rightarrow)$ where $\rightarrow$ is a binary relation over $\mathcal{A}$, i.e., $\rightarrow \subseteq \mathcal{A} \times \mathcal{A}$. For the sake of brevity, instead of $(a, b) \in \rightarrow$ we write $a \rightarrow b$.

Now, we model a computation by defining $\mathcal{A}$ to be a superset of the program states and $\rightarrow$ to be the transition relation from a certain program state to its successor state.

For a given state $a \in \mathcal{A}$, we say that that $\rightarrow$ is *terminating* w.r.t. a if and only if there is no infinite reduction $a \rightarrow a_0 \rightarrow a_1 \rightarrow \ldots$ Furthermore, we say that $\rightarrow$ is terminating if, and only if, it is terminating for all $a \in \mathcal{A}$.

In the remainder of this thesis, the set of program states $\mathcal{A}$ and the relation $\rightarrow$ will typically be either terms and a suitable term rewriting relation or queries and the left-to-right resolution used in logic programs.

Terms, Atoms, Substitutions

Both term rewriting and logic programming rely on the basic concepts of (possibly infinite) terms and substitutions built over sets of function symbols and variables. For logic programming, we additionally need the notion of atoms built from predicates and terms. This leads to the following formal definition of sets of function and predicate symbols.

Definition 2.1 (Signature)**.** *A* signature *is a pair* (Σ, Δ) *where* Σ *and* Δ *are finite sets of function respectively predicate symbols. If* $\Delta = \varnothing$*, we often just write* Σ *instead of* $(\Sigma, \varnothing)$*.*

Each symbol in $f \in \Sigma \cup \Delta$ *has an* arity $n \geq 0$ *and we often write* f/n *instead of* f *to denote that* f *has arity* n*.*

In the following, we always assume that Σ contains at least one constant $f/0$. This is not a restriction, since enriching the signature by a fresh constant does not change the termination behavior. This assumption is useful to ensure that we can always build finite ground terms over a given signature.

Definition 2.2 (Terms). *A* term *over Σ is a tree where every node is labeled with a function symbol from Σ or with a variable from $\mathcal{V} = \{X, Y, \ldots\}$. Every inner node with n children is labeled with some $f/n \in \Sigma$, while leaves are labeled with a variable $X \in \mathcal{V}$ or with $f/0 \in \Sigma$. We write $f(t_1, \ldots, t_n)$ for the term t with* root *f (denoted $root(t) = f$) and direct subtrees $t_1, \ldots, t_n$. A term t is called* finite *if all paths in the tree t are finite, otherwise it is* infinite. *A term is* rational *if it only contains finitely many subterms. The sets of all finite terms, all rational terms, and all (possibly infinite) terms over Σ are denoted by $\mathcal{T}(\Sigma, \mathcal{V})$, $\mathcal{T}^{rat}(\Sigma, \mathcal{V})$, and $\mathcal{T}^{\infty}(\Sigma, \mathcal{V})$, respectively. If $\vec{t}$ is the sequence $t_1, \ldots, t_n$, then $\vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V})$ means that $t_i \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$ for all i. $\vec{\mathcal{T}}(\Sigma, \mathcal{V})$ is defined analogously. We write $\mathcal{T}(\Sigma)$ instead of $\mathcal{T}(\Sigma, \varnothing)$ to denote* ground terms, *i.e., variable-free terms.*

Finally, for any set of variables $\mathcal{V}' \subseteq \mathcal{V}$ and any term $t \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$, let $\mathcal{V}'(t)$ be the set of all variables from $\mathcal{V}'$ occurring in t, i.e., $\mathcal{V}'(X) = \{X\}$ for $X \in \mathcal{V}'$, $\mathcal{V}'(X) = \varnothing$ for $X \notin \mathcal{V}'$, and $\mathcal{V}'(f(t_1, \ldots, t_n)) = \bigcup_{1 \leq i \leq n} \mathcal{V}'(t_i)$.

Given the above definition, the formal definition for atoms is straightforward.

Definition 2.3 (Atom). *An* atom *over (Σ, Δ) is a tree $p(t_1, \ldots, t_n)$, where $p/n \in \Delta$ and $t_1, \ldots, t_n \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$. $\mathcal{A}^{\infty}(\Sigma, \Delta, \mathcal{V})$ is the set of atoms and $\mathcal{A}^{rat}(\Sigma, \Delta, \mathcal{V})$ (and $\mathcal{A}(\Sigma, \Delta, \mathcal{V})$, respectively) are the atoms $p(t_1, \ldots, t_n)$ where $t_i \in \mathcal{T}^{rat}(\Sigma, \mathcal{V})$ (and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$, respectively) for all i. We write $\mathcal{A}(\Sigma, \Delta)$ instead of $\mathcal{A}(\Sigma, \Delta, \varnothing)$.*

To address or replace certain subterms, we introduce the notion of a position. The intuition is that a position describes a path from the root of the term to the subterm.

Definition 2.4 (Position). *For a term t we define the set of* positions *$Occ(t)$ as the least subset of $\mathbb{N}^*$ such that $\varepsilon \in Occ(t)$ and $i\,pos \in Occ(t)$, if $t = f(t_1, \ldots, t_n)$, $1 \leq i \leq n$, and $pos \in Occ(t_i)$. We denote the subterm of t at position pos as $t|_{pos}$ where $t|_{\varepsilon} = t$ and $f(t_1, \ldots, t_n)|_{i\,pos} = t_i|_{pos}$. For a position $pos \in Occ(t)$ we denote the replacement of $t|_{pos}$ in t with a term s by $t[s]_{pos}$ where $t[s]_{\varepsilon} = s$ and $f(t_1, \ldots, t_n)[s]_{i\,pos} = f(t_1, \ldots, t_i[s]_{pos}, \ldots, t_n)$.*

A common operation on terms is the instantiation of a term by a substitution, i.e., the replacement of all occurrences of certain variables by certain terms.

Definition 2.5 (Substitution). *A* substitution *is a function $\theta : \mathcal{V} \to \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$. We define the* domain *of a substitution θ as $Dom(\theta) = \{X \in \mathcal{V} \mid \theta(X) \neq X\}$ and similarly the* range *of a substitution θ as $Range(\theta) = \{\theta(X) \mid X \in Dom(\theta)\}$.*

By abuse of notation, we extend substitutions homomorphically to work on terms, atoms, etc. by applying them to all variables occurring in these expressions. If θ is a variable renaming (i.e., a one-to-one correspondence on $\mathcal{V}$), then $\theta(t)$ is a variant *of t.*

Instead of $\theta(X)$ we often write $X\theta$. We write $\theta\sigma$ to denote that the application of θ is followed by the application of σ, i.e., $X\theta\sigma = \sigma(\theta(X))$. For $Dom(\theta) = \{X_1, \ldots, X_n\}$ with $X_i\theta = t_i$, we often write $\{X_1/t_1, \ldots, X_n/t_n\}$.

Logic Programming

A *clause* c is a formula $H \leftarrow B_1, \dots, B_k.$ with $k \geq 0$ and $H, B_i \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$. H is called the *head* of the clause c and $B_1, \dots, B_k$ is called its *body*. A finite set of clauses $\mathcal{P} = \{c_1 \dots c_n\}$ over (Σ, Δ) 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 "$B_1, \dots, B_k$". The empty query is denoted $\square$.

To define the evaluation mechanism of resolution, we need the concept of a (most general) unifier of two atoms or terms.

Definition 2.6 (Most General Unifier)**.** *A substitution θ is a* unifier *of two atoms or terms s and t if and only if $s\theta = t\theta$. We write $s \sim t$ if there is a unifier of s and t. We call θ a* most general unifier (mgu) *of s and t if and only if θ is a unifier of s and t and for every unifiers σ of s and t there is a substitution μ such that $\sigma = \theta\mu$.*

We briefly present the procedural semantics of logic programs based on SLD-resolution using the left-to-right selection rule implemented by most Prolog systems. More details on logic programming can be found in [Apt97], for example.

Definition 2.7 (Derivation)**.** *Let Q be a query $A_1, \dots, A_m$, let c be a clause $H \leftarrow B_1, \dots, B_k$. Then Q' is a* resolvent *of Q and c using θ (denoted $Q \vdash_{c,\theta} Q'$) if θ is the mgu*[1] *of A_1 and H, and $Q' = (B_1, \dots, B_k, A_2, \dots, A_m)\theta$. We call A_1 the* selected atom.

A derivation *of a program $\mathcal{P}$ and Q is a possibly infinite sequence $Q_0, Q_1, \dots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$ for some substitution θ_{i+1} and some fresh variant c_{i+1} of a clause of $\mathcal{P}$. For a derivation $Q_0, \dots, Q_n$ as above, we also write $Q_0 \vdash^n_{\mathcal{P},\theta_1 \dots \theta_n} Q_n$ or $Q_0 \vdash^n_{\mathcal{P}} Q_n$, and we also write $Q_i \vdash_{\mathcal{P}} Q_{i+1}$ for $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$. The query Q* terminates *for $\mathcal{P}$ if all derivations of $\mathcal{P}$ and Q are finite, i.e., if $\vdash_{\mathcal{P}}$ is terminating for Q.*

If we restrict the substitutions used in derivations to functions from $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$, i.e., to substitutions for which the range contains only finite terms, the above notion of derivation corresponds to SLD-resolution using the left-to-right selection rule typically found in logic programming. Without this restriction, the above notion of derivation coincides with logic programming without an occur check [Col82] as implemented in common Prolog systems such as SICStus or SWI.

Finally, we need the concept of answer substitutions that define the results of the computation of a logic program.

Definition 2.8 (Answer Set)**.** *The* answer set *$Answer(Q, \mathcal{P})$ for a logic program $\mathcal{P}$ and a query Q is defined as the set of all substitutions $\theta|_{\mathcal{V}(Q)}$ such that $Q \vdash^n_{\mathcal{P},\theta} \square$ for some $n \in \mathbb{N}$.*

[1]Note that for finite sets of *rational* atoms or terms, unification is decidable, the mgu is unique modulo renaming, and it is a substitution with *rational* terms [Hue76].

Example 2.9. Consider the predicate $\mathsf{double}/2$ that is true when the second argument is twice as large as the first one. The following two clauses constitute a logic program $\mathcal{P}$ over $(\{\mathsf{0}, \mathsf{s}\}, \{\mathsf{double}\})$ where $\mathsf{0}$ and the successor function s are used to represent natural numbers:

$$\mathsf{double}(\mathsf{0}, \mathsf{0}). \tag{1}$$
$$\mathsf{double}(s(X), s(s(Y)) \leftarrow \mathsf{double}(X, Y). \tag{2}$$

For the query $Q = \mathsf{double}(\mathsf{s}(0), Z)$ we obtain the following derivation:

$$\mathsf{double}(\mathsf{s}(\mathsf{0}), Z) \vdash_{(2),\{X/0,Z/\mathsf{s}(\mathsf{s}(Y)\}} \mathsf{double}(\mathsf{0}, Y) \vdash_{(1),\{Y/0\}} \square$$

The set of answer substitutions is $\mathit{Answer}(Q, \mathcal{P}) = \{\{Z/\mathsf{s}(\mathsf{s}(\mathsf{0}))\}\}$.

Term Rewriting

Now we define term rewrite systems and introduce the notion of term rewriting. For further details on term rewriting we refer to [BN98].

Definition 2.10 (Term Rewriting)**.** *A* term rewrite system (TRS) $\mathcal{R}$ *is a finite set of rules* $\ell \to r$ *with* $\ell, r \in \mathcal{T}(\Sigma, \mathcal{V})$ *and* $\ell \notin \mathcal{V}$*. The* rewrite relation *for* $\mathcal{R}$ *is denoted* $\to_{\mathcal{R}}$*: for* $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ *we have* $s \to_{\mathcal{R}} t$ *if there is a rule* $\ell \to r$*, a position* $pos \in Occ(s)$ *and a substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ *with* $s|_{pos} = \ell\sigma$ *and* $t = s[r\sigma]_{pos}$*. Let* $\to^{n}_{\mathcal{R}}$*,* $\to^{\geq n}_{\mathcal{R}}$*,* $\to^{*}_{\mathcal{R}}$ *denote rewrite sequences of* n *steps, of at least* n *steps, and of arbitrarily many steps, respectively (where* $n \geq 0$*). A term* t *is* terminating *for* $\mathcal{R}$ *if* $\to_{\mathcal{R}}$ *is terminating w.r.t.* t*. A TRS* $\mathcal{R}$ *is* terminating *if* $\to_{\mathcal{R}}$ *is terminating.*

Note that the above definition is the standard rewrite relation. One usually requires $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ for all rules $\ell \to r \in \mathcal{R}$ as otherwise the standard rewrite relation is never well founded. Here, we do not follow this requirement as Chapter 3 introduces a variant which can be well founded even if $\mathcal{V}(r) \nsubseteq \mathcal{V}(\ell)$.

Example 2.11 (Term Rewrite System)**.** Consider the following rules $\mathcal{R}$ over $\{\mathsf{double}, \mathsf{s}, \mathsf{0}\}$ for the function returning the double value of its argument:

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{double}(\mathsf{s}(X)) &\to \mathsf{s}(\mathsf{s}(\mathsf{double}(X))) \end{aligned}$$

For the term $t = \mathsf{double}(\mathsf{s}(\mathsf{0}))$ we obtain the following rewrite sequence where we marked $s|_{pos}$ by underlining:

$$\underline{\mathsf{double}(\mathsf{s}(\mathsf{0}))} \to_{\mathcal{R}} \mathsf{s}(\mathsf{s}(\underline{\mathsf{double}(\mathsf{0})})) \to_{\mathcal{R}} \mathsf{s}(\mathsf{s}(\mathsf{0}))$$

Part I.

Termination Analysis of Logic Programs

3. Transformational Approach

Termination of logic programs is widely studied. Most automated techniques try to prove *universal termination* of definite logic programs, i.e., one tries to show that all derivations of a logic program are finite w.r.t. the left-to-right selection rule.

Both “direct” and “transformational” approaches have been proposed in the literature (see, e.g., [DD94] for an overview and [BCG$^+$07, CLS05, CLSS06, DS02, LMS03, MR03, MS07, ND05, ND07, NGSD08, SD05a, Sma04] for more recent work on “direct” approaches). “Transformational” approaches have been developed in [AM93, AZ95, CR93, GW93, KKS98, Mar94, Mar96, Raa97] and a comparison of these approaches is given in [Ohl01]. Moreover, similar transformational approaches also exist for other programming languages (e.g., see [GSST06] for an approach to prove termination of Haskell-programs via a transformation to term rewriting). Moreover, there is also work in progress to develop such approaches for imperative programs.

In order to be successful for termination analysis of logic programs, transformational methods

(I) should be *applicable* for a class of logic programs as large as possible and
(II) should produce TRSs whose termination is *easy to analyze automatically*.

Concerning (I), the above existing transformations can only be used for certain subclasses of logic programs. More precisely, all approaches except [Mar94, Mar96] are restricted to *well-moded* programs. The transformations of [Mar94, Mar96] also consider the classes of *simply well-typed* and *safely typed* programs. However in contrast to all previous transformations, we present a new transformation which is applicable for *any* (definite) logic program. Like most approaches for termination of logic programs, we restrict ourselves to programs without cuts and negation. While there are transformational approaches which go beyond definite programs [Mar96], it is not clear how to transform non-definite logic programs into TRSs that are suitable for *automated* termination analysis, cf. (II). In order to handle cuts and negation we refer to the pre-processing introduced in Chapter 5.

Concerning (II), one needs an implementation and an empirical evaluation to find out whether termination of the transformed TRSs can indeed be verified automatically for a large class of examples. Unfortunately, to our knowledge there is only a single other termination tool available which implements a transformational approach. This tool TALP [OCM00] is based on the transformations of [AZ95, CR93, GW93] which are shown to be equally powerful in [Ohl01]. So these transformations are indeed suitable for automated termination analysis, but consequently, TALP only accepts well-moded logic programs. This is in contrast to our approach which we implemented in our termination prover AProVE [GST06]. Our experiments on large collections of examples in Section 3.6 show that our transformation indeed produces TRSs that are suitable for automated termination analysis and that AProVE is currently the most powerful termination provers for logic programs.

To illustrate the starting point for our research, we briefly review related work on connecting termination analysis of logic programs and term rewrite systems. We start by recapitulating the classical transformation of [AZ95, CR93, GW93, Ohl01] Then, we give an overview on the structure of the remainder of this chapter.

The Classical Transformation

Our transformation is inspired by the transformation of [AZ95, CR93, GW93, Ohl01]. In this classical transformation, each argument position of each predicate is either determined to be an *input* or an *output* position by a *moding function* m. So for every predicate symbol p of arity n and every $1 \leq i \leq n$, we have $m(p,i) \in \{\boldsymbol{in}, \boldsymbol{out}\}$. Thus, $m(p,i)$ states whether the i-th argument of p is an input ($\boldsymbol{in}$) or an output ($\boldsymbol{out}$) argument.

As mentioned, the moding must be such that the logic program is *well moded* [AE93]. Well-modedness guarantees that each atom selected by the left-to-right selection rule is "sufficiently" instantiated during any derivation with a query that is ground on all input positions. More precisely, a program is well moded if, and only if, for any of its clauses $H \leftarrow B_1, \ldots, B_k$ with $k \geq 0$, we have

(a) $\mathcal{V}_{out}(H) \subseteq \mathcal{V}_{in}(H) \cup \mathcal{V}_{out}(B_1) \cup \ldots \cup \mathcal{V}_{out}(B_k)$ and

(b) $\mathcal{V}_{in}(B_i) \subseteq \mathcal{V}_{in}(H) \cup \mathcal{V}_{out}(B_1) \cup \ldots \cup \mathcal{V}_{out}(B_{i-1})$ for all $1 \leq i \leq k$

$\mathcal{V}_{in}(B)$ and $\mathcal{V}_{out}(B)$ are the variables in terms on B's input and output positions.

Example 3.1. *Consider the following variant of a small example from [Ohl01].*

$$\begin{array}{lcl} \mathsf{p}(X,X). & & \\ \mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y)) & \leftarrow & \mathsf{p}(\mathsf{f}(X),\mathsf{f}(Z)),\mathsf{p}(Z,\mathsf{g}(Y)). \end{array}$$

Let m be a moding with $m(\mathsf{p},1) = \boldsymbol{in}$ and $m(\mathsf{p},2) = \boldsymbol{out}$. Then the program is well moded: This is obvious for the first clause. For the second clause, (a) holds since the output variable Y of the head is also an output variable of the second body atom. Similarly, (b) holds since the input variable X of the first body atom is also an input variable of the head, and the input variable Z of the second body atom is also an output variable of the first body atom.

In the classical transformation from logic programs to TRSs, two new function symbols p_{in} and p_{out} are introduced for each predicate p. We write "$p(\vec{s},\vec{t})$" to denote that $\vec{s}$ and $\vec{t}$ are the sequences of terms on p's in- and output positions.

- For each fact $p(\vec{s},\vec{t})$, the TRS contains the rule $p_{in}(\vec{s}) \to p_{out}(\vec{t})$.
- For each clause c of the form $p(\vec{s},\vec{t}) \leftarrow p_1(\vec{s}_1,\vec{t}_1),\dots,p_k(\vec{s}_k,\vec{t}_k)$, the resulting TRS contains the following rules:

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

  Here, $\mathcal{V}(\vec{s})$ are the variables occurring in $\vec{s}$. Moreover, if $\mathcal{V}(\vec{s}) = \{x_1,\dots,x_n\}$, then "$u_{c,1}(p_{1_{in}}(\vec{s}_1),\mathcal{V}(\vec{s}))$" abbreviates the term $u_{c,1}(p_{1_{in}}(\vec{s}_1),x_1,\dots,x_n)$, etc.

If the resulting TRS is terminating, then the original logic program terminates for any query with ground terms on all input positions of the predicates, cf. [Ohl01]. However, the converse does not hold.

Example 3.2. *For the program of Example 3.1, the transformation results in the following TRS $\mathcal{R}$.*

$$\begin{aligned} \mathsf{p}_{in}(X) &\to \mathsf{p}_{out}(X)\\ \mathsf{p}_{in}(\mathsf{f}(X)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)),X)\\ \mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(Z)),X) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z),X,Z)\\ \mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(Y)),X,Z) &\to \mathsf{p}_{out}(\mathsf{g}(Y)) \end{aligned}$$

The original logic program is terminating for any query $\mathsf{p}(t_1,t_2)$ where t_1 is a ground term. However, the above TRS is not terminating:

$$\mathsf{p}_{in}(\mathsf{f}(X)) \to_{\mathcal{R}} \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)),X) \to_{\mathcal{R}} \mathsf{u}_1(\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)),X),X) \to_{\mathcal{R}} \dots$$

In the logic program, after resolving with the second clause, one obtains a query starting with $\mathsf{p}(\mathsf{f}(\ldots),\mathsf{f}(\ldots))$. Since p's output argument $\mathsf{f}(\ldots)$ is already partly instantiated, the second clause cannot be applied again for this atom. However, this information is neglected in the translated TRS. Here, one only regards the input argument of p in order to determine whether a rule can be applied. Note that many current tools for termination proofs of logic programs like cTI *[MB05],* TALP *[OCM00],* TermiLog *[LSS97], and* TerminWeb *[CT99] fail on Example 3.1.*

So this example already illustrates a drawback of the classical transformation: there are several terminating well-moded logic programs which are transformed into non-terminating TRSs. In such cases, one fails in proving the termination of the logic program. Even worse, most of the existing transformations are not applicable for logic programs that are not well moded.[1]

Example 3.3. *We modify Example 3.1 by replacing* $\mathsf{g}(Y)$ *with* $\mathsf{g}(W)$ *in the body of the second clause:*

$$\begin{array}{lcl}\mathsf{p}(X,X). & & \\ \mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y)) & \leftarrow & \mathsf{p}(\mathsf{f}(X),\mathsf{f}(Z)),\mathsf{p}(Z,\mathsf{g}(W)).\end{array}$$

Still, all queries $\mathsf{p}(t_1,t_2)$ *terminate if* t_1 *is ground. But this program is not well moded, as the second clause violates Condition (a):* $\mathcal{V}_{out}(\mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y))) = \{Y\} \not\subseteq \mathcal{V}_{in}(\mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y))) \cup \mathcal{V}_{out}(\mathsf{p}(\mathsf{f}(X),\mathsf{f}(Z))) \cup \mathcal{V}_{out}(\mathsf{p}(Z,\mathsf{g}(W))) = \{X,Z,W\}$. *Transforming the program as before yields a TRS with the rule* $\mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(W)),X,Z) \to \mathsf{p}_{out}(\mathsf{g}(Y))$. *So non-well-moded programs result in rules with variables like* Y *in the right- but not in the left-hand side. Such rules are usually forbidden in term rewriting and they do not terminate, since* Y *may be instantiated arbitrarily.*

Example 3.4. *A natural non-well-moded example is the* append*-program with the clauses*

$$\begin{array}{lcl}\mathsf{append}([\,],M,M). & & \\ \mathsf{append}([X|L],M,[X|N]) & \leftarrow & \mathsf{append}(L,M,N).\end{array}$$

and the moding $m(\mathsf{append},1) = \boldsymbol{in}$ *and* $m(\mathsf{append},2) = m(\mathsf{append},3) = \boldsymbol{out}$, *i.e., one only considers* append*'s first argument as input. Due to the first clause* $\mathsf{append}([\,],M,M)$, *this program is not well moded although all queries of the form* $\mathsf{append}(t_1,t_2,t_3)$ *are terminating if* t_1 *is ground.*

Term Rewriting Techniques for Termination of Logic Programs

Recently, several authors tackled the problem of applying termination techniques from term rewriting for (possibly non-well-moded) logic programs. A framework for integrating orders from term rewriting into *direct* termination approaches for logic programs is discussed in [DS02]. But in contrast to [DS02], transformational approaches like the one presented in this chapter can also apply more recent termination techniques from term rewriting for termination of logic programs (e.g., refined variants of the *dependency pair* method like [GTS05a, GTSF06, HM05a], *semantic labelling* [Zan95], *matchbounds* [GHW04], etc.). However, the automation of this framework is non-trivial in general. As an instance of this framework, the automatic application of polynomial interpretations

[1]Example 3.3 is neither well moded nor simply well typed nor safely typed (using "*Any*" and "*Ground*") as required by the transformations of [AM93, AZ95, CR93, GW93, KKS98, Mar94, Mar96, Raa97].

(well-known in term rewriting) to termination analysis of logic programs is investigated in [ND05, ND07]. Moreover, in Chapter 4 we extend this work further by also adapting the *dependency pair framework* [AG00, GTS05a, GTSF06] from TRSs to the logic programming setting.

Instead of integrating each termination technique from term rewriting separately, in the current chapter we want to make all these techniques available at once. Therefore, we choose a transformational approach. Our goal is a method which

(A) handles programs like Example 3.1 where the classical transformation fails,

(B) handles non-well-moded programs like Example 3.3 where most previous transformational techniques are not even applicable,

(C) allows the successful *automated* application of techniques from term rewriting for logic programs like Example 3.1 and 3.3 where tools based on direct approaches fail. For larger and more realistic examples we refer to the experiments in Section 3.6.

Structure of the Chapter

We start by introducing a modified notion of term rewriting in Section 3.1, so-called "infinitary constructor rewriting". Then, in Section 3.2 we modify the transformation from logic programs to TRSs to achieve (A) and (B). So restrictions like well-modedness, simple well-typedness, or safe typedness are no longer required.

Section 3.3 shows that the existing termination techniques for TRSs can easily be adapted in order to prove termination of infinitary constructor rewriting. For a full automation of the approach, one has to transform the set of queries that has to be analyzed for the logic program to a corresponding set of terms that has to be analyzed for the transformed TRS. This set of terms is characterized by a so-called *argument filter* and we present heuristics to find a suitable argument filter in Section 3.4. Section 3.5 gives a formal proof that our new transformation and our approach to automated termination analysis are strictly more powerful than the classical one. We present and discuss an extensive experimental evaluation of our results in Section 3.6 which shows that Goal (C) is achieved as well. In other words, the implementation of our approach can indeed compete with modern tools for direct termination analysis of logic programs and it succeeds for many programs where these tools fail. Finally, we summarize the contributions of this chapter in Section 3.7.

3.1. Preliminaries

Our new transformation from Section 3.2 results in TRSs where the notion of "rewriting" has to be slightly modified: we regard a restricted form of infinitary rewriting, called *infinitary constructor rewriting.* The reason is that logic programs use *unification*, whereas TRSs use *matching.*

To illustrate this difference, consider the logic program $\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X)$ which does not terminate for the query $\mathsf{p}(X)$: Unifying the query $\mathsf{p}(X)$ with the head of the variable-renamed rule $\mathsf{p}(\mathsf{s}(X_1)) \leftarrow \mathsf{p}(X_1)$ yields the new query $\mathsf{p}(X_1)$. Afterwards, unifying the new query $\mathsf{p}(X_1)$ with the head of the variable-renamed rule $\mathsf{p}(\mathsf{s}(X_2)) \leftarrow \mathsf{p}(X_2)$ yields the new query $\mathsf{p}(X_2)$, etc.

In contrast, the related TRS $\mathsf{p}(\mathsf{s}(X)) \to \mathsf{p}(X)$ terminates for all finite terms. When applying the rule to some subterm t, one has to *match* the left-hand side ℓ of the rule against t. For example, when applying the rule to the term $\mathsf{p}(\mathsf{s}(\mathsf{s}(Y)))$, one would use the matcher that instantiates X with $\mathsf{s}(Y)$. Thus, $\mathsf{p}(\mathsf{s}(\mathsf{s}(Y)))$ would be rewritten to the instantiated right-hand side $\mathsf{p}(\mathsf{s}(Y))$. Hence, one occurrence of the symbol s is eliminated in every rewrite step. This implies that rewriting will always terminate. So in contrast to unification (where one searches for a substitution θ with $t\theta = \ell\theta$), here we only use matching (i.e., we search for a substitution θ with $t = \ell\theta$, but we do not instantiate the term t that is being rewritten).

However, the infinite derivation of the logic program above corresponds to an infinite reduction of the TRS above with the *infinite* term $\mathsf{p}(\mathsf{s}(\mathsf{s}(\ldots)))$ containing infinitely many nested s-symbols. So to simulate unification by matching, we have to regard TRSs where the variables in rewrite rules may be instantiated by infinite constructor terms. It turns out that this form of rewriting also allows us to analyze the termination behavior of logic programming with infinite terms, i.e., of logic programming without occur check.

Thus, we consider infinite *substitutions* $\theta : \mathcal{V} \to \mathcal{T}^\infty(\Sigma, \mathcal{V})$ for the derivations defined in Definition 2.7 such that derivation coincides with logic programming without an occur check. Since we consider only definite logic programs, any program which is terminating without occur check is also terminating with occur check, but not vice versa. So if our approach detects "termination", then the program is indeed terminating, no matter whether one uses logic programming with or without occur check. In other words, our approach is sound for both kinds of logic programming, whereas most other approaches only consider logic programming with occur check.

Example 3.5. Regard a program $\mathcal{P}$ with the clauses $\mathsf{p}(X) \leftarrow \mathsf{equal}(X, \mathsf{s}(X)), \mathsf{p}(X)$ and $\mathsf{equal}(X, X)$. We obtain $\mathsf{p}(X) \vdash^2_\mathcal{P} \mathsf{p}(\mathsf{s}(\mathsf{s}(\ldots))) \vdash^2_\mathcal{P} \mathsf{p}(\mathsf{s}(\mathsf{s}(\ldots))) \vdash^2_\mathcal{P} \ldots$, where $\mathsf{s}(\mathsf{s}(\ldots))$ is the term containing infinitely many nested s-symbols. So the finite query $\mathsf{p}(X)$ leads to a derivation with infinite (rational) queries. While $\mathsf{p}(X)$ is not terminating according to Definition 2.7, it would be terminating if one uses logic programming with occur check. Indeed, tools like cTI [MB05] and TerminWeb [CT99] report that such queries are "terminating". So in contrast to our technique, such tools are in general not sound for logic programming without occur check, although this form of logic programming is typically used in practice.

Now we introduce the notion of *infinitary constructor rewriting.*

Definition 3.6 (Infinitary Constructor Rewriting)**.** *Given a* TRS $\mathcal{R} \subseteq (\mathcal{T}(\Sigma, \mathcal{V}) \setminus \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$*, we divide the signature into* defined *symbols* $\Sigma_D = \{f \mid \ell \to r \in \mathcal{R}, root(\ell) = f\}$ *and* constructors $\Sigma_C = \Sigma \setminus \Sigma_D$*.*

The infinitary constructor rewrite relation *for* $\mathcal{R}$ *is denoted* $\overset{\infty}{\to}_{\mathcal{R}}$*: for* $s, t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ *we have* $s \overset{\infty}{\to}_{\mathcal{R}} t$ *if there is a rule* $\ell \to r$*, a position pos and a substitution* $\sigma : \mathcal{V} \to \mathcal{T}^\infty(\Sigma_C, \mathcal{V})$ *with* $s|_{pos} = \ell\sigma$ *and* $t = s[r\sigma]_{pos}$*. Let* $\overset{\infty}{\to}^n_{\mathcal{R}}$*,* $\overset{\infty}{\to}^{\geq n}_{\mathcal{R}}$*,* $\overset{\infty}{\to}^*_{\mathcal{R}}$ *denote rewrite sequences of n steps, of at least n steps, and of arbitrary many steps, respectively (where* $n \geq 0$*). A term t is* $\overset{\infty}{\to}$-terminating *for* $\mathcal{R}$ *if* $\overset{\infty}{\to}_{\mathcal{R}}$ *is terminating for t. A TRS* $\mathcal{R}$ *is* $\overset{\infty}{\to}$-terminating *if* $\overset{\infty}{\to}_{\mathcal{R}}$ *is terminating.*

The above definition of $\overset{\infty}{\to}_{\mathcal{R}}$ differs from the standard rewrite relation $\to_{\mathcal{R}}$ in two aspects:

(i) We only permit instantiations of rule variables by constructor terms.

(ii) We use substitutions with possibly non-rational infinite terms.

In Examples 3.8 and 3.9 in the next section, we will motivate these modifications and show that there are TRSs which terminate w.r.t. the usual rewrite relation, but are non-terminating w.r.t. infinitary constructor rewriting and vice versa.

3.2. Transforming Logic Programs into Term Rewrite Systems

Now we modify the classical transformation of logic programs into TRSs to make it applicable for *arbitrary* (possibly non-well-moded) programs as well. In this section we first present the new transformation and then prove its soundness. Later in Section 3.5 we formally prove that the classical transformation is strictly subsumed by our new one.

The Improved Transformation

Instead of separating between input and output positions of a predicate p/n, now we keep *all* arguments both for p_{in} and p_{out} (i.e., p_{in} and p_{out} have arity n).

Definition 3.7 (Transformation)**.** *A logic program* $\mathcal{P}$ *over* (Σ, Δ) *is transformed into the following TRS* $\mathcal{R}_{\mathcal{P}}$ *over* $\Sigma_{\mathcal{P}} = \Sigma \cup \{p_{in}/n, p_{out}/n \mid p/n \in \Delta\} \cup \{u_{c,i} \mid c \in \mathcal{P}, 1 \leq i \leq k$*, where k is the number of atoms in the body of c*$\}$*.*

- *For each fact* $p(\vec{s})$ *in* $\mathcal{P}$*, the TRS* $\mathcal{R}_{\mathcal{P}}$ *contains the rule* $p_{in}(\vec{s}) \to p_{out}(\vec{s})$*.*

- *For each clause c of the form $p(\vec{s}) \leftarrow p_1(\vec{s}_1), \ldots, p_k(\vec{s}_k)$ in $\mathcal{P}$, $\mathcal{R}_\mathcal{P}$ contains:*

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

The following two examples motivate the need for infinitary constructor rewriting in Definition 3.7, i.e., they motivate Modifications (i) and (ii) in Section 3.1.

Example 3.8. *For the logic program of Example 3.1, the transformation of Definition 3.7 yields the following TRS.*

$$\begin{aligned}
\mathsf{p}_{in}(X, X) &\to \mathsf{p}_{out}(X, X) && (1)\\
\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) && (2)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z, \mathsf{g}(Y)), X, Y, Z) && (3)\\
\mathsf{u}_2(\mathsf{p}_{out}(Z, \mathsf{g}(Y)), X, Y, Z) &\to \mathsf{p}_{out}(\mathsf{f}(X), \mathsf{g}(Y)) && (4)
\end{aligned}$$

This example shows why rules of TRSs may only be instantiated with constructor terms (Modification (i)). The reason is that local variables like Z (i.e., variables occurring in the body but not in the head of a clause) give rise to rules $\ell \to r$ where $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$ (cf. Rule (2)). Such rules are never terminating in standard term rewriting. However, in our setting one may only instantiate Z with constructor terms. So in contrast to the old transformation in Example 3.2, now all terms $\mathsf{p}_{in}(t_1, t_2)$ $\overset{\infty}{\to}$-terminate for the TRS if t_1 is finite, since now the second argument of p_{in} prevents an infinite application of Rule (2). Indeed, constructor rewriting correctly simulates the behavior of logic programs, since the variables in a logic program are only instantiated by "constructor terms".

For the non-well-moded program of Example 3.3, one obtains a similar TRS where $\mathsf{g}(Y)$ is replaced by $\mathsf{g}(W)$ in the right-hand side of Rule (3) and the left-hand side of Rule (4). Again, all terms $\mathsf{p}_{in}(t_1, t_2)$ are $\overset{\infty}{\to}$-terminating for this TRS provided that t_1 is finite. Thus, we can now handle programs where the classical transformation of [AZ95, CR93, GW93, Ohl01] failed, cf. Goals (A) and (B) in Section 3.

Derivations in logic programming use *unification*, while rewriting is defined by *matching*. Example 3.9 shows that to simulate unification by matching, we have to consider substitutions with infinite and even non-rational terms (Modification (ii)).

Example 3.9. *Let $\mathcal{P}$ be $\mathsf{ordered}(\mathsf{cons}(X, \mathsf{cons}(\mathsf{s}(X), XS))) \leftarrow \mathsf{ordered}(\mathsf{cons}(\mathsf{s}(X), XS))$. If one only considers rewriting with finite or rational terms, then the transformed TRS $\mathcal{R}_\mathcal{P}$ is terminating. However, the query $\mathsf{ordered}(YS)$ is not terminating for $\mathcal{P}$. Thus,* *to obtain a sound approach, $\mathcal{R}_\mathcal{P}$ must also be non-$\overset{\infty}{\rightarrow}$-terminating. Indeed, the term $t = \mathsf{ordered}_{in}(\mathsf{cons}(X,\ \mathsf{cons}(\mathsf{s}(X),\ \mathsf{cons}(\mathsf{s}^2(X),\ \ldots))))$ is non-$\overset{\infty}{\rightarrow}$-terminating with $\mathcal{R}_\mathcal{P}$'s rule: $\mathsf{ordered}_{in}(\mathsf{cons}(X, \mathsf{cons}(\mathsf{s}(X), XS))) \rightarrow \mathsf{u}(\mathsf{ordered}_{in}(\mathsf{cons}(\mathsf{s}(X), XS)), X, XS)$. Here, the non-rational term t corresponds to the infinite derivation with the query $\mathsf{ordered}(YS)$.*

Soundness of the Transformation

We first show an auxiliary lemma that is needed to prove the soundness of the transformation. It relates derivations with the logic program $\mathcal{P}$ to rewrite sequences with the TRS $\mathcal{R}_\mathcal{P}$.

Lemma 3.10 (Connecting $\mathcal{P}$ and $\mathcal{R}_\mathcal{P}$)**.** *Let $\mathcal{P}$ be a program, let $\vec{t}$ be terms from $\mathcal{T}^{rat}(\Sigma, \mathcal{V})$, let $p(\vec{t}) \vdash^n_{\mathcal{P},\sigma} Q$. If $Q = \square$, then $p_{in}(\vec{t})\sigma \overset{\infty}{\rightarrow}{}^{\geq n}_{\mathcal{R}_\mathcal{P}} p_{out}(\vec{t})\sigma$. Otherwise, if Q is "$q(\vec{v}), \ldots$", then $p_{in}(\vec{t})\sigma \overset{\infty}{\rightarrow}{}^{\geq n}_{\mathcal{R}_\mathcal{P}} r$ for a term r containing the subterm $q_{in}(\vec{v})$.*

Proof. Let $p(\vec{t}) = Q_0 \vdash_{c_1,\theta_1} \ldots \vdash_{c_n,\theta_n} Q_n = Q$ with $\sigma = \theta_1 \ldots \theta_n$. We use induction on n. The base case $n = 0$ is trivial, since $Q = p(\vec{t})$ and $p_{in}(\vec{t}) \overset{\infty}{\rightarrow}{}^{0}_{\mathcal{R}_\mathcal{P}} p_{in}(\vec{t})$.

Now let $n \geq 1$. We first regard the case $Q_1 = \square$ and $n = 1$. Then c_1 is a fact $p(\vec{s})$ and θ_1 is the mgu of $p(\vec{t})$ and $p(\vec{s})$. Note that such mgu's instantiate all variables with constructor terms (as symbols of Σ are constructors of $\mathcal{R}_\mathcal{P}$). We obtain $p_{in}(\vec{t})\theta_1 = p_{in}(\vec{s})\theta_1 \overset{\infty}{\rightarrow}_{\mathcal{R}_\mathcal{P}} p_{out}(\vec{s})\theta_1 = p_{out}(\vec{t})\theta_1$ where $\sigma = \theta_1$.

Finally, let $Q_1 \neq \square$. Thus, c_1 is $p(\vec{s}) \leftarrow p_1(\vec{s_1}), \ldots, p_k(\vec{s_k})$ and Q_1 is $p_1(\vec{s_1})\theta_1, \ldots, p_k(\vec{s_k})\theta_1$ where θ_1 is the mgu of $p(\vec{t})$ and $p(\vec{s})$. There is an i with $1 \leq i \leq k$ such that for all j with $1 \leq j \leq i-1$ we have $p_j(\vec{s_j})\sigma_0 \ldots \sigma_{j-1} \vdash^{n_j}_{\mathcal{P},\sigma_j} \square$. Moreover, if $Q = \square$ then we can choose $i = k$ and $p_i(\vec{s_i})\sigma_0 \ldots \sigma_{i-1} \vdash^{n_i}_{\mathcal{P},\sigma_i} \square$. Otherwise, if Q is "$q(\vec{v}), \ldots$", then we can choose i such that $p_i(\vec{s_i})\sigma_0 \ldots \sigma_{i-1} \vdash^{n_i}_{\mathcal{P},\sigma_i} q(\vec{v}), \ldots$ Here, $n = n_1 + \ldots + n_i + 1$, $\sigma_0 = \theta_1$, $\sigma_1 = \theta_2 \ldots \theta_{n_1+1}$, $\ldots$, and $\sigma_i = \theta_{n_1+\ldots+n_{i-1}+2} \ldots \theta_{n_1+\ldots+n_i+1}$. So $\sigma = \sigma_0 \ldots \sigma_i$.

By the induction hypothesis we have $p_{j_{in}}(\vec{s_j})\sigma_0 \ldots \sigma_j \overset{\infty}{\rightarrow}{}^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\vec{s_j})\sigma_0 \ldots \sigma_j$ and thus also $p_{j_{in}}(\vec{s_j})\sigma \overset{\infty}{\rightarrow}{}^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\vec{s_j})\sigma$. Moreover, if $Q = \square$ then we also have $p_{i_{in}}(\vec{s_i})\sigma \overset{\infty}{\rightarrow}{}^{\geq n_i}_{\mathcal{R}_\mathcal{P}} p_{i_{out}}(\vec{s_i})\sigma$ where $i = k$. Otherwise, if Q is "$q(\vec{v}), \ldots$", then the induction hypothesis implies $p_{i_{in}}(\vec{s_i})\sigma \overset{\infty}{\rightarrow}{}^{\geq n_i}_{\mathcal{R}_\mathcal{P}} r'$, where r' contains $q_{in}(\vec{v})$. Thus

$$\begin{array}{rcll}
p_{in}(\vec{t})\sigma & = & p_{in}(\vec{s})\sigma & \overset{\infty}{\rightarrow}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,1}(p_{1_{in}}(\vec{s_1}), \mathcal{V}(\vec{s}))\sigma \\
 & & & \overset{\infty}{\rightarrow}{}^{\geq n_1}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,1}(p_{1_{out}}(\vec{s_1}), \mathcal{V}(\vec{s}))\sigma \\
 & & & \overset{\infty}{\rightarrow}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,2}(p_{2_{in}}(\vec{s_2}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s_1}))\sigma \\
 & & & \overset{\infty}{\rightarrow}{}^{\geq n_2}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,2}(p_{2_{out}}(\vec{s_2}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s_1}))\sigma \\
 & & & \overset{\infty}{\rightarrow}{}^{\geq n_3+\ldots+n_{i-1}}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,i}(p_{i_{in}}(\vec{s_i}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s_1}) \cup \ldots \cup \mathcal{V}(\vec{s_{i-1}}))\sigma
\end{array}$$

Moreover, if $Q = \Box$, then $i = k$ and the rewrite sequence yields $p_{out}(\vec{t})\sigma$, since

$$\begin{array}{rcl} u_{c_1,i}(p_{i_{in}}(\vec{s}_i), \mathcal{V}(\vec{s}) \cup \ldots \cup \mathcal{V}(\vec{s}_{i-1}))\sigma & \overset{\infty}{\to}{}^{\geq n_i}_{\mathcal{R}_\mathcal{P}} & u_{c_1,i}(p_{i_{out}}(\vec{s}_i), \mathcal{V}(\vec{s}) \cup \ldots \cup \mathcal{V}(\vec{s}_{i-1}))\sigma \\ & \overset{\infty}{\to}_{\mathcal{R}_\mathcal{P}} & p_{out}(\vec{s})\sigma \quad = \quad p_{out}(\vec{t})\sigma. \end{array}$$

Otherwise, if Q is "$q(\vec{v}), \ldots$", then rewriting yields a term containing $q_{in}(\vec{v})$:

$$u_{c_1,i}(p_{i_{in}}(\vec{s}_i), \mathcal{V}(\vec{s}) \cup \ldots \cup \mathcal{V}(\vec{s}_{i-1}))\sigma \quad \overset{\infty}{\to}{}^{\geq n_i}_{\mathcal{R}_\mathcal{P}} \quad u_{c_1,i}(r', \mathcal{V}(\vec{s})\sigma \cup \ldots \cup \mathcal{V}(\vec{s}_{i-1})\sigma).$$

$\Box$

For the soundness proof, we need another lemma which states that we can restrict ourselves to non-terminating queries which only consist of a single atom.

Lemma 3.11 (Non-Terminating Queries). *Let $\mathcal{P}$ be a logic program. Then for every infinite derivation $Q_0 \vdash_\mathcal{P} Q_1 \vdash_P \ldots$, there is a Q_i of the form "$q(\vec{v}), \ldots$" with $i > 0$ such that the query $q(\vec{v})$ is also non-terminating.*

Proof. Assume that for all $i > 0$, the first atom in Q_i does not have an infinite derivation. Then for each Q_i there are two cases: either the first atom fails or it can successfully be proved. In the former case, there is no infinite reduction from Q_i which contradicts the infiniteness of the derivation from Q_0. Thus for all $i > 0$, the first atom of Q_i is successfully proved in n_i steps during the derivation $Q_0 \vdash_\mathcal{P} Q_1 \vdash_\mathcal{P} \ldots$ Let m be the number of atoms in Q_1. But then $Q_{1+n_1+\ldots+n_m}$ is the empty query $\Box$ which again contradicts the infiniteness of the derivation. $\Box$

We use *argument filters* to characterize the classes of queries whose termination we want to analyze. Related definitions can be found in, e.g., [AG00, LS96].

Definition 3.12 (Argument Filter). *A function $\pi : \Sigma \cup \Delta \to 2^\mathbb{N}$ is an* argument filter *π over a signature (Σ, Δ) if and only if $\pi(f/n) \subseteq \{1, \ldots, n\}$ for every $f/n \in \Sigma \cup \Delta$. 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 $\pi(f/n) = \{i_1, \ldots, i_k\}$ with $i_1 < \ldots < i_k$. Here, the new terms and atoms are from the filtered signature (Σ_π, Δ_π) where $f/n \in \Sigma$ implies $f/k \in \Sigma_\pi$ and likewise for Δ_π. For a logic program $\mathcal{P}$ we write $(\Sigma_{\mathcal{P}_\pi}, \Delta_{\mathcal{P}_\pi})$ instead of $((\Sigma_\mathcal{P})_\pi, (\Delta_\mathcal{P})_\pi)$. For any TRS $\mathcal{R}$, we define $\pi(\mathcal{R}) = \{\pi(\ell) \to \pi(r) \mid \ell \to r \in \mathcal{R}\}$. The set of all argument filters over a signature (Σ, Δ) is denoted by $AF(\Sigma, \Delta)$. We write $AF(\Sigma)$ instead of $AF(\Sigma, \varnothing)$ and speak of an argument filter "over Σ". We also write $\pi(f)$ instead of $\pi(f/n)$ if the arity of f is clear from the context.*

An argument filter π' is a refinement *of a filter π if and only if $\pi'(f) \subseteq \pi(f)$ for all $f \in \Sigma \cup \Delta$.*

Argument filters specify those positions which have to be instantiated with finite ground terms. Then, we analyze termination of all queries Q where $\pi(Q)$ is a (finite) ground atom. In Example 3.1, we wanted to prove termination for all queries $\mathsf{p}(t_1, t_2)$ where t_1 is finite and ground. These queries are described by the filter $\pi(h) = \{1\}$ for all $h \in \{\mathsf{p}, \mathsf{f}, \mathsf{g}\}$. Thus, we have $\pi(\mathsf{p}(t_1, t_2)) = \mathsf{p}(\pi(t_1))$.

Note that argument filters also operate on *function* instead of just *predicate* symbols. Therefore, they can describe more sophisticated classes of queries than the classical approach of [AZ95, CR93, GW93, Ohl01] which only distinguishes between input and output positions of predicates. For example, if one wants to analyze all queries $\mathsf{append}(t_1, t_2, t_3)$ where t_1 is a finite list, one would use the filter $\pi(\mathsf{append}) = \{1\}$ and $\pi(\bullet) = \{2\}$, where "$\bullet$" is the list constructor (i.e., $\bullet(X, L) = [X|L]$). Of course, our method can easily prove that all these queries are terminating for the program of Example 3.4.

Now we show the soundness theorem: to prove termination of all queries Q where $\pi(Q)$ is a finite ground atom, it suffices to show $\overset{\infty}{\to}$-termination of all those terms $p_{in}(\vec{t})$ for the TRS $\mathcal{R}_\mathcal{P}$ where $\pi(p_{in}(\vec{t}))$ is a finite ground term and where $\vec{t}$ only contains function symbols from the logic program $\mathcal{P}$. Here, π has to be extended to the new function symbols p_{in} by defining $\pi(p_{in}) = \pi(p)$.

Theorem 3.13 (Soundness of the Transformation)**.** *Let $\mathcal{P}$ be a logic program and let π be an argument filter over (Σ, Δ). We extend π such that $\pi(p_{in}) = \pi(p)$ for all $p \in \Delta$. Let $S = \{p_{in}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V}),\ \pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})\}$. If all terms $s \in S$ are $\overset{\infty}{\to}$-terminating for $\mathcal{R}_\mathcal{P}$, then all queries $Q \in \mathcal{A}^{rat}(\Sigma, \Delta, \mathcal{V})$ with $\pi(Q) \in \mathcal{A}(\Sigma_\pi, \Delta_\pi)$ are terminating for $\mathcal{P}$.*[2]

Proof. Assume that there is a non-terminating query $p(\vec{t})$ as above with $p(\vec{t}) \vdash_\mathcal{P} Q_1 \vdash_\mathcal{P} Q_2 \vdash_\mathcal{P} \ldots$ By Lemma 3.11 there is an $i_1 > 0$ with $Q_{i_1} = q_1(\vec{v}_1), \ldots$ and an infinite derivation $q_1(\vec{v}_1) \vdash_\mathcal{P} Q'_1 \vdash_\mathcal{P} Q'_2 \vdash_P \ldots$ From $p(\vec{t}) \vdash^{i_1}_{\mathcal{P},\sigma_0} q_1(\vec{v}_1), \ldots$ and Lemma 3.10 we get $p_{in}(\vec{t})\sigma_0 \overset{\infty}{\to}{}^{\geq i_1}_{\mathcal{R}_\mathcal{P}} r_1$, where r_1 contains the subterm $q_{1_{in}}(\vec{v}_1)$.

By Lemma 3.11 again, there is an $i_2 > 0$ with $Q'_{i_2} = q_2(\vec{v}_2), \ldots$ and an infinite derivation $q_2(\vec{v}_2) \vdash_\mathcal{P} Q''_1 \vdash_\mathcal{P} \ldots$ From $q_1(\vec{v}_1) \vdash^{i_2}_{\mathcal{P},\sigma_1} q_2(\vec{v}_2), \ldots$ and Lemma 3.10 we get $p_{in}(\vec{t})\sigma_0\sigma_1 \overset{\infty}{\to}{}^{\geq i_1}_{\mathcal{R}_\mathcal{P}} r_1\sigma_1 \overset{\infty}{\to}{}^{\geq i_2}_{\mathcal{R}_\mathcal{P}} r_2$, where r_2 contains the subterm $q_{2_{in}}(\vec{v}_2)$.

Continuing this reasoning we obtain an infinite sequence $\sigma_0, \sigma_1, \ldots$ of substitutions. For each $j \geq 0$, let $\mu_j = \sigma_j\ \sigma_{j+1} \ldots$ result from the infinite composition of these substitutions.[3]

[2] It is currently open whether the converse holds as well. For a short discussion see Section 3.6.

[3] The composition of *infinitely* many substitutions $\sigma_0, \sigma_1, \ldots$ is defined as follows. The definition ensures that $t\sigma_0\sigma_1 \ldots$ is an instance of $t\sigma_0 \ldots \sigma_n$ for all terms (or atoms) t and all $n \geq 0$. It suffices to define the symbols at the positions of $t\sigma_0\sigma_1 \ldots$ for any term t. Obviously, pos is a position of $t\sigma_0\sigma_1 \ldots$ if, and only if, pos is a position of $t\sigma_0 \ldots \sigma_n$ for some $n \geq 0$. We define that the symbol of $t\sigma_0\sigma_1 \ldots$ at such a position pos is $f \in \Sigma$ if, and only if, f is at position pos in $t\sigma_0 \ldots \sigma_m$ for some $m \geq 0$. Otherwise, $(t\sigma_0 \ldots \sigma_n)|_{pos} = X_0 \in \mathcal{V}$. Let $n = i_0 < i_1 < \ldots$ be the maximal (finite or infinite) sequence with $\sigma_{i_j+1}(X_j) = \ldots = \sigma_{i_{j+1}-1}(X_j) = X_j$ and $\sigma_{i_{j+1}}(X_j) = X_{j+1}$ for all j. We require $X_j \neq X_{j+1}$, but permit $X_j = X_{j'}$ otherwise. If this sequence is finite (i.e., it has the form $n = i_0 < \ldots < i_m$), then we

Since $r_j\mu_j$ is an instance of $r_j\sigma_j\ldots\sigma_n$ for all $n \geq j$, we obtain that $p_{in}(\vec{t})\mu_0$ is non-$\overset{\infty}{\to}$-terminating for $\mathcal{R}_\mathcal{P}$:

$$p_{in}(\vec{t})\mu_0 \overset{\infty}{\to}{}^{\geq i_1}_{\mathcal{R}_\mathcal{P}} r_1\mu_1 \overset{\infty}{\to}{}^{\geq i_2}_{\mathcal{R}_\mathcal{P}} r_2\mu_2 \overset{\infty}{\to}{}^{\geq i_3}_{\mathcal{R}_\mathcal{P}} \ldots$$

As $\pi(p(\vec{t})) \in \mathcal{A}(\Sigma_\pi, \Delta_\pi)$ and thus $\pi(p_{in}(\vec{t})) = \pi(p_{in}(\vec{t})\mu_0) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})$, this is a contradiction. □

3.3. Termination of Infinitary Constructor Rewriting

One of the most powerful methods for automated termination analysis of rewriting is the *dependency pair* (DP) method [AG00] which is implemented in most current termination tools for TRSs. However, since the DP method only proves termination of term rewriting with *finite* terms, its use is not sound in our setting. Nevertheless, we now show that only very slight modifications are required to adapt dependency pairs from ordinary rewriting to infinitary constructor rewriting. So any rewriting tool implementing dependency pairs can easily be modified in order to prove termination of infinitary constructor rewriting as well. Then, it can also analyze termination of logic programs using the transformation of Definition 3.7.

Moreover, dependency pairs are a general framework that permits the integration of *any* termination technique for TRSs [GTS05a, Thm. 36]. Therefore, instead of adapting each technique separately, it is sufficient only to adapt the DP framework to infinitary constructor rewriting. Then, *any* termination technique can be directly used for infinitary constructor rewriting. In this section we first adapt the notions and the main termination criterion of the dependency pair method to infinitary constructor rewriting and then we show how to automate this criterion by adapting the "DP processors" of the DP framework.

Dependency Pairs for Infinitary Rewriting

Let $\mathcal{R}$ be a TRS. For each defined symbol $f/n \in \Sigma_D$, we extend the set of constructors Σ_C by a fresh *tuple symbol* $f^\sharp/n$. We often write F instead of $f^\sharp$. For $t = g(\vec{t})$ with $g \in \Sigma_D$, let $t^\sharp$ denote $g^\sharp(\vec{t})$.

Definition 3.14 (Dependency Pair [AG00]). *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 is a subterm of r, $root(t) \in \Sigma_D\}$.*

define $(t\sigma_0\sigma_1\ldots)|_{pos} = X_m$. Otherwise, the substitutions perform infinitely many variable renamings. In this case, we use one special variable Z_∞ and define $(t\sigma_0\sigma_1\ldots)|_{pos} = Z_\infty$. So if $\sigma_0(X) = Y$, $\sigma_1(Y) = X$, $\sigma_2(X) = Y$, $\sigma_3(Y) = X$, etc., we define $X\sigma_0\sigma_1\ldots = Y\sigma_0\sigma_1\ldots = Z_\infty$.

Example 3.15. *Consider again the logic program of Example 3.1 which was transformed into the following TRS $\mathcal{R}$ in Example 3.8.*

$$\begin{aligned} \mathsf{p}_{in}(X,X) &\to \mathsf{p}_{out}(X,X) & (1)\\ \mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) & (2)\\ \mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z) & (3)\\ \mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(Y)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)) & (4)\end{aligned}$$

For this TRS $\mathcal{R}$, we have $\Sigma_D = \{\mathsf{p}_{in}, \mathsf{u}_1, \mathsf{u}_2\}$ and $DP(\mathcal{R})$ is

$$\begin{aligned} \mathsf{P}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{P}_{in}(\mathsf{f}(X),\mathsf{f}(Z)) & (5)\\ \mathsf{P}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) & (6)\\ \mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{P}_{in}(Z,\mathsf{g}(Y)) & (7)\\ \mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{U}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z) & (8)\end{aligned}$$

While Definition 3.14 is from [AG00], all following definitions and theorems are new. They extend existing concepts from ordinary to infinitary constructor rewriting.

For termination, one tries to prove that there are no infinite *chains* of dependency pairs. Intuitively, a dependency pair corresponds to a function call and a chain represents a possible sequence of calls that can occur during rewriting. Definition 3.16 extends the notion of chains to infinitary constructor rewriting. To this end, we use an argument filter π that describes which arguments of function symbols have to be *finite* terms. So if π does not delete arguments (i.e., if $\pi(f) = \{1,\ldots,n\}$ for all f/n), then this corresponds to ordinary (finitary) constructor rewriting and if π deletes all arguments (i.e., if $\pi(f) = \varnothing$ for all f), then this corresponds to full infinitary constructor rewriting. In Definition 3.16, the TRS $\mathcal{D}$ usually stands for a set of dependency pairs. (Note that if $\mathcal{R}$ is a TRS, then $DP(\mathcal{R})$ is also a TRS.)

Definition 3.16 (Chain). *Let $\mathcal{D},\mathcal{R}$ be TRSs and π be an argument filter. A (possibly infinite) sequence of pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{D}$ is a $(\mathcal{D},\mathcal{R},\pi)$-*chain *if, and only if,*

- *for all $i \geq 1$, there are substitutions $\sigma_i : \mathcal{V} \to \mathcal{T}^\infty(\Sigma_C,\mathcal{V})$ such that $t_i\sigma_i \overset{\infty}{\to}{}^*_{\mathcal{R}}\, s_{i+1}\sigma_{i+1}$, and*
- *for all $i \geq 1$, we have $\pi(s_i\sigma_i), \pi(t_i\sigma_i) \in \mathcal{T}(\Sigma_\pi)$. Moreover, if the rewrite sequence from $t_i\sigma_i$ to $s_{i+1}\sigma_{i+1}$ has the form $t_i\sigma_i = q_0 \overset{\infty}{\to}_{\mathcal{R}} \ldots \overset{\infty}{\to}_{\mathcal{R}} q_m = s_{i+1}\sigma_{i+1}$, then for all terms in this rewrite sequence we have $\pi(q_0),\ldots,\pi(q_m) \in \mathcal{T}(\Sigma_\pi)$ as well. So all terms in the sequence have finite ground terms on those positions which are not filtered away by π.*

In Example 3.15, “(6), (7)” is a chain for any argument filter π: if one instantiates X

and Z with the same finite ground term, then (6)'s instantiated right-hand side rewrites to an instance of (7)'s left-hand side. Note that if one uses an argument filter π which permits an instantiation of X and Z with the infinite term $\mathsf{f}(\mathsf{f}(\ldots))$, then there is also an infinite chain "(6), (7), (6), (7), ..."

In order to prove termination of a program $\mathcal{P}$, by Theorem 3.13 we have to show that all terms $p_{in}(\vec{t})$ are $\overset{\infty}{\to}$-terminating for $\mathcal{R}_{\mathcal{P}}$ whenever $\pi(p_{in}(\vec{t}))$ is a finite ground term and $\vec{t}$ only contains function symbols from the logic program (i.e., $\vec{t}$ contains no defined symbols of the TRS $\mathcal{R}_{\mathcal{P}}$). Theorem 3.17 states that one can prove absence of infinite $(DP(\mathcal{R}_{\mathcal{P}}), \mathcal{R}_{\mathcal{P}}, \pi')$-chains instead. Here, π' is a filter which filters away "at least as much" as π. However, π' has to be chosen in such a way that the filtered TRSs $\pi'(DP(\mathcal{R}_{\mathcal{P}}))$ and $\pi'(\mathcal{R}_{\mathcal{P}})$ satisfy the "*variable condition*", i.e., $\mathcal{V}(\pi'(r)) \subseteq \mathcal{V}(\pi'(\ell))$ for all $\ell \to r \in DP(\mathcal{R}_{\mathcal{P}}) \cup \mathcal{R}_{\mathcal{P}}$. Then the filter π' detects all potentially infinite subterms in rewrite sequences (i.e., all subterms which correspond to "non-unification-free parts" of $\mathcal{P}$, i.e., to non-ground subterms when "executing" the program $\mathcal{P}$).

Theorem 3.17 (Proving Infinitary Termination). *Let $\mathcal{R}$ be a TRS over Σ and let π be an argument filter over Σ. We extend π to tuple symbols such that $\pi(F) = \pi(f)$ for all $f \in \Sigma_D$. Let π' be a refinement of π such that $\pi'(DP(\mathcal{R}))$ and $\pi'(\mathcal{R})$ satisfy the variable condition.*[4] *If there is no infinite $(DP(\mathcal{R}), \mathcal{R}, \pi')$-chain, then all terms $f(\vec{t})$ with $\vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma_C, \mathcal{V})$ and $\pi(f(\vec{t})) \in \mathcal{T}(\Sigma_\pi)$ are $\overset{\infty}{\to}$-terminating for $\mathcal{R}$.*

Proof. Assume there is a non-$\overset{\infty}{\to}$-terminating term $f(\vec{t})$ as above. Since $\vec{t}$ does not contain defined symbols, the first rewrite step in the infinite sequence is on the root position with a rule $\ell = f(\vec{\ell}) \to r$ where $\ell\sigma_1 = f(\vec{t})$. Since σ_1 does not introduce defined symbols, all defined symbols of $r\sigma_1$ occur on positions of r. So there is a subterm r' of r with defined root such that $r'\sigma_1$ is also non-$\overset{\infty}{\to}$-terminating. Let r' denote the smallest such subterm (i.e., for all proper subterms r'' of r', the term $r''\sigma_1$ is $\overset{\infty}{\to}$-terminating). Then $\ell^\sharp \to r'^\sharp$ is the first dependency pair of the infinite chain that we construct. Note that $\pi(\ell\sigma_1)$ and thus, $\pi(\ell^\sharp\sigma_1)$ and hence, also $\pi'(\ell^\sharp\sigma_1) = \pi'(F(\vec{t}))$ is a finite ground term by assumption. Moreover, as $\ell^\sharp \to r'^\sharp \in DP(\mathcal{R})$ and as $\pi'(DP(\mathcal{R}))$ satisfies the variable condition, $\pi'(r'^\sharp\sigma_1)$ is finite and ground as well.

The infinite sequence continues by rewriting $r'\sigma_1$'s proper subterms repeatedly. During this rewriting, the left-hand sides of rules are instantiated by constructor substitutions (i.e., substitutions with range $\mathcal{T}^{\infty}(\Sigma_C, \mathcal{V})$). As $\pi'(\mathcal{R})$ satisfies the variable condition, the terms remain finite and ground when applying the filter π'. Finally, a root rewrite step is

[4]To see why the variable condition is needed in Theorem 3.17, let $\mathcal{R} = \{\mathsf{g}(X) \to \mathsf{f}(X), \mathsf{f}(\mathsf{s}(X)) \to \mathsf{f}(X)\}$ and $\pi = \pi'$ where $\pi'(\mathsf{g}) = \varnothing$, $\pi'(\mathsf{f}) = \pi'(\mathsf{F}) = \pi'(\mathsf{s}) = \{1\}$. $\mathcal{R}$'s first rule violates the variable condition: $\mathcal{V}(\pi'(\mathsf{f}(X))) = \{X\} \not\subseteq \mathcal{V}(\pi'(\mathsf{g}(X))) = \varnothing$. There is no infinite chain, since π' does not allow us to instantiate the variable X in the dependency pair $\mathsf{F}(\mathsf{s}(X)) \to \mathsf{F}(X)$ by an infinite term. Nevertheless, there is a non-$\overset{\infty}{\to}$-terminating term $\mathsf{g}(\mathsf{s}(\mathsf{s}(\ldots)))$ which is filtered to a finite ground term $\pi'(\mathsf{g}(\mathsf{s}(\mathsf{s}(\ldots)))) = \mathsf{g}$.

performed again. Repeating this construction infinitely many times results in an infinite chain. □

The following corollary combines Theorem 3.13 and Theorem 3.17. It describes how we use the DP method for proving termination of logic programs.

Corollary 3.18 (Termination of Logic Programs by Dependency Pairs)**.**
Let $\mathcal{P}$ be a logic program and let π be an argument filter over (Σ, Δ). We extend π to $\Sigma_\mathcal{P}$ and to tuple symbols such that $\pi(p_{in}) = \pi(P_{in}) = \pi(p)$ for all $p \in \Delta$. For all other symbols f/n that are not from Σ or Δ, we define $\pi(f/n) = \{1, \ldots, n\}$. Let π' be a refinement of π such that $\pi'(DP(\mathcal{R}_\mathcal{P}))$ and $\pi'(\mathcal{R}_\mathcal{P})$ satisfy the variable condition. If there is no infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi')$-chain, then all queries $Q \in \mathcal{A}^{rat}(\Sigma, \Delta, \mathcal{V})$ with $\pi(Q) \in \mathcal{A}(\Sigma_\pi, \Delta_\pi)$ are terminating for $\mathcal{P}$.

Example 3.19. *We want to prove termination of Example 3.1 for all queries Q where $\pi(Q)$ is finite and ground for the filter $\pi(h) = \{1\}$ for all $h \in \{\mathsf{p}, \mathsf{f}, \mathsf{g}\}$. By Corollary 3.18, it suffices to show absence of infinite $(DP(\mathcal{R}), \mathcal{R}, \pi')$-chains. Here, $\mathcal{R}$ is the TRS $\{(1), \ldots, (4)\}$ from Example 3.8 and $DP(\mathcal{R})$ are Rules (5) – (8) from Example 3.15. The filter π' has to satisfy $\pi'(h) \subseteq \pi(h) = \{1\}$ for $h \in \{\mathsf{f}, \mathsf{g}\}$ and moreover, $\pi'(\mathsf{p}_{in})$ and $\pi'(\mathsf{P}_{in})$ must be subsets of $\pi(\mathsf{p}_{in}) = \pi(\mathsf{P}_{in}) = \pi(\mathsf{p}) = \{1\}$. Moreover, we have to choose π' such that the variable condition is fulfilled. So while π is always given, π' has to be determined automatically. Of course, there are only finitely many possibilities for π'. In particular, defining $\pi'(h) = \varnothing$ for all symbols h is always possible. But to obtain a successful termination proof afterwards, one should try to generate filters where the sets $\pi'(h)$ are as large as possible, since such filters provide more information about the finiteness of arguments. We will present suitable heuristics for finding such filters π' in Section 3.4. In our example, we use $\pi'(\mathsf{p}_{in}) = \pi'(\mathsf{P}_{in}) = \pi'(\mathsf{f}) = \pi'(\mathsf{g}) = \{1\}$, $\pi'(\mathsf{p}_{out}) = \pi'(\mathsf{u}_1) = \pi'(\mathsf{U}_1) = \{1, 2\}$, and $\pi'(\mathsf{u}_2) = \pi'(\mathsf{U}_2) = \{1, 2, 4\}$. For the non-well-moded Example 3.3 we choose $\pi'(\mathsf{g}) = \varnothing$ instead to satisfy the variable condition.*

So to automate the criterion of Corollary 3.18, we have to tackle two problems:

(a) We start with a given filter π describing the set of queries whose termination should be proved. Then we have to find a suitable argument filter π' that refines π in such a way that the variable condition of Theorem 3.17 is fulfilled and that the termination proof is "likely to succeed". This problem will be discussed in Section 3.4.

(b) For the chosen refined argument filter π', we have to prove that there is no infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi')$-chain. We show how to do this in the following subsection.

Automation by Adapting the DP Framework

Now we show how to prove absence of infinite $(DP(\mathcal{R}), \mathcal{R}, \pi)$-chains automatically. To this end, we adapt the *DP framework* of [GTS05a] to infinitary rewriting. In this framework, we now consider arbitrary *DP problems* $(\mathcal{D}, \mathcal{R}, \pi)$ where $\mathcal{D}$ and $\mathcal{R}$ are TRSs and π is an argument filter. Our goal is to show that there is no infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain. In this case, we call the problem *terminating*. Termination techniques should now be formulated as *DP processors* which operate on DP problems instead of TRSs. A DP processor *Proc* takes a DP problem as input and returns a new set of DP problems which then have to be solved instead. *Proc* is *sound* if for all DP problems d, d is terminating whenever all DP problems in $Proc(d)$ are terminating. So termination proofs start with the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi)$. Then this problem is transformed repeatedly by sound DP processors. If the final processors return empty sets of DP problems, then $\overset{\infty}{\to}$-termination for $\mathcal{R}$ is proved.

In Theorem 3.22, 3.24, and 3.26 we will recapitulate three of the most important existing DP processors [GTS05a] and describe how they must be modified for infinitary constructor rewriting. To this end, they now also have to take the argument filter π into account. The first processor uses an *estimated dependency graph* to estimate which dependency pairs can follow each other in chains.

Definition 3.20 (Estimated Dependency Graph)**.** *Let $(\mathcal{D}, \mathcal{R}, \pi)$ be a DP problem. The nodes of the* estimated $(\mathcal{D}, \mathcal{R}, \pi)$-dependency graph *are the pairs of $\mathcal{D}$ and there is an arc from $s \to t$ to $u \to v$ if, and only if, $CAP(t)$ and a variant u' of u unify with an mgu μ where $\pi(CAP(t)\mu) = \pi(u'\mu)$ is a finite term. Here, $CAP(t)$ replaces all subterms of t with defined root symbol by different fresh variables.*

Example 3.21. *For the DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ from Example 3.19 we obtain:*

$$\begin{tikzcd}(5) & (7) \arrow[l] \arrow[r, bend left] & (6) \arrow[l, bend left] \arrow[r] & (8)\end{tikzcd}$$

For example, there is an arc (6) $\to$ (7)*, as* $CAP(\mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y)) = \mathsf{U}_1(V, X, Y)$ *unifies with* $\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X'), \mathsf{f}(Z')), X', Y')$ *by instantiating the arguments of* U_1 *with finite terms. But there are no arcs* (5) $\to$ (5) *or* (5) $\to$ (6)*, since* $\mathsf{P}_{in}(\mathsf{f}(X), \mathsf{f}(Z))$ *and* $\mathsf{P}_{in}(\mathsf{f}(X'), \mathsf{g}(Y'))$ *do not unify, even if one instantiates Z and Y' by infinite terms (as permitted by the filter* $\pi'(\mathsf{P}_{in}) = \{1\}$*).*

Note that filters are used to *detect* potentially infinite arguments, but these arguments are not *removed*, since they can still be useful in the termination proof. In Example 3.21, they are needed to determine that (5) has no outgoing arcs.

If $s \to t, u \to v$ is a $(\mathcal{D}, \mathcal{R}, \pi)$-chain then there is an arc from $s \to t$ to $u \to v$ in the estimated dependency graph. Thus, absence of infinite chains can be proved separately for each maximal strongly connected component (SCC) of the graph. This observation is used by the following processor to modularize termination proofs by decomposing a DP problem into sub-problems. If there are n SCCs in the graph and if $\mathcal{D}_i$ are the dependency pairs of the i-th SCC (for $1 \leq i \leq n$), then one can decompose the set of dependency pairs $\mathcal{D}$ into the subsets $\mathcal{D}_1, \dots, \mathcal{D}_n$.

Theorem 3.22 (Dependency Graph Processor)**.** *For a DP problem $(\mathcal{D}, \mathcal{R}, \pi)$, let Proc return $\{(\mathcal{D}_1, \mathcal{R}, \pi), \dots, (\mathcal{D}_n, \mathcal{R}, \pi)\}$ where $\mathcal{D}_1, \dots, \mathcal{D}_n$ are the sets of nodes of the SCCs in the estimated dependency graph. Then Proc is sound.*

Proof. We prove that if $s \to t$, $u \to v$ is a chain, then there is an arc from $s \to t$ to $u \to v$ in the estimated dependency graph. This suffices for Theorem 3.22, since then every infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain corresponds to an infinite path in the graph. This path ends in an SCC with nodes $\mathcal{D}_i$ and thus, there is also an infinite $(\mathcal{D}_i, \mathcal{R}, \pi)$-chain. Hence, if all $(\mathcal{D}_i, \mathcal{R}, \pi)$ are terminating DP problems, then so is $(\mathcal{D}, \mathcal{R}, \pi)$.

Let $s \to t$, $u \to v$ be a $(\mathcal{D}, \mathcal{R}, \pi)$-chain, i.e., $t\sigma_1 \overset{\infty}{\to}{}^{*}_{\mathcal{R}} u\sigma_2$ for some constructor substitutions σ_1, σ_2 where $\pi(t\sigma_1)$ and $\pi(u\sigma_2)$ are finite. Let $pos_1, \dots, pos_n$ be the top positions where t has defined symbols. Then $CAP(t) = t[Y_1]_{pos_1} \dots [Y_n]_{pos_n}$ for fresh variables Y_j. Moreover, let the variant u' result from u by replacing every $X \in \mathcal{V}(u)$ by a fresh variable X'. Thus, the substitution σ with $\sigma(X') = \sigma_2(X)$ for all $X \in \mathcal{V}(u)$, $\sigma(X) = \sigma_1(X)$ for all $X \in \mathcal{V}(t)$, and $\sigma(Y_j) = u\sigma_2|_{pos_j}$ unifies $CAP(t)$ and u'. So there is also an mgu μ where $\sigma = \mu\tau$ for some substitution τ. Moreover, since $\pi(u\sigma_2) = \pi(u'\sigma)$ is finite, the term $\pi(u'\mu)$ is finite, too. Hence, by Definition 3.20 there is indeed an arc from $s \to t$ to $u \to v$. □

Example 3.23. *In Example 3.21, the only SCC of the dependency graph consists of (6) and (7). Thus, the dependency graph processor transforms the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ into $(\{(6), (7)\}, \mathcal{R}, \pi')$, i.e., it deletes the dependency pairs (5) and (8).*

The next processor is based on *reduction pairs* $(\succsim, \succ)$ where $\succsim$ and $\succ$ are relations on finite terms. Here, $\succsim$ is reflexive, transitive, monotonic (i.e., $s \succsim t$ implies $f(\dots s \dots) \succsim f(\dots t \dots)$ for all function symbols f), and stable (i.e., $s \succsim t$ implies $s\sigma \succsim t\sigma$ for all substitutions σ) and $\succ$ is a stable well-founded order compatible with $\succsim$ (i.e., $\succsim \circ \succ \subseteq \succ$ or $\succ \circ \succsim \subseteq \succ$). There are many techniques to search for such relations automatically (recursive path orders, polynomial interpretations, etc. [Der87]).

For a DP problem $(\mathcal{D}, \mathcal{R}, \pi)$, we now try to find a reduction pair $(\succsim, \succ)$ such that all filtered $\mathcal{R}$-rules are weakly decreasing (w.r.t. $\succsim$) and all filtered $\mathcal{D}$-dependency pairs are weakly or strictly decreasing (w.r.t. $\succsim$ or $\succ$).[5] Requiring $\pi(\ell) \succsim \pi(r)$ for all $\ell \to r \in \mathcal{R}$ ensures that in chains $s_1 \to t_1, s_2 \to t_2, \dots$ with $t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ as in Definition 3.16,

[5]We only consider *filtered* rules and dependency pairs. Thus, $\succsim$ and $\succ$ are only used to compare those parts of terms which remain *finite* for all instantiations in chains.

we have $\pi(t_i\sigma_i) \succsim \pi(s_{i+1}\sigma_{i+1})$. Hence, if a reduction pair satisfies the above conditions, then the strictly decreasing dependency pairs (i.e., those $s \to t \in \mathcal{D}$ where $\pi(s) \succ \pi(t)$) cannot occur infinitely often in chains. So the following processor deletes these pairs from $\mathcal{D}$. For any TRS $\mathcal{D}$ and any relation $\succ$, let $\mathcal{D}_{\succ_\pi} = \{s \to t \in \mathcal{D} \mid \pi(s) \succ \pi(t)\}$.

Theorem 3.24 (Reduction Pair Processor). *Let $(\succsim, \succ)$ be a reduction pair. Then the following DP processor Proc is sound. For $(\mathcal{D}, \mathcal{R}, \pi)$, Proc returns*

- $\{(\mathcal{D} \setminus \mathcal{D}_{\succ_\pi}, \mathcal{R}, \pi)\}$, *if* $\mathcal{D}_{\succ_\pi} \cup \mathcal{D}_{\succsim_\pi} = \mathcal{D}$ *and* $\mathcal{R}_{\succsim_\pi} = \mathcal{R}$
- $\{(\mathcal{D}, \mathcal{R}, \pi)\}$, *otherwise*

Proof. We prove this theorem by contradiction, i.e., we assume that $(\mathcal{D}, \mathcal{R}, \pi)$ is non-terminating and then proceed to show that $(\mathcal{D} \setminus \mathcal{D}_{\succ_\pi}, \mathcal{R}, \pi)$ has to be non-terminating, too.

From the assumption that $(\mathcal{D}, \mathcal{R}, \pi)$ is non-terminating, we know that there is an infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma_i \overset{\infty}{\to}{}^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$. For any term t we have $\pi(t\sigma) = \pi(t)\pi(\sigma)$ where $\pi(\sigma)(x) = \pi(\sigma(x))$ for all $x \in \mathcal{V}$. So by stability of $\succ$ and $\succsim$, $\mathcal{D}_{\succ_\pi} \cup \mathcal{D}_{\succsim_\pi} = \mathcal{D}$ implies

$$\pi(s_i\sigma_i) = \pi(s_i)\pi(\sigma_i) \underset{(\sim)}{\succ} \pi(t_i)\pi(\sigma_i) = \pi(t_i\sigma_i). \tag{9}$$

Note that $\pi(s_i\sigma_i)$ and $\pi(t_i\sigma_i)$ are finite. Thus, comparing them with $\succsim$ is possible. Similarly, by the observation $\pi(t\sigma) = \pi(t)\pi(\sigma)$ we also get that $t_i\sigma_i \overset{\infty}{\to}{}^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ implies $\pi(t_i\sigma_i) \overset{\infty}{\to}{}^*_{\pi(\mathcal{R})} \pi(s_{i+1}\sigma_{i+1})$. As $\mathcal{R}_{\succsim_\pi} = \mathcal{R}$ means that $\pi(\mathcal{R})$'s rules are decreasing w.r.t. $\succsim$, by monotonicity and stability of $\succsim$ we get $\pi(t_i\sigma_i) \succsim \pi(s_{i+1}\sigma_{i+1})$. With (9), this implies $\pi(s_1\sigma_1) \underset{(\sim)}{\succ} \pi(t_1\sigma_1) \succsim \pi(s_2\sigma_2) \underset{(\sim)}{\succ} \pi(t_2\sigma_2) \succsim \ldots$ As $\succ$ is compatible with $\succsim$ and well founded, $\pi(s_i\sigma_i) \succ \pi(t_i\sigma_i)$ only holds for finitely many i. So $s_j \to t_j, s_{j+1} \to t_{j+1}, \ldots$ is an infinite $(\mathcal{D} \setminus \mathcal{D}_{\succ_\pi}, \mathcal{R}, \pi)$ chain for some j and thus, the DP problem $(\mathcal{D} \setminus \mathcal{D}_{\succ_\pi}, \mathcal{R}, \pi)$ is non-terminating. □

Example 3.25. *For the DP problem $(\{(6), (7)\}, \mathcal{R}, \pi')$ in Example 3.23, one can easily find a reduction pair*[6] *where the dependency pair (7) is strictly decreasing and where (6) and all rules are weakly decreasing after applying the filter π':*

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{P}_{in}(\mathsf{f}(X)) & \succsim & \mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X) & \mathsf{p}_{in}(X) & \succsim & \mathsf{p}_{out}(X, X) \\
\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X) & \succ & \mathsf{P}_{in}(Z) & \mathsf{p}_{in}(\mathsf{f}(X)) & \succsim & \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X) \\
 & & & \mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X) & \succsim & \mathsf{u}_2(\mathsf{p}_{in}(Z), X, Z) \\
 & & & \mathsf{u}_2(\mathsf{p}_{out}(Z, \mathsf{g}(Y)), X, Z) & \succsim & \mathsf{p}_{out}(\mathsf{f}(X), \mathsf{g}(Y))
\end{array}$$

[6]For example, one can use the polynomial interpretation $|\mathsf{P}_{in}(t_1)| = |\mathsf{p}_{in}(t_1)| = |\mathsf{U}_1(t_1, t_2)| = |\mathsf{u}_1(t_1, t_2)| = |\mathsf{u}_2(t_1, t_2, t_3)| = |t_1|$, $|\mathsf{p}_{out}(t_1, t_2)| = |t_2|$, $|\mathsf{f}(t_1)| = |t_1| + 1$, and $|\mathsf{g}(t_1)| = 0$.

Thus, the reduction pair processor can remove (7) from the DP problem which results in $(\{(6)\}, \mathcal{R}, \pi')$. *By applying the dependency graph processor again, one obtains the empty set of DP problems, since now the estimated dependency graph only has the node (6) and no arcs. This proves that the initial DP problem* $(DP(\mathcal{R}), \mathcal{R}, \pi')$ *from Example 3.19 is terminating and thus, the logic program from Example 3.1 terminates for all queries* Q *where* $\pi(Q)$ *is finite and ground. Note that termination of the non-well-moded program from Example 3.3 can be shown analogously since finiteness of the initial DP problem can be proved in the same way. The only difference is that we obtain* g *instead of* $\mathsf{g}(Y)$ *in the last inequality above.*

As in Theorem 3.22 and 3.24, many other existing DP processors [GTS05a] can easily be adapted to infinitary constructor rewriting as well. Finally, one can also use the following processor to transform a DP problem $(\mathcal{D}, \mathcal{R}, \pi)$ for infinitary constructor rewriting into a DP problem $(\pi(\mathcal{D}), \pi(R), id)$ for ordinary rewriting. Afterwards, *any* existing DP processor for *ordinary* rewriting becomes applicable.[7] Since any termination technique for TRSs can immediately be formulated as a DP processor [GTS05a, Thm. 36], now any termination technique for ordinary rewriting can be directly used for infinitary constructor rewriting as well.

Theorem 3.26 (Argument Filter Processor). *Let Proc*$(\,(\mathcal{D}, \mathcal{R}, \pi)\,) = \{(\pi(\mathcal{D}), \pi(\mathcal{R}), id)\}$ *where* $id(f) = \{1, \ldots, n\}$ *for all* f/n. *Then Proc is sound.*

Proof. If $s_1 \to t_1, s_2 \to t_2, \ldots$ is an infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain with the substitutions σ_i as in Definition 3.16, then $\pi(s_1) \to \pi(t_1), \pi(s_2) \to \pi(t_2), \ldots$ is an infinite $(\pi(\mathcal{D}), \pi(\mathcal{R}), id)$-chain with the substitutions $\pi(\sigma_i)$. The reason is that $t_i\sigma_i \overset{\infty}{\to}{}^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ implies $\pi(t_i)\pi(\sigma_i) = \pi(t_i\sigma_i) \overset{\infty}{\to}{}^*_{\pi(\mathcal{R})} \pi(s_{i+1}\sigma_{i+1}) = \pi(s_{i+1})\pi(\sigma_{i+1})$. Moreover, by Definition 3.16, all terms in the rewrite sequence $\pi(t_i\sigma_i) \overset{\infty}{\to}{}^*_{\pi(\mathcal{R})} \pi(s_{i+1}\sigma_{i+1})$ are finite. □

3.4. Refining the Argument Filter

In Section 3.2 we introduced a new transformation from logic programs $\mathcal{P}$ to TRSs $\mathcal{R}_{\mathcal{P}}$ and showed that to prove the termination of a class of queries for $\mathcal{P}$, it is sufficient to analyze the termination behavior of $\mathcal{R}_{\mathcal{P}}$. Our criterion to prove termination of logic programs was summarized in Corollary 3.18.

The transformation itself is trivial to automate and as shown in Section 3.3, existing systems implementing the DP method can easily be adapted to prove termination of infinitary constructor rewriting. The missing part in the automation is the generation of a

[7] If $(\mathcal{D}, \mathcal{R}, \pi)$ results from the transformation of a logic program, then for $(\pi(\mathcal{D}), \pi(R), id)$ it is even sound to apply the existing DP processors for *innermost* rewriting [GTS05a, GTSF06]. These processors are usually more powerful than those for ordinary rewriting. The framework presented in [GTS05a] even supports constructor rewriting.

suitable argument filter from the user input, cf. Task (a) in Section 3.3. In this section, after presenting the general algorithm to refine argument filters, we introduce suitable heuristics. Finally, we extend the general algorithm for the refinement of argument filters by integrating a mode analysis based on argument filters. This allows us to handle logic programs where a predicate is used with several different modes (i.e., where different occurrences of the same predicate have different input and output positions). The usefulness of the different heuristics from this section and the power of our mode analysis will be evaluated empirically in Section 3.6.

Refinement Algorithm for Argument Filters

In our approach of Corollary 3.18, the user supplies an initial argument filter π to describe the set of queries whose termination should be proved. There are two issues with this approach. First, while argument filters provide the user with a more expressive tool to characterize classes of queries, termination problems are often rather posed in the form of a moding function for compatibility reasons. Fortunately, it is straightforward to extract an appropriate initial argument filter from such a moding function m: we define $\pi(p) = \{i \mid m(p,i) = \boldsymbol{in}\}$ for all $p \in \Delta$ and $\pi(f/n) = \{1,\dots,n\}$ for all function symbols $f/n \in \Sigma$.

Second, and less trivially, the variable condition $\mathcal{V}(\pi(r)) \subseteq \mathcal{V}(\pi(\ell))$ for all rules $\ell \to r \in DP(\mathcal{R}_\mathcal{P}) \cup \mathcal{R}_\mathcal{P}$ does not necessarily hold for the argument filter π. Thus, a refinement π' of π must be found such that the variable condition holds for π'. Then, our method from Corollary 3.18 can be applied.

Unfortunately, there are often many refinements π' of a given filter π such that the variable condition holds. The right choice of π' is crucial for the success of the termination analysis. As already mentioned in Example 3.19, the argument filter that simply filters away all arguments of all function symbols in the TRS, i.e., that has $\pi'(f) = \varnothing$ for all $f \in \Sigma_\mathcal{P}$, is a refinement of every argument filter π and it obviously satisfies the variable condition. But of course, only termination of trivial logic programs can be shown when using this refinement π'.

Example 3.27. *We consider the logic program of Example 3.1. As shown in Example 3.8, the following rule results (among others) from the translation of the logic program.*

$$\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{g}(Y)) \to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \tag{2}$$

Suppose that we want to prove termination of all queries $\mathsf{p}(t_1, t_2)$ *where both* t_1 *and* t_2 *are (finite) ground terms. This corresponds to the moding* $m(\mathsf{p},1) = m(\mathsf{p},2) = \boldsymbol{in}$*, i.e., to the initial argument filter* π *with* $\pi(\mathsf{p}) = \{1,2\}$*.*

In Corollary 3.18, we extend π *to* p_{in} *and* P_{in} *by defining it to be* $\{1,2\}$ *as well. In order* *to prove termination, we now have to find a refinement π' of π such that $\pi'(DP(\mathcal{R}_\mathcal{P}))$ and $\pi'(\mathcal{R}_\mathcal{P})$ satisfy the variable condition and such that there is no infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi')$-chain.*

Let us first try to define $\pi' = \pi$. Then π' does not filter away any arguments. Thus, $\pi'(\mathsf{p}_{in}) = \{1, 2\}$, $\pi'(\mathsf{u}_1) = \{1, 2, 3\}$, and $\pi'(\mathsf{f}) = \pi'(\mathsf{g}) = \{1\}$. But then clearly, the variable condition does not hold as Z occurs in $\pi'(r)$ but not in $\pi'(\ell)$ if $\ell \to r$ is Rule (2) above.

So we have to choose a different refinement π'. There remain three choices how we can refine π to π' in order to filter away the variable Z in the right-hand side of Rule (2): we can filter away the first argument of f by defining $\pi'(\mathsf{f}) = \varnothing$, we can filter away p_{in}'s second argument by defining $\pi(\mathsf{p}_{in}) = \{1\}$, or we can filter away the first argument of u_1 by defining $\pi(\mathsf{u}_1) = \{2, 3\}$.

The decision which of the three choices above should be taken must be done by a suitable *heuristic*. The following definition gives a formalization for such heuristics. Here we assume that the choice only depends on the term t containing a variable that leads to a violation of the variable condition and on the position *pos* of the variable. Then a *refinement heuristic* ρ is a function such that $\rho(t, pos)$ returns a function symbol f/n and an argument position $i \in \{1, \ldots, n\}$ such that filtering away the i-th argument of f would erase the position *pos* in the term t. For instance, if t is the right-hand side $\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y)$ of Rule (2) and *pos* is the position of the variable Z in this term (i.e., $pos = 121$), then $\rho(t, pos)$ can be either $(\mathsf{f}, 1)$, $(\mathsf{p}_{in}, 2)$, or $(\mathsf{u}_1, 1)$.

Definition 3.28 (Refinement Heuristic)**.** *A* refinement heuristic *is a partial mapping $\rho : \mathcal{T}(\Sigma_\mathcal{P}, \mathcal{V}) \times \mathbb{N}^* \to \Sigma_\mathcal{P} \times \mathbb{N}$ such that whenever $\rho(t, pos) = (f, i)$, then there is a position pos' with $pos'\, i$ being a prefix of pos and $root(t|_{pos'}) = f$.*

Given a TRS $\mathcal{R}_\mathcal{P}$ resulting from the transformation of a logic program $\mathcal{P}$ and a refinement heuristic ρ, Algorithm 1 computes a refinement π' of a given argument filter π such that the variable condition holds for $DP(\mathcal{R}_\mathcal{P})$ and $\mathcal{R}_\mathcal{P}$.

Termination of this algorithm is obvious as $\mathcal{R}_\mathcal{P}$ is finite and each change of the argument filter in **Step 2.2** reduces the number of unfiltered arguments. Note also that $\rho(r, pos)$ is always defined since *pos* is never the top position ε. The reason is that the TRS $\mathcal{R}_\mathcal{P}$ is non-collapsing (i.e., it has no right-hand side consisting just of a variable). The algorithm is correct as it only terminates if the variable condition holds for every dependency pair and every rule.

Note that if $\pi'(F) = \pi'(f)$ for every defined function symbol f and if we do not filter away the first argument position of the function symbols $u_{c,i}$, i.e., $1 \in \pi'(u_{c,i})$, then the satisfaction of the variable condition for $\mathcal{R}_\mathcal{P}$ implies that the variable condition for $DP(\mathcal{R}_\mathcal{P})$ holds as well. Thus, for heuristics that guarantee the above properties, we only have to consider $\mathcal{R}_\mathcal{P}$ in the above algorithm.

```
Input: argument filter $\pi$, refinement heuristic $\rho$, TRS $\mathcal{R}_\mathcal{P}$
Output: refined argument filter $\pi'$ such that $\pi'(DP(\mathcal{R}_\mathcal{P}))$ and $\pi'(\mathcal{R}_\mathcal{P})$ satisfy the
        variable condition
  1. $\pi' := \pi$
  2. If there is a rule $\ell \to r$ from $DP(\mathcal{R}_\mathcal{P}) \cup \mathcal{R}_\mathcal{P}$
     and a position $pos$ with $r|_{pos} \in \mathcal{V}(\pi'(r)) \setminus \mathcal{V}(\pi'(\ell))$, then:
    2.1. Let $(f, i)$ be the result of $\rho(r, pos)$, i.e., $(f, i) := \rho(r, pos)$.
    2.2. Modify $\pi'$ by removing $i$ from $\pi'(f)$, i.e., $\pi'(f) := \pi'(f) \setminus$
         $\{i\}$.
         For all other symbols from $\Sigma_\mathcal{P}$, $\pi'$ remains unchanged.
    2.3. Go back to Step 2.
```

Algorithm 1: General Refinement Algorithm

Simple Refinement Heuristics

The following definition introduces two simple possible refinement heuristics. If a term t has a position *pos* with a variable that violates the variable condition, then these heuristics filter away the respective argument position of the *innermost* resp. the *outermost* function symbol above the variable.

Definition 3.29 (Innermost/Outermost Refinement Heuristic)**.** *Let t be a term and let "$pos\,i$" resp. "$i\,pos$" be a position in t. The* innermost refinement heuristic ρ_{im} *is defined as follows:*

$$\rho_{im}(t, pos\,i) = (root(t|_{pos}), i)$$

The outermost refinement heuristic ρ_{om} *is defined as follows:*

$$\rho_{om}(t, i\,pos) = (root(t), i)$$

So if t is again the term $\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y)$, then the innermost refinement heuristic would result in $\rho_{im}(t, 121) = (\mathsf{f}, 1)$ and the outermost refinement heuristic gives $\rho_{om}(t, 121) = (\mathsf{u}_1, 1)$.

Both heuristics defined above are simple but problematic, as shown in Example 3.30. Filtering the innermost function symbol often results in the removal of an argument position that is relevant for termination of another rule. Filtering the outermost function symbol excludes the possibility of filtering the arguments of function symbols from the signature Σ of the original logic program. Moreover, the outermost heuristic also often removes the first argument of some $u_{c,i}$-symbol. Afterwards, a successful termination proof is hardly possible anymore.

Example 3.30. *Consider again the logic program of Example 3.1 which was transformed into the following TRS in Example 3.8.*

$$\begin{aligned}
\mathsf{p}_{in}(X,X) &\to \mathsf{p}_{out}(X,X) && (1)\\
\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) && (2)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z) && (3)\\
\mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(Y)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)) && (4)
\end{aligned}$$

As shown in Example 3.15 we obtain the following dependency pairs for the above rules.

$$\begin{aligned}
\mathsf{P}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{P}_{in}(\mathsf{f}(X),\mathsf{f}(Z)) && (5)\\
\mathsf{P}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) && (6)\\
\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{P}_{in}(Z,\mathsf{g}(Y)) && (7)\\
\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{U}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z) && (8)
\end{aligned}$$

As in Example 3.27 we want to prove termination of $\mathsf{p}(t_1,t_2)$ for all ground terms t_1 and t_2. Hence, we start with the argument filter π that does not filter away any arguments, i.e., $\pi(f/n) = \{1,\ldots,n\}$ for all $f \in \Sigma_\mathcal{P}$. We will now illustrate Algorithm 1 using our two heuristics.

Using the innermost refinement heuristic ρ_{im} in the algorithm, for the second DP (6) we get $\rho_{im}(\mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y),121) = (\mathsf{f},1)$. This requires us to filter away the only argument of f, i.e., $\pi'(\mathsf{f}) = \varnothing$. Now Z is contained in the right-hand side of the third DP (7), but not in the filtered left-hand side anymore. Thus, we now have to filter away the first argument of P_{in}, i.e., $\pi'(\mathsf{P}_{in}) = \{2\}$. Due to the DP (6), we now also have to remove the second argument X of U_1, i.e., $\pi'(\mathsf{U}_1) = \{1,3\}$. Consequently, we lose the information about finiteness of p's first argument and therefore cannot show termination of the program anymore. More precisely, there is an infinite $(DP(\mathcal{R}_\mathcal{P}),\mathcal{R}_\mathcal{P},\pi')$-chain consisting of the dependency pairs (6) and (7) using a substitution that instantiates the variables X and Z by the infinite term $\mathsf{f}(\mathsf{f}(\ldots))$. This is indeed a chain since all infinite terms are filtered away by the refined argument filter π'. Hence, the termination proof fails.

Using the outermost refinement heuristic ρ_{om} instead, for the second DP (6) we get $\rho_{om}(\mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y),121) = (\mathsf{U}_1,1)$, i.e., $\pi'(\mathsf{U}_1) = \{2,3\}$. Considering the third DP (7) we have to filter away the first argument of P_{in}, i.e., $\pi'(\mathsf{P}_{in}) = \{2\}$. Due to the DP (6), we now also have to remove the second argument of U_1, i.e., $\pi'(\mathsf{U}_1) = \{3\}$. So we obtain the same infinite chain as above since we lose the information about finiteness of p's first argument. Hence, we again cannot show termination.

A slightly improved version of the outermost refinement heuristic can be achieved by disallowing the filtering of the first arguments of the symbols $u_{c,i}$ and $U_{c,i}$.

Definition 3.31 (Improved Outermost Refinement Heuristic)**.** *Let t be a term and pos be a position in t. The* improved outermost refinement heuristic *$\rho_{om'}$ is defined as:*

$$\rho_{om'}(t, i\, pos) = \begin{cases} \rho_{om'}(t|_i, pos) & \textit{if } i = 1 \textit{ and either } root(t) = u_{c,i} \textit{ or } root(t) = U_{c,i} \\ (root(t), i) & \textit{otherwise} \end{cases}$$

Example 3.32. *Reconsider Example 3.30. Using Algorithm 1 with the improved outermost refinement heuristic, for the second rule (2) we get $\rho_{om'}(\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y), 121)$ $= \rho_{om'}(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), 21) = (\mathsf{p}_{in}, 2)$ requiring us to filter away the second argument of p_{in}, i.e., $\pi'(\mathsf{p}_{in}) = \{1\}$. Consequently, the algorithm filters away the third arguments of both u_1 and u_2, i.e., $\pi'(\mathsf{u}_1) = \{1, 2\}$ and $\pi'(\mathsf{u}_2) = \{1, 2, 4\}$. Now the variable condition holds for $\mathcal{R}_\mathcal{P}$. Therefore, by defining $\pi'(\mathsf{P}_{in}) = \pi'(\mathsf{p}_{in})$, $\pi'(\mathsf{u}_1) = \pi'(\mathsf{U}_1)$, and $\pi'(\mathsf{u}_2) = \pi'(\mathsf{U}_2)$, the variable condition also holds for $DP(\mathcal{R}_\mathcal{P})$. (As mentioned above, by filtering tuple symbols F in the same way as the original symbols f and by ensuring $1 \in \pi'(u_{c,i})$, it suffices to check the variable condition only for the rules $\mathcal{R}_\mathcal{P}$ and not for the dependency pairs $DP(\mathcal{R}_\mathcal{P})$.) This argument filter corresponds to the one chosen in Example 3.19 and as shown in Section 3.3 one can now easily prove $\overset{\infty}{\rightarrow}$-termination.*

Type-Based Refinement Heuristic

The improved outermost heuristic from Section 3.4 only filters symbols of the form p_{in}, p_{out}, P_{in}, and P_{out}. Therefore, the generated argument filters are similar to modings. However, there are cases where one needs to filter function symbols from the original logic program, too. In this section we show how to obtain a more powerful refinement heuristic using information from inferred types.

There are many approaches to (direct) termination analysis of logic programs that use type information in order to guess suitable "norms" or "ranking functions", e.g., [BCF92, BCG$^+$07, DDF93, MKS96]. In contrast to most of these approaches, we do not consider typed logic programs, but untyped ones and we use types only as a basis for a heuristic to prove termination of the transformed TRS. To our knowledge, this is the first time that types are considered in the transformational approach to termination analysis of logic programs.

Example 3.33. *Now we regard the logic program from Example 3.3. The rules after the* *transformation of Definition 3.7 are:*

$$\mathsf{p}_{in}(X,X) \to \mathsf{p}_{out}(X,X) \tag{1}$$
$$\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) \to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) \tag{2}$$
$$\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) \to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(W)),X,Y,Z) \tag{10}$$
$$\mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(W)),X,Y,Z) \to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)) \tag{11}$$

Using the improved outermost refinement heuristic $\rho_{om'}$ we start off as in Example 3.32 and obtain $\pi'(\mathsf{p}_{in}) = \{1\}$, $\pi'(\mathsf{u}_1) = \{1,2\}$, and $\pi'(\mathsf{u}_2) = \{1,2,4\}$. However, due to the last rule (11) we now get $\rho_{om'}(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)),21) = (\mathsf{p}_{out},2)$, i.e., $\pi'(\mathsf{p}_{out}) = \{1\}$. Considering the third rule (10), we have to filter p_{in} once more and obtain $\pi'(\mathsf{p}_{in}) = \varnothing$. So we again lose the information about finiteness of p's first argument and cannot show termination. Similar to Example 3.30, the innermost refinement heuristic which filters away the only argument of f also fails for this program.

So in the example above, neither the innermost nor the (improved) outermost refinement heuristic succeed. We therefore propose a better heuristic which is like the innermost refinement heuristic, but which avoids the filtering of certain arguments of original function symbols from the logic program. Close inspection of the cases where filtering such function symbols is required reveals that it is not advisable to filter away "reflexive" arguments. Here, we call an argument position i of a function symbol f *reflexive* (or "*recursive*"), if the arguments on position i have the same "type" as the whole term $f(\ldots)$ itself, cf. [Wal94]. A *type assignment* associates a predicate p/n with an n-tuple of types for its arguments and, similarly, a function f/n with an $(n+1)$-tuple where the last element specifies the result type of f.

Definition 3.34 (Types). *Let Θ be a set of* types *(i.e., a set of names). A* type assignment *τ over a signature (Σ,Δ) and a set of types Θ is a mapping $\tau : \Sigma \cup \Delta \to \Theta^*$ such that $\tau(p/n) \in \Theta^n$ for all $p/n \in \Delta$ and $\tau(f/n) \in \Theta^{n+1}$ for all $f/n \in \Sigma$. Let $f/n \in \Sigma$ be a function symbol and τ be a type assignment with $\tau(f) = (\theta_1,\ldots,\theta_n,\theta_{n+1})$. Then the* set of reflexive positions *of f/n is $Reflexive_\tau(f/n) = \{i \mid 1 \le i \le n$ and $\theta_i = \theta_{n+1}\}$.*

To infer a suitable type assignment for a logic program, we use the following simple algorithm. However, since we only use types as a heuristic to find suitable argument filters, any other type assignment would also yield a correct method for termination analysis. In other words, the choice of the type assignment only influences the power of our method, not its soundness. So unlike [BCG$^+$07], the correctness of our approach does not depend on the logic program or the query being well-typed. More sophisticated type inference algorithms were presented in [BGV05, CP98, GP02, JB92, Lu00, VB02], for example.

In our simple type inference algorithm, we define $\simeq$ as the reflexive and transitive closure of the following "similarity" relation on the argument positions: Two argument positions of (possibly different) function or predicate symbols are "similar" if there exists a program clause such that the argument positions are occupied by identical variables. Moreover, if a term $f(\ldots)$ occurs in the i-th position of a function or predicate symbol p, then the argument position of f's result is similar to the i-th argument position of p. (For a function symbol f/n we also consider the argument position $n+1$ which stands for the result of the function.) After having computed the relation $\simeq$, we then use a type assignment which corresponds to the equivalence classes imposed by $\simeq$. So our simple type inference algorithm is related to sharing analysis [BDB+96, CF99, LS02], i.e., the program analysis that aims at detecting program variables that in some program execution might be bound to terms having a common variable.

Example 3.35. *As an example, we compute a suitable type assignment for the logic program from Example 3.3:*

$$\begin{array}{lll} \mathsf{p}(X,X). & & \\ \mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y)) & \leftarrow & \mathsf{p}(\mathsf{f}(X),\mathsf{f}(Z)),\mathsf{p}(Z,\mathsf{g}(W)). \end{array}$$

Let p_i denote the i-th argument position of p, etc. Then due to the first clause we obtain $\mathsf{p}_1 \simeq \mathsf{p}_2$, since both argument positions are occupied by the variable X. Moreover, since Z occurs both in the first argument positions of f and p in the second clause, we also have $\mathsf{p}_1 \simeq \mathsf{f}_1$. Finally, since an f-term occurs in the first and second argument of p and since a g-term occurs in the second argument of p we also have $\mathsf{f}_2 \simeq \mathsf{p}_1 \simeq \mathsf{p}_2$ and $\mathsf{g}_2 \simeq \mathsf{p}_2$. In other words, the relation $\simeq$ imposes the two equivalence classes $\{\mathsf{p}_1,\mathsf{p}_2,\mathsf{f}_1,\mathsf{f}_2,\mathsf{g}_2\}$ and $\{\mathsf{g}_1\}$. Hence, we compute a type assignment with two types a and b where a and b correspond to $\{\mathsf{p}_1,\mathsf{p}_2,\mathsf{f}_1,\mathsf{f}_2,\mathsf{g}_2\}$ and $\{\mathsf{g}_1\}$, respectively. Thus, the type assignment is defined as $\tau(\mathsf{p}) = \tau(\mathsf{f}) = (a,a)$ and $\tau(\mathsf{g}) = (b,a)$.

Note that the first argument of f has the same type a as its result and hence, this argument position is reflexive. On the other hand, the first argument of g has a different type than its result and is therefore not reflexive. Hence, $Reflexive_\tau(\mathsf{f}) = \{1\}$ and $Reflexive_\tau(\mathsf{g}) = \varnothing$.

Now we can define the following heuristic based on type assignments. It is like the innermost refinement heuristic of Definition 3.29, but now reflexive arguments of function symbols from Σ (i.e., from the original logic program) are not filtered away.

Definition 3.36 (Type-based Refinement Heuristic)**.** *Let t be a term, let "$pos\,i$" be a position in t, and let τ be a type assignment. The* type-based refinement heuristic *ρ^τ_{tb} is defined as follows:*

$$\rho^\tau_{tb}(t, pos\,i) \;=\; \begin{cases} (root(t|_{pos}), i) & \text{if } root(t|_{pos}) \notin \Sigma \text{ or } i \notin Reflexive_\tau(root(t|_{pos})) \\ \rho^\tau_{tb}(t, pos) & \text{otherwise} \end{cases}$$

Note that the heuristic ρ^{τ}_{tb} never filters away the first argument of a symbol $u_{c,i}$ or $U_{c,i}$ from the TRSs $DP(\mathcal{R}_{\mathcal{P}})$ and $\mathcal{R}_{\mathcal{P}}$. Therefore, as mentioned above, we only have to check the variable condition for the rules of $\mathcal{R}_{\mathcal{P}}$, but not for the dependency pairs.

Example 3.37. *We continue with the logic program from Example 3.3 and use the type assignment computed in Example 3.35 above. The rules after the transformation of Definition 3.7 are the following, cf. Example 3.33.*

$$\begin{aligned}
\mathsf{p}_{in}(X,X) &\to \mathsf{p}_{out}(X,X) && (1)\\
\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) && (2)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(W)),X,Y,Z) && (10)\\
\mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(W)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)) && (11)
\end{aligned}$$

Due to the occurrence of Z in the right-hand side of the second rule (2), we compute:

$$\begin{aligned}
& \rho^{\tau}_{tb}(\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y),121) & \\
= \ & \rho^{\tau}_{tb}(\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y),12) & \textit{as } \mathsf{f}\in\Sigma \textit{ and } 1\in \textit{Reflexive}_{\tau}(\mathsf{f})\\
= \ & (\mathsf{p}_{in},2) & \textit{as } \mathsf{p}_{in}\notin\Sigma
\end{aligned}$$

Thus, we filter away the second argument of p_{in}, i.e., $\pi'(\mathsf{p}_{in}) = \{1\}$. Consequently, we obtain $\pi'(\mathsf{u}_1) = \{1,2\}$ and $\pi'(\mathsf{u}_2) = \{1,2,4\}$.

Considering the fourth rule (11) we compute:

$$\begin{aligned}
& \rho^{\tau}_{tb}(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)),21) & \\
= \ & (\mathsf{g},1) & \textit{as } 1\notin \textit{Reflexive}_{\tau}(\mathsf{g})
\end{aligned}$$

Thus, we filter away the only argument of g*, i.e., $\pi'(\mathsf{g}) = \varnothing$. By filtering the tuple symbols in the same way as the corresponding "lower-case" symbols, now the variable condition holds for $\mathcal{R}_{\mathcal{P}}$ and therefore also for $DP(\mathcal{R}_{\mathcal{P}})$. Indeed, this is the argument filter chosen in Example 3.19. With this filter, one can easily prove termination of the program, cf. Section 3.3.*

For the above example, it is sufficient only to avoid the filtering of reflexive positions. However, in general one should also avoid the filtering of all "unbounded" argument positions. An argument position of type θ is "unbounded" if it may contain subterms from a recursive data structure, i.e., if there exist infinitely many terms of type θ. The decrease of the terms on such argument positions might be the reason for the termination of the program and therefore, they should not be filtered away. To formalize the concept of unbounded argument positions, we define the set of *constructors* of a type θ to consist of all function symbols whose result has type θ. Then an argument position of a function symbol f is *unbounded* if it is reflexive or if it has a type θ with a constructor that has an unbounded argument position. For the sake of brevity, we also speak of just *unbounded positions* when referring to unbounded argument positions.

Definition 3.38 (Unbounded Positions)**.** *Let $\theta \in \Theta$ be a type and τ be a type assignment. A function symbol f/n with $\tau(f/n) = (\theta_1, \dots, \theta_n, \theta_{n+1})$ is a* constructor *of θ if, and only if, $\theta_{n+1} = \theta$. Let $Constructors_\tau(\theta)$ be the set of all constructors of θ.*

For such a function symbol f/n, we define the set of unbounded positions *as the smallest set such that $Reflexive_\tau(f/n) \subseteq Unbounded_\tau(f/n)$ and such that $i \in Unbounded_\tau(f/n)$ if there is a $g/m \in Constructors_\tau(\theta_i)$ and a $1 \le j \le m$ with $j \in Unbounded_\tau(g/m)$.*

In the logic program from Examples 3.3 and 3.35, we had $\tau(\mathsf{p}) = \tau(\mathsf{f}) = (a, a)$ and $\tau(\mathsf{g}) = (b, a)$. Thus, $Constructors_\tau(a) = \{\mathsf{f}, \mathsf{g}\}$ and $Constructors_\tau(b) = \varnothing$. Since the first argument position of f is reflexive, it is also unbounded. The first argument position of g is not unbounded, since it is not reflexive and there is no constructor of type b with an unbounded argument position. So in this example, there is no difference between reflexive and unbounded positions.

However, we will show in Example 3.40 that there are programs where these two notions differ. For that reason, we now improve our type-based refinement heuristic and disallow the filtering of unbounded (instead of just reflexive) positions.

Definition 3.39 (Improved Type-based Refinement Heuristic)**.** *Let t be a term, let "$pos\ i$" be a position in t, and let τ be a type assignment. The* improved type-based refinement heuristic *$\rho^\tau_{tb'}$ is defined as follows:*

$$\rho^\tau_{tb'}(t, pos\ i) \quad = \quad \begin{cases} (root(t|_{pos}), i) & \text{if } root(t|_{pos}) \notin \Sigma \text{ or } i \notin Unbounded_\tau(root(t|_{pos})) \\ \rho^\tau_{tb'}(t, pos) & \text{otherwise} \end{cases}$$

Example 3.40. *The following logic program inverts an integer represented by a sign (*neg *or* pos*) and by a natural number in Peano notation (using* s *and* 0*). So the integer number 1 is represented by the term* pos(s(0))*, the integer number -1 is represented by* neg(s(0))*, and the integer number 0 has the two representations* pos(0) *and* neg(0)*. Here* nat(t) *holds if, and only if, t represents a natural number (i.e., if t is a term containing just* s *and* 0*) and* inv *simply exchanges the function symbols* neg *and* pos*. The main predicate* safeinv *performs the desired inversion where* safeinv(t_1, t_2) *only holds if t_1 really represents an integer number and t_2 is its inversion.*

$$\begin{array}{lcl}
\mathsf{nat}(\mathsf{0}). & & \\
\mathsf{nat}(\mathsf{s}(X)) & \leftarrow & \mathsf{nat}(X). \\
\mathsf{inv}(\mathsf{neg}(X), \mathsf{pos}(X)). & & \\
\mathsf{inv}(\mathsf{pos}(X), \mathsf{neg}(X)). & & \\
\mathsf{safeinv}(X, \mathsf{neg}(Y)) & \leftarrow & \mathsf{inv}(X, \mathsf{neg}(Y)), \mathsf{nat}(Y). \\
\mathsf{safeinv}(X, \mathsf{pos}(Y)) & \leftarrow & \mathsf{inv}(X, \mathsf{pos}(Y)), \mathsf{nat}(Y).
\end{array}$$

The rules after the transformation of Definition 3.7 are:

$$\mathsf{nat}_{in}(\mathsf{0}) \to \mathsf{nat}_{out}(\mathsf{0}) \tag{12}$$
$$\mathsf{nat}_{in}(\mathsf{s}(X)) \to \mathsf{u}_1(\mathsf{nat}_{in}(X), X) \tag{13}$$
$$\mathsf{u}_1(\mathsf{nat}_{out}(X), X) \to \mathsf{nat}_{out}(\mathsf{s}(X)) \tag{14}$$
$$\mathsf{inv}_{in}(\mathsf{neg}(X), \mathsf{pos}(X)) \to \mathsf{inv}_{out}(\mathsf{neg}(X), \mathsf{pos}(X)) \tag{15}$$
$$\mathsf{inv}_{in}(\mathsf{pos}(X), \mathsf{neg}(X)) \to \mathsf{inv}_{out}(\mathsf{pos}(X), \mathsf{neg}(X)) \tag{16}$$
$$\mathsf{safeinv}_{in}(X, \mathsf{neg}(Y)) \to \mathsf{u}_2(\mathsf{inv}_{in}(X, \mathsf{neg}(Y)), X, Y) \tag{17}$$
$$\mathsf{u}_2(\mathsf{inv}_{out}(X, \mathsf{neg}(Y)), X, Y) \to \mathsf{u}_3(\mathsf{nat}_{in}(Y), X, Y) \tag{18}$$
$$\mathsf{u}_3(\mathsf{nat}_{out}(Y), X, Y) \to \mathsf{safeinv}_{out}(X, \mathsf{neg}(Y)) \tag{19}$$
$$\mathsf{safeinv}_{in}(X, \mathsf{pos}(Y)) \to \mathsf{u}_4(\mathsf{inv}_{in}(X, \mathsf{pos}(Y)), X, Y) \tag{20}$$
$$\mathsf{u}_4(\mathsf{inv}_{out}(X, \mathsf{pos}(Y)), X, Y) \to \mathsf{u}_5(\mathsf{nat}_{in}(Y), X, Y) \tag{21}$$
$$\mathsf{u}_5(\mathsf{nat}_{out}(Y), X, Y) \to \mathsf{safeinv}_{out}(X, \mathsf{pos}(Y)) \tag{22}$$

Let us assume that the user wants to prove termination of all queries $\mathsf{safeinv}(t_1, t_2)$ *where* t_1 *is ground. So we use the moding* $m(\mathsf{safeinv}, 1) = \boldsymbol{in}$ *and* $m(\mathsf{safeinv}, 2) = \boldsymbol{out}$*. Thus, as initial argument filter* π *we have* $\pi(\mathsf{safeinv}) = \{1\}$ *and hence* $\pi(\mathsf{safeinv}_{in}) = \pi(\mathsf{SAFEINV}_{in}) = \{1\}$*, while* $\pi(f/n) = \{1, \ldots, n\}$ *for all* $f \notin \{\mathsf{safeinv}, \mathsf{safeinv}_{in}, \mathsf{SAFEINV}_{in}\}$*. In Rule (17) one has to filter away the second argument of* inv_{in} *or the only argument of* neg *in order to remove the "extra" variable* Y *on the right-hand side. From a type inference for these rules we obtain the type assignment* τ *with* $\tau(\mathsf{s}) = (b, b)$*,* $\tau(\mathsf{0}) = (b)$*, and* $\tau(\mathsf{neg}) = \tau(\mathsf{pos}) = (b, a)$*. So "a" corresponds to the type of integers and "b" corresponds to the type of naturals. The constructors of the naturals are* $\mathit{Constructors}_\tau(b) = \{\mathsf{s}, \mathsf{0}\}$*. This is a recursive data structure since* s *has an unbounded argument:* $1 \in \mathit{Reflexive}_\tau(\mathsf{s}) \subseteq \mathit{Unbounded}_\tau(\mathsf{s})$*. Thus, while* neg*'s first argument position of type* b *is not reflexive, it is still unbounded, i.e.,* $1 \in \mathit{Unbounded}_\tau(\mathsf{neg})$*. Hence, our improved type-based heuristic decides to filter away the second argument of* inv_{in} *(as* inv_{in} *is not from the original signature* Σ*). Now termination is easy to show.*

If one had considered the original type-based heuristic instead, then the non-reflexive first argument of neg *would be filtered away. Due to Rule (17), then also the last argument of* u_2 *has to be removed by the filter. But then the variable* Y *would not occur anymore in the filtered left-hand side of Rule (18). So to satisfy the variable condition for Rule (18), we would have to filter away the only argument of* nat_{in}*. Similarly, the only argument of the corresponding tuple symbol* NAT_{in} *would also be filtered away, blocking any possibility for a successful termination proof.*

Mode Analysis based on Argument Filters and an Improved Refinement Algorithm

In logic programming, it is not unusual that a predicate is used with different modes (i.e., that different occurrences of the predicate have different input and output positions). Uniqueness of moding can then be achieved by creating appropriate copies of these predicate symbols and their clauses for every different moding.

Example 3.41. *Consider the following logic program for rotating a list taken from [Cod07]. Let $\mathcal{P}$ be the* append*-program consisting of the clauses from Example 3.4 and the new clause*

$$\mathsf{rotate}(N, O) \leftarrow \mathsf{append}(L, M, N), \mathsf{append}(M, L, O). \tag{23}$$

with the moding $m(\mathsf{rotate}, 1) = \boldsymbol{in}$ and $m(\mathsf{rotate}, 2) = \boldsymbol{out}$. For this moding, the program is terminating.

But while the first use of append *in Clause (23) supplies it with a ground term only on the last argument position, the second use in (23) is with ground terms only on the first two argument positions. Although the* append*-clauses are even well moded for both kinds of uses, the whole program is not.*

The logic program is transformed into the following TRS. As before, "$[X|L]$" is an abbreviation for $\bullet(X, L)$, i.e., $\bullet$ is the constructor for list insertion.

$$\mathsf{append}_{in}([\,], M, M) \to \mathsf{append}_{out}([\,], M, M) \tag{24}$$
$$\mathsf{append}_{in}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1(\mathsf{append}_{in}(L, M, N), X, L, M, N) \tag{25}$$
$$\mathsf{u}_1(\mathsf{append}_{out}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}(\bullet(X, L), M, \bullet(X, N)) \tag{26}$$
$$\mathsf{rotate}_{in}(N, O) \to \mathsf{u}_2(\mathsf{append}_{in}(L, M, N), N, O) \tag{27}$$
$$\mathsf{u}_2(\mathsf{append}_{out}(L, M, N), N, O) \to \mathsf{u}_3(\mathsf{append}_{in}(M, L, O), L, M, N, O) \tag{28}$$
$$\mathsf{u}_3(\mathsf{append}_{out}(M, L, O), L, M, N, O) \to \mathsf{rotate}_{out}(N, O) \tag{29}$$

Due to the "extra" variables L and M in the right-hand side of Rule (27) and the "extra" variable O in the right-hand side of Rule (28),[8] *the only refined argument filter which would satisfy the variable condition of Corollary 3.18 is the one where $\pi(\mathsf{append}_{in}) = \varnothing$.*[9] *As we can expect, for the queries described by this filter, the* append*-program is not terminating and, thus, our new approach fails, too.*

[8]In the left-hand side of Rule (27), the variable O in the second argument of rotate_{in} is removed by the initial filter that describes the desired set of queries given by the user. Consequently, one also has to filter away the last argument of u_2. Hence, then O is indeed an "extra" variable in the right-hand side of Rule (28).

[9]Alternatively, one could also filter away the first arguments of u_2 and u_3. But then one would also have to satisfy the variable condition for the dependency pairs and one would obtain $\pi(\mathsf{APPEND}_{in}) = \varnothing$. Hence, the termination proof attempt would fail as well.

The common solution [Apt97] is to produce two copies of the append*-clauses and to rename them apart. This is often referred to as "mode-splitting". First, we create labeled copies of the predicate symbol* append *and label the predicate of each* append*-atom by the input positions of the moding in which it is used. Then, we extend our moding to* $m(\mathsf{append}^{\{3\}}, 3) = m(\mathsf{append}^{\{1,2\}}, 1) = m(\mathsf{append}^{\{1,2\}}, 2) = \boldsymbol{in}$ *and* $m(\mathsf{append}^{\{3\}}, 1) = m(\mathsf{append}^{\{3\}}, 2) = m(\mathsf{append}^{\{1,2\}}, 3) = \boldsymbol{out}$*. In our example, termination of the resulting logic program can easily be shown using both the classical transformation from Section 3 or our new transformation:*

$$\begin{array}{lll}
\mathsf{rotate}(N, O) & \leftarrow & \mathsf{append}^{\{3\}}(L, M, N), \mathsf{append}^{\{1,2\}}(M, L, O). \\
\mathsf{append}^{\{3\}}([\,], M, M). & & \\
\mathsf{append}^{\{3\}}([X|L], M, [X|N]) & \leftarrow & \mathsf{append}^{\{3\}}(L, M, N). \\
\mathsf{append}^{\{1,2\}}([\,], M, M). & & \\
\mathsf{append}^{\{1,2\}}([X|L], L, [X|N]) & \leftarrow & \mathsf{append}^{\{1,2\}}(L, M, N).
\end{array}$$

In the example above, a pre-processing based on modings was sufficient for a successful termination proof. In general, though, this is insufficient to handle queries described by an argument filter. The following example demonstrates this.

Example 3.42. *Consider again the logic program* $\mathcal{P}$ *from Example 3.41 which is translated to the TRS* $\mathcal{R}_\mathcal{P} = \{(24), \ldots, (29)\}$*. This time we want to show termination for all queries of the form* $\mathsf{rotate}(t_1, t_2)$ *where* t_1 *is a finite list (possibly containing non-ground terms as elements). So* t_1 *is instantiated by terms of the form* $\bullet(r_1, \bullet(r_2, \ldots \bullet(r_n, [\,]) \ldots))$ *where the* r_i *can be arbitrary terms possibly containing variables.*[10]

To specify these queries, the user would provide the initial argument filter π *with* $\pi(\mathsf{rotate}) = \{1\}$ *and* $\pi(\bullet) = \{2\}$*. Now our aim is to prove termination of all queries that are ground under the filter* π*. Thus, the first argument of* rotate *is not necessarily a ground term (it is only guaranteed to be ground* after filtering away the second argument of $\bullet$*).*

Therefore, if one wanted to pre-process the program using modings, then one could not assume that the first argument of rotate *were ground. Instead, one would have to use the moding* $m(\mathsf{rotate}, 1) = m(\mathsf{rotate}, 2) = \boldsymbol{out}$*. Therefore, in the calls to* append*, all argument positions would be considered as "**out**". As a consequence, no renamed-apart copies of clauses would be created and the termination proof would fail.*

[10]Such a termination problem can also result from an initial termination problem that was described by modings. To demonstrate this, we could extend the program by the following clauses.

$$\begin{array}{lll}
\mathsf{p}(X, O) & \leftarrow & \mathsf{s2}\ell(X, N), \mathsf{rotate}(N, O). \\
\mathsf{s2}\ell(0, [\,]). & & \\
\mathsf{s2}\ell(s(X), [Y|N]) & \leftarrow & \mathsf{s2}\ell(X, N).
\end{array}$$

To prove termination of all queries described by the moding $m(\mathsf{p}, 1) = \boldsymbol{in}$ and $m(\mathsf{p}, 2) = \boldsymbol{out}$, one essentially has to show termination for all queries of the form $\mathsf{rotate}(t_1, t_2)$ where t_1 is a finite list.

In general, Algorithm 1 aims to compute an argument filter that filters away as few arguments as possible while ensuring that the variable condition holds. In this way we make sure that the maximal amount of information remains for the following termination analysis.

But as Examples 3.41 and 3.42 above demonstrate, there are cases where we need to create renamed-apart copies of clauses for certain predicates in order to obtain a viable refined argument filter. To this end, a first idea might be to combine an existing mode inference algorithm with Algorithm 1. However, it is not clear how to do such a combination. The problem is that we already need to know the refined argument filter in order to create suitable copies of clauses. At the same time, we already need the renamed-apart copies of the clauses in order to compute the refined argument filter. Thus, we have a classical "chicken-and-egg" problem. Moreover, such an approach would always fail for programs like Example 3.42 where there exists no suitable pre-processing based on modings.

Therefore, we replace Algorithm 1 by the following new Algorithm 2 that simultaneously refines the argument filter and creates renamed-apart copies on demand.

The idea of the algorithm is the following. Whenever our refinement heuristic suggests to filter away an argument of a symbol p_{in}, then instead of changing the argument filter appropriately, we introduce a new copy of the symbol p_{in}. To distinguish the different copies of the symbols p_{in}, we label them by the argument positions that are not filtered away.

In general, a removal of argument positions of p_{in} can already be performed by the initial filter π that the user provides in order to describe the desired set of queries. Therefore, if $\pi(p)$ does not contain all arguments $\{1, \dots, n\}$ for some predicate symbol p/n, then we already introduce a new symbol $p_{in}^{\pi(p)}$ and new copies of the rewrite rules originating from p. In these rules, we use the new symbol $p_{in}^{\pi(p)}$ instead of p_{in}.

Let us reconsider Example 3.42. To prove termination of all queries $\mathsf{rotate}(t_1, t_2)$ with a finite list t_1, the user would select the argument filter π that eliminates the second argument of rotate and the first argument of the list constructor $\bullet$. So we have $\pi(\mathsf{rotate}) = \{1\}$, $\pi(\bullet) = \{2\}$, and $\pi(\mathsf{append}) = \{1, 2, 3\}$. Then in addition to the rules (27) – (29) for the symbol rotate_{in} we also introduce the symbol $\mathsf{rotate}_{in}^{\{1\}}$. Moreover, in order to ensure that $\mathsf{rotate}_{in}^{\{1\}}$ does the same computation as rotate_{in}, we add the following copies of the rewrite rules (27) – (29) originating from the predicate rotate. Here, all root symbols of left- and right-hand sides are labeled with $\{1\}$.

$$\mathsf{rotate}_{in}^{\{1\}}(N, O) \to \mathsf{u}_2^{\{1\}}(\mathsf{append}_{in}(L, M, N), N, O) \tag{30}$$

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}(L, M, N), N, O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}(M, L, O), L, M, N, O) \tag{31}$$

$$\mathsf{u}_3^{\{1\}}(\mathsf{append}_{out}(M, L, O), L, M, N, O) \to \mathsf{rotate}_{out}^{\{1\}}(N, O) \tag{32}$$

```
Input: argument filter $\pi$, refinement heuristic $\rho$, TRS $\mathcal{R}_\mathcal{P}$
Output: refined argument filter $\pi'$ and modified TRS $\mathcal{R}'_\mathcal{P}$
        such that $\pi'(\mathcal{R}'_\mathcal{P})$ satisfies the variable condition

1. $\mathcal{R}'_\mathcal{P} := \mathcal{R}_\mathcal{P} \cup \{\ell^{\pi(p)} \to r^{\pi(p)} \mid \ell \to r \in \mathcal{R}_\mathcal{P}(p),\ p/n \in \Delta,\ \pi(p) \subsetneq \{1,\dots,n\}\}$

2. $\pi'(f) := \begin{cases} \pi(f), & \text{for all } f \in \Sigma \text{ (i.e., for functions of } \mathcal{P}) \\ I, & \text{for all } f = p_{in}^I \text{ with } p \in \Delta \\ \{1,\dots,n\}, & \text{for all other symbols } f/n \end{cases}$

3. If there is a rule $\ell \to r$ from $\mathcal{R}'_\mathcal{P}$
   and a position $pos$ with $r|_{pos} \in \mathcal{V}(\pi'(r)) \setminus \mathcal{V}(\pi'(\ell))$, then:

   3.1. Let $(f,i)$ be the result of $\rho(r,pos)$, i.e., $(f,i) := \rho(r,pos)$.

   3.2. We perform a case analysis depending on whether $f$ has
        the form $p_{in}^I$ for some $p \in \Delta$. Here, unlabeled symbols of
        the form $p_{in}/n$ are treated as if they were labeled with
        $I = \{1,\dots,n\}$.

        • If $f = p_{in}^I$, then we must have $r = u(p_{in}^I(...),\dots)$
          for some symbol $u$. We introduce a new function
          symbol $p_{in}^{I\setminus\{i\}}$ with $\pi'(p_{in}^{I\setminus\{i\}}) = I \setminus \{i\}$ if it has not
          yet been introduced. Then:

            ◦ We replace $p_{in}^I$ by $p_{in}^{I\setminus\{i\}}$ in the right-hand side
              of $\ell \to r$:

                $\mathcal{R}'_\mathcal{P} := \mathcal{R}'_\mathcal{P} \setminus \{\ell \to r\} \cup \{\ell \to \overline{r}\}$,

              where $\overline{r} = u(p_{in}^{I\setminus\{i\}}(...),\dots)$.

            ◦ $\mathcal{R}'_\mathcal{P} := \mathcal{R}'_\mathcal{P} \cup \{s^{I\setminus\{i\}} \to t^{I\setminus\{i\}} \mid s \to t \in \mathcal{R}'_\mathcal{P}(p)\}$.
              If this introduces new labeled function symbols
              $f/n$ where $\pi'$ was not yet defined on, we define
              $\pi'(f) = \{1,\dots,n\}$.

            ◦ Let $\ell' \to r'$ be the rule in $\mathcal{R}'_\mathcal{P}$ with $\ell' =$
              $u(p_{out}^I(...),\dots)$. We now replace $p_{out}^I$ by $p_{out}^{I\setminus\{i\}}$
              in the left-hand side of $\ell' \to r'$:

                $\mathcal{R}'_\mathcal{P} := \mathcal{R}'_\mathcal{P} \setminus \{\ell' \to r'\} \cup \{\overline{\ell'} \to r'\}$,

              where $\overline{\ell'} = u(p_{out}^{I\setminus\{i\}}(...),\dots)$.

        • Otherwise (i.e., if $f$ does not have the form $p_{in}$ or
          $p_{in}^I$), then modify $\pi'$ by removing $i$ from $\pi'(f)$, i.e.,
          $\pi'(f) := \pi'(f) \setminus \{i\}$.

   3.3. Go back to Step 3.
```

Algorithm 2: Improved Refinement Algorithm

So in **Step 1** of the algorithm, we initialize $\mathcal{R}'_\mathcal{P}$ to contain all rules of $\mathcal{R}_\mathcal{P}$. But in addition, $\mathcal{R}'_\mathcal{P}$ contains labeled copies of the rules resulting from those predicates p/n where $\pi(p) \subsetneq \{1, \ldots, n\}$. In these rules, the root symbols of left- and right-hand sides are labeled with $\pi(p)$.

Formally, for every predicate symbol $p \in \Delta$, let $\mathcal{R}_\mathcal{P}(p)$ denote those rules of $\mathcal{R}_\mathcal{P}$ which result from p-clauses (i.e., from clauses whose head is built with the predicate p). So $\mathcal{R}_\mathcal{P}(\mathsf{rotate})$ consists of the rule for rotate_{in} and the rules for u_2 and u_3, i.e., $\mathcal{R}_\mathcal{P}(\mathsf{rotate}) = \{(27), (28), (29)\}$.

Then for a term $t = f(t_1, \ldots, t_n)$ and a set of argument positions $I \subseteq \mathbb{N}$, let t^I denote $f^I(t_1, \ldots, t_n)$. So for $t = \mathsf{rotate}_{in}(N, O)$ and $I = \{1\}$, we have $t^I = \mathsf{rotate}_{in}^{\{1\}}(N, O)$. Hence if $\pi(\mathsf{rotate}) = \{1\}$, then we extend $\mathcal{R}'_\mathcal{P}$ by copies of the rules in $\mathcal{R}_\mathcal{P}(\mathsf{rotate})$ where the root symbols are labeled by $\{1\}$. In other words, we have to add the rules $\{\ell^{\pi(p)} \to r^{\pi(p)} \mid \ell \to r \in \mathcal{R}_\mathcal{P}(\mathsf{rotate})\} = \{(30), (31), (32)\}$.

In **Step 2**, we initialize our desired argument filter π'. This filter does not yet eliminate any arguments except for original function symbols from the logic program and for symbols of the form p^I_{in}. Since in our example, the initial argument filter π of the user is $\pi(\mathsf{rotate}) = \{1\}$, we have $\pi'(\mathsf{rotate}_{in}) = \{1, 2\}$, but $\pi'(\mathsf{rotate}_{in}^{\{1\}}) = \{1\}$. So for symbols p^I_{in}, the label I describes those arguments that are not filtered away. However, this does not hold for the other labeled symbols. So the labelling of the symbols $\mathsf{u}_2^{\{1\}}$, $\mathsf{u}_3^{\{1\}}$, and $\mathsf{append}_{out}^{\{1\}}$ only represents that they "belong" to the symbol $\mathsf{rotate}_{in}^{\{1\}}$. But the argument filter for these symbols can be determined arbitrarily. Initially, π' would not filter away any of their arguments, i.e., $\pi'(\mathsf{u}_2^{\{1\}}) = \{1, 2, 3\}$, $\pi'(\mathsf{u}_3^{\{1\}}) = \{1, 2, 3, 4, 5\}$, and $\pi'(\mathsf{rotate}_{out}^{\{1\}}) = \{1, 2\}$. The filter for original function symbols of the logic program is taken from the user-defined argument filter π. So since the user described the desired set of queries by setting $\pi(\bullet) = \{2\}$, we also have $\pi'(\bullet) = \{2\}$.

In **Steps 3** and **3.1**, we look for rules violating the variable condition as in Algorithm 1. Again, we use a refinement heuristic ρ to suggest a suitable function symbol f and an argument position i that should be filtered away. As before, we restrict ourselves to refinement heuristics ρ which never select the first argument of a symbol $u_{c,i}$. In this way, we only have to examine the rules (and not also the dependency pairs) for possible violations of the variable condition.

If f is not a (possibly labeled) symbol of the form p_{in} or p^I_{in}, then we proceed in **Step 3.2** as before (i.e., as in Step 2.2 of Algorithm 1). But if f is a (possibly labeled) symbol of the form p_{in} or p^I_{in}, then we do not modify the filter for f. If I are the non-filtered argument positions of f, then we introduce a new function symbol labeled with $I \setminus \{i\}$ instead and replace f by this new function symbol in the rule that violated the variable condition.

In our example, we had $\mathcal{R}'_{\mathcal{P}} = \{(24), \ldots, (29), (30), (31), (32)\}$ and π' was the filter that does not eliminate any arguments except for $\pi'(\mathsf{rotate}^{\{1\}}_{in}) = \{1\}$ and $\pi'(\bullet) = \{2\}$.

The rules (25), (27), and (30) violate the variable condition. In the following, we mark the violating variables by boxes. Let us regard Rule (25) first:

$$\mathsf{append}_{in}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1(\mathsf{append}_{in}(L, M, N), \boxed{X}, L, M, N) \qquad (25)$$

To remove the variable X from the right-hand side, in **Step 3.1** any refinement heuristic must suggest to filter away the second argument of u_1. As u_1 does not have the form p^I_{in}, we use the second case of **Step 3.2**. Thus, we change π' such that $\pi'(\mathsf{u}_1) = \{1, 2, 3, 4, 5\} \setminus \{2\} = \{1, 3, 4, 5\}$. Indeed, now this rule does not violate the variable condition anymore.

We reach **Step 3.3** and, thus, go back to **Step 3** where we again choose a rule that violates the variable condition. Let us now regard Rule (30):

$$\mathsf{rotate}^{\{1\}}_{in}(N, O) \to \mathsf{u}^{\{1\}}_2(\mathsf{append}_{in}(\boxed{L}, \boxed{M}, N), N, \boxed{O}) \qquad (30)$$

To remove the first violating variable L, in **Step 3.1** our refinement heuristic suggests to filter away the first argument of the symbol append_{in}. But instead of changing $\pi'(\mathsf{append}_{in})$, we introduce a new symbol $\mathsf{append}^{\{2,3\}}_{in}$ with $\pi'(\mathsf{append}^{\{2,3\}}_{in}) = \{2, 3\}$. Moreover, we replace the symbol append_{in} in the right-hand side of Rule (30) by the new symbol $\mathsf{append}^{\{2,3\}}_{in}$. Thus, Rule (30) is modified to

$$\mathsf{rotate}^{\{1\}}_{in}(N, O) \to \mathsf{u}^{\{1\}}_2(\mathsf{append}^{\{2,3\}}_{in}(L, \boxed{M}, N), N, \boxed{O}). \qquad (33)$$

To make sure that $\mathsf{append}^{\{2,3\}}_{in}$ has rewrite rules corresponding to the rules of append_{in}, we now have to add copies of all rules that result from the append-predicate. However, here we label every root symbol by $\{2, 3\}$. In other words, we have to add the following rules to $\mathcal{R}'_{\mathcal{P}}$:

$$\mathsf{append}^{\{2,3\}}_{in}([\,], M, M) \to \mathsf{append}^{\{2,3\}}_{out}([\,], M, M) \qquad (34)$$

$$\mathsf{append}^{\{2,3\}}_{in}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}^{\{2,3\}}_1(\mathsf{append}_{in}(L, M, N), X, L, M, N) \qquad (35)$$

$$\mathsf{u}^{\{2,3\}}_1(\mathsf{append}_{out}(L, M, N), X, L, M, N) \to \mathsf{append}^{\{2,3\}}_{out}(\bullet(X, L), M, \bullet(X, N)) \qquad (36)$$

Now the result of rewriting a term $\mathsf{append}^{\{2,3\}}_{in}(\ldots)$ will always be a term of the form $\mathsf{append}^{\{2,3\}}_{out}(\ldots)$. Therefore, we have to replace append_{out} by $\mathsf{append}^{\{2,3\}}_{out}$ in the left-hand side of Rule (31) (since (31) is the rule that always "follows" (30)). So the original rule (31)

$$\mathsf{u}^{\{1\}}_2(\mathsf{append}_{out}(L, M, N), N, O) \to \mathsf{u}^{\{1\}}_3(\mathsf{append}_{in}(M, L, O), L, M, N, O) \qquad (31)$$

is replaced by the modified rule

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}^{\{2,3\}}(L, M, N), N, O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}(M, L, O), L, M, N, O). \tag{37}$$

Thus, after the execution of **Step 3.2**, we have $\mathcal{R}'_{\mathcal{P}} = \{(24) - -(29), (33) - (36), (37), (32)\}$. In this way, we have introduced three new labeled symbols $\mathsf{append}_{in}^{\{2,3\}}$, $\mathsf{u}_1^{\{2,3\}}$, and $\mathsf{append}_{out}^{\{2,3\}}$. On the unlabeled symbols, the argument filter π' did not change, but we now additionally have $\pi'(\mathsf{append}_{in}^{\{2,3\}}) = \{2, 3\}$, $\pi'(\mathsf{u}_1^{\{2,3\}}) = \{1, 2, 3, 4, 5\}$, and $\pi'(\mathsf{append}_{out}^{\{2,3\}}) = \{1, 2, 3\}$.

We reach **Step 3.3** and, thus, go back to **Step 3** where we again choose a rule that violates the variable condition. Let us again regard Rule (30), albeit in its modified form as Rule (33). The variable M still violates the variable condition. In **Step 3.1**, the refinement heuristic suggests to filter away the second argument of the symbol $\mathsf{append}_{in}^{\{2,3\}}$. Instead of changing π', we again introduce a new symbol, namely $\mathsf{append}_{in}^{\{3\}}$ with $\pi'(\mathsf{append}_{in}^{\{3\}}) = \{3\}$, and replace the symbol $\mathsf{append}_{in}^{\{2,3\}}$ in the right-hand side of Rule (33) by $\mathsf{append}_{in}^{\{3\}}$. Thus, we obtain a further modification of Rule (33):

$$\mathsf{rotate}_{in}^{\{1\}}(N, O) \to \mathsf{u}_2^{\{1\}}(\mathsf{append}_{in}^{\{3\}}(L, M, N), N, \boxed{O}) \tag{38}$$

Again, we have to ensure that $\mathsf{append}_{in}^{\{3\}}$ has rewrite rules corresponding to the rules of append_{in}. Thus, we add copies of all rules that result from the append-predicate where every root symbol is labeled by $\{3\}$:

$$\mathsf{append}_{in}^{\{3\}}([\,], M, M) \to \mathsf{append}_{out}^{\{3\}}([\,], M, M) \tag{39}$$

$$\mathsf{append}_{in}^{\{3\}}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1^{\{3\}}(\mathsf{append}_{in}(L, M, N), X, L, M, N) \tag{40}$$

$$\mathsf{u}_1^{\{3\}}(\mathsf{append}_{out}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}^{\{3\}}(\bullet(X, L), M, \bullet(X, N)) \tag{41}$$

We also have to replace $\mathsf{append}_{out}^{\{2,3\}}$ by $\mathsf{append}_{out}^{\{3\}}$ in the left-hand side of Rule (37) (since (37) is the rule that always "follows" (33)). So the rule (37) is replaced by the modified rule

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}^{\{3\}}(L, M, N), N, O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}(M, L, O), L, M, N, O) \tag{42}$$

Thus, after the execution of **Step 3.2**, we have $\mathcal{R}'_{\mathcal{P}} = \{(24) - -(29), (38) - (41), (34) - -(36), (42), (32)\}$. Again we have introduced three new labeled symbols $\mathsf{append}_{in}^{\{3\}}$, $\mathsf{u}_1^{\{3\}}$, and $\mathsf{append}_{out}^{\{3\}}$. On the unlabeled symbols, the argument filter π' did not change, but we now additionally have $\pi'(\mathsf{append}_{in}^{\{3\}}) = \{3\}$, $\pi'(\mathsf{u}_1^{\{3\}}) = \{1, 2, 3, 4, 5\}$, and $\pi'(\mathsf{append}_{out}^{\{3\}}) = \{1, 2, 3\}$.

We reach **Step 3.3** and, thus, go back to **Step 3** where we again choose a rule that violates the variable condition. We again regard Rule (30), albeit in its modified form as Rule (38). The variable O still violates the variable condition. In **Step 3.1**, any refinement heuristic must suggest to filter away the third argument of the symbol $\mathsf{u}_2^{\{1\}}$. As $\mathsf{u}_2^{\{1\}}$ does not have the form p_{in}^{I}, we use the second case of **Step 3.2**. Thus, we change π' such that $\pi'(\mathsf{u}_2^{\{1\}}) = \{1,2,3\} \setminus \{3\} = \{1,2\}$. Indeed, now Rule (38) does not violate the variable condition anymore.

We reach **Step 3.3** and, thus, go back to **Step 3** where we again choose a rule that still violates the variable condition. Let us now regard Rule (42):

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}^{\{3\}}(L,M,N),N,O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}(M,L,\boxed{O}),L,M,N,\boxed{O}) \tag{42}$$

Here our refinement heuristic suggests to filter away the third argument of the symbol append_{in} in order to remove the extra variable O. Instead of changing π', we again introduce a new symbol, namely $\mathsf{append}_{in}^{\{1,2\}}$ with $\pi'(\mathsf{append}_{in}^{\{1,2\}}) = \{1,2\}$, and replace the symbol append_{in} in the right-hand side of Rule (42) by $\mathsf{append}_{in}^{\{1,2\}}$. Thus, we obtain a further modification of Rule (42):

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}^{\{3\}}(L,M,N),N,O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}^{\{1,2\}}(M,L,O),L,M,N,\boxed{O}) \tag{43}$$

Again, we have to ensure that $\mathsf{append}_{in}^{\{1,2\}}$ has rewrite rules corresponding to the rules of append_{in}. Thus, we add copies of all rules that result from the append-predicate where every root symbol is labeled by $\{1,2\}$:

$$\mathsf{append}_{in}^{\{1,2\}}([\,],M,M) \to \mathsf{append}_{out}^{\{1,2\}}([\,],M,M) \tag{44}$$

$$\mathsf{append}_{in}^{\{1,2\}}(\bullet(X,L),M,\bullet(X,N)) \to \mathsf{u}_1^{\{1,2\}}(\mathsf{append}_{in}(L,M,N),X,L,M,N) \tag{45}$$

$$\mathsf{u}_1^{\{1,2\}}(\mathsf{append}_{out}(L,M,N),X,L,M,N) \to \mathsf{append}_{out}^{\{1,2\}}(\bullet(X,L),M,\bullet(X,N)) \tag{46}$$

We also have to replace append_{out} by $\mathsf{append}_{out}^{\{1,2\}}$ in the left-hand side of Rule (32) (since (32) is the rule that always "follows" (42)). So the rule (32) is replaced by the modified rule

$$\mathsf{u}_3^{\{1\}}(\mathsf{append}_{out}^{\{1,2\}}(M,L,O),L,M,N,O) \to \mathsf{rotate}_{out}^{\{1\}}(N,O) \tag{47}$$

Thus, after the execution of **Step 3.2**, we now have $\mathcal{R}'_{\mathcal{P}} = \{(24){-}{-}(29), (38){-}(41), (34){-}{-}(36), (43){-}{-}(46), (47)\}$. Again we have introduced three new labeled symbols $\mathsf{append}_{in}^{\{1,2\}}$, $\mathsf{u}_1^{\{1,2\}}$, and $\mathsf{append}_{out}^{\{1,2\}}$. On the unlabeled symbols, the argument filter π' did not change, but we now additionally have $\pi'(\mathsf{append}_{in}^{\{1,2\}}) = \{1,2\}$, $\pi'(\mathsf{u}_1^{\{1,2\}}) = \{1,2,3,4,5\}$, and $\pi'(\mathsf{append}_{out}^{\{1,2\}}) = \{1,2,3\}$.

Note that now we have indeed separated the two copies of the append-rules where $\mathsf{append}_{in}^{\{3\}}$ corresponds to the version of append that has the third argument as input and

$\mathsf{append}_{in}^{\{1,2\}}$ is the version where the first two arguments serve as input. This copying of predicates works although the initial argument filter already filtered away arguments of function symbols like "$\bullet$" (i.e., the initial argument filter was already beyond the expressivity of modings).

Step 3 is repeated until the variable condition is not violated anymore. Note that Algorithm 2 always terminates since there are only finitely many possible labeled variants for every symbol. In our example, we obtain the following set of rules $\mathcal{R}'_{\mathcal{P}}$:

$$\mathsf{append}_{in}([\,], M, M) \to \mathsf{append}_{out}([\,], M, M) \tag{24}$$

$$\mathsf{append}_{in}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1(\mathsf{append}_{in}(L, M, N), X, L, M, N) \tag{25}$$

$$\mathsf{u}_1(\mathsf{append}_{out}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}(\bullet(X, L), M, \bullet(X, N)) \tag{26}$$

$$\mathsf{rotate}_{in}(N, O) \to \mathsf{u}_2(\mathsf{append}_{in}^{\{3\}}(L, M, N), N, O) \tag{48}$$

$$\mathsf{u}_2(\mathsf{append}_{out}^{\{3\}}(L, M, N), N, O) \to \mathsf{u}_3(\mathsf{append}_{in}(M, L, O), L, M, N, O) \tag{49}$$

$$\mathsf{u}_3(\mathsf{append}_{out}(M, L, O), L, M, N, O) \to \mathsf{rotate}_{out}(N, O) \tag{29}$$

$$\mathsf{rotate}_{in}^{\{1\}}(N, O) \to \mathsf{u}_2^{\{1\}}(\mathsf{append}_{in}^{\{3\}}(L, M, N), N, O) \tag{38}$$

$$\mathsf{u}_2^{\{1\}}(\mathsf{append}_{out}^{\{3\}}(L, M, N), N, O) \to \mathsf{u}_3^{\{1\}}(\mathsf{append}_{in}^{\{1,2\}}(M, L, O), L, M, N, O) \tag{43}$$

$$\mathsf{u}_3^{\{1\}}(\mathsf{append}_{out}^{\{1,2\}}(M, L, O), L, M, N, O) \to \mathsf{rotate}_{out}^{\{1\}}(N, O) \tag{47}$$

$$\mathsf{append}_{in}^{\{2,3\}}([\,], M, M) \to \mathsf{append}_{out}^{\{2,3\}}([\,], M, M) \tag{34}$$

$$\mathsf{append}_{in}^{\{2,3\}}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1^{\{2,3\}}(\mathsf{append}_{in}^{\{2,3\}}(L, M, N), X, L, M, N) \tag{50}$$

$$\mathsf{u}_1^{\{2,3\}}(\mathsf{append}_{out}^{\{2,3\}}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}^{\{2,3\}}(\bullet(X, L), M, \bullet(X, N)) \tag{51}$$

$$\mathsf{append}_{in}^{\{3\}}([\,], M, M) \to \mathsf{append}_{out}^{\{3\}}([\,], M, M) \tag{39}$$

$$\mathsf{append}_{in}^{\{3\}}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1^{\{3\}}(\mathsf{append}_{in}^{\{3\}}(L, M, N), X, L, M, N) \tag{52}$$

$$\mathsf{u}_1^{\{3\}}(\mathsf{append}_{out}^{\{3\}}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}^{\{3\}}(\bullet(X, L), M, \bullet(X, N)) \tag{53}$$

$$\mathsf{append}_{in}^{\{1,2\}}([\,], M, M) \to \mathsf{append}_{out}^{\{1,2\}}([\,], M, M) \tag{44}$$

$$\mathsf{append}_{in}^{\{1,2\}}(\bullet(X, L), M, \bullet(X, N)) \to \mathsf{u}_1^{\{1,2\}}(\mathsf{append}_{in}^{\{1,2\}}(L, M, N), X, L, M, N) \tag{54}$$

$$\mathsf{u}_1^{\{1,2\}}(\mathsf{append}_{out}^{\{1,2\}}(L, M, N), X, L, M, N) \to \mathsf{append}_{out}^{\{1,2\}}(\bullet(X, L), M, \bullet(X, N)) \tag{55}$$

The refined argument filter π' is given by

$$\begin{array}{lll lll lll}
\pi'(\mathsf{append}_{in}) & = & \{1,2,3\} & \pi'(\mathsf{rotate}_{in}^{\{1\}}) & = & \{1\} & \pi'(\mathsf{append}_{in}^{\{2,3\}}) & = & \{2,3\} \\
\pi'(\mathsf{append}_{out}) & = & \{1,2,3\} & \pi'(\mathsf{u}_2^{\{1\}}) & = & \{1,2\} & \pi'(\mathsf{append}_{out}^{\{2,3\}}) & = & \{1,2,3\} \\
\pi'(\bullet) & = & \{2\} & \pi'(\mathsf{u}_3^{\{1\}}) & = & \{1,2,3,4\} & \pi'(\mathsf{u}_1^{\{2,3\}}) & = & \{1,4,5\} \\
\pi'(\mathsf{u}_1) & = & \{1,3,4,5\} & \pi'(\mathsf{append}_{in}^{\{1,2\}}) & = & \{1,2\} & \pi'(\mathsf{u}_1^{\{3\}}) & = & \{1,5\} \\
\pi'(\mathsf{rotate}_{in}) & = & \{1,2\} & \pi'(\mathsf{append}_{out}^{\{1,2\}}) & = & \{1,2,3\} & \pi'(\mathsf{u}_1^{\{1,2\}}) & = & \{1,3,4\} \\
\pi'(\mathsf{u}_2) & = & \{1,2,3\} & \pi'(\mathsf{rotate}_{out}^{\{1\}}) & = & \{1,2\} & & & \\
\pi'(\mathsf{append}_{in}^{\{3\}}) & = & \{3\} & & & & & & \\
\pi'(\mathsf{append}_{out}^{\{3\}}) & = & \{1,2,3\} & & & & & & \\
\pi'(\mathsf{u}_3) & = & \{1,2,3,4,5\} & & & & & & \\
\pi'(\mathsf{rotate}_{out}) & = & \{1,2\} & & & & & &
\end{array}$$

Termination for $\mathcal{R}'_{\mathcal{P}}$ w.r.t. the terms specified by π' is now easy to show using our results from Section 3.3.

If one is only interested in termination of queries $\mathsf{rotate}(t_1, t_2)$ for a specific predicate symbol like rotate, then one can remove superfluous (copies of) rules from the TRS before starting the termination proof. For example, if one only wants to prove termination of queries $\mathsf{rotate}(t_1, t_2)$ for finite lists t_1, then it now suffices to prove $\overset{\infty}{\to}$-termination of the above TRS for those "start terms" $\mathsf{rotate}^{\{1\}}_{in}(\ldots)$ that are finite and ground under the filter π' and where the arguments of $\mathsf{rotate}^{\{1\}}_{in}$ do not contain any function symbols except $\bullet$ and $[\,]$. Since the rules for rotate_{in}, append_{in}, and $\mathsf{append}^{\{2,3\}}_{in}$ (i.e., the rules (24) – (26), (29), (34), and (48) – (51)) are not reachable from these "start terms", they can immediately be removed. In other words, for the queries $\mathsf{rotate}(t_1, t_2)$ we indeed need rules for $\mathsf{rotate}^{\{1\}}_{in}$, $\mathsf{append}^{\{1,2\}}_{in}$, and $\mathsf{append}^{\{3\}}_{in}$, but the rules for rotate_{in}, append_{in}, and $\mathsf{append}^{\{2,3\}}_{in}$ are superfluous.

Note however that such superfluous copies of rules are never problematic for the termination analysis. If the rules for $\mathsf{append}^{\{3\}}_{in}$ are $\overset{\infty}{\to}$-terminating for terms that are finite and ground under the filter π', then this also holds for the $\mathsf{append}^{\{2,3\}}_{in}$- and the append_{in}-rules, since here π' filters away less arguments. A corresponding statement holds for the connection between the $\mathsf{rotate}^{\{1\}}_{in}$- and the rotate_{in}-rules.

The following theorem proves the correctness of Algorithm 2. More precisely, it shows that one can use π' and $\mathcal{R}'_{\mathcal{P}}$ instead of π and $\mathcal{R}_{\mathcal{P}}$ in Theorem 3.13. So it is sufficient to prove that all terms in the set $S' = \{p^{\pi(p)}_{in}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi'(p^{\pi(p)}_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$ are $\overset{\infty}{\to}$-terminating w.r.t. the modified TRS $\mathcal{R}'_{\mathcal{P}}$. In Example 3.42, S' would be the set of all terms $\mathsf{rotate}^{\{1\}}_{in}(t_1, t_2)$ that are ground after filtering with π'. Hence, this includes all terms where the first argument is a finite list.

If all terms in S' are $\overset{\infty}{\to}$-terminating w.r.t. $\mathcal{R}'_{\mathcal{P}}$, we can conclude that all queries $Q \in \mathcal{A}^{rat}(\Sigma, \Delta, \mathcal{V})$ with $\pi(Q) \in \mathcal{A}(\Sigma_\pi, \Delta_\pi)$ are terminating for the original logic program. Since π' satisfies the variable condition for the TRS $\mathcal{R}'_{\mathcal{P}}$ (and also for $DP(\mathcal{R}'_{\mathcal{P}})$ if $1 \in \pi'(u_{c,i})$ for all symbols of the form $u_{c,i}$), one can also use π' and $\mathcal{R}'_{\mathcal{P}}$ for the termination criterion of Corollary 3.18. In other words, then it is sufficient to prove that there is no infinite $(DP(\mathcal{R}'_{\mathcal{P}}), \mathcal{R}'_{\mathcal{P}}, \pi')$-chain.

Theorem 3.43 (Soundness of Algorithm 2)**.** *Let $\mathcal{P}$ be a logic program and let π be an argument filter over (Σ, Δ). Let π' and $\mathcal{R}'_{\mathcal{P}}$ result from π and $\mathcal{R}_{\mathcal{P}}$ by Algorithm 2. Let $S = \{p_{in}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})\}$. Furthermore, let $S' = \{p^{\pi(p)}_{in}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi'(p^{\pi(p)}_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$. All terms $s \in S$ are $\overset{\infty}{\to}$-terminating for $\mathcal{R}_{\mathcal{P}}$ if all terms $s' \in S'$ are $\overset{\infty}{\to}$-terminating for $\mathcal{R}'_{\mathcal{P}}$.*

Proof. We first show that every reduction of a term from S with $\mathcal{R}_{\mathcal{P}}$ can be simulated by the reduction of a term from S' with $\mathcal{R}'_{\mathcal{P}}$. More precisely, we show the following proposition where $\mathbb{S}^n = \{t \mid p_{in}(\vec{t}) \overset{\infty}{\rightarrow}{}^n_{\mathcal{R}_\mathcal{P}} t$ for some $p \in \Delta$, $\vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$, and $\pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})\}$ and $\mathbb{S}' = \{t \mid p_{in}^{\pi(p)}(\vec{t}) \overset{\infty}{\rightarrow}{}^*_{\mathcal{R}'_\mathcal{P}} t$ for some $p \in \Delta$, $\vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$, and $\pi'(p_{in}^{\pi(p)}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$

If $s \in \mathbb{S}^n$ and $s' \in \mathbb{S}'$ with $Unlab(s') = s$, then $s \overset{\infty}{\rightarrow}_{\mathcal{R}_\mathcal{P}} t$ implies that there is a t' with $Unlab(t') = t$ and $s' \overset{\infty}{\rightarrow}_{\mathcal{R}'_\mathcal{P}} t'$. (56)

Here, *Unlab* removes all labels introduced by Algorithm 2:

$$Unlab(s) \quad = \quad \begin{cases} f(Unlab(s_1), \ldots, Unlab(s_n)), & \text{if } s = f^I(s_1, \ldots, s_n) \\ s, & \text{otherwise} \end{cases}$$

We prove (56) by induction on n. There are three possible cases for s and for the rule that is applied in the step from s to t.

<u>Case 1: $n = 0$ and thus, $s = p_{in}(\vec{s})$</u>

So $s \in S$ and there is a rule $\ell \rightarrow r \in \mathcal{R}_\mathcal{P}$ with $\ell = p_{in}(\vec{\ell})$ such that $s = \ell\sigma$ and $t = r\sigma$ for some substitution σ with terms from $\mathcal{T}^\infty(\Sigma, \mathcal{V})$.

Let $s' \in \mathbb{S}'$ with $Unlab(s') = s$. Thus, we also have $s' \in S'$ where $s' = p_{in}^{\pi(p)}(\vec{s})$ (since a term with a root symbol p_{in}^I cannot be obtained from S' if one has performed at least one rewrite step with $\mathcal{R}'_\mathcal{P}$). Due to the construction of $\mathcal{R}'_\mathcal{P}$, there exists a rule $\ell^{\pi(p)} \rightarrow r' \in \mathcal{R}'_\mathcal{P}$ where $Unlab(r') = r$. We define t' to be $r'\sigma$. Then we clearly have $s' = \ell^{\pi(p)}\sigma \overset{\infty}{\rightarrow}_{\mathcal{R}'_\mathcal{P}} r'\sigma = t'$ and $Unlab(t') = t$.

<u>Case 2: $n \geq 1$ and $s = u_{c,i}(\overline{s}, \vec{q})$, $\overline{s} \overset{\infty}{\rightarrow}_{\mathcal{R}_\mathcal{P}} \overline{t}$, $t = u_{c,i}(\overline{t}, \vec{q})$</u>

Since $s \in \mathbb{S}^n$, there exists a $p_{in}(\vec{s})$ with $\vec{s} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$ such that $p_{in}(\vec{s}) \overset{\infty}{\rightarrow}{}^*_{\mathcal{R}_\mathcal{P}} \overline{s}$, i.e., $\overline{s} \in \mathbb{S}^m$ for some $m \in \mathbb{N}$. Since the reduction from $p_{in}(\vec{s})$ to $\overline{s}$ is shorter than the overall reduction that led to s, we obtain that $m < n$.

Let $s' \in \mathbb{S}'$ with $Unlab(s') = s$. Hence, we have $s' = u_{c,i}^I(\overline{s'}, \vec{q})$ for some label I and $Unlab(\overline{s'}) = \overline{s}$. Since $s' \in \mathbb{S}'$, there exists a $p_{in}^J(\vec{s})$ with $\vec{s} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$ such that $p_{in}^J(\vec{s}) \overset{\infty}{\rightarrow}{}^*_{\mathcal{R}'_\mathcal{P}} \overline{s'}$. Hence, $\overline{s'} \in \mathbb{S}'$ as well. Now the induction hypothesis implies that there exists a $\overline{t'}$ such that $\overline{s'} \overset{\infty}{\rightarrow}_{\mathcal{R}'_\mathcal{P}} \overline{t'}$ and $Unlab(\overline{t'}) = \overline{t}$. We define $t' = u_{c,i}^I(\overline{t'}, \vec{q})$. Then we clearly have $s' \overset{\infty}{\rightarrow}_{\mathcal{R}'_\mathcal{P}} t'$ and $Unlab(t') = t$.

<u>Case 3: $n \geq 1$ and $s = u_{c,i}(p_{out}(\vec{s}), \vec{q})$</u>

Here, there exists a rule $\ell \rightarrow r \in \mathcal{R}_\mathcal{P}$ with $\ell = u_{c,i}(p_{out}(\vec{\ell}), \vec{x})$ such that $s = \ell\sigma$ and $t = r\sigma$.

Let $s' \in \mathbb{S}'$ with $Unlab(s') = s$. Hence, we have $s' = u_{c,i}^I(p_{out}^J(\vec{s}), \vec{q})$ for some labels I and J. Since $s' \in \mathbb{S}'$, s' resulted from rewriting the term $u_{c,i}^I(p_{in}^J(\vec{s}), \vec{q})$ which must be an instantiated right-hand side of a rule from $\mathcal{R}'_\mathcal{P}$. Due to the construction of $\mathcal{R}'_\mathcal{P}$, then there also exists a rule $\ell' \rightarrow r' \in \mathcal{R}'_\mathcal{P}$ where $\ell' = u_{c,i}^I(p_{out}^J(\vec{\ell}), \vec{x})$ and $Unlab(r') = r$. We define $t' = r'\sigma$. Then we have $s' = \ell'\sigma \overset{\infty}{\rightarrow}_{\mathcal{R}'_\mathcal{P}} r'\sigma = t'$ and clearly $Unlab(t') = t$.

We now proceed to prove the theorem by contradiction. Assume there is a term $s_0 \in S$ that is non-$\overset{\infty}{\to}$-terminating w.r.t. $\mathcal{R}_\mathcal{P}$, i.e., there is an infinite sequence of terms $s_0, s_1, s_2, \ldots$ with $s_i \overset{\infty}{\to}_{\mathcal{R}_\mathcal{P}} s_{i+1}$. We must have $s_0 = p_{in}(\vec{t})$ with $\vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$ and $\pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})$. Let $s'_0 = p_{in}^{\pi(p)}(\vec{t})$. Then $s'_0 \in S'$, since $\pi'(p_{in}^{\pi(p)}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})$. The reason is that $\pi'(p_{in}^{\pi(p)}) = \pi(p) = \pi(p_{in})$ and for all $f \in \Sigma$ we have $\pi'(f) \subseteq \pi(f)$.

So by (56), $s'_0 \in \mathbb{S}'$ and $Unlab(s'_0) = s_0$ imply that there is an s'_1 with $Unlab(s'_1) = s_1$ and $s'_0 \overset{\infty}{\to}_{\mathcal{R}'_\mathcal{P}} s'_1$. Clearly, this also implies $s'_1 \in \mathbb{S}'$. By applying (56) repeatedly, we therefore obtain an infinite sequence of labeled terms $s'_0, s'_1, s'_2, \ldots$ with $s'_i \overset{\infty}{\to}_{\mathcal{R}'_\mathcal{P}} s'_{i+1}$. □

3.5. Formal Comparison of the Transformational Approaches

In this section we formally compare the power of the classical transformation with the power of our new approach. In the classical approach, the class of queries is characterized by a moding function whereas in our approach, it is characterized by an argument filter. Therefore, the following definition establishes a relationship between modings and argument filters.

Definition 3.44 (Argument Filter Induced by Moding). *Let (Σ, Δ) be a signature and let m be a moding over the set of predicate symbols Δ. Then for every predicate symbol $p \in \Delta$ we define the* induced argument filter *π_m over $\Sigma_\mathcal{P}$ as $\pi_m(p_{in}) = \pi_m(P_{in}) = \{i \mid m(p,i) = \textbf{in}\}$ and $\pi_m(p_{out}) = \{i \mid m(p,i) = \textbf{out}\}$. All other function symbols f from $\Sigma_\mathcal{P}$ are not filtered, i.e., $\pi_m(f/n) = \{1, \ldots, n\}$.*

Example 3.45. *Regard again the well-moded logic program from Example 3.1.*

$$\begin{array}{lll}\mathsf{p}(X,X). & & \\ \mathsf{p}(\mathsf{f}(X),\mathsf{g}(Y)) & \leftarrow & \mathsf{p}(\mathsf{f}(X),\mathsf{f}(Z)), \mathsf{p}(Z,\mathsf{g}(Y)).\end{array}$$

We used the moding m with $m(\mathsf{p},1) = \textbf{in}$ and $m(\mathsf{p},2) = \textbf{out}$. Thus, for the induced argument filter π_m we have $\pi_m(\mathsf{p}_{in}) = \pi_m(\mathsf{P}_{in}) = \{1\}$ and $\pi_m(\mathsf{p}_{out}) = \{2\}$.

As the classical approach is only applicable to well-moded logic programs, we restrict our comparison to this class. For non-well-moded programs, our new approach is clearly more powerful, since it can often prove termination (cf. Section 3.6), whereas the classical transformation is never applicable.

Our goal is to show the connection between the TRSs resulting from the two transformations. If one refines π_m to a filter π'_m by Algorithm 1 using *any* arbitrary refinement heuristic, then the TRS of the classical transformation corresponds to the TRS of our new transformation after filtering it with π'_m.

Example 3.46. *We continue with Example 3.45. The TRS $\mathcal{R}_\mathcal{P}$ resulting from our new transformation was given in Example 3.8:*

$$\begin{aligned}
\mathsf{p}_{in}(X,X) &\to \mathsf{p}_{out}(X,X) && (1)\\
\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) && (2)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z) && (3)\\
\mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(Y)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y)) && (4)
\end{aligned}$$

If we apply the induced argument filter π_m, then we obtain the TRS $\pi_m(\mathcal{R}_\mathcal{P})$:

$$\begin{aligned}
\mathsf{p}_{in}(X) &\to \mathsf{p}_{out}(X)\\
\mathsf{p}_{in}(\mathsf{f}(X)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)),X,Y)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z),X,Y,Z)\\
\mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(Y)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{g}(Y))
\end{aligned}$$

The second rule has the "extra" variable Y on the right-hand side, i.e., it does not satisfy the variable condition. Thus, we have to refine the filter π_m to a filter π'_m with $\pi'_m(\mathsf{u}_1) = \pi'_m(\mathsf{U}_1) = \{1,2\}$ and $\pi'_m(\mathsf{u}_2) = \pi'_m(\mathsf{U}_2) = \{1,2,4\}$. The resulting TRS $\pi'_m(\mathcal{R}_\mathcal{P})$ is identical to the TRS $\mathcal{R}^{old}_\mathcal{P}$ resulting from the classical transformation, cf. Example 3.2:

$$\begin{aligned}
\mathsf{p}_{in}(X) &\to \mathsf{p}_{out}(X)\\
\mathsf{p}_{in}(\mathsf{f}(X)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)),X)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(Z)),X) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z),X,Z)\\
\mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(Y)),X,Z) &\to \mathsf{p}_{out}(\mathsf{g}(Y))
\end{aligned}$$

The following theorem shows that our approach (with Corollary 3.18) succeeds whenever the classical transformation of Section 3 yields a terminating TRS.

Theorem 3.47 (Subsumption of the Classical Transformation)**.** *Let $\mathcal{P}$ be a well-moded logic program over a signature (Σ,Δ) w.r.t. the moding m. Let $\mathcal{R}^{old}_\mathcal{P}$ be the result of applying the classical transformation of Section 3 and let $\mathcal{R}_\mathcal{P}$ be the result of our new transformation from Definition 3.7. Then there is a refinement of π'_m of π_m such that (a) $\pi'_m(\mathcal{R}_\mathcal{P})$ and $\pi'_m(DP(\mathcal{R}_\mathcal{P}))$ satisfy the variable condition and (b) if $\mathcal{R}^{old}_\mathcal{P}$ is terminating (with ordinary rewriting), then there is no infinite $(DP(\mathcal{R}_\mathcal{P}),\mathcal{R}_\mathcal{P},\pi'_m)$-chain. Thus, in particular, termination of $\mathcal{R}^{old}_\mathcal{P}$ implies that $\mathcal{R}_\mathcal{P}$ is $\overset{\infty}{\to}$-terminating (with infinitary constructor rewriting) for all terms $p_{in}(\vec{t})$ with $p \in \Delta$, $\vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma,\mathcal{V})$, and $\pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})$.*

Proof. Let π'_m result from Algorithm 1 using any refinement heuristic ρ which does not filter away the first argument of any $u_{c,i}$.

We now analyze the structure of the TRS $\pi'_m(\mathcal{R}_\mathcal{P})$. For any predicate symbol $p \in \Delta$, let "$p(\vec{s},\vec{t})$" denote that $\vec{s}$ and $\vec{t}$ are the sequences of terms on p's in- and output positions w.r.t. the moding m.

When Algorithm 1 is applied to compute the refinement π'_m of π_m, one looks for a rule $\ell \to r$ from $\pi_m(\mathcal{R}_\mathcal{P})$ such that $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$. Such a rule cannot result from the facts of the logic program. The reason is that for each fact $p(\vec{s},\vec{t})$, $\pi_m(\mathcal{R}_\mathcal{P})$ contains the rule

$$p_{in}(\vec{s}) \to p_{out}(\vec{t})$$

and by well-modedness, we have $\mathcal{V}(\vec{t}) \subseteq \mathcal{V}(\vec{s})$.

For each rule c of the form $p(\vec{s},\vec{t}) \;\leftarrow\; p_1(\vec{s}_1,\vec{t}_1),\ldots,p_k(\vec{s}_k,\vec{t}_k)$ in $\mathcal{P}$, the TRS $\pi_m(\mathcal{R}_\mathcal{P})$ contains:

$$\begin{array}{l}
p_{in}(\vec{s}) \;\to\; u_{c,1}(p_{1_{in}}(\vec{s}_1), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}))\\
u_{c,1}(p_{1_{out}}(\vec{t}_1), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t})) \;\to\; u_{c,2}(p_{2_{in}}(\vec{s}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}) \cup \mathcal{V}(\vec{s}_1) \cup \mathcal{V}(\vec{t}_1))\\
\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\
u_{c,k}(p_{k_{out}}(\vec{t}_k), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}) \cup \mathcal{V}(\vec{s}_1) \cup \mathcal{V}(\vec{t}_1) \cup \ldots \cup \mathcal{V}(\vec{s}_{k-1}) \cup \mathcal{V}(\vec{t}_{k-1})) \;\to\; p_{out}(\vec{t})
\end{array}$$

For the first rule, by well-modedness we have $\mathcal{V}(\vec{s}_1) \subseteq \mathcal{V}(\vec{s})$ and thus, the only "extra" variables on the right-hand side of the first rule must be from $\mathcal{V}(\vec{t})$. There is only one possibility to refine the argument filter in order to remove them: one has to filter away the respective argument positions of $u_{c,1}$. Hence, the filtered right-hand side of the first rule is $u_{c,1}(p_{1_{in}}(\vec{s}_1), \mathcal{V}(\vec{s}))$ and the filtered left-hand side of the second rule is $u_{c,1}(p_{1_{out}}(\vec{t}_1), \mathcal{V}(\vec{s}))$.

Similarly, for the second rule, well-modedness implies $\mathcal{V}(\vec{s}_2) \cup \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s}_1) \cup \mathcal{V}(\vec{t}_1) \subseteq \mathcal{V}(\vec{t}_1) \cup \mathcal{V}(\vec{s})$. So the only "extra" variables on the right-hand side of the second rule are again from $\mathcal{V}(\vec{t})$. As before, to remove them one has to filter away the respective argument positions of $u_{c,2}$. Moreover, since $\mathcal{V}(\vec{s}_1) \subseteq \mathcal{V}(\vec{s})$ we obtain the filtered right-hand side $u_{c,2}(p_{2_{in}}(\vec{s}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1))$ for the second rule and the filtered left-hand side $u_{c,2}(p_{2_{out}}(\vec{t}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1))$ side in the third rule.

An analogous argument holds for the other rules. The last rule has no extra variables, since $\mathcal{V}(\vec{t}) \subseteq \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1) \cup \ldots \cup \mathcal{V}(\vec{t}_k)$ by well-modedness.

So for any rule c of the logic program $\mathcal{P}$, $\pi'_m(\mathcal{R}_\mathcal{P})$ has the following rules:

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

Hence, $\pi'_m(\mathcal{R}_\mathcal{P}) = \mathcal{R}^{old}_\mathcal{P}$. Since the refined argument filter π'_m does not filter away the first argument of any $u_{c,i}$, by defining $\pi'_m(U_{c,i}) := \pi'_m(u_{c,i})$, then the variable condition is satisfied for both $\pi'_m(\mathcal{R}_\mathcal{P})$ and $\pi'_m(DP(\mathcal{R}_\mathcal{P}))$ and, thus, (a) is fulfilled.

Now to prove (b), we assume that $\mathcal{R}_{\mathcal{P}}^{old}$ is terminating. We have to show that then there is no infinite $(DP(\mathcal{R}_{\mathcal{P}}), \mathcal{R}_{\mathcal{P}}, \pi'_m)$-chain. By the soundness of the argument filter processor (Theorem 3.26), it suffices to show that there is no infinite $(\pi'_m(DP(\mathcal{R}_{\mathcal{P}})), \pi'_m(\mathcal{R}_{\mathcal{P}}), id)$-chain.

Note that $\pi'_m(DP(\mathcal{R}_{\mathcal{P}})) = DP(\pi'_m(\mathcal{R}_{\mathcal{P}}))$. The reason is that all $u_{c,i}$ only occur on the root level in $\mathcal{R}_{\mathcal{P}}$. Moreover, all p_{in}-symbols only occur in the first argument of a $u_{c,i}$ and $1 \in \pi'_m(u_{c,i})$. In other words, occurrences of defined function symbols are not removed by the filter π'_m. So we have

$$\begin{array}{ll}
 & u \to v \in \pi'_m(DP(\mathcal{R}_{\mathcal{P}})) \\
\text{if, and only if,} & \text{there is a rule } \ell \to r \in \mathcal{R}_{\mathcal{P}} \text{ with } u = \pi'_m(\ell^\sharp), v = \pi'_m(t^\sharp) \\
 & \text{for a subterm } t \text{ of } r \text{ with defined root} \\
\text{if, and only if,} & \text{there is a rule } \ell \to r \in \mathcal{R}_{\mathcal{P}} \text{ with } u = (\pi'_m(\ell))^\sharp, v = \\
 & (\pi'_m(t))^\sharp \\
 & \text{for a subterm } \pi'_m(t) \text{ of } \pi'_m(r) \text{ with defined root} \\
\text{if, and only if,} & \text{there is a rule } \ell \to r \in \pi'_m(\mathcal{R}_{\mathcal{P}}) \text{ with } u = \ell^\sharp, v = t^\sharp \\
 & \text{for a subterm } t \text{ of } r \text{ with defined root} \\
\text{if, and only if,} & u \to v \in DP(\pi'_m(\mathcal{R}_{\mathcal{P}}))
\end{array}$$

Hence, $\pi'_m(\mathcal{R}_{\mathcal{P}}) = \mathcal{R}_{\mathcal{P}}^{old}$ and $\pi'_m(DP(\mathcal{R}_{\mathcal{P}})) = DP(\pi'_m(\mathcal{R}_{\mathcal{P}})) = DP(\mathcal{R}_{\mathcal{P}}^{old})$. Thus, it suffices to show absence of infinite $(DP(\mathcal{R}_{\mathcal{P}}^{old}), \mathcal{R}_{\mathcal{P}}^{old}, id)$-chains. But this follows from termination of $\mathcal{R}_{\mathcal{P}}^{old}$, cf. [AG00, Thm. 6], since $(\mathcal{P}, \mathcal{R}, id)$-chains correspond to chains for ordinary (non-infinitary) rewriting.

Hence by Theorem 3.17, termination of $\mathcal{R}_{\mathcal{P}}^{old}$ also implies that all terms $p_{in}(\vec{t})$ with $p \in \Delta$, $\vec{t} \in \vec{\mathcal{T}}^\infty(\Sigma, \mathcal{V})$, and $\pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_\pi})$ are $\overset{\infty}{\to}$-terminating w.r.t. $\mathcal{R}_{\mathcal{P}}$ (using infinitary constructor rewriting). □

The reverse direction of the above theorem does not hold, though. As a counterexample, regard again the logic program from Example 3.1, cf. Example 3.46. As shown in Example 3.2, the TRS resulting from the classical transformation is not terminating. Still, for the filter π'_m from Example 3.46, there is no infinite $(DP(\mathcal{R}_{\mathcal{P}}), \mathcal{R}_{\mathcal{P}}, \pi'_m)$-chain and thus, our method of Corollary 3.18 succeeds with the termination proof. In other words, our new approach is *strictly* more powerful than the classical transformation, even on well-moded programs.

Thus, a termination analyzer based on our new transformation should be strictly more successful in practice, too. That this is in fact the case will be demonstrated in the next section.

3.6. Experiments and Discussion

We integrated our approach (including all refinements presented) in the termination tool AProVE [GST06] which implements the DP framework. To evaluate our results, we tested AProVE against four other representative termination tools for logic programming: TALP [OCM00] is the only other available tool based on transformational methods (it uses the classical transformation of Section 3), whereas Polytool [ND07], TerminWeb [CT99], and cTI [MB05] are based on direct approaches. We now describe the results of our experimental evaluation and then discuss the limitations of our approach.

Experimental Evaluation

We ran the tools on a set of 296 examples in fully automatic mode.[11] This set includes all logic programming examples from the *Termination Problem Data Base* [TPD07] which is used in the annual international *Termination Competition* [MZ07]. It contains collections provided by the developers of several different tools including all examples from the experimental evaluation of [BCG$^+$07]. However, to eliminate the influence of the translation from Prolog to 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. All tools were run locally on an AMD Athlon 64 at 2.2 GHz under GNU/Linux 2.6. For each example we used a time limit of 60 seconds. This is similar to the way that tools are evaluated in the annual competitions for termination tools. For every tool we give the number of LPs which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total running time ("Total") in seconds.

	AProVE	Polytool	TerminWeb	cTI	TALP
Successes	232	204	177	167	163
Failures	57	82	118	129	112
Timeouts	7	10	1	0	21
Total	1471.4	622.7	95.3	10.4	413.5

As shown in the table above, AProVE succeeds on more examples than any other tool. The comparison of AProVE and TALP shows that our approach improves significantly upon the previous transformational method that TALP is based on, cf. Goals (A) and (B). In particular, TALP fails for all non-well-moded programs.

[11]We combined *termsize* and *list-length* norm for TerminWeb and allowed 5 iterations before widening for cTI. Apart from that, we used the default settings of the tools. For both AProVE and Polytool we used the (fully automated) original executables from the *Termination Competition* 2007 [MZ07]. To refine argument filters, this version of AProVE uses the refinement heuristic $\rho_{tb'}$ from Definition 3.39. For a list of the main termination techniques used in AProVE, we refer to [GTS05a, GTSF06]. Of these techniques, only the ones in Section 3.3 were adapted to infinitary constructor rewriting.

While we have shown our technique to be strictly more powerful than the previous transformational method, due to the higher arity of the function symbols produced by our transformation, proving termination could take more time in some cases. However, in the above experiments this did not affect the practical power of our implementation. In fact, AProVE is able to prove termination well within the time limit for all examples where TALP succeeds. Further analysis shows that while AProVE never takes more than 15 seconds longer than TALP, there are indeed 6 examples where AProVE is more than 15 seconds *faster* than TALP.

The comparison with Polytool, TerminWeb, and cTI demonstrates that our new transformational approach is not only comparable in power, but usually more powerful than direct approaches. In fact, there is only a single example where one of the other tools (namely Polytool) succeeds and AProVE fails. This is the rather contrived example from (2) in Section 3.6 which we developed to demonstrate the limitations of our method. Polytool is only able to handle this example via a pre-processing step based on partial evaluation [NBDL06, SD03b, TC04]. In this example, this pre-processing step results in a trivially terminating logic program. Thus, if one combined this pre-processing with any of the other tools, then they would also be able to prove termination of this particular example.[12] Integrating some form of partial evaluation into AProVE might be an interesting possibility for further improvement. For all other examples, AProVE can show termination whenever at least one of the other tools succeeds. Moreover, there are several examples where AProVE succeeds whereas no other tool shows the termination. These include examples where the termination proof requires more complex orders. For instance, termination of the example `SGST06/hbal_tree.pl` can be proved using recursive path orders with status and termination of `talp/apt/mergesort_ap.pl` is shown using matrix orders.[13]

Note that 52 examples in this collection are known to be non-terminating, i.e., there are at most 244 terminating examples. In other words, there are only at most 12 terminating examples where AProVE did not manage to prove termination. With this performance, AProVE won the *Termination Competition* with Polytool being the second most powerful tool. The best tool for non-termination analysis of logic programs was NTI [PM06].

However, from the experiments above one should not draw the conclusion that the transformational approach is always better than the direct approach to termination analysis of logic programs. There are several extensions (e.g., termination inference [CT99, MB05], non-termination analysis [PM06], handling numerical data structures [SD04, SD05b]) that can currently only be handled by direct techniques and tools.

[12]Similarly, with such a pre-processing the existing "direct" tools would also be able to prove termination of the program in Example 3.1.

[13]For recursive path orders with status and matrix orders see [Les83] resp. [EWZ06].

Regarding the use of term rewriting techniques for termination analysis of logic programs, it is interesting to note that the currently most powerful tool for direct termination analysis of logic programs (Polytool) implements the framework of [ND05, ND07] for applying techniques from term rewriting (most notably polynomial interpretations) to logic programs directly. This framework forms the basis for further extensions to other TRS-termination techniques. For example, it can be extended further by adapting also the dependency pair framework to the logic programming setting as demonstrated in Chapter 4 of this thesis.

So transformational and direct approaches both have their advantages and the most powerful solution would be to combine direct tools like Polytool with a transformational prover like AProVE which is based on the contributions of this section as demonstrated in Chapter 4. But it is clear that it is indeed beneficial to use termination techniques from TRSs for logic programs, both for direct and for transformational approaches.

In addition to the experiments described above (which compare different termination provers), we also performed experiments with several versions of AProVE in order to evaluate the different heuristics and algorithms for the computation of argument filters from Section 3.4. The following table shows that indeed our improved type-based refinement heuristic (tb′) significantly outperforms the simple improved outermost (om′) and innermost (im) heuristics. In fact, all examples that could be proved terminating by any of the simple heuristics can also be proved terminating by the type-based heuristic.

	AProVE tb′	AProVE om′	AProVE im
Successes	232	218	195
Failures	57	76	98
Timeouts	7	2	3

So far, for all experiments we used Algorithm 2 in order to compute a refined argument filter from the initial one. To evaluate the advantage of this improved algorithm over Algorithm 1, we performed experiments with the two algorithms (again using the type-based refinement heuristic tb′). The following table shows that Algorithm 2 is indeed significantly more powerful than Algorithm 1.

	AProVE Algorithm 2	AProVE Algorithm 1
Successes	232	212
Failures	57	74
Timeouts	7	10

Preliminary versions of parts of this chapter appeared in [SGST07]. However, the table below clearly shows that the results of Section 3.4 (which are new compared to [SGST07]) improve the power of termination analysis substantially. To this end, we compare our new implementation that uses the improved type-based refinement heuristic (tb′) and the improved refinement algorithm (Algorithm 2) from Section 3.4 with the version of AProVE from the *Termination Competition* 2006 that only contains the results of [SGST07]. To find argument filters, it uses a simple ad-hoc heuristic which turns out to be clearly disadvantageous to the new sophisticated techniques from Section 3.4.

	AProVE tb′	AProVE [SGST07]
Successes	232	208
Failures	57	69
Timeouts	7	19

To run AProVE, for details on our experiments, and to access our collection of examples, we refer to `http://aprove.informatik.rwth-aachen.de/eval/TOCL/`.

Limitations

Our experiments also contain examples which demonstrate the limitations of our approach. Of course, our implementation in AProVE usually fails if there are features outside of definite logic programming (e.g., built-in predicates, negation as failure, meta-programming, etc.). A novel approach for the handling of meta-logical features such as cuts and meta-programming is presented in Chapter 5.

In the following, we discuss the limitations of the approach when applying it for definite logic programming. In principle, there could be three points of failure:

(i) The transformation of Theorem 3.13 could fail, i.e., there could be a logic program which is terminating for the set of queries, but not all corresponding terms are $\overset{\infty}{\to}$-terminating in the transformed TRS. We do not know of any such example. It is currently open whether this step is in fact complete.

(ii) The approach via dependency pairs (Theorem 3.17) can fail to prove $\overset{\infty}{\to}$-termination of the transformed TRS, although the TRS is $\overset{\infty}{\to}$-terminating. In particular, this can happen because of the variable condition required for Theorem 3.17. This is demonstrated by the following logic program $\mathcal{P}$:

$$\begin{array}{lcl}
\mathsf{p}(X) & \leftarrow & \mathsf{q}(\mathsf{f}(Y)), \mathsf{p}(Y). \\
\mathsf{p}(\mathsf{g}(X)) & \leftarrow & \mathsf{p}(X). \\
\mathsf{q}(\mathsf{g}(Y)). & &
\end{array}$$

The resulting TRS $\mathcal{R}_\mathcal{P}$ is

$$\begin{aligned}
\mathsf{p}_{in}(X) &\to \mathsf{u}_1(\mathsf{q}_{in}(\mathsf{f}(Y)), X)\\
\mathsf{u}_1(\mathsf{q}_{out}(\mathsf{f}(Y)), X) &\to \mathsf{u}_2(\mathsf{p}_{in}(Y), X, Y)\\
\mathsf{u}_2(\mathsf{p}_{out}(Y), X, Y) &\to \mathsf{p}_{out}(X)\\
\mathsf{p}_{in}(\mathsf{g}(X)) &\to \mathsf{u}_3(\mathsf{p}_{in}(X), X)\\
\mathsf{u}_3(\mathsf{p}_{out}(X), X) &\to \mathsf{p}_{out}(\mathsf{g}(X))\\
\mathsf{q}_{in}(\mathsf{g}(Y)) &\to \mathsf{q}_{out}(\mathsf{g}(Y))
\end{aligned}$$

and there are the following dependency pairs.

$$\begin{aligned}
\mathsf{P}_{in}(X) &\to \mathsf{Q}_{in}(\mathsf{f}(Y)) &\quad (57)\\
\mathsf{P}_{in}(X) &\to \mathsf{U}_1(\mathsf{q}_{in}(\mathsf{f}(Y)), X) &\quad (58)\\
\mathsf{U}_1(\mathsf{q}_{out}(\mathsf{f}(Y)), X) &\to \mathsf{P}_{in}(Y) &\quad (59)\\
\mathsf{U}_1(\mathsf{q}_{out}(\mathsf{f}(Y)), X) &\to \mathsf{U}_2(\mathsf{p}_{in}(Y), X, Y) &\quad (60)\\
\mathsf{P}_{in}(\mathsf{g}(X)) &\to \mathsf{P}_{in}(X) &\quad (61)\\
\mathsf{P}_{in}(\mathsf{g}(X)) &\to \mathsf{U}_3(\mathsf{p}_{in}(X), X) &\quad (62)
\end{aligned}$$

We want to prove termination of all queries $\mathsf{p}(t)$ where t is finite and ground (i.e., $m(\mathsf{p}, 1) = \boldsymbol{in}$). Looking at the logic program $\mathcal{P}$, it is obvious that they are all terminating. However, there is no argument filter π such that $\pi(\mathcal{R}_\mathcal{P})$ and $\pi(DP(\mathcal{R}_\mathcal{P}))$ satisfy the variable condition and such that there is no infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi)$-chain.

To see this, note that if $\pi(\mathsf{P}_{in}) = \varnothing$ or $\pi(\mathsf{g}) = \varnothing$ then we can build an infinite chain with the last dependency pair where we instantiate X by the infinite term $\mathsf{g}(\mathsf{g}(\ldots))$. So, let $\pi(\mathsf{P}_{in}) = \pi(\mathsf{g}) = \{1\}$. Due to the variable condition of the dependency pair (59) we know $\pi(\mathsf{f}) = \pi(\mathsf{q}_{out}) = \{1\}$ and $1 \in \pi(\mathsf{U}_1)$. Hence, to satisfy the variable condition in dependency pair (58) we must set $\pi(\mathsf{q}_{in}) = \varnothing$. But then the last rule of $\pi(\mathcal{R}_\mathcal{P})$ does not satisfy the variable condition.

(iii) Finally it can happen that the resulting DP problem of Theorem 3.17 is terminating, but that our implementation fails to prove it. The reason can be that one should apply other DP processors or DP processors with other parameters. After all, termination of DP problems is undecidable. This is shown by the following example where we are interested in all queries $\mathsf{f}(t_1, t_2)$ where t_1 and t_2 are ground terms:

$$\begin{array}{lcl}
\mathsf{f}(X, Y) & \leftarrow & \mathsf{g}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(X))))), Y).\\
\mathsf{f}(\mathsf{s}(X), Y) & \leftarrow & \mathsf{f}(X, Y).\\
\mathsf{g}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(X)))))), Y) & \leftarrow & \mathsf{f}(X, Y).
\end{array}$$

$\overset{\infty}{\rightarrow}$-termination can (for example) be proved if one uses a polynomial order with coefficients from $\{0, 1, 2, 3, 4, 5\}$. But the current automation does not use such polynomials and thus, it fails when trying to prove termination of this example.

While the DP method can also be used for non-termination proofs if one considers ordinary rewriting, this is less obvious for infinitary constructor rewriting. The reason is that the main termination criterion is "complete" for ordinary rewriting, but incomplete for infinitary constructor rewriting (cf. the counterexample (ii) to the completeness of Theorem 3.17 above). Therefore, in order to also prove *non-termination* of logic programs, a combination of our method with a loop-checker for logic programs would be fruitful. As mentioned before, a very powerful non-termination tool for logic programs is NTI [PM06]. Our collection of 296 examples contains 233 terminating examples (232 of these can be successfully shown by AProVE), 52 non-terminating examples, and 11 examples whose termination behavior is unknown. NTI can prove non-termination of 42 of the 52 non-terminating examples. Hence, a combination of AProVE and NTI would successfully analyze the termination behavior of 274 of the 296 examples.

3.7. Summary

In this chapter, we developed a new transformation from logic programs $\mathcal{P}$ to TRSs $\mathcal{R}_\mathcal{P}$. To prove the termination of a class of queries for $\mathcal{P}$, it is now sufficient to analyze the termination behavior of $\mathcal{R}_\mathcal{P}$ on a corresponding class of terms w.r.t. infinitary constructor rewriting. This class of terms is characterized by a so-called argument filter and we showed how to generate such argument filters from the given class of queries for $\mathcal{P}$. Our approach is even sound for logic programming without occur check. To prove termination of infinitary rewriting automatically, we showed how to adapt the DP framework of [AG00, GTS05a, GTSF06] from ordinary term rewriting to infinitary constructor rewriting. Then the DP framework can be used for termination proofs of $\mathcal{R}_\mathcal{P}$ and thus, for automated termination analysis of $\mathcal{P}$. Since *any* termination technique for TRSs can be formulated as a DP processor [GTS05a], now any such technique can also be used for logic programs.

In addition to the results presented in [SGST07], we showed that our new approach subsumes the classical transformational approach to termination analysis of logic programs. We also provided new heuristics and algorithms for refining the initial argument filter that improve the power of our method (and hence, also of its implementation) substantially.

Moreover, we implemented all contributions in our termination prover AProVE and performed extensive experiments which demonstrate that our results are indeed applicable in practice. More precisely, due to our contributions, AProVE has become the currently most powerful automated termination prover for logic programs.

4. Direct Approach

As we have seen in the previous chapter, termination analysis for logic programming traditionally aims at proving that a given logic program terminates w.r.t. a specific set of queries. So far we have only discussed transformational approaches, i.e., non-termination preserving transformations from logic programming to term rewriting.

In contrast, direct termination proofs are usually performed by finding ranking functions that map the sequence of program states to a sequence of elements of a well-founded domain such that the sequence is decreasing w.r.t. a well-founded order of the domain. Practically, it is sufficient to consider only the states that are involved in loops of the program.

Direct techniques in termination analysis of LPs can be divided into two groups: the global approach versus the local approach [BCG$^+$07, CG03, CT99, DSVB92, DDV99, DLSS01, ND05]. In the global approach, one wants to find only one ranking function for all loops [DSVB92, DDV99, ND05]. In contrast, techniques in the local approach apply different ranking functions for different loops [BCG$^+$07, CG03, CT99, DLSS01]. Some automated techniques in the global approach are based on a constraint-based framework to search for a suitable ranking function. This is done by first generating a set of symbolic constraints from all termination conditions. Then, a constraint solver is used to solve the set of constraints, yielding a suitable ranking function for the proof. In the local approach, most techniques use a given small set of norms, and try to prove that (a combination of) these norms can be applied for the termination proof of the program. Unfortunately, by restricting the norms that can be used, implementations are considerably less powerful in practice than the theory allows. It is unclear at this stage whether a search for arbitrary norms in the local approach could also be automated using a constraint-based technique like [DDV99].

While the constraint-based global approach is very suitable for automation, it has some drawbacks. Since it generates the constraints for all termination conditions and solves them at once, it may be very time-consuming, especially for non-terminating programs. This is because the time for solving a set of constraints often increases exponentially with its size. Moreover, if a complex well-founded order is needed for the termination proof (e.g., a lexicographical combination of orders), it is often difficult to find such an order using the constraint-based global approach.

Example 4.1 (ack). *Consider a logic program $\mathcal{P}$ computing the Ackermann function. We use a variant with a predecessor predicate* p/2 *in order to illustrate how our technique handles local variables. We want to prove termination of this program w.r.t. the set of queries* $\mathcal{S} = \{\mathsf{ack}(t_1, t_2, t_3) \mid t_1$ *and* t_2 *are ground terms* $\}$.

$$\mathsf{p}(\mathsf{s}(X), X).$$
$$\mathsf{ack}(\mathsf{0}, X, \mathsf{s}(X)).$$
$$\mathsf{ack}(X, \mathsf{0}, Z) \leftarrow \mathsf{p}(X, Y), \mathsf{ack}(Y, \mathsf{s}(\mathsf{0}), Z).$$
$$\mathsf{ack}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{ack}(\mathsf{s}(X), Y, Z'), \mathsf{ack}(X, Z', Z).$$

Proving termination of this example based on the local approach involves two ranking functions: the first one measures the size of the first argument and the other one measures that of the second argument of the predicate ack/3. *However, with the constraint-based global approach, it is impossible to find a single ranking function for the termination proof (if one is restricted to ranking functions based on polynomial interpretations). As a matter of fact, both the tool* cTI *[MB05] and the tool* Polytool *[ND05, ND07] fail to prove termination of this example.*

In spite of the success of the new transformational method presented in Chapter 3, there remain LPs whose termination can currently only be proved by tools working with direct approaches. An example is the "*der*"-program from [DS02, ND05]. On the other hand, there are also many LPs where currently only transformational tools succeed (e.g., the example "*LP/SGST06-shuffle*" from the *Termination Problem Data Base* (TPDB) [TPD07]). In [NGSD08], we developed a new approach which solves this problem by adapting TRS-techniques so that they can be applied to LPs directly. In this way, we intended to combine the advantages of direct and transformational approaches. Indeed, a first prototypical implementation shows that our approach from [NGSD08] can handle both the examples "*der*" and "*shuffle*" above as well as other examples that could not be handled by *any* tool up to now (e.g., "*LP/SGST06-snake*" from the TPDB).

In this chapter we introduce a novel modular framework for termination analysis of LPs that extends and generalizes our approach of [NGSD08]. To this end, instead of adapting the dependency pair *approach* of [AG00], we adapt the dependency pair *framework* that we introduced in [GTS05a, GTSF06] to the LP context. With this new technique, termination analysis of programs like Example 4.1 can be performed by decomposing them into several simple sub-problems. Each of them can be solved independently by using any suitable well-founded order or indeed any of the modular techniques formulated in the framework.

The main difference to our work in [NGSD08] is that instead of a fixed combination of dependency graph analysis and polynomial orders, in the new framework any technique can be applied to the current modular problem at any time. Due to this flexible approach, we can propose a modular transformation from sub-problems in this new framework to our infinitary constructor dependency pair framework from Section 3.3.

This chapter is organized as follows. In Section 4.1, we provide some preliminaries about call sets and orders. In Section 4.2, we introduce the new modular framework for proving termination of LPs based on the dependency pair framework. We instantiate this framework with three modular techniques in Section 4.3. Finally, we summarize the contributions of this chapter in Section 4.4.

4.1. Preliminaries

A *quasi-order* on a set $\mathcal{S}$ is a reflexive and transitive binary relation $\succsim$ over $\mathcal{S}$. In this chapter, we use quasi-orders comparing atoms with each other and comparing terms with each other. We define the *associated equivalence relation* $\approx$ as $s \approx t$ if, and only if, $s \succsim t$ and $t \succsim s$. A *well-founded order* on $\mathcal{S}$ is a transitive relation $\succ$ where there is no infinite sequence $s_0 \succ s_1 \succ \dots$ with $s_i \in \mathcal{S}$. A *reduction pair* $(\succsim, \succ)$ consists of a quasi-order $\succsim$ and a well-founded order $\succ$ that are *compatible* (i.e., $t_1 \succsim t_2 \succ t_3$ implies $t_1 \succ t_3$).[1]

Let $\mathcal{P}$ be a logic program and let $\mathcal{S}$ be a set of atomic queries. The *call set*, $\mathit{Call}(\mathcal{S}, \mathcal{P})$, is the set of all atoms A, such that a variant of A is the selected atom in some derivation for $\mathcal{P}$ and Q for some $Q \in \mathcal{S}$. In this chapter, we use ranking functions and reduction pairs built from norms and level mappings [BCF94]. A *norm* is a mapping $\|\cdot\| : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathbb{N}$. A *level mapping* is a mapping $|\cdot| : \mathcal{A}(\Sigma, \Delta, \mathcal{V}) \to \mathbb{N}$. An *interargument relation* for a predicate p/n is a relation $\mathcal{R}_{p/n} = \{p(t_1, \dots, t_n) \mid t_i \in \mathcal{T}(\Sigma, \mathcal{V}) \wedge \varphi_p(t_1, \dots, t_n)\}$, where (1) $\varphi_p(t_1, \dots, t_n)$ 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 $t_1, \dots, t_n$ by applying function symbols of $\mathcal{P}$. $\mathcal{R}_{p/n}$ is *valid* if, and only if, for every $p(t_1, \dots, t_n) \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$: $p(t_1, \dots, t_n) \vdash^+_{\mathcal{P}} \square$ implies $p(t_1, \dots, t_n) \in \mathcal{R}_{p/n}$. A reduction pair $(\succsim, \succ)$ is *rigid* on a term t or an atom A if for all substitutions σ, we have $t \approx t\sigma$ and $A \approx A\sigma$, respectively. A reduction pair $(\succsim, \succ)$ is rigid on a set of terms or atoms if it is rigid on all its elements.

Example 4.2 (call set, norm, and level mapping for ack)**.** *We again regard the program $\mathcal{P}$ and the set of queries $\mathcal{S}$ in Example 4.1. Then we have $\mathit{Call}(\mathcal{P}, \mathcal{S}) = \mathcal{S} \cup \{\, \mathsf{p}(t_1, t_2) \mid t_1 \textit{ is a ground term, } t_2 \textit{ is a variable}\}$. Consider the reduction pair $(\succsim, \succ)$ which is induced*[2] *by a norm $\|\mathsf{0}\| = 0$, $\|\mathsf{s}(t)\| = 1 + \|t\|$, $\|X\| = 0$ for all variables X, and by an associated level mapping $|\mathsf{p}(t_1, t_2)| = 0$ and $|\mathsf{ack}(t_1, t_2, t_3)| = \|t_1\|$. Thus, we have* $\mathsf{s}(\mathsf{0}) \succ \mathsf{0}$, $\mathsf{ack}(\mathsf{s}(\mathsf{0}), X, Y) \succ \mathsf{ack}(\mathsf{0}, X, Y)$, *and* $\mathsf{ack}(\mathsf{0}, X, Y) \approx \mathsf{ack}(\mathsf{0}, \mathsf{0}, \mathsf{0})$. *Note that $(\succsim, \succ)$ is rigid on $\mathit{Call}(\mathcal{P}, \mathcal{S})$. An example for a valid interargument relation w.r.t. $(\succsim, \succ)$ is $\mathcal{R}_{\mathsf{p}/2} = \{\mathsf{p}(t_1, t_2) \mid t_1 \succ t_2\}$.*

[1] In contrast to the definition of "reduction pairs" in term rewriting (cf. Section 3.3), for the theoretical results in Section 4.2 we do not require $\succsim$ and $\succ$ to be closed under substitutions.

[2] So for terms t_1, t_2 we define $t_1 \,{}_{(}\!\succsim\!{}_{)}\, t_2$ if, and only if, $\|t_1\| \,{}_{(}\!\geq\!{}_{)}\, \|t_2\|$ and for atoms A_1, A_2 we define $A_1 \,{}_{(}\!\succsim\!{}_{)}\, A_2$ if, and only if, $|A_1| \,{}_{(}\!\geq\!{}_{)}\, |A_2|$.

4.2. Dependency Triple Framework

Definition 4.3 adapts the notion of *dependency pairs* [AG00] (cf. Definition 3.14) from TRSs to the LP setting. Note that in term rewriting one uses nested terms for calling auxiliary functions, while in logic programming auxiliary calls are implemented by intermediate body atoms. Thus instead of descending into right-hand sides of rules for building dependency pairs, for building dependency triples we have to consider all atoms in the body of a clause. Furthermore, here we need to keep track of the intermediate body atoms up to the atom that we are currently considering.

Definition 4.3 (Dependency Triple)**.** *A* dependency triple *is a clause $H \leftarrow I, B$ where H and B are atoms and I is a list of atoms. For a logic program P, we define the set $DT(\mathcal{P})$ of all dependency triples as $DT(\mathcal{P}) = \{H \leftarrow I, B \mid H \leftarrow I, B, \ldots \in \mathcal{P}\}$.*

For any finite logic program $\mathcal{P}$, the set of dependency triples $DT(\mathcal{P})$ is a finite set. To see this, let k be the number of clauses in $\mathcal{P}$ and let m be the maximal number of atoms in the body of a clause. Then the number of dependency triples in $DT(\mathcal{P})$ is bounded by $k * m$ and, thus, finite.

Example 4.4 (dependency triples of ack)**.** *Reconsider the program from Example 4.1. The dependency triples $DT(\mathcal{P})$ of the program are:*

$$\mathsf{ack}(X, \mathsf{0}, Z) \leftarrow \mathsf{p}(X, Y). \tag{1}$$

$$\mathsf{ack}(X, \mathsf{0}, Z) \leftarrow \mathsf{p}(X, Y), \mathsf{ack}(Y, \mathsf{s}(\mathsf{0}), Z). \tag{2}$$

$$\mathsf{ack}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{ack}(\mathsf{s}(X), Y, Z'). \tag{3}$$

$$\mathsf{ack}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{ack}(\mathsf{s}(X), Y, Z'), \mathsf{ack}(X, Z', Z). \tag{4}$$

Intuitively, a dependency triple $H \leftarrow I, B$ represents 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. The idea how to use dependency triples for termination analysis is to show that one cannot have infinite "chains" of such calls.

Definition 4.5 (Chain)**.** *Let $\mathcal{D}$ and $\mathcal{P}$ be sets of clauses. Let $\mathcal{C}$ be a set of atoms. A (possibly infinite) list of dependency triples $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ from $\mathcal{D}$ is a* $\mathcal{D}$-chain *w.r.t. $\mathcal{C}$ and $\mathcal{P}$ if, and only if, 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, $\sigma_i \in Answer(I_i\theta_i, \mathcal{P})$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}$.*

The above definition of a chain corresponds closely to the notion of chain introduced in Definition 3.16. Each dependency triple with k intermediate body atoms corresponds to $k + 1$ dependency pairs. In Theorem 4.22 we use this relation to transform dependency triples into a number of dependency pairs. The substitution $\theta_i\sigma_i$ corresponds to the substitutions used to instantiate the corresponding $k+1$ dependency pairs. Finally, the set $\mathcal{C}$ of atoms could be represented by, e.g., an argument filter (cf. Definition 3.12) defining which parts of an atom must be ground. Then the condition that $B_i\theta_i\sigma_i \in \mathcal{C}$ corresponds to the condition of Definition 3.16 that all terms in the chain have to be finite ground terms after application of the argument filter.

Example 4.6 (chain of ack)**.** Again we consider $\mathcal{P}$ and $\mathcal{S}$ from Example 4.1. The list of triples (2), (3) is a $DT(\mathcal{P})$-chain w.r.t. $Call(\mathcal{S},\mathcal{P})$ and $\mathcal{P}$. To see this, consider the substitutions $\theta_0, \sigma_0, \theta_1$ with $\theta_0 = \{X/\mathsf{s}(\mathsf{s}(0)), Z/0\}$, $\sigma_0 = \{Y/\mathsf{s}(0)\}$, and $\theta_1 = \{X/0, Y/0, Z/0\}$. Then, for $A = \mathsf{ack}(\mathsf{s}(\mathsf{s}(0)), 0, 0) \in \mathcal{S}$ we have $H_0\theta_0 = \mathsf{ack}(X, 0, Z)\theta_0 = \mathsf{ack}(\mathsf{s}(\mathsf{s}(0)), 0, 0) = A = A\theta_0$. Furthermore, $\sigma_0 \in Answer(\mathsf{p}(X, Y)\theta_0) = Answer(\mathsf{p}(\mathsf{s}(\mathsf{s}(0)), Y))$ and $B_0\theta_0\sigma_0 = \mathsf{ack}(\mathsf{s}(0), \mathsf{s}(0), 0) = \mathsf{ack}(\mathsf{s}(X), \mathsf{s}(Y), Z)\theta_1 \in Call(\mathcal{S},\mathcal{P})$.

Now we adapt the notion of dependency pair problems as the modular problems of the dependency pair framework [GTS05a] to the concept of dependency triple problems.

Definition 4.7 (Dependency Triple Problem)**.** *A* dependency triple 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. We say that a problem $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is* terminating *if, and only if, there are no infinite $\mathcal{D}$-chains w.r.t. $\mathcal{C}$ and $\mathcal{P}$. We say that a problem is* non-terminating *if, and only if, it is not* terminating.

Based on these problems and on the notion of termination, we can now show that our new dependency triple framework is sound and complete.

Theorem 4.8 (Soundness, Completeness)**.** *A logic program $\mathcal{P}$ is terminating w.r.t. a set of atomic queries $\mathcal{S}$ if, and only if, the dependency triple problem $(DT(\mathcal{P}), Call(\mathcal{S},\mathcal{P}), \mathcal{P})$ is terminating.*

Proof. For soundness, assume that $\mathcal{P}$ is not terminating w.r.t. $Call(\mathcal{S},\mathcal{P})$. Then there is an infinite derivation $Q_0, Q_1, \ldots$ with $Q_0 \in \mathcal{S}$ and $Q_i \vdash_{c_i,\delta_i} Q_{i+1}$. W.l.o.g. assume that the clause c_i has the form $H_i \leftarrow A_i^1, \ldots, A_i^{k_i}$. Let $j > 0$ be the minimal index such that the first atom A in Q_j starts an infinite derivation. Such a j always exists as shown in Lemma 3.11. As we started from an atomic query, there must be some m_0 such that $A = A_0^{m_0}\delta_0\delta_1\ldots\delta_{j-1}$. Then $H_0 \leftarrow A_0^1, \ldots, A_0^{m_0-1}, A_0^{m_0} \in DT(\mathcal{P})$ is the first dependency triple in our $DT(\mathcal{P})$-chain w.r.t. $Call(\mathcal{S},\mathcal{P})$ and $\mathcal{P}$ with $\theta_0 = \delta_0$ and $\sigma_0 = \delta_1 \ldots \delta_{j-1}$. As A is the selected atom in some derivation for $\mathcal{P}$ and $Q_0 \in \mathcal{S}$, we have $A \in Call(\mathcal{S},\mathcal{P})$. We repeat this construction starting from A and obtain $H_1 \leftarrow A_1^1, \ldots, A_1^{m_1-1}, A_1^{m_1} \in DT(\mathcal{P})$ such that $\theta_1 = \delta_j = mgu(A_0^{m_0}, A_1^{m_1})$. By repeating this construction over and over, we obtain an infinite $DT(\mathcal{P})$-chain w.r.t. $Call(\mathcal{S},\mathcal{P})$ and $\mathcal{P}$. Thus, the problem $(DT(\mathcal{P}), Call(\mathcal{S},\mathcal{P}), \mathcal{P})$ is not terminating.

For completeness, assume that $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ is a non-terminating $DT(\mathcal{P})$-chain w.r.t. $Call(\mathcal{S}, \mathcal{P})$ and $\mathcal{P}$. Thus, there are substitutions θ_i, σ_i and an $A \in Call(\mathcal{S}, \mathcal{P})$ such that $\theta_0 = mgu(A, H_0)$ and for all i, $\sigma_i \in Answer(I_i\theta_i, \mathcal{P})$ 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 $\sigma_0 \in Answer(I_0\theta_0, \mathcal{P})$ we obtain the derivation $I_0\theta_0 \vdash^*_{\sigma_0} \Box$ and, consequently, $I_0\theta_0, B_0\theta_0, R_0\theta_0 \vdash^*_{\sigma_0} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0$. Together with the first step we obtain the derivation $Q_0 \vdash_{\theta_0\sigma_0} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0$. Now, as $B_0\theta_0\sigma_0\theta_1 = H_1\theta_1$ and as there is a rule $c_1 = H_1 \leftarrow I_1, B_1, R_1 \in \mathcal{P}$, we continue the derivation with $B_0\theta_0\sigma_0, R_0\theta_0\sigma_0 \vdash_{\theta_1} I_1\theta_1, B_1\theta_1, R_0\theta_0\sigma_0\theta_1,\ R_1\theta_1$. Due to $\sigma_1 \in Answer(I_1\theta_1, \mathcal{P})$ we continue with $I_1\theta_1, B_1\theta_1, R_0\theta_0\sigma_0\theta_1, R_1\theta_1 \vdash_{\sigma_1} B_1\theta_1\sigma_1, R_0\theta_0\sigma_0\theta_1\sigma_1, R_1\theta_1\sigma_1$.

By continuing this construction, we obtain an infinite chain $A \vdash_{\theta_0\sigma_0} B_0\theta_0\sigma_0, R_0\theta_0\sigma_0 \vdash_{\theta_{1,\sigma_1}} B_1\theta_1\sigma_1, R_0\theta_0\sigma_0\theta_1\sigma_1, R_1\theta_1\sigma_1 \vdash_{\theta_2\sigma_2} B_2\theta_2\sigma_2, \ldots \vdash_{\theta_{3,\sigma_3}} \ldots$. Thus, the logic program $\mathcal{P}$ is not terminating w.r.t. $Call(\mathcal{S}, \mathcal{P})$. From $A \in Call(\mathcal{S}, \mathcal{P})$ we know there is a $Q \in \mathcal{S}$ such that a variant A' of A is the selected atom in some derivation for $\mathcal{P}$ and Q. As the termination behavior of A and A' is identical, we obtain that $\mathcal{P}$ is not terminating w.r.t. $Q \in \mathcal{S}$. $\Box$

Finally, we adapt the notion of dependency pair processors to our framework in order to formalize the modular techniques to be used in our framework.

Definition 4.9 (Dependency Triple Processor). *A dependency triple processor Proc is a function that takes a dependency triple problem as input and returns a set of dependency triple problems, i.e., for any problem $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ we have $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \{(\mathcal{D}_1, \mathcal{C}_1, \mathcal{P}_1), \ldots, (\mathcal{D}_n, \mathcal{C}_n, \mathcal{P}_n)\}$.*

A processor Proc is sound *if, and only if, whenever all $(\mathcal{D}_i, \mathcal{C}_i, \mathcal{P}_i)$ are terminating, then also $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is terminating. A processor Proc is* complete *if, and only if, whenever there is some non-terminating $(\mathcal{D}_i, \mathcal{C}_i, \mathcal{P}_i)$, then $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is also non-terminating.*

The intuition behind such a processor is that it takes a dependency triple problem and splits it up into sub-problems, returns a simplified problem, or shows termination of the given problem by some other means.

Now we have all the ingredients needed in our framework for analyzing termination of logic programs. The idea for termination analysis is to apply sound processors repeatedly to the initial dependency triple problem and to the resulting sub-problems and simplified problems. If we can show termination of all sub-problems, i.e., for each leaf of our proof tree some processor returns the empty set, then we have shown termination of the original logic program. While soundness is essential to be able to conclude termination of the original problem, completeness is essential to allow the development of modular techniques for non-termination analysis, e.g., by adapting the techniques of [GTS05b, PM06].

4.3. Dependency Triple Processors

Note that while we now know the basic components of our framework, i.e., dependency triple problems and dependency triple processors, we are still missing concrete instances of dependency triple processors that can be used to split, simplify, and solve dependency triple problems. First, we will introduce a processor based on the so-called *dependency graph*. To this end, we adapt the notion of the (estimated) dependency graph [AG00] from TRSs to LPs.[3] While "dependency triples" are related to the "binary clauses" of [CT99], our notion of dependency graphs for LPs is similar to the "atom dependency graph" of [DLSS01]. But in contrast to [DLSS01], we use dependency graphs to modularize termination proofs such that *several different* reduction pairs can be used in the termination proof of one program.

Definition 4.10 (Dependency Graph)**.** *Let $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ be a dependency triple problem. The* dependency graph *for $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is a directed graph whose vertices are the clauses of $\mathcal{D}$ and there is an arc from a vertex N to a vertex M if, and only if, N, M is a $\mathcal{D}$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$.*

As the real dependency graph is not computable, we need an estimation. Here, we adapt the idea of [AG00]. Note that in this estimation, we ignore the intermediate body atoms, which corresponds closely to the use of the *CAP* function in [AG00]. [4] [5]

Definition 4.11 (Estimated Dependency Graph)**.** *Let $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ be a dependency triple problem. The* estimated dependency graph *for $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is a directed graph whose vertices are the clauses $\mathcal{D}$ and there is an arc from a vertex $H_i \leftarrow I_i, B_i$ to a vertex $H_j \leftarrow I_j, B_j$, if, and only if, B_i unifies with H_j and there are atoms $A_i, A_j \in \mathcal{C}$ such that A_i unifies with H_i and A_j unifies with H_j.*

Example 4.12 (dependency graph for ack)**.** The following graph shows the dependency graph for the ack-program from Example 4.1.

$$\begin{tikzcd}
\boxed{(1)\ \mathsf{ack}(X,0,Z) \leftarrow \mathsf{p}(X,Y).} & \boxed{(2)\ \mathsf{ack}(X,0,Z) \leftarrow \mathsf{p}(X,Y), \mathsf{ack}(Y,\mathsf{s}(0),Z).} \\
\boxed{(3)\ \mathsf{ack}(\mathsf{s}(X),\mathsf{s}(Y),Z) \leftarrow \mathsf{ack}(\mathsf{s}(X),Y,Z').} \arrow[u] \arrow[ur, leftrightarrow] \arrow[r, leftrightarrow] \arrow[loop below] & \boxed{(4)\ \mathsf{ack}(\mathsf{s}(X),\mathsf{s}(Y),Z) \leftarrow \mathsf{ack}(\mathsf{s}(X),Y,Z'), \mathsf{ack}(X,Z',Z).} \arrow[ul] \arrow[u, leftrightarrow] \arrow[loop below]
\end{tikzcd}$$

Note that for this example, the estimated dependency graph coincides with the real dependency graph. This is, for instance, not the case for $(\{\mathsf{p} \leftarrow \mathsf{q}(\mathsf{a}), \mathsf{p}\}, \{\mathsf{q}(\mathsf{b})\}, \{\mathsf{p}\})$.

[3] Our notion should not be confused with the notion of the "(predicate) dependency graph" from [BAK91, DLSS01, Plü90] that simply represents the dependencies between different predicate symbols.

[4] In [NGSD08] we did not distinguish between the real and the estimated dependency graph. There, the dependency graph corresponds to the estimated one here.

[5] For the dependency graphs in term rewriting, better estimations have been developed (cf. [HM05a, GTS05b]). Similar improvements should be possible in our new framework when taking the intermediate body atoms of the dependency triples into account.

To be able to use the estimated dependency graph in general, we need to prove that it is an overapproximation of the real dependency graph, i.e., that the former contains at least all edges of the latter.

Lemma 4.13 (Estimated Dependency Graph is an Over-Approximation)**.** *The estimated dependency graph for $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ overapproximates the real dependency graph for $(\mathcal{D}, \mathcal{C}, \mathcal{P})$, i.e., whenever there is an arc between a vertex N and a vertex M in the real graph, there is also a vertex in the estimated graph.*

Proof. If there is an arc from $N = H_i \leftarrow I_i, B_i$ to $M = H_j \leftarrow I_j, B_j$ in the real graph, we know that $(H_i \leftarrow I_i, B_i), (H_j \leftarrow I_j, B_j)$ is a $\mathcal{D}$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$. Thus, there are substitutions θ_0, θ_1, and σ_0 and some $A \in \mathcal{C}$ such that $\theta_0 = mgu(A, H_i)$, $\theta_1 = mgu(B_i\theta_0\sigma_0, H_j)$ and $B_i\theta_0\sigma_0 \in \mathcal{C}$. As we assume B_i and H_j to be variable disjoint, we can define a new substitution δ such that $\delta|_{\mathcal{V}(B_i)} = \theta_0\sigma_0\theta_1$ and $\delta|_{\mathcal{V}(H_j)} = \theta_1$. Then, clearly, δ is a unifier of B_i and H_j and there is an arc in the estimated dependency graph. □

Now, every infinite derivation of the program has to have an infinite suffix that corresponds to a *strongly connected component* (SCC) of the dependency graph. The basic idea of the processor based on the dependency graph is to split the dependency triple problem into sub-problems corresponding to the SCCs of the dependency graph. In this way, dependency triples that are not part of an SCC can be deleted and (mutually) recursive clusters of dependency triples can be analyzed separately.

Example 4.14 (SCCs for ack)**.** The dependency graph in Example 4.12 contains exactly one SCC: $\{(2), (3), (4)\}$. This implies that (1) cannot occur infinitely often in an infinite chain and can be deleted.

Before we introduce the processor based on the dependency graph formally, we note that all processors deleting elements from $\mathcal{D}$, $\mathcal{C}$, or $\mathcal{P}$ are always complete. This follows directly from the following lemma.

Lemma 4.15 (Completeness of Reducing Problems)**.** *Let $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ and $(\mathcal{D}', \mathcal{C}', \mathcal{P}')$ be dependency triple problems with $\mathcal{D}' \subseteq \mathcal{D}$, $\mathcal{C}' \subseteq \mathcal{C}$, and $\mathcal{P}' \subseteq \mathcal{P}$. Then $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ is non-terminating whenever $(\mathcal{D}', \mathcal{C}', \mathcal{P}')$ is non-terminating.*

Proof. Let $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ be a non-terminating $\mathcal{D}'$*-chain* w.r.t. $\mathcal{C}'$ and $\mathcal{P}'$. Thus, 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, $\sigma_i \in Answer(I_i\theta_i, \mathcal{P}')$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}'$.

From $\mathcal{D}' \subseteq \mathcal{D}$ we know that $H_i \leftarrow I_i, B_i \in \mathcal{D}$, from $\mathcal{C}' \subseteq \mathcal{C}$ we obtain $A \in \mathcal{C}$ and $B_i\theta_i\sigma_i \in \mathcal{C}$, and, finally, from $\mathcal{P}' \subseteq \mathcal{P}$ we get $\sigma_i \in Answer(I_i\theta_i, \mathcal{P})$. □

Now we can formulate our first processor based on the dependency graph. This is a straightforward adaption of the one in [AG00] and Definition 3.22.

Theorem 4.16 (Dependency Graph Processor). *Let* $Proc((\mathcal{D},\mathcal{C},\mathcal{P})) = \{(\mathcal{D}_1,\mathcal{C},\mathcal{P}),\dots,(\mathcal{D}_n,\mathcal{C},\mathcal{P})\}$*, where* $\mathcal{D}_1,\dots,\mathcal{D}_n$ *are the SCCs of the (estimated) dependency graph for* $(\mathcal{D},\mathcal{C},\mathcal{P})$*. Then Proc is sound and complete.*

Proof. For soundness, we proceed by contradiction. Assume that $(\mathcal{D},\mathcal{C},\mathcal{P})$ is non-terminating, i.e., there is an infinite $\mathcal{D}$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$. By definition of the dependency graph, this infinite chain corresponds to an infinite path in the real dependency graph. From the finiteness of $\mathcal{D}$ it follows that a suffix of that path must be contained entirely in one of the SCCs. To see this, assume that the path leaves every SCC that it enters at some time. Then there must be an infinite number of SCCs and, thus, an infinite number of vertices in the dependency graph. But this is a contradiction to the finiteness of $\mathcal{D}$. Without loss of generality, assume that this suffix is contained in $(\mathcal{D}_1,\mathcal{C},\mathcal{P})$. Then there is an infinite $\mathcal{D}_1$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$ and $(\mathcal{D}_1,\mathcal{C},\mathcal{P})$ is non-terminating. According to Lemma 4.13, the estimated dependency graph contains the real dependency graph and, consequently, the application of *Proc* also yields a non-terminating dependency triple problem.

For completeness, we observe that $\mathcal{D}_i \subseteq \mathcal{D}$ for all $1 \le i \le n$. Using Lemma 4.15 we immediately obtain the completeness of *Proc*. □

Example 4.17 (dependency graph processor on ack). For the dependency triple problem $(DT(\mathcal{P}), Call(\mathcal{S},\mathcal{P}), \mathcal{P})$ where $\mathcal{P}$ is the logic program from Example 4.1 and *Proc* is the processor from Theorem 4.16, by applying *Proc* we obtain one new dependency triple problem $(\{(2),(3),(4)\}, Call(\mathcal{S},\mathcal{P}), \mathcal{P})$, i.e., we delete (1) from $DT(\mathcal{P})$.

While we now can use the dependency triple processor based on the dependency graph to delete some dependency triples, for those that are part of a strongly connected component, we need a different dependency triple processor. One of the most useful dependency pair processor from [GTS05a, GTSF06] is the processor based on *reduction pairs*.

The basic idea is to inspect each SCC of the dependency graph separately and to find a reduction pair $(\succsim,\succ)$ such that *some* dependency triples of the SCC are *strictly* decreasing (i.e., w.r.t. $\succ$) and all others are *weakly* decreasing (i.e., w.r.t. $\succsim$). The following definition formalizes when a dependency triple is considered to be "decreasing". It relies on interargument relations for the predicates of the program. We have explained how to synthesize such interargument relations and how to find reduction pairs automatically that make dependency triples "decreasing" in our paper [NGSD08].

Note that the interargument relations are needed to approximate the semantics of the intermediate body atoms I in our dependency triples $H \leftarrow I, B$. This corresponds closely to demanding that all rules in $\mathcal{R}$ in our dependency pairs $(\mathcal{D},\mathcal{R},\pi)$ are weakly decreasing w.r.t. $(\succsim,\succ)$. In both cases, we need to ensure that the derivation of the intermediate body atoms of the triples and the evaluation of nested terms in the right hand sides of dependency terms, respectively, do not lead to a strict increase in the size of arguments considered by our reduction pair.

Definition 4.18 (decreasing dependency triples). *Let $\mathcal{P}$ be a logic program. Let $(\succsim, \succ)$ be a reduction pair and $\mathfrak{R} = \{\mathcal{R}_{p_1}, \ldots, \mathcal{R}_{p_k}\}$ be a set of interargument relations based on $(\succsim, \succ)$ for the predicates $p_1, \ldots, p_k$ defined in $\mathcal{P}$. Let $N = H \leftarrow p_1(\vec{t_1}), \ldots, p_n(\vec{t_n}), B$ be a dependency triple in $DT(\mathcal{P})$.*

N is weakly decreasing *(denoted $(\succsim, \mathfrak{R}) \models N$) if $H\sigma \succsim B\sigma$ holds for any substitution σ where $(\succsim, \succ)$ is rigid on $H\sigma$ and where $p_1(\vec{t_1})\sigma \in \mathcal{R}_{p_1}, \ldots, p_n(\vec{t_n})\sigma \in \mathcal{R}_{p_n}$. Analogously, N is* strictly decreasing *(denoted $(\succ, \mathfrak{R}) \models N$) if $H\sigma \succ B\sigma$ holds for any such σ.*

Example 4.19 (decreasing dependency triples for ack). Consider the reduction pair $(\succsim, \succ)$ from Example 4.2. Let $\mathfrak{R}$ be the set of valid interargument relations where $\mathcal{R}_{\mathsf{ack}/3} = \{\mathsf{ack}(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in \mathcal{T}(\Sigma, \mathcal{V})\}$ and where $\mathcal{R}_{\mathsf{p}/2}$ is defined as in Example 4.2. Then we have $(\succ, \mathfrak{R}) \models (2)$. The reason is that for any substitution σ where $(\succsim, \succ)$ is rigid on $\mathsf{ack}(X, \mathsf{0}, Z)\sigma$ (i.e., where $X\sigma$ is a ground term) and where $\mathsf{p}(X, Y)\sigma \in \mathcal{R}_{\mathsf{p}/2}$ (i.e., where $X\sigma \succ Y\sigma$), we have $\mathsf{ack}(X, \mathsf{0}, Z)\sigma \succ \mathsf{ack}(Y, \mathsf{s}(\mathsf{0}), Z)\sigma$. Similarly, we also have $(\succsim, \mathfrak{R}) \models (3)$ and $(\succ, \mathfrak{R}) \models (4)$.

We can now formulate our second dependency triple processor.

Theorem 4.20 (Reduction Pair Processor). *Let $(\succsim, \succ)$ be a reduction pair and $\mathfrak{R} = \{\mathcal{R}_{p_1}, \ldots, \mathcal{R}_{p_k}\}$ be a set of valid interargument relations for the predicates $p_1, \ldots, p_k$ defined in $\mathcal{P}$. Then the following processor Proc is sound and complete. Here, $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) =$*

- $\{(\mathcal{D} \setminus \mathcal{D}_\succ, \mathcal{C}, \mathcal{P})\}$, *if*
  - $(\succsim, \succ)$ *is rigid on $\mathcal{C}$ and*
  - *there is a non-empty subset $\mathcal{D}_\succ \subseteq \mathcal{D}$ such that $(\succ, \mathfrak{R}) \models N$ for all $N \in \mathcal{D}_\succ$ and $(\succsim, \mathfrak{R}) \models N$ for all $N \in \mathcal{D} \setminus \mathcal{D}_\succ$*
- $\{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}$, *otherwise*

Proof. The definition of *Proc* has two cases. First, if $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}$, *Proc* is trivially sound and complete.

Second, we consider soundness of the case when $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \{(\mathcal{D} \setminus \mathcal{D}_\succ, \mathcal{C}, \mathcal{P})\}$. Then there is a set of valid interargument relations $\mathfrak{R}$, a reduction pair $(\succsim, \succ)$ that is rigid on $\mathcal{C}$, and a non-empty set $\mathcal{D}_\succ \subseteq \mathcal{D}$ such that $(\succ, \mathfrak{R}) \models N$ for all $N \in \mathcal{D}_\succ$ and $(\succsim, \mathfrak{R}) \models N$ for all $N \in \mathcal{D} \setminus \mathcal{D}_\succ$. 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 $\mathcal{D}$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$. Now, if no clause from $\mathcal{D}_\succ$ appears infinitely often, there is an infinite suffix which forms a $(\mathcal{D}\setminus\mathcal{D}_\succ)$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$ and $(\mathcal{D}\setminus\mathcal{D}_\succ, \mathcal{C}, \mathcal{P})$ is non-terminating, which contradicts our assumption. Thus, at least one clause from $\mathcal{D}_\succ$ must appear infinitely often in the chain. Without loss of generality, let this chain be $(H_0 \leftarrow I_0.B_0), (H_1 \leftarrow I_1, B_1), \ldots$ with $A \in \mathcal{C}$ and substitutions θ_i, σ_i such that $\theta_0 = mgu(A, H_0)$ and for all i, $\sigma_i \in Answer(I_i\theta_i, \mathcal{P})$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}$. We can now build the sequence

$$\begin{array}{rcl}
H_i\theta_i & \approx & \text{(by rigidity, since } H_i\theta_i = B_{i-1}\theta_{i-1}\sigma_{i-1}\theta_i \\
 & & \text{and } B_{i-1}\theta_{i-1}\sigma_{i-1} \in \mathcal{C}) \\
H_i\theta_i\sigma_i\theta_{i+1} & \succsim & \\
B_i\theta_i\sigma_i\theta_{i+1} & = & \\
H_{i+1}\theta_{i+1} & \approx & \text{(by rigidity, since } H_{i+1}\theta_{i+1} = B_i\theta_i\sigma_i\theta_{i+1} \\
 & & \text{and } B_i\theta_i\sigma_i \in \mathcal{C}) \\
H_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} & \succsim & \\
B_i\theta_{i+1}\sigma_{i+1}\theta_{i+2} & = & \\
\dots & &
\end{array}$$

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

For completeness, we observe that $\mathcal{D} \setminus \mathcal{D}_{\succ} \subseteq \mathcal{D}$. Using Lemma 4.15 we immediately obtain the completeness of *Proc*. □

Example 4.21 (finishing the proof for ack)**.** After applying the reduction pair processor, we are left with the dependency triple problem $(\{(3)\}, Call(\mathcal{S}, \mathcal{P}), \mathcal{P})$.

Now, consider the reduction pair $(\succsim, \succ)$ which is induced by a norm $\|\mathsf{s}(t)\| = 1 + \|t\|$, $\|X\| = 0$ for all variables X, and by an associated level mapping $|\mathsf{ack}(t_1, t_2, t_3)| = \|t_2\|$. Then, we have $(\succ, \mathfrak{R}) \models (3)$.

For the remaining dependency triple problem $(\varnothing, Call(\mathcal{S}, \mathcal{P}), \mathcal{P})$ we can use the dependency graph processor *Proc*: $Proc((\varnothing, Call(\mathcal{S}, \mathcal{P}), \mathcal{P})) = \varnothing$. Thus, we have indeed proven termination of the logic program from Example 4.1.

In this way, our method can use different reduction pairs for different SCCs of the dependency graph. Moreover, one can also use several different reduction pairs in the termination analysis of one single SCC, since SCCs are handled in an incremental way by removing dependency triples one after another.

However, in our approach we may only use reduction pairs $(\succsim, \succ)$ that are rigid on $Call(\mathcal{S}, \mathcal{P})$. This prevents an increase of atoms and terms due to further instantiations in subsequent derivation steps. For details, we refer to [ND05].

Modular Transformation to Term Rewriting

With the processors based on the dependency graph and on reduction pairs, we have introduced a powerful and useful framework for direct termination analysis of logic programs that subsumes our contribution in [NGSD08].

The full power of our transformational method from Chapter 3 depends on being able to use all the other modular techniques available for term rewriting, though. For instance, this includes *semantic labelling* [Zan95] and *matchbounds* [GHW04].

While some of these techniques as well as further more sophisticated processors from [GTS05a, GTSF06] might be adapted to our new framework, it would be very interesting to reuse existing techniques developed for term rewriting directly. Additionally, the use of reduction pairs based on path orders (for example recursive path orders [Les83]) is not possible with our framework yet, as it is open how to specify interargument relations for syntactic orders in a general way.

For these cases, we present another dependency triple processor based on the *transformation* of Chapter 3.

Theorem 4.22 (Transformation Processor). *Let $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ be a dependency triple problem over the signature (Σ, Δ). Let $\pi_{\mathcal{C}}$ be an argument filter such that we have $\pi_{\mathcal{C}}(A) \in \mathcal{A}(\Sigma, \Delta)$ for all $A \in \mathcal{C}$ and $\pi_{\mathcal{C}}(p) = \pi_{\mathcal{C}}(p_{in})$ for all $p \in \Delta$. Let $\mathcal{R}_{\mathcal{D}}$ and $\mathcal{R}_{\mathcal{P}}$ result from $\mathcal{D}$ resp. $\mathcal{P}$ by the transformation of Definition 3.7. Let $\pi'_{\mathcal{C}}$ be a refinement of $\pi_{\mathcal{C}}$ such that $\pi'_{\mathcal{C}}(\mathcal{R}_{\mathcal{D}})$ and $\pi'_{\mathcal{C}}(\mathcal{R}_{\mathcal{P}})$ satisfy the variable condition.*

Then Proc with $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \varnothing$, if $(DP(\mathcal{R}_{\mathcal{D}}), \mathcal{R}_{\mathcal{P}}, \pi'_{\mathcal{C}})$ is a terminating dependency pair problem, and $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}$, otherwise, is a sound and complete dependency triple processor.

Proof. For the case $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \{(\mathcal{D}, \mathcal{C}, \mathcal{P})\}$, soundness and completeness hold trivially. For the case $Proc((\mathcal{D}, \mathcal{C}, \mathcal{P})) = \varnothing$ completeness holds trivially, too, while soundness can be proved by contradiction. Assume there is an infinite $\mathcal{D}$-chain w.r.t. $\mathcal{C}$ and $\mathcal{P}$. Without loss of generality, this chain has the form $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ and there is a substitution θ_0 such that $\theta_0 = mgu(A, H_0)$ for some $A \in \mathcal{C}$. From the soundness of the transformation (in particular Lemma 3.10) and the definition of $\pi'_{\mathcal{C}}$ we know there are terms $s_0, s_1, \ldots$ and $t_0, t_1, \ldots$ such that $s_0 \in \mathcal{S}_{\pi'_{\mathcal{C}}}$ and $s_0 \overset{\infty}{\rightarrow}_{DP(\mathcal{R}_{\mathcal{D}})} t_0 \overset{\infty}{\rightarrow}{}^{*}_{\mathcal{R}_{\mathcal{P}}} s_1 \overset{\infty}{\rightarrow}_{DP(\mathcal{R}_{\mathcal{D}})} t_1 \overset{\infty}{\rightarrow}{}^{*}_{\mathcal{R}_{\mathcal{P}}} \ldots$. As all substitutions only use constructor symbols and $\pi'_{\mathcal{C}}(\mathcal{R}_{\mathcal{D}})$ and $\pi'_{\mathcal{C}}(\mathcal{R}_{\mathcal{P}})$ satisfy the variable condition, this implies that $(DP(\mathcal{R}_{\mathcal{D}}), \mathcal{R}_{\mathcal{P}}, \pi'_{\mathcal{C}})$ is non-terminating, which is a contradiction. □

By this transformation we are guaranteed to have the best of both worlds. We can use the direct (and, in general, more efficient) dependency triple framework to solve as many sub-problems as possible. If we cannot solve a particular hard sub-problem, we can use the transformation and obtain a problem in the dependency pair framework for infinitary constructor rewriting which is already reduced compared to the problem we would obtain from directly transforming the original logic program.

Example 4.23 (translation for ack). After applying the dependency triple processor based on the dependency graph and the dependency triple processor based on reduction pairs to the initial dependency triple problem for Example 4.1 once, we are left with the dependency triple problem $(\{\mathsf{ack}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{ack}(\mathsf{s}(X), Y, Z').\}, \mathit{Call}(\mathcal{S}, \mathcal{P}), \mathcal{P})$. The result of our new processor based on transformation would be the infinitary constructor rewriting dependency pair problem $(DP(\mathcal{R}_\mathcal{D}), \mathcal{R}_\mathcal{P}, \pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})})$ where the set $DP(\mathcal{R}_\mathcal{D})$ contains the two dependency pairs $\mathsf{ACK}_{in}(\mathsf{s}(X), \mathsf{s}(Y), Z) \to \mathsf{U}(\mathsf{ack}_{in}(\mathsf{s}(X), Y, Z'), X, Y, Z)$ and $\mathsf{ACK}_{in}(\mathsf{s}(X), \mathsf{s}(Y), Z) \to \mathsf{ACK}_{in}(\mathsf{s}(X), Y, Z')$ and for $\pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})}$ we have, in particular, $\pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})}(\mathsf{ack}_{in}) = \pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})}(\mathsf{ACK}_{in}) = \{1, 2\}$ and $\pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})}(\mathsf{u}) = \pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})}(\mathsf{s}) = \{1\}$.

By applying the dependency pair processor based on the dependency graph from Theorem 3.22 once, we can delete the first dependency pair and obtain the new dependency triple problem $(\{\mathsf{ACK}_{in}(\mathsf{s}(X), \mathsf{s}(Y), Z) \to \mathsf{ACK}_{in}(\mathsf{s}(X), Y, Z')\}, \mathcal{R}_\mathcal{P}, \pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})})$. One application of the dependency pair processor based on reduction pairs from Theorem 3.24 with for example any recursive path order deletes this pair, too. Then, by applying Theorem 3.22 once more, we have successfully shown termination of the dependency pair problem $(DP(\mathcal{R}_\mathcal{D}), \mathcal{R}_\mathcal{P}, \pi'_{\mathit{Call}(\mathcal{S},\mathcal{P})})$ and, thus, termination of the dependency triple problem $(DT(\mathcal{P}), \mathit{Call}(\mathcal{S}, \mathcal{P}), \mathcal{P})$, i.e., termination of $\mathcal{P}$ w.r.t. $\mathcal{S}$.

Automating the Framework

Last but not least, we show how to automate our new framework. The one main ingredient needed is the following general strategy that defines which processors should be used on the initial dependency triple problem in what order.

1. For a logic program $\mathcal{P}$, start with the problem $(DT(\mathcal{P}), \mathit{Call}(\mathcal{S}, \mathcal{P}), \mathcal{P})$.
2. Apply the processor based on the *dependency graph* from Theorem 4.16.
   If there are no further sub-problems, return "Success".
3. Apply the processor based on *reduction pairs* from Theorem 4.20. If some triples have been deleted, go to Step 2.
4. Apply the processor based on *transformation* from Theorem 4.22. If some triples have been deleted, go to Step 2.
5. Return "Failure".

We are left with explaining how the termination analysis technique based on polynomial interpretations from [ND05, ND07] can be applied to the framework. The basic idea is that instead of fixing a polynomial interpretation and interargument relations *before* performing the termination proof, we only fix the degree of the polynomials used in the polynomial interpretation (e.g., linear or quadratic ones). Then we can automatically generate symbolic constraints and try to solve them afterwards. In this way, polynomial interpretations and interargument relations can be synthesized fully automatically. For a detailed description of how to do this, we refer to [NGSD08, Section 4.2].

4.4. Summary

We have introduced a new framework for termination analysis of LPs: the dependency triple framework. Our contribution is threefold: **(1)** like our earlier work in [NGSD08], it results in a weaker condition for verifying termination of LPs, where the decrease condition is established for parts of the strongly connected components of the dependency graph, and not at the clause level, as it has been done before; **(2)** it introduces a modular approach in which termination conditions can be separated into different groups, each of which can be treated independently by automatically searching for different suitable well-founded orderings; **(3)** it combines direct and transformational approaches by allowing for modular application of transformations.

A difference between the dependency pair framework for TRSs and our approach is that instead of separating between defined symbols and constructors as for TRSs, we separate between predicate and function symbols of the LP. Another main difference is that in the dependency pair method for TRSs, one requires a weak decrease for the rules of the TRS in order to take the effect of "nested" functions in recursive arguments into account. In the LP-context, these nested functions correspond to body atoms preceding recursive calls. We store these atoms in an additional component of the dependency pair (yielding dependency triples) and take their effect into account by considering interargument relations.

The author of this thesis was involved in the implementation of two of the most powerful automated termination analyzers for LPs (Polytool, which follows the approach of [ND05, ND07, NGSD08], and AProVE [GST06], which uses the transformation from Chapter 3 to transforms LPs to TRSs and then tries to prove termination of the resulting TRS.) AProVE was the most successful termination prover for logic programs, functional programs, and term rewrite systems in all annual *International Competitions of Termination Tools* 2004 – 2007 [MZ07], where Polytool obtained a close second place for logic programs in the 2007 competition. As mentioned in Chapter 3, there exist many LPs where termination can currently only be proved by transformational tools like AProVE, but there are also examples where the termination proof only succeeds with direct tools like Polytool. The results of this chapter combine the advantages of both approaches by adapting TRS-techniques like dependency pairs to direct termination approaches for LPs and even allowing for transformations to be applied at a modular level.

A first prototypical implementation of the techniques described in [NGSD08] and this chapter (excluding the dependency triple processor based on transformation) already proves termination for 220 out of 296 examples under conditions identical to the experimental setup of Chapter 3. In the table below, we refer to the new implementation as Polytool 2 to distinguish it from the version of Polytool that implements [ND05].

	AProVE	Polytool 2	Polytool	TerminWeb	cTI	TALP
Successes	**232**	*220*	204	177	167	163
Failures	**57**	*65*	82	118	129	112
Timeouts	**7**	*11*	10	1	0	21
Total	**1471.4**	*1517.4*	622.7	95.3	10.4	413.5

This implementation also shows that in this way one can handle (a) examples that could up to now only be solved with direct tools such as [ND05, "*der*"], (b) examples that could up to now only be solved with transformational tools based on dependency pairs such as [TPD07, "*LP/SGST06-shuffle*"], as well as (c) examples like [TPD07, "*LP/SGST06-snake*"] that could not be solved by any tool up to now.

Our modular transformation from Theorem 4.22 and the general strategy outlined at the end of the previous section yield an approach that is more powerful than the one of Chapter 3 and more powerful than the one of this chapter without Theorem 4.22. To see this, consider the logic program with consists of all clauses from [ND05, "*der*"] and all clauses from [TPD07, "*LP/SGST06-shuffle*"]. This program cannot be shown terminating by either approach.

Using the dependency triple processor from Theorem 4.16, though, we can split the initial dependency triple problem into four problems, one for d from *der* and three for append, reverse, and shuffle from *shuffle*. The problems for d, append, and reverse can be handled by the processors from Theorems 4.16 and 4.20. For shuffle, we need to apply the processor from Theorem 4.22. The resulting dependency pair problem can then easily be solved by the techniques from Chapter 3.

Future Work

Note that the current formulation of our new framework assumes unification with occur check. Future work should be to investigate how this new framework can be adapted to the case of unification without occur check. This requires to keep track of potentially infinite terms similar to the way this is handled in Chapter 3.

While this chapter only adapted basic concepts of the dependency pair method to the LP setting, it would be interesting to adapt further more sophisticated dependency pair processors [GTS05a, GTSF06] to our dependency triple framework as well. One could also develop completely new processors in this framework that rely on special properties of logic programming

5. Logic Programs with Cuts

As noted in Chapters 3 and 4, termination of logic programs is widely studied. This is mostly due to the importance of termination analysis when one is developing and using Prolog programs.

Still, the presented techniques for termination analysis are limited to *definite* logic programs. There are several major differences between this notion and Prolog programs:

(i) In definite logic programs, the only method of computation is left-to-right, depth-first search (SLD resolution). In practice, virtually all Prolog programs make use of additional extra-logical constructs to **cut** the search space (! operator) or implement some kind of negation (\+ operator). Currently, there is no termination analysis for logic programs with cuts.

(ii) When speaking of termination, one might be interested in either universal termination (finiteness of the SLD tree) or existential termination (failure or first answer after a finite number of derivation steps). With very few exceptions (cf. [Mar96]), only universal termination is analyzed.

(iii) In definite logic programs there is a clear distinction between predicate symbols and function symbols and, consequently, between atoms and terms. In Prolog there is no such distinction and when one is using so-called meta-programming, "atoms" may well be arguments of other atoms or terms. Using meta-programming, negation-as-failure can, e.g., be expressed by the two clauses $\mathsf{not}(X) \leftarrow X, !, \mathsf{fail}$ and $\mathsf{not}(X)$. For an atomic query Q, $\mathsf{not}(Q)$ can be proven if, and only if, Q fails.

(iv) SLD resolution uses unification with occur check. For efficiency, most Prolog implementations do not make use of the occur check. Except for our transformational approach from Chapter 3, all methods for termination analysis of logic programs assume unification with occur check.

In Chapter 3 we presented a new transformation from logic programs to term rewrite systems that is correct for unification without occur check and, thus, can handle (iv).

In this chapter we show how to handle (i) – (iii) by introducing a non-termination-preserving pre-processing step for logic programs with cuts based on symbolic evaluation. By handling logic programs with cuts, we can also handle logic programs with negation-as-failure (which can be expressed using a cut, cf. (iii)). In Example 5.33 we show that our pre-processing indeed works for negation-as-failure, too. By adding a cut to the end of a query, we can in principle also analyze existential instead of universal termination, but our pre-processing often is not precise enough (cf. Example 5.48).

Probably the most common use of the cut is to exploit the order of the clauses by adding a cut as the first part of the body for a certain clause. Then, all following clauses can assume that the head of this clause does not unify with the selected atom of the query. The following example demonstrates this particular kind of use of the cut and will lead us through the rest of this chapter.

Example 5.1. Consider the following logic program $\mathcal{P}$:

$$\begin{aligned}
\mathsf{div}(X,\mathsf{0},Z) &\leftarrow\ !,\mathsf{fail}. && (5)\\
\mathsf{div}(\mathsf{0},Y,Z) &\leftarrow\ !,=(Z,\mathsf{0}). && (6)\\
\mathsf{div}(X,Y,\mathsf{s}(Z)) &\leftarrow\ \mathsf{minus}(X,Y,U),\mathsf{div}(U,Y,Z). && (7)\\
=(X,X). & && (8)\\
\mathsf{minus}(\mathsf{0},Y,\mathsf{0}). & && (9)\\
\mathsf{minus}(X,\mathsf{0},X). & && (10)\\
\mathsf{minus}(\mathsf{s}(X),\mathsf{s}(Y),Z) &\leftarrow\ \mathsf{minus}(X,Y,Z). && (11)
\end{aligned}$$

and the set of queries $\mathcal{Q} = \{\mathsf{div}(t_1,t_2,t_3) \mid t_1,t_2 \text{ are ground}\}$. Any termination analyzer that ignores the cut must fail on this example as $\mathsf{div}(\mathsf{0},\mathsf{0},Z)$ leads to the subtraction of $\mathsf{0}$ using the third div-rule and, thus, starts an infinite derivation.

The goal of this chapter is to handle especially these kinds of examples as well as negation-as-failure. But as the cut can be used virtually everywhere, we also have to deal with the less intuitive behavior exhibited by logic programs with cuts. In fact, even when one is using the cut in the spirit of Example 5.1, the behavior differs significantly from one's intuition on logic programming.

Example 5.2. Obviously, the linear query $\mathsf{p}(X,Y)$ might not terminate while the non-linear query $\mathsf{p}(X,X)$ terminates. Consider for example the logic program consisting of the single clause $\mathsf{p}(\mathsf{0},\mathsf{1}) \leftarrow \mathsf{p}(\mathsf{0},\mathsf{1})$.

For definite logic programs, the linear query always allows more derivations. For logic programs with cut this need not be the case. Consider for example the following logic program, which terminates for the linear case, but not for the non-linear case:

$$\begin{aligned}
\mathsf{p}(\mathsf{0},\mathsf{1}) &\leftarrow\ !.\\
\mathsf{p}(\mathsf{0},\mathsf{0}) &\leftarrow\ \mathsf{p}(\mathsf{0},\mathsf{0}).
\end{aligned}$$

These effects can become significant when $\mathsf{p}/2$ is used by another clause. For example, given the clause $\mathsf{q} \leftarrow \mathsf{p}(X,Y)$, the query q terminates. This is trivial to show using the pre-processing technique introduced in this chapter.

Structure of the Chapter

In the remainder of this chapter, our goal is to prove universal termination of logic programs with cuts for a (typically infinite) set of queries $\mathcal{Q}$. As in the preceding chapters, these sets are typically given by the user by providing a predicate and by specifying which of its arguments are instantiated by ground terms.

In Section 5.1 we introduce some required notation and explain differences to the concept of logic programming used in Chapters 3 and 4. Then we present a set of simple inference rules that characterize the behavior of logic programming with cut for concrete queries in Section 5.2. We proceed to show how these rules can be extended to handle classes of queries represented by abstract queries in Section 5.3. Using these rules for abstract queries we can automatically build so-called termination graphs that are the basis for our pre-processing step in Section 5.4. How to generate a new, cut-free logic program from such a graph automatically is the topic of Section 5.5. Here, we will show that termination of the cut-free logic program implies termination of the original one. We summarize the contributions of this chapter in Section 5.6.

5.1. Preliminaries

As we intend to handle meta-programming, too, we do not distinguish between predicate symbols and function symbols in this chapter. However, we distinguish between individual cuts to make their scope explicit. Thus, there is only one signature Σ containing all "predicate" and "function" symbols as well as the cut operator $!/0$ and labeled versions $!_m/0$ for $m \in \mathbb{N}$. Instead of atoms and terms we will just consider terms from $\mathcal{T}(\Sigma, \mathcal{V})$.

To be able to represent sets of queries, we introduce *abstract terms*, i.e., terms containing two kinds of variables. The set $\mathcal{A}$ is the set of all *abstract variables* where each variable represents a fixed but arbitrary term. The variables corresponding to variables in logic programming are from the set $\mathcal{N}$. Thus, as abstract terms we consider all terms from the set $\mathcal{T}(\Sigma, \mathcal{V})$ where $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$. *Concrete terms* are terms from $\mathcal{T}(\Sigma, \mathcal{N})$, i.e., terms containing no abstract variables. Throughout the chapter, we often use the notation $\mathcal{N}(t)$ and $\mathcal{A}(t)$ to denote the set of all non-abstract and abstract variables occurring in a term t as defined in Definition 2.2, respectively. Likewise, in many cases it is necessary to consider restrictions of substitutions. The restriction of σ to a set of variables $\mathcal{V}' \subseteq \mathcal{V}$ (denoted $\sigma|_{\mathcal{V}'}$) is defined as $\sigma|_{\mathcal{V}'}(X) = \sigma(X)$, if $X \in \mathcal{V}'$, and $\sigma|_{\mathcal{V}'}(X) = X$, otherwise.

A clause of a logic program with cuts is a clause $H \leftarrow B$ where the head H is a term over Σ and $\mathcal{V}$ and the body B is a list of terms over Σ and $\mathcal{V}$. We call such lists *goals over* Σ *and* $\mathcal{V}$. The set of all goals over Σ and $\mathcal{V}$ is $\mathit{Goal}(\Sigma, \mathcal{V}) = \mathcal{T}(\Sigma, \mathcal{V})^*$. We denote the empty goal by $\square$ and the concatenation of two terms t and t' by t, t'. The concatenation of any term t with $\square$ (i.e., $t, \square$) is again just t. Furthermore, for a logic program $\mathcal{P} = \{c_1, \ldots, c_k\}$, $\mathit{Slice}(\mathcal{P}, t)$ denotes the set of all clauses for the root of t, i.e., $\mathit{Slice}(\mathcal{P}, p(t_1, \ldots, t_n)) = \{c_i \mid c_i = p(s_1, \ldots, s_n) \leftarrow B_i \in \mathcal{P}\}$ and $\mathit{Slice}(\mathcal{P}, X) = \varnothing$ for $X \in \mathcal{N}$.

Finally, to denote the term resulting from replacing all occurrences of a function symbol f in a term t by another function symbol g, we introduce the notation $t[f/g]$.

5.2. Concrete Derivations

In Example 5.1 we have seen that the introduction of the cut into logic programming requires a more detailed analysis of the backtracking behavior of these programs. Instead of representing the current state of the computation by just a goal and a stack of backtrack information, we choose a more explicit representation where backtrack information is given by lists of goals which are optionally labeled by the clauses that may be applied to these goals next. This, together with explicit marks for the scope of a cut, will allow us to express the non-local effect of the cut by a local rule.

The main idea is to label each cut with a fresh natural number when it is introduced by a step in the derivation. By additionally inserting such a number into the backtracking list, we can determine the scope of the correspondingly labeled cut.

More precisely, our states are lists of three different types of elements:

- The list may contain a goal $q \in \mathit{Goal}(\Sigma, \mathcal{V})$ which just represents itself.
- A labeled goal $q_m^i \in \mathit{Goal}(\Sigma, \mathcal{V}) \times \mathbb{N} \times \mathbb{N}$ represents that we must apply the i-th clause to the goal q. The m determines how a cut introduced by the body of the i-th clause will be labeled.
- A natural number $m \in \mathbb{N}$ in our backtracking lists marks that, when a cut labeled by m is reached, all elements preceding m are discarded. We denote m as $?_m$ in our backtracking lists.

The following example demonstrates the intended use of these states.

Example 5.3. Consider again the logic program for div from Example 5.1 and the query $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z)$. This would be represented by the concrete state consisting of just the goal $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z)$. As this atom unifies with the head of all three clauses for div, we obtain the identical behavior from the state $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^5 \mid \mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^6 \mid \mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^7 \mid ?_1$. This denotes that we first try to apply clause (5) and then backtrack using first clause (6) and finally clause (7). Now, we can evaluate the first labeled goal using (5) and obtain $!_1, \mathsf{fail} \mid \mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^6 \mid \mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^7 \mid ?_1$. By applying the cut we get rid of the backtracking goals $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^6$ and $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z)_1^7$ and obtain the state $\mathsf{fail} \mid ?_1$, which eventually fails. Note that due to the cut, we did not have to backtrack using the other div clauses.

In contrast, consider the state $\mathsf{minus}(0,0,Z)$. Here, the first two clauses for minus are applicable and our state becomes $\mathsf{minus}(0,0,Z)_1^9 \mid \mathsf{minus}(0,0,Z)_1^{10} \mid ?_1$. By using (9) we obtain $\square \mid \mathsf{minus}(0,0,Z)_1^{10} \mid ?_1$. Now, as we consider universal termination, we need to backtrack by removing the first element of our backtrack list and get $\mathsf{minus}(0,0,Z)_1^{10} \mid ?_1$ which we evaluate to $\square \mid ?_1$ using (10). Further backtracking leads to the empty word ε where, finally, the computation stops.

The following definition formalizes the representation of such a concrete state.

Definition 5.4 (Concrete State)**.** *The set of* concrete states $\mathit{State}(\Sigma, \mathcal{V})$ *is the set of all finite words over* $\mathit{Goal}(\Sigma,\mathcal{V}) \cup (\mathit{Goal}(\Sigma,\mathcal{V}) \times \mathbb{N} \times \mathbb{N}) \cup \mathbb{N}$*.*

Example 5.5. Let $\cdot$ denote the composition of our backtracking lists. Now, consider again some of the states from Example 5.3. A state consisting of just a goal (for instance $\mathsf{div}(0,0,Z)$) is represented by itself. The state $\mathsf{div}(0,0,Z)_1^5 \mid \mathsf{div}(0,0,Z)_1^6 \mid \mathsf{div}(0,0,Z)_1^7 \mid ?_1$, where we explicitly list all alternative clauses that might be applied to $\mathsf{div}(0,0,Z)$, is represented as $(\mathsf{div}(0,0,Z),5,1) \cdot (\mathsf{div}(0,0,Z),6,1) \cdot (\mathsf{div}(0,0,Z),7,1) \cdot 1$. Finally, the state $!_1, \mathsf{fail} \mid \mathsf{div}(0,0,Z)_1^6 \mid \mathsf{div}(0,0,Z)_1^7 \mid ?_1$ is represented by $!_1, \mathsf{fail} \cdot (\mathsf{div}(0,0,Z),6,1) \cdot (\mathsf{div}(0,0,Z),7,1) \cdot 1$.

With the help of this representation we can express derivations in logic programming with cut by eight simple inference rules. For readability we use the intuitive notation.

Definition 5.6 (Concrete Inference Rules)**.**

$$\frac{\square \mid S}{S}\ (\textsc{Success}) \qquad\qquad \frac{?_m \mid S}{S}\ (\textsc{Failure})$$

$$\frac{!_m, q \mid S \mid ?_m \mid S'}{q \mid ?_m \mid S'}\ (\textsc{Cut})\ \ \textit{where } S \textit{ contains no } ?_m \qquad \frac{!_m, q \mid S}{q}\ (\textsc{Cut})\ \ \textit{where } S \textit{ contains no } ?_m \qquad \frac{!, q \mid S}{q \mid S}\ (\textsc{Cut})$$

$$\frac{t, q \mid S}{(t,q)_m^{i_1} \mid \ldots \mid (t,q)_m^{i_k} \mid ?_m \mid S}\ (\textsc{Case})\quad \textit{where } m \textit{ is fresh, } i_1 < \ldots < i_k, \textit{ and } \mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$$

$$\frac{(t,q)_m^i \mid S}{B_i'\sigma, q\sigma \mid S}\ (\textsc{Eval})\quad \textit{where } c_i = H_i \leftarrow B_i,\ \mathit{mgu}(t, H_i) = \sigma, \textit{ and } B_i' = B_i[!/!_m].$$

$$\frac{(t,q)_m^i \mid S}{S}\ (\textsc{Backtrack})\qquad \textit{where } c_i = H_i \leftarrow B_i \textit{ and } t \not\sim H_i.$$

In the above rules we use the following conventions. First, the unlabeled term t must not be a !, i.e., $t \notin \{!\} \cup \{!_m \mid m \in \mathbb{N}\}$. Second, the list of terms q may be $\square$ and then $t, q = t, \square$ collapses to just t. Third, S and S' denote concrete states.

Note that these rules do not overlap, i.e., there is at most one rule that can be applied to any state. The only cases when no rule is applicable are when the state is the empty list (denoted ε) or when the selected "atom" is a variable. In the latter case, Prolog halts the execution with an instantiation error while our formalism is stuck.[6]

We now describe the intuition behind the individual rules.

- SUCCESS: This rule is applicable if the first goal of our backtracking list could be proved. As we handle universal termination, we have to backtrack in the case of SUCCESS of the current goal. For analyzing existential termination we could modify this rule to yield the empty word. We refrain from this as existential termination can readily be expressed in terms of universal termination with cut by adding a ! to the end of the query.

- FAILURE: While $\square$ explicitly represents the empty list of goals and, therefore, success, FAILURE is represented by the lack of further backtracking possibilities, i.e., by the question mark introduced by the corresponding CASE. If this rule is applicable, the labeled question mark is the first element of our backtracking list, and, thus, is not needed anymore.

- CASE: In order to make the backtracking possibilities explicit, the resolution of a clause with the first atom of the current goal is split into two separate operations. The CASE analysis determines which clauses c_j can potentially be applied to the first atom of the current goal by slicing the logic program according to the root symbol of the atom t. It replaces the current goal by a goal labeled with the index of the first such clause and adds copies of the current goal labeled by the indices of the other potentially applicable clauses as backtracking possibilities. Additionally, these goals are labeled by a fresh natural number, and an appropriately labeled question mark is added to the end of the list of new backtracking goals in order to denote the scope of cuts introduced at that level.

- EVAL: Then, if the first atom of our labeled goal unifies with the head of the corresponding clause, we apply the EVAL rule. This rule replaces the first atom of the current goal by the body of the corresponding rule and applies the most general unifier to the result.

- BACKTRACK: If the first atom of our labeled goal does not unify with the head of the corresponding clause, we apply the BACKTRACK rule. The reason is that in

[6]A related approach for modeling logic programming with cut has been introduced by [KB01] as a set of rules in rewriting logic. In contrast to our approach, they do not need explicit marks to denote the scope of cuts because of the so-called "suffix criterion". Unfortunately, due to approximations and generalizations, this criterion does not work for abstract queries. Furthermore, their formalism uses operations that are too fine-grained for our purposes. Even if we could build our analysis on their lower level, more extensive rule set, it would be unclear how to obtain a finite analysis.

this case, the corresponding clause cannot be used (as t and H_i do not unify). We just backtrack to the next possibility in our backtracking list.

- CUT: Finally, there are three variants of the CUT rule. The first rule removes all backtracking information on the level where it was introduced. Note that here we profit from the explicit scope represented by the labeled question mark and, thus, have turned the cut into a *local* operation. Note further that, in general, one must not delete the labeled question mark as the current goal could still contain a correspondingly labeled cut.

  The second variant is introduced to handle the splitting of states into sub lists which is needed to obtain a finite analysis in Section 5.3. The problem is that by splitting the list we might separate the question mark from the correspondingly labeled cut. If we started from a state consisting of a single goal and if we only applied our eight rules, this rule would never become applicable.

  The third variant is for ignoring cuts introduced by meta-programming. This corresponds to the behavior of typical Prolog systems. To illustrate this, consider the logic program consisting of the clause $\mathsf{p}(X) \leftarrow X, \mathsf{fail}$ and the fact $\mathsf{p}(X)$. The goal $\mathsf{p}(!)$ can be proven as the cut is ignored and, consequently, backtracking leads to the application of $\mathsf{p}(X)$.

Example 5.7. Consider again the logic program for div from Example 5.1 and the query $\mathsf{div}(0,0,Z)$ from Example 5.3. Using our inference rules we obtain the following tree.

$$\begin{tikzcd}
\boxed{\mathsf{div}(0,0,Z)} \arrow[d, "\textsc{Case}"] \\
\boxed{\mathsf{div}(0,0,Z)^5_1 \mid \mathsf{div}(0,0,Z)^6_1 \mid \mathsf{div}(0,0,Z)^7_1 \mid ?_1} \arrow[d, "\textsc{Eval}"] \\
\boxed{!_1, \mathsf{fail} \mid \mathsf{div}(0,0,Z)^6_1 \mid \mathsf{div}(0,0,Z)^7_1 \mid ?_1} \arrow[d, "\textsc{Cut}"] \\
\boxed{\mathsf{fail} \mid ?_1} \arrow[d, "\textsc{Case}"] \\
\boxed{?_2 \mid ?_1} \arrow[d, "\textsc{Failure}"] \\
\boxed{?_1} \arrow[d, "\textsc{Failure}"] \\
\boxed{\varepsilon}
\end{tikzcd}$$

Note that we do not need special treatment for fail as we can handle it just like any other undefined predicate by using the CASE rule to effectively delete the first goal.

Similar, for the query $\mathsf{minus}(0, 0, Z)$ from Example 5.3 we obtain the following derivation using the rules from Definition 5.6.

$$\begin{tikzcd}
\boxed{\mathsf{minus}(0,0,Z)} \arrow[d, "\textsc{Case}"] \\
\boxed{\mathsf{minus}(0,0,Z)_1^9 \mid \mathsf{minus}(0,0,Z)_1^{10} \mid \mathsf{minus}(0,0,Z)_1^{11} \mid ?_1} \arrow[d, "\textsc{Eval}"] \\
\boxed{\square \mid \mathsf{minus}(0,0,Z)_1^{10} \mid \mathsf{minus}(0,0,Z)_1^{11} \mid ?_1} \arrow[d, "\textsc{Success}"] \\
\boxed{\mathsf{minus}(0,0,Z)_1^{10} \mid \mathsf{minus}(0,0,Z)_1^{11} \mid ?_1} \arrow[d, "\textsc{Eval}"] \\
\boxed{\square \mid \mathsf{minus}(0,0,Z)_1^{11} \mid ?_1} \arrow[d, "\textsc{Success}"] \\
\boxed{\mathsf{minus}(0,0,Z)_1^{11} \mid ?_1} \arrow[d, "\textsc{Backtrack}"] \\
\boxed{?_1} \arrow[d, "\textsc{Failure}"] \\
\boxed{\varepsilon}
\end{tikzcd}$$

Using these eight simple rules, we can indeed characterize the derivation behavior of logic programs with cuts. This is stated by the following proposition.

Proposition 5.8 (Characterization of Logic Programming with Cut)**.** *For any (infinite) derivation starting in a cut-free goal q, there is a corresponding (infinite) derivation of the initial state consisting of just q using the inference rules from Definition 5.6.*

Finally, we define what it means for a state to be terminating.

Definition 5.9 (Termination of States)**.** *We say that a given state $S \in State(\Sigma, \mathcal{V})$ is* terminating *if, and only if, there is no infinite derivation starting from S using the inference rules from Definition 5.6.*

Note that by Proposition 5.8 termination of a state corresponding to some cut-free goal implies termination of that goal w.r.t. logic programming with cut. The same also holds for a goal q that contains cuts if we start with $q[!/!_1]$ instead of q.

5.3. Abstract Derivations

To be able to represent sets of queries, in Section 5.1 we introduced abstract terms, i.e., terms containing two kinds of variables by defining the set of variables $\mathcal{V}$ to be the disjoint union of the set $\mathcal{A}$ of all abstract variables and the set $\mathcal{N}$ of variables corresponding to variables in logic programming.

To constrain by which terms abstract variables may be instantiated, we add knowledge about instantiation and unification status. All knowledge is expressed by a knowledge base representable by a triple $KB = (\mathcal{G}, \mathcal{F}, \mathcal{U})$ where $\mathcal{G} \subseteq \mathcal{A}$, $\mathcal{F} \subseteq \mathcal{N}$, and $\mathcal{U} \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$. Here, $\mathcal{G}$ is the set of all abstract variables whose instantiations are restricted to ground terms. When unifying abstract variables with each other, shared variables from $\mathcal{N}$ (i.e., variables that occur in both terms represented by the abstract variables) may be instantiated. To distinguish shared variables from variables that occur free in our abstract terms, we keep track of the latter in $\mathcal{F}$. Finally, $\mathcal{U}$ represents a set of pairs of terms, where a pair of terms (s, t) represents that s and t are not unifiable after instantiating the abstract variables, i.e., that we have $s\gamma \not\sim t\gamma$ for a given instantiation γ of the abstract variables. We can now define an abstract state based on a concrete state with abstract variables and a knowledge base.

Definition 5.10 (Abstract State)**.** *The set of* abstract states $AState(\Sigma, \mathcal{N}, \mathcal{A})$ *is a set of pairs* $(s; KB)$ *of a concrete state* $s \in State(\Sigma, \mathcal{N} \cup \mathcal{A})$ *and a* knowledge base KB.

For a substitution γ to be a concretization of an abstract state, it needs to respect the knowledge from the knowledge base $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. First, it is only allowed to instantiate abstract variables. This can be expressed by $Dom(\gamma) \subseteq \mathcal{A}$ or, equivalently, by $\gamma|_{\mathcal{A}} = \gamma$. Second, for all abstract variables a, after we apply γ, the resulting term must not contain any abstract variables. This can be expressed as $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$. To demand that abstract variables from $\mathcal{G}$ are only replaced by ground terms, we state that $Range(\gamma)$ restricted to $\mathcal{G}$ contains no variables, i.e., $\mathcal{N}(Range(\gamma|_{\mathcal{G}})) = \varnothing$. Likewise, we need to prevent γ from introducing variables from $\mathcal{F}$ into the instantiations of abstract variables. This is expressed as $\mathcal{F}(Range(\gamma)) = \varnothing$. Finally, for all pairs $(t, t') \in \mathcal{U}$ we need to specify that $t\gamma$ and $t'\gamma$ do not unify, i.e., that $t\gamma \not\sim t'\gamma$.

Definition 5.11 (Concretization)**.** *A substitution γ is a* concretization *w.r.t. a knowledge base* $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ *if, and only if,* $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$, $\mathcal{N}(Range(\gamma|_{\mathcal{G}})) = \varnothing$, $\mathcal{F}(Range(\gamma)) = \varnothing$, *and* $\bigwedge_{(t,t') \in \mathcal{U}} t\gamma \not\sim t'\gamma$.

For an abstract state $(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$*, we define the set of* concretizations $\mathcal{C}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ *as the set* $\{S\gamma \mid \gamma$ *is a concretization w.r.t. the knowledge base* $(\mathcal{G}, \mathcal{F}, \mathcal{U})\}$.

Example 5.12. Consider the abstract state $\mathsf{minus}(T_1, T_2, T_3); (\{T_1, T_2\}, \varnothing, \{(T_1, T_3)\})$ with $T_i \in \mathcal{A}$ for all i. This represents all concrete states $\mathsf{minus}(t_1, t_2, t_3)$ where t_1, t_2 are ground terms and t_1 and t_3 do not unify, i.e., t_3 does not match t_1. For example, the concrete state $\mathsf{minus}(\mathsf{0}, \mathsf{0}, Z)$ is not represented as $\mathsf{0}$ and Z unify. In contrast, the concrete state $\mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), \mathsf{0})$ is represented and, using Clause (11), can be reduced to $\mathsf{minus}(\mathsf{0}, \mathsf{0}, \mathsf{0})$. But this clause cannot be applied to all concretizations. Consider e.g. the concrete state $\mathsf{minus}(\mathsf{0}, \mathsf{0}, \mathsf{s}(\mathsf{0}))$ represented by our abstract state, for which no clause is applicable.

As the example above demonstrates, we need to adapt our inference rules to reflect that a clause can be applied only for some instances, and to exploit the information from the knowledge base contained in the abstract state.

As we are now considering sets of concrete states represented by abstract states, the general idea of our rules is that all states represented by the parent node are terminating if all the states represented by its children are terminating.

Definition 5.13 (Sound Rules)**.** *A rule* $\rho : \mathit{AState}(\Sigma, \mathcal{N}, \mathcal{A}) \to 2^{\mathit{AState}(\Sigma,\mathcal{N},\mathcal{A})}$ *is a* sound rule *if for all abstract states* $(S; KB)$*, all* $S\gamma \in \mathcal{C}(S; KB)$ *are terminating if all states in* $\{R\gamma' \mid (R; KB') \in \rho(S; KB), R\gamma' \in \mathcal{C}(R; KB')\}$ *are terminating.*

The rules for Success, Failure, Cut, and Case do not mandate changes to the knowledge base and are, thus, straightforward to adapt to the abstract case. Note that we introduce two Case rules, one for the case that we know the root symbol of the first term of the first goal and one for the case that we do not. The latter is the case whenever we reach an abstract variable as the first term of the first goal due to meta-programming. Consider the logic program consisting of just the clause $\mathsf{p}(X) \leftarrow X$ and the abstract state $\mathsf{p}(T_1); KB$ for the knowledge base $KB = (\varnothing, \varnothing, \varnothing)$. This leads to a state $\mathsf{p}(T_1)_1^i \mid ?_1; KB$ for some i and on to $T_1 \mid ?_1; KB$. Now, we do not know the root symbol of T_1 and must consider all possibilities by branching to many nodes – one node for each $p \in \Sigma$.

Now, we define the first set of abstract rules corresponding to the first part of the five original rules from Definition 5.6. We will handle the Backtrack and Eval rules in later definitions and even introduce additional rules in order to allow for a finite analysis.

Definition 5.14 (Abstract Inference Rules – Part 1 (Success, Failure, Cut, Case))**.**

$$\frac{\square \mid S; KB}{S; KB}\ (\textsc{Success}) \qquad\qquad \frac{?_m \mid S; KB}{S; KB}\ (\textsc{Failure})$$

$$\frac{!_m, q \mid S \mid ?_m \mid S'; KB}{q \mid ?_m \mid S'; KB}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m \qquad \frac{!_m, q \mid S; KB}{q; KB}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m \qquad \frac{!, q \mid S}{q \mid S}\ (\textsc{Cut})$$

$$\frac{t, q \mid S; KB}{(t,q)_m^{i_1} \mid \ldots \mid (t,q)_m^{i_k} \mid ?_m \mid S; KB}\ (\textsc{Case}) \quad \begin{array}{l}\textit{where } m \textit{ is fresh, } t \notin \mathcal{A},\ i_1 < \ldots < i_k,\\ \textit{and } \mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}\end{array}$$

$$\frac{a, q \mid S; KB}{(a,q)_m^{i_1^{p_1}} \mid \ldots \mid (a,q)_m^{i_{k_{p_1}}^{p_1}} \mid ?_m \mid S; KB \quad \ldots \quad (a,q)_m^{i_1^{p_n}} \mid \ldots \mid (a,q)_m^{i_{k_{p_n}}^{p_n}} \mid ?_m \mid S; KB}\ (\textsc{Case})$$

where m *is fresh,* $a \in \mathcal{A}$*, and for each* $p \in \Sigma = \{p_1, \ldots, p_n\}$*,*
$i_1^p < \ldots < i_{k_p}^p$ *and* $\mathit{Slice}(\mathcal{P}, p(\vec{t})) = \{c_{i_1^p}, \ldots, c_{i_{k_p}^p}\}$

In the above rules, in addition to the notation used in Definition 5.6, we write $S \mid S'; KB$ for an abstract state $((S \mid S'); KB)$ with knowledge base KB.

Lemma 5.15 (Soundness of SUCCESS, FAILURE, CUT, and CASE). *The rules* SUCCESS*,* FAILURE*,* CUT*, and* CASE *from Definition 5.14 are sound.*

Proof. For SUCCESS we need to show that if all concretizations $S\gamma' \in \mathcal{C}(S; KB)$ are terminating, then so are all concretizations $\square \mid S\gamma \in \mathcal{C}(\square \mid S; KB)$. We proceed by contradiction, i.e., we show that if for some concretization $\square \mid S\gamma \in \mathcal{C}(\square \mid S; KB)$ there is an infinite derivation, then there is also an infinite derivation for some $S\gamma' \in \mathcal{C}(S; KB)$. Assume $\square \mid S\gamma \in \mathcal{C}(\square \mid S; KB)$ has an infinite derivation. The only rule from Definition 5.6 that is applicable is the concrete SUCCESS rule. Thus, this derivation must start with a step from $\square \mid S\gamma$ to $S\gamma$ and there is an infinite derivation starting from $S\gamma$. As γ is a concretization w.r.t. KB, we have that $S\gamma \in \mathcal{C}(S; KB)$, which concludes our proof for SUCCESS.

Likewise, for the first CUT rule we have to show that if $!_m, q\gamma \mid S\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}(!_m, q \mid S \mid ?_m \mid S'; KB)$ has an infinite derivation, $q\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}(q \mid ?_m \mid S'; KB)$ and $q\gamma \mid ?_m \mid S'\gamma$ has an infinite derivation. To this end, notice that the first step in the derivation of $!_m, q\gamma \mid S\gamma \mid ?_m \mid S'\gamma$ has to be an application of the first concrete CUT rule resulting in $q\gamma \mid ?_m \mid S'\gamma$ that, thus, has an infinite derivation, too. As KB remains unchanged, we immediately obtain $q\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}(q \mid ?_m \mid S'; KB)$. For the second CUT rule, assume $!_m, q\gamma \mid S\gamma$ has an infinite derivation and S, and, therefore, $S\gamma$ do not contain $?_m$. Then, the only applicable concrete rule is the second CUT rule. We obtain $q\gamma$ which has to have an infinite derivation and $q\gamma \in \mathcal{C}(q; KB)$. For the third CUT rule, assume $!, q\gamma \mid S\gamma$ has an infinite derivation. Then, the only applicable concrete rule is the third CUT rule. We obtain $q\gamma \mid S\gamma$, which has to have an infinite derivation and $q\gamma \mid S\gamma \in \mathcal{C}(q \mid S; KB)$.

For the first CASE rule, assume there is an infinite derivation from $t\gamma, q\gamma \mid S\gamma \in \mathcal{C}(t, q \mid S; KB)$. The only applicable concrete rule is CASE, which results in $(t\gamma, q\gamma)_m^{i_1} \mid \ldots \mid (t\gamma, q\gamma)_m^{i_k} \mid ?_m \mid S\gamma$, which starts an infinite derivation. As we do not change KB, this concrete state is an element of $\mathcal{C}((t, q)_m^{i_1} \mid \ldots \mid (t, q)_m^{i_k} \mid ?_m \mid S; KB)$. For the second CASE rule, assume there is an infinite derivation from $a\gamma, q\gamma \mid S\gamma \in \mathcal{C}(a, q \mid S; KB)$. W.l.o.g., $a\gamma = p(t_1, \ldots, t_n)$. The only applicable concrete rule is also CASE which results in $(a\gamma, q\gamma)_m^{i_1^p} \mid \ldots \mid (a\gamma, q\gamma)_m^{i_{k_p}^p} \mid ?_m \mid S\gamma$ which starts an infinite derivation. As we do not change KB, this concrete state is an element of $\mathcal{C}((a, q)_m^{i_1^p} \mid \ldots \mid (a, q)_m^{i_{k_p}^p} \mid ?_m \mid S; KB)$.

For FAILURE, assume there is an infinite derivation from $?_m \mid S\gamma \in \mathcal{C}(?_m \mid S; KB)$. The only applicable concrete rule is FAILURE, which results in $S\gamma$ which starts an infinite derivation. As we do not change KB, this concrete state is an element of $\mathcal{C}(S; KB)$. $\square$

So far, like for the concrete rules, the applicable rule can uniquely be determined by looking at the top element in the backtracking stack. Now, for the concrete EVAL and

Backtrack rule we determine which of these two rules to choose by trying to unify the first atom with the head of the corresponding clause. As demonstrated by Example 5.12, in the abstract case we might need to apply Eval in some cases and Backtrack in others. Assume we are in a state $(t,q)^i_m$ and the i-th clause is $H_i \leftarrow B_i$. Consider that the abstract variables represent arbitrary but fixed terms. Thus, whenever the most general unifier of H_i and t instantiates an abstract variable by another abstract variable or a non-variable term, this might or might not succeed.

If already the abstract term t does not unify with H_i, by variable disjointness of t and H_i we know that no concretization $t\gamma$ unifies with H_i. Likewise, if $mgu(t, H_i) = \sigma$, but σ contradicts information in $\mathcal{U}$, we know that for all concretizations $t\gamma \not\sim H_i$. In this case, we can use the backtrack rule for all concretization of our abstract state.

Definition 5.16 (Abstract Inference Rules – Part 2 (Backtrack))**.**

$$\frac{(t,q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t, H_i)\})}\ (\textsc{Backtrack})$$

where $c_i = H_i \leftarrow B_i$ and there is no substitution δ with $Dom(\delta) \subseteq \mathcal{A}$ and $\mathcal{V}(Range(\delta)) \subseteq \mathcal{N}$ such that $t\delta \sim H_i\delta$ and $\bigwedge_{(s,s') \in \mathcal{U}} (s\delta \not\sim s'\delta)$.

Lemma 5.17 (Soundness of Backtrack)**.** *The rule* Backtrack *from Definition 5.16 is sound.*

Proof. Assume there is an infinite derivation from $(t\gamma, q\gamma)^i_m \mid S\gamma \in \mathcal{C}((t,q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$. From the fact that there is no substitution δ with $Dom(\delta) \subseteq \mathcal{A}$ and $\mathcal{V}(Range(\delta)) \subseteq \mathcal{N}$, we know that there is no concretization γ' such that $t\gamma' \sim H_i$. In particular, we have that $t\gamma \not\sim H_i$ and, therefore, the only applicable concrete rule is Backtrack, which results in $S\gamma$, which starts an infinite derivation. From $t\gamma \not\sim H_i$ and $\mathcal{A}(H_i) = \varnothing$ we know $H_i\gamma = H_i$ and, therefore, $t\gamma \not\sim H_i\gamma$. Thus, γ is also a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t, H_i)\})$ and $S\gamma \in \mathcal{C}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t, H_i)\}))$. □

When the abstract Backtrack rule is not applicable, we still cannot be sure that $t\gamma$ unifies with H_i for all concretizations, as demonstrated by Example 5.12. Thus, in the abstract case we have a rule Eval with two successor states that combines both the concrete Eval and the concrete Backtrack rule.

Note that this does not invalidate the need for the abstract Backtrack rule as without it we cannot simulate that a clause is not applied due to a cut. To see this, consider the logic program consisting of the clauses $\mathsf{p} \leftarrow !$ and $\mathsf{p} \leftarrow \mathsf{p}$. Without Backtrack, there would always be a successor node resulting from Eval with the second clause.

To avoid that the most general unifier σ of t and H_i changes variables that are still used in other goals in the backtracking list, we demand that all variables in the range of

σ are variables not occurring anywhere in our backtracking list or knowledge base. We denote such *fresh variables* as belonging to the subset $\mathcal{V}_{fresh}$ of $\mathcal{V}$. Thus, the condition on σ is $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$.

When σ instantiates an abstract variable a, all variables in $a\sigma$ have to be abstract variables, too. Without the condition $\mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$ that one instantiates, for instance, the abstract variable T_1 which represents $\mathsf{s}(\mathsf{0})$ by $\mathsf{s}(X)$. But this does not correspond to correct operation of the concrete Eval rule.

Furthermore, it makes no sense to have σ introduce new abstract variables unless they are needed. Thus, we constrain the abstract variables occurring in the range of σ restricted to non-abstract variables to contain only those abstract variables that are introduced by σ restricted to abstract variables. This can be expressed by the condition $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$.

Note that all these restrictions are without loss of generality as they can be obtained by a simple variable renaming from any most general unifier of t and H_i.

Given a most general unifier that satisfies the above conditions, we can update the knowledge base accordingly. In addition to the abstract variables from $\mathcal{G}$, we also know that all (abstract) variables in the range of $a\sigma$ for all $a \in \mathcal{G}$ are instantiated to ground terms. Thus, the new set $\mathcal{G}'$ of abstract variables instantiated by ground terms is $\mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. As we can assume all variables in $H_i \leftarrow B_i$ to be from $\mathcal{V}_{fresh}$, the ones that are not in H_i cannot be used in the instantiations of abstract variables. Thus, we add $\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ to $\mathcal{F}$. Furthermore, if we replace a variable from $\mathcal{F}$ by a non-abstract variable that is not in the range of any variable not from $\mathcal{F}$ and the head H_i, we know that this variable cannot occur in the instantiation of abstract variables, either. Thus, we also add $\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}_{H_i})}))$ to $\mathcal{F}$, which yields the new set of free variables $\mathcal{F}'$. The set of non-unifying term pairs is updated differently for the successor corresponding to the application of the concrete Eval rule and for the one corresponding to the application of the concrete Backtrack rule. For the former, we can apply $\sigma|_{\mathcal{G}}$ to $\mathcal{U}$ as for this path to be taken, the ground variables must have had this shape from the beginning. For the latter, we know that for our instantiation, t and H_i do not unify. Thus, we can add this pair to $\mathcal{U}$.

There remains a problem with the instantiation of possibly shared variables. When σ replaces a non-abstract variable of t not from $\mathcal{F}$, this variable may occur in the instantiations of abstract variables not from $\mathcal{G}'$. Thus, we need to replace all these abstract variables by fresh abstract variables. This is done by the substitution $\alpha_{\mathcal{A}\setminus\mathcal{G}'}$ as defined in the definition below. Even worse, when σ replaces an abstract variable not from $\mathcal{G}$, this variable can contain any shared variable. Thus, we need to replace all abstract variables not from $\mathcal{G}'$ as well as all non-abstract variables not from $\mathcal{F}'$. This is done by $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$. The decision which approximation to use is performed by the *Approx* function.

The state for the successor corresponding to the application of the concrete Eval rule is updated by replacing t by the correspondingly instantiated body of the i-th clause. Note that we replace all cuts by a cut labeled with the scope label m. Furthermore, like for $\mathcal{U}$, we can apply σ restricted to $\mathcal{G}$ to the rest of the backtracking list. For the successor corresponding to the application of the concrete Backtrack rule, we just remove the current goal from the list.

Definition 5.18 (Abstract Inference Rules – Part 3 (Eval))**.**

$$\frac{(t,q)^i_m \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{B'_i\sigma', q\sigma' \mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}) \qquad S;(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t,H_i)\})}\ (\textsc{Eval})$$

where $c_i = H_i \leftarrow B_i$, $mgu(t,H_i) = \sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\mathcal{G}' = \mathcal{G}\cup\mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F}\cup(\mathcal{N}(Range(\sigma|_{\mathcal{F}}))\setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))})))\cup(\mathcal{N}(B_i)\setminus\mathcal{N}(H_i))$, $\sigma' = Approx(\sigma,\mathcal{G},\mathcal{F})$, *and* $B'_i = B_i[!/!_m]$.

Approx replaces some variables by fresh abstract variables:

$$Approx(\sigma,\mathcal{G},\mathcal{F}) = \begin{cases} \sigma & \textit{if } \mathcal{A}(t)\subseteq\mathcal{G} \textit{ and } \mathcal{N}(t)\subseteq\mathcal{F} \\ \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'} & \textit{if } \mathcal{A}(t)\subseteq\mathcal{G} \textit{ and } \mathcal{N}(t)\not\subseteq\mathcal{F} \\ \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} & \textit{if } \mathcal{A}(t)\not\subseteq\mathcal{G} \end{cases}$$

Here, for $a_{fresh}\in\mathcal{A}\cap\mathcal{V}_{fresh}$, *we define* $\alpha_{\mathcal{M}}$ *for a set of variables* $\mathcal{M}$ *as follows:*

$$\alpha_{\mathcal{M}}(x) = \begin{cases} a_{fresh} & \textit{if } x\in\mathcal{M} \\ x & \textit{otherwise} \end{cases}$$

Lemma 5.19 (Soundness of Eval)**.** *The rule* Eval *from Definition 5.18 is sound.*

Proof. Assume $(t\gamma,q\gamma)^i_m \mid S\gamma \in \mathcal{C}(t,q\mid S;(\mathcal{G},\mathcal{F},\mathcal{U}))$ has an infinite derivation. There are two cases depending on whether $t\gamma$ and H_i unify.

First, if $t\gamma$ does not unify with H_i, the unique applicable concrete rule is Backtrack and we obtain $S\gamma$ which has to start an infinite derivation. From $t\gamma \not\sim H_i$, $\mathcal{V}(H_i)\subseteq\mathcal{N}$, and $Dom(\gamma)\subseteq\mathcal{A}$, we know that $H_i\gamma = H_i$ and, therefore, $t\gamma\not\sim H_i\gamma$ and γ is a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t,H_i)\})$. Thus, $S\gamma\in\mathcal{C}(S;(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t,H_i)\}))$.

Second, if $t\gamma\sim H_i$, the unique applicable concrete rule is Eval. From $H_i\gamma = H_i$ we know that $t\gamma\sim H_i\gamma$ and thus t also unifies with H_i. Let $mgu(t\gamma,H_i)=\sigma''$. Then due to $H_i\gamma = H_i$ and $mgu(t,H_i)=\sigma$ there must be a substitution σ''' such that $\gamma\sigma''=\sigma\sigma'''$. W.l.o.g., we demand that $\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}_{fresh}$.

By application of the concrete Eval rule we obtain $B'_i\sigma'', q\gamma\sigma''\mid S\gamma$ where $B'_i = B_i[!/!_m]$. We are, thus, left to show that $B'_i\sigma'', q\gamma\sigma''\mid S\gamma\in\mathcal{C}(B'_i\sigma', q\sigma'\mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}))$, i.e., that there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $B'_i\sigma'' = B'_i\sigma'\gamma'$, $q\gamma\sigma'' = q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

We perform a case analysis over $\sigma' \in \{\sigma, \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}, \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\}$.

<u>Case 1: $\sigma' = \sigma$, i.e., $\mathcal{A}(t) \subseteq \mathcal{G}$ and $\mathcal{N}(t) \subseteq \mathcal{F}$:</u>
Here, we can assume $Dom(\sigma) = \mathcal{G}(t) \cup \mathcal{F}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$ and $\gamma'(a) = \gamma(a)$ otherwise.

We first show that w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma'''' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ by showing that σ'''' is a most general unifier of $t\gamma$ and H_i. That σ'''' is most general follows from σ and σ'' being most general unifiers of t and H_i resp. $t\gamma$ and H_i. To see this consider that by the definition of σ''' as $\gamma\sigma'' = \sigma\sigma'''$ we have $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{N}}))}$. Clearly, $\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ is more general than $\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{N}}))}$ and, consequently, σ'''' is more general than σ'' which is a most general unifier of $t\gamma$ and H_i. We now show that σ'''' is still a unifier of $t\gamma$ and H_i:

$$
\begin{aligned}
t\gamma\sigma'''' &\overset{Def.\sigma''''}{=} t\gamma\sigma|_{\mathcal{N}(t\gamma)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t\gamma)}))}\\
&\overset{\mathcal{N}(Range(\gamma|_{\mathcal{V}(t)}))=\varnothing}{=} t\gamma\sigma|_{\mathcal{N}(t)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t)}))}\\
&\overset{\gamma|_{\mathcal{A}(t)}=(\gamma\sigma'')|_{\mathcal{A}(t)}=\sigma|_{\mathcal{A}(t)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t)}))}}{=} t\sigma|_{\mathcal{A}(t)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t)}))}\sigma|_{\mathcal{N}(t)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t)}))}\\
&\overset{\mathcal{N}(Range(\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t)}))}))=\varnothing}{=} t\sigma|_{\mathcal{A}(t)}\sigma|_{\mathcal{N}(t)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{V}(t)}))}\\
&\overset{\mathcal{V}=\mathcal{A}\uplus\mathcal{N}}{=} t\sigma\sigma'''|_{\mathcal{A}(Range(\sigma))}\\
&\overset{\sigma=mgu(t,H_i)}{=} H_i\sigma\sigma'''|_{\mathcal{A}(Range(\sigma))}\\
&\overset{\mathcal{V}=\mathcal{A}\uplus\mathcal{N}}{=} H_i\sigma|_{\mathcal{A}(H_i)}\sigma|_{\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{V}(H_i)}))}\\
&\overset{\mathcal{A}(H_i)=\varnothing}{=} H_i\sigma|_{\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(H_i)}))}\\
&\overset{Def.\sigma''''}{=} H_i\sigma''''
\end{aligned}
$$

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma') = \varnothing$, $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus (\mathcal{A} \setminus \mathcal{A}(Range(\sigma)))$. For $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} a\gamma\sigma'' \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

To show that $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$, we make a case analysis over $a \in \mathcal{G}' = \mathcal{G} \uplus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. For $a \in \mathcal{G}$ we know that $\mathcal{N}(a\gamma) = \varnothing$ and by $\gamma'|_{\mathcal{G}} \overset{\mathcal{A}(Range(\sigma))\subseteq\mathcal{V}_{fresh}\wedge Def.\gamma'}{=} \gamma|_{\mathcal{G}}$ we obtain $\mathcal{N}(a\gamma') = \varnothing$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $\mathcal{N}(a\gamma') = \mathcal{N}(a\sigma''')$. For all $a' \in Dom(\sigma|_{\mathcal{G}})$, $\mathcal{N}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{N}(a'\gamma\sigma'') \overset{a'\in\mathcal{G}}{=} \varnothing$. Thus, $\mathcal{N}(a\gamma') = \varnothing$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus \mathcal{A}(Range(\sigma))) \uplus \mathcal{A}(Range(\sigma))$. For $a \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}' \setminus \mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_\mathcal{A}))$ there must be an $a' \in Dom(\sigma|_\mathcal{A})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'') \overset{a' \in G}{=} \varnothing$.

Finally, we have $\bigwedge_{(t,t') \in \mathcal{U}\sigma|_\mathcal{G}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s') \in \mathcal{U}} s\sigma|_\mathcal{G}\gamma' \not\sim s'\sigma|_\mathcal{G}\gamma' = \bigwedge_{(s,s') \in \mathcal{U}} s\gamma \not\sim s'\gamma$ as $a\sigma|_\mathcal{G}\gamma' = a\gamma$ for all abstract variables in $a \in \mathcal{A}(\mathcal{U})$ by definition of γ'. To see this, consider the partition $\mathcal{A}(\mathcal{U}) = (\mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_\mathcal{G})) \uplus (\mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_\mathcal{G}))$. If $a \in \mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_\mathcal{G})$ we have $a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a \notin Dom(\sigma|_\mathcal{G})}{=} a\sigma|_\mathcal{G}\gamma'$. If $a \in \mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_\mathcal{G})$ we have $a\gamma \overset{a \in \mathcal{G}}{=} a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{a \in \mathcal{G}}{=} a\sigma|_\mathcal{G}\sigma''' \overset{Def.\gamma' \wedge \mathcal{V}(Range(\sigma|_\mathcal{A})) \subseteq \mathcal{A}}{=} a\sigma|_\mathcal{G}\gamma'$.

Now, we are left to show that $B_i'\sigma'' = B_i'\sigma'\gamma'$, $q\gamma\sigma'' = q\sigma'\gamma'$, and $S\gamma = S\sigma|_\mathcal{G}\gamma'$.

For S there are two cases according to the partition $\mathcal{A}(S) = (\mathcal{A}(S) \setminus Dom(\sigma|_\mathcal{G})) \uplus (\mathcal{A}(S) \cap Dom(\sigma|_\mathcal{G}))$. Analogous to the analysis for $\mathcal{A}(\mathcal{U})$ above, we have $a\gamma = a\sigma|_\mathcal{G}\gamma'$ for both cases. With $\gamma|_\mathcal{A} = \gamma$ and $\gamma'|_\mathcal{A} = \gamma'$ we obtain $S\gamma = S\sigma|_\mathcal{G}\gamma'$.

For B_i' we analyze two cases according to the partition $\mathcal{V}(B_i) \subseteq (\mathcal{N}(B_i) \cap \mathcal{N}(H_i)) \uplus (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i))$. For $x \in \mathcal{N}(B_i) \cap \mathcal{N}(H_i)$ note that $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$ implies $x \in Dom(\sigma)$. Then, we have $x\sigma'' \overset{\sigma'' = \sigma|_\mathcal{N}\sigma'''|_{\mathcal{A}(Range(\sigma|_\mathcal{N}))}}{=} x\sigma|_\mathcal{N}\sigma'''|_{\mathcal{A}(Range(\sigma|_\mathcal{N}))} \overset{Def.\gamma'}{=} x\sigma|_\mathcal{N}\gamma' \overset{x \in \mathcal{N}}{=} x\sigma\gamma' \overset{\sigma = \sigma'}{=} x\sigma'\gamma'$. For $x \in \mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ note that σ' and σ'' are most general unifiers and, thus, w.l.o.g. do not instantiate the fresh variable x. Therefore, $x\sigma'' \overset{x \notin Dom(\sigma'')}{=} x \overset{x \notin Dom(\gamma')}{=} x\gamma' \overset{x \notin Dom(\sigma')}{=} x\sigma'\gamma'$. Thus, $B_i'\sigma'' = B_i'\sigma'\gamma'$.

Now, for q consider the partition $\mathcal{V}(q) = (\mathcal{A}(q) \setminus Dom(\sigma')) \uplus (\mathcal{A}(q) \cap Dom(\sigma')) \uplus (\mathcal{N}(q) \setminus \mathcal{F}) \uplus (\mathcal{F}(q) \setminus Dom(\sigma')) \uplus (\mathcal{F}(q) \cap Dom(\sigma'))$ which leads to the following sub cases:

- $a \in \mathcal{A}(q) \setminus Dom(\sigma')$:
  From $\mathcal{V}(t) = \mathcal{G}(t) \uplus \mathcal{F}(t)$ we get $\mathcal{V}(t\gamma) = \mathcal{N}(\mathcal{F}(t)\gamma) \cup \mathcal{N}(\mathcal{G}(t)\gamma) = \mathcal{F}(t)$. Together with $\mathcal{F}(a\gamma) = \varnothing$ we obtain $\mathcal{V}(a\gamma) \cap \mathcal{N}(t\gamma) = \mathcal{N}(a\gamma) \cap \mathcal{F}(t) \overset{\mathcal{F}(a\gamma) = \varnothing}{=} \varnothing$ and with $\mathcal{V}(a\gamma) \cap \mathcal{N}(H_i) = \varnothing$ also $\mathcal{V}(a\gamma) \cap Dom(\sigma'') = \varnothing$. Thus, we have $a\gamma\sigma'' \overset{\mathcal{N}(a\gamma) \cap Dom(\sigma'') = \varnothing}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a \notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $a \in \mathcal{A}(q) \cap Dom(\sigma')$:
  Note that $a \in Dom(\sigma')$ implies $a \in Dom(\sigma)$. We immediately have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma' \wedge \mathcal{V}(Range(\sigma|_\mathcal{A})) \subseteq \mathcal{A}}{=} a\sigma\gamma' \overset{\sigma = \sigma'}{=} a\sigma'\gamma'$.

- $x \in \mathcal{N}(q) \setminus \mathcal{F}$:
  From $x \notin \mathcal{F}(t) = \mathcal{N}(t\gamma) = \mathcal{N}(t)$ we know that $x \notin Dom(\sigma'')$ and $x \notin Dom(\sigma')$. From $\gamma|_\mathcal{A} = \gamma$ and $\gamma'|_\mathcal{A} = \gamma'$ and $x \notin \mathcal{A}$ we get $x \notin Dom(\gamma)$ and $x \notin Dom(\gamma')$. Thus, we have $x\gamma\sigma'' \overset{x \notin Dom(\gamma)}{=} x\sigma'' \overset{x \notin Dom(\sigma'')}{=} x \overset{x \notin Dom(\gamma')}{=} x\gamma' \overset{x \notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(q) \setminus Dom(\sigma')$:

From $x \notin Dom(\sigma')$ we know $x \notin \mathcal{F}(t) = \mathcal{N}(t\gamma)$ and, consequently, $x \notin Dom(\sigma'')$. With $x \notin \mathcal{A}$ we have $x\gamma\sigma'' \overset{x\notin Dom(\gamma)}{=} x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(q) \cap Dom(\sigma')$:
  Note that $x \in Dom(\sigma')$ and $\sigma' = \sigma$ imply $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{x\notin\mathcal{A}}{=} x\sigma'' \overset{\sigma''=\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}}{=} x\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))} \overset{Def.\gamma'}{=} x\sigma|_{\mathcal{N}}\gamma' \overset{x\in\mathcal{N}\wedge\sigma=\sigma'}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(q)$ and, consequently, $q\gamma\sigma'' = q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma$.

Case 2: $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$, i.e., $\mathcal{A}(t) \subseteq \mathcal{G}$ and $\mathcal{N}(t) \nsubseteq \mathcal{F}$:

Here, we can assume $Dom(\sigma) = \mathcal{G}(t) \cup \mathcal{N}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in \mathcal{A} \setminus \mathcal{G}'$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{\mathcal{A}\setminus\mathcal{G}'}$ are fresh.

First, w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ by the identical argument as for the case of $\sigma' = \sigma$ where we made use of $\mathcal{A}(t) \subseteq \mathcal{G}$, which still holds for this case.

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})) \uplus (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))))$. For $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} a\gamma\sigma'' \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ there is an $a' \notin \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ such that $a'\alpha_{\mathcal{A}\setminus\mathcal{G}'} = a$ and $\mathcal{A}(a\gamma') \overset{a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}=a}{=} \mathcal{A}(a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} a'\gamma\sigma'' \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a'\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

By the identical argument as for the case of $\sigma' = \sigma$, we obtain $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})))) \uplus \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}' \setminus \mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{A}})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'') \overset{a'\in G}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ there must be an $a' \in \mathcal{A} \setminus \mathcal{G}'$ such that $a = a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}$. Now assume $x \in F'(a\gamma')$. Then $x \in \mathcal{F}'(a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. Now, for $x \in \mathcal{F}'(a'\gamma\sigma'')$ there would have to be a $z \in \mathcal{N}(a'\gamma)$ such that $x \in \mathcal{N}(z\sigma'')$. As all variables in the range of σ'' and in $\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ are fresh, $x \notin \mathcal{F} \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)))$. From $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ we get $z \in Dom(\sigma)$. As $z \in \mathcal{N}(a'\gamma)$

and $\mathcal{F}(a'\gamma) = \varnothing$, we know $z \notin \mathcal{F}$. As all variables in $\mathcal{N}(H_i)$ are fresh, $z \notin \mathcal{N}(H_i)$. Thus, $z \in \mathcal{N} \setminus (\mathcal{N}(H_i) \cup \mathcal{F})$ and, consequently, $x \in \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{N}(H_i)\cup\mathcal{F})}))$. But as $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)) \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{N}(H_i)\cup\mathcal{F})})))$, this contradicts our assumption that $x \in \mathcal{F}'(a'\gamma\sigma'') = \mathcal{F}'(a\gamma')$.

Finally, we have $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ by the identical argument as the one used in the case of $\sigma' = \sigma$.

Now, we are left to show that $B_i'\sigma'' = B_i'\sigma'\gamma'$, $q\gamma\sigma'' = q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For S, we can use the identical argument as for the case of $\sigma' = \sigma$.

For B_i' we analyze two cases according to the partition $\mathcal{V}(B_i) = (\mathcal{N}(B_i) \cap \mathcal{N}(H_i)) \uplus (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i))$. For $x \in \mathcal{N}(B_i) \cap \mathcal{N}(H_i)$ note that $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$ implies $x \in Dom(\sigma)$. Then, we have $x\sigma'' \overset{\sigma''=\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}}{=} x\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))} \overset{Def.\gamma' \wedge \sigma\sigma=\sigma}{=} x\sigma|_{\mathcal{N}}\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{x\in\mathcal{N}}{=} x\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}=\sigma'}{=} x\sigma'\gamma'$. For $x \in \mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ note that σ' and σ'' are most general unifiers and, thus, w.l.o.g. do not instantiate the fresh variable x. Therefore, $x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$. Thus, $B_i'\sigma'' = B_i'\sigma'\gamma'$.

Now, for q consider the partition $\mathcal{V}(q) = (\mathcal{A}(q) \setminus \mathcal{G}) \uplus (\mathcal{G}(q) \cap Dom(\sigma')) \uplus (\mathcal{G}(q) \setminus Dom(\sigma')) \uplus (\mathcal{N}(q) \setminus Dom(\sigma')) \uplus (\mathcal{N}(q) \cap Dom(\sigma'))$ which leads to the following subcases:

- $a \in \mathcal{A}(q) \setminus \mathcal{G}$:
  We immediately have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma' \wedge \sigma\sigma=\sigma}{=} a\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(q) \cap Dom(\sigma')$:
  Note that $a \in \mathcal{G}(q) \cap Dom(\sigma')$ implies $a \in Dom(\sigma)$. Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'}{=} a\sigma\gamma' \overset{\mathcal{A}(a\sigma)\subseteq\mathcal{G}'}{=} a\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(q) \setminus Dom(\sigma')$:
  We have $a\gamma\sigma'' \overset{a\in\mathcal{G}}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a\notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $x \in \mathcal{N}(q) \setminus Dom(\sigma')$:
  From $x \notin Dom(\sigma')$ we know $x \notin \mathcal{V}(t)$. From $\mathcal{A}(t) \subseteq \mathcal{G}$ and $\mathcal{N}(Range(\gamma|_{\mathcal{G}})) = \varnothing$ we know that $x \notin \mathcal{V}(t\gamma)$ and together with $\mathcal{N}(H_i) \subseteq \mathcal{N}_{fresh}$, we have $x \notin Dom(\sigma'')$. Then, we have $x\gamma\sigma'' \overset{x\notin Dom(\gamma)}{=} x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{N}(q) \cap Dom(\sigma')$:
  Note that $x \in \mathcal{N}(q) \cap Dom(\sigma')$ implies $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{\gamma|_{\mathcal{A}}=\gamma}{=} x\sigma'' \overset{\sigma''=\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}}{=} x\sigma|_{\mathcal{N}}\sigma'''|\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \overset{x\in\mathcal{N}}{=} x\sigma\sigma'''|\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \overset{x\in Dom(\sigma)\wedge Def.\gamma'}{=} x\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma'=\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(q)$ and, consequently, $q\gamma\sigma'' = q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$.

<u>Case 3: $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$, i.e., $\mathcal{A}(t) \not\subseteq \mathcal{G}$:</u>
Here, we can assume $Dom(\sigma) = \mathcal{A}(t) \cup \mathcal{N}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in (\mathcal{A} \setminus \mathcal{G}') \cup (\mathcal{N} \setminus \mathcal{F}')$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ are fresh.

First, we show that w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma''''|_{\mathcal{F}\cup\mathcal{N}(H_i)} = \sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}$ and $\sigma''''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))} = \sigma''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}$ by showing that σ'''' is a most general unifier of $t\gamma$ and H_i. That σ'''' is most general follows from σ and σ'' being most general unifiers of t and H_i resp. $t\gamma$ and H_i. To see this consider that by the definition of σ''' as $\gamma\sigma'' = \sigma\sigma'''$ we have $\sigma''|_{\mathcal{F}\cup\mathcal{N}(H_i)} = \sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}$. Clearly, $\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}$ is more general than $\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}$ and, consequently, σ'''' is more general than σ'' which is a most general unifier of $t\gamma$ and H_i. We now show that σ'''' is still a unifier of $t\gamma$ and H_i:

$$
\begin{aligned}
t\gamma\sigma'''' &\overset{\substack{\mathcal{V}(Range(\sigma'')\cup Range(\sigma)\cup\\ Range(\sigma'''))\subseteq\mathcal{V}_{fresh}}}{=} t\gamma\sigma''''|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma''''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\\
&\overset{\mathcal{F}(Range(\gamma))=\varnothing\wedge\gamma_{\mathcal{A}}=\gamma}{=} t\sigma''''|_{\mathcal{F}\cup\mathcal{N}(H_i)}\gamma\sigma''''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\\
&\overset{Def.\sigma''''}{=} t\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma''|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\\
&\overset{\substack{\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}=\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\\ \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}))}}}{=} t\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\\
&\qquad \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}))}\\
&\overset{Dom(\sigma)\cap\mathcal{V}(Range(\sigma'''))=\varnothing}{=} t\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\\
&\qquad \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}))}\\
&\overset{\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} t\sigma\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}))}\\
&\overset{\sigma=mgu(t,H_i)}{=} H_i\sigma\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}))}\\
&\overset{\mathcal{V}(H_i)\subseteq\mathcal{F}\cup\mathcal{N}(H_i)}{=} H_i\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}\\
&\overset{Def.\gamma''''}{=} H_i\sigma''''
\end{aligned}
$$

We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma') = \varnothing$, $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})) \uplus (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))).$ For the case that $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} a\gamma\sigma'' \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$ there is an $a' \notin \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$ such that $a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} = a$ and $\mathcal{A}(a\gamma') \overset{a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=a}{=}$

$\mathcal{A}(a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} a'\gamma\sigma'' \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a'\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

By the identical argument as for the case of $\sigma' = \sigma$, we obtain $\mathcal{N}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))) \uplus \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}'\setminus\mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{A}})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. Now, for $x \in \mathcal{F}'(a'\gamma\sigma'')$ there would have to be a $z \in \mathcal{N}(a'\gamma)$ such that $x \in \mathcal{N}(z\sigma'')$. As all variables in the range of σ'' and in $\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ are fresh, $x \notin \mathcal{F} \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)))$. From $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ we get $z \in Dom(\sigma)$. As $z \in \mathcal{N}(a'\gamma)$ and $\mathcal{F}(a'\gamma) = \varnothing$, we know $z \notin \mathcal{F}$. As all variables in $\mathcal{N}(H_i)$ are fresh, $z \notin \mathcal{N}(H_i)$. Thus, $z \in \mathcal{N} \setminus (\mathcal{N}(H_i) \cup \mathcal{F})$ and, consequently, $x \in \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{N}(H_i)\cup\mathcal{F})}))$. But as $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)) \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus(\mathcal{N}(H_i)\cup\mathcal{F})})))$, this contradicts our assumption that $x \in \mathcal{F}'(a'\gamma\sigma'') = \mathcal{F}'(a\gamma')$. For $a \in \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ there must be an $a' \in \mathcal{A} \setminus \mathcal{G}'$ such that $a = a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}$. Now assume $x \in F'(a\gamma')$. Then $x \in \mathcal{F}'(a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. By the identical argument as for the case of $a \in \mathcal{A}(Range(\sigma))$ we can show that $x \notin \mathcal{F}'(a'\gamma\sigma'')$.

Finally, we have $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ by the identical argument as the one used in the case of $\sigma' = \sigma$.

Now, we are left to show that $B_i'\sigma'' = B_i'\sigma'\gamma'$, $q\gamma\sigma'' = q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For S, we can use the identical argument as for the case of $\sigma' = \sigma$.

For B_i' we analyze two cases according to the partition $\mathcal{V}(B_i) = (\mathcal{N}(B_i) \cap \mathcal{N}(H_i)) \uplus (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i))$. For the case of $x \in \mathcal{N}(B_i) \cap \mathcal{N}(H_i)$ note that $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$ implies $x \in Dom(\sigma)$. Then, we have $x\sigma'' \overset{\sigma''|_{\mathcal{F}\cup\mathcal{N}(H_i)}=\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}}{=} x\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))} \overset{Def.\gamma'\wedge\sigma\sigma=\sigma}{=} x\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{x\in\mathcal{N}(H_i)}{=} x\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=\sigma'}{=} x\sigma'\gamma'$. For $x \in \mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ note that σ' and σ'' are most general unifiers and, thus, w.l.o.g. do not instantiate the fresh variable x. Therefore, $x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$. Thus, $B_i'\sigma'' = B_i'\sigma'\gamma'$.

Now, for q consider the partition $\mathcal{V}(q) = (\mathcal{A}(q)\setminus\mathcal{G}) \uplus (\mathcal{G}(q) \cap Dom(\sigma')) \uplus (\mathcal{G}(q) \setminus Dom(\sigma')) \uplus (\mathcal{N}(q) \setminus \mathcal{F}) \uplus (\mathcal{F}(q) \setminus Dom(\sigma')) \uplus (\mathcal{F}(q) \cap Dom(\sigma'))$ which leads to the following sub cases:

- $a \in \mathcal{A}(q) \setminus \mathcal{G}$:
  From $\mathcal{A}(t) \not\subseteq \mathcal{G}$ we know that $a \in Dom(\sigma')$. Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'\wedge\sigma\sigma=\sigma}{=} a\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(q) \cap Dom(\sigma')$:

Note that $a \in \mathcal{G}(q) \cap Dom(\sigma')$ implies $a \in Dom(\sigma)$. Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'}{=} a\sigma\gamma' \overset{\mathcal{A}(a\sigma)\subseteq\mathcal{G}'}{=} a\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(q) \setminus Dom(\sigma')$:
  We have $a\gamma\sigma'' \overset{a\in\mathcal{G}}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a\notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $x \in \mathcal{N}(q) \setminus \mathcal{F}$:
  From $\mathcal{A}(t) \not\subseteq \mathcal{G}$ we know $x \in Dom(\sigma')$. Then, we have $x\gamma\sigma'' \overset{Def.\sigma'''}{=} x\sigma\sigma''' \overset{Def.\gamma'\wedge\sigma\sigma=\sigma}{=} x\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(q) \setminus Dom(\sigma')$:
  From $x \notin Dom(\sigma')$ we get $x \notin \mathcal{V}(t)$. From $x \in \mathcal{F}(q) \subseteq \mathcal{F}$ we get $x \notin \mathcal{N}(Range(\gamma))$. Together, we obtain $x \notin \mathcal{V}(t\gamma)$ and, consequently, $x \notin Dom(\sigma'')$. Then, we have $x\gamma\sigma'' \overset{x\notin Dom(\gamma)}{=} x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(q) \cap Dom(\sigma')$:
  Note that $x \in \mathcal{F}(q) \cap Dom(\sigma')$ implies $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{\gamma|_{\mathcal{A}}=\gamma}{=} x\sigma'' \overset{\sigma''|_{\mathcal{F}\cup\mathcal{N}(H_i)}=\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}}{=} x\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))} \overset{Def.\gamma'}{=} x\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{x\in\mathcal{F}}{=} x\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=\sigma'}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(q)$ and, consequently, $q\gamma\sigma'' = q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ and, consequently, our proof for the soundness of the EVAL rule. $\square$

With these adapted rules any concrete derivations can be simulated with abstract derivations using the rules from Definition 5.6. Unfortunately, even for terminating goals, in general, we obtain an infinite derivation tree. The reason is that the number of times an abstract evaluation may succeed is not limited as there is no bound on the size of the terms represented by the abstract variables. This is demonstrated by the following example.

Example 5.20. The infinite nature of the trees built by the rules of Parts 1 – 3 can be demonstrated using Clause (11). For simplicity, consider the following simpler logic program consisting of just one rule:

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{12}$$

For queries of the form $\mathsf{p}(t)$ where t is ground, the logic program clearly terminates.

Now, consider the tree built using the rules from Parts 1 – 3:

$$\begin{tikzcd}
\boxed{\mathsf{p}(T_1);(\{T_1\},\varnothing,\varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_1)_1^{12} \mid ?_1;(\{T_1\},\varnothing,\varnothing)} \arrow[d, "T_1/\mathsf{s}(T_2)"', "\textsc{Eval}"] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(T_2) \mid ?_1;(\{T_1,T_2\},\varnothing,\varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{?_1;(\{T_1\},\varnothing,\{(\mathsf{p}(T_1),\mathsf{p}(\mathsf{s}(X)))\})} \arrow[d, "\textsc{Failure}"] \\
\boxed{\mathsf{p}(T_2)_2^{12} \mid ?_2 \mid ?_1;(\{T_1,T_2\},\varnothing,\varnothing)} \arrow[d, "T_2/\mathsf{s}(T_3)"', "\textsc{Eval}"] \arrow[dr, "\textsc{Eval}"] & \boxed{\varepsilon;(\{T_1\},\varnothing,\{(\mathsf{p}(T_1),\mathsf{p}(\mathsf{s}(X)))\})} \\
\boxed{\mathsf{p}(T_3) \mid ?_2 \mid ?_1;(\{T_1,T_2,T_3\},\varnothing,\varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{?_2 \mid ?_1;(\{T_1,T_2\},\varnothing,\{(\mathsf{p}(T_2),\mathsf{p}(\mathsf{s}(X)))\})} \arrow[d, "\textsc{Failure}"] \\
\ldots & \ldots
\end{tikzcd}$$

This process can obviously be continued infinitely often.

Thus, in order to obtain a finite graph instead of an infinite tree, we need a way to refer back to previous nodes in our tree structure. In the following section we introduce operations that allow to obtain a finite graph for any start query.

5.4. Termination Graph

The main idea in avoiding infinite graphs is to introduce an INSTANCE rule for the case that our current state is an instance of a previous state. In these cases we can conclude termination of the current state from termination of that previous state, provided that this process is well founded. To this end, we show in Section 5.5 how to extract a cut-free logic program from our graphs such that termination of these logic programs implies termination of all states in the graph.

Example 5.21. Consider again the graph from Example 5.20. If we ignore the $?_1$ for the moment, the abstract state $\mathsf{p}(T_1);(\{T_1\},\varnothing,\varnothing)$ of the first node and the abstract state $\mathsf{p}(T_2);(\{T_1,T_2\},\varnothing,\varnothing)$ of the third node are very similar. Indeed, if one uses a substitution $\mu = \{T_1/T_2\}$ we see that the third state is an instance of the first state.

The basic idea of the following INSTANCE rule is that instead of showing that an abstract state is terminating, we can show that this state is an instance of another state. Let $S;(\mathcal{G},\mathcal{F},\mathcal{U})$ be our current state and $S';(\mathcal{G}',\mathcal{F}',\mathcal{U}')$ the more general state, i.e., there is a substitution μ such that $S = S'\mu$.

For a rule based on this idea to be sound, we have to ensure that all concrete states represented by $S;(\mathcal{G},\mathcal{F},\mathcal{U})$ are also concrete states of $S';(\mathcal{G}',\mathcal{F}',\mathcal{U}')$. Thus, we have to ensure that all abstract variables from $\mathcal{G}'$ are instantiated by μ to a term for which all variables are from $\mathcal{G}$, i.e., for all $a \in \mathcal{G}'$, $\mathcal{V}(a\mu) \subseteq \mathcal{G}$. We allow that μ is a variable renaming for the non-abstract variables. Further instantiation of the non-abstract variables would lead to problems similar to the ones presented in Example 5.2. For the non-abstract variables from $\mathcal{F}'$, we demand that these are mapped to non-abstract variables from $\mathcal{F}$, i.e., $\mathcal{F}'\mu \subseteq \mathcal{F}$. Furthermore, if μ introduced variables from $\mathcal{F}'\mu$, this would mean that abstract variables in S would be instantiated by terms containing a variable from $\mathcal{F}'$. Thus, we demand $\mathcal{F}'\mu(\mathit{Range}(\mu|_\mathcal{A})) = \varnothing$. Finally, we have to show that the non-unification information in $\mathcal{U}'$ is more general than the one in $\mathcal{U}$. This is obviously the case if $\mathcal{U}'\mu$ is a subset of $\mathcal{U}$.

Definition 5.22 (Abstract Inference Rules – Part 4 (Instance))**.**

$$\frac{S; (\mathcal{G},\mathcal{F},\mathcal{U})}{S'; (\mathcal{G}',\mathcal{F}',\mathcal{U}')}\ (\textsc{Instance})\quad \begin{array}{l}\textit{if there is a } \mu \textit{ such that } S = S'\mu\textit{, for all } a \in \mathcal{G}',\\ \mathcal{V}(a\mu) \subseteq \mathcal{G}\textit{, } \mu|_\mathcal{N} \textit{ is a variable renaming, } \mathcal{F}'\mu \subseteq \mathcal{F},\\ \mathcal{F}'\mu(\mathit{Range}(\mu|_\mathcal{A})) = \varnothing\textit{, and } \mathcal{U}'\mu \subseteq \mathcal{U}.\end{array}$$

Lemma 5.23 (Soundness of Instance)**.** *The rule* Instance *from Definition 5.22 is sound.*

Proof. Assume we have an infinite derivation starting from $S\gamma \in \mathcal{C}(S; (\mathcal{G},\mathcal{F},\mathcal{U}))$. We show that there is a substitution γ' such that $S'\gamma' \in \mathcal{C}(S'; (\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ and $S'\gamma'$ has an infinite derivation.

As $\mu|_\mathcal{N}$ is a variable renaming, there must be a μ^{-1} such that $\mu|_\mathcal{N}\mu^{-1} = \mu^{-1}\mu|_\mathcal{N} = id$. Let $\gamma' = \mu\gamma\mu^{-1}$. Clearly, as $S'\mu = S$ and μ^{-1} is a variable renaming, $S'\gamma' = S\gamma\mu^{-1}$ has an infinite derivation. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}')$.

For $x \in \mathcal{N}$ we have $x\mu \in \mathcal{N}$ and, thus, $x\gamma' \overset{\mathit{Def.}\gamma'}{=} x\mu\gamma\mu^{-1} \overset{x\mu\in\mathcal{N}}{=} x\mu\mu^{-1} \overset{\mathit{Def.}\mu^{-1}}{=} x$, i.e., $\gamma'|_\mathcal{A} = \gamma'$. From $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ and $\mathcal{A}(\mathit{Range}(\mu^{-1})) = \varnothing$, we also obtain $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \bigcup_{a\in\mathcal{A}} \mathcal{A}(\mu\gamma\mu^{-1}) = \varnothing$.

We know that for all $a \in \mathcal{G}'$, $\mathcal{V}(a\mu) \subseteq \mathcal{G}$. Further, as γ is a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ we know that for all $a \in \mathcal{G}$, $\mathcal{N}(a\gamma) = \varnothing$. Thus, for all $a \in \mathcal{G}'$, we have $\mathcal{N}(a\gamma') \overset{\mathit{Def.}\gamma'}{=} \mathcal{N}(a\mu\gamma\mu^{-1}) = \mathcal{N}(a\mu\gamma) = \varnothing$ and, therefore, $\mathcal{N}(\mathit{Range}(\gamma'|_{\mathcal{G}'})) = \varnothing$.

We need to show that $\mathcal{F}'(\mathit{Range}(\gamma')) = \mathcal{F}'(\mathit{Range}(\mu\gamma\mu^{-1})) = \mathcal{F}'(\mathit{Range}(\mu|_\mathcal{N}\mu|_\mathcal{A}\gamma\mu^{-1})) = \varnothing$. From $\gamma'|_\mathcal{N} = id$ this is equivalent to $\mathcal{F}'(a\mu|_\mathcal{A}\gamma\mu^{-1}) = \varnothing$ for all $a \in \mathit{Dom}(\mu|_\mathcal{A}\gamma\mu^{-1})$. Now, $\mathcal{F}'(a\mu|_\mathcal{A}\gamma\mu^{-1}) = \varnothing$ is equivalent to $\mathcal{F}'\mu|_\mathcal{N}(a\mu|_\mathcal{A}\gamma\mu^{-1}\mu|_\mathcal{N}) = \mathcal{F}'\mu|_\mathcal{N}(a\mu|_\mathcal{A}\gamma) = \varnothing$. Now, this holds as $\mathcal{F}'\mu(\mathit{Range}(\mu|_\mathcal{A})) = \varnothing$, $\mathcal{F}'\mu \subseteq \mathcal{F}$, and $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$. Thus, we can conclude $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$.

Finally, from $\bigwedge_{(s,s')\in\mathcal{U}} t\gamma \not\sim t'\gamma$ and $\mathcal{U}'\mu \subseteq \mathcal{U}$, we know that $\bigwedge_{(s,s')\in\mathcal{U}'\mu} t\gamma \not\sim t'\gamma$ which is equivalent to $\bigwedge_{(t,t')\in\mathcal{U}'} t\mu\gamma \not\sim t'\mu\gamma$. As μ^{-1} is a variable renaming, trivially $\bigwedge_{(t,t')\in\mathcal{U}'} t\mu\gamma\mu^{-1} \not\sim t'\mu\gamma\mu^{-1}$ and, consequently, $\bigwedge_{(t,t')\in\mathcal{U}'} t\gamma' \not\sim t'\gamma'$.

This concludes our proof as $\mu\gamma\mu^{-1}$ satisfies all conditions of a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}')$. □

Example 5.24. Still, this is not enough to obtain a finite analysis for Example 5.20 as we need to get rid of superfluous backtracking goals, even if they only consist of the question marks. To see this, note that in Example 5.21 we ignored the $?_1$ although it is an integral part of the abstract state considered.

To solve this problem, we introduce the PARALLEL rule that allows us to split a backtracking list into separate problems. While this rule may lose precision, it is virtually always needed for obtaining a finite graph. As the splitting of backtracking lists is incorrect in general, we will introduce the notions of active cuts and active marks to characterize positions where splitting is allowed. Here active cuts for a state S are represented by a subset of $\mathbb{N}$ containing all those m for which $!_m$ occurs in S or can be introduced by EVAL applied to a labeled goal $(t, q)^i_m$ occurring in S. Likewise, the active marks for a state S are represented by a subset of $\mathbb{N}$ containing those m for which $?_m$ occurs in S. Note that we can exclude $?_m$ occurring as the first or last element of S. The former is possible, as an application of the FAILURE rule would remove such a $!_m$ without any side effects. The latter is possible, as applying the first cut rule to a state ending in $?_m$ is identical to applying the second CUT rule to the same state without $?_m$.

Definition 5.25 (Abstract Inference Rules – Part 5 (PARALLEL))**.**

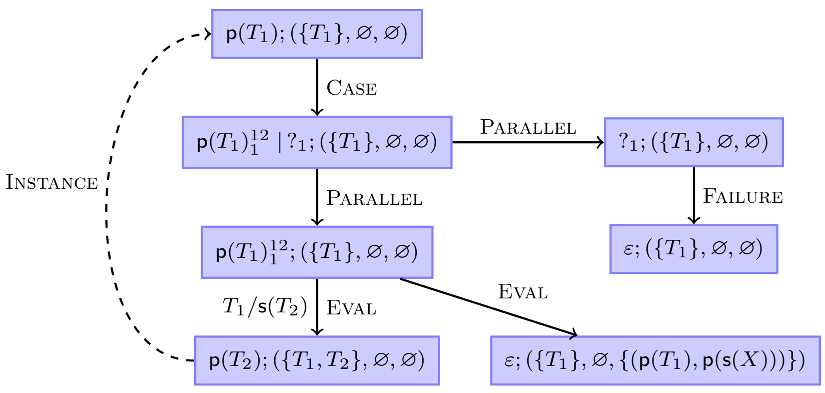
$$\frac{S \mid S'; KB}{S; KB \qquad S'; KB}\ (\textsc{Parallel}) \quad \textit{if } AC(S) \cap AM(S') = \varnothing$$

Here, the active cuts $AC(S)$ *of a state* S *are defined as the set of all* m *such that* $S = S' \mid q, !_m, q' \mid S''$ *or* $S = S' \mid (t, q)^j_m \mid S''$ *and* $c_j = H_j \leftarrow B_j, !, B'_j$*, while the* active marks $AM(S)$ *of a state* S *are defined as all* m *such that* $S = S' \mid ?_m \mid S''$ *and* $S' \neq \varepsilon \neq S''$*.*

Lemma 5.26 (Soundness of PARALLEL)**.** *The rule* PARALLEL *from Definition 5.25 is sound.*

Proof. Assume that $S\gamma \mid S'\gamma \in \mathcal{C}(S \mid S'; KB)$ has an infinite derivation. Then there are three cases. If $S\gamma$ has an infinite derivation, we immediately have that $S\gamma \in \mathcal{C}(S; KB)$ has an infinite derivation. If $S\gamma$ does not have an infinite derivation and, after finitely many steps, we reach the state $S'\gamma$, we have that $S'\gamma \in \mathcal{C}(S'; KB)$ has an infinite derivation. Finally, if $S\gamma$ has no infinite derivation, but we do not reach $S'\gamma$, S' must be of the form $S'' \mid ?_m \mid S'''$ with $S'' \neq \varepsilon$ and in the derivation of $S\gamma \mid S'\gamma$ we apply the CUT rule to $!_m, q \mid S''''\gamma \mid S''\gamma \mid ?_m \mid S'''\gamma$, i.e., $m \in AC(S)$. As $S\gamma \mid S'\gamma$ has an infinite derivation, we get $S''' \neq \varepsilon$. But $S'' \neq \varepsilon \neq S'''$ implies $m \in AM(S')$. Thus we have a contradiction to $AC(S) \cap AM(S') = \varnothing$. □

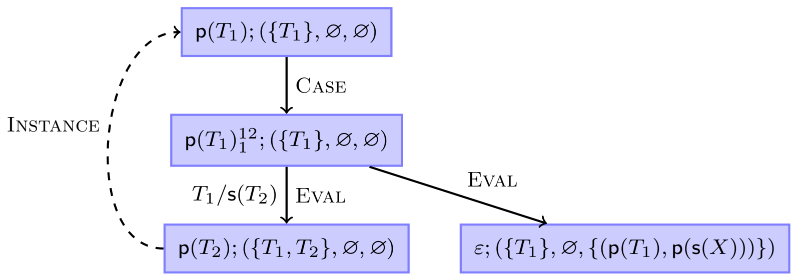
Example 5.27. Consider again the one-rule logic program from Example 5.20. Now, using the rules from Parts 1 – 5 we can obtain the following finite tree.



$\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)$
CASE
$\mathsf{p}(T_1)_1^{12} \mid ?_1; (\{T_1\}, \varnothing, \varnothing)$
PARALLEL
$?_1; (\{T_1\}, \varnothing, \varnothing)$
FAILURE
$\varepsilon; (\{T_1\}, \varnothing, \varnothing)$
INSTANCE
PARALLEL
$\mathsf{p}(T_1)_1^{12}; (\{T_1\}, \varnothing, \varnothing)$
$T_1/\mathsf{s}(T_2)$
EVAL
EVAL
$\mathsf{p}(T_2); (\{T_1, T_2\}, \varnothing, \varnothing)$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$

By using PARALLEL and FAILURE we can always get rid of question marks at the end of a state. From here on we will always implicitly use these two rules in such cases.

Example 5.28. Consider again the one-rule logic program from Example 5.20. Using implicit removal of trailing question marks, we obtain the following finite tree.



$\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)$
CASE
INSTANCE
$\mathsf{p}(T_1)_1^{12}; (\{T_1\}, \varnothing, \varnothing)$
$T_1/\mathsf{s}(T_2)$
EVAL
EVAL
$\mathsf{p}(T_2); (\{T_1, T_2\}, \varnothing, \varnothing)$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$

So far, we have used the PARALLEL rule only to remove superfluous trailing question marks. In general, backtracking lists can, of course, contain non-trivial information that needs to be removed for finiteness of the analysis.

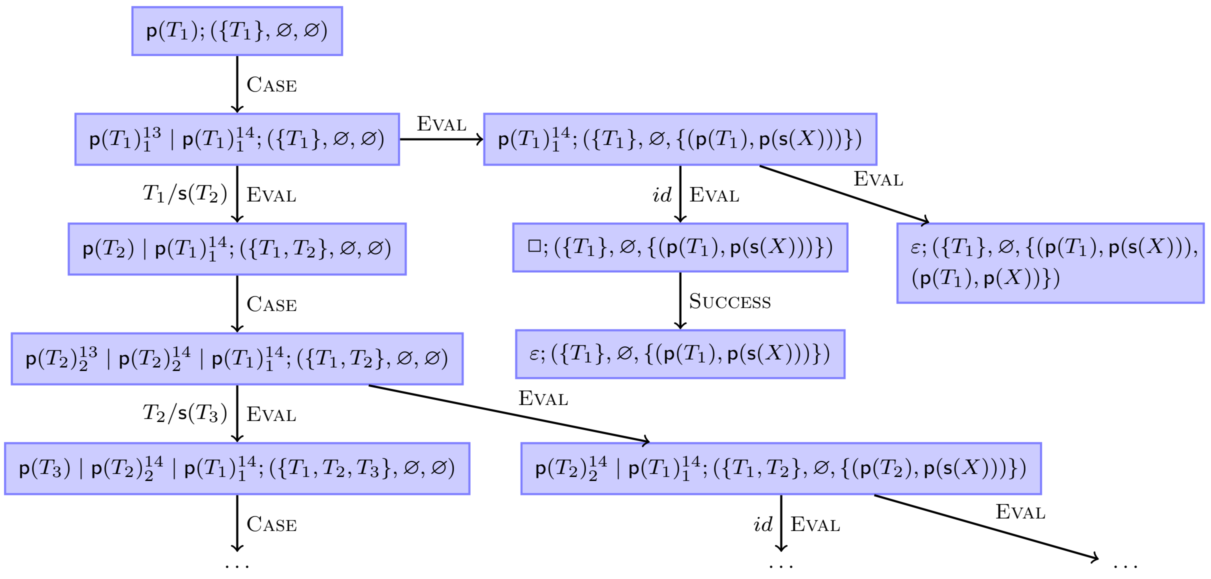
Example 5.29. Consider the following simple logic program consisting just of one rule and one fact:

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{13}$$

$$\mathsf{p}(X). \tag{14}$$

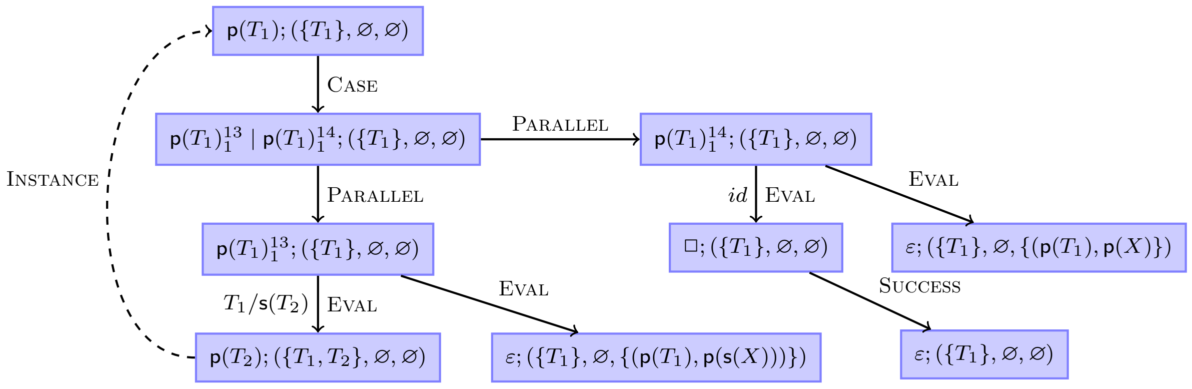
For queries of the form $\mathsf{p}(t)$ where t is ground, the logic program once again clearly terminates.

Now, consider the tree built using the rules from Parts 1 – 5 where Parallel is only used implicitly to remove trailing question marks:


$\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)$
Case
$\mathsf{p}(T_1)_1^{13} \mid \mathsf{p}(T_1)_1^{14}; (\{T_1\}, \varnothing, \varnothing)$
Eval
$\mathsf{p}(T_1)_1^{14}; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$
$T_1/\mathsf{s}(T_2)$ Eval
id Eval
Eval
$\mathsf{p}(T_2) \mid \mathsf{p}(T_1)_1^{14}; (\{T_1, T_2\}, \varnothing, \varnothing)$
$\square; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X))), (\mathsf{p}(T_1), \mathsf{p}(X))\})$
Case
Success
$\mathsf{p}(T_2)_2^{13} \mid \mathsf{p}(T_2)_2^{14} \mid \mathsf{p}(T_1)_1^{14}; (\{T_1, T_2\}, \varnothing, \varnothing)$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$
Eval
$T_2/\mathsf{s}(T_3)$ Eval
$\mathsf{p}(T_3) \mid \mathsf{p}(T_2)_2^{14} \mid \mathsf{p}(T_1)_1^{14}; (\{T_1, T_2, T_3\}, \varnothing, \varnothing)$
$\mathsf{p}(T_2)_2^{14} \mid \mathsf{p}(T_1)_1^{14}; (\{T_1, T_2\}, \varnothing, \{(\mathsf{p}(T_2), \mathsf{p}(\mathsf{s}(X)))\})$
Case
id Eval
Eval
...
...
...

This process can obviously be continued infinitely often without encountering an instance of a previous state. The reason is that each application of the Case rule produces another backtracking target.

Now, by using Parallel more liberally, we can obtain the following alternative tree:


$\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)$
Case
$\mathsf{p}(T_1)_1^{13} \mid \mathsf{p}(T_1)_1^{14}; (\{T_1\}, \varnothing, \varnothing)$
Parallel
$\mathsf{p}(T_1)_1^{14}; (\{T_1\}, \varnothing, \varnothing)$
Instance
Parallel
id Eval
Eval
$\mathsf{p}(T_1)_1^{13}; (\{T_1\}, \varnothing, \varnothing)$
$\square; (\{T_1\}, \varnothing, \varnothing)$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(X)\})$
$T_1/\mathsf{s}(T_2)$ Eval
Eval
Success
$\mathsf{p}(T_2); (\{T_1, T_2\}, \varnothing, \varnothing)$
$\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})$
$\varepsilon; (\{T_1\}, \varnothing, \varnothing)$

Thus, by using the Parallel rule in these non-trivial cases, we can easily close the tree.

Note that Instance is not only useful for closing the graph, but also for generalizing a state. This allows for example to get rid of superfluous information in the knowledge base.

Example 5.30. Consider once more the one-rule logic program from Example 5.20. Now, using another generalization we can obtain the following tree.

$$\begin{tikzcd}
\boxed{\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_1)_1^{12}; (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Eval}", "T_1/\mathsf{s}(T_2)"'] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(T_2); (\{T_1, T_2\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Instance}"] & \boxed{\varepsilon; (\{T_1\}, \varnothing, \{(\mathsf{p}(T_1), \mathsf{p}(\mathsf{s}(X)))\})} \arrow[d, "\textsc{Instance}"] \\
\boxed{\mathsf{p}(T_2); (\{T_2\}, \varnothing, \varnothing)} \arrow[uuu, dashed, bend left=60, "\textsc{Instance}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)}
\end{tikzcd}$$

From here on we will always use implicit generalizations in our examples to keep the knowledge base relevant to the current state. In particular, information about variables no longer used in the state can safely be disregarded.

Example 5.31. Consider for the last time the one-rule logic program from Example 5.20. Now, using an implicit generalization step we obtain the following tree.

$$\begin{tikzcd}
\boxed{\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_1)_1^{12}; (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Eval}", "T_1/\mathsf{s}(T_2)"'] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(T_2); (\{T_2\}, \varnothing, \varnothing)} \arrow[uu, dashed, bend left=60, "\textsc{Instance}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)}
\end{tikzcd}$$

In the examples above, we used the INSTANCE node both to refer back to an existing node and to remove superfluous knowledge by creating a new node. There are also cases where the INSTANCE rule is needed to create a new, more general node with a generalized state.

Example 5.32. Consider the following logic program consisting of just one rule:

$$\mathsf{p}(X) \quad \leftarrow \quad \mathsf{p}(\mathsf{s}(X)). \tag{15}$$

Now, consider the query $\mathsf{p}(0)$ and the corresponding tree built using the rules from Parts 1 – 5 as we have applied them so far:

$$\begin{tikzcd}
\boxed{\mathsf{p}(\mathsf{0}); (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(\mathsf{0})^{15}_1; (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Eval}", "id"'] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(\mathsf{s}(\mathsf{0})); (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} \\
\boxed{\mathsf{p}(\mathsf{s}(\mathsf{0}))^{15}_2; (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Eval}", "id"'] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{0}))); (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} \\
\dots &
\end{tikzcd}$$

This process can obviously be continued infinitely often. But if we use the Instance rule to generalize 0 to an abstract variable T_1, we obtain the following finite tree.

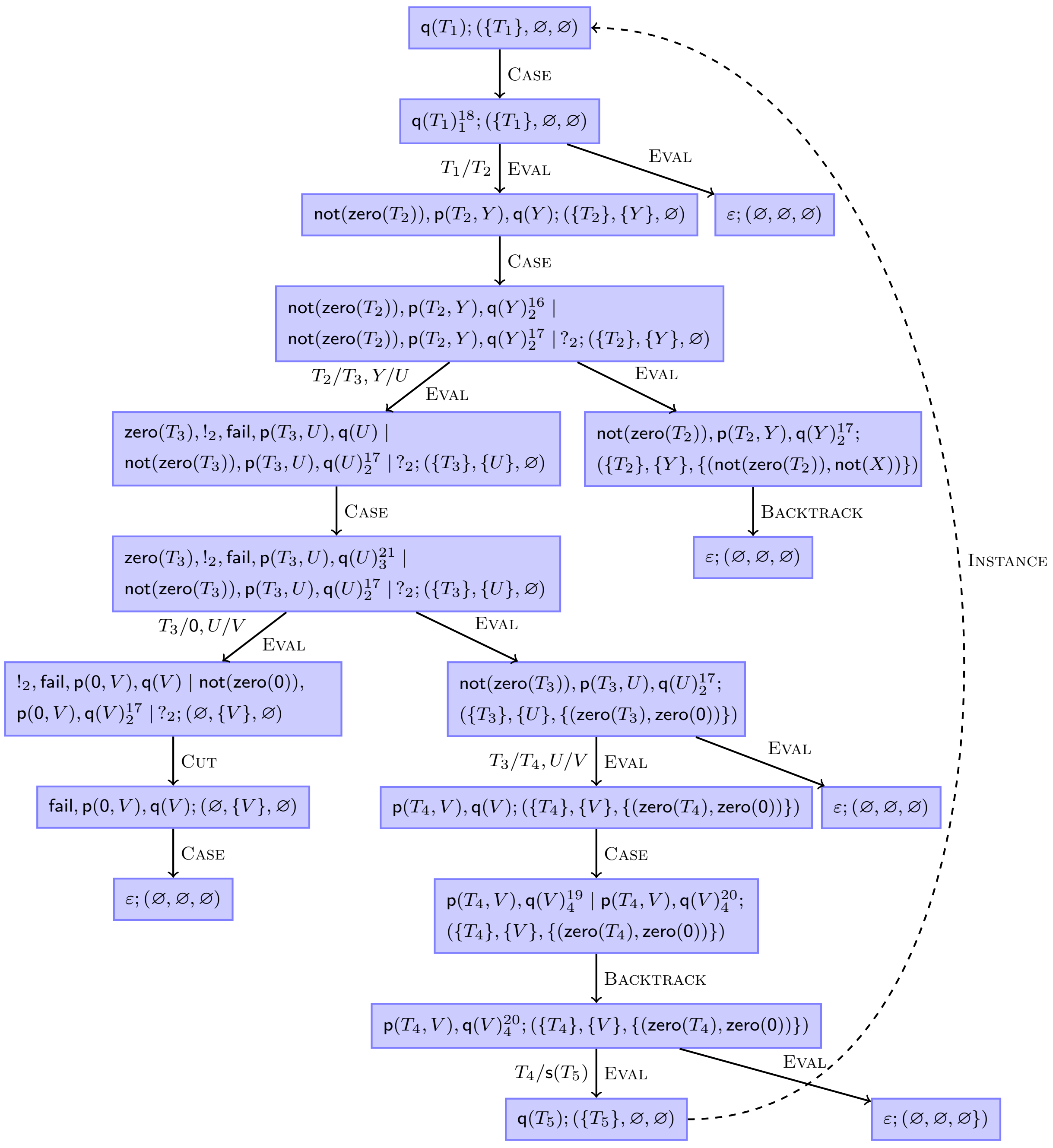
$$\begin{tikzcd}
\boxed{\mathsf{p}(\mathsf{0}); (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Instance}"] & \\
\boxed{\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_1)^{15}_1; (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Eval}", "T_1/T_2"'] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(\mathsf{s}(T_2)); (\{T_2\}, \varnothing, \varnothing)} \arrow[uu, dashed, bend left=60, "\textsc{Instance}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)}
\end{tikzcd}$$

With the rules from Parts 1 – 5 we can also handle programs using meta-programming, e.g., negation-as-failure expressed by two clauses. While we can also handle the case that we have to evaluate an abstract variable (cf. the definition of *Slice* and Case), the example below shows that in typical examples using negation-as-failure, we do not even need this feature.

Example 5.33. Consider the following logic program:

$$\begin{aligned}
\mathsf{not}(X) &\leftarrow X, !, \mathsf{fail}. && (16)\\
\mathsf{not}(X). & && (17)\\
\mathsf{q}(X) &\leftarrow \mathsf{not}(\mathsf{zero}(X)), \mathsf{p}(X,Y), \mathsf{q}(Y). && (18)\\
\mathsf{p}(\mathsf{0},\mathsf{0}). & && (19)\\
\mathsf{p}(\mathsf{s}(X),X). & && (20)\\
\mathsf{zero}(\mathsf{0}). & && (21)
\end{aligned}$$

Now, consider the class of queries $\mathsf{q}(t)$ where t is a ground term. The logic program clearly terminates for these queries as clause (19) can never be applied. The corresponding tree built using the rules from Parts 1 – 5 is:


$\mathsf{q}(T_1); (\{T_1\}, \varnothing, \varnothing)$
Case
$\mathsf{q}(T_1)_1^{18}; (\{T_1\}, \varnothing, \varnothing)$
T_1/T_2 Eval
Eval
$\mathsf{not}(\mathsf{zero}(T_2)), \mathsf{p}(T_2, Y), \mathsf{q}(Y); (\{T_2\}, \{Y\}, \varnothing)$
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
Case
$\mathsf{not}(\mathsf{zero}(T_2)), \mathsf{p}(T_2, Y), \mathsf{q}(Y)_2^{16} \mid \mathsf{not}(\mathsf{zero}(T_2)), \mathsf{p}(T_2, Y), \mathsf{q}(Y)_2^{17} \mid ?_2; (\{T_2\}, \{Y\}, \varnothing)$
$T_2/T_3, Y/U$ Eval
Eval
$\mathsf{zero}(T_3), !_2, \mathsf{fail}, \mathsf{p}(T_3, U), \mathsf{q}(U) \mid \mathsf{not}(\mathsf{zero}(T_3)), \mathsf{p}(T_3, U), \mathsf{q}(U)_2^{17} \mid ?_2; (\{T_3\}, \{U\}, \varnothing)$
$\mathsf{not}(\mathsf{zero}(T_2)), \mathsf{p}(T_2, Y), \mathsf{q}(Y)_2^{17}; (\{T_2\}, \{Y\}, \{(\mathsf{not}(\mathsf{zero}(T_2)), \mathsf{not}(X))\})$
Backtrack
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
Instance
Case
$\mathsf{zero}(T_3), !_2, \mathsf{fail}, \mathsf{p}(T_3, U), \mathsf{q}(U)_3^{21} \mid \mathsf{not}(\mathsf{zero}(T_3)), \mathsf{p}(T_3, U), \mathsf{q}(U)_2^{17} \mid ?_2; (\{T_3\}, \{U\}, \varnothing)$
$T_3/\mathsf{0}, U/V$ Eval
Eval
$!_2, \mathsf{fail}, \mathsf{p}(\mathsf{0}, V), \mathsf{q}(V) \mid \mathsf{not}(\mathsf{zero}(\mathsf{0})), \mathsf{p}(\mathsf{0}, V), \mathsf{q}(V)_2^{17} \mid ?_2; (\varnothing, \{V\}, \varnothing)$
$\mathsf{not}(\mathsf{zero}(T_3)), \mathsf{p}(T_3, U), \mathsf{q}(U)_2^{17}; (\{T_3\}, \{U\}, \{(\mathsf{zero}(T_3), \mathsf{zero}(\mathsf{0}))\})$
Cut
$\mathsf{fail}, \mathsf{p}(\mathsf{0}, V), \mathsf{q}(V); (\varnothing, \{V\}, \varnothing)$
Case
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
$T_3/T_4, U/V$ Eval
Eval
$\mathsf{p}(T_4, V), \mathsf{q}(V); (\{T_4\}, \{V\}, \{(\mathsf{zero}(T_4), \mathsf{zero}(\mathsf{0}))\})$
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
Case
$\mathsf{p}(T_4, V), \mathsf{q}(V)_4^{19} \mid \mathsf{p}(T_4, V), \mathsf{q}(V)_4^{20}; (\{T_4\}, \{V\}, \{(\mathsf{zero}(T_4), \mathsf{zero}(\mathsf{0}))\})$
Backtrack
$\mathsf{p}(T_4, V), \mathsf{q}(V)_4^{20}; (\{T_4\}, \{V\}, \{(\mathsf{zero}(T_4), \mathsf{zero}(\mathsf{0}))\})$
$T_4/\mathsf{s}(T_5)$ Eval
Eval
$\mathsf{q}(T_5); (\{T_5\}, \varnothing, \varnothing)$
$\varepsilon; (\varnothing, \varnothing, \varnothing\})$

In the next section, Example 5.43 shows that the termination problem of this example can be reduced to a trivial termination problem.

While we are able to close the graph for the simple examples above, in general we are unable to close the graph as, for some paths, we never obtain an instance of a previous node.

Example 5.34. Consider the following simple logic program consisting just of one rule and one fact:

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X), \mathsf{q}. \tag{22}$$

$$\mathsf{q}. \tag{23}$$

For queries of the form $\mathsf{p}(t)$ where t is ground, the logic program again clearly terminates. Now, consider the tree built using the rules from Parts 1 – 4:

$$\begin{tikzcd}
\boxed{\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_1)^{22}_{1}; (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "T_1/\mathsf{s}(T_2)"', "\textsc{Eval}"] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(T_2), \mathsf{q}; (\{T_2\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} \\
\boxed{\mathsf{p}(T_2)^{22}_{2}\mathsf{q}; (\{T_2\}, \varnothing, \varnothing)} \arrow[d, "T_2/\mathsf{s}(T_3)"', "\textsc{Eval}"] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\mathsf{p}(T_3), \mathsf{q}, \mathsf{q}; (\{T_3\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} \\
\dots &
\end{tikzcd}$$

This can obviously be continued infinitely often without encountering an instance of a previous state. The reason is that each application of a clause adds a q to the query.

For these cases we introduce the final abstract rule SPLIT for splitting a goal of more than one term. The basic idea is to take a state $t, q \mid S; KB$ and split it into two new states, $t \mid S; KB$ for the selected "atom" and $q\mu \mid S; KB$ for the following goal where μ represents an approximation of the answer substitutions for t. For such a rule to be correct, one would have to restrict that the active cuts for t, q have no corresponding active marks in S. Furthermore, if some instantiation of t fails, we have to backtrack to $S; KB$. Thus, we only define SPLIT for backtracking lists of one element. To obtain such a list, we can use PARALLEL which has similar restrictions on active cuts and active marks.

Thus, we refine our idea to take a state $t, q; (\mathcal{G}, \mathcal{F}, \mathcal{U})$ and split it into two successors $t; (\mathcal{G}, \mathcal{F}, \mathcal{U})$ and $q\mu; (\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$. Here, μ is an overapproximation of the answer substitutions where we assume that all free variables of t are potentially instantiated and thus replace them by a fresh abstract variable. If t contains possibly shared variables, i.e., non-abstract variables not in $\mathcal{F}$ or abstract variables not in $\mathcal{G}$, we also have to replace all such variables in q by fresh abstract variables as they might be instantiated by the answer substitution for t. When assuming that the free variables of t are instantiated, one has to remove them from the set of free variables $\mathcal{F}$.

The set of abstract variables instantiated by ground terms may grow by, for instance, a free variable being replaced with a ground term by the answer substitution for t. Here, we can use any groundness analysis that, given a predicate and a set of argument positions known to be ground, analyzes which argument positions are instantiated with ground terms by the answer substitution. Finally, we can also apply the overapproximation μ to specialize our non-unification information in $\mathcal{U}$.

Definition 5.35 (Abstract Inference Rules – Part 6 (Split)).

$$\frac{t, q; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{t; (\mathcal{G}, \mathcal{F}, \mathcal{U}) \qquad q\mu; (\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)} \ (\textsc{Split})$$

where $\mu = ApproxSub(t, \mathcal{G}, \mathcal{F})$, $\mathcal{G}' = \mathcal{G} \cup ApproxGnd(t, \mu)$, and $\mathcal{F}' = \mathcal{F} \setminus \mathcal{F}(t)$.

Here, ApproxSub approximates the substitutions of the answer sets of all concretizations w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ of t:

$$ApproxSub(t, \mathcal{G}, \mathcal{F}) = \begin{cases} \alpha_{\mathcal{F}(t)} & \textit{if } \mathcal{V}(t) \subseteq \mathcal{G} \cup \mathcal{F} \\ \alpha_{\mathcal{F}(t)} \alpha_{\mathcal{A} \setminus \mathcal{G}} \alpha_{\mathcal{N} \setminus \mathcal{F}} & \textit{otherwise} \end{cases}$$

Finally, ApproxGnd approximates the abstract variables that have to be instantiated by ground terms using a given groundness analysis $Ground_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ which given a predicate p and a set of ground argument positions computes the set of ground arguments positions after a successful computation using the clauses from $\mathcal{P}$:

$$ApproxGnd(t, \mu) = \{\mathcal{A}(t_i\mu) \mid t = p(t_1, \ldots, t_n), i \in Ground_{Slice(\mathcal{P},t)}(p, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$$

Lemma 5.36 (Soundness of Split). *The rule* Split *from Definition 5.35 is sound.*

Proof. Assume that $t\gamma, q\gamma \in \mathcal{C}(t, q; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite derivation. Then, there are two cases. If $t\gamma$ has an infinite derivation, we immediately have that $t\gamma \in \mathcal{C}(t; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite derivation. If $t\gamma$ does not have an infinite derivation and we did not reach a state of the form $q\gamma\mu' \mid S'\gamma$ for some answer substitution μ' and state S', we would reach the state ε, which contradicts our assumption that $t\gamma, q\gamma$ has an infinite derivation. Therefore, if $t\gamma$ does not have an infinite derivation, we reach states of the form $q\gamma\mu' \mid S'\gamma$ for answer substitutions μ' and states S'. If all $q\gamma\mu'$ did not have an infinite derivation, this would contradict our assumption that $t\gamma, q\gamma$ has an infinite derivation. Thus, there must be a state $q\gamma\mu'$ that has an infinite derivation. We now show that there is a concretization γ' such that $q\gamma\mu' = q\mu\gamma'$. There are two subcases. First, if $\mathcal{V}(t) \subseteq \mathcal{G} \cup \mathcal{F}$ we have $\mathcal{V}(t\gamma) \subseteq \mathcal{F}$ as γ is a concretization and, therefore, for all $a \in \mathcal{G}(t)$, $\mathcal{N}(a\gamma) = \varnothing$. Thus, we have $Dom(\mu') \subseteq \mathcal{F}(t\gamma)$. From $\mu = \alpha_{\mathcal{F}(t)}$ we know that for all $x \in \mathcal{F}(t\gamma) = \mathcal{F}(t)$, $x\mu \in \mathcal{A}$ is a fresh variable. We define $\gamma'(x\mu) = x\mu'$ for $x \in \mathcal{F}(t)$ and $\gamma'(x) = \gamma(x)$

otherwise. Then, obviously, $q\gamma\mu' = q\mu\gamma'$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$. As we have only defined γ' for abstract variables, clearly $\gamma'|_{\mathcal{A}} = \gamma'$. From $\mathcal{A}(\mathit{Range}(\mu')) = \varnothing$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we know that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$. We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (\mathit{ApproxGnd}(t,\mu) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have defined $a\gamma' = a\gamma$ and thus $\mathcal{N}(a\gamma') = \mathcal{N}(a\gamma) = \varnothing$. For $a \in \mathit{ApproxGnd}(t,\mu) \setminus \mathcal{G}$ by definition of *ApproxGnd* and equality of $\gamma\mu'$ and $\mu\gamma'$ we know that $a\gamma'$ is a ground term, i.e., $\mathcal{N}(a\gamma') = \varnothing$. Furthermore, note that $\mathcal{F}(\mathit{Range}(\mu')) \subseteq \mathcal{F}(t)$ and $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$. Thus, $\mathcal{F}(\mathit{Range}(\gamma')) \subseteq \mathcal{F}(t)$ and, consequently, $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$. For all $(s,s') \in \mathcal{U}$ we have $s\gamma \not\sim s'\gamma$ and, consequently, $s\gamma\mu' \not\sim s'\gamma\mu'$. But from $\gamma\mu' = \mu\gamma'$ we get $s\mu\gamma' \not\sim s'\mu\gamma'$. Thus, for all $(s'', s''') \in \mathcal{U}\mu$, we have $s\gamma' \not\sim s'\gamma'$. Second, if $\mathcal{V}(t) \not\subseteq \mathcal{G} \cup \mathcal{F}$, the answer substitution μ' can potentially instantiate any non-ground term in $q\gamma$ except for variables from $\mathcal{F}(q) \setminus \mathcal{F}(t)$. We define γ' in such a way that $\gamma\mu' = \mu\gamma'$. This is always possible because $\mathit{Dom}(\mu') \cap (\mathcal{F} \setminus \mathcal{F}(t)) = \varnothing$ and all variables in $\mathit{Range}(\mu)$ are fresh. Then, clearly, $q\gamma\mu' = q\mu\gamma$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$. As we only need to define γ' for abstract variables, clearly $\gamma'|_{\mathcal{A}} = \gamma'$. From $\mathcal{A}(\mathit{Range}(\mu')) = \varnothing$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we know that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$. We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (\mathit{ApproxGnd}(t,\mu) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have effectively defined $a\gamma' = a\gamma$ and thus $\mathcal{N}(a\gamma') = \mathcal{N}(a\gamma) = \varnothing$. For $a \in \mathit{ApproxGnd}(t,\mu) \setminus \mathcal{G}$ by definition of *ApproxGnd* and equality of $\gamma\mu'$ and $\mu\gamma'$ we know that $a\gamma'$ is a ground term, i.e., $\mathcal{N}(a\gamma') = \varnothing$. Furthermore, note that $\mathcal{F}(\mathit{Range}(\mu')) \subseteq \mathcal{F}(t)$ and $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$. Thus, $\mathcal{F}(\mathit{Range}(\gamma')) \subseteq \mathcal{F}(t)$ and, consequently, $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$. For all $(s,s') \in \mathcal{U}$ we have $s\gamma \not\sim s'\gamma$ and, consequently $s\gamma\mu' \not\sim s'\gamma\mu'$. But from $s\gamma\mu' = s\mu\gamma'$ and $s'\gamma\mu' = s'\mu\gamma'$ we get $s\mu\gamma' \not\sim s'\mu\gamma'$. Thus, for all $(s'', s''') \in \mathcal{U}\mu$, we have $s\gamma' \not\sim s'\gamma'$. □

Example 5.37. Consider again the logic program from Example 5.34. Now, consider the tree built using the rules from Parts 1 – 6:

$$\begin{tikzcd}
\boxed{\mathsf{p}(T_1); (\{T_1\}, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & & \\
\boxed{\mathsf{p}(T_1)^{22}_1; (\{T_1\}, \varnothing, \varnothing)} \arrow[r, "\textsc{Eval}"] \arrow[d, "T_1/\mathsf{s}(T_2)"', "\textsc{Eval}"] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} & \\
\boxed{\mathsf{p}(T_2), \mathsf{q}; (\{T_2\}, \varnothing, \varnothing)} \arrow[r, "\textsc{Split}"] \arrow[d, "\textsc{Split}"] & \boxed{\mathsf{q}; (\varnothing, \varnothing, \varnothing)} \arrow[d, "\textsc{Case}"] & \\
\boxed{\mathsf{p}(T_2); (\{T_2\}, \varnothing, \varnothing)} \arrow[uuu, dashed, bend left=60, "\textsc{Instance}"] & \boxed{\mathsf{q}^{23}_2; (\varnothing, \varnothing, \varnothing)} \arrow[d, "id"', "\textsc{Eval}"] \arrow[dr, "\textsc{Eval}"] & \\
\boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)} & \boxed{\square; (\varnothing, \varnothing, \varnothing)} \arrow[l, "\textsc{Success}"'] & \boxed{\varepsilon; (\varnothing, \varnothing, \varnothing)}
\end{tikzcd}$$

Thus, with the help of the SPLIT rule, we can easily close the tree.

5.5. Termination Graph to Logic Program

In this section we show how to extract a cut-free logic program from so-called termination graphs built by the abstract rules from the preceding section. For a graph G and a rule Rule, we use the notation $Rule(G)$ to denote all nodes of G to which Rule has been applied. We denote by $Succ(i, n)$ the i-th child of n.

Definition 5.38 (Termination Graph)**.** *A finite graph built from an initial state* $(s; KB)$ *using the rules from Definitions 5.14 – 5.35 is called a* termination graph *if, and only if, there is no cycle consisting only of* Instance *nodes and all leaves are of the form* $(\varepsilon; KB')$ *for some knowledge base* KB'*.*

First, we must not have cycles consisting only of Instance nodes as we do not obtain any clauses for them and thus, could falsely prove termination.

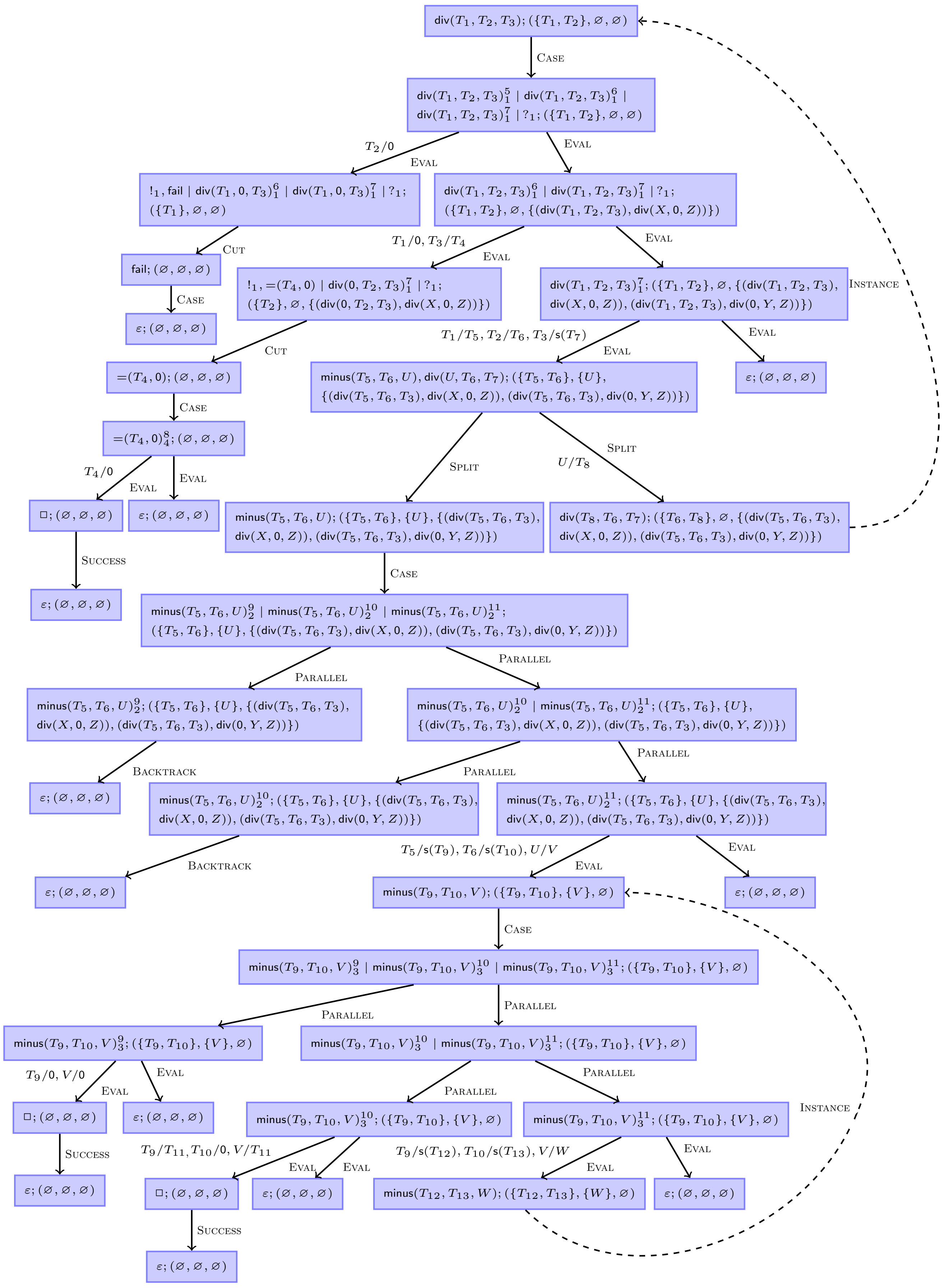
Second, we must have applied some rule to all nodes of the graph except for final states, i.e., those states where our computation stops. These are exactly those nodes for which the state consists only of the empty backtracking list ε.

Example 5.39. Consider again the graph from Example 5.33. All leaves are abstract states of the form $\varepsilon; (\varnothing, \varnothing, \varnothing)$. The only cycle traversing the only Instance node $\mathsf{q}(T_5); (\{T_5\}, \varnothing, \varnothing)$ also contains, for instance, the Case node $\mathsf{q}(T_1); (\{T_1\}, \varnothing, \varnothing)$. Thus, this graph is indeed a termination graph.

Note that we can always obtain a termination graph for any given logic program $\mathcal{P}$ and given abstract state $S; KB$. The first observation is that by using Parallel and Split, for any abstract state we can obtain a number of successor states consisting of just one "atomic" query $p(t_1, \ldots, t_k); KB$ or just a question mark, i.e., $?_m; KB$. For the latter we can apply Failure to obtain final states. For the former, our second observation is that by using Instance, we can generalize states of the form $p(t_1, \ldots, t_n); KB$ to states of the form $p(T_1, \ldots, T_k); (\varnothing, \varnothing, \varnothing)$ where $T_1, \ldots, T_k$ are fresh variables from $\mathcal{A}$. The conditions for Instance are trivially satisfied as we do not instantiate any variables from $\mathcal{N}$ and as $\mathcal{G}'$, $\mathcal{F}'$, and $\mathcal{U}'$ are empty.

To the new states we apply Case, Backtrack, Success, Failure, and Eval as long as possible where we always use Parallel and Split to deconstruct non-trivial backtracking lists and, if applicable, Instance to generalize $p(t'_1, \ldots, t'_n); KB$ to $p(T_1, \ldots, T_k)$; $(\varnothing, \varnothing, \varnothing)$. As the signature of $\mathcal{P}$ is finite and as we can reuse the generalized states, after finitely many steps we obtain a termination graph.

Example 5.40. The following graph can be obtained using Definitions 5.14 – 5.35 for the program from Example 5.1 and the set of queries $\mathcal{Q} = \{div(t_1, t_2, t_3) \mid t_1, t_2 \text{ are ground}\}$:


div(T1, T2, T3); ({T1, T2}, ∅, ∅)
Case
div(T1, T2, T3)^5_1 | div(T1, T2, T3)^6_1 | div(T1, T2, T3)^7_1 | ?1; ({T1, T2}, ∅, ∅)
T2/0
Eval
Eval
!1, fail | div(T1, 0, T3)^6_1 | div(T1, 0, T3)^7_1 | ?1; ({T1}, ∅, ∅)
div(T1, T2, T3)^6_1 | div(T1, T2, T3)^7_1 | ?1; ({T1, T2}, ∅, {(div(T1, T2, T3), div(X, 0, Z))})
Cut
fail; (∅, ∅, ∅)
Case
ε; (∅, ∅, ∅)
T1/0, T3/T4
Eval
Eval
!1, =(T4, 0) | div(0, T2, T3)^7_1 | ?1; ({T2}, ∅, {(div(0, T2, T3), div(X, 0, Z))})
div(T1, T2, T3)^7_1; ({T1, T2}, ∅, {(div(T1, T2, T3), div(X, 0, Z)), (div(T1, T2, T3), div(0, Y, Z))})
Instance
Cut
=(T4, 0); (∅, ∅, ∅)
Case
=(T4, 0)^8_4; (∅, ∅, ∅)
T4/0
Eval
Eval
□; (∅, ∅, ∅)
ε; (∅, ∅, ∅)
Success
ε; (∅, ∅, ∅)
T1/T5, T2/T6, T3/s(T7)
Eval
Eval
minus(T5, T6, U), div(U, T6, T7); ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
ε; (∅, ∅, ∅)
Split
U/T8
Split
minus(T5, T6, U); ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
div(T8, T6, T7); ({T6, T8}, ∅, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
Case
minus(T5, T6, U)^9_2 | minus(T5, T6, U)^10_2 | minus(T5, T6, U)^11_2; ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
Parallel
Parallel
minus(T5, T6, U)^9_2; ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
minus(T5, T6, U)^10_2 | minus(T5, T6, U)^11_2; ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
Backtrack
Parallel
Parallel
ε; (∅, ∅, ∅)
minus(T5, T6, U)^10_2; ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
minus(T5, T6, U)^11_2; ({T5, T6}, {U}, {(div(T5, T6, T3), div(X, 0, Z)), (div(T5, T6, T3), div(0, Y, Z))})
Backtrack
T5/s(T9), T6/s(T10), U/V
Eval
Eval
ε; (∅, ∅, ∅)
minus(T9, T10, V); ({T9, T10}, {V}, ∅)
ε; (∅, ∅, ∅)
Case
minus(T9, T10, V)^9_3 | minus(T9, T10, V)^10_3 | minus(T9, T10, V)^11_3; ({T9, T10}, {V}, ∅)
Parallel
Parallel
minus(T9, T10, V)^9_3; ({T9, T10}, {V}, ∅)
minus(T9, T10, V)^10_3 | minus(T9, T10, V)^11_3; ({T9, T10}, {V}, ∅)
T9/0, V/0
Eval
Eval
Parallel
Parallel
□; (∅, ∅, ∅)
ε; (∅, ∅, ∅)
minus(T9, T10, V)^10_3; ({T9, T10}, {V}, ∅)
minus(T9, T10, V)^11_3; ({T9, T10}, {V}, ∅)
Instance
Success
T9/T11, T10/0, V/T11
Eval
Eval
T9/s(T12), T10/s(T13), V/W
Eval
Eval
ε; (∅, ∅, ∅)
□; (∅, ∅, ∅)
ε; (∅, ∅, ∅)
minus(T12, T13, W); ({T12, T13}, {W}, ∅)
ε; (∅, ∅, ∅)
Success
ε; (∅, ∅, ∅)

Remember that our goal is to show termination of the initial state of the graph. By soundness of the abstract rules, we are left to prove that the cycles in our graph cannot be traversed infinitely often. Here, we have to show that the Instance edges for div and minus cannot be traversed infinitely often.

To show that the Instance edges cannot be traversed infinitely often, we build clauses for all paths in the graph that start in the topmost node, in the left child of the only Split node, or in the successor node of the Instance node for minus. We look at all paths that either end in one of the Success or Instance nodes or in the left child of the only Split node.

For the path from $\mathsf{div}(T_1, T_2, T_3); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\Box; (\varnothing, \varnothing, \varnothing)$, by looking at the substitutions $T_1/\mathsf{0}, T_3/T_4$ and $T_4/\mathsf{0}$ on the path, we obtain the fact $\mathsf{div}_1(\mathsf{0}, T_2, \mathsf{0})$. Note that we can use fresh natural numbers to label different occurrences of the same root symbol.

For the path from $\mathsf{div}(T_1, T_2, T_3); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\mathsf{div}(T_8, T_6, T_7); (\{T_6, T_8\}, \varnothing, \{(\mathsf{div}(T_5), T_6, T_3), \mathsf{div}(X, \mathsf{0}, Z)), (\mathsf{div}(T_5), T_6, T_3), \mathsf{div}(\mathsf{0}, Y, Z))\})$ we have to consider the substitutions $T_1/T_5, T_2/T_6, T3/\mathsf{s}(T_7)$ and U/T_8. We can also use that $\mathsf{minus}(T_5, T_6, U)$ needs to be derived to $\Box$ before we can continue with div. Thus, we obtain the new clause $\mathsf{div}_1(T_5, T_6, \mathsf{s}(T_7)) \leftarrow \mathsf{minus}_1(T_5, T_6, T_8), \mathsf{div}_1(T_8, T_6, T_7)$. Note that we used the same fresh natural number for both occurrences of div as they are linked by an Instance edge.

Continuing in this manner, we obtain the following logic program for which we have to show termination w.r.t. the set of queries $\mathcal{Q} = \{\mathsf{div}_1(t_1, t_2, t_3) \mid t_1, t_2 \text{ are ground}\}$ as specified by the knowledge base in the first node.

$$\begin{array}{rcl}
\mathsf{div}_1(\mathsf{0}, T_2, \mathsf{0}). & & \\
\mathsf{div}_1(T_5, T_6, \mathsf{s}(T_7)) & \leftarrow & \mathsf{minus}_1(T_5, T_6, T_8), \mathsf{div}_1(T_8, T_6, T_7). \\
\mathsf{div}_1(T_5, T_6, \mathsf{s}(T_7)) & \leftarrow & \mathsf{minus}_1(T_5, T_6, U). \\
\mathsf{minus}_1(\mathsf{s}(\mathsf{0}), \mathsf{s}(T_{10}), \mathsf{0}). & & \\
\mathsf{minus}_1(\mathsf{s}(T_{11}), \mathsf{s}(\mathsf{0}), T_{11}). & & \\
\mathsf{minus}_1(\mathsf{s}(T_{12}), \mathsf{s}(T_{13}), V) & \leftarrow & \mathsf{minus}_2(T_{12}, T_{13}, V). \\
\mathsf{minus}_2(\mathsf{0}, T_{10}, \mathsf{0}). & & \\
\mathsf{minus}_2(T_{11}, \mathsf{0}, T_{11}). & & \\
\mathsf{minus}_2(\mathsf{s}(T_{12}), \mathsf{s}(T_{13}), W) & \leftarrow & \mathsf{minus}_2(T_{12}, T_{13}, W).
\end{array}$$

The above logic program can easily be proved terminating both by the transformational method presented in Chapter 3, by the direct one from Chapter 4, and, of course, by their combination through Theorem 4.22. Indeed, virtually all methods for proving termination of logic programs will succeed to prove termination of this definite logic program.

We now show how to obtain a logic program and a set of queries (characterized by a moding function) from a termination graph. To this end, we first need the notion of a clause path to characterize paths in the termination graph from which we generate the clauses for the logic program.

Definition 5.41 (Clause Path)**.** *A path $\pi = n_1 \dots n_k$ is a* clause path *w.r.t. G if, and only if, $k > 1$ and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Split}(G))$,
- $n_k \in \mathit{Success}(G) \cup \mathit{Instance}(G) \cup \mathit{Succ}(1, \mathit{Split}(G))$,
- *for all* $1 \leq j < k$, $n_j \notin \mathit{Instance}(G)$, *and*
- *for all* $1 < j < k$, $n_{j-1} \in \mathit{Split}(G) \implies n_j = \mathit{Succ}(2, n_{j-1})$.

Intuitively, the above definition characterizes all non-trivial paths that start at the successor of an INSTANCE node or at a left successor of a SPLIT node, end at a SUCCESS node, at an INSTANCE node, or at a left successor of a SPLIT node, and do not traverse an INSTANCE edge or left successors of SPLIT nodes. We do not traverse INSTANCE edges to make sure there are only finitely many clause paths. Instead of following left successors of SPLIT nodes, we get clause paths starting at these nodes. These will correspond to the intermediate body atoms in the generated logic program.

Now, for obtaining a cut-free logic program, the idea is to construct one clause for each clause path $\pi = n_1 \dots n_k$. Here, the head of the new clause is the renamed state of n_1 where we apply the substitutions between n_1 and n_k. The last body atom is the renamed state of n_k. As intermediate body atoms we use renamed states of nodes n that are left children of some $n_i \in \mathit{Split}(G)$. Note that also here, we apply the substitutions between n_i and n_k to the respective intermediate body atom.

The renaming contributes to the success of our method in three ways. First, it allows to use different predicate symbols for different nodes. This is needed, for instance, in Example 5.40 where the cut-free logic program would not terminate if one was to identify minus_1 and minus_2. Second, it allows to represent a whole state by one atom, even if this state consisted of a non-atomic goal or a backtracking list with more than one element. Third, it simplifies the problem even for states consisting of a goal of one atom by flattening the context of the variables to one predicate symbol.

The only remaining problem is that paths may contain BACKTRACK or SUCCESS edges or go to the right child of some EVAL node. Here, the substitutions that correspond to the same cut scope, i.e., the same number m, must not be regarded for instantiating the head of the new clause. The reason is that backtracking undoes the substitutions of the clauses tried before. Thus, we collect the substitutions between n_1 and n_k backwards and whenever we pass a a problematic edge, we keep track of the current scope m. When we come from the left child of some EVAL node with the same m, we disregard the corresponding substitution. Note that the same is not necessary for the intermediate body atoms as we do not allow any non-empty backtracking list after the goal to split.

Definition 5.42 (Logic Programs and Queries from Termination Graph)**.** *The logic program $\mathcal{P}_G$ and the moding function m_G for a termination graph G are defined as $\mathcal{P}_G = \bigcup_{\pi \text{ clause-path w.r.t. } G} Clause(\pi)$ and $m_G(p,i) = \textbf{in}$ if, and only if, for all clause-paths $\pi = (n_1; (\mathcal{G},\mathcal{F},\mathcal{U}))\ldots$ w.r.t. G, whenever $Rename(n_1) = p(x_1,\ldots,x_n)$ then $x_i \in \mathcal{G}$.*

For a path $\pi = n_1 \ldots n_k$, we define $Clause(\pi) = Rename(n_1)\sigma_{\pi,\varnothing} \leftarrow I_\pi, Rename(n_k)$. Here, the Rename function is defined as follows. For $n \in Success(G)$, $Rename(n)$ is $\square$ and for $n \in Instance(G)$, it is $Rename(Succ(1,n))\mu$ where μ is the substitution associated with this INSTANCE *node. Otherwise, $Rename(n)$ is $p_n(\mathcal{V}(n))$ where p_n is a fresh predicate symbol and $\mathcal{V}(S;KB) = \mathcal{V}(S)$.*

Finally, $\sigma_{\pi,M}$ and I_π are defined as follows (where the substitutions μ and σ' and the index m are from the corresponding node n_{j-1}).

$$\sigma_{n_1\ldots n_j,M} = \begin{cases} id & \text{if } j = 1 \\ \sigma_{n_1\ldots n_{j-1},M}\mu & \text{if } n_{j-1} \in Split(G),\ n_j = Succ(2,n_{j-1}) \\ \sigma_{n_1\ldots n_{j-1},M}\sigma' & \text{if } n_{j-1} \in Eval(G),\ n_j = Succ(1,n_{j-1}), \\ & \text{and } m \notin M \text{ for } n_{j-1} = (t,q)^i_m \mid S;KB \\ \sigma_{n_1\ldots n_{j-1},M}\sigma|_{\mathcal{G}} & \text{if } n_{j-1} \in Eval(G),\ n_j = Succ(1,n_{j-1}), \\ & \text{and } m \in M \text{ for } n_{j-1} = (t,q)^i_m \mid S;(\mathcal{G},\mathcal{F},\mathcal{U}) \\ \sigma_{n_1\ldots n_{j-1},M\cup\{m\}} & \text{if } n_{j-1} \in Eval(G),\ n_j = Succ(2,n_{j-1}) \\ & \text{or } n_{j-1} \in Backtrack(G) \cup Success(G) \\ & \text{and } n_j = (t,q)^i_m \mid S;KB \\ \sigma_{n_1\ldots n_{j-1},M} & \text{otherwise} \end{cases}$$

$$I_{n_j\ldots n_k} = \begin{cases} \square & \text{if } j = k \\ Rename(Succ(1,n_j))\sigma_{n_j\ldots n_k,\varnothing}, I_{n_{j+1}\ldots n_k} & \text{if } n_j \in Split(G), n_{j+1} = Succ(2,n_j) \\ I_{n_{j+1}\ldots n_k} & \text{otherwise} \end{cases}$$

Example 5.43. The graph G in Example 5.33 is a termination graph. Applying the above definition we obtain the single clause logic program $\mathcal{P}_G$ with the clause $\mathsf{q}(\mathsf{s}(T_5)) \leftarrow \mathsf{q}(T_5)$ and the moding function m_G where $m_G(\mathsf{q},1) = \textbf{in}$. Termination of $\mathcal{P}_G$ is trivial and can be shown by any technique.

The following lemma shows how the clauses of $LP(G)$ can be used to simulate concrete derivations of concrete states described by the states of the graph.

Lemma 5.44 (Simulation using LP(G)). *Let $S_0\gamma_0 \in \mathcal{C}(S_0; KB_0)$ and $S_k\gamma_k \in \mathcal{C}(S_k; KB')$ for some path $\pi = n_0 \dots n_k$ w.r.t. G with $n_i = S_i; KB_i$ where for $i < k$, $n_i \notin Instance(G)$.*

If there is a concrete derivation from $S_0\gamma_0$ to $S_k\gamma_k \mid S'\gamma_k$, then $Rename(n_0)\gamma_0 \vdash_{Clause(\pi)} \circ \vdash_{\mathcal{P}_G}^* Rename(n_k)\gamma_k$.

Proof. We perform the proof by induction, using as the induction relation the lexicographic combination of the length k of the concrete derivation from $S_0\gamma_0$ to $S_k\gamma_k \mid S'\gamma_k$ and the edge relation of the graph G' obtained from G by keeping only outgoing edges from nodes in $Instance(G) \cup Split(G) \cup Parallel(G)$. This relation is well-founded, as every cycle of the termination graph must contain at least one outgoing edge from $Instance(G) \cup Split(G) \cup Parallel(G)$.

For $k = 0$ we have $Clause(\pi) = Rename(n_0) \leftarrow Rename(n_0)$. Thus, obviously we have $Rename(n_0)\gamma_0 \vdash_{Clause(\pi)} Rename(n_0)\gamma_0$. For $k > 0$, we can assume the lemma holds for concrete derivations of length at most $k - 1$.

We now perform a case analysis based on n_k and n_{k-1}. If $n_{k-1} \in Split(G)$ and $n_k = Succ(2, n_{k-1})$, i.e., we traverse the right child of a Split node, we know that by the induction hypothesis $Rename(n_0)\gamma_0 \vdash_{Clause(n_1 \dots n_{k-1})} Rename(n_{k-1})\gamma_{k-1}$. From the proof of Split we know that $\gamma_{k-1}\mu' = \mu\gamma_k$ where $Rename(Succ(1, n_{k-1}))\gamma_{k-1} \vdash_{\mathcal{P}_G}^* \square$ with answer substitution μ'. We may assume the latter due to the induction hypothesis and the derivation being shorter than k. Thus, we obtain:

$$\begin{array}{rl}
 & Rename(n_0)\gamma_0 \\
\vdash_{Clause(\pi)} \circ \vdash_{\mathcal{P}_G} & Rename(Succ(1, n_{k-1}))\mu\gamma_k, Rename(n_k)\mu^{-1}\gamma_{k-1} \\
= & Rename(Succ(1, n_{k-1}))\gamma_{k-1}\mu', Rename(n_k)\mu^{-1}\gamma_{k-1} \\
\vdash_{\mathcal{P}_G} & Rename(n_k)\mu^{-1}\gamma_{k-1}\mu' \\
= & Rename(n_k)\mu^{-1}\mu\gamma_k \\
= & Rename(n_k)\gamma_k
\end{array}$$

If $n_{k-1} \in Eval(G)$ and $n_k = Succ(1, n_{k-1})$, i.e., we traverse the left child of an Eval node, we know that by the induction hypothesis, $Rename(n_0)\gamma_0 \vdash_{Clause(n_1 \dots n_{k-1})} \circ \vdash_{\mathcal{P}_G}^*$ $Rename(n_{k-1})\gamma_{k-1}$. From the proof of Eval we know that $q\gamma_{k-1}\sigma'' = q\sigma'\gamma_k$ and $S\gamma_{k-1} = S\sigma|_{\mathcal{G}}\gamma_k$. Let $n_{k-1} = t, q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl}
 & Rename(n_0)\gamma_0 \\
\vdash_{Clause(\pi)} \circ \vdash_{\mathcal{P}_G} & Rename(q\gamma_{k-1}\sigma'' \mid S\gamma_{k-1}; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\
= & Rename(q\sigma'\gamma_k \mid S\sigma|_{\mathcal{G}}\gamma_k; (\mathcal{G}, \mathcal{G}, \mathcal{U})) \\
= & Rename(n_k)\gamma_k
\end{array}$$

If $n_{k-1} \in Eval(G)$ and $n_k = Succ(2, n_{k-1})$ or $n_{k-1} \in Backtrack(G) \cup Success(G)$ and

$n_{k-1} = (t,q)^i_m \mid S; KB_{k-1}$, i.e., we backtrack by removing the first element of the backtracking list, we perform a case analysis based on the existence of a CASE node introducing m in $n_1 \dots n_{k-1}$. If such a CASE node does not exist in our path, there is also no SPLIT node in our path. Thus, $\sigma_\pi = id$ and $I_\pi = \Box$. Then, $Rename(n_0)\gamma_0 \vdash_{Rename(n_0) \leftarrow Rename(n_k)} Rename(n_k)\gamma_k$. If there is such a CASE node in our path, i.e., there is a j such that $n_j \in Case(G)$ and $n_j = (t',q')^{i'}_m \mid S'; KB_j$, we know by the induction hypothesis that $Rename(n_0)\gamma_0 \vdash_{Clause(n_1 \dots n_j)} \circ \vdash_{\mathcal{P}_G} Rename(n_j)\gamma_j$. As $\sigma_{n_j \dots n_k} = id$ and $I_{n_j \dots n_k} = \Box$ we have $Rename(n_0)\gamma_0 \vdash_{Clause(\pi)} \circ \vdash_{\mathcal{P}_G} Rename(n_k)\gamma_k$.

Finally, if n_{k-1} is in none of the above sets, we have $\sigma_\pi = \sigma_{n_1 \dots n_{k-1}}$ and $I_\pi = I_{n_1 \dots n_{k-1}}$. Again, from the induction hypothesis we know that $Rename(n_0)\gamma_0 \vdash_{Clause(n_1 \dots n_{k-1})} \circ \vdash_{\mathcal{P}_G} Rename(n_{k-1})\gamma_{k-1}$. From the definition of the abstract rules used, we know that $\mathcal{V}(s_k) \subseteq \mathcal{V}(s_{k-1})$ and, thus, $Rename(n_0)\gamma_0 \vdash_{Clause(\pi)} \circ \vdash_{\mathcal{P}_G} Rename(n_k)\gamma_{k-1} = Rename(n_k)\gamma_k$. $\Box$

Now, we need a way to relate concrete infinite derivations to infinite paths in the graph. As the INSTANCE rule allows renaming of variables and as the question marks introduced by CASE are unique, we need to view outgoing edges of INSTANCE nodes not just as referring back to a previous node, but to a renamed-apart copy of the termination graph where variable names and CASE marks have been renamed accordingly. We call the infinite graph that we obtain in this way the *unrolled termination graph.*

Lemma 5.45 (Infinite Path for Infinite Derivation). *Let $S\gamma \in \mathcal{C}(S; KB)$ for some $n_0 = S; KB \in G$. If $S\gamma$ has an infinite concrete derivation $s_0 s_1 s_2 \dots$ with $s_0 = S\gamma$, there is an infinite path $n_0 n_1 \dots$ in the unrolled termination graph and there are indices $j_0, j_1, \dots$ such that for all i there is a prefix s'_{j_i} of s_{j_i} with $s'_{j_i} \in \mathcal{C}(n_i)$ and s'_{j_i} has an infinite derivation.*

Proof. By the proofs of soundness for all abstract rules, we know that for any node n, if some $s'_0 \in \mathcal{C}(n)$ has an infinite concrete derivation $r_0 r_1 r_2 \dots$, for some j there is some prefix r'_j of r_j that has an infinite derivation $r'_j r'_{j+1} r'_{j+2} \dots$ where r'_k is a prefix of r_k for $k \geq j$ and $r'_j \in \mathcal{C}(n')$ for some successor node n' of n.

To see this, consider the following case analysis over all abstract rules. For SUCCESS, FAILURE, CUT, CASE, and BACKTRACK we clearly have $j = 1$ and $r'_1 = r_1$. That $r_1 \in \mathcal{C}(Succ(1,n))$ holds follows directly from the proofs of these rules. Similarly, for INSTANCE we have $j = 0$, $r'_0 = r_0$, and $r'_0 \in \mathcal{C}(Succ(1,n))$.

For PARALLEL we perform a case analysis. If $S\gamma$ from $r_0 = S\gamma \mid S'\gamma$ has an infinite derivation, we have $j = 0$, $r'_0 = S\gamma$, and $r'_0 \in \mathcal{C}(Succ(1,n))$. If $S\gamma$ does not have an infinite derivation, there must be a j such that $r_j = S'\gamma$. We have $r'_j = r_j$ and $r'_j \in \mathcal{C}(Succ(2,n))$.

For SPLIT we perform a case analysis. If $t\gamma$ from $r_0 = t\gamma, q\gamma$ has an infinite derivation, $j = 0$, $r'_0 = t\gamma$, and $r'_0 \in \mathcal{C}(Succ(1,n))$. If $t\gamma$ has not infinite derivation, there must be a j such that $r_j = q\gamma\mu'$ for $t\gamma \vdash^*_{\mathcal{P}} \Box$ with answer substitution μ'. We have $r'_j = r_j$ and $r'_j \in \mathcal{C}(Succ(2,n))$.

By applying the above reasoning repeatedly, we obtain the infinite list of indices $j_0 j_1 j_2 \dots$ and, thus, prove the lemma. $\square$

Theorem 5.46 (Correctness)**.** *If G is a termination graph for $\mathcal{P}$ such that $\mathcal{P}_G$ is terminating w.r.t. the moding m_G, then all concretizations of all states have only finite derivations w.r.t. the rules of Definition 5.6.*

Proof. Assume $\mathcal{P}_G$ is terminating w.r.t. the moding function m_G, but there is a concretization $S\gamma \in \mathcal{C}(S; KB)$ from some $n_0 = S; KB$ that has an infinite concrete derivation. Then, according to Lemma 5.45 there is an infinite path $n_0 n_1 n_2 \dots$ in the unrolled termination graph and there are indices $j_0 j_1 j_2 \dots$ such that for all i there is a prefix s'_{j_i} of s_{j_i} with $s'_{j_i} \in \mathcal{C}(n_i)$ and s'_{j_i} has an infinite derivation.

Then, there is a k such that the infinite suffix $n_k n_{k+1} n_{k+2} \dots$ of $n_0 n_1 n_2 \dots$ consists of an infinite sequence of clause paths $\pi_0 \pi_1 \pi_2 \dots$ and there are indices $l_0 l_1 l_2 \dots$ such that for $n_{l_m} = S_{l_m}; KB_{l_m}$ and $s'_{l_m} \in \mathcal{C}(n_{l_m})$ we have $s'_{l_m} \in \mathcal{C}(n_m)$ where $\pi_m = n_m \dots$.

Now, according to Lemma 5.44, for all $m \in \mathbb{N}$, we have $\mathit{Rename}(n_{l_m})\gamma_{l_m} \vdash_{\mathit{Clause}(\pi_m)} \circ \vdash_{\mathcal{P}_G} \mathit{Rename}(n_{l_{m+1}})\gamma_{l_{m+1}}$.

Thus, $\mathit{Rename}(n_{l_0})\gamma_{l_0}$ has an infinite derivation w.r.t. $\mathcal{P}_G$. As $S_{l_0}\gamma_{l_0} \in \mathcal{C}(n_{l_0})$, $\mathcal{P}_G$ is not terminating w.r.t. the moding function m_G. This contradicts our initial assumption and, thus, proves the theorem. $\square$

Corollary 5.47 (Termination Analysis of Logic Programs with Cuts)**.** *A logic program $\mathcal{P}$ is terminating w.r.t. a class of queries described by a symbol $p \in \Sigma$ and a moding function $m : \Sigma \times \mathbb{N} \to \{\boldsymbol{in}, \boldsymbol{out}\}$ if G is a termination graph for the initial node $p(T_1, \dots, T_n), (\mathcal{G}, \varnothing, \varnothing)$ where $\mathcal{G} = \{T_i \mid m(p, i) = \boldsymbol{in}\}$ and $\mathcal{P}_G$ terminates w.r.t. the moding function m_G.*

The reverse direction of the corollary above does not hold. The following example demonstrates that our pre-processing is not termination-preserving.

Example 5.48. Consider the logic program $\mathcal{P}$ consisting of the following clauses:

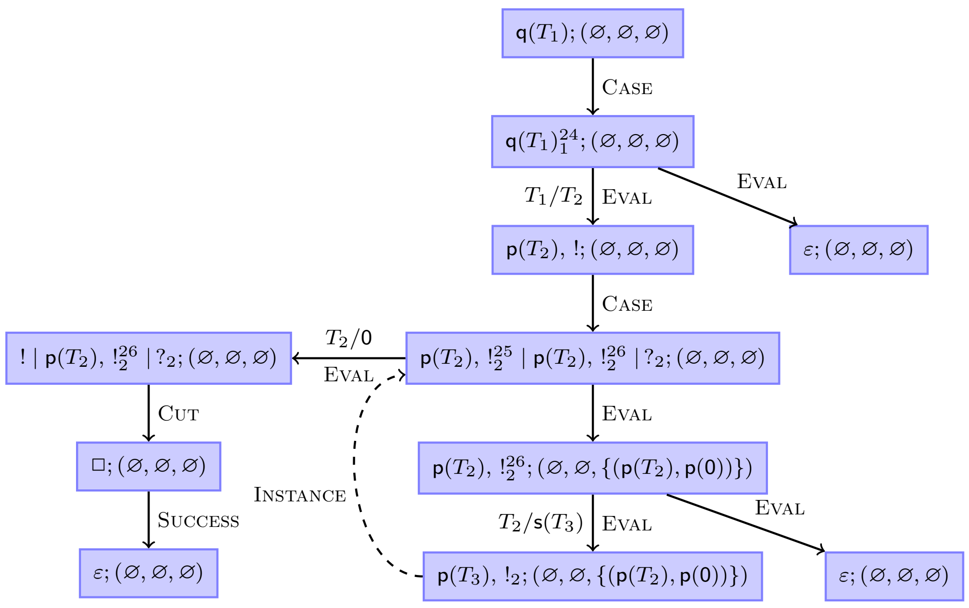
$$\mathsf{q}(X) \leftarrow \mathsf{p}(X), !. \tag{24}$$

$$\mathsf{p}(\mathsf{0}). \tag{25}$$

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{26}$$

Now, consider the set of queries described by the symbol q and the moding function m with $m(\mathsf{q}, 1) = \mathbf{out}$, i.e., the set of all queries $\mathsf{q}(t)$ for arbitrary terms t. In our setting of unification with occur check, we can assume t to be a finite term. But then, for any t, after finitely many steps removing one s, there are two possibilities. First, the derivation can fail because we reach some function symbol different to $\mathsf{0}$ and s. Second, we reach $\mathsf{0}$ or a variable and, consequently, the cut. Thus, $\mathcal{P}$ terminates w.r.t. m.

We obtain the following termination graph G for the initial node $\mathsf{q}(T_1)$.


$\mathsf{q}(T_1); (\varnothing, \varnothing, \varnothing)$
CASE
$\mathsf{q}(T_1)_1^{24}; (\varnothing, \varnothing, \varnothing)$
T_1/T_2 EVAL
EVAL
$\mathsf{p}(T_2), !; (\varnothing, \varnothing, \varnothing)$
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
CASE
$! \mid \mathsf{p}(T_2), !_2^{26} \mid ?_2; (\varnothing, \varnothing, \varnothing)$
$T_2/0$
EVAL
$\mathsf{p}(T_2), !_2^{25} \mid \mathsf{p}(T_2), !_2^{26} \mid ?_2; (\varnothing, \varnothing, \varnothing)$
CUT
EVAL
$\square; (\varnothing, \varnothing, \varnothing)$
INSTANCE
$\mathsf{p}(T_2), !_2^{26}; (\varnothing, \varnothing, \{(\mathsf{p}(T_2), \mathsf{p}(0))\})$
SUCCESS
$T_2/\mathsf{s}(T_3)$ EVAL
EVAL
$\varepsilon; (\varnothing, \varnothing, \varnothing)$
$\mathsf{p}(T_3), !_2; (\varnothing, \varnothing, \{(\mathsf{p}(T_2), \mathsf{p}(0))\})$
$\varepsilon; (\varnothing, \varnothing, \varnothing)$

We obtain $\mathcal{P}_G$ containing the following clauses:

$$\mathsf{p}_1(0).$$

$$\mathsf{p}_1(\mathsf{s}(T_3)) \leftarrow \mathsf{p}_1(T_3).$$

For the moding function m_G we obtain $m_G(\mathsf{p}_1, 1) = \textbf{out}$. But, for instance, the query $\mathsf{p}_1(X)$ does not terminate.

5.6. Summary

We have introduced a novel non-termination preserving pre-processing method to eliminate the effect of cuts in many practically relevant cases. After this pre-processing, any technique for proving universal termination of logic programming can be applied. As our method also works for meta-programming and as cuts can be used to express negation-as-failure as well as existential termination, the contributions of this chapter solve challenges (i), (ii), and (iii) from the beginning of this chapter.

The success for meta-programming and, in particular, negation-as-failure is demonstrated by Example 5.33 as the logic program obtained after the pre-processing step is trivially termination (cf. Example 5.43).

Regarding existential termination, Example 5.48 shows that without further improvements, our approach in its current form is not very strong for showing existential termination. Note that the graph does contain all information needed for the termination proof. The problem is just that this information is not used when constructing the cut-free logic programs.

This pre-processing has been implemented in our tool AProVE and the implementation indeed succeeds in proving termination of both Example 5.1 and Example 5.33. It has also been tested successfully on a number of further examples where the cut is needed to ensure termination.

Future work would primarily be to make the abstract rules more precise, e.g., by using sharing analysis to track possibly shared variables more precisely, by better approximating answer substitutions for the SPLIT rule, or by using an approach based on argument filters (cf. Chapter 3) instead of simply tracking abstract variables instantiated by ground terms.

Finally, additional shape analysis for the structure of terms used in answer substitutions should improve power on queries expressing existential termination (cf. Example 5.48).

Part II.

Automating Search by Encoding to SAT

6. Recursive Path Order

In the first part of this thesis we presented powerful techniques for termination analysis of logic programs. Regardless of whether we use the transformational approach of Chapter 3, the direct approach of Chapter 4, their combination through Theorem 4.22, or whether we apply the pre-processing from Chapter 5 – in the end it all boils down to searching for a well-founded order on terms.

To this end, many tools search for polynomial orders, i.e., polynomial interpretations of terms into natural numbers. Still, there are many natural examples where no tool based on polynomial orders can show termination.

Example 6.1. The following logic program from the [TPD07] can be used to compute the conjunctive normal form (CNF) of a propositional formula represented by $\mathsf{a}/2$ and $\mathsf{o}/2$ for conjunction and disjunction, respectively, and $\mathsf{n}/1$ for negation.

$$\mathsf{cnfequiv}(X,Y) \leftarrow \mathsf{transform}(X,Z), \mathsf{cnfequiv}(Z,Y).$$
$$\mathsf{cnfequiv}(X,X).$$
$$\mathsf{transform}(\mathsf{n}(\mathsf{n}(X)),X).$$
$$\mathsf{transform}(\mathsf{n}(\mathsf{a}(X,Y)),\mathsf{o}(\mathsf{n}(X),\mathsf{n}(Y))).$$
$$\mathsf{transform}(\mathsf{n}(\mathsf{o}(X,Y)),\mathsf{a}(\mathsf{n}(X),\mathsf{n}(Y))).$$
$$\mathsf{transform}(\mathsf{o}(X,\mathsf{a}(Y,Z)),\mathsf{a}(\mathsf{o}(X,Y),\mathsf{o}(X,Z))).$$
$$\mathsf{transform}(\mathsf{o}(\mathsf{a}(X,Y),Z),\mathsf{a}(\mathsf{o}(X,Z),\mathsf{o}(Y,Z))).$$
$$\mathsf{transform}(\mathsf{o}(X1,Y),\mathsf{o}(X2,Y)) \leftarrow \mathsf{transform}(X1,X2).$$
$$\mathsf{transform}(\mathsf{o}(X,Y1),\mathsf{o}(X,Y2)) \leftarrow \mathsf{transform}(Y1,Y2).$$
$$\mathsf{transform}(\mathsf{a}(X1,Y),\mathsf{a}(X2,Y)) \leftarrow \mathsf{transform}(X1,X2).$$
$$\mathsf{transform}(\mathsf{a}(X,Y1),\mathsf{a}(X,Y2)) \leftarrow \mathsf{transform}(Y1,Y2).$$
$$\mathsf{transform}(\mathsf{n}(X1),\mathsf{n}(X2)) \leftarrow \mathsf{transform}(X1,X2).$$

The clauses for $\mathsf{transform}$ implement one transformation step which can either be elimination of double negation, De Morgan's rule, or distribution. The last 5 clauses allow to descend into formulas. Finally, the first answer for $\mathsf{cnfequiv}(t,X)$, where t is a propositional formula, is the CNF of t. At the end of Chapter 7 we are able to use recursive path order to show termination w.r.t. the set of queries $\{\mathsf{cnfequiv}(t_1,t_2) \mid t_1 \text{ is ground }\}$.

Since 2006, several papers have illustrated the huge potential in applying SAT solvers for various types of termination problems for term rewrite systems (TRSs). The key idea is classic: the specific NP-complete termination problem for a TRS $\mathcal{R}$ is encoded to a propositional formula φ which is satisfiable if and only if $\mathcal{R}$ has the desired termination property. Satisfiability of φ is tested using a state-of-the-art SAT solver [LP08, ES07] and the termination proof for $\mathcal{R}$ is reconstructed from a satisfying assignment of φ. However, in order to obtain significant speedups, it is crucial to base the approach on polynomial-size encodings which are also small in practice.

The first such attempt addresses LPO-termination [KK04]. This work is based on BDDs and does not yield competitive results. A significant improvement is described in [CLS06] which presents an extremely fast SAT-based implementation. Successful SAT encodings of other termination techniques are presented in [EWZ06, FGM+07, HW06, ZM07]. A common theme in all of these works is to represent (finite domain) integer variables as binary numbers in bit representation and to encode arithmetic constraints as Boolean functions on these representations.

This chapter introduces the first SAT-based encoding for full recursive path orders. The main new and interesting contributions compared to [CLS06] are (1) the encoding for the lexicographic comparison w.r.t. permutations and (2) the encoding for the multiset extension of the base order.

Our encoding of RPO is implemented in the termination prover AProVE [GST06]. The combination of a termination prover with a SAT solver yields a surprisingly fast implementation of RPO. All 865 TRSs in the *Termination Problem Data Base (TPDB)* [TPD06] are analyzed in 50 seconds for the case of strict precedences, i.e., the SAT solver decides whether a given TRS is terminating with RPO in, on average, less than 100 ms. Allowing non-strict precedences takes only 85.3 seconds. Moreover, power increases considerably compared to the implementation of LPO described in [CLS06]: 25 additional termination proofs are obtained.

Note that this collection also contains TRSs that are the results of transforming logic programs to term rewriting in the spirit of Chapter 3. Such TRSs typically produce particularly hard search problems as the arities of the many function symbols introduced by the translation are rather high (5 – 20 arguments per function symbol). Thus, an efficient implementation is absolutely essential for the practical application of recursive path orders to TRSs resulting from logic programming.

After the necessary preliminaries on RPO in Section 6.1, Section 6.2 shows how to encode both multiset comparisons and lexicographic comparisons w.r.t. permutations, and how to combine them into a single class of orders. In Section 6.3 we describe the implementation of our results and provide extensive experimental evidence indicating speedups in orders of magnitude. We summarize the contributions of this chapter in Section 6.4.

6.1. Preliminaries

The classical approach to prove termination of a TRS $\mathcal{R}$ is to find a *reduction order* $\succ$ which orients all rules $\ell \to r$ in $\mathcal{R}$ (i.e., $\ell \succ r$). A reduction order is an order which is well-founded, monotonic, and stable (closed under contexts and substitutions). In practice, most reduction orders amenable to automation are *simplification orders* [Der82].

Three of the most prominent simplification orders are the lexicographic path order (LPO) [KL80], the multiset path order (MPO) [Der82], and the recursive path order (RPO) [Les83], which combines the lexicographic and multiset path order allowing also permutations in the lexicographic comparison. This section introduces their definitions using a formulation that is suitable for the subsequent SAT encoding.

We assume all terms to be constructed over a signature Σ of function symbols and variables $\mathcal{V}$. For a quasi-order $\succsim$ (i.e., a transitive and reflexive relation), we define $s \succ t$ if, and only if, $s \succsim t$ and $t \not\succsim s$ and we define $s \sim t$ if, and only if, both $s \succsim t$ and $t \succsim s$. Path orders are defined in terms of lexicographic and multiset extensions of a base order (on terms). We often denote tuples of terms as $\bar{s} = \langle s_1, \dots s_n \rangle$, etc.

Definition 6.2 (lexicographic extension). *Let* $\succsim$ *be a quasi-order. The* lexicographic extensions *of* $\succ$*,* $\sim$*, and* $\succsim$ *are defined on sequences of terms:*

- $\langle s_1, \dots, s_n \rangle \sim^{lex} \langle t_1, \dots, t_m \rangle$ *if and only if* $n = m$ *and* $s_i \sim t_i$ *for all* $1 \le i \le n$
- $\langle s_1, \dots, s_n \rangle \succ^{lex} \langle t_1, \dots, t_m \rangle$ *if and only if (a)* $m = 0$ *and* $n > 0$*; or (b)* $s_1 \succ t_1$*; or (c)* $s_1 \sim t_1$ *and* $\langle s_2, \dots, s_n \rangle \succ^{lex} \langle t_2, \dots, t_m \rangle$*.*
- $\succsim^{lex} \;=\; \sim^{lex} \cup \succ^{lex}$

So for tuples of numbers $\bar{s} = \langle 3, 3, 4, 0 \rangle$ and $\bar{t} = \langle 3, 2, 5, 6 \rangle$, we have $\bar{s} >^{lex} \bar{t}$ as $s_1 = t_1$ and $s_2 > t_2$ (where $>$ is the usual order on numbers).

The multiset extension of an order $\succ$ is defined as follows: $\bar{s} \succ^{mul} \bar{t}$ holds if $\bar{t}$ is obtained by replacing at least one element of $\bar{s}$ by a finite number of (strictly) smaller elements. However, the order of the elements in $\bar{s}$ and $\bar{t}$ is irrelevant. For example, let $\bar{s} = \langle 3, 3, 4, 0 \rangle$ and $\bar{t} = \langle 4, 3, 2, 1, 1 \rangle$. We have $\bar{s} >^{mul} \bar{t}$ because $s_1 = 3$ is replaced by the smaller elements $t_3 = 2$, $t_4 = 1$, $t_5 = 1$ and $s_4 = 0$ is replaced by zero smaller elements. So each element in $\bar{t}$ is "covered" by some element in $\bar{s}$. Such a cover is either by a larger s_i (then s_i may cover several t_j) or by an equal s_i (then one s_i covers one t_j). In this chapter we formalize the multiset extension by a *multiset cover* which is a pair of mappings (γ, ε). Intuitively, γ expresses which elements of $\bar{s}$ cover which elements in $\bar{t}$ and ε expresses for which s_i this cover is by means of equal terms and for which by means of greater terms. This formalization facilitates encodings to propositional logic.

So in the example above, we have $\gamma(1) = 3$, $\gamma(2) = 2$ (since t_1 is covered by s_3 and t_2 is covered by s_2), and $\gamma(3) = \gamma(4) = \gamma(5) = 1$ (since t_3, t_4, and t_5 are all covered by s_1).

Moreover, $\varepsilon(2) = \varepsilon(3) = \mathit{true}$ (since s_2 and s_3 are replaced by equal components), whereas $\varepsilon(1) = \varepsilon(4) = \mathit{false}$ (since s_1 and s_4 are replaced by (possibly zero) smaller components). Of course, in general multiset covers are not unique. For example, t_2 could also be covered by s_1 instead of s_2.

Definition 6.3 (multiset cover)**.** *Let $\bar{s} = \langle s_1, \dots s_n\rangle$ and $\bar{t} = \langle t_1, \dots t_m\rangle$ be tuples of terms. A* multiset cover *(γ, ε) is a pair of mappings $\gamma : \{1, \dots, m\} \to \{1, \dots, n\}$ and $\varepsilon : \{1, \dots, n\} \to \{\mathit{false}, \mathit{true}\}$ such that for each $1 \le i \le n$, if $\varepsilon(i)$ (indicating equality) then $\{j \mid \gamma(j) = i\}$ is a singleton set.*

For $\bar{s} = \langle s_1, \dots s_n\rangle$ and $\bar{t} = \langle t_1, \dots t_m\rangle$ we define that $\bar{s} \succsim^{mul} \bar{t}$ if there exists a multiset cover (γ, ε) such that $\gamma(j) = i$ implies that either: $\varepsilon(i) = \mathit{true}$ and $s_i \sim t_j$, or $\varepsilon(i) = \mathit{false}$ and $s_i \succ t_j$.

Definition 6.4 (multiset extension)**.** *Let $\succsim$ be a quasi-order on terms. The* multiset extensions *of $\succsim$, $\succ$, and $\sim$ are defined on tuples of terms:*

- *$\langle s_1, \dots, s_n\rangle \succsim^{mul} \langle t_1, \dots, t_m\rangle$ if and only if there exists a multiset cover (γ, ε) such that for all i, j, $\gamma(j) = i \;\Rightarrow\;$ (`if` $\varepsilon(i)$ `then` $s_i \sim t_j$ `else` $s_i \succ t_j$).*
- *$\langle s_1, \dots, s_n\rangle \succ^{mul} \langle t_1, \dots, t_m\rangle$ if and only if $\langle s_1, \dots, s_n\rangle \succsim^{mul} \langle t_1, \dots, t_m\rangle$ and for some i, $\neg\varepsilon(i)$, i.e., some s_i is not used for equality but rather replaced by zero or more smaller arguments t_j.*
- *$\langle s_1, \dots, s_n\rangle \sim^{mul} \langle t_1, \dots, t_m\rangle$ if and only if $\langle s_1, \dots, s_n\rangle \succsim^{mul} \langle t_1, \dots, t_m\rangle$, $n = m$, and for all i, $\varepsilon(i)$, i.e., all s_i are used to cover some t_j by equality.*

Let $\geq_\Sigma$ denote a quasi-order (a so-called *precedence*) on the set of function symbols Σ and let $>_\Sigma \;=\; (\geq_\Sigma \setminus \leq_\Sigma)$ and $\approx_\Sigma \;=\; (\geq_\Sigma \cap \leq_\Sigma)$. Then $\geq_\Sigma$ induces corresponding lexicographic and multiset path orders on terms.

Definition 6.5 (lexicographic and multiset path orders)**.** *For a precedence $\geq_\Sigma$ and $\rho \in \{\mathit{lpo}, \mathit{mpo}\}$ we define the relations $\succ_\rho$ and $\sim_\rho$ on terms. We use the notation $\bar{s} = \langle s_1, \dots s_n\rangle$ and $\bar{t} = \langle t_1, \dots t_m\rangle$.*

- *$s \succ_\rho t$ if, and only if, $s = f(\bar{s})$ and one of the following holds:*

  (1) $s_i \succ_\rho t$ or $s_i \sim_\rho t$ for some $1 \le i \le n$; or

  (2) $t = g(\bar{t})$ and $s \succ_\rho t_j$ for all $1 \le j \le m$ and either:
  (i) $f >_\Sigma g$ or (ii) $f \approx_\Sigma g$ and $\bar{s} \succ_\rho^{ext} \bar{t}$;

- *$s \sim_\rho t$ if, and only if, (a) $s = t$; or (b) $s = f(\bar{s})$, $t = g(\bar{t})$, $f \approx_\Sigma g$, and $\bar{s} \sim_\rho^{ext} \bar{t}$;*

where $\succ_\rho^{ext}$ and $\sim_\rho^{ext}$ are the lexicographic or multiset extensions of $\succ_\rho$ and $\sim_\rho$ for the respective cases when $\rho = \mathit{lpo}$ and $\rho = \mathit{mpo}$.

Example 6.6. Consider the following three TRSs for adding numbers:

$$\text{(a)} \ \left\{ \ \mathsf{plus}(\mathsf{0}, y) \to y \ , \quad \mathsf{plus}(\mathsf{s}(x), y) \to \mathsf{plus}(x, \mathsf{s}(y)) \ \right\}$$

$$\text{(b)} \ \left\{ \ \mathsf{plus}(x, \mathsf{0}) \to x \ , \quad \mathsf{plus}(x, \mathsf{s}(y)) \to \mathsf{s}(\mathsf{plus}(y, x)) \ \right\}$$

$$\text{(c)} \ \left\{ \ \mathsf{plus}(x, \mathsf{0}) \to x \ , \quad \mathsf{plus}(x, \mathsf{s}(y)) \to \mathsf{plus}(\mathsf{s}(x), y) \ \right\}$$

Example (a) is LPO-terminating for the precedence $\mathsf{plus} >_\Sigma \mathsf{s}$, but not MPO-terminating for any precedence. Example (b) is MPO-terminating for $\mathsf{plus} >_\Sigma \mathsf{s}$, but not LPO-terminating for any precedence as the second rule swaps x and y. Example (c) is neither LPO- nor MPO-terminating. However, termination could be proved using a path order where lexicographic comparison proceeds from right to left instead of left to right. The following definitions extend this observation to arbitrary permutations of the order in which we compare arguments.

As remarked before, the RPO combines such an extension of the LPO with MPO. This combination is facilitated by a *status function* which indicates for each function symbol if its arguments are to be compared based on a multiset extension or based on a lexicographic extension using some permutation μ.

Definition 6.7 (status function)**.** *A* status function σ *maps each symbol* $f \in \Sigma$ *of arity* n *either to the symbol* ***mul*** *or to a permutation* μ_f *on* $\{1, \dots, n\}$*.*

Definition 6.8 (recursive path order with status)**.** *For a precedence* $\geq_\Sigma$ *and status function* σ *we define the relations* $\succ_{rpo}$ *and* $\sim_{rpo}$ *on terms. We use the notation* $\bar{s} = \langle s_1, \dots s_n \rangle$ *and* $\bar{t} = \langle t_1, \dots t_m \rangle$*.*

- $s \succ_{rpo} t$ *if, and only if,* $s = f(\bar{s})$ *and one of the following holds:*
  (1) $s_i \succ_{rpo} t$ *or* $s_i \sim_{rpo} t$ *for some* $1 \leq i \leq n$*; or*
  (2) $t = g(\bar{t})$ *and* $s \succ_{rpo} t_j$ *for all* $1 \leq j \leq m$ *and either:*
  (i) $f >_\Sigma g$ *or* *(ii)* $f \approx_\Sigma g$ *and* $\bar{s} \succ^{f,g}_{rpo} \bar{t}$*;*
- $s \sim_{rpo} t$ *if, and only if, (a)* $s = t$*; or (b)* $s = f(\bar{s})$*,* $t = g(\bar{t})$*,* $f \approx_\Sigma g$*, and* $\bar{s} \sim^{f,g}_{rpo} \bar{t}$*;*

where $\succ^{f,g}_{rpo}$ *and* $\sim^{f,g}_{rpo}$ *are the tuple extensions of* $\succ_{rpo}$ *and* $\sim_{rpo}$ *defined by:*

- $\langle s_1, \dots s_n \rangle \succ^{f,g}_{rpo} \langle t_1, \dots t_m \rangle$ *if, and only if, one of the following holds:*
  (1) σ *maps* f *and* g *to permutations* μ_f *and* μ_g*; and*
  $\mu_f \langle s_1, \dots, s_n \rangle \succ^{lex}_{rpo} \mu_g \langle t_1, \dots, t_m \rangle$*;*
  (2) σ *maps* f *and* g *to* ***mul****; and* $\langle s_1, \dots s_n \rangle \succ^{mul}_{rpo} \langle t_1, \dots t_m \rangle$*.*
- $\langle s_1, \dots s_n \rangle \sim^{f,g}_{rpo} \langle t_1, \dots t_m \rangle$ *if, and only if, one of the following holds:*

(1) σ maps f and g to μ_f and μ_g; and $\mu_f\langle s_1, \ldots, s_n\rangle \sim^{lex}_{rpo} \mu_g\langle t_1, \ldots, t_m\rangle$;

(2) σ maps f and g to $\texttt{mul}$; and $\langle s_1, \ldots s_n\rangle \sim^{mul}_{rpo} \langle t_1, \ldots t_m\rangle$.

Def. 6.8 can be specialized to capture the previous path orders by taking specific forms of status functions: LPO when σ maps all symbols to the identity permutation; lexicographic path order w.r.t. permutation (LPOS) when σ maps all symbols to some permutation; MPO when σ maps all symbols to $\texttt{mul}$.

The RPO termination problem is to determine for a given TRS if there exists a precedence and a status function such that the system is RPO-terminating. There are two variants of the problem: "strict-" and "quasi-RPO termination" depending on whether the precedence $\geq_\Sigma$ is strict or not (i.e., on whether $f \approx_\Sigma g$ can hold for $f \neq g$). The corresponding decision problems, strict- and quasi-RPO termination, are decidable and NP-complete [CT94]. In this chapter we address the implementation of decision procedures for RPO termination problems by encoding them into corresponding SAT problems.

6.2. Encoding RPO problems

We introduce an encoding τ which maps constraints of the form $s \succ_{rpo} t$ to propositional statements about the status and the precedence of the symbols in the terms s and t. A satisfying assignment for the encoding of such a constraint indicates a precedence and a status function such that the constraint holds.

The first part of the encoding is straightforward and similar to the one in [CLS06] and our work for lexicographic path orders [CSL+06]. All "missing" cases (e.g., $\tau(x \succ_{rpo} t)$ for variables x) are defined to be *false*. The encodings for $\succ^{f,g}_{rpo}$ and $\sim^{f,g}_{rpo}$ are defined later in this section.

$$\tau(f(\bar{s}) \succ_{rpo} t) = \bigvee_{i=1}^{n} (\tau(s_i \succ_{rpo} t) \vee \tau(s_i \sim_{rpo} t)) \quad \vee \quad \tau_2(f(\bar{s}) \succ_{rpo} t) \tag{1}$$

$$\tau_2(f(\bar{s}) \succ_{rpo} g(\bar{t})) = \bigwedge_{j=1}^{m} \tau(f(\bar{s}) \succ_{rpo} t_j) \wedge \left(\begin{array}{l} (f >_\Sigma g) \vee \\ ((f \approx_\Sigma g) \wedge \tau(\bar{s} \succ^{f,g}_{rpo} \bar{t})) \end{array} \right) \tag{2}$$

$$\tau(s \sim_{rpo} s) = \mathit{true} \tag{3}$$

$$\tau(f(\bar{s}) \sim_{rpo} g(\bar{t})) = (f \approx_\Sigma g) \wedge \tau(\bar{s} \sim^{f,g}_{rpo} \bar{t}) \tag{4}$$

The propositional encoding of statements about precedences of the form $f >_\Sigma g$ or $f \approx_\Sigma g$ is performed following the approach applied in [CLS06].

Let $|\Sigma| = m$. The basic idea is to interpret the symbols in Σ as indices in a partial order taking finite domain values from the set $\{0, \ldots, m-1\}$. Each symbol $f \in \Sigma$ is thus modeled as $\langle f_k, \ldots, f_1\rangle$ with f_k the most significant bit and $k = \lceil \log_2 m \rceil$. The binary value of $\langle f_k, \ldots, f_1\rangle$ represents the position of f in the partial order. Of course,

$f_k, \ldots, f_1$ may be equal to $g_k, \ldots, g_1$ for $f \neq g$, if a (possibly strict) partial order imposes no order between f and g, or if a non-strict partial order imposes $f \approx_\Sigma g$. Constraints of the form $(f >_\Sigma g)$ or $(f \approx_\Sigma g)$ on Σ are interpreted as constraints on indices and it is straightforward to encode them in k-bit arithmetic: A constraint of the form $(f \approx_\Sigma g)$ is encoded in k bits by

$$\|(f \approx_\Sigma g)\|_k = \bigwedge_{1 \leq i \leq k} (f_i \leftrightarrow g_i).$$

A constraint of the form $(f >_\Sigma g)$ is encoded in k bits by

$$\|(f >_\Sigma g)\|_k = \begin{cases} (f_1 \wedge \neg g_1) & \text{if } k = 1 \\ (f_k \wedge \neg g_k) \vee ((f_k \leftrightarrow g_k) \wedge \|(f > g)\|_{k-1}) & \text{if } k > 1 \end{cases}$$

Now, we show how to encode lexicographic comparisons w.r.t. permutations. Then we continue with an encoding for multiset comparisons. Finally, we combine these two encodings into $\succ_{rpo}^{f,g}$ and $\sim_{rpo}^{f,g}$.

Encoding Lexicographic Comparisons w.r.t. Permutation

For lexicographic comparisons with permutations, we associate with each symbol $f \in \Sigma$ (of arity n) a permutation μ_f encoded through n^2 propositional variables $f_{i,k}$ with $i, k \in \{1, \ldots, n\}$. Here, $f_{i,k}$ is *true* if, and only if, $\mu_f(i) = k$ (i.e., the i-th argument of $f(s_1, \ldots, s_n)$ is considered at k-th position when comparing lexicographically). To ease presentation, we define that $f_{i,k}$ is *false* for $k > n$.

For the encoding to be correct, we impose constraints on the variables $f_{i,k}$ to ensure that they indeed correspond to a permutation on $\{1, \ldots, n\}$. So for each $i \in \{1, \ldots, n\}$ there must be exactly one $k \in \{1, \ldots, n\}$ and for each $k \in \{1, \ldots, n\}$ there must be exactly one $i \in \{1, \ldots, n\}$ such that $f_{i,k}$ is *true*.

We denote by $one(b_1, \ldots, b_n)$ the constraint expressing that exactly one of the bits $b_1, \ldots, b_n$ is *true*. Then our encoding includes a formula of the form

$$\bigwedge_{f/n \in \Sigma} \left(\bigwedge_{i=1}^{n} one(f_{i,1}, \ldots, f_{i,n}) \quad \wedge \quad \bigwedge_{k=1}^{n} one(f_{1,k}, \ldots, f_{n,k}) \right) \tag{5}$$

Here, we apply a linear encoding for constraints of the form $one(b_1, \ldots, b_n)$ which introduces $\approx 2n$ fresh Boolean variables which we denote here as $one_{b_i, \ldots, b_n}$ (expressing that one of the variables $b_i, \ldots, b_n$ is *true*) and $zero_{b_i, \ldots, b_n}$ (expressing that all of the variables $b_i, \ldots, b_n$ are *false*) for $1 < i \leq n$. The encoding applies a ternary propositional connective

$x \to y\,; z$ (denoting `if-then-else`) that is equivalent to $(x \to y) \wedge (\neg x \to z)$:

$$\bigwedge_{1 \le i \le n} \left(\begin{array}{l} \left(one_{b_i,\dots,b_n} \leftrightarrow \left(\begin{array}{ccc} (b_i & \to & zero_{b_{i+1},\dots,b_n} \\ & ; & one_{b_{i+1},\dots,b_n} \end{array} \right) \right) \\ \wedge \left(zero_{b_i,\dots,b_n} \leftrightarrow \neg b_i \wedge zero_{b_{i+1},\dots,b_n} \right) \end{array} \right)$$

where $one_{b_{n+1},\dots,b_n} = \mathit{false}$ and $zero_{b_{n+1},\dots,b_n} = \mathit{true}$.

This encoding introduces $\approx 2n$ conjuncts, each involving a formula with at most 4 Boolean variables. So the encoding is more concise than the more straightforward one which introduces a quadratic number of conjuncts $\neg b_i \vee \neg b_j$ for all $1 \le i < j \le n$.

Now consider the encoding of $\bar{s} \sim_{rpo}^{f,g} \bar{t}$ and $\bar{s} \succ_{rpo}^{f,g} \bar{t}$ for the case where the arguments of f and g are compared lexicographically (thus, we use the notations $\sim_{lex}^{f,g}$ and $\succ_{lex}^{f,g}$). Like in Definition 6.8, let $\bar{s} = \langle s_1, \dots, s_n \rangle$ and $\bar{t} = \langle t_1, \dots, t_m \rangle$. Now equality constraints of the form $\bar{s} \sim_{lex}^{f,g} \bar{t}$ are encoded by stating that for all k, the arguments s_i and t_j used at the k-th position in the comparison (as denoted by $f_{i,k}$ and and $g_{j,k}$) must be equal. This implies that $\bar{s} \sim_{lex}^{f,g} \bar{t}$ only holds if $n = m$.

$$\tau(\bar{s} \sim_{lex}^{f,g} \bar{t}) \quad = \quad (n = m) \wedge \left(\bigwedge_{k=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} f_{i,k} \wedge g_{j,k} \to \tau(s_i \sim_{rpo} t_j) \right) \qquad (6)$$

To encode $\bar{s} \succ_{lex}^{f,g} \bar{t}$, we define auxiliary relations $\succ_{lex}^{f,g,k}$, where $k \in \mathbb{N}$ denotes that the k-th component of $\bar{s}$ and $\bar{t}$ is currently being compared. Thus $\succ_{lex}^{f,g} = \succ_{lex}^{f,g,1}$, since the comparison starts with the first component. For any k, there are three cases to consider when encoding $\bar{s} \succ_{lex}^{f,g,k} \bar{t}$. If there is no s_i that can be used for the k-th comparison (i.e., $k > n$), then we encode to *false*. If there is such an s_i but no such t_j (i.e., $m < k \le n$), then we encode to *true*. If there are both an s_i and a t_j used for the k-th comparison (i.e., $f_{i,k}$ and $g_{j,k}$ hold), then we encode to a disjunction that either s_i is greater than t_j, or s_i is equal to t_j, and we continue the encoding at position $k+1$. Since exactly one $f_{i,k}$ is *true*, the disjunction and conjunction over all $f_{i,k}$ (with $i \in \{1, ..., n\}$) coincide (similar for $g_{j,k}$). Here, we use a disjunction of conjunctions, as this will be more convenient when introducing argument filters in Chapter 7.

$$\tau(\bar{s} \succ_{lex}^{f,g,k} \bar{t}) \quad = \quad \begin{cases} \mathit{false} & \text{if } k > n \\ \mathit{true} & \text{if } m < k \le n \\ \bigvee_{i=1}^{n} \Big(f_{i,k} \wedge \Big(\bigwedge_{j=1}^{m} g_{j,k} \to & \text{otherwise} \\ \quad (\tau(s_i \succ_{rpo} t_j) \vee (\tau(s_i \sim_{rpo} t_j) \wedge \tau(\bar{s} \succ_{lex}^{f,g,k+1} \bar{t}))) \Big) \Big) & \end{cases} \qquad (7)$$

Example 6.9. Consider again the TRS of Example 6.6(c):

$$\{ \ \mathsf{plus}(x, \mathsf{0}) \to x \ , \quad \mathsf{plus}(x, \mathsf{s}(y)) \to \mathsf{plus}(\mathsf{s}(x), y) \ \}$$

In the encoding of the constraints for the second rule, we have to encode the comparison $\langle x, \mathsf{s}(y)\rangle \succ_{lex}^{\mathsf{plus},\mathsf{plus},1} \langle \mathsf{s}(x), y\rangle$, which yields:

$$\begin{array}{lll}
 & \Big(\mathsf{plus}_{1,1} & \wedge \Big(\mathsf{plus}_{1,1} \rightarrow \Big(\tau(x \succ_{rpo} \mathsf{s}(x)) \vee (\tau(x \sim_{rpo} \mathsf{s}(x)) \wedge \tau(\langle x, \mathsf{s}(y)\rangle \succ_{lex}^{\mathsf{plus},\mathsf{plus},2} \langle \mathsf{s}(x), y\rangle))\Big)\Big) \\
 & & \wedge \Big(\mathsf{plus}_{2,1} \rightarrow \Big(\tau(x \succ_{rpo} y) \vee (\tau(x \sim_{rpo} y) \wedge \tau(\langle x, \mathsf{s}(y)\rangle \succ_{lex}^{\mathsf{plus},\mathsf{plus},2} \langle \mathsf{s}(x), y\rangle))\Big)\Big)\ \Big) \\
\vee & \Big(\mathsf{plus}_{2,1} & \wedge \Big(\mathsf{plus}_{1,1} \rightarrow \Big(\tau(\mathsf{s}(y) \succ_{rpo} \mathsf{s}(x)) \vee (\tau(\mathsf{s}(y) \sim_{rpo} \mathsf{s}(x)) \wedge \tau(\langle x, \mathsf{s}(y)\rangle \succ_{lex}^{\mathsf{plus},\mathsf{plus},2} \langle \mathsf{s}(x), y\rangle))\Big)\Big) \\
 & & \wedge \Big(\mathsf{plus}_{2,1} \rightarrow \Big(\tau(\mathsf{s}(y) \succ_{rpo} y) \vee (\tau(\mathsf{s}(y) \sim_{rpo} y) \wedge \tau(\langle x, \mathsf{s}(y)\rangle \succ_{lex}^{\mathsf{plus},\mathsf{plus},2} \langle \mathsf{s}(x), y\rangle))\Big)\Big)\ \Big)
\end{array}$$

Seeing that $\tau(x \succ_{rpo} \mathsf{s}(x)) = \tau(x \sim_{rpo} \mathsf{s}(x)) = \tau(x \succ_{rpo} y) = \tau(x \sim_{rpo} y) = \tau(\mathsf{s}(y) \succ_{rpo} \mathsf{s}(x)) = \tau(\mathsf{s}(y) \sim_{rpo} \mathsf{s}(x)) = \mathit{false}$ and $\tau(\mathsf{s}(y) \succ_{rpo} y) = \mathit{true}$, we can simplify the above formula to $\mathsf{plus}_{2,1} \wedge \neg\mathsf{plus}_{1,1}$. Together with the constraint (5) which ensures that the variables $\mathsf{plus}_{i,k}$ specify a valid permutation μ_{plus}, this implies that $\mathsf{plus}_{1,2}$ and $\neg\mathsf{plus}_{2,2}$ must be *true*. And indeed, for the permutation $\mu_{\mathsf{plus}} = \langle 2, 1\rangle$ the tuple $\mu_{\mathsf{plus}}(\langle x, \mathsf{s}(y)\rangle) = \langle \mathsf{s}(y), x\rangle$ is greater than the tuple $\mu_{\mathsf{plus}}(\langle \mathsf{s}(x), y\rangle) = \langle y, \mathsf{s}(x)\rangle$.

Encoding Multiset Comparisons

For multiset comparisons, we associate $\bar{s}$ and $\bar{t}$ with a multiset cover (γ, ε) encoded by $n * m$ propositional variables $\gamma_{i,j}$ and n variables ε_i. Here, $\gamma_{i,j}$ is *true* if, and only if, $\gamma(j) = i$ (s_i covers t_j) and ε_i is *true* if, and only if, $\varepsilon(i) = \mathit{true}$ (s_i is used for equality).

For the encoding to be correct, we again have to impose constraints on these variables to ensure that (γ, ε) indeed forms a multiset cover. So for each $j \in \{1, \ldots, m\}$ there must be exactly one $i \in \{1, \ldots, n\}$ such that $\gamma_{i,j}$ is *true*, and for each $i \in \{1, \ldots, n\}$, if ε_i is *true* then there must be exactly one $j \in \{1, \ldots, m\}$ such that $\gamma_{i,j}$ is *true*. Thus, our encoding includes the following formula:

$$\bigwedge_{j=1}^{m} \mathit{one}(\gamma_{1,j}, \ldots, \gamma_{n,j}) \qquad \wedge \qquad \bigwedge_{i=1}^{n} (\varepsilon_i \rightarrow \mathit{one}(\gamma_{i,1}, \ldots, \gamma_{i,m})) \tag{8}$$

Now we encode $\bar{s} \succ_{rpo}^{f,g} \bar{t}$ for the case where f and g have multiset status. To have an analogous notation to the case of lexicographic comparisons, we use the notation $\succ_{mul}^{f,g}$ instead of $\succ_{rpo}^{mul}$. The encoding of $\succsim_{mul}^{f,g}$, $\succ_{mul}^{f,g}$, and $\sim_{mul}^{f,g}$ is similar to Definition 6.4. To encode $\bar{s} \succsim_{mul}^{f,g} \bar{t}$, one has to require that if $\gamma_{i,j}$ and ε_i are *true*, $s_i \sim_{rpo} t_j$ holds, and else, if $\gamma_{i,j}$ is *true* and ε_i is not, $s_i \succ_{rpo} t_j$ holds. For $\succ_{mul}^{f,g}$, we must have at least one s_i that is not used for equality, and for $\sim_{mul}^{f,g}$, all s_i must be used for equality.

$$\begin{aligned}
\tau(\bar{s} \succsim_{mul}^{f,g} \bar{t}) &= \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} (\gamma_{i,j} \rightarrow (\varepsilon_i \rightarrow \tau(s_i \sim_{rpo} t_j); \tau(s_i \succ_{rpo} t_j))) && (9) \\
\tau(\bar{s} \succ_{mul}^{f,g} \bar{t}) &= \tau(\bar{s} \succsim_{mul}^{f,g} \bar{t}) \wedge \neg \bigwedge_{i=1}^{n} \varepsilon_i && (10) \\
\tau(\bar{s} \sim_{mul}^{f,g} \bar{t}) &= \tau(\bar{s} \succsim_{mul}^{f,g} \bar{t}) \wedge \bigwedge_{i=1}^{n} \varepsilon_i && (11)
\end{aligned}$$

Example 6.10. Consider again the rules for the TRS from Example 6.6(b):

$$\left\{\ \mathsf{plus}(x,\mathsf{0}) \to x\ ,\quad \mathsf{plus}(x,\mathsf{s}(y)) \to \mathsf{s}(\mathsf{plus}(y,x))\ \right\}$$

In the encoding of the constraints for the second rule, we have to encode the comparison $\langle x,\mathsf{s}(y)\rangle \succ^{f,g}_{mul} \langle y,x\rangle$, which yields:

$$\begin{array}{ll} & \Big(\gamma_{1,1} \to \Big((\varepsilon_1 \to \tau(x \sim_{rpo} y)) \wedge (\neg\varepsilon_1 \to \tau(x \succ_{rpo} y))\Big)\Big) \\ \wedge & \Big(\gamma_{1,2} \to \Big((\varepsilon_1 \to \tau(x \sim_{rpo} x)) \wedge (\neg\varepsilon_1 \to \tau(x \succ_{rpo} x))\Big)\Big) \\ \wedge & \Big(\gamma_{2,1} \to \Big((\varepsilon_2 \to \tau(\mathsf{s}(y) \sim_{rpo} y)) \wedge (\neg\varepsilon_2 \to \tau(\mathsf{s}(y) \succ_{rpo} y))\Big)\Big) \\ \wedge & \Big(\gamma_{2,2} \to \Big((\varepsilon_2 \to \tau(\mathsf{s}(y) \sim_{rpo} x)) \wedge (\neg\varepsilon_2 \to \tau(\mathsf{s}(y) \succ_{rpo} x))\Big)\Big) \\ \wedge & \neg(\varepsilon_1 \wedge \varepsilon_1) \end{array}$$

Seeing that $\tau(x \sim_{rpo} y) = \tau(x \succ_{rpo} y) = \tau(x \succ_{rpo} x) = \tau(\mathsf{s}(y) \sim_{rpo} y) = \tau(\mathsf{s}(y) \sim_{rpo} x) = \tau(\mathsf{s}(y) \succ_{rpo} x) = \mathit{false}$ and $\tau(x \sim_{rpo} x) = \tau(\mathsf{s}(y) \succ_{rpo} y) = \mathit{true}$, we can simplify the above formula to $\neg\gamma_{1,1} \wedge (\neg\gamma_{1,2} \vee \varepsilon_1) \wedge (\neg\gamma_{2,1} \vee \neg\varepsilon_2) \wedge \neg\gamma_{2,2} \wedge (\neg\varepsilon_1 \vee \neg\varepsilon_2)$. Together with the constraint (8) which ensures that the variables $\gamma_{i,j}$ and ε_i specify a valid multiset cover (γ,ε), this implies that $\gamma_{2,1}$, $\neg\varepsilon_2$, $\gamma_{1,2}$, and ε_1 must hold. And indeed, the multiset cover (γ,ε) with $\gamma(1) = 2$, $\gamma(2) = 1$, $\varepsilon(1) = \mathit{true}$, and $\varepsilon(2) = \mathit{false}$, shows that the tuple $\langle x,\mathsf{s}(y)\rangle$ is greater than the tuple $\langle y,x\rangle$.

Combining Lexicographic and Multiset Comparisons

We have shown how to encode lexicographic and multiset comparisons. In order to combine $\succ^{f,g}_{lex}$ and $\succ^{f,g}_{mul}$ into $\succ^{f,g}_{rpo}$ as well as $\sim^{f,g}_{lex}$ and $\sim^{f,g}_{mul}$ into $\sim^{f,g}_{rpo}$, we introduce for each symbol $f \in \Sigma$ a variable m_f, which is *true* if, and only if, the arguments of f are to be compared as multisets (i.e., the status function maps f to ***mul***).

$$\tau(\bar{s} \succ^{f,g}_{rpo} \bar{t}) \;=\; \Big(m_f \wedge m_g \wedge \tau(\bar{s} \succ^{f,g}_{mul} \bar{t})\Big) \vee \Big(\neg m_f \wedge \neg m_g \wedge \tau(\bar{s} \succ^{f,g,1}_{lex} \bar{t})\Big) \tag{12}$$

$$\tau(\bar{s} \sim^{f,g}_{rpo} \bar{t}) \;=\; \Big(m_f \wedge m_g \wedge \tau(\bar{s} \sim^{f,g}_{mul} \bar{t})\Big) \vee \Big(\neg m_f \wedge \neg m_g \wedge \tau(\bar{s} \sim^{f,g}_{lex} \bar{t})\Big) \tag{13}$$

Similar to Def. 6.8, the above encoding function τ can be specialized to other standard path orderings: lexicographic path order w.r.t. permutation (LPOS) when m_f is set to *false* for all $f \in \Sigma$; LPO when additionally $f_{i,k}$ is set to *true* if, and only if, $i = k$; MPO when m_f is set to *true* for all $f \in \Sigma$.

Size of the Encoding

We conclude this section with an approximation of the size of the propositional formula obtained when $s \succ_{rpo} t$ is encoded, where $s = f(s_1,\ldots,s_n)$ and $t = g(t_1,\ldots,t_m)$ with the total size of terms s and t being k.

A single step of unfolding Def. 6.8 results in a formula containing at least n copies of t (with all its subterms) and m copies of s (with all its subterms) occurring in constraints of the form $s' \succ_{rpo} t'$.

Hence, without memoing, the final encoding is clearly exponential in k. To obtain a polynomial encoding, we introduce sharing of common subformulas in the propositional formula. The approach is similar to that proposed by Tseitin to obtain a linear CNF transformation of Boolean formulas [Tse68].

If $\tau(s' \succ_{rpo} t')$ occurs in the encoding $\tau(s \succ_{rpo} t)$, then we do not immediately perform the encoding of $s' \succ_{rpo} t'$ as well. Instead, we introduce a fresh Boolean variable of the form $X_{s' \succ_{rpo} t'}$, and encode also the meaning of such fresh variables.

The encoding of $X_{s' \succ_{rpo} t'}$ is of the form $X_{s' \succ_{rpo} t'} \leftrightarrow \tau(s' \succ_{rpo} t')$. Again, when constructing $\tau(s' \succ_{rpo} t')$, we replace all subformulas $\tau(s'' \succ_{rpo} t'')$ encountered by Boolean variables $X_{s'' \succ_{rpo} t''}$.

In total, there are at most $\mathcal{O}(k^2)$ fresh Boolean variables to encode. As the encodings of multiset comparisons and lexicographic comparisons are both of size $\mathcal{O}(k^3)$, the size of the overall encoding is in $\mathcal{O}(k^5)$. Thus, the size of the encoding is indeed polynomial.

A finer analysis shows that not all multiset and lexicographic comparisons are large. For example, for $s = \mathsf{f}(\mathsf{a}_1, \ldots, \mathsf{a}_n)$ and $t = \mathsf{g}(\mathsf{b}_1, \ldots, \mathsf{b}_n)$ with constants a_i and b_j, there is one comparison of two n-tuples with encoding size $\mathcal{O}(n^3)$, but the other $n^2 + 2n$ comparisons only need size $\mathcal{O}(n)$ each. In fact, one can show that the size of the overall encoding is in $\mathcal{O}(k^3)$.

6.3. Implementation and Experiments

We implemented the encoding of RPO in the termination analyzer AProVE [GST06]. For analyzing satisfiability of our propositional formulas, we used the SAT4J solver [LP08]. (We also tried other SAT solvers like MiniSAT [ES07] and obtained similar results.) The encoding can be restricted to instances of RPO like LPO or MPO.

We tested the implementation on all 865 TRSs from the TPDB [TPD06]. The experiments were run on a 2.2 GHz AMD Athlon 64 with a time-out of 60 seconds (corresponding to the most common setting used in the international *Termination Competition* [MZ07]).

The following table compares our new SAT-based approach for direct application of path orders to the previous dedicated solvers for path orders in AProVE 1.2 which did not use SAT solving. The columns contain the data for LPO with strict and non-strict precedence (denoted *lpo*/*qlpo*), for LPO with status (*lpos*/*qlpos*), for MPO (*mpo*/*qmpo*), and for RPO with status (*rpo*/*qrpo*). For each encoding we give the number of TRSs which could be proved terminating (with the number of time-outs in brackets) and the analysis time (in seconds) for the full collection.

Solver	*lpo*	*qlpo*	*lpos*	*qlpos*	*mpo*	*qmpo*	*rpo*	*qrpo*
SAT	**123** (0)	**127** (0)	**141** (0)	**155** (0)	**92** (0)	**98** (0)	**146** (0)	**162** (0)
	31.0	**44.7**	**26.1**	**40.6**	**49.4**	**74.2**	**50.0**	**85.3**
ded.	**123** (5)	**127**(16)	**141** (6)	**154**(45)	**92** (7)	**98**(31)	**145**(10)	**158**(65)
	334.4	**1426.3**	**460.4**	**3291.7**	**653.2**	**2669.1**	**908.6**	**4708.2**

The table shows that with our new SAT encoding, performance improves by orders of magnitude over existing dedicated solvers for direct analysis with path orders. Note that without a time-out, this effect would be aggravated. While there are up to 65 time-outs for the dedicated solver, there are no time-outs at all for our SAT-based solver. Indeed, even for RPO with non-strict precedence, the average time per example is below 0.1 seconds. The table also shows that the use of RPO instead of LPO increases power substantially. This increase in power comes with a large penalty in runtime when the dedicated solver is used, while in the SAT-based setting, runtimes increase only mildly.

6.4. Summary

This chapter extends the SAT-based approach of [CLS06] and our work for lexicographic path orders [CSL$^+$06] to the more powerful class of recursive path orders. The main new challenges were the encoding of multiset comparisons as well as of lexicographic comparisons w.r.t. permutations.

The contributions of this chapter solve this problem by a novel SAT encoding which combines all of the constraints originating from these notions into a single search process. Through implementation and experimentation we showed that our encoding leads to speedups in orders of magnitude over existing termination tools as well as increased termination proving power. To experiment with our SAT-based implementation and for further details on our experiments in this chapter and in Chapter 7, please visit our evaluation web site at `http://aprove.informatik.rwth-aachen.de/eval/SATRPO/`.

Note that this kind of efficiency improvement is essential for the application of recursive path orders to TRSs resulting from logic programs by, for instance, our transformation from Chapter 3. In the following chapter, after combining the results of this chapter with dependency pairs and argument filters, we demonstrate that the application of recursive path orders to search problems originating from logic programs enables us to show termination for programs where all other techniques and tools for logic programming fail.

7. Argument Filters

In Chapter 6 we introduced a propositional encoding of recursive path orders and demonstrated that SAT solving can drastically speed up the solving of RPO termination problems. The key idea was that the encoding of a term rewrite system $\mathcal{R}$ is satisfiable if and only if $\mathcal{R}$ is RPO-terminating and that each model of the encoding indicates a particular RPO which orients the rules in $\mathcal{R}$.

However, recursive path orders on their own are too weak for many interesting termination problems, and hence RPO is typically combined with more sophisticated termination proving techniques. One of the most popular and powerful such techniques is the *dependency pair* (DP) method [AG00]. Essentially, for any TRS the DP method generates a set of inequalities between terms. If one can find a well-founded order satisfying these inequalities, then termination is proved. A main advantage of the DP method is that it permits the use of orders which need not be monotonic. This allows the application of recursive path orders combined with *argument filters*.

For every function symbol f, an argument filter π specifies which parts of a term $f(\ldots)$ may be eliminated before comparing terms. As stated in [HM05a], "the dependency pairs method derives much of its power from the ability to use argument filters to simplify constraints". However, argument filters represent a severe bottleneck for the automation of dependency pairs, as the search space for argument filters is enormous. In recent refinements of the DP method [GTSF03, TGS04, GTSF06], the choice of π also influences the set of *usable rules* which contribute to the inequalities that have to be oriented.

This chapter extends the approach of Chapter 6 by providing a propositional encoding which combines the search for an RPO with the search for an argument filter. This extension is non-trivial as the choice of an argument filter π influences the structure of the terms in the rules as well as the set of rules which contribute to the inequalities that need to be oriented. The key idea is to combine all of the constraints on π which influence the definition of the RPO and the definition of the usable rules and to encode these constraints in SAT. This encoding captures the synergy between precedences on function symbols and argument filters. In our approach there exist an argument filter π and an RPO which orient a set of inequalities if and only if the encoding of the inequalities is satisfiable. Moreover, each model of the encoding corresponds to a suitable argument filter and a suitable RPO which orient the inequalities.

After the necessary preliminaries on the DP method in Section 7.1, Section 7.2 extends the approach of Chapter 6 to consider argument filters. Section 7.3 shows how to extend this encoding to account for the influence of an argument filter on the set of usable rules.

In Section 7.4 we describe the implementation of our results in the termination prover AProVE [GST06] and provide extensive experimental evidence indicating speedups in orders of magnitude. We summarize the contributions of this chapter in Section 7.5 where we also show how RPO and argument filtering can successfully be applied to termination analysis of logic programs.

7.1. Preliminaries

This section briefly describes the components of the dependency pair framework [AG00, GTS05a, GTSF06] needed to present the results of this chapter. Notions like *defined symbols*, *dependency pairs*, and *reduction pairs* are defined exactly as in Chapter 3.

It is well known that recursive path orders on their own are not very powerful for proving termination.

Example 7.1. Consider the following TRS $\mathcal{R}$ for division on natural numbers [AG00].

$$\begin{array}{rcll}
\mathsf{minus}(x,\mathsf{0}) & \to & x & (1)\\
\mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{minus}(x,y) & (2)\\
\mathsf{quot}(\mathsf{0},\mathsf{s}(y)) & \to & \mathsf{0} & (3)\\
\mathsf{quot}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x,y),\mathsf{s}(y))) & (4)
\end{array}$$

The rules (1) – (3) can easily be oriented using any RPO, but rule (4) cannot. To see this, observe that if we instantiate y by the term $\mathsf{s}(x)$, we obtain $\mathsf{quot}(\mathsf{s}(x),\mathsf{s}(\mathsf{s}(x))) \prec_{emb} \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x,\mathsf{s}(x)),\mathsf{s}(\mathsf{s}(x))))$. Thus, no simplification order and, consequently, no RPO can show termination of $\mathcal{R}$. This drawback was the reason for developing more powerful approaches like the dependency pair method.

The defined symbols of $\mathcal{R}$ are minus and quot, and there are three dependency pairs:

$$\begin{array}{rcll}
\mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{MINUS}(x,y) & (5)\\
\mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{MINUS}(x,y) & (6)\\
\mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{QUOT}(\mathsf{minus}(x,y),\mathsf{s}(y)) & (7)
\end{array}$$

The main result underlying the dependency pair method states that a term rewrite system $\mathcal{R}$ is terminating if and only if there is no infinite (minimal) $\mathcal{R}$-*chain* of its dependency pairs $DP(\mathcal{R})$ [AG00]. In contrast to Definitions 3.16 and 4.5, we try to prove termination of all terms and, consequently, do not need to integrate argument filters or sets of queries, respectively. In other words, there is no infinite sequence of dependency pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $DP(\mathcal{R})$ such that for all i there is a substitution σ_i where $t_i\sigma_i$ is terminating with respect to $\mathcal{R}$ and $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$. To prove absence of such infinite chains automatically, we consider so-called *dependency pair problems*. A dependency pair problem $(\mathcal{P}, \mathcal{R})$ is a pair of term rewrite systems $\mathcal{P}$ and $\mathcal{R}$ and poses the question: “Is there an infinite $\mathcal{R}$-chain of dependency pairs from $\mathcal{P}$?” The goal is to solve the dependency pair problem $(DP(\mathcal{R}), \mathcal{R})$ in order to determine termination of $\mathcal{R}$.

Termination techniques now operate on dependency pair problems and are called *DP processors*. Formally, a DP processor *Proc* takes a dependency pair problem as input and returns a new dependency pair problem which then has to be solved instead. A processor *Proc* is *sound* if for all dependency pair problems $(\mathcal{P}, \mathcal{R})$ where $Proc(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R})$, there is an infinite $\mathcal{R}$-chain of pairs from $\mathcal{P}'$ whenever there is an infinite $\mathcal{R}$-chain of pairs from $\mathcal{P}$. Soundness of a DP processor is required to prove termination and in particular, to conclude that there is no infinite $\mathcal{R}$-chain if $Proc(\mathcal{P}, \mathcal{R}) = (\varnothing, \mathcal{R})$.

So termination proofs in our framework start with the initial DP problem $(DP(\mathcal{R}), \mathcal{R})$. Then the DP problem is simplified repeatedly by sound DP processors. If one reaches the DP problem $(\varnothing, \mathcal{R})$, then termination is proved. In the following, we present one of the most important processors of the framework, the so-called *reduction pair processor* which was the inspiration for Theorems 3.24 and 4.20.

For a DP problem $(\mathcal{P}, \mathcal{R})$, the reduction pair processor generates inequality constraints which should be satisfied by a *reduction pair*. Note that in this context, $\succ$ need not be monotonic. A typical choice for a reduction pair $(\succsim, \succ)$ is to use simplification orders in combination with *argument filters* [AG00] (we again adopt the notation of [KNT99]).

Note that the following definition of argument filter is an extension of Definition 3.12 as it allows not only to filter arguments of a term, but also to collapse a term $f(t_1, \dots, t_n)$ to one of its arguments. If $\pi(f) = i$, then $\pi(f(t_1, \dots, t_n)) = t_i$.

Definition 7.2 (Argument Filter). *An* argument filter π *maps every n-ary function symbol to an argument position* $i \in \{1, \dots, n\}$ *or to a (possibly empty) list* $[i_1, \dots, i_p]$ *with* $1 \leq i_1 < \cdots < i_p \leq n$. *An argument filter* π *induces a mapping from terms to terms:*

$$\pi(t) = \begin{cases} t & \textit{if } t \textit{ is a variable} \\ \pi(t_i) & \textit{if } t = f(t_1, \dots, t_n) \textit{ and } \pi(f) = i \\ f(\pi(t_{i_1}), \dots, \pi(t_{i_p})) & \textit{if } t = f(t_1, \dots, t_n) \textit{ and } \pi(f) = [i_1, \dots, i_p] \end{cases}$$

For a relation $\succ$ *on terms, let* $\succ^\pi$ *be the relation where* $s \succ^\pi t$ *holds if and only if* $\pi(s) \succ \pi(t)$. *An argument filter with* $\pi(f) = i$ *is called* collapsing on f.

Arts and Giesl showed in [AG00] that if $(\succsim, \succ)$ is a reduction pair and π is an argument filter then $(\succsim^\pi, \succ^\pi)$ is also a reduction pair. In particular, we focus on reduction pairs of the form $(\succsim^\pi_{rpo}, \succ^\pi_{rpo})$ to prove termination of examples like Example 7.1 where the direct application of simplification orders fails.

The constraints generated by the reduction pair processor require that (a) all dependency pairs in $\mathcal{P}$ are weakly or strictly decreasing and, (b) all *usable* rules $\mathcal{U}(\mathcal{P}, \mathcal{R})$ are weakly decreasing. Here, a rule $f(\ldots) \to r$ from $\mathcal{R}$ is *usable* if f occurs in the right-hand side of a dependency pair from $\mathcal{P}$ or of a usable rule. In Example 7.1, the symbols occurring in the right-hand sides of the dependency pairs (5) – (7) are MINUS, QUOT, s, and minus. Therefore the minus-rules (1) and (2) are usable. Since the right-hand sides of the minus-rules do not contain additional symbols, these are in fact all of the usable rules. Hence, the quot-rules (3) and (4) are not usable.

As shown in [HM04, TGS04], under certain conditions on the reduction pair, Restriction (b) ensures that in chains $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$, we have $t_i\sigma_i \succsim s_{i+1}\sigma_{i+1}$. The required conditions hold in particular for any reduction pair constructed using simplification orders and argument filters and specifically for $(\succsim_{rpo}^{\pi}, \succ_{rpo}^{\pi})$. Hence, the strictly decreasing pairs of $\mathcal{P}$ cannot occur infinitely often in chains. This enables the processor to delete such pairs from $\mathcal{P}$. In the following, for any term rewrite system $\mathcal{Q}$ and relation $\succ$, we denote $\mathcal{Q}_{\succ} = \{s \to t \in \mathcal{Q} \mid s \succ t\}$.

Theorem 7.3 (Reduction Pair Processor)**.** *Let $(\succsim, \succ)$ be a reduction pair for a simplification order $\succ$ and let π be an argument filter. Then the following DP processor Proc is sound.*

$$Proc(\mathcal{P},\mathcal{R}) = \begin{cases} (\mathcal{P} \setminus \mathcal{P}_{\succ^\pi}, \mathcal{R}) & \textit{if } \mathcal{P}_{\succ^\pi} \cup \mathcal{P}_{\succsim^\pi} = \mathcal{P} \textit{ and } \mathcal{R}_{\succsim^\pi} \supseteq \mathcal{U}(\mathcal{P},\mathcal{R}) \\ (\mathcal{P},\mathcal{R}) & \textit{otherwise} \end{cases}$$

Example 7.4. For the TRS of Example 7.1, according to Theorem 7.3 we search for a reduction pair solving the following inequality constraints (where $\underset{(\sim)}{\succ}$ denotes $\succ \cup \succsim$):

$$\begin{array}{rcll}
\mathsf{minus}(x,\mathsf{0}) & \succsim & x & \\
\mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) & \succsim & \mathsf{minus}(x,y) & \\
\mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{MINUS}(x,y) & (8)\\
\mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{MINUS}(x,y) & (9)\\
\mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{QUOT}(\mathsf{minus}(x,y),\mathsf{s}(y)) & (10)
\end{array}$$

By Theorem 7.3, all dependency pairs corresponding to strictly decreasing inequalities (8) – (10) can be removed. To solve the inequalities we may take $(\succsim_{rpo}^{\pi}, \succ_{rpo}^{\pi})$ where $\pi(\mathsf{minus})=1$, $\pi(\mathsf{s})=\pi(\mathsf{MINUS})=\pi(\mathsf{QUOT})=[1]$, and where $\succsim_{rpo}$ and $\succ_{rpo}$ are induced by the partial order $\mathsf{QUOT} >_{\Sigma} \mathsf{MINUS}$. Thus, the constraints after applying π are the following:

$$\begin{array}{rcl}
x & \succsim & x \\
\mathsf{s}(x) & \succsim & x \\
\mathsf{MINUS}(\mathsf{s}(x)) & \underset{(\sim)}{\succ} & \mathsf{MINUS}(x) \\
\mathsf{QUOT}(\mathsf{s}(x)) & \underset{(\sim)}{\succ} & \mathsf{MINUS}(x) \\
\mathsf{QUOT}(\mathsf{s}(x)) & \underset{(\sim)}{\succ} & \mathsf{QUOT}(x)
\end{array}$$

Thus, after filtering, inequalities (8) – (10) are all strict for any simplification order and, consequently, any RPO. Hence, they are removed by the reduction pair processor. This results in the new DP problem $(\varnothing, \mathcal{R})$ which proves termination of Example 7.1.

As an additional example we consider the following variant of addition.

Example 7.5. Consider the TRS (on the left) for addition using an accumulator and its three dependency pairs (on the right):

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathsf{plus}(x,y) & \to & \mathsf{plus}'(x,y,\mathsf{0}) & \mathsf{PLUS}(x,y) & \to & \mathsf{PLUS}'(x,y,\mathsf{0}) \\
\mathsf{plus}'(\mathsf{0},\mathsf{0},z) & \to & z & & & \\
\mathsf{plus}'(\mathsf{s}(x),y,z) & \to & \mathsf{plus}'(x,y,\mathsf{s}(z)) & \mathsf{PLUS}'(\mathsf{s}(x),y,z) & \to & \mathsf{PLUS}'(x,y,\mathsf{s}(z)) \\
\mathsf{plus}'(x,\mathsf{s}(y),z) & \to & \mathsf{plus}'(y,x,\mathsf{s}(z)) & \mathsf{PLUS}'(x,\mathsf{s}(y),z) & \to & \mathsf{PLUS}'(y,x,\mathsf{s}(z))
\end{array}$$

The rule $\mathsf{PLUS}'(\mathsf{s}(x),y,z) \to \mathsf{PLUS}'(x,y,\mathsf{s}(z))$ cannot be oriented by RPO. Lexicographic comparison fails as the first two arguments are swapped. Multiset comparison is prevented by the third argument ($\mathsf{s}(z)$ cannot be covered). But an argument filter $\pi(\mathsf{PLUS}') = [1,2]$, which eliminates the third argument of PLUS', enables the multiset comparison.

To orient all DPs we use $(\succsim^{\pi}_{rpo}, \succ^{\pi}_{rpo})$ where $\pi(\mathsf{PLUS}){=}\pi(\mathsf{PLUS}'){=}[1,2]$, $\pi(\mathsf{s}){=}[1]$, and where $\succsim_{rpo}$ and $\succ_{rpo}$ are induced by the precedence $\mathsf{PLUS} >_{\Sigma} \mathsf{PLUS}'$ and the status function σ that maps PLUS and PLUS' to *mul*. Since the problematic third accumulator argument (in the last two DPs) is filtered away, all three DPs are strictly decreasing and can be removed, as there are no usable rules. This results in the DP problem $(\varnothing, \mathcal{R})$ which proves termination for the TRS.

Note that while argument filters are very powerful in the context of the DP framework, they also present a severe bottleneck for automation, as the search space for argument filters is enormous (exponential in the arities of the function symbols).

We conclude this brief description of the required components of the dependency pair framework with a statement of the central *decision problem* associated with argument filters, RPO, and dependency pairs:

> For a given dependency pair problem $(\mathcal{P}, \mathcal{R})$, does there exist a reduction pair $(\succsim^{\pi}_{rpo}, \succ^{\pi}_{rpo})$ for some argument filter π and lexicographic path order induced by some partial order $\geq_{\Sigma}$ such that all rules in $\mathcal{P}$ and in $\mathcal{R}$ are weakly decreasing and at least one rule in $\mathcal{P}$ is strictly decreasing?

In the following section we show how to encode constraints like "$s \succ^{\pi}_{rpo} t$" and "$s \succsim^{\pi}_{rpo} t$" as propositional formulas. Such an encoding enables us to encode the decision problem stated above as a SAT problem. Based on the solution of the SAT problem one can then identify the dependency pairs which can be removed from $\mathcal{P}$.

7.2. Encoding RPO with Argument Filters

In this section we consider recursive path orders with argument filters and the corresponding decision problem. Consider first a naive brute force approach. For any given argument filter π we generate the formula

$$\bigwedge_{\ell\to r\in\mathcal{U}(\mathcal{P},\mathcal{R})} \pi(\ell) \succsim_{rpo} \pi(r) \quad\wedge\quad \bigwedge_{s\to t\in\mathcal{P}} \pi(s) \succsim_{rpo} \pi(t) \quad\wedge\quad \bigvee_{s\to t\in\mathcal{P}} \pi(s) \succ_{rpo} \pi(t) \tag{11}$$

The constraints "$\pi(s) \succsim_{rpo} \pi(t)$" and "$\pi(s) \succ_{rpo} \pi(t)$" can be encoded as described in Chapter 6. Then SAT solving can search for an RPO satisfying (11) for the given filter π. However, this approach is hopelessly inefficient, potentially calling the SAT solver for each of the exponentially many argument filters. Even if one considers the less naive enumeration algorithms implemented in [GST06] and [HM05b], for many examples the SAT solver would be called exponentially often.

A contribution of this chapter is to show instead how to encode the argument filters into the propositional formula and delegate the search for an argument filter to the SAT solver. In this way, the SAT solver is only called once with an encoding of Formula (11) and it can search for an argument filter and for a precedence at the same time. This is clearly advantageous, since the filter and the precedence highly influence each other.

So our goal is to encode constraints like "$s \succ^{\pi}_{rpo} t$" (or "$s \succsim^{\pi}_{rpo} t$") into propositional formulas such that every model of the encoding corresponds to a concrete filter π, a status σ and precedence $\geq_\Sigma$ which satisfy "$s \succ^{\pi}_{rpo} t$" (or "$s \succsim^{\pi}_{rpo} t$"). We first provide an explicit definition which then provides the basis for specifying constraints on precedences and argument filters, satisfaction of which gives "$s \succ^{\pi}_{lpo} t$" (or "$s \succsim^{\pi}_{lpo} t$"). The essential difference with Definition 6.8 is that all cases are refined to consider the effect of π.

Definition 7.6 (RPO modulo π). *For a precedence $\geq_\Sigma$, a status function σ, and an argument filter π on Σ we define the relations $\succ^{\pi}_{rpo}$, $\sim^{\pi}_{rpo}$, $\succsim^{\pi}_{rpo}$ on terms. Again, we use the notation $\bar{s} = \langle s_1, \ldots, s_n\rangle$ and $\bar{t} = \langle t_1, \ldots, t_m\rangle$.*

- *$s \succ^{\pi}_{rpo} t$ if, and only if, $s = f(\bar{s})$ and one of the following holds:*

  (1) (a) $\pi(f) = i$ and $s_i \succ^{\pi}_{rpo} t$; or
  (b) $\pi(f) = [i_1, \ldots, i_p]$ and for some $i \in [i_1, \ldots, i_p]$, $s_i \succ^{\pi}_{rpo} t$ or $s_i \sim^{\pi}_{rpo} t$; or

  (2) $t = g(\bar{t})$ and
  (a) $\pi(g) = j$ and $s \succ^{\pi}_{rpo} t_j$; or
  (b) $\pi(f) = [i_1, ..., i_p]$, $\pi(g) = [j_1, ..., j_q]$, $s \succ^{\pi}_{rpo} t_j$ for all $j \in [j_1, \ldots, j_q]$, and either (i) $f >_\Sigma g$ or
  (ii) $f \approx_\Sigma g$ and $\langle s_{i_1}, \ldots, s_{i_p}\rangle \succ^{f,g,\pi}_{rpo} \langle t_{j_1}, \ldots, t_{j_q}\rangle$.

- *$s \sim^{\pi}_{rpo} t$ if, and only if, one of the following holds:*

  (1) $s = t$; or

  (2) $s = f(\bar{s})$ and $\pi(f) = i$ and $s_i \sim^{\pi}_{rpo} t$; or

  (3) $t = g(\bar{t})$ and $\pi(g) = j$ and $s \sim^{\pi}_{rpo} t_j$; or

  (4) $s = f(\bar{s})$, $t = g(\bar{t})$, $\pi(f) = [i_1, ..., i_p]$, $\pi(g) = [j_1, ..., j_q]$,
  $$f \approx_{\Sigma} g, \text{ and } \langle s_{i_1}, \ldots, s_{i_p} \rangle \sim^{f,g,\pi}_{rpo} \langle t_{j_1}, \ldots, t_{j_q} \rangle.$$

- *$s \succsim^{\pi}_{rpo} t$ if, and only if, $s \succ^{\pi}_{rpo} t$ or $s \sim^{\pi}_{rpo} t$.*

where $\succ^{f,g,\pi}_{rpo}$ and $\sim^{f,g,\pi}_{rpo}$ are the tuple extensions of $\succ^{\pi}_{rpo}$ and $\sim^{\pi}_{rpo}$ defined analogously to $\succ^{f,g}_{rpo}$ and $\sim^{f,g}_{rpo}$ from Definition 6.8 w.r.t. $\succ_{rpo}$ and $\sim_{rpo}$.

It follows directly from Definitions 6.8, 7.2, and 7.6 that for all terms s and t we have $s \succ^{\pi}_{rpo} t \Leftrightarrow \pi(s) \succ_{rpo} \pi(t)$ and $s \succsim^{\pi}_{rpo} t \Leftrightarrow \pi(s) \succsim_{rpo} \pi(t)$.

The decision problem associated with Definition 7.6 is stated as follows: For terms s and t, does there exist a precedence $\geq_{\Sigma}$ and an argument filter π such that $s \succ^{\pi}_{rpo} t$ resp. $s \succsim^{\pi}_{rpo} t$ holds. This problem again comes in two flavors: "strict-RPO" and "quasi-RPO" depending on whether $\geq_{\Sigma}$ is required to be strict or not. Our aim is to encode these decision problems as constraints on $\geq_{\Sigma}$ and π, similar to the encoding of $s \succ_{rpo} t$ as a constraint on the precedence in Chapter 6. The difference is that now we have two types of constraints: constraints on the precedence $\geq_{\Sigma}$ and constraints on the argument filter π. To express constraints on argument filters we use atoms of the following forms: "$\pi(f) = i$" to constrain π to map f to the value i; "$\pi(f) \ni i$" to constrain π to map f either to a list containing i or to i itself; and "$list(\pi(f))$" to constrain π to map f to a list. So "$list(\pi(f))$" means that π is not collapsing on f.

To encode argument filters, each n-ary function symbol $f \in \Sigma$ is associated with n propositional variables $f_1, \ldots, f_n$, and another variable $list_f$. Here, f_i is *true* if, and only if, $\pi(f) \ni i$, and $list_f$ is *true* if, and only if, $list(\pi(f))$. To ensure that these $n+1$ propositional variables indeed correspond to an argument filter, we impose the following constraints which express that if π collapses f then it is replaced by exactly one[1] of its subterms:

$$\neg list_f \rightarrow one(f_1, \ldots, f_n).$$

To encode the combination of RPO with argument filters, consider again the equations (1) – (4) from Chapter 6. Each reference to a subterm must now be "wrapped" by the question: "has this subterm been filtered by π?" In the following, similar to the encoding of Section 6.2, all "missing" cases are defined to be *false*. Equations (1′) – (3′) enhance Equations (1) – (3) from Chapter 6. Equations (4a′) and (4b′) enhance Equation (4) from Chapter 6 in the cases when one of the terms is a variable x and (4c′)

[1]For *one* we use the encoding from Section 6.2.

considers the other case and examines whether the filter collapses the root symbols or not.

$$
\begin{aligned}
\tau(f(\bar{s}) \succ^{\pi}_{rpo} t) &= \bigvee_{i=1}^{n} \left(f_i \wedge \left(\begin{array}{l} \tau(s_i \succ^{\pi}_{rpo} t) \quad \vee \\ (list_f \wedge \tau(s_i \sim^{\pi}_{rpo} t)) \end{array} \right) \right) \vee \ \tau_2(f(\bar{s}) \succ^{\pi}_{rpo} t) && (1') \\
\tau_2(f(\bar{s}) \succ^{\pi}_{rpo} g(\bar{t})) &= \bigwedge_{j=1}^{m} \big(g_j \to \tau(f(\bar{s}) \succ^{\pi}_{rpo} t_j)\big) \quad \wedge && (2') \\
&\quad \Big(list_g \to \Big(list_f \wedge \Big((f >_{\Sigma} g) \vee ((f \approx_{\Sigma} g) \wedge \tau(\bar{s} \succ^{f,g,\pi}_{rpo} \bar{t}))\Big)\Big)\Big) \\
\tau(s \sim^{\pi}_{rpo} s) &= true && (3') \\
\tau(f(\bar{s}) \sim^{\pi}_{rpo} x) &= (\neg list_f \wedge \bigwedge_{i=1}^{n} \big(f_i \to \tau(s_i \sim^{\pi}_{rpo} x)\big) \quad \text{for variables } x && (4a') \\
\tau(x \sim^{\pi}_{rpo} g(\bar{t})) &= (\neg list_g \wedge \bigwedge_{j=1}^{m} \big(g_j \to \tau(x \sim^{\pi}_{rpo} t_j)\big) \quad \text{for variables } x && (4b') \\
\tau(f(\bar{s}) \sim^{\pi}_{rpo} g(\bar{t})) &= \left(\neg list_f \to \bigwedge_{i=1}^{n} \big(f_i \to \tau(s_i \sim^{\pi}_{rpo} g(\bar{t}))\big) \right) \quad \wedge && (4c') \\
&\quad \left((list_f \wedge \neg list_g) \to \bigwedge_{j=1}^{m} \big(g_j \to \tau(f(\bar{s}) \sim^{\pi}_{rpo} t_j)\big) \right) \quad \wedge \\
&\quad \Big((list_f \wedge list_g) \to \Big((f \approx_{\Sigma} g) \wedge \tau(\bar{s} \sim^{f,g,\pi}_{rpo} \bar{t}) \Big) \Big)
\end{aligned}
$$

For the lexicographic comparison w.r.t. permutations, we enhance Formula (5) of Chapter 6 to specify the relation between filters and permutations. Only non-filtered arguments are permuted. Moreover, for an n-ary symbol f with $\ell < n$ non-filtered arguments, the permutation should map all ℓ non-filtered arguments to positions from $\{1, \ldots, \ell\}$. Formula (5a′) states that if some argument of f is considered at the k-th position (i.e., some $f_{i,k}$ is *true*), then there is exactly one such argument. Formula (5b′) specifies that filtered arguments may not be used in the permutation. So if the i-th argument of f is filtered (i.e., f_i is *false*), then the permutation variables $f_{i,k}$ (for $1 \leq k \leq n$) are also *false*. Formula (5c′) states that if the i-th argument of f is not filtered (i.e., f_i is *true*), then the i-th argument of f is considered at exactly one position in the permutation. Finally, Formula (5d′) expresses that all ℓ non-filtered arguments are permuted "to the left", i.e., to positions from $\{1, \ldots, \ell\}$. Hence, if an argument is mapped to position k, then some argument is also mapped to position $k - 1$.

$$\bigwedge_{k=1}^{n} \left(\bigvee_{i=1}^{n} f_{i,k} \to one(f_{1,k}, \ldots, f_{n,k}) \right) \qquad (5a')$$

$$\bigwedge_{i=1}^{n} \left(\neg f_i \to \bigwedge_{k=1}^{n} \neg f_{i,k} \right) \qquad (5b')$$

$$\bigwedge_{i=1}^{n} (f_i \rightarrow one(f_{i,1}, \ldots, f_{i,n})) \tag{5c$'$}$$

$$\bigwedge_{k=2}^{n} \left(\bigvee_{i=1}^{n} f_{i,k} \rightarrow \bigvee_{i=1}^{n} f_{i,k-1} \right) \tag{5d$'$}$$

For the encoding of $f(\bar{s}) \sim_{rpo}^{f,g,\pi} g(\bar{t})$ and $f(\bar{s}) \succ_{rpo}^{f,g,\pi} g(\bar{t})$, we use the notation $\sim_{lex}^{f,g,\pi}$ and $\succ_{lex}^{f,g,\pi}$ when the arguments of f and g are compared lexicographically. For an equality constraint of the form $\bar{s} \sim_{lex}^{f,g,\pi} \bar{t}$ we enhance Equation (6) of Chapter 6. There must be a one-to-one correspondence between the non-filtered arguments of $\bar{s}$ and of $\bar{t}$ via the permutations for f and for g. To express this, we use a constraint of the form *eq_arity(f,g)* in Equation (6$'$) which states that the number of non-filtered arguments of f and of g are the same. It corresponds to the constraint $(n = m)$ in Equation (6) of Chapter 6 and is encoded as

$$\begin{aligned} eq\_arity(f,g) &= \bigwedge_{k=1}^{\max(n,m)} \left(\bigvee_{i=1}^{n} f_{i,k} \leftrightarrow \bigvee_{j=1}^{m} g_{j,k} \right) \\ \tau(\bar{s} \sim_{lex}^{f,g,\pi} \bar{t}) &= eq\_arity(f,g) \wedge \bigwedge_{k=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \big(f_{i,k} \wedge g_{j,k} \rightarrow \tau(s_i \sim_{rpo}^{\pi} t_j)\big) \end{aligned} \tag{6$'$}$$

Next we enhance Equation (7) of Chapter 6 to define $\succ_{rpo}^{f,g,\pi} = \succ_{rpo}^{f,g,1,\pi}$. For $m < k \leq n$ we now require that f considers an argument at the k-th position. The remaining cases are structurally identical to the corresponding cases of Equation (7) of Chapter 6.

$$\tau(\bar{s} \succ_{lex}^{f,g,k,\pi} \bar{t}) = \begin{cases} false & \text{if } k > n \\ \bigvee_{i=1}^{n} f_{i,k} & \text{if } m < k \leq n \\ \bigvee_{i=1}^{n} \Big(f_{i,k} \wedge \Big(\bigwedge_{j=1}^{m} g_{j,k} \rightarrow & \text{otherwise} \\ \quad (\tau(s_i \succ_{rpo}^{\pi} t_j) \vee (\tau(s_i \sim_{rpo}^{\pi} t_j) \wedge \tau(\bar{s} \succ_{lex}^{f,g,k+1,\pi} \bar{t}))) \Big) \Big) & \end{cases} \tag{7$'$}$$

For the multiset comparison, we enhance Formula (8) of Chapter 6 such that the multiset cover only considers non-filtered arguments of $f(\bar{s})$ and $g(\bar{t})$. Formula (8a$'$) states that if the j-th argument of g is not filtered (i.e., g_j is *true*), then there must be exactly one argument of f that covers it. Formula (8b$'$) states that if the i-th argument of f is filtered (i.e., f_i is *false*), then it cannot cover any arguments of g. Formula (8c$'$) specifies that if the j-th argument of g is filtered (i.e., g_j is *false*), then there is no argument of f that covers it. Finally, the newly labeled Formula (8d$'$) is taken straight from the original

Formula (8) of Chapter 6

$$\bigwedge_{j=1}^{m} (g_j \to one(\gamma_{1,j}, \ldots, \gamma_{n,j})) \tag{8a$'$}$$

$$\bigwedge_{i=1}^{n} \left(\neg f_i \to \neg \bigvee_{j=1}^{m} \gamma_{i,j} \right) \tag{8b$'$}$$

$$\bigwedge_{j=1}^{m} \left(\neg g_j \to \neg \bigvee_{i=1}^{n} \gamma_{i,j} \right) \tag{8c$'$}$$

$$\bigwedge_{i=1}^{n} \left(\varepsilon_i \to one(\gamma_{i,1}, \ldots, \gamma_{i,m}) \right) \tag{8d$'$}$$

Now we define $\tau(\bar{s} \succsim_{mul}^{f,g,\pi} \bar{t})$ analogously to Equation (9) of Chapter 6 and for the encoding of $\succ_{mul}^{f,g,\pi}$ and $\sim_{mul}^{f,g,\pi}$, we restrict Equations (10) and (11) of Chapter 6 to arguments that are not filtered:

$$\tau(\bar{s} \succsim_{mul}^{f,g,\pi} \bar{t}) \;=\; \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \big(\gamma_{i,j} \to \big(\varepsilon_i \to \tau(s_i \sim_{rpo}^{\pi} t_j); \tau(s_i \succ_{rpo}^{\pi} t_j)\big)\big) \tag{9$'$}$$

$$\tau(\bar{s} \succ_{mul}^{f,g,\pi} \bar{t}) \;=\; \tau(\bar{s} \succsim_{mul}^{f,g,\pi} \bar{t}) \wedge \neg \bigwedge_{i=1}^{n} (f_i \to \varepsilon_i) \tag{10$'$}$$

$$\tau(\bar{s} \sim_{mul}^{f,g,\pi} \bar{t}) \;=\; \tau(\bar{s} \succsim_{mul}^{f,g,\pi} \bar{t}) \wedge \bigwedge_{i=1}^{n} (f_i \to \varepsilon_i) \tag{11$'$}$$

Finally, for the combination of lexicographic and multiset comparisons, we simply change the equations (12) and (13) of Chapter 6 to use $\succ_{mul}^{f,g,\pi}$ instead of $\succ_{mul}^{f,g}$ etc.:

$$\tau(\bar{s} \succ_{rpo}^{f,g,\pi} \bar{t}) \;=\; \Big(m_f \wedge m_g \wedge \tau(\bar{s} \succ_{mul}^{f,g,\pi} \bar{t})\Big) \vee \Big(\neg m_f \wedge \neg m_g \wedge \tau(\bar{s} \succ_{lex}^{f,g,1,\pi} \bar{t})\Big) \tag{12$'$}$$

$$\tau(\bar{s} \sim_{rpo}^{f,g,\pi} \bar{t}) \;=\; \Big(m_f \wedge m_g \wedge \tau(\bar{s} \sim_{mul}^{f,g,\pi} \bar{t})\Big) \vee \Big(\neg m_f \wedge \neg m_g \wedge \tau(\bar{s} \sim_{lex}^{f,g,\pi} \bar{t})\Big) \tag{13$'$}$$

Example 7.7. Consider the first arguments of QUOT in the left-hand side and the right-hand side of dependency pair (7) of Example 7.1. Using the above encoding, after simplification of conjunctions, disjunctions, and implications with *true* and *false* and using the side conditions for permutation, multiset cover, and argument filter we obtain:

$$\begin{aligned}
& \tau(\mathsf{s}(x) \succ_{rpo}^{\pi} \mathsf{minus}(x,y)) \\
= \;& (\mathsf{s}_1 \wedge list_{\mathsf{s}} \wedge \neg list_{\mathsf{minus}} \wedge \mathsf{minus}_1) \vee \\
& ((\mathsf{minus}_1 \to \mathsf{s}_1 \wedge list_{\mathsf{s}}) \wedge \neg \mathsf{minus}_2 \wedge (list_{\mathsf{minus}} \to list_{\mathsf{s}} \wedge (\mathsf{s} >_{\Sigma} \mathsf{minus} \vee (\mathsf{s} \approx_{\Sigma} \mathsf{minus} \wedge \\
& \;\neg m_{\mathsf{s}} \wedge \neg m_{\mathsf{minus}} \wedge \mathsf{s}_{1,1} \wedge \neg \mathsf{minus}_{1,1} \wedge \neg \mathsf{minus}_{2,1} \wedge \mathsf{s}_1 \wedge \neg \mathsf{minus}_1 \wedge \neg \mathsf{minus}_2))))
\end{aligned}$$

Thus, $\mathsf{s}(x) \succ_{rpo}^{\pi} \mathsf{minus}(x, y)$ holds if and only if

- minus is collapsed to its first argument and s is not filtered; or
- s and minus are not collapsed, s is greater than minus in the precedence, the second argument of minus is filtered, and whenever minus keeps the first argument then s keeps the first argument, too; or
- s and minus are not collapsed, s is equal to minus in the precedence, s keeps the first argument while minus filters both arguments.

Example 7.8. We solved the inequality $\mathsf{PLUS'}(x, \mathsf{s}(y), z) \;{}_{(}\!\succsim\!{}_{)}\; \mathsf{PLUS'}(y, x, \mathsf{s}(z))$ in Ex. 7.5 by the argument filter $\pi(\mathsf{PLUS'}) = [1, 2]$ and RPO. To find such argument filters and the status and precedence of the RPO, such inequalities are now encoded into propositional formulas. Indeed, the formula resulting from our inequality is satisfiable by the corresponding setting of the propositional variables (i.e., $m_{\mathsf{PLUS'}} = list_{\mathsf{PLUS'}} = \mathsf{PLUS'_1} = \mathsf{PLUS'_2} = true$ and $\mathsf{PLUS'_3} = false$). So we use a multiset comparison for the filtered tuples $\langle x, \mathsf{s}(y)\rangle$ and $\langle y, x\rangle$. Hence, as in Example 6.10 we set $\gamma_{1,2} = \varepsilon_1 = \gamma_{2,1} = true$ and $\varepsilon_2 = false$.

In recent refinements of the DP method [GTSF06], the choice of the argument filter π also influences the set of usable rules which contribute to the inequalities that have to be oriented. In the following section, we show how to extend the encoding of RPO and argument filters in order to take this refinement into account as well. Similar to Section 6.2 one can easily show that the size of our encoding is again polynomial.

7.3. Argument Filters and Usable Rules

Recent improvements of the DP method [GTSF03, TGS04, GTSF06] significantly reduce the number of rules required to be weakly decreasing in the reduction pair processor of Theorem 7.3. We first recapitulate the improved reduction pair processor and then adapt our propositional encoding accordingly.

The idea is that one can restrict the set of usable rules by taking the argument filter into account: in right-hand sides of dependency pairs or rules, an occurrence of f in the i-th argument of g will never be the cause to introduce a usable f-rule if the argument filter eliminates g's i-th argument. For instance, when taking $\pi(\mathsf{QUOT}) = [2]$ in Example 7.1, the right-hand sides of the *filtered* dependency pairs do not contain minus anymore. Thus, no rule is considered usable. In Definition 7.9, we define these restricted usable rules for a term t (initially corresponding to the right-hand side of a dependency pair). Here, we make the TRS $\mathcal{R}$ explicit to facilitate a straightforward encoding in Definition 7.12 afterwards.

Definition 7.9 (Usable Rules modulo π [GTSF03, TGS04, GTSF06])**.** *For any function symbol f, let $Rls_{\mathcal{R}}(f) = \{\ell \to r \in \mathcal{R} \mid \ell = f(s_1, \ldots, s_n)\}$. Let $\mathcal{R}$ be a TRS and π an argument filter. For any term t, the* usable rules *$\mathcal{U}_\pi(t, \mathcal{R})$ modulo π are given by:*

$$\begin{array}{rcl} \mathcal{U}_\pi(x, \mathcal{R}) & = & \varnothing \quad \textit{for all variables } x \\ \mathcal{U}_\pi(f(t_1, \ldots, t_n), \mathcal{R}) & = & Rls_{\mathcal{R}}(f) \quad \cup \\ & & \bigcup_{\ell \to r \in Rls_{\mathcal{R}}(f)} \mathcal{U}_\pi(r, \mathcal{R} \setminus Rls_{\mathcal{R}}(f)) \quad \cup \\ & & \bigcup_{\pi(f) \ni i} \mathcal{U}_\pi(t_i, \mathcal{R} \setminus Rls_{\mathcal{R}}(f)) \end{array}$$

For a set of rules $\mathcal{P}$, let $\mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}_\pi(t, \mathcal{R})$.

We refine the reduction pair processor of Theorem 7.3 to consider usable rules modulo π.

Theorem 7.10 (Reduction Pair Processor modulo π [TGS04])**.** *Let $(\succsim, \succ)$ be a reduction pair for a simplification order $\succ$ and let π be an argument filter. Then the following DP processor Proc is sound.*

$$Proc(\mathcal{P}, \mathcal{R}) = \begin{cases} (\mathcal{P} \setminus \mathcal{P}_{\succ^\pi}, \mathcal{R}) & \textit{if } \mathcal{P}_{\succ^\pi} \cup \mathcal{P}_{\succsim^\pi} = \mathcal{P} \textit{ and } \mathcal{R}_{\succsim^\pi} \supseteq \mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \\ (\mathcal{P}, \mathcal{R}) & \textit{otherwise} \end{cases}$$

Example 7.11. Consider the following TRS (together with the minus-rules (1), (2))

$$\begin{array}{rcll} \mathsf{minus}(x, \mathsf{0}) & \to & x & (1) \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) & (2) \\ \mathsf{ge}(x, \mathsf{0}) & \to & \mathsf{true} & (12) \\ \mathsf{ge}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{false} & (13) \\ \mathsf{ge}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{ge}(x, y) & (14) \\ \mathsf{div}(x, y) & \to & \mathsf{if}(\mathsf{ge}(x, y), x, y) & (15) \\ \mathsf{if}(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) & (16) \\ \mathsf{if}(\mathsf{false}, x, \mathsf{s}(y)) & \to & \mathsf{0} & (17) \end{array}$$

The usable rules are the minus- and ge-rules since minus occurs in the right-hand side of the dependency pair $\mathsf{IF}(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{DIV}(\mathsf{minus}(x, y), \mathsf{s}(y))$ resulting from rule (16) and ge occurs in the dependency pair $\mathsf{DIV}(x, y) \to \mathsf{IF}(\mathsf{ge}(x, y), x, y)$ resulting from rule (15). However, if one chooses an argument filter with $\pi(\mathsf{DIV}) = [1]$ and $\pi(\mathsf{IF}) = [2]$, then the ge-rules are no longer usable since ge does not occur in the right-hand side of the filtered dependency pair $\mathsf{DIV}(x) \to \mathsf{IF}(x)$. Now Theorem 7.10 only requires the filtered minus-rules and the dependency pairs to be decreasing.

As demonstrated in [GTSF03, TGS04, GTSF06] and confirmed by the experiments described in Section 7.4, introducing argument filters to the specification of usable rules results in a significant gain of termination proving power. However, Theorem 7.10 is not straightforward to automate using SAT solvers. The technique of Section 7.2 assumes a given set of inequalities which is then encoded to a propositional formula. The problem with Theorem 7.10 is that that the set of inequalities to be oriented depends on the chosen argument filter. Hence, the search for an argument filter should be combined with the computation of the usable rules. As discussed before, an enumeration of argument filters is hopelessly inefficient. Therefore, we modify the encoding of the inequalities in Formula (11) such that for every rule $\ell \to r \in \mathcal{R}$, the condition under which $\ell \to r$ is usable is considered. Only under this condition one has to require the inequality $\pi(\ell) \succsim_{rpo} \pi(r)$. To this end, instead of encoding formula (11) we encode the following formula.

$$\underbrace{\bigwedge_{\ell\to r\in\mathcal{U}_\pi(\mathcal{P},\mathcal{R})} \ell \succsim_{lpo}^\pi r}_{(a)} \;\wedge\; \underbrace{\bigwedge_{s\to t\in\mathcal{P}} s \succsim_{lpo}^\pi t}_{(b)} \;\wedge\; \underbrace{\bigvee_{s\to t\in\mathcal{P}} s \succ_{lpo}^\pi t}_{(c)} \qquad (11')$$

The subformulas (b) and (c) are identical to those in Formula (11) and are encoded as a conjunction and disjunction of encodings of the forms $\tau(s \succsim_{lpo}^\pi t)$ and $\tau(s \succ_{lpo}^\pi t)$ using the encoding of Section 7.2. The definition of the usable rules in Definition 7.9 now induces the following encoding of subformula (a) as a propositional formula $\omega(\mathcal{P},\mathcal{R})$.[2] As in Section 7.2, we use argument filter constraints of the form "$\pi(f) \supseteq i$". Moreover, we introduce a new propositional variable u_f for every defined function symbol f of $\mathcal{U}(\mathcal{P},\mathcal{R})$ which indicates whether f's rules are usable.

Definition 7.12 (Encoding Usable Rules modulo Argument Filter)**.** *For a term t and a TRS $\mathcal{R}$ the formula $\omega(t,\mathcal{R})$ is defined as follows:*

$$\begin{array}{rcll}
\omega(x,\mathcal{R}) & = & \mathit{true} & \text{for } x \in \mathcal{V} \\
\omega(f(t_1,\ldots,t_n),\mathcal{R}) & = & \bigwedge_{1\le i\le n} (\pi(f) \supseteq i \to \omega(t_i,\mathcal{R})) & \text{for } f \notin \mathcal{D}_\mathcal{R} \\
\omega(f(t_1,\ldots,t_n),\mathcal{R}) & = & u_f \quad \wedge & \text{for } f \in \mathcal{D}_\mathcal{R} \\
 & & \bigwedge_{\ell\to r\in Rls_\mathcal{R}(f)} \omega(r,\mathcal{R}\setminus Rls_\mathcal{R}(f)) \quad \wedge & \\
 & & \bigwedge_{1\le i\le n} (\pi(f) \supseteq i \to \omega(t_i,\mathcal{R}\setminus Rls_\mathcal{R}(f))) &
\end{array}$$

For a set of rules $\mathcal{P}$, let

$$\omega(\mathcal{P},\mathcal{R}) = \left(\bigwedge_{s\to t\in\mathcal{P}} \omega(t,\mathcal{R})\right) \wedge \left(\bigwedge_{f\in\mathcal{D}_{\mathcal{U}(\mathcal{P},\mathcal{R})}} u_f \to \left(\bigwedge_{\ell\to r\in Rls_\mathcal{R}(f)} \tau(\ell \succsim_{lpo}^\pi r)\right)\right).$$

[2]The definition of ω can easily be adapted to more advanced definitions of usable rules as well, cf. e.g. [AG00, GTSF03, GTS05b].

For a DP problem $(\mathcal{P}, \mathcal{R})$ we encode the formula (11′). Every model of this encoding corresponds to a precedence $\geq_\Sigma$, a status function σ, and an argument filter π satisfying the constraints of the improved reduction pair processor from Theorem 7.10. Thus, we can now use SAT solving to automate Theorem 7.10 as well.

Example 7.13. Consider again the TRS $\mathcal{R}$ from Example 7.11. Using the encoding of Definition 7.12, for $\mathcal{P} = DP(\mathcal{R})$ we obtain:

$$\begin{array}{rcl}
\omega(\mathcal{P}, \mathcal{R}) & = & (\pi(\mathsf{DIV}) \supseteq 1 \to u_{\mathsf{minus}}) \wedge (\pi(\mathsf{IF}) \supseteq 1 \to u_{\mathsf{ge}}) \wedge \\
(u_{\mathsf{minus}} & \to & (\tau(\mathsf{minus}(x, \mathsf{0}) \succsim^{\pi}_{lpo} x) \wedge \tau(\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim^{\pi}_{lpo} \mathsf{minus}(x, y)))) \wedge \\
(u_{\mathsf{ge}} & \to & (\tau(\mathsf{ge}(x, \mathsf{0}) \succsim^{\pi}_{lpo} \mathsf{true}) \wedge \tau(\mathsf{ge}(\mathsf{0}, \mathsf{s}(y)) \succsim^{\pi}_{lpo} \mathsf{false}) \wedge \\
& & \tau(\mathsf{ge}(\mathsf{s}(x), \mathsf{s}(y)) \succsim^{\pi}_{lpo} \mathsf{ge}(x, y))))
\end{array}$$

7.4. Implementation and Experiments

The propositional encodings for RPO with argument filters and for the reduction pair processors described in Sections 7.2 and 7.3 are implemented and integrated in our termination prover AProVE. Our implementation uses several optimizations to minimize encoding size:

(i) We apply basic simplification axioms for *true* and *false* as well as standard Boolean simplifications to flatten nested conjunctions and disjunctions.

(ii) When building the formulas top-down, at each point we maintain the sets of atomic constraints (on precedences and argument filters) that must be *true* and *false* from this point on. This information is then applied to simplify all constraints generated below (in the top-down process) and to prune the encoding process.

(iii) We memo and identify identical subformulas in the propositional encodings and represent formulas as directed acyclic graphs (or Boolean circuits) instead of trees. This decreases the size of the representation considerably. (The usefulness of sharing when solving LPO constraints was already discussed in [GG97].) For instance, consider the constraint from Example 7.7. Already in this tiny example, the subformula $\mathsf{s}_1 \wedge list_{\mathsf{s}}$ occurs twice.

Optimization (ii) typically reduces the number of propositional variables in the resulting CNF by a factor of at least 2. Optimizations (i) and (iii) together further reduce the number of propositional variables by a typical factor of 10.

We tested the implementation on all 865 TRSs from the TPDB [TPD06] with the identical setup as in Section 6.3.

Apart from the reduction pair processor, we also used the *dependency graph processor* [AG00, GTS05a, HM05a], which is the other main processor of the dependency pair framework.

The following table compares our new SAT-based approach for path orders within the DP framework to the previous dedicated solvers in AProVE 1.2 which did not use SAT solving. The columns contain the data for LPO with strict and non-strict precedence (denoted *lpo*/*qlpo*), for LPO with status (*lpos*/*qlpos*), for MPO (*mpo*/*qmpo*), and for RPO with status (*rpo*/*qrpo*). For each encoding we give the number of TRSs which could be proved terminating (with the number of time-outs in brackets) and the analysis time (in seconds) for the full collection.

Solver	*lpo*	*qlpo*	*lpos*	*qlpos*	*mpo*	*qmpo*	*rpo*	*qrpo*
SAT	**357** (0) **79.3**	**389** (0) **199.6**	**362** (0) **69.0**	**395** (2) **261.1**	**369** (0) **110.9**	**408** (1) **267.8**	**375** (0) **108.8**	**416** (2) **331.4**
ded.	**350**(55) **4039.6**	**374**(79) **5469.4**	**355**(57) **4522.8**	**380**(92) **6476.5**	**359**(69) **5169.7**	**391**(82) **5839.5**	**364**(74) **5536.6**	**394**(102) **7186.1**

The table shows that with our new SAT encoding, performance improves by orders of magnitude over existing dedicated solvers. Note that without a time-out, this effect would be aggravated. By using SAT, the number of time-outs reduces dramatically from up to 102 to at most 2. The two remaining SAT examples with time-out have function symbols of high arity and can only be shown terminating by further sophisticated termination techniques in addition to RPO. Apart from these two, there are only 15 examples that take longer than two seconds and only 3 of these take longer than 10 seconds.

The table also shows that combining RPO with argument filters significantly increases the number of examples that can be shown terminating. The best configuration, *qrpo*, shows termination for 416 examples compared to 162 examples in the experiments of Chapter 6. While the increase in runtime is prohibitive when using dedicated algorithms, when using our SAT encodings, one should always combine RPO with argument filters.

7.5. Summary

In Chapter 6 we demonstrated the power of propositional encoding and application of SAT solving to RPO termination analysis. This chapter extends the SAT-based approach to consider the more realistic setting of dependency pair problems with RPO and argument filter. The main challenge derives from the strong dependencies between the notions of RPO, argument filters, and the set of rules which need to be oriented. The key to a solution is to introduce and encode in SAT all of the constraints originating from these notions into a single search process. We introduced such an encoding and through implementation and experimentation showed that it meets the challenge, yielding speedups in orders of magnitude over existing termination tools as well as increasing termination proving power.

In our experiments with our transformational approach for termination analysis of logic programs in Chapter 3, we used both our SAT-based implementation of polynomial orders [FGM+07] and the SAT-based implementation of RPO and argument filters from this chapter to search for well-founded orders.

While our implementation of RPO and argument filtering was used in many situations, here, we discuss three of the logic programs in more details. For these examples, AProVE could only find a proof using RPO and argument filters, and no other tool for termination analysis of logic programs could prove their termination at all. For each of these three examples and for each of the LP termination tools, the following table lists whether the tool was successful for the example and how many seconds it took.

	AProVE	Polytool 2	Polytool	TerminWeb	cTI	TALP
`shapes`	**Success**	*Timeout*	Timeout	Failure	Failure	Failure
	15.28	*60*	60	0.19	0.04	0.28
`cnfequiv`	**Success**	*Failure*	Failure	Failure	Failure	Failure
	10.50	*7.08*	3.29	0.13	0.04	0.24
`hbal_tree`	**Success**	*Failure*	Failure	Failure	Failure	Timeout
	10.57	*5.91*	2.95	0.08	0.03	60

The example `shapes` was an entry to to the *4th Annual Prolog programming contest* in Leuven, Belgium, in 1997. In a matrix of black and white pixels, it counts white shapes, i.e., areas of connected white pixels.

We already know `cnfequiv` from Example 6.1. It was written in 2004 by Martin Jurdzinski to transform propositional formulas into conjunctive normal form (CNF).

Finally, `hbal_tree` contains an algorithm that, given a natural number n, constructs all height-balanced trees of n nodes by backtracking. It is *P-59* from Werner Hett's collection *P-99: Ninety-Nine Prolog Problems.*

Thus, we can conclude that the contributions of Chapter 6 and of this chapter have improved the efficiency of RPO implementations by orders of magnitude. Furthermore, they have yielded the first automated termination proofs for a number of natural logic programs.

8. Conclusion

In Chapter 3 we developed a new transformation from logic programs to TRSs (Contribution (i)). To prove termination of the resulting TRSs automatically, we showed how to adapt the DP framework to infinitary constructor rewriting. We formally proved that our new approach strictly subsumes the classical transformational approach. All contributions of the chapter were implemented in our termination prover AProVE and evaluated successfully in extensive experiments (cf. Section 3.6). Due to these contributions, AProVE is currently the most powerful automated termination prover for logic programs. This is also evidenced by AProVE reaching the highest score in all the annual international Termination Competitions since 2004.

In Chapter 4 we introduced a new framework for termination analysis of LPs: the dependency triple framework. This framework allows for the first constraint-based modular approach for termination analysis (Contribution (ii)). A prototypical implementation in the tool Polytool [ND07] yielded the most powerful automated direct termination prover for logic programs (cf. Sections 3.6 and 4.4). This is further evidenced by Polytool reaching the second highest score in the Termination Competition 2007. The modularization also allows to combine direct and transformational approaches by the modular application of the new transformation from Chapter 3 (Contribution (iii)).

In Chapter 5 we introduced a novel non-termination preserving pre-processing method to eliminate the effect of cuts in logic programs (Contribution (iv)). After this pre-processing, any technique for proving universal termination of definite logic programming can be applied. Our method also works for meta-programming, and cuts can be used to express negation-as-failure as well as existential termination. This pre-processing has been implemented in our tool AProVE and has successfully been evaluated on logic programs that use cuts and negation-as-failure. For details, see Section 5.5.

In Chapter 6 we presented a polynomial-size encoding from recursive path orders to propositional logic such that the resulting formula is satisfiable if, and only if, termination can be shown with such an order (Contribution (v)). Our encoding combines the solving of constraints on partial orders and performing multiset comparisons and lexicographic comparisons w.r.t. arbitrary permutations into a single search process. Through extensive experiments in Section 6.3 we showed that our encoding is faster than existing dedicated solvers by orders of magnitude.

In Chapter 7 we extended the encoding of Chapter 6 to consider the more realistic setting of dependency pair problems with recursive path orders and argument filters (Contribution (vi)). The key idea was again to encode all of the constraints originating from these notions into a single search process. Through implementation and experimentation presented in 7.4 we showed that this encoding also yields speedups in orders of magnitude over existing termination tools.

The Big Picture

We conclude this thesis by presenting a strategy how to combine and apply Contributions (i) – (vi) in order to obtain a powerful automated termination prover for logic programs.

Assume that we are given a logic program $\mathcal{P}$ and a set of queries. This is the typical setting for termination analyzers in logic programming.

1. If $\mathcal{P}$ contains cuts, apply the pre-processing step from Chapter 5 to $\mathcal{P}$. Show universal termination for the resulting logic program instead.
2. Use the contributions of Chapter 4 and build the initial dependency triple problem $(DT(\mathcal{P}), Call(\mathcal{S}, \mathcal{P}), \mathcal{P})$.
3. Apply dependency triple processors from Chapter 4 based on the dependency graph and on reduction pairs as long as possible.
4. If termination cannot be shown for all sub-problems, use the processor based on the transformation from Chapter 3 to obtain a dependency pair problem.
5. Use the adapted dependency pair framework from Chapter 3. Among others, use the encodings from Chapters 6 and 7 to search for well-founded orders.

Empirical Results

The theoretical contributions developed in this thesis have been implemented in tools for automated termination analysis – Contributions (vii) and (ix) in our fully automated termination prover AProVE and Contribution (viii) in a prototypical implementation of Polytool which we refer to as Polytool 2 to distinguish it from the version of Polytool that implements [ND07].

To evaluate Contributions (vii) and (viii), we tested AProVE and Polytool 2 against four other representative termination tools for logic programming: TALP [OCM00] is the only other available tool based on transformational methods (it uses the classical transformation [Ohl01]), whereas Polytool [ND07], TerminWeb [CT99], and cTI [MB05] are based on direct approaches.

We ran all available termination provers for logic programming on a set of 296 examples of which 52 examples are known to be non-terminating, i.e., there are at most 244 terminating examples. This set basically includes all logic programming examples from the *Termination Problem Data Base* [TPD07], which is used in the annual international *Termination Competition* [MZ07]. For details on the examples or on the configuration of the tools, we refer to Section 3.6. For every tool we give the number of LPs which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total running time ("Total") in seconds.

	AProVE	Polytool 2	Polytool	TerminWeb	cTI	TALP
Successes	**232**	*220*	204	177	167	163
Failures	**57**	*65*	82	118	129	112
Timeouts	**7**	*11*	10	1	0	21
Total	**1471.4**	*1517.4*	622.7	95.3	10.4	413.5

The table show that there are only at most 12 terminating examples where AProVE did not manage to prove termination. With this performance, AProVE won the Termination Competition with Polytool being the second most powerful tool. The comparison with Polytool 2, Polytool, TerminWeb, and cTI demonstrates that our transformational approach is not only comparable in power, but usually more powerful than direct approaches. Once fully implemented, Contribution (iii) would clearly yield an even more powerful termination prover.

Note that there are several examples where AProVE succeeds whereas no other tool shows their termination. For instance, termination of the example `SGST06/hbal_tree.pl` can only be shown using Contribution (ix), i.e., RPO in combination with argument filters.

To evaluate Contribution (ix), we tested our implementation on all 865 TRSs from the TPDB [TPD06]. For details on the setup of the experiments, we refer to Section 7.4. The following table compares our new SAT-based approach for path orders and argument filters to the previous dedicated solvers in AProVE which did not use SAT solving. The columns contain the data for recursive path orders with non-strict precedence (denoted *rpo*) and for lexicographic path orders with strict precedence (*lpo*). For each encoding we again give the number of "Successes", "Failures", and "Timeouts". Additionally, in the fourth row we give the analysis time (in seconds) for the full collection of 865 examples.

	SAT-based *rpo*	dedicated *rpo*	SAT-based *lpo*	dedicated *lpo*
Successes	**416**	394	*357*	350
Failures	**447**	369	*508*	460
Timeouts	**2**	102	*0*	55
Total Time	**331.4**	7186.1	*79.3*	4039.6

The table shows that with our new SAT encoding, performance improves by orders of magnitude over existing dedicated solvers. The table also shows that the use of RPO instead of LPO also increases power substantially in the combination with argument filters. This increase in power comes with a large penalty in runtime when the dedicated solver is used, while in the SAT-based setting, runtimes increase only mildly.

In our experiments to evaluate Contribution (ix), we found three logic programs for which AProVE could only find a proof using RPO and argument filters. The following table shows that no other tool for termination analysis of logic programs could prove their termination at all. For more details, we refer to Section 7.5.

	AProVE	Polytool 2	Polytool	TerminWeb	cTI	TALP
`shapes`	**Success**	*Timeout*	Timeout	Failure	Failure	Failure
	15.28	*60*	60	0.19	0.04	0.28
`cnfequiv`	**Success**	*Failure*	Failure	Failure	Failure	Failure
	10.50	*7.08*	3.29	0.13	0.04	0.24
`hbal_tree`	**Success**	*Failure*	Failure	Failure	Failure	Timeout
	10.57	*5.91*	2.95	0.08	0.03	60

Future Work

The contributions of this thesis have advanced the state of the art in termination analysis of both logic programming and term rewriting significantly. Still, there are many ways in which these contributions might be extended in future work. For example, one should adapt further processors from term rewriting to the framework presented in Chapter 4 and develop completely new processors in this framework that rely on special properties of logic programming.

Another important topic for future work regarding Contributions (i) – (iv) is the effective and efficient handling of built-in data structures like integers and of built-in (meta) predicates. Likewise, the techniques of Contributions (ii) and (iv) should be adapted to handle both unification with and unification without an occur check.

The graph-based approach of Chapter 5 could be made more precise by, e.g., integrating a sharing analysis. Adding some kind of structure shape analysis would allow both for a more precise graph and for improved power on queries expressing existential termination.

While we applied the graph-based approach of Chapter 5 to a programming languages based on logic programming (Prolog), in [GSST06] we applied a similar graph-based approach to a functional programming language (Haskell). Interesting future work is to develop a new graph-based approach for termination analysis of imperative programs. Indeed, we are currently investigating such an approach for Java programs.

Finally, the encodings presented in Chapters 6 and 7 could be extended to handle more complex orders (e.g., higher-order recursive path order, recursive path order for rewriting modulo associativity and commutativity, context-sensitive recursive path order). And, of course, one might apply ideas developed here for encodings of other search problems.

Bibliography

[AE93] K. R. Apt and S. Etalle. On the Unification Free Prolog Programs. In *MFCS '93*, volume 711 of *LNCS*, pages 1–19, 1993.

[AEF$^+$08] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving Context-Sensitive Dependency Pairs. In *LPAR '08*, volume 5330 of *LNAI*, 2008. To appear.

[AG00] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[AGIL07] B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving Termination of Context-Sensitive Rewriting With MU-TERM. In *PROLE '06*, volume 188 of *Electronic Notes in Theoretical Computer Science*, pages 105–115, 2007.

[AM93] G. Aguzzi and U. Modigliani. Proving Termination of Logic Programs by Transforming them into Equivalent Term Rewriting Systems. In *FSTTCS '93*, volume 761 of *LNCS*, pages 114–124, 1993.

[Apt97] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.

[AZ95] Thomas Arts and Hans Zantema. Termination of Logic Programs Using Semantic Unification. In *LOPSTR '95*, volume 1048 of *LNCS*, pages 219–233, 1995.

[BAK91] R. N. Bol, K. R. Apt, and J. W. Klop. An Analysis of Loop Checking Mechanisms for Logic Programs. *Theoretical Computer Science*, 86(1):35–79, 1991.

[BCF92] A. Bossi, N. Cocco, and M. Fabris. Typed Norms. In *ESOP '92*, volume 582 of *LNCS*, pages 73–92, 1992.

[BCF94] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124(2):297–328, 1994.

[BCG$^+$07] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.

[BDB$^+$96] M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A Freeness and Sharing Analysis of Logic Programs Based on a Pre-Interpretation. In *SAS '96*, volume 1145 of *LNCS*, pages 128–142, 1996.

[BGV05] M. Bruynooghe, J. P. Gallagher, and W. Van Humbeeck. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *SAS '05*, volume 3672 of *LNCS*, pages 35–51, 2005.

[BKN07] L. Bulwahn, A. Krauss, and T. Nipkow. Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL. In *TPHOLs '07*, volume 4732 of *LNCS*, pages 38–53, 2007.

[BM79] R.S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.

[BR03] C. Borralleras and A. Rubio. Termptation (Barcelona, Spain), 2003. `http://www.lsi.upc.es/~albert/term.html`.

[CF99] A. Cortesi and G. Filé. Sharing is Optimal. *Journal of Logic Programming*, 38(3):371–386, 1999.

[CG03] M. Codish and S. Genaim. Proving Termination One Loop at a Time. In *WLPE '03*. K. U. Leuven, 2003.

[CLS05] M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.

[CLS06] M. Codish, V. Lagoon, and P. J. Stuckey. Solving Partial Order Constraints for LPO Termination. In *RTA '06*, volume 4098 of *LNCS*, pages 4–18, 2006.

[CLSS06] M. Codish, V. Lagoon, P. Schachte, and P. J. Stuckey. Size-Change Termination Analysis in k-Bits. In *ESOP '06*, volume 3924 of *LNCS*, pages 230–245, 2006.

[CMU03] E. Contejean, C. Marché, and X. Urbain. C*i*ME (Paris, France), 2003. `http://cime.lri.fr/`.

[Cod07] M. Codish. Collection of Benchmarks, 2007. Available at `http://lvs.cs.bgu.ac.il/~mcodish/suexec/terminweb/bin/terminweb.cgi?command=examples`.

[Col82] A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. Tärnlund, editors, *Logic Programming*. Academic Press, Oxford, 1982.

[CP98] W. Charatonik and A. Podelski. Directional Type Inference for Logic Programs. In *SAS '98*, volume 1503 of *LNCS*, pages 278–294, 1998.

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *SAS '05*, volume 3672 of *LNCS*, pages 87–101, 2005.

[CR93] M. Chtourou and M. Rusinowitch. Méthode Transformationelle pour la Preuve de Terminaison des Programmes Logiques. Unpublished manuscript, 1993.

[CSL$^+$06] M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J. Giesl. SAT Solving for Argument Filterings. In *LPAR '06*, volume 4246 of *LNAI*, pages 30–44, 2006.

[CT94] H. Comon and R. Treinen. Ordering Constraints on Trees. In *CAAP '94*, volume 787 of *LNCS*, pages 1–14, 1994.

[CT99] M. Codish and C. Taboch. A Semantic Basis for Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[DD94] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[DDF93] S. Decorte, D. De Schreye, and M. Fabris. Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis. In *ILPS '93*, pages 420–436, Boston, 1993. MIT Press.

[DDV99] S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based automatic termination analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.

[Der82] N. Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1–2):69–116, 1987.

[DLSS01] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1–2):117–156, 2001.

[DS02] D. De Schreye and A. Serebrenik. Acceptability with General Orderings. In *Computational Logic: Logic Programming and Beyond. Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *LNCS*, pages 187–210. 2002.

[DSVB92] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Framework for Analyzing the Termination of Definite Logic Programs with respect to Call Patterns. In *FGCS '92*, pages 481–488, New York, 1992. ACM Press.

[End06] J. Endrullis. Jambox (Amsterdam, The Netherlands), 2006. `http://joerg.endrullis.de/`.

[ES07] N. Eén and N. Sörensson. MiniSAT Solver, 2007. `http://minisat.se/`.

[EWZ06] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. In *IJCAR '06*, volume 4130 of *LNAI*, pages 574–588, 2006.

[FGK03] O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO: A Multi-strategy Termination Proof Tool based on Induction. In *WST '03*, pages 77–79, 2003.

[FGM$^+$07] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT '07*, volume 4501 of *LNCS*, pages 340–354, 2007.

[FGM$^+$08] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal Termination. In *RTA '08*, volume 5117 of *LNCS*, pages 110–125, 2008.

[FNO$^+$08] C. Fuhs, R. Navarro-Marset, C. Otto, J. Giesl, S. Lucas, and P. Schneider-Kamp. Search Techniques for Rational Polynomial Orders. In *AISC '08*, volume 5144 of *LNAI*, pages 109–124, 2008.

[GG97] T. Genet and I. Gnaedig. Termination Proofs using GPO Ordering Constraints with Shared Term Data Structure. In *TAPSOFT '97*, volume 1214 of *LNCS*, pages 249–260, 1997.

[GHW04] A. Geser, D. Hofbauer, and J. Waldmann. Match-Bounded String Rewriting Systems. *Applicable Algebra in Engineering, Communication and Computing*, 15(3–4):149–171, 2004.

[GP02] J. P. Gallagher and G. Puebla. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *PADL '02*, volume 2257 of *LNCS*, pages 243–261, 2002.

[GSST06] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *RTA '06*, volume 4098 of *LNCS*, pages 297–312, 2006.

[GST06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the DP Framework. In *IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.

[GTS05a] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.

[GTS05b] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *FroCoS '05*, volume 3717 of *LNAI*, pages 216–231, 2005.

[GTSF03] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving Dependency Pairs. In *LPAR '03*, volume 2850 of *LNAI*, pages 165–179, 2003.

[GTSF04] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs With AProVE. In *RTA '04*, volume 3091 of *LNCS*, pages 210–220, 2004.

[GTSF06] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[GTSS07] J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving Termination by Bounded Increase. In *CADE '07*, volume 4603 of *LNAI*, pages 443–459, 2007.

[GW93] H. Ganzinger and U. Waldmann. Termination Proofs of Well-moded Logic Programs via Conditional Rewrite Systems. In *CTRS '92*, volume 656 of *LNCS*, pages 430–437, 1993.

[GZ03] J. Giesl and H. Zantema. Liveness in Rewriting. In *RTA '03*, volume 2706 of *LNCS*, pages 321–336, 2003.

[HM04] N. Hirokawa and A. Middeldorp. Dependency Pairs Revisited. In *RTA '04*, volume 3091 of *LNCS*, pages 249–268, 2004.

[HM05a] N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.

[HM05b] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool. In *RTA '05*, volume 3467 of *LNCS*, pages 175–184, 2005.

[HM07] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.

[Hue76] G. Huet. *Résolution d'Equations dans les Langages d'Ordre 1, 2, ..., ω.* PhD thesis, Université Paris VII, France, 1976.

[HW06] D. Hofbauer and J. Waldmann. Termination of String Rewriting with Matrix Interpretations. In *RTA '06*, volume 4098 of *LNCS*, pages 328–342, 2006.

[JB92] 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.

[KB70] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[KB01] M. Kulas and C. Beierle. Defining Standard Prolog in Rewriting Logic. In *WRLA '00*, volume 36 of *ENTCS*, 2001.

[KK04] M. Kurihara and H. Kondo. Efficient BDD Encodings for Partial Order Constraints with Application to Expert Systems in Software Verification. In *IEA/AIE '04*, volume 3029 of *LNCS*, pages 827–837, 2004.

[KKS98] M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Transformational Methodology for Proving Termination of Logic Programs. *Journal of Logic Programming*, 34(1):1–41, 1998.

[KL80] S. Kamin and J. J. Lévy. Two Generalizations of the Recursive Path Ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.

[KNT99] K. Kusakari, M. Nakamura, and Y. Toyama. Argument Filtering Transformation. In *PPDP '99*, volume 1702 of *LNCS*, pages 47–61, 1999.

[Kop06] A. Koprowski. TPA (Eindhoven, The Netherlands), 2006. `http://www.win.tue.nl/tpa/`.

[Kor07] M. Korp. TTTbox (Innsbruck, Austria), 2007. `http://cl-informatik.uibk.ac.at/~mkorp/tttbox.php`.

[Les83] P. Lescanne. Computer Experiments with the REVE Term Rewriting System Generator. In *POPL '83*, pages 99–108, New York, 1983. ACM Press.

[LMS03] V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination Analysis with Types Is More Accurate. In *ICLP '03*, volume 2916 of *LNCS*, pages 254–268, 2003.

[LP08] D. Le Berre and A. Parrain. SAT4J: The Java SAT Library. `http://www.sat4j.org`, 2008.

[LS96] M. Leuschel and M. H. Sørensen. Redundant Argument Filtering of Logic Programs. In *LOPSTR '96*, volume 1207 of *LNCS*, pages 83–103, 1996.

[LS02] V. Lagoon and P. J. Stuckey. Precise Pair-Sharing Analysis of Logic Programs. In *PPDP '02*, pages 99–108, New York, 2002. ACM Press.

[LSS97] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A System for Checking Termination of Queries to Logic Programs. In *CAV '97*, volume 1254 of *LNCS*, pages 444–447, 1997.

[Lu00] L. Lu. A Precise Type Analysis of Logic Programs. In *PPDP '00*, pages 214–225, New York, 2000. ACM Press.

[Mar94] M. Marchiori. Logic Programs as Term Rewriting Systems. In *ALP '94*, volume 850 of *LNCS*, pages 223–241, 1994.

[Mar96] M. Marchiori. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*, volume 1101 of *LNCS*, pages 375–390, 1996.

[MB05] 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.

[MKS96] J. Martin, A. King, and P. Soper. Typed Norms for Typed Logic Programs. In *LOPSTR '96*, volume 1207 of *LNCS*, pages 224–238, 1996.

[MR03] F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.

[MS07] F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.

[MV06] P. Manolios and D. Vroon. Termination Analysis with Calling Context Graphs. In *CAV '06*, volume 4144 of *LNCS*, pages 401–414, 2006.

[MZ07] C. Marché and H. Zantema. The Termination Competition. In *RTA '07*, volume 4533 of *LNCS*, pages 303–313, 2007. See also `http://www.lri.fr/~marche/termination-competition/`.

[NBDL06] M. T. Nguyen, M. Bruynooghe, D. De Schreye, and M. Leuschel. Program Specialisation as a Pre-Processing Step for Termination Analysis. In *WST '06*, pages 7–11, 2006.

[ND05] M. T. Nguyen and D. De Schreye. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In *ICLP '05*, volume 3668 of *LNCS*, pages 311–325, 2005.

[ND07] M. T. Nguyen and D. De Schreye. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In *LOPSTR '06*, volume 4407 of *LNCS*, pages 210–218, 2007.

[NGSD08] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.

[OCM00] E. Ohlebusch, C. Claves, and C. Marché. TALP: A Tool for the Termination Analysis of Logic Programs. In *RTA '00*, volume 1833 of *LNCS*, pages 270–273, 2000.

[Ohl01] E. Ohlebusch. Termination of Logic Programs: Transformational Methods Revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1–2):73–116, 2001.

[Plü90] L. Plümer. *Termination Proofs for Logic Programs.* Springer, 1990.

[PM06] E. Payet and F. Mesnard. Nontermination Inference of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006.

[Raa97] F. van Raamsdonk. Translating Logic Programs into Conditional Rewriting Systems. In *ICLP '97*, pages 168–182, Boston, 1997. MIT Press.

[SD03a] A. Serebrenik and D. De Schreye. Hasta-La-Vista: Termination Analyser for Logic Programs. In *WLPE '03*, pages 60–74. K. U. Leuven, 2003.

[SD03b] A. Serebrenik and D. De Schreye. Proving Termination with Adornments. In *LOPSTR '03*, volume 3018 of *LNCS*, pages 108–109, 2003.

[SD04] A. Serebrenik and D. De Schreye. Inference of Termination Conditions for Numerical Loops in Prolog. *Theory and Practice of Logic Programming*, 4(5&6):719–751, 2004.

[SD05a] A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[SD05b] A. Serebrenik and D. De Schreye. Termination of Floating Point Computations. *Journal of Automated Reasoning*, 34(2):141–177, 2005.

[SGST] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*. To appear.

[SGST07] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In *LOPSTR '06*, volume 4407 of *LNCS*, pages 177–193, 2007.

[Sma04] J.-G. Smaus. Termination of Logic Programs Using Various Dynamic Selection Rules. In *ICLP '04*, volume 3132 of *LNCS*, pages 43–57, 2004.

[STA$^+$07] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving Termination using Recursive Path Orders and SAT Solving. In *FroCoS '07*, volume 4720 of *LNAI*, pages 267–282, 2007.

[TC04] L. Tamary and M. Codish. Abstract Partial Evaluation for Termination Analysis. In *WST '04*, pages 47–50, 2004. Available at `http://aib.informatik.rwth-aachen.de/2004/2004-07.pdf`.

[TGC02] C. Taboch, S. Genaim, and M. Codish. TerminWeb: Semantic Based Termination Analyser for Logic Programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

[TGS04] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved Modular Termination Proofs Using Dependency Pairs. In *IJCAR '04*, volume 3097 of *LNAI*, pages 75–90, 2004.

[TGS08] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Deciding Innermost Loops. In *RTA '08*, volume 5117 of *LNCS*, pages 366–380, 2008.

[TPD06] The Termination Problem Data Base 3.2 (June 12, 2006), 2006. Available at `http://www.lri.fr/~marche/tpdb/`.

[TPD07] The Termination Problem Data Base 4.0 (May 24, 2007), 2007. Available at `http://www.lri.fr/~marche/tpdb/`.

[Tse68] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. 1968. Reprinted in J. Siekmann and G. Wrightson (editors), *Automation of Reasoning*, 2:466–483, 1983.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Available online at `http://www.abelard.org/turpap2/tp2-ie.asp`.

[TZGS08] R. Thiemann, H. Zantema, J. Giesl, and P. Schneider-Kamp. Adding Constants to String Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 19(1):27–38, 2008.

[VB02] C. Vaucheret and F. Bueno. More Precise Yet Efficient Type Inference for Logic Programs. In *SAS '02*, volume 2477 of *LNCS*, pages 102–116, 2002.

[Wal94] C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[Wal04] Johannes Waldmann. Matchbox: A Tool for Match-Bounded String Rewriting. In *RTA '04*, volume 3091 of *LNCS*, pages 85–94, 2004.

[WSW06] I. Wehrman, A. Stump, and E. Westbrook. Slothrop : Knuth-Bendix Completion with a Modern Termination Checker. In *RTA '06*, volume 4098 of *LNCS*, pages 287–296, 2006.

[Wul06] J. van der Wulp. TEPARLA (Eindhoven, The Netherlands), 2006. `http://www.win.tue.nl/~hzantema/torpa.html`.

[Zan95] H. Zantema. Termination of Term Rewriting by Semantic Labelling. *Fundamenta Informaticae*, 24(1–2):89–105, 1995.

[Zan05] Hans Zantema. Termination of String Rewriting Proved Automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.

[ZM07] H. Zankl and A. Middeldorp. Satisfying KBO Constraints. In *RTA '07*, volume 4533 of *LNCS*, pages 328–342, 2007.

Index

Curriculum Vitae

Name	Jan Peter Schneider-Kamp
Geburtsdatum	23. Januar 1978
Geburtsort	Geldern

Bildungsgang

1988–1997	Liebfrauenschule Mülhausen Abschluss: Allgemeine Hochschulreife
1997–1998	Zivildienst in der Montessori-Gesamtschule Krefeld
1998–2003	Studium der Informatik an der RWTH Aachen Abschluss: Diplom
2003–2008	Wissenschaftlicher Angestellter am Lehr- und Forschungsgebiet Informatik 2 (Prof. Dr. Jürgen Giesl), RWTH Aachen
2009–	Assistant Professor, University of Southern Denmark

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter:: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.