Termination Analysis of Logic Programs based on Dependency Graphs*

Manh Thang Nguyen[1], Jürgen Giesl[2], Peter Schneider-Kamp[2], and Danny De Schreye[1]

[1] Department of Computer Science, K. U. Leuven, Belgium
{ManhThang.Nguyen, Danny.DeSchreye}@cs.kuleuven.be
[2] LuFG Informatik 2, RWTH Aachen, Germany
{giesl, psk}@informatik.rwth-aachen.de

Abstract. This paper introduces a modular framework for termination analysis of logic programming. To this end, we adapt the notions of dependency pairs and dependency graphs (which were developed for term rewriting) to the logic programming domain. The main idea of the approach is that termination conditions for a program are established based on the decomposition of its dependency graph into its strongly connected components. These conditions can then be analysed separately by possibly different well-founded orders. We propose a constraint-based approach for automating the framework. Then, for example, termination techniques based on polynomial interpretations can be plugged in as a component to generate well-founded orders.

1 Introduction

Termination analysis in logic programming (LP) traditionally aims at proving that a given logic program terminates w.r.t. a specific set of queries. Termination proofs are usually done by finding ranking functions that map the states of the program to a sequence of elements of a well-founded domain such that the sequence is decreasing w.r.t. the well-founded order of the domain. Practically, it is sufficient to consider only the states that are involved in loops of the program.

Techniques in termination analysis of LPs can be divided into two groups: the global versus the local approach [4, 6, 5, 8, 10, 12, 26]. In the global approach, one wants to find only **one ranking function** for all loops [8, 10, 26]. In contrast, techniques in the local approach apply **different ranking functions** for different loops [4, 5, 12]. Some automated techniques in the global approach are based on a constraint-based framework to search for a suitable ranking function. This is done by first generating a set of symbolic constraints from all termination conditions. Then, a constraint solver is used to solve the set of constraints, yielding a suitable ranking function for the proof. In the local approach, most techniques use a given small set of norms, and try to prove that (a combination of) these norms can be applied for the termination proof of the program. It is unclear at

* Appeared in *Proc. LOPSTR '07*, LNCS 4915, pages 8-22, 2008.

this stage whether a search for arbitrary norms in the local approach could also be automated using a constraint-based technique like [10].

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

Example 1 (ack). Consider a logic program P computing the Ackermann function. We used a variant with a predecessor predicate $p/2$ in order to illustrate how our technique handles local variables. We want to prove termination of this program w.r.t. the set of queries $S = \{ack(t_1, t_2, t_3) \mid t_1$ and t_2 are ground terms, t_3 is an arbitrary term$\}$.

$$
\begin{array}{l}
p(s(X), X).\\
ack(0, X, s(X)).\\
ack(X, 0, Z) \;{:}{-}\; p(X, Y), ack(Y, s(0), Z).\\
ack(s(X), s(Y), Z) \;{:}{-}\; ack(s(X), Y, Z'), ack(X, Z', Z).
\end{array}
$$

Proving termination of this example based on the local approach involves two ranking functions: the first one measures the size of the first argument and the other measures the size of the second argument of the predicate $ack/3$. However, with the constraint-based global approach, it is impossible to find a single ranking function for the termination proof (if one is restricted to ranking functions based on polynomial interpretations). As a matter of fact, both tools cTI *[25] and* Polytool *[26, 27] fail to prove termination of this example.*

In addition to the local and global approaches which work *directly* on logic programs, there are also several *transformational* approaches which transform logic programs to *term rewrite systems* (TRSs). One of the most recent techniques in this line of work is [31]. However, as demonstrated in [31], it turned out that there remain many LPs whose termination can currently only be proved by tools working with direct approaches. (An example is the "*der*"-program from [9, 26].) On the other hand, there are also many LPs where currently only transformational tools succeed (e.g., the example "*LP/SGST06-shuffle*" from the *Termination Problem Data Base* (TPDB) [32] that is used in the annual *International Competition of Termination Tools* [24]). The present paper tries to solve this problem by porting TRS-techniques so that they can be applied to LPs directly. In this way, we intend to combine the advantages of direct and transformational approaches. Indeed, a first prototypical implementation shows that the new approach of the present paper can handle both the examples "*der*" and "*shuffle*" above as well as other examples that could not be handled by *any* tool up to now (e.g., "*LP/SGST06-snake*" from the TPDB).

More precisely, in this paper we introduce a modular framework for termination analysis of LPs. To this end, the dependency pair technique for termination analysis of TRSs introduced in [1] is adapted to the LP context. With this new technique, termination analysis of programs like Ex. 1 can be done by decomposing it into several simple sub-problems. Each of them can be solved independently by using any suitable well-founded order.

We also propose a constraint-based approach for automating the approach in which termination techniques based on polynomial interpretations can be plugged in as a component to search for well-founded orders.

The paper is organised as follows. In Sect. 2, we provide some preliminaries. In Sect. 3, we introduce a modular framework for proving termination of LPs based on dependency graphs. In Sect. 4, we present a constraint-based approach to automate the framework. Finally, we end with a conclusion in Sect. 5.

2 Preliminaries

A *quasi-order* on a set S is a reflexive and transitive binary relation $\succsim$ defined on elements of S. In this paper, we use quasi-orders comparing atoms with each other and comparing terms with each other. We define the *associated equivalence relation* $\approx$ as $s \approx t$ iff $s \succsim t$ and $t \succsim s$. A *well-founded order* on S is a transitive relation $\succ$ where there is no infinite sequence $s_0 \succ s_1 \succ \ldots$ with $s_i \in S$. A *reduction pair* $(\succsim, \succ)$ consists of a quasi-order $\succsim$ and a well-founded order $\succ$ that are *compatible* (i.e., $t_1 \succsim t_2 \succ t_3$ implies $t_1 \succ t_3$).[3]

We assume familiarity with standard notions of logic programs. In the paper, P denotes a pure logic program and $Term_P$, $Atom_P$ denote the sets of terms and atoms constructed from P respectively. Given an atom A, $rel(A)$ is the predicate occurring in A. Given two atoms A and B, we denote by $mgu(A, B)$ their most general unifier. A *query* Q is a finite sequence of atoms. We consider termination of P w.r.t. Q using the left-to-right selection rule that is commonly used in implementations of logic programming.[4]

Let S be a set of atomic queries. The call set, $Call(P, S)$, is the set of all atoms A, such that a variant of A is the selected atom in some derivation for (P, Q), for some $Q \in S$. In this paper, we use ranking functions and reduction pairs built from norms and level mappings [3]. A *norm* is a mapping $\| \cdot \| : Term_P \to \mathbb{N}$. A *level mapping* is a mapping $| \cdot | : Atom_P \to \mathbb{N}$. An *interargument relation* for a predicate p/n is a relation $R_{p/n} = \{p(t_1, \ldots, t_n) \mid t_i \in Term_P \wedge \varphi_p(t_1, \ldots, t_n)\}$, where (1) $\varphi_p(t_1, \ldots, t_n)$ is a formula of an arbitrary boolean combination of inequalities, and (2) each inequality in φ_p is either $s_i \succsim s_j$ or $s_i \succ s_j$, where s_i, s_j are constructed from $t_1, \ldots, t_n$ by applying function symbols of P. $R_{p/n}$ is *valid* iff for every $p(t_1, \ldots, t_n) \in Atom_P$: $P \models p(t_1, \ldots, t_n)$ implies

[3] In contrast to the definition of "reduction pairs" in term rewriting [21], for the theoretical results in Sect. 3 we do not require $\succsim$ and $\succ$ to be closed under substitutions. But to automate our method, in Sect. 4 we choose relations $\succsim$ and $\succ$ that result from polynomial interpretations and that are closed under substitutions.

[4] By fixing the selection rule, methods for termination analysis can exploit this and become much stronger. This is similar to termination analysis of term rewriting (in particular, when using dependency pairs). Here, termination of *innermost* rewriting is easier to show than termination of full rewriting.

$p(t_1, \dots, t_n) \in R_{p/n}$. A reduction pair $(\succsim, \succ)$ is *rigid* on a term or an atom A if for all substitutions σ, we have $A \approx A\sigma$. A reduction pair $(\succsim, \succ)$ is rigid on a set of terms or atoms if it is rigid on all its elements.

Example 2 (call set, norm, and level mapping for ack). We again regard the program P and the set of queries S in Ex. 1. Then we have $Call(P, S) = S \cup \{\, p(t_1, t_2) \mid t_1$ is a ground term, t_2 is a variable$\}$. Consider the reduction pair $(\succsim, \succ)$ which is induced[5] *by a norm $\|0\| = 0$, $\|s(t)\| = 1 + \|t\|$, $\|X\| = 0$ for all variables X, and by an associated level mapping $|p(t_1, t_2)| = 0$ and $|ack(t_1, t_2, t_3)| = \|t_1\|$. Thus, we have $s(0) \succ 0$, $ack(s(0), X, Y) \succ ack(0, X, Y)$, and $ack(0, X, Y) \approx ack(0, 0, 0)$. Note that $(\succsim, \succ)$ is rigid on $Call(P, S)$. An example for a valid interargument relation w.r.t. $(\succsim, \succ)$ is $R_{p/2} = \{p(t_1, t_2) \mid t_1 \succ t_2\}$.*

3 Dependency Graphs in Logic Programming

Def. 3 adapts the notion of *dependency pairs* [1] from TRSs to the LP setting.

Definition 3 (dependency triple). *A* dependency triple *is a tuple of three elements $\langle H, I, B \rangle$ in which H and B are atoms and I is a list of atoms. For a logic program P, we define the set $DT(P)$ of all dependency triples as $DT(P) = \{\langle H, I, B \rangle \mid H \text{ :- } I, B, \dots \in P\}$.*

Given a program, the number of its dependency triples is finite.

Example 4 (dependency triples of ack). Reconsider the program from Ex. 1. The dependency triples $DT(P)$ of the program are:

$$\langle ack(X, 0, Z), [\,], p(X, Y) \rangle \tag{1}$$
$$\langle ack(X, 0, Z), [p(X, Y)], ack(Y, s(0), Z) \rangle \tag{2}$$
$$\langle ack(s(X), s(Y), Z), [\,], ack(s(X), Y, Z') \rangle \tag{3}$$
$$\langle ack(s(X), s(Y), Z), [ack(s(X), Y, Z')], ack(X, Z', Z) \rangle \tag{4}$$

Now we adapt the notion of the (estimated) *dependency graph* [1] from TRSs to LPs.[6] While "dependency triples" are related to the "binary clauses" of [5], our notion of dependency graphs for LPs is similar to the "atom dependency graph" of [12]. But in contrast to [12], we use dependency graphs to modularize termination proofs such that *several different* reduction pairs can be used in the termination proof of one program.

The nodes of the dependency graph are the dependency triples and there must be an arc from a dependency triple N to a dependency triple M whenever an attempt to solve the "proof goal" N could load to the "proof goal" M. To estimate this, we use the notion of *connectivity.*

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

[6] Our notion should not be confused with the notion of the "(predicate) dependency graph" from [2, 12, 28] that simply represents the dependencies between different predicate symbols.

Definition 5 (connectivity). *Let $\langle H_1, I_1, B_1\rangle$ and $\langle H_2, I_2, B_2\rangle$ be two dependency triples. $\langle H_1, I_1, B_1\rangle$ is* connectable *to $\langle H_2, I_2, B_2\rangle$ iff B_1 unifies with a renamed apart variant of H_2.*

Example 6 (connectivity for ack's dependency triples). In Ex. 1, dependency triple (2) is connectable to (3) and (4), and both dependency triples (3) and (4) are connectable to all dependency triples (1), (2), (3), and (4).

Definition 7 (dependency graph). *Let DT be a set of dependency triples. The dependency graph associated with DT is a directed graph whose vertices are the dependency triples DT and there is an arc from a vertex N to a vertex M iff N is connectable to M. Let P be a logic program. The dependency graph associated with $DT(P)$ is called the dependency graph of P, denoted as $DG(P)$.*

Example 8 (dependency graph for ack). Fig. 1 shows the dependency graph for the ack-program in Ex. 1.

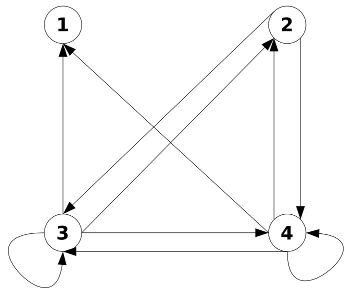

Fig. 1. The dependency graph for the *ack*-program.

Now every infinite execution of the program corresponds to a cycle in the dependency graph. In our setting, a set $\mathcal{C} \neq \emptyset$ of dependency triples is called a *cycle* if for all $N, M \in \mathcal{C}$ there is a non-empty path from N to M in the graph which only traverses dependency triples of $\mathcal{C}$. A cycle $\mathcal{C}$ is a *strongly connected component* (SCC) if $\mathcal{C}$ is not a proper subset of another cycle.

Note that in standard graph terminology, a path $N_0 \to N_1 \to \ldots \to N_k$ in a directed graph forms a cycle if $N_0 = N_k$ and $k \geq 1$. In our context we identify cycles with the *set* of elements that occur in it, i.e., we call $\{N_0, N_1, \ldots, N_{k-1}\}$ a cycle, cf. [15]. Since a set never contains multiple occurrences of an element, this results in several cycling paths being identified with the same set. Similarly, an SCC is a graph in standard graph terminology, whereas we identify an SCC with the set of elements occurring in it. Then indeed, SCCs are the same as maximal cycles.

Example 9 (cycles and SCCs for ack). The dependency graph in Fig. 1 has six cycles $\mathcal{C}_1 = \{(3)\}$, $\mathcal{C}_2 = \{(4)\}$, $\mathcal{C}_3 = \{(2),(3)\}$, $\mathcal{C}_4 = \{(2),(4)\}$, $\mathcal{C}_5 = \{(3),(4)\}$, $\mathcal{C}_6 = \{(2),(3),(4)\}$, and one strongly connected component $\mathcal{C}_6 = \{(2),(3),(4)\}$.

Note that each vertex in the dependency graph corresponds to a possible transition from one state to another state in the computational execution of the program. Each loop of the execution corresponds to a cycle in the graph. Intuitively, a program is terminating if there is no cycle in the graph which is traversed infinitely many times.

To use dependency graphs for termination proofs, we proceed as in [1, 16, 19]. The idea is to inspect each SCC of the dependency graph separately and to find a reduction pair $(\succsim, \succ)$ such that *some* dependency triples of the SCC are *strictly* decreasing (w.r.t. $\succ$) and all others are *weakly* decreasing (w.r.t. $\succsim$). The following definition formalizes when a dependency triple is considered to be "decreasing". It relies on interargument relations for the predicates of the program. Sect. 4 explains how to synthesize such interargument relations and how to find reduction pairs automatically that make dependency triples "decreasing".

Definition 10 (decreasing dependency triples). *Let P be a program. Let $(\succsim, \succ)$ be a reduction pair and $R = \{R_{p_1}, \ldots, R_{p_k}\}$ be a set of interargument relations based on $(\succsim, \succ)$ for the predicates $p_1, \ldots, p_k$ defined in P. Let $N = \langle H, [I_1, \ldots, I_n], B\rangle$ be a dependency triple in $DT(P)$. N is* weakly decreasing *(denoted $(\succsim, R) \models N$) if $H\sigma \succsim B\sigma$ holds for any substitution σ where $(\succsim, \succ)$ is rigid on $H\sigma$ and where $I_1\sigma \in R_{rel(I_1)}, \ldots, I_n\sigma \in R_{rel(I_n)}$. Analogously, N is* strictly decreasing *(denoted $(\succ, R) \models N$) if $H\sigma \succ B\sigma$ holds for any such σ.*

Example 11 (decreasing dependency triples for ack). Consider the reduction pair $(\succsim, \succ)$ from Ex. 2. Let R be the set of valid interargument relations where $R_{ack/3} = \{ack(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in Term_P\}$ and where $R_{p/2}$ is defined as in Ex. 2. Then we have $(\succ, R) \models (2)$. The reason is that for any substitution σ where $(\succsim, \succ)$ is rigid on $ack(X, 0, Z)\sigma$ (i.e., where $X\sigma$ is a ground term) and where $p(X, Y)\sigma \in R_{p/2}$ (i.e., where $X\sigma \succ Y\sigma$), we have $ack(X, 0, Z)\sigma \succ ack(Y, s(0), Z)\sigma$. Similarly, we also have $(\succsim, R) \models (3)$ and $(\succ, R) \models (4)$.

Note that we can restrict ourselves to those SCCs of the dependency graph that can be invoked by calls from $Call(P, S)$. The reason is that only those SCCs can be involved in loops of the execution of the program P, when starting with a query from S. Therefore, we define which SCCs are *reachable* from $Call(P, S)$.

Definition 12 (reachable SCCs). *Let P be a program, S be a set of atomic queries, and $N = \langle H, [I_1, \ldots, I_n], B\rangle$ be a dependency triple. N is* reachable *from $Call(P, S)$ if there is an $A \in Call(P, S)$ such that A unifies with a renamed apart variant of H. An SCC $\mathcal{C}$ in $DG(P)$ is* reachable *from $Call(P, S)$ if there is an $N \in \mathcal{C}$ which is reachable from $Call(P, S)$.*

In the *ack*-example, the only SCC in the dependency graph is reachable from the set $Call(P, S)$ of Ex. 2. But if the *ack*-program contained another clause "q :– q", then the SCC with the resulting dependency triple $\langle q, [\,], q\rangle$ would not be reachable from the call set of Ex. 2. Since it suffices to prove absence of infinite loops only for the *reachable* SCCs, one could then still prove termination of all queries from S. But if one had to regard *all* SCCs, then the termination proof would fail, since the SCC with the dependency triple $\langle q, [\,], q\rangle$ gives rise to an infinite loop. The set of reachable SCCs can easily be (over-)approximated automatically as soon as one has an (over-)approximation of $Call(P, S)$, cf. Sect. 4.

To prove termination, we select an arbitrary reachable SCC $\mathcal{C}$ of the dependency graph. Then, we try to find a reduction pair $(\succsim, \succ)$ such that some dependency triples $\mathcal{C}_\succ \subseteq \mathcal{C}$ are strictly decreasing and all other dependency triples (from $\mathcal{C} \setminus \mathcal{C}_\succ$) are weakly decreasing. This means that the strictly decreasing dependency triples from $\mathcal{C}_\succ$ can never "occur" *infinitely often* in any execution of the program. Thus, we remove the vertices $\mathcal{C}_\succ$ (and all edges originating or ending in these vertices) from the dependency graph. Afterwards the procedure is repeated (with a possibly different reduction pair). If one finally ends up with a graph without reachable SCCs, then termination of the program is proved.

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

However, in our approach we may only use reduction pairs $(\succsim, \succ)$ that are rigid on $Call(P, S)$. This prevents an increase of atoms and terms due to further instantiations in subsequent derivation steps. For details, we refer to [26].

Definition 13 (acceptability). *Let P be a program and S be a set of atomic queries. A subgraph G of the dependency graph $DG(P)$ is called* acceptable *w.r.t. S iff either G has no SCC reachable from $Call(P, S)$ or else, G has such an SCC $\mathcal{C}$ and there is a reduction pair $(\succsim, \succ)$ and a set of valid interargument relations $R = \{R_{p_1}, \ldots, R_{p_k}\}$ based on $(\succsim, \succ)$ for the predicates $p_1, \ldots, p_k$ in P, such that*

- *$(\succsim, \succ)$ is rigid on $Call(P, S)$,*
- *there is a non-empty subset $\mathcal{C}_\succ \subseteq \mathcal{C}$ such that $(\succ, R) \models N$ for all $N \in \mathcal{C}_\succ$ and $(\succsim, R) \models N$ for all $N \in \mathcal{C} \setminus \mathcal{C}_\succ$, and*
- *the graph resulting from G by removing all vertices in $\mathcal{C}_\succ$ is also acceptable.*

Example 14 (termination of ack). The dependency graph of the ack-program in Fig. 1 has only one SCC. First, we select a reduction pair $(\succsim, \succ)$. We re-use the reduction pair from Ex. 2 and the valid interargument relations R from Ex. 11. As shown in Ex. 11, then (2) and (4) are strictly decreasing, whereas (3) is only weakly decreasing. Thus, we remove (2) and (4) from the dependency graph.

The remaining graph has only one vertex (3) and an edge from (3) to itself. Thus, now the only SCC is $\{(3)\}$. We select another reduction pair $(\succsim', \succ')$ which is defined by the same norm $|| \cdot ||$ as in Ex. 2 and by a new level mapping with $|ack(t_1, t_2, t_3)| = \|t_2\|$. Now we have $(\succ', R) \models (3)$, i.e., (3) can be removed.

The remaining graph is empty and thus, it has no SCC. Hence, termination of the ack-program is proved.

The following theorem states the soundness of our approach.[7]

Theorem 15 (soundness). *A program P is terminating w.r.t. a set of atomic queries S if its dependency graph $DG(P)$ is acceptable w.r.t. S.*

Proof. If P is not terminating w.r.t. S, then there is an $A \in Call(P, S)$, an infinite sequence of (variable renamed) dependency triples $N_0, N_1, \ldots$ with $N_i = \langle H_i, [I_{i1}, \ldots, I_{in_i}], B_i \rangle$, and substitutions $\theta_0, \theta_1, \ldots$ and $\sigma_0, \sigma_1, \ldots$ such that

[7] Note that the proof of Thm. 15 is similar to the one for the dependency pair method in [1]. So in contrast to the "local approaches" [4, 5, 12] for logic programs and the size-change-based methods [23, 29, 33] for other programming paradigms, Thm. 15 does not rely on Ramsey's theorem [6, 30].

- $\theta_0 = mgu(A, H_0)$
- σ_i is a computed answer substitution for the query $(I_{i1}, \ldots, I_{in_i})\theta_i$
- $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$

Since there is an edge from N_i to N_{i+1} for all i in the dependency graph, the sequence $N_0, N_1, \ldots$ contains an infinite tail which traverses a cycle of the dependency graph infinitely often.

For any subgraph G of the dependency graph, we show that if this infinite tail is contained in G, then G cannot be acceptable. We use induction on the number of vertices in G. The claim is obviously true if G does not contain any SCC reachable from $\mathit{Call}(P, S)$. Thus, let G contain a reachable SCC $\mathcal{C}$ as in Def. 13. If the infinite tail is still contained in the acceptable subgraph resulting from removing all vertices from $\mathcal{C}_{\succ}$, the claim follows from the induction hypothesis.

It remains to regard the case where the infinite tail $N_i, N_{i+1}, \ldots$ only traverses dependency triples from $\mathcal{C}$ and where a dependency triple from $\mathcal{C}_{\succ}$ is traversed infinitely often. Thus, we obtain an infinite sequence

$$\begin{array}{ll}
H_i\theta_i & \approx \text{(by rigidity, since } H_i\theta_i = B_{i-1}\theta_{i-1}\sigma_{i-1}\theta_i \\
 & \quad \text{and } B_{i-1}\theta_{i-1}\sigma_{i-1} \in \mathit{Call}(P, S)) \\
H_i\theta_i\sigma_i\theta_{i+1} & \succsim \\
B_i\theta_i\sigma_i\theta_{i+1} & = \\
H_{i+1}\theta_{i+1} & \approx \text{(by rigidity, since } H_{i+1}\theta_{i+1} = B_i\theta_i\sigma_i\theta_{i+1} \\
 & \quad \text{and } B_i\theta_i\sigma_i \in \mathit{Call}(P, S)) \\
H_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} & \succsim \\
B_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} & = \\
\ldots &
\end{array}$$

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

Thm. 15 can be considered an extension of Thm. 1 in [9], where a strict decrease is required for every (mutually) recursive clause of the program, instead of a decrease on the SCCs as in our theorem above. In particular, Ex. 1 cannot be solved using Thm. 1 of [9].

The converse direction of Thm. 15 does not hold since "acceptability" requires the reduction pair to be rigid on $\mathit{Call}(P, S)$. Hence, the program with the two clauses "$p(X) :- q(X, Y), p(Y)$" and "$q(a, b)$" and the set of queries $S = \{p(X)\}$ from [9] is a counterexample to the completeness direction of Thm. 15.

4 Toward automation

Now we discuss how to automate our approach. In Sect. 4.1, we present a general algorithm to mechanize the technique of Def. 13 and Thm. 15. Then, in Sect. 4.2 we show how to plug in existing approaches for the generation of polynomial interpretations in order to synthesize suitable reduction pairs automatically.

4.1 A general framework

Def. 13 and Thm. 15 provide a method to detect termination of a program P w.r.t. a set of queries S. The method can be automated as follows:

1. Compute the dependency graph $DG(P)$ and remove all vertices which are not reachable from $Call(P, S)$. Decompose the remaining graph into its SCCs.
2. If the set of SCCs is empty, stop with "success" (the program is terminating). Otherwise, select one SCC from the set.
3. If the selected SCC cannot be proved to be acceptable, we stop with "fail" (the program may be non-terminating). If the SCC is acceptable, we delete the strictly decreasing vertices from it and decompose the remaining graph into its SCCs. We add this set of SCCs to the remaining set of SCCs and continue with Step 2.

Step 1 guarantees that all remaining vertices and hence, also all remaining SCCs are reachable from $Call(P, S)$. Therefore, it is obvious that all SCCs decomposed later in Step 3 are also reachable from $Call(P, S)$.

Fig. 2 shows an algorithm based on Step 1-3. In the figure, $reach(G)$ removes all dependency triples from the dependency graph G which are not reachable from $Call(P, S)$, $gcc(G)$ computes the set of SCCs of a graph G, $select(S)$ returns an element selected from the set S, $minus(S_1, S_2)$ returns a set containing all elements that are in the set S_1 but not in S_2, ":=" is the assignment and "=" is the comparison operator. The function $exist(G, O)$ checks if there exists a reduction pair and a set of interargument relations such that G is acceptable. If yes, then the reduction pair is assigned to O. The function $induce(G, O)$ returns a graph which results from G by removing all vertices N where $(\succ, R) \models N$ and their related arcs. Finally, $union(S_1, S_2)$ returns a set that is the union of the sets S_1 and S_2.

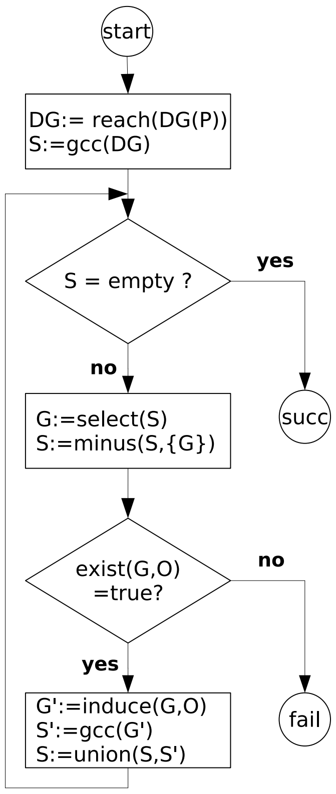

Fig. 2. Our algorithm to verify termination of programs.

Since $Call(P, S)$ can be infinite in general, it is undecidable whether a dependency triple is reachable from $Call(P, S)$. Heuristically, it can be done by first abstracting $Call(P, S)$ to a finite set of call patterns and then checking if there exists a call pattern which unifies with the vertex [26, 27].

The function $exist(G, O)$ is the core of the algorithm. Interestingly, it does not force us to use a fixed type of orders. Therefore, the algorithm can be considered a framework where different termination techniques for finding well-founded orders can be plugged in to support the function $exist(G, O)$. In Sect. 4.2, we discuss how the termination analysis technique based on polynomial interpretations from [26, 27] can be applied to the framework.

4.2 Generating well-founded orders

Since arbitrary techniques can be applied to search for reduction pairs required in the function $exist(G, O)$, an obvious option is to use polynomial interpretations, one of the most powerful techniques in termination analysis of logic programming and term rewriting systems [7, 14, 20, 22, 26, 27].[8] The main idea of the technique is to map each function and predicate symbol to a polynomial, under a polynomial interpretation $|\cdot|_I$. The polynomials are considered as functions of type $\mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$, and the coefficients of the polynomials are also in $\mathbb{N}$. In this way, terms and atoms are mapped to polynomials as well.

Example 16 (polynomial interpretation for ack). The norm and level mapping of Ex. 2 correspond to the polynomial interpretation $|0|_I = 0$, $|s(X)|_I = 1 + X$, $|p(X, Y)|_I = 0$, $|ack(X, Y, Z)|_I = X$. *So we have* $|ack(s(X), s(Y), Z)|_I = |s(X)|_I = 1 + X$ *and* $|ack(X, Z', Z)|_I = |X|_I = X$.

For any polynomial interpretation I, we define a quasi-order $\succsim_I$ on terms and atoms: $t_1 \succsim_I t_2$ iff $|t_1|_I \geq |t_2|_I$ holds for all instantiations of the variables in the polynomials $|t_1|_I$ and $|t_2|_I$ by natural numbers. (It suffices to regard only natural numbers n where $n \geq |c|_I$ for all (constant) function symbols $c/0$ of P.) Similarly, the well-founded order $\succ_I$ is defined as $t_1 \succ_I t_2$ iff $|t_1|_I > |t_2|_I$ holds for all instantiations of the variables in the polynomials $|t_1|_I$ and $|t_2|_I$ by such natural numbers. Obviously, $(\succsim_I, \succ_I)$ is always a reduction pair. Moreover, a term or atom t is rigid w.r.t. $(\succsim_I, \succ_I)$ iff $|t|_I$ contains no variables.

Now, all conditions in Def. 13 can be stated as constraints on polynomials. A reduction pair $(\succsim_I, \succ_I)$ satisfies the conditions in Def. 13 iff the polynomial interpretation $|\cdot|_I$ satisfies the resulting constraints on the polynomials.

Of course, we do not choose a particular polynomial interpretation. Instead, we want to *search* for a suitable one automatically. In the philosophy of the constraint-based approach in [10, 27], we introduce a general symbolic form for the polynomial associated with each predicate and function symbol, and for interargument relations. Since there is no finite symbolic representation for all possible polynomials, we restrict ourselves to fixed types of polynomials. For example, each function and predicate symbol can be associated with a linear polynomial and each interargument relation for a predicate can be expressed in linear form as follows.[9] Here, f_i, p_i^L, and p_i^R are "abstract" symbolic coefficients.

[8] Other possible options would be recursive path orders [11], matrix orders [13], etc.

[9] As already observed for term rewriting, in the vast majority of examples, *linear* polynomial interpretations are already sufficient if they are used in connection with the dependency pair method. But of course, our approach also permits the use of polynomials with higher degree.

In order to complete the termination proof, one has to find suitable instantiations of these coefficients with natural numbers.

- $|f(X_1,\ldots,X_n)|_I = f_0 + \sum_{i=1}^{n} f_i X_i$,
- $R_{p/n} = \{\, p(t_1,\ldots,t_n) \mid p_0^L + \sum_{i=1}^{n} p_i^L |t_i|_I \;\geq\; p_0^R + \sum_{i=1}^{n} p_i^R |t_i|_I \,\}$.

Based on the symbolic forms for polynomial interpretations and interargument relations, all termination conditions expressed in Def. 13 can also be reformulated symbolically. Specifically, the conditions for the function *exist*(G, O) (which checks whether G is acceptable) are expressed as a set of polynomial constraints with symbolic coefficients (e.g. $f_i, p_i^L, p_i^R, \ldots$). The central question is how to search for an instantiation of these symbolic coefficients such that the set of constraints is satisfied. In [27], we introduced a transformational approach to transform all constraints into a sufficient set of Diophantine constraints on natural numbers where all unknown symbolic coefficients become variables (cf. also [20]). A solution for the Diophantine constraints gives a suitable reduction pair $(\succsim_I, \succ_I)$ and a set of valid interargument relations based on the reduction pair. Finding such a solution can be done by using any available Diophantine constraint solver, e.g. [7, 14]. Finally, the rigidity condition can be symbolised based on the *rigid type graph*. For more details, we refer to [26, 27].

Example 17 (symbolic termination conditions for ack). Reconsider Ex. 1. We define an "abstract" symbolic polynomial interpretation as $|0|_I = c$, $|s(X)|_I = s_0 + s_1 X$, $|p(X,Y)|_I = p_0 + p_1 X + p_2 Y$, $|ack(X,Y,Z)|_I = a_0 + a_1 X + a_2 Y + a_3 Z$, *and a set of interargument relations* $R = \{R_{p/2}, R_{ack/3}\}$ *with*

$$\begin{array}{lll} R_{p/2} & = \{p(t_1,t_2) & \mid\; p_0^L + p_1^L|t_1|_I + p_2^L|t_2|_I \;\geq \\ & & \quad\; p_0^R + p_1^R|t_1|_I + p_2^R|t_2|_I \quad \} \\ R_{ack/3} & = \{ack(t_1,t_2,t_3) & \mid\; a_0^L + a_1^L|t_1|_I + a_2^L|t_2|_I + a_3^L|t_3|_I \;\geq \\ & & \quad\; a_0^R + a_1^R|t_1|_I + a_2^R|t_2|_I + a_3^R|t_3|_I \quad \}. \end{array}$$

The conditions for acceptability of the dependency graph can be reformulated as follows:

1. For any dependency triple $N \in \{(2),(3),(4)\}$, *we require* $(\succsim_I, R) \models N$:

$$\begin{array}{l} \forall X,Y,Z\; [\quad p_0^L + p_1^L X + p_2^L Y \;\geq\; p_0^R + p_1^R X + p_2^R Y \\ \qquad\quad \Rightarrow a_0 + a_1 X + a_2 c + a_3 Z \;\geq\; a_0 + a_1 Y + a_2(s_0 + s_1 c) + a_3 Z\,] \quad \wedge \\[1ex] \forall X,Y,Z,Z'\; [a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;\geq \\ \qquad\quad a_0 + a_1(s_0 + s_1 X) + a_2 Y + a_3 Z'\,] \quad \wedge \\[1ex] \forall X,Y,Z,Z'\; [\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \;\geq \\ \qquad\quad a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z' \\ \qquad\quad \Rightarrow a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;\geq \\ \qquad\quad a_0 + a_1 X + a_2 Z' + a_3 Z\,] \end{array}$$

2. There exists some dependency triple $N \in \{(2),(3),(4)\}$ *with* $(\succ_I, R) \models N$:

$$\begin{array}{l}\forall X, Y, Z\ [\quad p_0^L + p_1^L X + p_2^L Y \ \geq\ p_0^R + p_1^R X + p_2^R Y \\ \qquad\qquad \Rightarrow a_0 + a_1 X + a_2 c + a_3 Z \ >\ a_0 + a_1 Y + a_2(s_0 + s_1 c) + a_3 Z\,] \quad \vee \\ \forall X, Y, Z, Z'\ [a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \ > \\ \qquad\qquad a_0 + a_1(s_0 + s_1 X) + a_2 Y + a_3 Z'\,] \quad \vee \\ \forall X, Y, Z, Z'\ [\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \ \geq \\ \qquad\qquad a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z' \\ \qquad\qquad \Rightarrow a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \ > \\ \qquad\qquad a_0 + a_1 X + a_2 Z' + a_3 Z\,]\end{array}$$

3. *The valid interargument condition for $p/2$:*

$$\forall X\ [p_0^L + p_1^L(s_0 + s_1 X) + p_2^L X \ \geq\ p_0^R + p_1^R(s_0 + s_1 X) + p_2^R X\,]$$

4. *The valid interargument condition for $ack/3$:*

$$\begin{array}{l}\forall X\ [a_0^L + a_1^L c + a_2^L X + a_3^L(s_0 + s_1 X) \ \geq\ a_0^R + a_1^R c + a_2^R X + a_3^R(s_0 + s_1 X)\,] \quad \wedge \\ \forall X, Y, Z\ [\quad p_0^L + p_1^L X + p_2^L Y \ \geq\ p_0^R + p_1^R X + p_2^R Y \\ \qquad\qquad \wedge\ a_0^L + a_1^L Y + a_2^L(s_0 + s_1 c) + a_3^L Z \ \geq \\ \qquad\qquad a_0^R + a_1^R Y + a_2^R(s_0 + s_1 c) + a_3^R Z \\ \qquad\qquad \Rightarrow a_0^L + a_1^L X + a_2^L c + a_3^L Z \ \geq\ a_0^R + a_1^R X + a_2^R c + a_3^R Z\,] \quad \wedge \\ \forall X, Y, Z, Z'\ [\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \ \geq \\ \qquad\qquad a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z' \\ \qquad\qquad \wedge\ a_0^L + a_1^L X + a_2^L Z' + a_3^L Z \ \geq \\ \qquad\qquad a_0^R + a_1^R X + a_2^R Z' + a_3^R Z \\ \qquad\qquad \Rightarrow a_0^L + a_1^L(s_0 + s_1 X) + a_2^L(s_0 + s_1 Y) + a_3^L Z \ \geq \\ \qquad\qquad a_0^R + a_1^R(s_0 + s_1 X) + a_2^R(s_0 + s_1 Y) + a_3^R Z\,]\end{array}$$

5. *The rigidity property for $Call(P, S) = \{ack(t_1, t_2, t_3) \mid t_1$ and t_2 are ground terms, t_3 is an arbitrary term$\} \cup \{p(t_1, t_2) \mid t_1$ is a ground term, t_2 is a variable$\}$:*

$$p_2 = 0 \wedge a_3 = 0$$

All the constraints above are satisfied by the following instantiation of the symbolic variables: $c = 0$, $s_0 = s_1 = 1$, $p_0 = p_1 = p_2 = 0$, $a_0 = 0$, $a_1 = 1$, $a_2 = a_3 = 0$, $p_0^L = 0$, $p_1^L = 1$, $p_2^L = 0$, $p_0^R = p_1^R = 1$, $p_2^R = 0$ and $a_i^L = a_i^R = 0$ for all $i \in \{0, 1, 2, 3\}$. This instantiation turns the abstract polynomial interpretation of Ex. 17 into the concrete polynomial interpretation of Ex. 16 (i.e., now it corresponds to the norm and level mapping of Ex. 2). Similarly, the "abstract" interargument relations of of Ex. 17 are turned into the concrete interargument relations of Ex. 2 and Ex. 11 (i.e., $R_{p/2} = \{p(t_1, t_2) \mid t_1 \succ_I t_2\}$ and $R_{ack/3} = \{ack(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in Term_P\}$).

So instead of fixing a polynomial interpretation and interargument relations *before* performing the termination proof, now we only fix the degree of the polynomials used in the polynomial interpretation (e.g., linear or quadratic ones). Then we can automatically generate symbolic constraints and try to solve them afterwards. In this way, suitable polynomial interpretations and interargument relations can be synthesized fully automatically.

5 Conclusion

We have introduced a new framework for termination analysis of LPs based on dependency triples and dependency graphs. Although the notion of dependency pairs and dependency graphs is very popular in the domain of termination analysis of TRS [1, 15, 16, 18, 19], this is the first time that it is applied for LP termination analysis directly. Our contribution is twofold: **(1)** it results in a weaker condition for verifying termination of LPs, where the decrease condition is established for the strongly connected components of the dependency graph, instead of at the clause level as it has been done before; **(2)** it introduces a modular approach in which termination conditions can be separated into different groups, each of which can be treated independently by automatically searching for different suitable well-founded orderings.

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

The authors of this paper were involved in the implementation of two of the most powerful automated termination analysers for LPs (Polytool which follows the approach of [26, 27] and AProVE [17] which transforms LPs to TRSs and then tries to prove termination of the resulting TRS [31].) AProVE was the most successful termination prover for logic programs, functional programs, and term rewrite systems in all annual *International Competitions of Termination Tools* 2004 - 2007 [24], where Polytool obtained a close second place for logic programs in the 2007 competition. As mentioned in [31], there exist many LPs where termination can currently only be proved by transformational tools like AProVE and there are also many examples where the termination proof only succeeds with direct tools like Polytool, cf. Sect. 1. Our current work intends to combine the advantages of both approaches by adapting TRS-techniques like dependency pairs to direct termination approaches for LPs. While the present paper only adapted basic concepts of the dependency pair method to the LP setting, in the future we will also try to adapt further more sophisticated "dependency pair processors" [16, 18] as well.

Currently, we are working on an implementation of the results of this paper within Polytool. Here, we try to re-use algorithms from the dependency pair implementation of AProVE. As mentioned in Sect. 1, a first prototypical implementation already shows that in this way one can handle (a) examples that could up to now only be solved with direct tools such as [26, "*der*"], (b) examples that could up to now only be solved with transformational tools based on dependency pairs such as [32, "*LP/SGST06-shuffle*"], as well as (c) examples like [32, "*LP/SGST06-snake*"] that could not be solved by any tool up to now. Note that the Diophantine constraints resulting from our new approach according to Sect. 4 are usually smaller and simpler than the ones generated by the previous version of Polytool [26, 27]. But already in the previous version of Polytool, solving these constraints automatically was no problem in practice. (To this end, the SAT-based constraint solver of AProVE was used [14].) Thus, this solver will also be used for the automatic generation of the required polynomial interpretations and interargument relations in our new approach.

6 Acknowledgement

We are grateful to the referees for many helpful suggestions. Manh Thang Nguyen is supported by *FWO/2006/09: Termination analysis: Crossing paradigm borders*. Peter Schneider-Kamp and Jürgen Giesl are supported by the *Deutsche Forschungsgemeinschaft (DFG)*, grant *GI 274/5-1*.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.
3. A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, 1994.
4. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
6. M. Codish and S. Genaim. Proving termination one loop at a time. In *Proc. WLPE '03*, 2003.
7. E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.
8. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS '92*, pages 481–488, 1992.
9. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In *Computational Logic: Logic Programming and Beyond*, LNCS 2407, pages 187–210. 2002.
10. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based automatic termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.
11. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.
12. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):117–156, 2001.

13. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In *Proc. IJCAR '06*, LNAI 4130, pages 574–588, 2006.
14. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT '07*, LNCS 4501, pages 340–354, 2007.
15. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
16. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
17. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
18. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
19. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.
20. H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.
21. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. PPDP '99*, pages 48–62, 1999. LNCS 1702.
22. D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
24. C. Marché and H. Zantema. The termination competition. In *Proc. RTA '07*, LNCS 4533, pages 303–313, 2007. See also the website `http://www.lri.fr/~marche/termination-competition`.
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. M. T. Nguyen and D. De Schreye. Polynomial interpretations as a basis for termination analysis of logic programs. In *Proc. ICLP '05*, LNCS 3668, pages 311–325, 2005.
27. M. T. Nguyen and D. De Schreye. Polytool: Proving termination automatically based on polynomial interpretations. In *Proc. LOPSTR '06*, LNCS 4407, pages 210–218, 2007. Extended version appeared as Technical report, Department of Computer Science, K. U. Leuven, Belgium.
28. L. Plümer. *Termination Proofs for Logic Programs.* Springer-Verlag, 1990.
29. A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. LICS '04*, pages 32–41, 2004.
30. F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Society*, 30:264–286, 1930.
31. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs by term rewriting. In *Proc. LOPSTR '06*, LNCS 4407, pages 177–193, 2007.
32. The termination problem data base. `http://www.lri.fr/~marche/tpdb`.
33. R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.