Automated Termination Proofs for Logic Programs by Term Rewriting

Peter Schneider-Kamp, RWTH Aachen, Germany
Jürgen Giesl, RWTH Aachen, Germany
Alexander Serebrenik, TU Eindhoven, The Netherlands
René Thiemann, University of Innsbruck, Austria

There are two kinds of approaches for termination analysis of logic programs: "transformational" and "direct" ones. Direct approaches prove termination directly on the basis of the logic program. Transformational approaches transform a logic program into a term rewrite system (TRS) and then analyze termination of the resulting TRS instead. Thus, transformational approaches make all methods previously developed for TRSs available for logic programs as well. However, the applicability of most existing transformations is quite restricted, as they can only be used for certain subclasses of logic programs. (Most of them are restricted to *well-moded* programs.) In this paper we improve these transformations such that they become applicable for *any* definite logic program. To simulate the behavior of logic programs by TRSs, we slightly modify the notion of rewriting by permitting infinite terms. We show that our transformation results in TRSs which are indeed suitable for *automated* termination analysis. In contrast to most other methods for termination of logic programs, our technique is also sound for logic programming *without occur check*, which is typically used in practice. We implemented our approach in the termination prover AProVE and successfully evaluated it on a large collection of examples.

Authors' addresses:
Peter Schneider-Kamp, Jürgen Giesl, LuFG Informatik 2, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany, {psk,giesl}@informatik.rwth-aachen.de
Alexander Serebrenik, Dept. of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, a.serebrenik@tue.nl
René Thiemann, Institute of Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria, rene.thiemann@uibk.ac.at. Most of the work was carried out while R. Thiemann was at the LuFG Informatik 2, RWTH Aachen.

1. INTRODUCTION

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 the logic programs 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., [De Schreye and Decorte 1994] for an overview and [Bruynooghe et al. 2007; Codish et al. 2005; Codish et al. 2006; De Schreye and Serebrenik 2002; Lagoon et al. 2003; Mesnard and Ruggieri 2003; Mesnard and Serebrenik 2007; Nguyen and De Schreye 2005; 2007; Nguyen et al. 2008; Serebrenik and De Schreye 2005a; Smaus 2004] for more recent work on "direct" approaches). "Transformational" approaches have been developed in [Aguzzi and Modigliani 1993; Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Krishna Rao et al. 1998; Marchiori 1994; 1996; van Raamsdonk 1997] and a comparison of these approaches is given in [Ohlebusch 2001].

In order to be successful, 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 [Marchiori 1994; 1996] are restricted to *well-moded* programs. The transformations of [Marchiori 1994; 1996] 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 cut and negation. While there are transformational approaches which go beyond definite programs [Marchiori 1996], it is not clear how to transform non-definite logic programs into TRSs that are suitable for *automated* termination analysis, cf. (II).

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 [Ohlebusch et al. 2000] is based on the transformations of [Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993] which are shown to be equally powerful in [Ohlebusch 2001]. 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 [Giesl et al. 2006]. Our experiments on large collections of examples in Section 7 show that our transformation indeed produces TRSs that are suitable for automated termination analysis and that AProVE is currently among 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: in Section 1.1 we recapitulate the classical transformation of [Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Ohlebusch

2001] and in Section 1.2 we discuss the approach of adapting TRS-techniques to the logic programming setting (which can be seen as an alternative to our approach of transforming logic programs to TRSs). Then in Section 1.3 we give an overview on the structure of the remainder of the paper.

1.1   The Classical Transformation

Our transformation is inspired by the transformation of [Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Ohlebusch 2001]. 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* [Apt and Etalle 1993]. 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 iff for any of its clauses $H :- B_1, \ldots, B_k$ 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 1.1. *Consider the following variant of a small example from [Ohlebusch 2001].*

$$\mathsf{p}(X, X)$$
$$\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \;\; :- \;\; \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(Y))$$

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

- For each clause c of the form $p(\vec{s}, \vec{t}) :- p_1(\vec{s_1}, \vec{t_1}), \ldots, p_k(\vec{s_k}, \vec{t_k})$, the resulting TRS contains the following rules:

$$p_{in}(\vec{s}) \rightarrow u_{c,1}(p_{1_{in}}(\vec{s}_1), \mathcal{V}(\vec{s}))$$

$$u_{c,1}(p_{1_{out}}(\vec{t}_1), \mathcal{V}(\vec{s})) \rightarrow u_{c,2}(p_{2_{in}}(\vec{s}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1))$$

$$\ldots$$

$$u_{c,k}(p_{k_{out}}(\vec{t}_k), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1) \cup \ldots \cup \mathcal{V}(\vec{t}_{k-1})) \rightarrow p_{out}(\vec{t})$$

Here, $\mathcal{V}(\vec{s})$ are the variables occurring in $\vec{s}$. Moreover, if $\mathcal{V}(\vec{s}) = \{x_1, \ldots, x_n\}$, then "$u_{c,1}(p_{1_{in}}(\vec{s}_1), \mathcal{V}(\vec{s}))$" abbreviates the term $u_{c,1}(p_{1_{in}}(\vec{s}_1), x_1, \ldots, x_n)$, 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. [Ohlebusch 2001]. However, the converse does not hold.

Example 1.2. *For the program of Example 1.1, the transformation results in the following TRS $\mathcal{R}$.*

$$\mathsf{p}_{in}(X) \rightarrow \mathsf{p}_{out}(X)$$

$$\mathsf{p}_{in}(\mathsf{f}(X)) \rightarrow \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X)$$

$$\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(Z)), X) \rightarrow \mathsf{u}_2(\mathsf{p}_{in}(Z), X, Z)$$

$$\mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(Y)), X, Z) \rightarrow \mathsf{p}_{out}(\mathsf{g}(Y))$$

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:

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

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 *[Mesnard and Bagnara 2005],* TALP *[Ohlebusch et al. 2000],* TermiLog *[Lindenstrauss et al. 1997], and* TerminWeb *[Codish and Taboch 1999] fail on Example 1.1.*

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 1.3. *We modify Example 1.1 by replacing $\mathsf{g}(Y)$ with $\mathsf{g}(W)$ in the body of the second clause:*

$$\mathsf{p}(X, X)$$
$$\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \;:\!-\; \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(W))$$

---

[1]Example 1.3 is neither well moded nor simply well typed nor safely typed (using the types "*Any*" and "*Ground*") as required by the transformations of [Aguzzi and Modigliani 1993; Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Krishna Rao et al. 1998; Marchiori 1994; 1996; van Raamsdonk 1997].

Still, all queries $p(t_1, t_2)$ *terminate if* t_1 *is ground. But this program is not well moded, as the second clause violates Condition (a):* $\mathcal{V}_{out}(p(f(X), g(Y))) = \{Y\} \nsubseteq \mathcal{V}_{in}(p(f(X), g(Y))) \cup \mathcal{V}_{out}(p(f(X), f(Z))) \cup \mathcal{V}_{out}(p(Z, g(W))) = \{X, Z, W\}$. *Transforming the program as before yields a TRS with the rule* $u_2(p_{out}(g(W)), X, Z) \rightarrow p_{out}(g(Y))$. *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 1.4. *A natural non-well-moded example is the* append*-program with the clauses*

$$append([\,], M, M)$$
$$append([X|L], M, [X|N]) \;:\!\!-\; append(L, M, N)$$

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

1.2 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 [De Schreye and Serebrenik 2002].[2] However, the automation of this framework is non-trivial in general. As an instance of this framework, the automatic application of polynomial interpretations (well-known in term rewriting) to termination analysis of logic programs is investigated in [Nguyen and De Schreye 2005; 2007]. Moreover, [Nguyen et al. 2008] extends this work further by also adapting a basic version of the *dependency pair approach* [Arts and Giesl 2000] from TRSs to the logic programming setting. This provides additional evidence that techniques developed for term rewriting can successfully be applied to termination analysis of logic programs.

Instead of integrating each termination technique from term rewriting separately, in the current paper we want to make all these techniques available at once. Therefore, unlike [De Schreye and Serebrenik 2002; Nguyen and De Schreye 2005; 2007; Nguyen et al. 2008], we choose a transformational approach. Our goal is a method which

(A) handles programs like Example 1.1 where classical transformations like the one of Section 1.1 fail,

(B) handles non-well-moded programs like Example 1.3 where most current transformational techniques are not even applicable,

(C) allows the successful *automated* application of powerful techniques from rewriting for logic programs like Example 1.1 and 1.3 where current tools based on

---

[2]But in contrast to [De Schreye and Serebrenik 2002], transformational approaches like the one presented in this paper 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 [Giesl et al. 2005; Giesl et al. 2006; Hirokawa and Middeldorp 2005], *semantic labelling* [Zantema 1995], *matchbounds* [Geser et al. 2004], etc.).

direct approaches fail. For larger and more realistic examples we refer to the experiments in Section 7.

1.3 Structure of the Paper

After presenting required preliminaries in Section 2, in Section 3 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. Our new transformation 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*. For that reason, the logic program $\mathsf{p}(\mathsf{s}(X)) :\!- \mathsf{p}(X)$ does not terminate for the query $\mathsf{p}(X)$ whereas the TRS $\mathsf{p}(\mathsf{s}(X)) \to \mathsf{p}(X)$ terminates for all finite terms. However, the infinite derivation of the logic program corresponds to an infinite reduction of the TRS 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 analyzes the termination behavior of logic programs with infinite terms, i.e., of logic programming without occur check.

Section 4 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 5. Section 6 gives a formal proof that our new transformation and our approach to automated termination analysis are strictly more powerful than the classical ones of Section 1.1. We present and discuss an extensive experimental evaluation of our results in Section 7 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 conclude in Section 8.

Preliminary versions of parts of this paper appeared in [Schneider-Kamp et al. 2007]. However, the present article extends [Schneider-Kamp et al. 2007] considerably (in particular, by the results of the Sections 5 and 6). Section 6 contains a new formal comparison with the existing classical transformational approach to termination of logic programs and *proves* formally that our approach is more powerful. The new contributions of Section 5 improve the power of our method substantially as can be seen in our new experiments in Section 7. Moreover, in contrast to [Schneider-Kamp et al. 2007], this article contains the full proofs of all results and a discussion on the limitations of our approach in Section 7.2.

2. PRELIMINARIES ON LOGIC PROGRAMMING AND REWRITING

We start with introducing the basics on (possibly infinite) terms and atoms. Then we present the required notions on logic programming and on term rewriting in Sections 2.1 and 2.2, respectively.

A *signature* is a pair (Σ, Δ) where Σ and Δ are finite sets of function and predicate

symbols. Each $f \in \Sigma \cup \Delta$ has an *arity* $n \geq 0$ and we often write f/n instead of f. We always assume that Σ contains at least one constant $f/0$.

Definition 2.1 *(Infinite Terms and Atoms)*. A *term* over Σ is a tree where every node is labelled with a function symbol from Σ or with a variable from $\mathcal{V} = \{X, Y, \ldots\}$. Every inner node with n children is labelled with some $f/n \in \Sigma$ while leaves are labelled with a variable $X \in \mathcal{V}$ or with $f/0 \in \Sigma$. We write $f(t_1, \ldots, t_n)$ for the term with root 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 different 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. For a term t, let $\mathcal{V}(t)$ be the set of all variables occurring in t. A *position pos* $\in \mathbb{N}^*$ in a (possibly infinite) term t addresses a subtree $t|_{pos}$ of t. The term $t[s]_{pos}$ results from replacing the subterm $t|_{pos}$ at position *pos* in t by the term s. So for *pos* $= \varepsilon$ we have $t|_{\varepsilon} = t$ and $t[s]_{\varepsilon} = s$. Otherwise *pos* $= i \, pos'$ for some $i \in \mathbb{N}$ and $t = f(t_1, \ldots, t_n)$. Then we have $t|_{pos} = t|_{i \, pos'} = t_i|_{pos'}$ and $t[s]_{pos} = t[s]_{i \, pos'} = f(t_1, \ldots, t_i[s]_{pos'} \ldots, t_n)$.

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 $\mathcal{A}(\Sigma, \Delta, \mathcal{V})$, resp.) are the atoms $p(t_1, \ldots, t_n)$ where $t_i \in \mathcal{T}^{rat}(\Sigma, \mathcal{V})$ (and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$, resp.) for all i. We write $\mathcal{A}(\Sigma, \Delta)$ and $\mathcal{T}(\Sigma)$ instead of $\mathcal{A}(\Sigma, \Delta, \varnothing)$ and $\mathcal{T}(\Sigma, \varnothing)$.

2.1 Logic Programming

A *clause* c is a formula $H :- B_1, \ldots, B_k$ with $k \geq 0$ and $H, B_i \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$. H is c's *head* and $B_1, \ldots, B_k$ is c's *body*. A finite set of clauses $\mathcal{P}$ is a *(logic) program*. A clause with empty body is a *fact* and a clause with empty head is a *query*. We usually omit ":-" in queries and just write "$B_1, \ldots, B_k$". The empty query is denoted $\square$. In queries, we also admit rational instead of finite atoms $B_1, \ldots, B_k$.

Since we are also interested in logic programming without occur check we consider infinite *substitutions* $\theta : \mathcal{V} \to \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$. Here, we allow $\theta(X) \neq X$ for infinitely many $X \in \mathcal{V}$. Instead of $\theta(X)$ we often write $X\theta$. If θ is a variable renaming (i.e., a one-to-one correspondence on $\mathcal{V}$), then $t\theta$ is a *variant* of t, where t can be any expression (e.g., a term, atom, clause, etc.). We write $\theta\sigma$ to denote that the application of θ is followed by the application of σ.

A substitution θ is a *unifier* of two terms s and t if and only if $s\theta = t\theta$. We call θ the *most general unifier (mgu)* of s and t if and only if θ is a unifier of s and t and for all 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 [Apt 1997], for example.

Definition 2.2 *(Derivation, Termination)*. Let Q be a query $A_1, \ldots, A_m$, let c be a clause $H :- B_1, \ldots, B_k$. Then Q' is a *resolvent* of Q and c using θ (denoted $Q \vdash_{c,\theta} Q'$) if θ is the mgu[3] of A_1 and H, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$.

---

[3]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 [Huet 1976].

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

Our notion of derivation coincides with logic programming without an occur check [Colmerauer 1982] as implemented in recent Prolog systems such as SICStus or SWI. 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 2.3. *Regard a program* $\mathcal{P}$ *with the clauses* $\mathsf{p}(X) :\!- \mathsf{equal}(X, \mathsf{s}(X)), \mathsf{p}(X)$ *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 2.2, it would be terminating if one uses logic programming with occur check. Indeed, tools like* cTI *[Mesnard and Bagnara 2005] and* TerminWeb *[Codish and Taboch 1999] 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.*

2.2 Term Rewriting

Now we define TRSs and introduce the notion of *infinitary constructor rewriting*. For further details on term rewriting we refer to [Baader and Nipkow 1998].

Definition 2.4 *(Infinitary Constructor Rewriting)*. *A TRS* $\mathcal{R}$ *is a finite set of rules* $\ell \to r$ *with* $\ell, r \in \mathcal{T}(\Sigma, \mathcal{V})$ *and* $\ell \notin \mathcal{V}$.[4] *We divide the signature into defined symbols* $\Sigma_D = \{f \mid \ell \to r \in \mathcal{R}, \mathrm{root}(\ell) = f\}$ *and constructors* $\Sigma_C = \Sigma \setminus \Sigma_D$. *$\mathcal{R}$'s infinitary constructor rewrite relation is denoted* $\to_{\mathcal{R}}$: *for* $s, t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ *we have* $s \to_{\mathcal{R}} t$ *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* $\to^n_{\mathcal{R}}$, $\to^{\geq n}_{\mathcal{R}}$, $\to^*_{\mathcal{R}}$ *denote rewrite sequences of n steps, of at least n steps, and of arbitrary many steps, respectively (where* $n \geq 0$). *A term t is terminating for $\mathcal{R}$ if there is no infinite sequence of the form* $t \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$ *A TRS $\mathcal{R}$ is terminating if all terms are terminating for $\mathcal{R}$.*

The above definition of $\to_{\mathcal{R}}$ differs from the usual rewrite relation in two aspects:

(i) We only permit instantiations of rule variables by constructor terms.
(ii) We use substitutions with possibly non-rational infinite terms.

---

[4]In standard term rewriting, one usually requires $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ for all rules $\ell \to r$. The reason is that otherwise the standard rewrite relation is never well founded. However, the *infinitary constructor rewrite relation* defined here can be well founded even if $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$.

In Example 3.2 and 3.3 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. TRANSFORMING LOGIC PROGRAMS INTO TERM REWRITE SYSTEMS

Now we modify the transformation of logic programs into TRSs from Section 1 to make it applicable for *arbitrary* (possibly non-well-moded) programs as well. We present the new transformation in Section 3.1 and prove its soundness in Section 3.2. Later in Section 6 we will formally prove that the classical transformation is strictly subsumed by our new one.

3.1 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 3.1 *(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 $p_{in}(\vec{s}) \rightarrow p_{out}(\vec{s})$.
- For each clause c of the form $p(\vec{s}) :- p_1(\vec{s}_1), \ldots, p_k(\vec{s}_k)$ in $\mathcal{P}$, $\mathcal{R}_\mathcal{P}$ contains:

$$p_{in}(\vec{s}) \rightarrow u_{c,1}(p_{1_{in}}(\vec{s}_1), \mathcal{V}(\vec{s}))$$
$$u_{c,1}(p_{1_{out}}(\vec{s}_1), \mathcal{V}(\vec{s})) \rightarrow u_{c,2}(p_{2_{in}}(\vec{s}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s}_1))$$
$$\cdots$$
$$u_{c,k}(p_{k_{out}}(\vec{s}_k), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{s}_1) \cup \ldots \cup \mathcal{V}(\vec{s}_{k-1})) \rightarrow p_{out}(\vec{s})$$

The following two examples motivate the need for infinitary constructor rewriting in Definition 3.1, i.e., they motivate Modifications (i) and (ii) in Section 2.2.

Example 3.2. *For the logic program of Example 1.1, the transformation of Definition 3.1 yields the following TRS.*

$$\mathsf{p}_{in}(X, X) \rightarrow \mathsf{p}_{out}(X, X) \tag{1}$$
$$\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{g}(Y)) \rightarrow \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \tag{2}$$
$$\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \rightarrow \mathsf{u}_2(\mathsf{p}_{in}(Z, \mathsf{g}(Y)), X, Y, Z) \tag{3}$$
$$\mathsf{u}_2(\mathsf{p}_{out}(Z, \mathsf{g}(Y)), X, Y, Z) \rightarrow \mathsf{p}_{out}(\mathsf{f}(X), \mathsf{g}(Y)) \tag{4}$$

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 $\ell \rightarrow r$ 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 1.2, now all terms $\mathsf{p}_{in}(t_1, t_2)$ 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 1.3, one obtains a similar TRS where $g(Y)$ *is replaced by* $g(W)$ *in the right-hand side of Rule (3) and the left-hand side of Rule (4). Again, all terms* $p_{in}(t_1, t_2)$ *are terminating for this TRS provided that* t_1 *is finite. Thus, we can now handle programs where the classical transformation of [Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Ohlebusch 2001] failed, cf. Goals (A) and (B) in Section 1.2.*

Derivations in logic programming use *unification*, while rewriting is defined by *matching*. Example 3.3 shows that to simulate unification by matching, we have to consider substitutions with infinite and even non-rational terms (Modification (ii)).

Example 3.3. Let $\mathcal{P}$ be ordered$(\text{cons}(X, \text{cons}(\text{s}(X), XS))) \mathrel{:-} \text{ordered}(\text{cons}(\text{s}(X), XS))$. *If one only considers rewriting with finite or rational terms, then the transformed TRS* $\mathcal{R}_\mathcal{P}$ *is terminating. However, the query* ordered(YS) *is not terminating for* $\mathcal{P}$. *Thus, to obtain a sound approach,* $\mathcal{R}_\mathcal{P}$ *must also be non-terminating. Indeed, the term* $t = \text{ordered}_{in}(\text{cons}(X, \text{cons}(\text{s}(X), \text{cons}(\text{s}^2(X), \ldots))))$ *is non-terminating with* $\mathcal{R}_\mathcal{P}$*'s rule* ordered$_{in}(\text{cons}(X, \text{cons}(\text{s}(X), XS))) \rightarrow \text{u}(\text{ordered}_{in}(\text{cons}(\text{s}(X), XS)), X, XS)$. *The non-rational term* t *corresponds to the infinite derivation with the query* ordered(YS).

3.2   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 3.4 (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* $p_{in}(\vec{t})\sigma \rightarrow^{\geq n}_{\mathcal{R}_\mathcal{P}} p_{out}(\vec{t})\sigma$. *Otherwise, if* Q *is "*$q(\vec{v}), \ldots$*", then* $p_{in}(\vec{t})\sigma \rightarrow^{\geq n}_{\mathcal{R}_\mathcal{P}} r$ *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 $p_{in}(\vec{t}) \rightarrow^0_{\mathcal{R}_\mathcal{P}} p_{in}(\vec{t})$.

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 $p_{in}(\vec{t})\theta_1 = p_{in}(\vec{s})\theta_1 \rightarrow_{\mathcal{R}_\mathcal{P}} p_{out}(\vec{s})\theta_1 = p_{out}(\vec{t})\theta_1$ where $\sigma = \theta_1$.

Finally, let $Q_1 \neq \square$. Thus, c_1 is $p(\vec{s}) \mathrel{:-} p_1(\vec{s_1}), \ldots, p_k(\vec{s_k})$ 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 $p_{j_{in}}(\vec{s_j})\sigma_0 \ldots \sigma_j \rightarrow^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\vec{s_j})\sigma_0 \ldots \sigma_j$ and thus also $p_{j_{in}}(\vec{s_j})\sigma \rightarrow^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\vec{s_j})\sigma$. Moreover, if $Q = \square$ then we also have $p_{i_{in}}(\vec{s_i})\sigma \rightarrow^{\geq n_i}_{\mathcal{R}_\mathcal{P}} p_{i_{out}}(\vec{s_i})\sigma$ where $i = k$. Otherwise, if Q is "$q(\vec{v}), \ldots$", then the

induction hypothesis implies $p_{i_{in}}(\vec{s_i})\sigma \to_{\mathcal{R}_{\mathcal{P}}}^{\geq n_i} r'$, where r' contains $q_{in}(\vec{v})$. Thus

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

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

$$
\begin{aligned}
u_{c_1,i}(p_{i_{in}}(\vec{s_i}),\mathcal{V}(\vec{s})\cup\ldots\cup\mathcal{V}(\vec{s_{i-1}}))\sigma \;\to_{\mathcal{R}_{\mathcal{P}}}^{\geq n_i} &\;\; u_{c_1,i}(p_{i_{out}}(\vec{s_i}),\mathcal{V}(\vec{s})\cup\ldots\cup\mathcal{V}(\vec{s_{i-1}}))\sigma \\
\to_{\mathcal{R}_{\mathcal{P}}} &\;\; p_{out}(\vec{s})\sigma \;\;=\;\; p_{out}(\vec{t})\sigma.
\end{aligned}
$$

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

$$
u_{c_1,i}(p_{i_{in}}(\vec{s_i}),\mathcal{V}(\vec{s})\cup\ldots\cup\mathcal{V}(\vec{s_{i-1}}))\sigma \;\to_{\mathcal{R}_{\mathcal{P}}}^{\geq n_i}\; u_{c_1,i}(r',\mathcal{V}(\vec{s})\sigma\cup\ldots\cup\mathcal{V}(\vec{s_{i-1}})\sigma). \quad\square
$$

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 3.5 (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 $\square$ which again contradicts the infiniteness of the derivation. $\square$

We use *argument filters* to characterize the classes of queries whose termination we want to analyze. Related definitions can be found in, e.g., [Arts and Giesl 2000; Leuschel and Sørensen 1996].

Definition 3.6 (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 $\pi(f) = \{i_1,\ldots,i_k\}$ 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 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".

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 1.1, 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 [Arts and Zantema 1995; Chtourou and Rusinowitch 1993; Ganzinger and Waldmann 1993; Ohlebusch 2001] 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 1.4.

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 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 3.7 (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 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 $\mathcal{P}$.*

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 3.5 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 3.4 we get $p_{in}(\vec{t})\sigma_0 \to^{\geq i_1}_{\mathcal{R}_\mathcal{P}} r_1$, where r_1 contains the subterm $q_{1_{in}}(\vec{v}_1)$.

By Lemma 3.5 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 3.4 we get $p_{in}(\vec{t})\sigma_0\sigma_1 \to^{\geq i_1}_{\mathcal{R}_\mathcal{P}} r_1\sigma_1 \to^{\geq i_2}_{\mathcal{R}_\mathcal{P}} r_2$, 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.[5] 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-terminating for $\mathcal{R}_\mathcal{P}$:

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

---

[5]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 $t\sigma_0\sigma_1 \ldots$ iff *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 $f \in \Sigma$ iff 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 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 $X\sigma_0\sigma_1 \ldots = Y\sigma_0\sigma_1 \ldots = Z_\infty$.

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. $\square$

4.   TERMINATION OF INFINITARY CONSTRUCTOR REWRITING

One of the most powerful methods for automated termination analysis of rewriting is the *dependency pair* (DP) method [Arts and Giesl 2000] 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 3.1.

Moreover, dependency pairs are a general framework that permits the integration of *any* termination technique for TRSs [Giesl et al. 2005, 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 Section 4.1, we adapt the notions and the main termination criterion of the dependency pair method to infinitary constructor rewriting and in Section 4.2 we show how to automate this criterion by adapting the "DP processors" of the DP framework.

4.1   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 4.1 *(Dependency Pair [Arts and Giesl 2000]).* The set of *dependency pairs* for a TRS $\mathcal{R}$ is $DP(\mathcal{R}) = \{\ell^\sharp \rightarrow t^\sharp \mid \ell \rightarrow r \in \mathcal{R},\ t \text{ is a subterm of } r,\ \mathrm{root}(t) \in \Sigma_D\}$.

Example 4.2. In the TRS $\mathcal{R}$ of Example 3.2, we have $\Sigma_D = \{\mathsf{p}_{in}, \mathsf{u}_1, \mathsf{u}_2\}$ and $DP(\mathcal{R})$ is

$$\mathsf{P}_{in}(\mathsf{f}(X), \mathsf{g}(Y)) \rightarrow \mathsf{P}_{in}(\mathsf{f}(X), \mathsf{f}(Z)) \tag{5}$$

$$\mathsf{P}_{in}(\mathsf{f}(X), \mathsf{g}(Y)) \rightarrow \mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \tag{6}$$

$$\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \rightarrow \mathsf{P}_{in}(Z, \mathsf{g}(Y)) \tag{7}$$

$$\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y) \rightarrow \mathsf{U}_2(\mathsf{p}_{in}(Z, \mathsf{g}(Y)), X, Y, Z) \tag{8}$$

While Definition 4.1 is from [Arts and Giesl 2000], 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 4.3 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 4.3, 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 4.3 *(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* iff

- for all $i \geq 1$, there are substitutions $\sigma_i : \mathcal{V} \to \mathcal{T}^\infty(\Sigma_C, \mathcal{V})$ such that $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$, 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 $t_i\sigma_i = q_0 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} q_m = s_{i+1}\sigma_{i+1}$, 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 4.2, "(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 $f(f(\ldots))$, then there is also an infinite chain "(6), (7), (6), (7), ..."

For termination of a program $\mathcal{P}$, by Theorem 3.7 we have to show that if $\pi(p_{in}(\vec{t}))$ 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 $\mathcal{R}_\mathcal{P}$), then $p_{in}(\vec{t})$ is terminating for $\mathcal{R}_\mathcal{P}$. Theorem 4.4 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 $\mathcal{P}$).

THEOREM 4.4 (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.[6] 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 terminating for $\mathcal{R}$.*

PROOF. Assume there is a non-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

---

[6]To see why the variable condition is needed in Theorem 4.4, let $\mathcal{R} = \{g(X) \to f(X), f(s(X)) \to f(X)\}$ and $\pi = \pi'$ where $\pi'(g) = \varnothing$, $\pi'(f) = \pi'(F) = \pi'(s) = \{1\}$. $\mathcal{R}$'s first rule violates the variable condition: $\mathcal{V}(\pi'(f(X))) = \{X\} \not\subseteq \mathcal{V}(\pi'(g(X))) = \varnothing$. There is no infinite chain, since π' does not allow us to instantiate the variable X in the dependency pair $F(s(X)) \to F(X)$ by an infinite term. Nevertheless, there is a non-terminating term $g(s(s(\ldots)))$ which is filtered to a finite ground term $\pi'(g(s(s(\ldots)))) = g$.

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-terminating. Let r' denote the smallest such subterm (i.e., for all proper subterms r'' of r', the term $r''\sigma_1$ is 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 performed again. Repeating this construction infinitely many times results in an infinite chain.   □

The following corollary combines Theorem 3.7 and Theorem 4.4. It describes how we use the DP method for proving termination of logic programs.

COROLLARY 4.5 (TERMINATION OF LOGIC PROG. 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 4.6. *We want to prove termination of Example 1.1 for all queries Q where $\pi(Q)$ is finite and ground for the filter $\pi(h) = \{1\}$ for all $h \in \{p, f, g\}$. By Corollary 4.5, 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 3.2 and $DP(\mathcal{R})$ are Rules $(5) - (8)$ from Example 4.2. The filter π' has to satisfy $\pi'(h) \subseteq \pi(h) = \{1\}$ for $h \in \{f, g\}$ and moreover, $\pi'(p_{in})$ and $\pi'(P_{in})$ must be subsets of $\pi(p_{in}) = \pi(P_{in}) = \pi(p) = \{1\}$. 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 5. In our example, we use $\pi'(p_{in}) = \pi'(P_{in}) = \pi'(f) = \pi'(g) = \{1\}$, $\pi'(p_{out}) = \pi'(u_1) = \pi'(U_1) = \{1, 2\}$, and $\pi'(u_2) = \pi'(U_2) = \{1, 2, 4\}$. For the non-well-moded Example 1.3 we choose $\pi'(g) = \varnothing$ instead to satisfy the variable condition.*

So to automate the criterion of Corollary 4.5, we have to tackle two problems:

(I) We start with a given filter π which describes 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 4.4 is fulfilled and that the termination proof is "likely to succeed". This problem will be discussed in Section 5.

(II) For the chosen 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.

4.2 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 [Giesl et al. 2005] 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 *finite*. 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 finite whenever all DP problems in *Proc*(d) are finite. 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 termination is proved.

In Theorem 4.9, 4.11, and 4.13 we will recapitulate three of the most important existing DP processors [Giesl et al. 2005] 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 4.7 *(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 $u \to v$ iff $CAP(t)$ 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 4.8. *For the DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ from Example 4.6 we obtain:*

$$(5) \longleftarrow (7) \ \overset{\frown}{\underset{\smile}{}} \ (6) \longrightarrow (8)$$
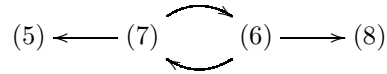
For example, there is an arc $(6) \to (7)$, as $CAP(\mathsf{U}_1(\mathsf{p}_{in}(\mathsf{f}(X), \mathsf{f}(Z)), X, Y)) = \mathsf{U}_1(V, X, Y)$ unifies with $\mathsf{U}_1(\mathsf{p}_{out}(\mathsf{f}(X'), \mathsf{f}(Z')), X', Y')$ by instantiating the arguments of U_1 with finite terms. But there are 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 4.8, 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 $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

THEOREM 4.9 (DEPENDENCY GRAPH PROCESSOR). *For a DP problem* $(\mathcal{D}, \mathcal{R}, \pi)$, *let Proc return* $\{(\mathcal{D}_1, \mathcal{R}, \pi), \ldots, (\mathcal{D}_n, \mathcal{R}, \pi)\}$ *where* $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *are the sets of nodes of the SCCs in the estimated dependency graph. Then Proc is sound.*

PROOF. We prove that if $s \rightarrow t$, $u \rightarrow v$ is a chain, then there is an arc from $s \rightarrow t$ to $u \rightarrow v$ in the estimated dependency graph. This suffices for Theorem 4.9, 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 finite DP problems, then so is $(\mathcal{D}, \mathcal{R}, \pi)$.

Let $s \rightarrow t$, $u \rightarrow v$ be a $(\mathcal{D}, \mathcal{R}, \pi)$-chain, i.e., $t\sigma_1 \rightarrow^*_{\mathcal{R}} u\sigma_2$ for some constructor substitutions σ_1, σ_2 where $\pi(t\sigma_1)$ and $\pi(u\sigma_2)$ are finite. Let $pos_1, \ldots, pos_n$ be the top positions where t has defined symbols. Then $CAP(t) = t[Y_1]_{pos_1} \ldots [Y_n]_{pos_n}$ 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 4.7 there is indeed an arc from $s \rightarrow t$ to $u \rightarrow v$. □

Example 4.10. *In Example 4.8, the only SCC 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 $f(\ldots s \ldots) \succsim f(\ldots t \ldots)$ 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. [Dershowitz 1987]).

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 $\succ$).[7] Requiring $\pi(\ell) \succsim \pi(r)$ for all $\ell \rightarrow r \in \mathcal{R}$ ensures that in chains $s_1 \rightarrow t_1, s_2 \rightarrow t_2, \ldots$ with $t_i\sigma_i \rightarrow^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ as in Definition 4.3, 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 $s \rightarrow t \in \mathcal{D}$ 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 $\mathcal{D}_{\succ_\pi} = \{s \rightarrow t \in \mathcal{D} \mid \pi(s) \succ \pi(t)\}$.

THEOREM 4.11 (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*

---

[7]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.

PROOF. Let $(\mathcal{D}, \mathcal{R}, \pi)$ be infinite, i.e., there is an infinite $(\mathcal{D}, \mathcal{R}, \pi)$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$. 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

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

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 $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$ implies $\pi(t_i\sigma_i) \to_{\pi(\mathcal{R})}^* \pi(s_{i+1}\sigma_{i+1})$. 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 $\pi(s_1\sigma_1) \underset{(\succsim)}{\succ} \pi(t_1\sigma_1) \succsim \pi(s_2\sigma_2) \underset{(\succsim)}{\succ} \pi(t_2\sigma_2) \succsim \ldots$ 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. □

Example 4.12. For the DP problem $(\{(6), (7)\}, \mathcal{R}, \pi')$ in Example 4.10, one can easily find a reduction pair[8] where the dependency pair (7) is strictly decreasing and where (6) and all rules are weakly decreasing after applying the filter π':

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

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 4.6 is finite and thus, the logic program from Example 1.1 terminates for all queries Q where $\pi(Q)$ is finite and ground. Note that termination of the non-well-moded program from Example 1.3 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 4.9 and 4.11, many other existing DP processors [Giesl et al. 2005] 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.[9] Since any termination technique for TRSs can immediately be formulated as a DP processor [Giesl et al. 2005, Thm. 36], now any termination technique for ordinary rewriting can be directly used for infinitary constructor rewriting as well.

THEOREM 4.13 (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.

---

[8]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 $|\mathsf{g}(t_1)| = 0$.
[9]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 [Giesl et al. 2005; Giesl et al. 2006]. These processors are usually more powerful than those for ordinary rewriting. The framework presented in [Giesl et al. 2005] even supports constructor rewriting.

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 4.3, 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 $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$ implies $\pi(t_i)\pi(\sigma_i) = \pi(t_i\sigma_i) \to_{\pi(\mathcal{R})}^* \pi(s_{i+1}\sigma_{i+1}) = \pi(s_{i+1})\pi(\sigma_{i+1})$. Moreover, by Definition 4.3, all terms in the rewrite sequence $\pi(t_i\sigma_i) \to_{\pi(\mathcal{R})}^* \pi(s_{i+1}\sigma_{i+1})$ are finite. □

5. REFINING THE ARGUMENT FILTER

In Section 3 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 4.5.

The transformation itself is trivial to automate and as shown in Section 4, 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 suitable argument filter from the user input, cf. Task (I) at the end of Section 4.1. After presenting the general algorithm to refine argument filters in Section 5.1, we introduce suitable heuristics in Sections 5.2 and 5.3. Finally, we extend the general algorithm for the refinement of argument filters by integrating a mode analysis based on argument filters in Section 5.4. 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 Sections 5.2 and 5.3 and the power of our extension in Section 5.4 will be evaluated empirically in Section 7.

5.1 Refinement Algorithm for Argument Filters

In our approach of Corollary 4.5, the user supplies an initial argument filter π to describe the set of queries whose termination should be proved. However, the variable condition $\mathcal{V}(\pi(r)) \subseteq \mathcal{V}(\pi(\ell))$ for all $\ell \to r \in DP(\mathcal{R}_{\mathcal{P}}) \cup \mathcal{R}_{\mathcal{P}}$ 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 4.5 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 4.6, 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 5.1. *We consider the logic program of Example 1.1. As shown in Example 3.2, 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 $m(\mathsf{p}, 1) = m(\mathsf{p}, 2) =$

in, i.e., to the argument filter π with $\pi(\mathsf{p}) = \{1,2\}$.

In Corollary 4.5, 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 5.2 *(Refinement Heuristic).* A *refinement heuristic* is a 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 $\mathrm{root}(t|_{pos'}) = f$.

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 $\mathcal{R}_\mathcal{P}$.

Input: argument filter π, refinement heuristic ρ, TRS $\mathcal{R}_\mathcal{P}$
Output: refined argument filter π' 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 π' by removing i from $\pi'(f)$, i.e., $\pi'(f) := \pi'(f) \setminus \{i\}$.
        For all other symbols from $\Sigma_\mathcal{P}$, π' remains unchanged.

   2.3. Go back to **Step 2**.

Algorithm 1: General Refinement Algorithm

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.

5.2   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 5.3 *(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:

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

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

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

So if t is again the term $u_1(p_{in}(f(X), f(Z)), X, Y)$, then the innermost refinement heuristic would result in $\rho_{im}(t, 121) = (f, 1)$ and the outermost refinement heuristic gives $\rho_{om}(t, 121) = (u_1, 1)$.

Both heuristics defined above are simple but problematic, as shown in Example 5.4. 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 5.4. *Consider again the logic program of Example 1.1 which was transformed into the following TRS in Example 3.2.*

$$p_{in}(X, X) \rightarrow p_{out}(X, X) \tag{1}$$
$$p_{in}(f(X), g(Y)) \rightarrow u_1(p_{in}(f(X), f(Z)), X, Y) \tag{2}$$
$$u_1(p_{out}(f(X), f(Z)), X, Y) \rightarrow u_2(p_{in}(Z, g(Y)), X, Y, Z) \tag{3}$$
$$u_2(p_{out}(Z, g(Y)), X, Y, Z) \rightarrow p_{out}(f(X), g(Y)) \tag{4}$$

As shown in Example 4.2 we obtain the following dependency pairs for the above

rules.

$$P_{in}(f(X), g(Y)) \rightarrow P_{in}(f(X), f(Z)) \tag{5}$$

$$P_{in}(f(X), g(Y)) \rightarrow U_1(p_{in}(f(X), f(Z)), X, Y) \tag{6}$$

$$U_1(p_{out}(f(X), f(Z)), X, Y) \rightarrow P_{in}(Z, g(Y)) \tag{7}$$

$$U_1(p_{out}(f(X), f(Z)), X, Y) \rightarrow U_2(p_{in}(Z, g(Y)), X, Y, Z) \tag{8}$$

As in Example 5.1 we want to prove termination of $p(t_1, t_2)$ 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 $\rho_{im}(U_1(p_{in}(f(X), f(Z)), X, Y), 121) = (f, 1)$. This requires us to filter away the only argument of f, i.e., $\pi'(f) = \varnothing$. 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., $\pi'(P_{in}) = \{2\}$. Due to the DP (6), we now also have to remove the second argument X of U_1 i.e., $\pi'(U_1) = \{1, 3\}$. 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 $f(f(\ldots))$. 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 $\rho_{om}(U_1(p_{in}(f(X), f(Z)), X, Y), 121) = (U_1, 1)$, i.e., $\pi'(U_1) = \{2, 3\}$. Considering the third DP (7) we have to filter away the first argument of P_{in}, i.e., $\pi'(P_{in}) = \{2\}$. Due to the DP (6), we now also have to remove the second argument of U_1, i.e., $\pi'(U_1) = \{3\}$. 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 5.5 *(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:

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

Example 5.6. *Reconsider Example 5.4. Using the improved outermost refinement heuristic, for the second rule (2) we get $\rho_{om'}(u_1(p_{in}(f(X), f(Z)), X, Y), 121) = \rho_{om'}(p_{in}(f(X), f(Z)), 21) = (p_{in}, 2)$ requiring us to filter away the second argument of p_{in}, i.e., $\pi'(p_{in}) = \{1\}$. Consequently, the algorithm filters away the third arguments of both u_1 and u_2, i.e., $\pi'(u_1) = \{1, 2\}$ and $\pi'(u_2) = \{1, 2, 4\}$. Now the variable condition holds for $\mathcal{R}_{\mathcal{P}}$. Therefore, by defining $\pi'(P_{in}) = \pi'(p_{in})$, $\pi'(u_1) = \pi'(U_1)$, and $\pi'(u_2) = \pi'(U_2)$, the variable condition also holds for $DP(\mathcal{R}_{\mathcal{P}})$. (As mentioned at the end of Section 5.1, 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 4.6 and as shown in Section 4.2 one can now easily prove termination.*

5.3 Type-Based Refinement Heuristic

The improved outermost heuristic from Section 5.2 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., [Bossi et al. 1992; Bruynooghe et al. 2007; Decorte et al. 1993; Martin et al. 1996]. 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 5.7. *Now we regard the logic program from Example 1.3. The rules after the transformation of Definition 3.1 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 5.6 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 5.4, 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*, if the arguments on position i have the same "type" as the whole term $f(\ldots)$ itself, cf. [Walther 1994]. 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 5.8 *(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 $\tau : \Sigma \cup \Delta \rightarrow \Theta^*$ 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 $Reflexive_\tau(f/n) = \{i \mid 1 \le i \le n \text{ and } \theta_i = \theta_{n+1}\}$.

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 [Bruynooghe et al. 2007], 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 [Bruynooghe et al. 2005; Charatonik and Podelski 1998; Gallagher and Puebla 2002; Janssens and Bruynooghe 1992; Lu 2000; Vaucheret and Bueno 2002], 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 [Bruynooghe et al. 1996; Cortesi and Filé 1999; Lagoon and Stuckey 2002], 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 5.9. *As an example, we compute a suitable type assignment for the logic program from Example 1.3:*

$$\mathsf{p}(X, X)$$
$$\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \ :\!- \ \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(W))$$

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 5.3, but now reflexive arguments of function symbols from Σ (i.e., from the original logic program) are not filtered away.

Definition 5.10 *(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_{tb}^τ is defined as follows:

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

Note that the heuristic ρ_{tb}^τ 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 at the end of Section 5.1, we only have to check the variable condition for the rules of $\mathcal{R}_\mathcal{P}$, but not for the dependency pairs.

Example 5.11. *We continue with the logic program from Example 1.3 and use the type assignment computed in Example 5.9 above. The rules after the transformation of Definition 3.1 are the following, cf. Example 5.7.*

$$\mathsf{p}_{in}(X, X) \to \mathsf{p}_{out}(X, X) \tag{1}$$

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

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

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

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

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

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:

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

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 4.6. With this filter, one can easily prove termination of the program, cf. Section 4.2.*

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

For a function symbol f/n as above, 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 1.3 and 5.9, 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 5.14 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 5.13 *(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* $\rho_{tb'}^\tau$ is defined as follows:

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

Example 5.14. *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 iff* 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.*

$$
\begin{aligned}
&\mathsf{nat}(0) \\
&\mathsf{nat}(\mathsf{s}(X)) && :\!- \ \mathsf{nat}(X) \\
&\mathsf{inv}(\mathsf{neg}(X), \mathsf{pos}(X)) \\
&\mathsf{inv}(\mathsf{pos}(X), \mathsf{neg}(X)) \\
&\mathsf{safeinv}(X, \mathsf{neg}(Y)) && :\!- \ \mathsf{inv}(X, \mathsf{neg}(Y)), \mathsf{nat}(Y) \\
&\mathsf{safeinv}(X, \mathsf{pos}(Y)) && :\!- \ \mathsf{inv}(X, \mathsf{pos}(Y)), \mathsf{nat}(Y)
\end{aligned}
$$

The rules after the transformation of Definition 3.1 are:

$$\mathsf{nat}_{in}(0) \rightarrow \mathsf{nat}_{out}(0) \tag{12}$$

$$\mathsf{nat}_{in}(\mathsf{s}(X)) \rightarrow \mathsf{u}_1(\mathsf{nat}_{in}(X), X) \tag{13}$$

$$\mathsf{u}_1(\mathsf{nat}_{out}(X), X) \rightarrow \mathsf{nat}_{out}(\mathsf{s}(X)) \tag{14}$$

$$\mathsf{inv}_{in}(\mathsf{neg}(X), \mathsf{pos}(X)) \rightarrow \mathsf{inv}_{out}(\mathsf{neg}(X), \mathsf{pos}(X)) \tag{15}$$

$$\mathsf{inv}_{in}(\mathsf{pos}(X), \mathsf{neg}(X)) \rightarrow \mathsf{inv}_{out}(\mathsf{pos}(X), \mathsf{neg}(X)) \tag{16}$$

$$\mathsf{safeinv}_{in}(X, \mathsf{neg}(Y)) \rightarrow \mathsf{u}_2(\mathsf{inv}_{in}(X, \mathsf{neg}(Y)), X, Y) \tag{17}$$

$$\mathsf{u}_2(\mathsf{inv}_{out}(X, \mathsf{neg}(Y)), X, Y) \rightarrow \mathsf{u}_3(\mathsf{nat}_{in}(Y), X, Y) \tag{18}$$

$$\mathsf{u}_3(\mathsf{nat}_{out}(Y), X, Y) \rightarrow \mathsf{safeinv}_{out}(X, \mathsf{neg}(Y)) \tag{19}$$

$$\mathsf{safeinv}_{in}(X, \mathsf{pos}(Y)) \rightarrow \mathsf{u}_4(\mathsf{inv}_{in}(X, \mathsf{pos}(Y)), X, Y) \tag{20}$$

$$\mathsf{u}_4(\mathsf{inv}_{out}(X, \mathsf{pos}(Y)), X, Y) \rightarrow \mathsf{u}_5(\mathsf{nat}_{in}(Y), X, Y) \tag{21}$$

$$\mathsf{u}_5(\mathsf{nat}_{out}(Y), X, Y) \rightarrow \mathsf{safeinv}_{out}(X, \mathsf{pos}(Y)) \tag{22}$$

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 $m(\mathsf{safeinv}, 1) = \textbf{\textit{in}}$ and $m(\mathsf{safeinv}, 2) = \textbf{\textit{out}}$. 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 $\tau(\mathsf{s}) = (b, b)$, $\tau(0) = (b)$, 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 $Constructors_\tau(b) = \{\mathsf{s}, 0\}$. This is a recursive data structure since s has an unbounded argument: $1 \in Reflexive_\tau(\mathsf{s}) \subseteq Unbounded_\tau(\mathsf{s})$. Thus, while neg's first argument position of type b is not reflexive, it is still unbounded, i.e., $1 \in Unbounded_\tau(\mathsf{neg})$. 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.

5.4   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 5.15. *Consider the following logic program for rotating a list taken from [Codish 2007]. Let $\mathcal{P}$ be the* append-*program consisting of the clauses from Example 1.4 and the new clause*

$$\text{rotate}(N, O) :- \text{append}(L, M, N), \text{append}(M, L, O) \qquad (23)$$

with the moding $m(\text{rotate}, 1) = \textbf{in}$ and $m(\text{rotate}, 2) = \textbf{out}$. 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.

$$\text{append}_{in}([\,], M, M) \rightarrow \text{append}_{out}([\,], M, M) \qquad (24)$$

$$\text{append}_{in}(\bullet(X, L), M, \bullet(X, N)) \rightarrow \text{u}_1(\text{append}_{in}(L, M, N), X, L, M, N) \quad (25)$$

$$\text{u}_1(\text{append}_{out}(L, M, N), X, L, M, N) \rightarrow \text{append}_{out}(\bullet(X, L), M, \bullet(X, N)) \qquad (26)$$

$$\text{rotate}_{in}(N, O) \rightarrow \text{u}_2(\text{append}_{in}(L, M, N), N, O) \qquad (27)$$

$$\text{u}_2(\text{append}_{out}(L, M, N), N, O) \rightarrow \text{u}_3(\text{append}_{in}(M, L, O), L, M, N, O) \quad (28)$$

$$\text{u}_3(\text{append}_{out}(M, L, O), L, M, N, O) \rightarrow \text{rotate}_{out}(N, O) \qquad (29)$$

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),[10] the only refined argument filter which would satisfy the variable condition of Corollary 4.5 is the one where $\pi(\text{append}_{in}) = \varnothing$.[11] As we can expect, for the queries described by this filter, the append-*program is not terminating and, thus, our new approach fails, too.*

The common solution [Apt 1997] 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 labelled 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 $m(\text{append}^{\{3\}}, 3) = m(\text{append}^{\{1,2\}}, 1) = m(\text{append}^{\{1,2\}}, 2) = \textbf{in}$ and $m(\text{append}^{\{3\}}, 1) = m(\text{append}^{\{3\}}, 2) = m(\text{append}^{\{1,2\}}, 3) = \textbf{out}$. In our example, termination of the resulting logic program can easily be shown using both the classical transformation from Section 1.1 or our new transformation:*

$$
\begin{aligned}
\text{rotate}(N, O) \quad &:- \quad \text{append}^{\{3\}}(L, M, N), \text{append}^{\{1,2\}}(M, L, O) \\
\text{append}^{\{3\}}([\,], M, M) \\
\text{append}^{\{3\}}([X|L], M, [X|N]) \quad &:- \quad \text{append}^{\{3\}}(L, M, N) \\
\text{append}^{\{1,2\}}([\,], M, M) \\
\text{append}^{\{1,2\}}([X|L], L, [X|N]) \quad &:- \quad \text{append}^{\{1,2\}}(L, M, N)
\end{aligned}
$$

---

[10]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).

[11]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 $\pi(\text{APPEND}_{in}) = \varnothing$. Hence, the termination proof attempt would fail as well.

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 5.16. *Consider again the logic program $\mathcal{P}$ from Example 5.15 which is translated to the TRS $\mathcal{R}_{\mathcal{P}} = \{(24), \dots, (29)\}$. This time we want to show termination for all queries of the form* 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 $\bullet(r_1, \bullet(r_2, \dots \bullet(r_n, [\,]) \dots))$ where the r_i can be arbitrary terms possibly containing variables.*[12]

To specify these queries, we use the initial argument filter π with $\pi($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 $m($rotate$, 1) = m($rotate$, 2) = \boldsymbol{out}$. Therefore, in the calls to* append*, all argument positions would be considered as "$\boldsymbol{out}$". As a consequence, no renamed-apart copies of clauses would be created and the termination proof would fail.*

In general, our refinement algorithm from Section 5.1 (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 5.15 and 5.16 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 5.16 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

---

[12]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.

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

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 rotate(t_1, t_2) where t_1 is a finite list.

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

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

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

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, unlabelled symbols of the form p_{in}/n
are treated as if they were labelled with $I = \{1, \ldots, n\}$.

    • If $f = p_{in}^{I}$, then we must have $r = u(p_{in}^{I}(\ldots), \ldots)$ 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 $\overline{r} = u(p_{in}^{I \setminus \{i\}}(\ldots), \ldots)$.

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

      ◦ Let $\ell' \to r'$ be the rule in $\mathcal{R}'_{\mathcal{P}}$ with $\ell' = u(p_{out}^{I}(\ldots), \ldots)$. 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 $\overline{\ell'} = u(p_{out}^{I \setminus \{i\}}(\ldots), \ldots)$.

    • Otherwise (i.e., if f does not have the form p_{in} or p_{in}^{I}), then
modify π' 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

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 $\{1, \ldots, n\}$ 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 5.16. 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 labelled with $\{1\}$.

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

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

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

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 labelled copies of the rules resulting from those predicates p/n where $\pi(p) \subsetneq \{1, \dots, n\}$. In these rules, the root symbols of left- and right-hand sides are labelled 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 $t = f(t_1, \dots, t_n)$ and a set of argument positions $I \subseteq \mathbb{N}$, let t^I denote $f^I(t_1, \dots, t_n)$. 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 labelled 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_{in}^I. 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_{in}^I, the label I describes those arguments that are not filtered away. However, this does not hold for the other labelled 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 labelled) symbol of the form p_{in} or p_{in}^I, 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 labelled) symbol of the form p_{in} or p_{in}^I, 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 labelled 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 $\pi'(\text{rotate}_{in}^{\{1\}}) = \{1\}$ 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:

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

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_{in}^I, we use the second case of **Step 3.2**. Thus, we change π' such that $\pi'(\text{u}_1) = \{1, 2, 3, 4, 5\} \setminus \{2\} = \{1, 3, 4, 5\}$. 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):

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

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 $\pi'(\text{append}_{in})$, we introduce a new symbol $\text{append}_{in}^{\{2,3\}}$ with $\pi'(\text{append}_{in}^{\{2,3\}}) = \{2, 3\}$. Moreover, we replace the symbol append_{in} in the right-hand side of Rule (30) by the new symbol $\text{append}_{in}^{\{2,3\}}$. Thus, Rule (30) is modified to

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

To make sure that $\text{append}_{in}^{\{2,3\}}$ 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 $\mathcal{R}'_{\mathcal{P}}$:

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

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

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

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

So the original rule (31)

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

is replaced by the modified rule

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

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 labelled symbols $\mathsf{append}_{in}^{\{2,3\}}$, $\mathsf{u}_1^{\{2,3\}}$, and $\mathsf{append}_{out}^{\{2,3\}}$. On the unlabelled 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):

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

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 labelled by $\{3\}$:

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

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

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

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

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

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 labelled symbols $\mathsf{append}_{in}^{\{3\}}$, $\mathsf{u}_1^{\{3\}}$, and $\mathsf{append}_{out}^{\{3\}}$. On the unlabelled 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):

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

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):

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

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 labelled by $\{1,2\}$:

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

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

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

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

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

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 labelled
symbols $\mathsf{append}_{in}^{\{1,2\}}$, $\mathsf{u}_1^{\{1,2\}}$, and $\mathsf{append}_{out}^{\{1,2\}}$. On the unlabelled symbols, the argu-
ment 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 labelled
variants for every symbol. In our example, we obtain the following set of rules $\mathcal{R}'_\mathcal{P}$:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The refined argument filter π' is given by

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

Termination for $\mathcal{R}_{\mathcal{P}}'$ w.r.t. the terms specified by π' is now easy to show using our results from Section 4.

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 termination of the above TRS for those "start terms" $\mathsf{rotate}_{in}^{\{1\}}(\ldots)$ that are finite and ground under the filter π' and where the arguments of $\mathsf{rotate}_{in}^{\{1\}}$ do not contain any function symbols except $\bullet$ and $[]$. Since the rules for rotate_{in}, append_{in}, and

$\mathsf{append}_{in}^{\{2,3\}}$ (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 $\mathsf{rotate}_{in}^{\{1\}}$, $\mathsf{append}_{in}^{\{1,2\}}$, and $\mathsf{append}_{in}^{\{3\}}$, but the rules for rotate_{in}, append_{in}, and $\mathsf{append}_{in}^{\{2,3\}}$ are superfluous.

Note however that such superfluous copies of rules are never problematic for the termination analysis. If the rules for $\mathsf{append}_{in}^{\{3\}}$ are terminating for terms that are finite and ground under the filter π', then this also holds for the $\mathsf{append}_{in}^{\{2,3\}}$- and the append_{in}-rules, since here π' filters away less arguments. A corresponding statement holds for the connection between the $\mathsf{rotate}_{in}^{\{1\}}$- 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 3.7. So it is sufficient to prove that all terms in the set $S' = \{p_{in}^{\pi(p)}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi'(p_{in}^{\pi(p)}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$ are terminating w.r.t. the modified TRS $\mathcal{R}'_{\mathcal{P}}$. In Example 5.16, S' would be the set of all terms $\mathsf{rotate}_{in}^{\{1\}}(t_1, t_2)$ 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 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 4.5. 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 5.17 (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 $S' = \{p_{in}^{\pi(p)}(\vec{t}) \mid p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi'(p_{in}^{\pi(p)}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$. All terms $s \in S$ are terminating for $\mathcal{R}_{\mathcal{P}}$ if all terms $s' \in S'$ are 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 $\mathbb{S}^n = \{t \mid p_{in}(\vec{t}) \rightarrow_{\mathcal{R}_{\mathcal{P}}}^n t$ for some $p \in \Delta,\ \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \text{and}\ \pi(p_{in}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi}})\}$ and $\mathbb{S}' = \{t \mid p_{in}^{\pi(p)}(\vec{t}) \rightarrow_{\mathcal{R}'_{\mathcal{P}}}^* t$ for some $p \in \Delta, \vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \text{and}\ \pi'(p_{in}^{\pi(p)}(\vec{t})) \in \mathcal{T}(\Sigma_{\mathcal{P}_{\pi'}})\}$

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

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

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

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.

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

So $s \in S$ and there is a rule $\ell \to r \in \mathcal{R}_\mathcal{P}$ 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 $\ell^{\pi(p)} \to r' \in \mathcal{R}'_\mathcal{P}$ where $Unlab(r') = r$. We define t' to be $r'\sigma$. Then we clearly have $s' = \ell^{\pi(p)}\sigma \to_{\mathcal{R}'_\mathcal{P}} r'\sigma = t'$ and $Unlab(t') = t$.

Case 2: $n \geq 1$ and $s = u_{c,i}(\overline{s}, \vec{q})$, $\overline{s} \to_{\mathcal{R}_\mathcal{P}} \overline{t}$, $t = u_{c,i}(\overline{t}, \vec{q})$

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 $p_{in}(\vec{s}) \to^*_{\mathcal{R}_\mathcal{P}} \overline{s}$, 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 $p_{in}^J(\vec{s}) \to^*_{\mathcal{R}'_\mathcal{P}} \overline{s'}$. Hence, $\overline{s'} \in \mathbb{S}'$ as well. Now the induction hypothesis implies that there exists a $\overline{t'}$ such that $\overline{s'} \to_{\mathcal{R}'_\mathcal{P}} \overline{t'}$ and $Unlab(\overline{t'}) = \overline{t}$. We define $t' = u_{c,i}^I(\overline{t'}, \vec{q})$. Then we clearly have $s' \to_{\mathcal{R}'_\mathcal{P}} t'$ and $Unlab(t') = t$.

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

Here, there exists a rule $\ell \to r \in \mathcal{R}_\mathcal{P}$ 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 $\ell' \to r' \in \mathcal{R}'_\mathcal{P}$ 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 $s' = \ell'\sigma \to_{\mathcal{R}'_\mathcal{P}} r'\sigma = t'$ 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-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 $s_i \to_{\mathcal{R}_\mathcal{P}} s_{i+1}$. 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 $s'_0 \to_{\mathcal{R}'_\mathcal{P}} s'_1$. Clearly, this also implies $s'_1 \in \mathbb{S}'$. By applying (56) repeatedly, we therefore obtain an infinite sequence of labelled terms $s'_0, s'_1, s'_2, \ldots$ with $s'_i \to_{\mathcal{R}'_\mathcal{P}} s'_{i+1}$. $\square$

6. FORMAL COMPARISON OF THE TRANSFORMATIONAL APPROACHES

In this section we formally compare the power of the classical transformation from Section 1.1 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 6.1 *(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 $\pi_m(p_{in}) = \pi_m(P_{in}) = \{i \mid m(p, i) = \boldsymbol{in}\}$ and $\pi_m(p_{out}) = \{i \mid m(p, i) = \boldsymbol{out}\}$. All other function symbols f from $\Sigma_{\mathcal{P}}$ are not filtered, i.e., $\pi_m(f/n) = \{1, \ldots, n\}$.

Example 6.2. *Regard again the well-moded logic program from Example 1.1.*

$$\mathsf{p}(X, X)$$
$$\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \;:\!-\; \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(Y))$$

We used the moding m with $m(\mathsf{p}, 1) = \boldsymbol{in}$ and $m(\mathsf{p}, 2) = \boldsymbol{out}$. 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 7), 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 6.3. *We continue with Example 6.2. The TRS $\mathcal{R}_{\mathcal{P}}$ resulting from our new transformation was given in Example 3.2:*

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

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

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

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 $\mathcal{R}_{\mathcal{P}}^{old}$ resulting from the classical transformation, cf. Example 1.2:

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

The following theorem shows that our approach (with Corollary 4.5) succeeds whenever the classical transformation of Section 1.1 yields a terminating TRS.

THEOREM 6.4 (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 $\mathcal{R}_{\mathcal{P}}^{old}$ be the result of applying the classical transformation of Section 1.1 and let $\mathcal{R}_{\mathcal{P}}$ be the result of our new transformation from Definition 3.1. 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 $\mathcal{R}_{\mathcal{P}}^{old}$ 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 $\mathcal{R}_{\mathcal{P}}^{old}$ implies that $\mathcal{R}_{\mathcal{P}}$ is 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 $p(\vec{s}, \vec{t}) :\!- p_1(\vec{s_1}, \vec{t_1}), \ldots, p_k(\vec{s_k}, \vec{t_k})$ in $\mathcal{P}$, the TRS $\pi_m(\mathcal{R}_{\mathcal{P}})$ contains:

$$
\begin{aligned}
p_{in}(\vec{s}) \;&\to\; u_{c,1}(p_{1_{in}}(\vec{s_1}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t})) \\
u_{c,1}(p_{1_{out}}(\vec{t_1}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t})) \;&\to\; u_{c,2}(p_{2_{in}}(\vec{s_2}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}) \cup \mathcal{V}(\vec{s_1}) \cup \mathcal{V}(\vec{t_1})) \\
&\;\;\vdots \\
u_{c,k}(p_{k_{out}}(\vec{t_k}), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}) \cup \mathcal{V}(\vec{s_1}) \cup \mathcal{V}(\vec{t_1}) \cup \ldots \cup \mathcal{V}(\vec{s_{k-1}}) \cup \mathcal{V}(\vec{t_{k-1}})) \;&\to\; p_{out}(\vec{t})
\end{aligned}
$$

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:

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

Hence, $\pi'_m(\mathcal{R}_\mathcal{P}) = \mathcal{R}_\mathcal{P}^{old}$. 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 4.13), 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

$$u \rightarrow v \in \pi'_m(DP(\mathcal{R}_\mathcal{P}))$$

iff there is a rule $\ell \rightarrow r \in \mathcal{R}_\mathcal{P}$ with $u = \pi'_m(\ell^\sharp), v = \pi'_m(t^\sharp)$
for a subterm t of r with defined root

iff there is a rule $\ell \rightarrow r \in \mathcal{R}_\mathcal{P}$ with $u = (\pi'_m(\ell))^\sharp, v = (\pi'_m(t))^\sharp$
for a subterm $\pi'_m(t)$ of $\pi'_m(r)$ with defined root

iff there is a rule $\ell \rightarrow r \in \pi'_m(\mathcal{R}_\mathcal{P})$ with $u = \ell^\sharp, v = t^\sharp$
for a subterm t of r with defined root

iff $u \rightarrow v \in DP(\pi'_m(\mathcal{R}_\mathcal{P}))$

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, if 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. [Arts and Giesl 2000, Thm. 6].

Hence by Theorem 4.4, 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 terminating w.r.t. $\mathcal{R}_\mathcal{P}$ (using infinitary constructor rewriting). $\square$

The reverse direction of the above theorem does not hold, though. As a counterexample, regard again the logic program from Example 1.1, cf. Example 6.3. As shown in Example 1.2, the TRS resulting from the classical transformation is not terminating. Still, for the filter π'_m from Example 6.3, there is no infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi'_m)$-chain and thus, our method of Corollary 4.5 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.

7. EXPERIMENTS AND DISCUSSION

We integrated our approach (including all refinements presented) in the termination tool AProVE [Giesl et al. 2006] which implements the DP framework. To evaluate our results, we tested AProVE against four other representative termination tools for logic programming: TALP [Ohlebusch et al. 2000] is the only other available tool based on transformational methods (it uses the classical transformation of Section 1.1), whereas Polytool [Nguyen and De Schreye 2007], TerminWeb [Codish and Taboch 1999], and cTI [Mesnard and Bagnara 2005] are based on direct approaches. Section 7.1 describes the results of our experimental evaluation and in Section 7.2 we discuss the limitations of our approach.

7.1 Experimental Evaluation

We ran the tools on a set of 296 examples in fully automatic mode.[13] This set includes all logic programming examples from the *Termination Problem Data Base* [TPDB 2007] which is used in the annual international *Termination Competition* [Marché and Zantema 2007]. It contains collections provided by the developers of several different tools including all examples from the experimental evaluation of [Bruynooghe et al. 2007]. 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 2.5. 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.

	AProVE	Polytool	TerminWeb	cTI	TALP
Successes	232	204	177	167	163
Failures	57	82	118	129	112
Timeouts	7	10	1	0	21

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.

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 7.2 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 [Nguyen et al. 2006; Serebrenik and De Schreye 2003; Tamary and Codish 2004]. In this example, this pre-processing step results in a trivially terminating logic program. Thus, if one combined this

---

[13] 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 [Marché and Zantema 2007]. To refine argument filters, this version of AProVE uses the refinement heuristic $\rho_{tb'}$ from Definition 5.13.

pre-processing with any of the other tools, then they would also be able to prove termination of this particular example.[14] 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 a more complex orderings. For instance, termination of the example SGST06/hbal_tree.pl can be proved using a recursive path order with status [Lescanne 1983] and termination of talp/apt/mergesort_ap.pl is shown using matrix orders [Endrullis et al. 2006].

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 [Payet and Mesnard 2006].

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 [Codish and Taboch 1999; Mesnard and Bagnara 2005], non-termination analysis [Payet and Mesnard 2006], handling numerical data structures [Serebrenik and De Schreye 2004; 2005b]) that can currently only be handled by direct techniques and tools.

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 [Nguyen and De Schreye 2005; 2007] 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 basic results of the dependency pair method to the logic programming setting [Nguyen et al. 2008]. Preliminary investigations with a prototypical implementation indicate that in this way, one can prove termination of several examples where the transformational approach with AProVE currently fails.

So transformational and direct approaches both have their advantages and the most powerful solution might be to combine direct tools like Polytool with a transformational prover like AProVE which is based on the contributions of this paper. 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 5. The following table shows that indeed our improved type-based refinement heuristic (tb′) from Section 5.3 significantly outperforms the

---

[14]Similarly, with such a pre-processing the existing "direct" tools would also be able to prove termination of the program in Example 1.1.

simple improved outermost (om′) and innermost (im) heuristics from Section 5.2. 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 (from Section 5.4) in order to compute a refined argument filter from the initial one. To evaluate the advantage of this improved algorithm over Algorithm 1 (from Section 5.1), we performed experiments with the two algorithms (again using the type-based refinement heuristic tb′ from Section 5.3). 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

As mentioned in Section 1.3, preliminary versions of parts of this paper appeared in [Schneider-Kamp et al. 2007]. However, the table below clearly shows that the results of Section 5 (which are new compared to [Schneider-Kamp et al. 2007]) 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 5 with the version of AProVE from the *Termination Competition* 2006 that only contains the results of [Schneider-Kamp et al. 2007]. 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 5.

	AProVE tb′	AProVE [Schneider-Kamp et al. 2007]
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/`.

7.2 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 pure logic programming (e.g., built-in predicates, negation as failure, etc.). But in the following, we discuss the limitations of the approach when applying it for pure logic programming. In principle, there could be three points of failure:

(1) The transformation of Theorem 3.7 could fail, i.e., there could be a logic program which is terminating for the set of queries, but not all corresponding terms

are terminating in the transformed TRS. We do not know of any such example. It is currently open whether this step is in fact complete.

(2) The approach via dependency pairs (Theorem 4.4) can fail to prove termination of the transformed TRS, although the TRS is terminating. In particular, this can happen because of the variable condition required for Theorem 4.4. This is demonstrated by the following logic program $\mathcal{P}$:

$$
\begin{aligned}
\mathsf{p}(X) &\;:\!-\; \mathsf{q}(\mathsf{f}(Y)), \mathsf{p}(Y) \\
\mathsf{p}(\mathsf{g}(X)) &\;:\!-\; \mathsf{p}(X) \\
\mathsf{q}(\mathsf{g}(Y)) &
\end{aligned}
$$

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.

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

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 it is possible to build an infinite chain with the last dependency pair where one instantiates X by the infinite term $\mathsf{g}(\mathsf{g}(\ldots))$. So, let $\pi(\mathsf{P}_{in}) = \pi(\mathsf{g}) = \{1\}$. Then 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.

(3) Finally, it can happen that the resulting DP problem of Theorem 4.4 is finite, 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, finiteness 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:

$$
\begin{aligned}
\mathsf{f}(X,Y) &\;\;:\!\!-\;\; \mathsf{g}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(X)))))),Y) \\
\mathsf{f}(\mathsf{s}(X),Y) &\;\;:\!\!-\;\; \mathsf{f}(X,Y) \\
\mathsf{g}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(X)))))),Y) &\;\;:\!\!-\;\; \mathsf{f}(X,Y)
\end{aligned}
$$

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 therefore 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 (2) to the completeness of (Theorem 4.4) 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 [Payet and Mesnard 2006]. 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 behaviour of 274 of the 296 examples.

8. CONCLUSION

In this paper, 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 [Arts and Giesl 2000; Giesl et al. 2005; Giesl et al. 2006] 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 [Giesl et al. 2005], now any such technique can also be used for logic programs.

In addition to the results presented in [Schneider-Kamp et al. 2007], 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.

REFERENCES

AGUZZI, G. AND MODIGLIANI, U. 1993. Proving Termination of Logic Programs by Transforming them into Equivalent Term Rewriting Systems. In *FSTTCS '93*. LNCS, vol. 761. 114–124.

APT, K. R. AND ETALLE, S. 1993. On the Unification Free Prolog Programs. In *MFCS '93*. LNCS, vol. 711. 1–19.

APT, K. R. 1997. *From Logic Programming to Prolog*. Prentice Hall, London.

ARTS, T. AND ZANTEMA, H. 1995. Termination of Logic Programs Using Semantic Unification. In *LOPSTR '95*. LNCS, vol. 1048. 219–233.

ARTS, T. AND GIESL, J. 2000. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science 236*, 133–178.

BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

BOSSI, A., COCCO, N., AND FABRIS, M. 1992. Typed Norms. In *ESOP '92*. LNCS, vol. 582. 73–92.

BRUYNOOGHE, M., DEMOEN, B., BOULANGER, D., DENECKER, M., AND MULKERS, A. 1996. A Freeness and Sharing Analysis of Logic Programs Based on a Pre-Interpretation. In *SAS '96*. LNCS, vol. 1145. 128–142.

BRUYNOOGHE, M., GALLAGHER, J. P., AND VAN HUMBEECK, W. 2005. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *SAS '05*. LNCS, vol. 3672. 35–51.

BRUYNOOGHE, M., CODISH, M., GALLAGHER, J. P., GENAIM, S., AND VANHOOF, W. 2007. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems 29*, 2, Article 10.

CHARATONIK, W. AND PODELSKI, A. 1998. Directional Type Inference for Logic Programs. In *SAS '98*. LNCS, vol. 1503. 278–294.

CHTOUROU, M. AND RUSINOWITCH, M. 1993. Méthode Transformationelle pour la Preuve de Terminaison des Programmes Logiques. Unpublished manuscript.

CODISH, M. AND TABOCH, C. 1999. A Semantic Basis for Termination Analysis of Logic Programs. *Journal of Logic Programming 41,* 1, 103–123.

CODISH, M., LAGOON, V., AND STUCKEY, P. 2005. Testing for Termination with Monotonicity Constraints. In *ICLP '05*. LNCS, vol. 3668. 326–340.

CODISH, M., LAGOON, V., SCHACHTE, P., AND STUCKEY, P. 2006. Size-Change Termination Analysis in k-Bits. In *ESOP '06*. LNCS, vol. 3924. 230–245.

CODISH, M. 2007. Collection of Benchmarks. Available at `http://lvs.cs.bgu.ac.il/~mcodish/suexec/terminweb/bin/terminweb.cgi?command=examples`.

COLMERAUER, A. 1982. Prolog and Infinite Trees. In *Logic Programming*, K. L. Clark and S. Tärnlund, Eds. Academic Press, Oxford.

CORTESI, A. AND FILÉ, G. 1999. Sharing is Optimal. *Journal of Logic Programming 38,* 3, 371–386.

DE SCHREYE, D. AND DECORTE, S. 1994. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming 19-20*, 199–260.

DE SCHREYE, D. AND SEREBRENIK, A. 2002. Acceptability with General Orderings. In *Computational Logic: Logic Programming and Beyond. Essays in Honour of Robert A. Kowalski, Part I*, A. C. Kakas and F. Sadri, Eds. LNCS, vol. 2407. 187–210.

DECORTE, S., DE SCHREYE, D., AND FABRIS, M. 1993. Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis. In *ILPS '93*. MIT Press, Boston, 420–436.

DERSHOWITZ, N. 1987. Termination of Rewriting. *Journal of Symbolic Computation 3,* 1–2, 69–116.

ENDRULLIS, J., WALDMANN, J., AND ZANTEMA, H. 2006. Matrix Interpretations for Proving Termination of Term Rewriting. In *IJCAR '06*. LNAI, vol. 4130. 574–588.

GALLAGHER, J. P. AND PUEBLA, G. 2002. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *PADL '02*. LNCS, vol. 2257. 243–261.

GANZINGER, H. AND WALDMANN, U. 1993. Termination Proofs of Well-moded Logic Programs via Conditional Rewrite Systems. In *CTRS '92*. LNCS, vol. 656. 430–437.

GESER, A., HOFBAUER, D., AND WALDMANN, J. 2004. Match-Bounded String Rewriting Systems. *Applicable Algebra in Engineering, Communication and Computing 15*, 3–4, 149–171.

GIESL, J., THIEMANN, R., AND SCHNEIDER-KAMP, P. 2005. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*. LNAI, vol. 3452. 301–331.

GIESL, J., SCHNEIDER-KAMP, P., AND THIEMANN, R. 2006. AProVE 1.2: Automatic Termination Proofs in the DP Framework. In *IJCAR '06*. LNAI, vol. 4130. 281–286.

GIESL, J., THIEMANN, R., SCHNEIDER-KAMP, P., AND FALKE, S. 2006. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning 37*, 3, 155–203.

HIROKAWA, N. AND MIDDELDORP, A. 2005. Automating the Dependency Pair Method. *Information and Computation 199*, 1,2, 172–199.

HUET, G. 1976. Résolution d'Equations dans les Langages d'Ordre 1, 2, . . . , ω. PhD thesis.

JANSSENS, G. AND BRUYNOOGHE, M. 1992. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming 13*, 2–3, 205–258.

KRISHNA RAO, M. R. K., KAPUR, D., AND SHYAMASUNDAR, R. K. 1998. Transformational Methodology for Proving Termination of Logic Programs. *Journal of Logic Programming 34*, 1, 1–41.

LAGOON, V. AND STUCKEY, P. J. 2002. Precise Pair-Sharing Analysis of Logic Programs. In *PPDP '02*. ACM Press, New York, 99–108.

LAGOON, V., MESNARD, F., AND STUCKEY, P. J. 2003. Termination Analysis with Types Is More Accurate. In *ICLP '03*. LNCS, vol. 2916. 254–268.

LESCANNE, P. 1983. Computer Experiments with the REVE Term Rewriting System Generator. In *POPL '83*. ACM Press, New York, 99–108.

LEUSCHEL, M. AND SØRENSEN, M. H. 1996. Redundant Argument Filtering of Logic Programs. In *LOPSTR '96*. LNCS, vol. 1207. 83–103.

LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. 1997. TermiLog: A System for Checking Termination of Queries to Logic Programs. In *CAV '97*. LNCS, vol. 1254. 444–447.

LU, L. 2000. A Precise Type Analysis of Logic Programs. In *PPDP '00*. ACM Press, New York, 214–225.

MARCHÉ, C. AND ZANTEMA, H. 2007. The Termination Competition. In *RTA '07*. LNCS, vol. 4533. 303–313. See also `http://www.lri.fr/~marche/termination-competition/`.

MARCHIORI, M. 1994. Logic Programs as Term Rewriting Systems. In *ALP '94*. LNCS, vol. 850. 223–241.

MARCHIORI, M. 1996. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*. LNCS, vol. 1101. 375–390.

MARTIN, J., KING, A., AND SOPER, P. 1996. Typed Norms for Typed Logic Programs. In *LOPSTR '96*. LNCS, vol. 1207. 224–238.

MESNARD, F. AND RUGGIERI, S. 2003. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic 4*, 2, 207–259.

MESNARD, F. AND BAGNARA, R. 2005. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming 5*, 1&2, 243–257.

MESNARD, F. AND SEREBRENIK, A. 2007. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming 8*, 1, 111–119.

NGUYEN, M. T. AND DE SCHREYE, D. 2005. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In *ICLP '05*. LNCS, vol. 3668. 311–325.

NGUYEN, M. T., BRUYNOOGHE, M., DE SCHREYE, D., AND LEUSCHEL, M. 2006. Program Specialisation as a Pre-Processing Step for Termination Analysis. In *WST '06*. 7–11. Available at `http://www.easychair.org/FLoC-06/WST-preproceedings.pdf`.

NGUYEN, M. T. AND DE SCHREYE, D. 2007. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In *LOPSTR '06*. LNCS, vol. 4407. 210–218.

NGUYEN, M. T., GIESL, J., SCHNEIDER-KAMP, P., AND DE SCHREYE, D. 2008. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*. LNCS, vol. 4915. To appear.

OHLEBUSCH, E., CLAVES, C., AND MARCHÉ, C. 2000. TALP: A Tool for the Termination Analysis of Logic Programs. In *RTA '00*. LNCS, vol. 1833. 270–273.

OHLEBUSCH, E. 2001. Termination of Logic Programs: Transformational Methods Revisited. *Applicable Algebra in Engineering, Communication and Computing 12,* 1–2, 73–116.

PAYET, E. AND MESNARD, F. 2006. Nontermination Inference of Logic Programs. *ACM Transactions on Programming Languages and Systems 28,* 2, 256–289.

VAN RAAMSDONK, F. 1997. Translating Logic Programs into Conditional Rewriting Systems. In *ICLP '97*. MIT Press, Boston, 168–182.

SCHNEIDER-KAMP, P., GIESL, J., SEREBRENIK, A., AND THIEMANN, R. 2007. Automated Termination Analysis for Logic Programs by Term Rewriting. In *LOPSTR '06*. LNCS, vol. 4407. 177–193.

SEREBRENIK, A. AND DE SCHREYE, D. 2003. Proving Termination with Adornments. In *LOPSTR '03*. LNCS, vol. 3018. 108–109.

SEREBRENIK, A. AND DE SCHREYE, D. 2004. Inference of Termination Conditions for Numerical Loops in Prolog. *Theory and Practice of Logic Programming 4,* 5&6, 719–751.

SEREBRENIK, A. AND DE SCHREYE, D. 2005a. On Termination of Meta-Programs. *Theory and Practice of Logic Programming 5,* 3, 355–390.

SEREBRENIK, A. AND DE SCHREYE, D. 2005b. Termination of Floating Point Computations. *Journal of Automated Reasoning 34,* 2, 141–177.

SMAUS, J.-G. 2004. Termination of Logic Programs Using Various Dynamic Selection Rules. In *ICLP '04*. LNCS, vol. 3132. 43–57.

TAMARY, L. AND CODISH, M. 2004. Abstract Partial Evaluation for Termination Analysis. In *WST '04*. 47–50. Available at `http://aib.informatik.rwth-aachen.de/2004/2004-07.pdf`.

TPDB 2007. The Termination Problem Data Base 4.0 (May 24, 2007). Available at `http://www.lri.fr/~marche/tpdb/`.

VAUCHERET, C. AND BUENO, F. 2002. More Precise Yet Efficient Type Inference for Logic Programs. In *SAS '02*. LNCS, vol. 2477. 102–116.

WALTHER, C. 1994. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence 71,* 1, 101–157.

ZANTEMA, H. 1995. Termination of Term Rewriting by Semantic Labelling. *Fundamenta Informaticae 24,* 1–2, 89–105.