Automated Termination Analysis for Logic Programs by Term Rewriting*

P. Schneider-Kamp[1], J. Giesl[1], A. Serebrenik[2], and R. Thiemann[1]

[1] LuFG Informatik 2, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany, `{psk,giesl,thiemann}@informatik.rwth-aachen.de`

[2] Dept. of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, `a.serebrenik@tue.nl`

Abstract. 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.

1 Introduction

Termination of logic programs is widely studied (see, e.g., [12] for an overview and [9, 13, 20, 26, 33] for more recent work on "direct" approaches). "Transformational" approaches have been developed in [1, 5, 8, 15, 19, 23, 24, 30] and a comparison of these approaches is given in [28]. 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 transformations can only be used for certain subclasses of logic programs. More precisely, all approaches except [23, 24] are restricted to *well-moded* programs. [23, 24] also consider the classes of *simply well-typed* and *safely typed* programs. We present a new transformation which, in contrast to

* Supported by the Deutsche Forschungsgsmeinschaft DFG under grant GI 274/5-1. In *Proc. LOPSTR "06*, LNCS, 2007.

all previous transformations, is applicable for *any* (definite[1]) logic program.

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 [29] is based on the transformations of [5, 8, 15] which are shown to be equally powerful in [28]. 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. Our experiments on large collections of examples in Sect. 5 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.

Our transformation is inspired by the transformation of [5, 8, 15, 28]. In this classical transformation, each argument position of each predicate is either labelled as *input* or *output*. As mentioned, the labelling must be such that the labelled program is *well moded* [3]. Well-modedness guarantees that each atom 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 $H \mathrel{:-} B_1, \ldots, B_k$ 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. We illustrate our concepts with a variant of a small example from [28]. Let p*'s first argument position be* input *and the second be* output.

$$\begin{array}{l} \mathsf{p}(X, X) \\ \mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \mathrel{:-} \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(Y)) \end{array}$$

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 [28], two new function symbols p_{in} and p_{out} are introduced for each predicate p. We write "$p(\boldsymbol{s}, \boldsymbol{t})$" to denote that $\boldsymbol{s}$ and $\boldsymbol{t}$ are the sequences of terms on p's in- and output positions.

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

[1] 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 [24], it is not clear how to transform non-definite logic programs into TRSs that are suitable for *automated* termination analysis, cf. (II).

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

Here, $\mathcal{V}(\boldsymbol{s})$ are the variables occurring in $\boldsymbol{s}$. Moreover, if $\mathcal{V}(\boldsymbol{s}) = \{x_1, ..., x_n\}$, then "$u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))$" abbreviates the term $u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), x_1, ..., x_n)$, 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. [28].

Example 2. For Ex. 1, the transformation results in the following TRS $\mathcal{R}$.

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

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

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

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 current tools for termination proofs of logic programs like cTI *[25],* Hasta-La-Vista *[32],* TALP *[29],* TermiLog *[22], and* TerminWeb *[10] fail on Ex. 1.*[2]

So this example already illustrates a drawback of the classical transformation of [28]: 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.[3]

Example 3. We modify Ex. 1 by replacing $\mathsf{g}(Y)$ with $\mathsf{g}(W)$ in the body:

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

Still, all queries $\mathsf{p}(t_1, t_2)$ terminate if t_1 is ground. But this program is not well moded, as the second clause violates Condition (a): $\mathcal{V}_{out}(\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y))) = \{Y\} \not\subseteq \mathcal{V}_{in}(\mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y))) \cup \mathcal{V}_{out}(\mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z))) \cup \mathcal{V}_{out}(\mathsf{p}(Z, \mathsf{g}(W))) = \{X, Z, W\}$. Trans-

[2] They can handle Ex. 1 if one performs a program specialization step before [31]. Our example collection at `http://aprove.informatik.rwth-aachen.de/eval/LP/` illustrates the advantages of different tools and also includes several examples where "direct" tools fail because the termination proof requires complex ranking functions.

[3] Ex. 3 is neither well moded nor simply well typed nor safely typed (using the types "*Any*" and "*Ground*") as required by the transformations [1, 5, 8, 15, 19, 23, 24, 30].

forming the program as before yields a TRS with the rule $\mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(W)), X, Z) \to \mathsf{p}_{out}(\mathsf{g}(Y))$*. So non-well-moded programs result in rules with variables like* Y *in the right- but not in the left-hand side. Such rules are usually forbidden in term rewriting and they do not terminate, since* Y *may be instantiated arbitrarily.*

A natural non-well-moded example is the append-program with the clauses $\mathsf{append}([\,], XS, XS)$ and $\mathsf{append}([X|XS], YS, [X|ZS]) \coloneq \mathsf{append}(XS, YS, ZS)$. If one only considers append's first argument as input, then this program is not well moded although all queries $\mathsf{append}(t_1, t_2, t_3)$ are terminating if t_1 is ground.

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 [13].[4] 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 rewriting) to termination analysis of logic programs is investigated in [27].

Instead of integrating each termination technique from term rewriting separately, we want to make all these techniques available at once. Therefore, unlike [13, 27], we choose a transformational approach. Our goal is a method which

(A) handles programs like Ex. 1 where classical transformations like [28] fail,

(B) handles non-well-moded programs like Ex. 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 Ex. 1 and 3 where current tools based on direct approaches fail. For larger and more realistic examples we refer to Sect. 5.

After presenting required preliminaries in Sect. 2, in Sect. 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 $\mathsf{p}(\mathsf{s}(X)) \coloneq \mathsf{p}(X)$ 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.

Sect. 4 shows that the existing termination techniques for TRSs can easily be adapted in order to prove termination of infinitary constructor rewriting. We conclude with an experimental evaluation of our results in Sect. 5 which shows

[4] But in contrast to [13], we also apply more recent powerful termination techniques from rewriting (e.g., *dependency pairs* [4, 16]) for termination of logic programs.

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.

2 Preliminaries on Logic Programming and Rewriting

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 4 (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 node labelled with f/n has n children and leaves are labelled with variables 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 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 $\boldsymbol{t}$ is the sequence $t_1, \ldots, t_n$, then $\boldsymbol{t} \in \boldsymbol{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V})$ means that $t_i \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$ for all i. $\boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V})$ is defined analogously. A* position *p in a (possibly infinite) term t addresses a subtree $t|_p$ of t where the path from* $\mathrm{root}(t)$ *to* $\mathrm{root}(t|_p)$ *is finite. The term $t[s]_p$ results from replacing the subterm $t|_p$ at position p in t by the term s.*

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 $\mathcal{T}(\Sigma, \varnothing)$.*

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.[5]

[5] One can even define the composition of *infinitely* many substitutions $\sigma_0, \sigma_1, \ldots$ such 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 $t\sigma_0\sigma_1...$ for any term t. Obviously, p is a position of $t\sigma_0\sigma_1...$ iff p is a position of $t\sigma_0...\sigma_n$ for some $n \geq 0$. We define that the symbol of $t\sigma_0\sigma_1...$ at such a position p is $f \in \Sigma$ iff f is at position p in $t\sigma_0...\sigma_m$ for some $m \geq 0$. Otherwise, $(t\sigma_0...\sigma_n)|_p = X_0 \in \mathcal{V}$. Let $n = i_0 < i_1 < ...$ be the maximal (finite or infinite) sequence with $\sigma_{i_j+1}(X_j) = ... = \sigma_{i_{j+1}-1}(X_j) = X_j$ and $\sigma_{i_{j+1}}(X_j)$

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 [2], for example.

Definition 5 (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*[6] *of A_1 and H, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$.*

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 $Q_0 \vdash_{\mathcal{P}} Q_1$. 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 occur check [11] 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 programs, whereas most other approaches only consider logic programming with occur check.

Example 6. 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 Def. 5, it would be terminating if one uses logic programming with occur check. Indeed, tools like* cTI *[25] and* TerminWeb *[10] report that such queries are "terminating". So in contrast to our technique, such tools are in general not sound for logic programming without occur check, although this form of logic programming is typically used in practice.*

Now we define TRSs and introduce the notion of *infinitary constructor rewriting*. For further details on term rewriting we refer to [6].

Definition 7 (Infinitary Constructor Rewriting). *A* TRS *$\mathcal{R}$ is a finite set of rules $l \to r$ with $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$ and $l \notin \mathcal{V}$. We divide the signature in* defined *symbols $\Sigma_D = \{f \mid l \to r \in \mathcal{R}, \mathrm{root}(l) = f\}$ 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 $l \to r$, a position p and a substitution $\sigma : \mathcal{V} \to \mathcal{T}^\infty(\Sigma_C, \mathcal{V})$*

$= X_{j+1}$ 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 $(t\sigma_0\sigma_1\ldots)|_p = X_m$. Otherwise, the substitutions perform infinitely many variable renamings. In this case, we use one special variable Z_∞ and define $(t\sigma_0\sigma_1\ldots)|_p = Z_\infty$. So if $\sigma_0(X) = Y$, $\sigma_1(Y) = X$, $\sigma_2(X) = Y$, $\sigma_3(Y) = X$, etc., we define $X\sigma_0\sigma_1\ldots = Y\sigma_0\sigma_1\ldots = Z_\infty$.

[6] 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 [18].

with $s|_p = l\sigma$ and $t = s[r\sigma]_p$. 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 and (ii) we use substitutions with possibly non-rational infinite terms. In Ex. 9 and 10 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 TRSs

Now we modify the transformation of logic programs into TRSs from Sect. 1 to make it applicable for *arbitrary* (possibly non-well-moded) programs as well. 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 8 (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 $p(\boldsymbol{s})$ in $\mathcal{P}$, the TRS $\mathcal{R}_{\mathcal{P}}$ contains the rule $p_{in}(\boldsymbol{s}) \to p_{out}(\boldsymbol{s})$.*
- *For each clause c of the form $p(\boldsymbol{s}) :- p_1(\boldsymbol{s}_1), \ldots, p_k(\boldsymbol{s}_k)$ in $\mathcal{P}$, $\mathcal{R}_{\mathcal{P}}$ contains:*

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

The following two examples motivate the need for infinitary constructor rewriting in Def. 8, i.e., they motivate Modifications (i) and (ii).

Example 9. For the logic program of Ex. 1, we obtain the following TRS.

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

This example shows why rules of TRSs may only be instantiated with constructor terms (Modification (i)). The reason is that local variables like Z (i.e., variables occurring in the body but not in the head of a clause) give rise to rules $l \to r$ where $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ (cf. the second rule). 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 Ex. 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 the second rule. 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 Ex. 3, one obtains a similar TRS where $\mathsf{g}(Y)$ *is replaced by* $\mathsf{g}(W)$ *in the right-hand side of the third and the left-hand side of the last rule. Thus, we can now handle programs where the classical transformation of [5, 8, 15, 28] failed, cf. Goals (A) and (B).*

Derivations in logic programming use *unification*, while rewriting is defined by *matching*. Ex. 10 shows that to simulate unification by matching, we have to consider substitutions with infinite and even non-rational terms (Modification (ii)).

Example 10. *Let* $\mathcal{P}$ *be* $\mathsf{ordered}(\mathsf{cons}(X, \mathsf{cons}(\mathsf{s}(X), XS))) \mathrel{:-} \mathsf{ordered}(\mathsf{cons}(\mathsf{s}(X), XS))$. *If one only considers rewriting with finite or rational terms, then the transformed TRS* $\mathcal{R}_\mathcal{P}$ *is terminating. However, the query* $\mathsf{ordered}(YS)$ *is not terminating for* $\mathcal{P}$. *Thus, to obtain a sound approach,* $\mathcal{R}_\mathcal{P}$ *must also be non-terminating. Indeed,* $\mathsf{ordered}_{in}(\mathsf{cons}(X,\ \mathsf{cons}(\mathsf{s}(X),\ \mathsf{cons}(\mathsf{s}^2(X),\ \ldots))))$ *is non-terminating with* $\mathcal{R}_\mathcal{P}$*'s rule* $\mathsf{ordered}_{in}(\mathsf{cons}(X, \mathsf{cons}(\mathsf{s}(X), XS))) \to \mathsf{u}(\mathsf{ordered}_{in}(\mathsf{cons}(\mathsf{s}(X), XS)), X, XS)$. *This non-rational term corresponds to the infinite derivation with* $\mathsf{ordered}(YS)$.

Lemma 11 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 11 (Connecting $\mathcal{P}$ and $\mathcal{R}_\mathcal{P}$). *Let* $\mathcal{P}$ *be a program, let* $\boldsymbol{t}$ *be terms from* $\mathcal{T}^{rat}(\Sigma, \mathcal{V})$, *let* $p(\boldsymbol{t}) \vdash^n_{\mathcal{P},\sigma} Q$. *If* $Q = \Box$, *then* $p_{in}(\boldsymbol{t})\sigma \to^{\geq n}_{\mathcal{R}_\mathcal{P}} p_{out}(\boldsymbol{t})\sigma$. *Otherwise, if* Q *is "*$q(\boldsymbol{v}), \ldots$*", then* $p_{in}(\boldsymbol{t})\sigma \to^{\geq n}_{\mathcal{R}_\mathcal{P}} r$ *for a term* r *containing the subterm* $q_{in}(\boldsymbol{v})$.

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

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

Finally, let $Q_1 \neq \Box$. Thus, c_1 is $p(\boldsymbol{s}) \mathrel{:-} p_1(\boldsymbol{s_1}), \ldots, p_k(\boldsymbol{s_k})$, Q_1 is $p_1(\boldsymbol{s_1})\theta_1, \ldots, p_k(\boldsymbol{s_k})\theta_1$, and θ_1 is the mgu of $p(\boldsymbol{t})$ and $p(\boldsymbol{s})$. There is an i with $1 \leq i \leq k$ such that for all j with $1 \leq j \leq i-1$ we have $p_j(\boldsymbol{s}_j)\sigma_0 \ldots \sigma_{j-1} \vdash^{n_j}_{\mathcal{P},\sigma_j} \Box$. Moreover, if $Q = \Box$ then $i = k$ and $p_i(\boldsymbol{s}_i)\sigma_0 \ldots \sigma_{i-1} \vdash^{n_i}_{\mathcal{P},\sigma_i} \Box$ and if Q is "$q(\boldsymbol{v}), \ldots$", then $p_i(\boldsymbol{s}_i)\sigma_0 \ldots \sigma_{i-1} \vdash^{n_i}_{\mathcal{P},\sigma_i} q(\boldsymbol{v}), \ldots$ Here, $n = n_1 + \ldots + n_i + 1$, $\sigma_0 = \theta_1$, $\sigma_1 = \theta_2 \ldots \theta_{n_1+1}$, …, 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 $p_{j_{in}}(\boldsymbol{s}_j)\sigma_0 \ldots \sigma_j \to^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\boldsymbol{s}_j)\sigma_0 \ldots \sigma_j$ and thus also $p_{j_{in}}(\boldsymbol{s}_j)\sigma \to^{\geq n_j}_{\mathcal{R}_\mathcal{P}} p_{j_{out}}(\boldsymbol{s}_j)\sigma$. Moreover, if $Q = \Box$ then we also have $p_{i_{in}}(\boldsymbol{s}_i)\sigma \to^{\geq n_i}_{\mathcal{R}_\mathcal{P}} p_{i_{out}}(\boldsymbol{s}_i)\sigma$ where $i = k$. Otherwise, if Q is "$q(\boldsymbol{v}), \ldots$", then the induction hypothesis implies $p_{i_{in}}(\boldsymbol{s}_i)\sigma \to_{\mathcal{R}_\mathcal{P}}^{\geq n_i} r'$, where r' contains $q_{in}(\boldsymbol{v})$. Thus

$$\begin{array}{rcll} p_{in}(\boldsymbol{t})\sigma & = & p_{in}(\boldsymbol{s})\sigma \to_{\mathcal{R}_\mathcal{P}} & u_{c_1,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))\sigma \\ & & \to_{\mathcal{R}_\mathcal{P}}^{\geq n_1} & u_{c_1,1}(p_{1_{out}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))\sigma \\ & & \to_{\mathcal{R}_\mathcal{P}} & u_{c_1,2}(p_{2_{in}}(\boldsymbol{s}_2), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1))\sigma \\ & & \to_{\mathcal{R}_\mathcal{P}}^{\geq n_2} & u_{c_1,2}(p_{2_{out}}(\boldsymbol{s}_2), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1))\sigma \\ & & \to_{\mathcal{R}_\mathcal{P}}^{\geq n_3+\ldots+n_{i-1}} & u_{c_1,i}(p_{i_{in}}(\boldsymbol{s}_i), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1) \cup \ldots \cup \mathcal{V}(\boldsymbol{s}_{i-1}))\sigma \end{array}$$

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

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

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

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

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

Lemma 12 (Form of 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 "$q(\boldsymbol{v}), \ldots$" with $i > 0$ such that the query $q(\boldsymbol{v})$ is also non-terminating.*

Proof. Assume that for all $i > 0$, the first atom in Q_i is successfully proved in n_i steps during the derivation $Q_0 \vdash_\mathcal{P} Q_1 \vdash_P \ldots$ (Otherwise, the derivation would not be infinite.) Let m be the number of atoms in Q_1. But then $Q_{1+n_1+\ldots+n_m}$ is the empty query $\Box$ which contradicts the infiniteness of the derivation. $\Box$

To characterize the classes of queries whose termination we want to analyze, we use *argument filterings*. Related definitions can be found in, e.g., [4, 21].

Definition 13 (Argument Filtering). *An* argument filtering π over a signature (Σ, Δ) *is a function* $\pi : \Sigma \cup \Delta \to 2^\mathbb{N}$ *where* $\pi(f/n) \subseteq \{1, \ldots, n\}$ *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* $i_1 < \ldots < i_k$. *For any TRS* $\mathcal{R}$, *we define* $\pi(\mathcal{R}) = \{\pi(l) \to \pi(r) \mid l \to r \in \mathcal{R}\}$.

Argument filterings 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 Ex. 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 filtering $\pi(h) = \{1\}$ 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 filterings also operate on *function* instead of just *predicate* symbols. Therefore, they can describe more sophisticated classes of queries than the classical approach of [28] 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 filtering $\pi(\mathsf{append}) =$

$\{1\}$ and $\pi(.) = \{2\}$, where "." is the list constructor (i.e., $.(X, L) = [X|L]$). Of course, our method can easily prove that all these queries are terminating.

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 $p_{in}(\boldsymbol{t})$ for the TRS $\mathcal{R}_{\mathcal{P}}$ where $\pi(p_{in}(\boldsymbol{t}))$ is a finite ground term and where $\boldsymbol{t}$ 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 14 (Soundness of the Transformation). *Let $\mathcal{P}$ be a logic program and let π be an argument filtering over (Σ, Δ). We extend π such that $\pi(p_{in}) = \pi(p)$ for all $p \in \Delta$. Let $S = \{p_{in}(\boldsymbol{t}) \mid p \in \Delta,\ \boldsymbol{t} \in \boldsymbol{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V}),\ \pi(p_{in}(\boldsymbol{t})) \in \mathcal{T}(\Sigma)\,\}$. 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 $\pi(Q) \in \mathcal{A}(\Sigma, \Delta)$ are terminating for $\mathcal{P}$.*

Proof. Assume that there is a non-terminating query $p(\boldsymbol{t})$ as above with $p(\boldsymbol{t}) \vdash_{\mathcal{P}} Q_1 \vdash_{\mathcal{P}} Q_2 \vdash_{\mathcal{P}} \ldots$ By Lemma 12 there is an $i_1 > 0$ with $Q_{i_1} = q_1(\boldsymbol{v}_1), \ldots$ and an infinite derivation $q_1(\boldsymbol{v}_1) \vdash_{\mathcal{P}} Q'_1 \vdash_{\mathcal{P}} Q'_2 \vdash_{P} \ldots$ From $p(\boldsymbol{t}) \vdash^{i_1}_{\sigma_0, \mathcal{P}} q_1(\boldsymbol{v}_1), \ldots$ and Lemma 11 we get $p_{in}(\boldsymbol{t})\sigma_0 \rightarrow^{\geq i_1}_{\mathcal{R}_{\mathcal{P}}} r_1$, where r_1 contains the subterm $q_{1_{in}}(\boldsymbol{v}_1)$.

By Lemma 12 again, there is an $i_2 > 0$ with $Q'_{i_2} = q_2(\boldsymbol{v}_2), \ldots$ and an infinite derivation $q_2(\boldsymbol{v}_2) \vdash_{\mathcal{P}} Q''_1 \vdash_{\mathcal{P}} \ldots$ From $q_1(\boldsymbol{v}_1) \vdash^{i_2}_{\sigma_1, \mathcal{P}} q_2(\boldsymbol{v}_2), \ldots$ and Lemma 11 we get $p_{in}(\boldsymbol{t})\sigma_0\sigma_1 \rightarrow^{\geq i_1}_{\mathcal{R}_{\mathcal{P}}} r_1\sigma_1 \rightarrow^{\geq i_2}_{\mathcal{R}_P} r_2$, where r_2 contains the subterm $q_{2_{in}}(\boldsymbol{v}_2)$.

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. Since $r_j\mu_j$ is an instance of $r_j\sigma_j \ldots \sigma_n$ for all $n \geq j$ (cf. Footnote 5), we obtain that $p_{in}(\boldsymbol{t})\mu_0$ is non-terminating for $\mathcal{R}_{\mathcal{P}}$:

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

As $\pi(p(\boldsymbol{t})) \in \mathcal{A}(\Sigma, \Delta)$ and thus $\pi(p_{in}(\boldsymbol{t})\mu_0) \in \mathcal{T}(\Sigma)$, this is a contradiction. □

4 Termination of Infinitary Constructor Rewriting

One of the most powerful methods for automated termination analysis of rewriting is the *dependency pair* (DP) method [4] 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 Def. 8.

Moreover, dependency pairs are a general framework that permits the integration of *any* termination technique for TRSs [16, 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 without adapting it as well.

For a TRS $\mathcal{R}$ over Σ, for each $f/n \in \Sigma_D$ let $f^\sharp/n$ be a fresh *tuple symbol*. We often write F instead of $f^\sharp$. For $t = g(\boldsymbol{t})$ with $g \in \Sigma_D$, let $t^\sharp$ denote $g^\sharp(\boldsymbol{t})$.

Definition 15 (Dependency Pair [4]). *The set of* dependency pairs *for a TRS* $\mathcal{R}$ *is* $DP(\mathcal{R}) = \{l^\sharp \to t^\sharp \mid l \to r \in \mathcal{R},\ t$ *is a subterm of* r, $\mathrm{root}(t) \in \Sigma_D\}$.

Example 16. In the TRS $\mathcal{R}$ *of Ex. 9, we have* $\Sigma_D = \{\mathsf{p}_{in}, \mathsf{u}_1, \mathsf{u}_2\}$ *and* $DP(\mathcal{R})$ *is*

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

While Def. 15 is from [4], 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. Def. 17 extends the notion of chains to infinitary constructor rewriting. To this end, we use an argument filtering 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) rewriting and if π deletes all arguments (i.e., if $\pi(f) = \varnothing$ for all f), then this corresponds to full infinitary rewriting. In Def. 17, 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 17 (Chain). *Let* $\mathcal{D}, \mathcal{R}$ *be TRSs and* π *be a filtering over* Σ. *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*

- *there are substitutions* $\sigma_i : \mathcal{V} \to \mathcal{T}^\infty(\Sigma_C, \mathcal{V})$ *such that* $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$. *Here,* Σ_C *are the constructors of the TRS* $\mathcal{R}$.
- $\pi(s_i\sigma_i), \pi(t_i\sigma_i) \in \mathcal{T}(\Sigma)$ *and for all terms* q *in the rewrite sequence from* $t_i\sigma_i$ *to* $s_{i+1}\sigma_{i+1}$ *we have* $\pi(q) \in \mathcal{T}(\Sigma)$ *as well. So all terms in the sequence have finite ground terms on those positions which are not filtered away by* π.

In Ex. 16, "(2), (3)" is a chain for any argument filtering π: if one instantiates X and Z with the same finite ground term, then (2)'s instantiated right-hand side rewrites to an instance of (3)'s left-hand side. Note that if one uses an argument filtering π which permits an instantiation of X and Z with the infinite term $\mathsf{f}(\mathsf{f}(\ldots))$, then there is also an infinite chain "(2), (3), (2), (3), ..."

For termination of a program $\mathcal{P}$, by Thm. 14 we have to show that if $\pi(p_{in}(\boldsymbol{t}))$ is a finite ground term and $\boldsymbol{t}$ only contains function symbols from the logic program (i.e., $\boldsymbol{t}$ contains no defined symbols of the TRS $\mathcal{R}_\mathcal{P}$), then $p_{in}(\boldsymbol{t})$ is terminating for $\mathcal{R}_\mathcal{P}$. Thm. 18 states that one can prove absence of infinite $(DP(\mathcal{R}_\mathcal{P}), \mathcal{R}_\mathcal{P}, \pi')$-chains instead. Here, π' is a filtering 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., $\mathcal{V}(\pi'(r)) \subseteq \mathcal{V}(\pi'(l))$ for all $l \to r \in DP(\mathcal{R}_\mathcal{P}) \cup \mathcal{R}_\mathcal{P}$. Then the filtering π' detects all potentially infinite subterms in rewrite sequences (i.e., all subterms which correspond to "non-unification-free parts" of $\mathcal{P}$).

Theorem 18 (Proving Infinitary Termination). *Let $\mathcal{R}$ be a TRS over Σ and let π be an argument filtering over Σ. Let π' be an argument filtering with $\pi'(f) \subseteq \pi(f)$ for all $f \in \Sigma$. Moreover, π' should also be defined on tuple symbols such that $\pi'(F) \subseteq \pi'(f)$ for all $f \in \Sigma_D$. Assume that $\pi'(DP(\mathcal{R}))$ and $\pi'(\mathcal{R})$ satisfy the variable condition.[7] If there is no infinite $(DP(\mathcal{R}), \mathcal{R}, \pi')$-chain, then all terms $f(\boldsymbol{t})$ with $\boldsymbol{t} \in \boldsymbol{\mathcal{T}}^\infty(\Sigma_C, \mathcal{V})$ and $\pi(f(\boldsymbol{t})) \in \mathcal{T}(\Sigma)$ are terminating for $\mathcal{R}$.*

Proof. Assume there is a non-terminating term $f(\boldsymbol{t})$ as above. Since $\boldsymbol{t}$ does not contain defined symbols, the first rewrite step in the infinite sequence is on the root position with a rule $l = f(\boldsymbol{l}) \to r$ where $l\sigma_1 = f(\boldsymbol{t})$. Since σ_1 does not introduce defined symbols, all defined symbols of $r\sigma_1$ occur on positions of r. So there is a subterm r' of r with defined root such that $r'\sigma_1$ is also non-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 $l^\sharp \to r'^\sharp$ is the first dependency pair of the infinite chain that we construct. Note that $\pi(l\sigma_1)$ and thus, $\pi'(l^\sharp\sigma_1) = \pi'(F(\boldsymbol{t}))$ is a finite ground term by assumption. Moreover, as $l^\sharp \to r'^\sharp \in DP(\mathcal{R})$ 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. As $\pi'(\mathcal{R})$ satisfies the variable condition, the terms remain finite and ground when applying the filtering π'. Finally, a root rewrite step is performed again. Repeating this construction infinitely many times results in an infinite chain. □

Example 19. We want to prove termination of Ex. 1 for all queries Q where $\pi(Q)$ is finite and ground for the filtering $\pi(h) = \{1\}$ for all $h \in \{\mathsf{p}, \mathsf{f}, \mathsf{g}\}$. By Thm. 14 and 18, it suffices to show absence of infinite $(DP(\mathcal{R}), \mathcal{R}, \pi')$-chains. Here, $\mathcal{R}$ is the TRS from Ex. 9 and $DP(\mathcal{R})$ are Rules (1) – (4) from Ex. 16. The filtering π' has to satisfy $\pi'(\mathsf{p}_{in}) \subseteq \pi(\mathsf{p}_{in}) = \pi(\mathsf{p}) = \{1\}$, $\pi'(h) \subseteq \pi(h) = \{1\}$ for $h \in \{\mathsf{f}, \mathsf{g}\}$, and $\pi'(H) \subseteq \pi'(h)$ for all defined symbols h. 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. This can indeed be automated, since 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, in our implementation we generate filterings where the sets $\pi'(h)$ are as large as possible, since such filterings provide more information about the finiteness of arguments. So in our example, we use $\pi'(\mathsf{p}_{in}) = \pi'(\mathsf{P}_{in}) = \pi'(\mathsf{f}) = \pi'(\mathsf{g}) = \{1\}$, $\pi'(\mathsf{p}_{out}) = \pi'(\mathsf{u}_1) = \pi'(\mathsf{U}_1) = \{1, 2\}$, and $\pi'(\mathsf{u}_2) = \pi'(\mathsf{U}_2) = \{1, 2, 4\}$. For the non-well-moded Ex. 3 we choose $\pi'(\mathsf{g}) = \varnothing$ instead to satisfy the variable condition.

Finally, 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 [16] to infinitary rewriting.

[7] To see why the variable condition is needed in Thm. 18, let $\mathcal{R} = \{\mathsf{g}(X) \to \mathsf{f}(X), \mathsf{f}(\mathsf{s}(X)) \to \mathsf{f}(X)\}$ and $\pi = \pi'$ where $\pi'(\mathsf{g}) = \varnothing$, $\pi'(\mathsf{f}) = \pi'(\mathsf{F}) = \pi'(\mathsf{s}) = \{1\}$. $\mathcal{R}$'s first rule violates the variable condition: $\mathcal{V}(\pi'(\mathsf{f}(X))) = \{X\} \not\subseteq \mathcal{V}(\pi'(\mathsf{g}(X))) = \varnothing$. There is no infinite chain, since π' does not allow us to instantiate the variable X in the dependency pair $\mathsf{F}(\mathsf{s}(X)) \to \mathsf{F}(X)$ by an infinite term. Nevertheless, there is a non-terminating term $\mathsf{g}(\mathsf{s}(\mathsf{s}(\ldots)))$ which is filtered to a finite ground term $\pi'(\mathsf{g}(\mathsf{s}(\mathsf{s}(\ldots)))) = \mathsf{g}$.

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 filtering. 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 $\mathit{Proc}(d)$ 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 Thm. 22, 24, and 26 we will recapitulate three of the most important existing DP processors [16] and describe how they must be modified for infinitary constructor rewriting.[8] To this end, they now also have to take the argument filtering π into account. The first processor uses an *estimated dependency graph* to estimate which dependency pairs can follow each other in chains.

Definition 20 (Estimated Dependency Graph). *Let $(\mathcal{D}, \mathcal{R}, \pi)$ be a DP problem. The nodes of the* estimated $(\mathcal{D}, \mathcal{R}, \pi)$-dependency graph *are the pairs of $\mathcal{D}$ and there is an arc from $s \to t$ to $u \to v$ 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 21. For the DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ from Ex. 19 we obtain:

$$\begin{tikzcd} (1) & (3) \arrow[l] \arrow[r, bend left] & (2) \arrow[l, bend left] \arrow[r] & (4) \end{tikzcd}$$

For example, there is an arc $(2) \to (3)$, 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 $(1) \to (1)$ or $(1) \to (2)$, 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 filtering $\pi'(\mathsf{P}_{in}) = \{1\}$).

Note that filterings are used to *detect* potentially infinite arguments, but they are not *removed*, since they can still be useful in the termination proof. In Ex. 21, these arguments are needed to determine that there are no arcs from (1).

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.

Theorem 22 (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 nodes of the SCCs in the estimated dependency graph. Then Proc is sound.*

[8] Their soundness proofs can be found in `http://aprove.informatik.rwth-aachen.de/eval/LP/SGST06.ps`.

Example 23. In Ex. 21, the only SCC consists of (2) and (3). Thus, the dependency graph processor transforms the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ into $(\{(2),(3)\}, \mathcal{R}, \pi')$, i.e., it deletes the dependency pairs (1) and (4).

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 (LPO, polynomial interpretations, etc. [14]).

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 $\succ$).[9] Requiring $\pi(l) \succsim \pi(r)$ for all $l \to r \in \mathcal{R}$ ensures that in chains $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$ as in Def. 17, we have $\pi(t_i\sigma_i) \succsim \pi(s_{i+1}\sigma_{i+1})$. Hence, if a reduction pair satisfies the above conditions, then the strictly decreasing dependency pairs (i.e., those $s \to t \in \mathcal{D}$ where $\pi(s) \succ \pi(t)$) cannot occur infinitely often in chains. So the following processor deletes these pairs from $\mathcal{D}$. For any TRS $\mathcal{D}$ and any relation $\succ$, let $\mathcal{D}_{\succ_\pi} = \{s \to t \in \mathcal{D} \mid \pi(s) \succ \pi(t)\}$.

Theorem 24 (Reduction Pair Processor). *Let $(\succsim, \succ)$ be a reduction pair. Then the following DP processor Proc is sound. For $(\mathcal{D}, \mathcal{R}, \pi)$, Proc returns*

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

Example 25. For the DP problem $(\{(2),(3)\}, \mathcal{R}, \pi')$ in Ex. 23, one can easily find a reduction pair[10] where the dependency pair (3) is strictly decreasing and where (2) and all rules are weakly decreasing after applying the filtering π':

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

Thus, the reduction pair processor can remove (3) from the DP problem which results in $(\{(2)\}, \mathcal{R}, \pi')$. 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 (2) and no arcs. This proves that the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \pi')$ from Ex. 19 is finite and thus, the logic program from Ex. 1 terminates for all queries Q where $\pi(Q)$ is finite and ground. Note that termination of the non-well-moded program from Ex. 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.

[9] 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.

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

As in Thm. 22 and 24, many other existing DP processors [16] 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 $(\pi(D), \pi(R), id)$ for ordinary rewriting. Afterwards, *any* existing DP processor for *ordinary* rewriting becomes applicable.[11] Since any termination technique for TRSs can immediately be formulated as a DP processor [16, Thm. 36], now any termination technique for ordinary rewriting can be directly used for infinitary constructor rewriting as well.

Theorem 26 (Argument Filtering 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.*

5 Experiments and 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 $\mathcal{R}_\mathcal{P}$ when using infinitary constructor rewriting. This approach is even sound for logic programming without occur check. We showed how to adapt the DP framework of [4, 16] 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 [16], now any such technique can also be used for logic programs.

We integrated our approach in the termination tool AProVE [17] which implements the DP framework. To evaluate our results, we tested AProVE against three other representative termination tools for logic programming: TALP [29] is the only other available tool based on transformational methods (it uses the classical transformation [28] described in Sect. 1), whereas cTI [25] and TerminWeb [10] are based on direct approaches. We ran the tools on a set of 296 examples in fully automatic mode.[12] This set includes all logic programming examples from the *Termination Problem Data Base* which is used in the annual *International Termination Competition*[13] and which contains several collections provided by the developers of other tools. Moreover, we also included all examples from the experimental evaluation of [7]. 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. Here, TALP succeeds on 163 examples, cTI proves termination of 167 examples, TerminWeb succeeds on 178 examples, and AProVE verifies termination of 208 examples (including all where TALP is successful).

[11] If $(\mathcal{D}, \mathcal{R}, \pi)$ results from the transformation of a logic program, then for $(\pi(D), \pi(R), id)$ it is even sound to apply the existing DP processors for *innermost* rewriting [16]. These processors are usually more powerful than those for ordinary rewriting.

[12] 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.

[13] For details, see `http://www.lri.fr/~marche/termination-competition/`.

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 cTI and TerminWeb demonstrates that our new transformational approach is comparable in power to direct approaches. But there is a substantial set of programs where AProVE succeeds and direct tools fail (cf. Goal (C)) and there is also a substantial set of examples where direct tools succeed and AProVE fails. More precisely, AProVE succeeds on 57 examples where cTI fails and on 46 examples where TerminWeb fails. On the other hand, there are 16 examples where cTI succeeds whereas AProVE cannot prove termination and there are also 16 examples where TerminWeb succeeds and AProVE fails.

Thus, transformational and direct approaches both have their advantages and the most powerful solution would be to combine direct tools like cTI or TerminWeb with a transformational prover like AProVE which is based on the contributions of this paper. This also indicates that it is indeed beneficial to use termination techniques from TRSs for logic programs as well. To run AProVE, for details on our experiments, to access our collection of examples, and for a discussion on the limitations[14] of our approach and its implementation, we refer to `http://aprove.informatik.rwth-aachen.de/eval/LP/`.

Acknowledgements. We thank M. Codish, D. De Schreye, and F. Mesnard for helpful comments and R. Bagnara and S. Genaim for help with the experiments.

References

1. G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. In *Proc. 13th FST & TCS*, LNCS 761, pages 114–124, 1993.
2. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
3. K. R. Apt and S. Etalle. On the unification free Prolog programs. In *Proc. 18th MFCS*, LNCS 711, pages 1–19, 1993.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
5. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Proc. 5th LOPSTR*, LNCS 1048, pages 219–233, 1995.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge, 1998.
7. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 2006. To appear.

[14] Our approach could fail for 3 reasons: (1) The transformation of Thm. 14 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 such examples and it could be that this step is indeed complete. (2) The approach via dependency pairs (Thm. 18) can fail to prove termination of the transformed TRS. (3) Our implementation can fail to prove finiteness of the resulting DP problem from Thm. 18. On the website, we give examples for Failures (2) and (3).

8. M. Chtourou and M. Rusinowitch. Méthode transformationelle pour la preuve de terminaison des programmes logiques. Unpublished manuscript, 1993.
9. M. Codish, V. Lagoon, and P. Stuckey. Testing for termination with monotonicity constraints. In *Proc. 21st ICLP*, LNCS 3668, pages 326–340, 2005.
10. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
11. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. Tärnlund, editors, *Logic Programming*. Academic Press, 1982.
12. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19&20:199–260, 1994.
13. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In *Computational Logic. Logic Prog. and Beyond.*, LNCS 2407, pages 187–210, 2002.
14. N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3:69–116, 1987.
15. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. *Proc. 3rd CTRS*, LNCS 656, pages 216–222, 1993.
16. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th LPAR*, LNAI 3452, pages 301–331, 2005.
17. J. Giesl, P. Schneider-Kamp, R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. 3rd IJCAR*, LNAI 4130, pp. 281–286, 2006.
18. G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. PhD, 1976.
19. M. Krishna Rao, D. Kapur, and R. Shyamasundar. Transformational methodology for proving termination of logic programs. *J. Log. Prog.*, 34(1):1–42, 1998.
20. V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination analysis with types is more accurate. In *Proc. 19th ICLP*, LNCS 2916, pages 254–268, 2003.
21. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In *Proc. 6th LOPSTR*, LNCS 1207, pages 83–103, 1996.
22. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. *Proc. 9th CAV*, LNCS 1254, p. 444-447, 1997.
23. M. Marchiori. Logic programs as term rewriting systems. In *Proc. 4th ALP*, LNCS 850, pages 223–241, 1994.
24. M. Marchiori. Proving existential termination of normal logic programs. In *Proc. 5th AMAST*, LNCS 1101, pages 375–390, 1996.
25. F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1&2):243–257, 2005.
26. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transaction on Computational Logic*, 4(2):207–259, 2003.
27. M. T. Nguyen and D. De Schreye. Polynomial interpretations as a basis for termination analysis of logic programs. *Proc. 21. ICLP*, LNCS 3668, p.311-325, 2005.
28. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Algebra in Engineering, Communication and Computing*, 12:73–116, 2001.
29. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In *Proc. 11th RTA*, LNCS 1833, pages 270–273, 2000.
30. F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In *Proc. 14th ICLP*, pages 168–182. MIT Press, 1997.
31. A. Serebrenik and D. De Schreye. Proving termination with adornments. In *Proc. 13th LOPSTR*, LNCS 3018, pages 108–109, 2003.
32. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. *Theory and Practice of Logic Programming*, 4:719–751, 2004.
33. J.-G. Smaus. Termination of logic programs using various dynamic selection rules. In *Proc. 20th ICLP*, LNCS 3132, pages 43–57, 2004.