Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Bounded Programs: A New Decidable Class of Logic Programs with Function Symbols

Sergio Greco, Cristian Molinaro, Irina Trubitsyna
DIMES, Università della Calabria
87036 Rende (CS), Italy
{greco,cmolinaro,trubitsyna}@dimes.unical.it

Abstract

While function symbols are widely acknowledged as an important feature in logic programming, they make common inference tasks undecidable. To cope with this problem, recent research has focused on identifying classes of logic programs imposing restrictions on the use of function symbols, but guaranteeing decidability of common inference tasks. This has led to several criteria, called *termination criteria*, providing sufficient conditions for a program to have finitely many stable models, each of finite size. This paper introduces the new class of *bounded programs* which guarantees the aforementioned property and strictly includes the classes of programs determined by current termination criteria. Different results on the correctness, the expressiveness, and the complexity of the class of bounded programs are presented.

1 Introduction

In recent years, there has been a great deal of interest in enhancing answer set solvers by supporting functions symbols. Function symbols often make modeling easier and the resulting encodings more readable and concise. The main problem with the introduction of function symbols is that common inference tasks become undecidable. In order to cope with this issue, research has focused on identifying sufficient conditions for a program to have finitely many stable models, each of finite size, leading to different criteria (that we call *termination criteria*).

In this paper, we present a more powerful technique to check whether a program has a finite number of stable models and each of them is of finite size. The termination criterion we propose relies on two powerful tools: the *labelled argument graph*, a directed graph whose edges are labelled with useful information on how terms are propagated from the body to the head of rules, and the *activation graph*, a directed graph specifying whether a rule can "activate" another rule (we will make clear what this means in the following). The labelled argument graph is used in synergy with the activation graph for a better understanding of how terms are propagated. Another relevant aspect that distinguishes our criterion from past work is that current termination criteria analyze one group of arguments (depending on each other) at a time, without looking at how groups of arguments are related. On the contrary, our criterion is able to perform an analysis of how groups of arguments affect each other.

Example 1 Consider the logic program $\mathcal{P}_1$ below:

$$\begin{array}{ll} r_0: & \mathtt{count([a,b,c],0).} \\ r_1: & \mathtt{count(L,I+1)} \leftarrow \mathtt{count([X|L],I).} \end{array}$$

The bottom-up evaluation of $\mathcal{P}_1$ terminates yielding the set of atoms $\mathtt{count([a,b,c],0)}$, $\mathtt{count([b,c],1)}$, $\mathtt{count([c],2)}$, and $\mathtt{count([\,],3)}$. The query goal $\mathtt{count([\,],L)}$ can be used to retrieve the length $\mathtt{L}$ of list $\mathtt{[a,b,c]}$.[1] □

None of the current termination criteria is able to realize that the bottom-up evaluation of $\mathcal{P}_1$ terminates. Intuitively, this is because they analyze how the first argument of count affects itself and how the second argument of count affects itself, but miss noticing that the growth of latter is bounded by the reduction of the former. One of the novelties of our criterion is the capability of doing this kind of reasoning—indeed, as shown in Example 8, our criterion is able to realize that the bottom-up evaluation of $\mathcal{P}_1$ terminates.

The following example shows another classical program computing the concatenation of two lists.

Example 2 Consider the following logic program $\mathcal{P}_2$:

$$\begin{array}{l} \mathtt{append([\,],L,L).} \\ \mathtt{append([X|L1],L2,[X|L3])} \leftarrow \mathtt{append(L1,L2,L3).} \end{array}$$

As an example, the top-down evaluation of $\mathcal{P}_2$ w.r.t. the query goal $\mathtt{append([a,b],[c,d],W)}$ gives the concatenation of the lists $\mathtt{[a,b]}$ and $\mathtt{[c,d]}$. This query cannot be evaluated by standard bottom-up evaluators as the second rule in $\mathcal{P}_2$ is not *range restricted* (because variable $\mathtt{X}$ in the rule head does not appear in a positive body literal). However, one might apply rewriting techniques such as magic-set to rewrite $\mathcal{P}_2$ into a program whose bottom-up evaluation gives the same result [Beeri and Ramakrishnan, 1991]. In this case, $\mathcal{P}_2$ and the

[1]Notice that $\mathcal{P}_1$ has been written so as to count the number of elements in a list when evaluated in a bottom-up fashion, and therefore differs from the classical formulation relying on a top-down evaluation strategy. However, as shown in Example 2, programs relying on a top-down evaluation strategy can be rewritten into programs whose bottom-up evaluation gives the same result. Our criterion can then be applied to rewritten programs.

query goal append([a, b], [c, d], W) give the following program $\mathcal{P}'_2$:

```
r0 : magic_append([a, b], [c, d]).
r1 : magic_append(L1, L2)      ← magic_append([X|L1], L2).
r2 : append([ ], L, L)         ← magic_append([ ], L).
r3 : append([X|L1],L2, [X|L3]) ← magic_append([X|L1], L2),
                                 append(L1, L2, L3).        □
```

Also in the example above, current termination criteria are not able to recognize termination of the bottom-up evaluation of $\mathcal{P}'_2$, while the new criterion introduced in this paper does.

A significant body of work has been done on termination of logic programs under top-down evaluation [Schreye and Decorte, 1994; Voets and Schreye, 2011; Marchiori, 1996; Ohlebusch, 2001; Codish *et al.*, 2005; Serebrenik and De Schreye, 2005; Nishida and Vidal, 2010; Schneider-Kamp *et al.*, 2009b; 2009a; 2010; Nguyen *et al.*, 2007; Bruynooghe *et al.*, 2007; Bonatti, 2004; Baselice *et al.*, 2009] and in the area of term rewriting [Zantema, 1995; Sternagel and Middeldorp, 2008; Arts and Giesl, 2000; Endrullis *et al.*, 2008; Ferreira and Zantema, 1996]. In this paper, we consider logic programs with function symbols *under the stable model semantics* [Gelfond and Lifschitz, 1988; 1991], and thus all the excellent works above cannot be straightforwardly applied to our setting (for a discussion on this see, e.g., [Calimeri *et al.*, 2008; Alviano *et al.*, 2010]). In our context, [Calimeri *et al.*, 2008] introduced the class of *finitely ground programs*, guaranteeing the existence of a finite set of stable models, each of finite size, for programs in the class. Since membership in the class is not decidable, decidable subclasses have been proposed: ω-*restricted programs* [Syrjänen, 2001], λ-*restricted programs* [Gebser *et al.*, 2007], *finite domain programs* [Calimeri *et al.*, 2008], *argument-restricted programs* [Lierler and Lifschitz, 2009], *safe programs* [Greco *et al.*, 2012], and Γ-*acyclic programs* [Greco *et al.*, 2012].

The main contribution of this paper is the introduction of the new decidable class $\mathcal{BP}$ of *bounded programs* that strictly includes all the aforementioned classes in the literature. We present different results on the correctness, the expressiveness, and the complexity of $\mathcal{BP}$.

2 Logic Programs with Function Symbols

Syntax. We assume to have infinite sets of *constants*, *variables*, *predicate symbols*, and *function symbols*. Each predicate and function symbol is associated with a fixed *arity*, which is a natural number.

A *term* is either a constant, a variable, or an expression of the form $f(t_1, ..., t_m)$, where f is a function symbol of arity m and the t_i's are terms (in the first two cases we say the term is *simple* while in the last case we say it is *complex*). The binary relation *subterm* over terms is recursively defined as follows: every term is a subterm of itself; if t is a complex term of the form $f(t_1, ..., t_m)$, then every t_i is a subterm of t, for $1 \leq i \leq m$; if t_1 is a subterm of t_2 and t_2 is a subterm of t_3, then t_1 is a subterm of t_3.

An *atom* is of the form $p(t_1, ..., t_n)$, where p is a predicate symbol of arity n and the t_i's are terms (we also say that the atom is a p-atom). A *literal* is either an atom A (*positive* literal) or its negation $\neg A$ (*negative* literal).

A *rule* r is of the form:

$$A_1 \vee ... \vee A_m \leftarrow B_1, ..., B_k, \neg C_1, ..., \neg C_n$$

where $m > 0$, $k \geq 0$, $n \geq 0$, and $A_1, ..., A_m, B_1, ..., B_k, C_1, ..., C_n$ are atoms. The disjunction $A_1 \vee ... \vee A_m$ is called the *head* of r and is denoted by $head(r)$; the conjunction $B_1, ..., B_k, \neg C_1, ..., \neg C_n$ is called the *body* of r and is denoted by $body(r)$. The *positive* (resp. *negative*) *body* of r is the conjunction $B_1, ..., B_k$ (resp. $\neg C_1, ..., \neg C_n$) and is denoted by $body^+(r)$ (resp. $body^-(r)$). With a slight abuse of notation we use $head(r)$ (resp. $body(r)$, $body^+(r)$, $body^-(r)$) to also denote the *set* of atoms (resp. literals) appearing in the head (resp. body, positive body, negative body) of r. If $m = 1$, then r is *normal*; if $n = 0$, then r is *positive*. A *program* is a finite set of rules. A program is *normal* (resp. *positive*) if every rule in it is normal (resp. positive). A term (resp. an atom, a literal, a rule, a program) is said to be *ground* if no variables occur in it. A ground normal rule with an empty body is also called a *fact*. We assume that programs are *range restricted*, i.e., for each rule, the variables appearing in the head or in negative body literals also appear in some positive body literal.

The *definition* of a predicate symbol p in a program $\mathcal{P}$ consists of all rules in $\mathcal{P}$ with p in the head. Predicate symbols are partitioned into two different classes: *base* predicate symbols, whose definition can contain only facts, and *derived* predicate symbols, whose definition can contain any rule. Facts defining base predicate symbols are called *database facts*.[2]

Given a predicate symbol p of arity n, the i-th *argument* of p is an expression of the form $p[i]$, for $1 \leq i \leq n$. If p is a base (resp. derived) predicate symbol, then $p[i]$ is said to be a *base* (resp. *derived*) argument. The set of all arguments of a program $\mathcal{P}$ is denoted by $arg(\mathcal{P})$.

Semantics. Consider a program $\mathcal{P}$. The *Herbrand universe* $H_{\mathcal{P}}$ of $\mathcal{P}$ is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in $\mathcal{P}$. The *Herbrand base* $B_{\mathcal{P}}$ of $\mathcal{P}$ is the set of ground atoms which can be built using predicate symbols appearing in $\mathcal{P}$ and ground terms of $H_{\mathcal{P}}$. A rule r' is a *ground instance* of a rule r in $\mathcal{P}$ if r' can be obtained from r by substituting every variable in r with some ground term in $H_{\mathcal{P}}$. We use $ground(r)$ to denote the set of all ground instances of r and $ground(\mathcal{P})$ to denote the set of all ground instances of the rules in $\mathcal{P}$, i.e., $ground(\mathcal{P}) = \cup_{r \in \mathcal{P}} ground(r)$. An *interpretation* of $\mathcal{P}$ is any subset I of $B_{\mathcal{P}}$. The truth value of a ground atom A w.r.t. I, denoted $value_I(A)$, is *true* if $A \in I$, *false* otherwise. The truth value of $\neg A$ w.r.t. I, denoted $value_I(\neg A)$, is *true* if $A \notin I$, *false* otherwise. The truth value of a conjunction of ground literals $C = L_1, ..., L_n$ w.r.t. I is $value_I(C) = \min(\{value_I(L_i) \mid 1 \leq i \leq n\})$—here the ordering *false* < *true* holds—whereas the truth value of a disjunction of ground literals $D = L_1 \vee ... \vee L_n$ w.r.t. I is $value_I(D) = \max(\{value_I(L_i) \mid 1 \leq i \leq n\})$; if $n = 0$, then $value_I(C) = true$ and $value_I(D) = false$. A ground rule r is *satisfied* by I, denoted $I \models r$, if $value_I(head(r)) \geq value_I(body(r))$; we write $I \not\models r$ if r is not satisfied by I. Thus, a ground rule r with empty body is satisfied by I if

[2]Database facts are not shown in our examples as they are not relevant for the proposed criterion.

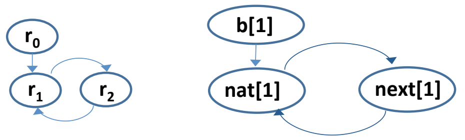

Figure 1: Activation (left) and argument (right) graphs of $\mathcal{P}_3$.

$value_I(head(r)) = true$. An interpretation of $\mathcal{P}$ is a *model* of $\mathcal{P}$ if it satisfies every ground rule in $ground(\mathcal{P})$. A model M of $\mathcal{P}$ is minimal if no proper subset of M is a model of $\mathcal{P}$. The set of minimal models of $\mathcal{P}$ is denoted by $\mathcal{MM}(\mathcal{P})$.

Given an interpretation I of $\mathcal{P}$, let $\mathcal{P}^I$ denote the ground positive program derived from $ground(\mathcal{P})$ by *(i)* removing every rule containing a negative literal $\neg A$ in the body with $A \in I$, and *(ii)* removing all negative literals from the remaining rules. An interpretation I is a *stable model* of $\mathcal{P}$ if and only if $I \in \mathcal{MM}(\mathcal{P}^I)$. The set of stable models of $\mathcal{P}$ is denoted by $\mathcal{SM}(\mathcal{P})$. It is well known that stable models are minimal models (i.e., $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$). Furthermore, minimal and stable model semantics coincide for positive programs (i.e., $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$). A positive normal program has a unique minimal model.

An argument $q[i]$ in $arg(\mathcal{P})$ is said to be *limited* iff for any finite set of database facts D and for every stable model M of $\mathcal{P} \cup D$, the set $\{t_i \mid q(t_1, ..., t_i, ..., t_n) \in M\}$ is finite.

3 Termination Analysis Tools

In this section, we describe different tools that will be used in the paper to analyze logic programs with function symbols.

Activation graph. The *activation graph* of a program has been introduced in [Greco *et al.*, 2012] to define the class $\mathcal{SP}$ of *safe* programs. In a nutshell, the nodes of the graph are the rules of the program and there is an edge from rule r_1 to rule r_2 iff r_1 "activates" r_2. Activation of rules is defined as follows. Let $\mathcal{P}$ be a program and r_1, r_2 be (not necessarily distinct) rules of $\mathcal{P}$. We say that r_1 *activates* r_2 iff there exist two ground rules $r'_1 \in ground(r_1)$, $r'_2 \in ground(r_2)$ and a set of ground atoms D such that *(i)* $D \not\models r'_1$, *(ii)* $D \models r'_2$, and *(iii)* $D \cup head(r'_1) \not\models r'_2$. This intuitively means that if D does not satisfy r'_1, D satisfies r'_2, and $head(r'_1)$ is added to D to satisfy r'_1, this causes r'_2 not to be satisfied anymore (and then to be "activated").

The *activation graph* of a program $\mathcal{P}$, denoted $\Omega(\mathcal{P})$, is a directed graph whose nodes are the rules of $\mathcal{P}$, and there is an edge (r_i, r_j) in the graph iff r_i activates r_j.

Example 3 Consider the following program $\mathcal{P}_3$:

$$\begin{array}{ll} r_0: & \texttt{nat(X)} \leftarrow \texttt{b(X)}. \\ r_1: & \texttt{next(f(X))} \leftarrow \texttt{nat(X)}. \\ r_2: & \texttt{nat(X)} \leftarrow \texttt{next(f(X))}. \end{array}$$

where b is a base predicate symbol. It is easy to see that r_0 activates r_1, but not vice versa; r_1 activates r_2 and vice versa. The activation graph of P_3 is shown in Figure 1 (left). □

Argument graph. The *argument graph*, used in [Calimeri *et al.*, 2008] to define the class $\mathcal{FD}$ of *finite domain* programs, represents the propagation of values among arguments of a program. The *argument graph* of a program $\mathcal{P}$, denoted $G(\mathcal{P})$, is a directed graph whose nodes are the arguments of $\mathcal{P}$, and there is an edge $(q[j], p[i])$ iff there is a rule $r \in \mathcal{P}$ such that *(i)* an atom $p(t_1, ..., t_n)$ appears in $head(r)$, *(ii)* an atom $q(u_1, ..., u_m)$ appears in $body^+(r)$, and *(iii)* terms t_i and u_j have a common variable. For instance, the argument graph of program $\mathcal{P}_3$ in Example 3 is reported in Figure 1 (right).

A variation of the argument graph, called *propagation graph*, has been proposed in [Greco *et al.*, 2012] to define the class $\mathcal{AP}$ of Γ-*acyclic* programs. The main difference is that in the propagation graph only arguments where complex terms may be introduced or propagated are considered.

Argument ranking. The *argument ranking* of a program has been proposed in [Lierler and Lifschitz, 2009] to define the class $\mathcal{AR}$ of *argument restricted* programs.

For any atom A of the form $p(t_1, ..., t_n)$, A^0 denotes the predicate symbol p, and A^i denotes term t_i, for $1 \leq i \leq n$. The depth $d(X, t)$ of a variable X in a term t that contains X is recursively defined as follows:

$$\begin{array}{l} d(X, X) = 0, \\ d(X, f(t_1, ..., t_m)) = 1 + \max\limits_{i \,:\, t_i \; contains \; X} d(X, t_i). \end{array}$$

An *argument ranking* for a program $\mathcal{P}$ is a partial function ϕ from $arg(\mathcal{P})$ to non-negative integers such that, for every rule r of $\mathcal{P}$, every atom A occurring in the head of r, and every variable X occurring in a term A^i, if $\phi(A^0[i])$ is defined, then $body^+(r)$ contains an atom B such that X occurs in a term B^j, $\phi(B^0[j])$ is defined, and the following condition is satisfied

$$\phi(A^0[i]) - \phi(B^0[j]) \geq d(X, A^i) - d(X, B^j).$$

The set of *restricted arguments* of $\mathcal{P}$ is $AR(\mathcal{P}) = \{p[i] \mid p[i] \in arg(\mathcal{P}) \wedge \exists\phi \; s.t. \; \phi(p[i]) \text{ is defined}\}$. A program $\mathcal{P}$ is said to be *argument restricted* iff $AR(\mathcal{P}) = arg(\mathcal{P})$.

Example 4 Consider again program $\mathcal{P}_3$ of Example 3. An argument ranking ϕ can be defined as follows: $\phi(\texttt{b[1]}) = 0$, $\phi(\texttt{nat[1]}) = 0$, and $\phi(\texttt{next[1]}) = 1$. Thus, $\mathcal{P}_3$ is argument restricted. □

Argument restricted programs strictly include ω-restricted, λ-restricted, and finite domain programs.

4 Bounded Programs

In this section, we introduce a new criterion, more general than the ones in the literature (cf. Section 1), guaranteeing that a program has a finite set of stable models and each of them is finite.

For ease of presentation, we assume that if the same variable occurs in two terms appearing in the head and in the body of a rule, then one term is a subterm of the other and complex terms are of the form $f(t_1, ..., t_m)$ with the t_i's being simple terms. There is no loss of generality in such assumptions as every program can be rewritten into an equivalent one satisfying such conditions (e.g., a rule of the form $\texttt{p(f(h(X)))} \leftarrow \texttt{q(g(X))}$ can be rewritten into the three rules $\texttt{p(f(X))} \leftarrow \texttt{p}'\texttt{(X)}$, $\texttt{p}'\texttt{(h(X))} \leftarrow \texttt{p}''\texttt{(X)}$, and $\texttt{p}''\texttt{(X)} \leftarrow \texttt{q(g(X))}$).

We start by introducing a new graph derived from the argument graph by labelling edges with additional information.

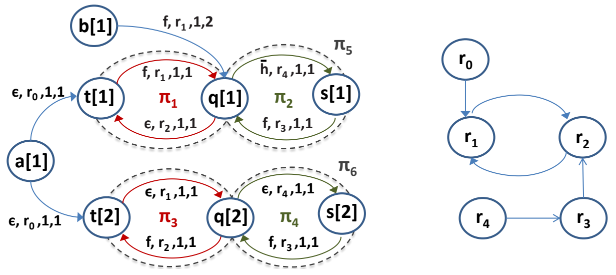

Figure 2: Labelled argument and activation graphs of $\mathcal{P}_5$.

Definition 1 (*Labelled argument graph*) The *labelled argument graph* of a program $\mathcal{P}$, denoted $\mathcal{G}_L(\mathcal{P})$, is a directed graph whose set of nodes is $arg(\mathcal{P})$ and the set of labelled edges is defined as follows. For each pair of nodes $p[i], q[j] \in arg(\mathcal{P})$ and for every rule $r \in \mathcal{P}$ such that *(i)* an atom $p(t_1, ..., t_n)$ appears in $head(r)$, *(ii)* an atom $q(u_1, ..., u_m)$ appears in $body^+(r)$, *(iii)* terms t_i and u_j have a common variable X, there is an edge $(q[j], p[i], \langle \alpha, r, h, k \rangle)$, where h and k are natural numbers denoting the positions of $p(t_1, ..., t_n)$ in $head(r)$ and $q(u_1, ..., u_m)$ in $body^+(r)$, respectively,[3] and

- $\alpha = \epsilon$ if $u_j = t_i$;
- $\alpha = f$ if $u_j = X$ and $t_i = f(..., X, ...)$;
- $\alpha = \overline{f}$ if $u_j = f(..., X, ...)$ and $t_i = X$. □

Example 5 Consider the following program $\mathcal{P}_5$

$$
\begin{array}{lll}
r_0: & \mathtt{t(X,X)} & \leftarrow \mathtt{a(X)}. \\
r_1: & \mathtt{q(f(X),Y)} & \leftarrow \mathtt{t(X,Y),b(X)}. \\
r_2: & \mathtt{t(X,f(Y))} & \leftarrow \mathtt{q(X,Y)}. \\
r_3: & \mathtt{q(f(X),f(Y))} & \leftarrow \mathtt{s(X,Y)}. \\
r_4: & \mathtt{s(X,Y)} & \leftarrow \mathtt{q(h(X),Y)}.
\end{array}
$$

where a and b are base predicate symbols. The labelled argument and activation graphs of $\mathcal{P}_5$ are depicted in Figure 2. For instance, the edge $(\mathtt{t}[1], \mathtt{q}[1], \langle \mathtt{f}, r_1, 1, 1 \rangle)$ in $\mathcal{G}_L(\mathcal{P}_5)$ says that the first atom in the positive body of r_1 is a t-atom, its first term is a variable, say X, and the first atom in the head of r_1 is a q-atom whose first term is of the form $\mathtt{f}(..., \mathtt{X}, ...)$. We will show how this kind of information can be profitably used to analyze programs (e.g., to keep track of how complex terms are generated from argument to argument). □

A *path* ρ from a_1 to b_m in a labelled argument graph $\mathcal{G}_L(\mathcal{P})$ is a non-empty sequence $(a_1, b_1, \langle \alpha_1, r_1, h_1, k_1 \rangle), \ldots, (a_m, b_m, \langle \alpha_m, r_m, h_m, k_m \rangle)$ of labelled edges of $\mathcal{G}_L(\mathcal{P})$ s.t. $b_i = a_{i+1}$ for all $1 \leq i < m$; if the first and last nodes coincide (i.e., $a_1 = b_m$), then ρ is called a *cyclic path*. We define $\lambda_1(\rho) = \alpha_1 \ldots \alpha_m$, denoting the sequence of ϵ and function symbols labelling the edges of ρ, $\lambda_2(\rho) = r_1, ..., r_m$, denoting the sequence of rules labelling the edges of ρ, and $\lambda_3(\rho) = \langle r_1, h_1, k_1 \rangle \ldots \langle r_m, h_m, k_m \rangle$, denoting the sequence of triples ⟨rule, head atom identifier, body atom identifier⟩ labelling the edges of ρ. In the case where the indication of the start edge is not relevant, we will call a cyclic path a *cycle*. Given a cycle π consisting of n labelled edges $e_1, ..., e_n$, we can derive n different cyclic paths starting from each of the e_i's—we use $\tau(\pi)$ to denote the set of such cyclic paths. As an example, if π is a cycle consisting of labelled edges e_1, e_2, e_3, then $\tau(\pi) = \{(e_1, e_2, e_3), (e_2, e_3, e_1), (e_3, e_2, e_1)\}$. Given two cycles π_1 and π_2, we write $\pi_1 \approx \pi_2$ iff $\exists \rho_1 \in \tau(\pi_1)$ and $\exists \rho_2 \in \tau(\pi_2)$ such that $\lambda_3(\rho_1) = \lambda_3(\rho_2)$. A cycle is *basic* if it does not contain two occurrences of the same edge.

[3]We assume that literals in the head (resp. body) are ordered with the first one being associated with 1, the second one with 2, etc.

We say that a node $p[i]$ of $\mathcal{G}_L(\mathcal{P})$ *depends on* a node $q[j]$ of $\mathcal{G}_L(\mathcal{P})$ iff there is a path from $q[j]$ to $p[i]$ in $\mathcal{G}_L(\mathcal{P})$. Moreover, we say that $p[i]$ *depends on* a cycle π iff it depends on a node $q[j]$ appearing in π. Clearly, nodes belonging to a cycle π depend on π. We say that $\lambda_2(\rho) = r_1, ..., r_m$ denotes a cyclic path in the activation graph $\Omega(\mathcal{P})$ iff $(r_1, r_2), ..., (r_{m-1}, r_m), (r_m, r_1)$ are edges of $\Omega(\mathcal{P})$.

Definition 2 (*Active cycle*) Given a program $\mathcal{P}$, we say that a cycle π in $\mathcal{G}_L(\mathcal{P})$ is *active* iff $\exists \rho \in \tau(\pi)$ such that $\lambda_2(\rho)$ denotes a cyclic path in the activation graph $\Omega(\mathcal{P})$. □

Thus, checking if a cycle in the labelled argument graph is active requires to look at the activation graph. Here the basic idea is to check if the propagation of terms along a cycle of the labelled argument graph can really take place based on the information reported in the activation graph. We illustrate this aspect in the following example.

Example 6 Consider the labelled argument graph and the activation graph of program $\mathcal{P}_5$ (cf. Figure 2). Cycles π_1 and π_3 in Figure 2 are active as $(r_1, r_2), (r_2, r_1)$ is a cyclic path in $\Omega(\mathcal{P}_5)$. On the contrary, it is easy to check that cycles π_2 and π_4 in Figure 2, are not active. □

In the previous example, the labelled edges of the non-active cycle π_4 say that a complex term with function symbol f might be generated from s[2] to q[2] (using rule r_3), be propagated from q[2] to s[2] (using rule r_4), and so on so forth possibly causing the generation of complex terms of unbounded size. However, in order for this to happen, rules r_3 and r_4 should activate each other, which is not the case from an analysis of the activation graph. Thus, we can conclude that the generation of unbounded terms cannot really happen.

On the other hand, active cycles might lead to the generation of terms of unbounded size. To establish whether this can really be the case, we perform a deeper analysis of the cycles in the labelled argument graph. Specifically, we use grammars to analyze edge labels to get a better understanding of what terms can be propagated among arguments. We introduce two distinct languages which allows us to distinguish between "growing" paths, which could give rise to terms of infinite size, and "balanced" paths, where propagated terms do not grow (see Definition 4).

Definition 3 (*Grammars* $\Gamma_{\mathcal{P}}$ *and* $\Gamma'_{\mathcal{P}}$) Let $\mathcal{P}$ be a program and $F_{\mathcal{P}}$ the set of function symbols occurring in $\mathcal{P}$. The *grammars* $\Gamma_{\mathcal{P}}$ and $\Gamma'_{\mathcal{P}}$ are 4-tuples (N, T, R, S) and (N', T, R', S), respectively, where $N = \{S, S_1, S_2\}$ and $N' = \{S\}$ are the sets of nonterminal symbols, $T = \{f \mid f \in F_{\mathcal{P}}\} \cup \{\overline{f} \mid f \in F_{\mathcal{P}}\}$ is the set of terminal symbols, S is the start symbol, R is the set of production rules:

- $S \ \to S_1\, f_i\, S_2, \qquad\qquad \forall f_i \in F_{\mathcal{P}};$
- $S_1 \to f_i\, S_1\, \overline{f}_i\, S_1 \mid \epsilon, \qquad \forall f_i \in F_{\mathcal{P}};$
- $S_2 \to (S_1 \mid f_i)\, S_2 \mid \epsilon, \qquad \forall f_i \in F_{\mathcal{P}}.$

and R' is the set of production rules:

- $S \to f_i\, S\, \overline{f}_i\, S \mid \epsilon, \qquad \forall f_i \in F_{\mathcal{P}}.$

The languages $\mathcal{L}(\Gamma_{\mathcal{P}})$ and $\mathcal{L}(\Gamma'_{\mathcal{P}})$ are the sets of strings generated by $\Gamma_{\mathcal{P}}$ and $\Gamma'_{\mathcal{P}}$, respectively. □

Notice that $\mathcal{L}(\Gamma_{\mathcal{P}}) \cap \mathcal{L}(\Gamma'_{\mathcal{P}}) = \emptyset$. Grammar $\Gamma_{\mathcal{P}}$ has been introduced in [Greco *et al.*, 2012] to analyze cycles of the propagation graph. Here we propose a more detailed analysis of the relationships among arguments by considering also $\Gamma'_{\mathcal{P}}$. Intuitively, strings in $\mathcal{L}(\Gamma_{\mathcal{P}})$ describe growing sequences of functions symbols used to compose and decompose complex terms, so that starting from a given term we obtain a larger term. On the other hand, strings in $\mathcal{L}(\Gamma'_{\mathcal{P}})$ describe "balanced" sequences of functions symbols used to compose and decompose complex terms, so that starting from a given term we obtain the same term.

Definition 4 (*Classification of cycles*) Given a program $\mathcal{P}$ and a cycle π in $\mathcal{G}_L(\mathcal{P})$, we say that π is

- *growing* if there is $\rho \in \tau(\pi)$ s.t. $\lambda_1(\rho) \in \mathcal{L}(\Gamma_{\mathcal{P}})$,
- *balanced* if there is $\rho \in \tau(\pi)$ s.t. $\lambda_1(\rho) \in \mathcal{L}(\Gamma'_{\mathcal{P}})$,
- *failing* otherwise. □

Consider the labelled argument graph in Figure 2. Cycles π_1, π_3, and π_4 are growing, whereas cycle π_2 is failing. Observe that, in general, a failing cycle is not active, but the vice versa is not true. In fact, cycle π_4 from Figure 2 is not active even if it is not failing.

The tools introduced so far are used to define the *binding* operator $\Psi_{\mathcal{P}}$ (Definition 5 below). If operator $\Psi_{\mathcal{P}}$ is applied to a set of limited arguments A, then it gives a set $\Psi_{\mathcal{P}}(A) \supseteq A$ of arguments which are guaranteed to be limited too. The idea of our termination criterion is to compute the fixpoint of $\Psi_{\mathcal{P}}$ starting from a set of limited arguments so as to get a set of limited arguments, which can be used as an underestimation of the limited arguments of the program. If the fixpoint computation gives us all arguments of $\mathcal{P}$, then $\mathcal{P}$ is *bounded*.

Definition 5 ($\Psi_{\mathcal{P}}$ *operator*) Let $\mathcal{P}$ be a program and $A \subseteq arg(\mathcal{P})$. We define $\Psi_{\mathcal{P}}(A)$ as the set of arguments $q[k]$ of $\mathcal{P}$ such that for each basic cycle π in $\mathcal{G}_L(\mathcal{P})$ on which $q[k]$ depends, at least one of the following conditions holds:

1. π is not active or is not growing;
2. π contains an edge $(s[j], p[i], \langle f, r, l_1, l_2\rangle)$ and, letting $p(t_1, ..., t_n)$ be the l_1-th atom in the head of r, for every variable X in t_i, there is an atom $b(u_1, ..., u_m)$ in $body^+(r)$ s.t. X appears in a term u_h and $b[h] \in A$;
3. there is a basic cycle π' in $\mathcal{G}_L(\mathcal{P})$ s.t. $\pi' \approx \pi$, π' is not balanced, and π' passes only through arguments in A. □

Observe that base arguments belong to $\Psi_{\mathcal{P}}(A)$ for any set of arguments A, because they do not depend on cycles.

Proposition 1 *Let $\mathcal{P}$ be a program and A a set of limited arguments of $\mathcal{P}$. Then, all arguments in $\Psi_{\mathcal{P}}(A)$ are limited.*

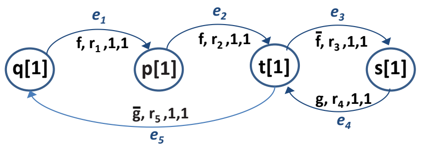

Figure 3: An example of a non-simple basic cycle.

Example 7 Consider program $\mathcal{P}_5$ of Example 5, whose labelled argument and activation graphs are reported in Figure 2. Notice that basic cycles π_1 and π_3 are growing and active, π_2 is failing and non-active, π_4 is growing and non-active. The basic cycle π_5 composed by π_1 and π_2 (denoted by a dashed line) is failing and non-active, and the basic cycle π_6 composed by π_3 and π_4 (denoted by a dashed line) is growing and non-active. Furthermore, base arguments $\texttt{a}[1]$ and $\texttt{b}[1]$ do not depend on any cycle; $\texttt{t}[1]$, $\texttt{q}[1]$, $\texttt{s}[1]$ depend on π_1, π_2, and π_5; $\texttt{t}[2]$, $\texttt{q}[2]$, $\texttt{s}[2]$ depend on π_3, π_4, and π_6. By iteratively applying operator $\Psi_{\mathcal{P}_5}$ starting from $\emptyset$ we have:
(i) $A_1 = \Psi_{\mathcal{P}_5}(\emptyset) = \{\texttt{a}[1], \texttt{b}[1]\}$;
(ii) $A_2 = \Psi_{\mathcal{P}_5}(A_1) = A_1 \cup \{\texttt{t}[1], \texttt{q}[1], \texttt{s}[1]\}$ as Condition 2 of Definition 5 applies to π_1, and Condition 1 applies to π_2 and π_5;
(iii) $A_3 = \Psi_{\mathcal{P}_5}(A_2) = A_2 \cup \{\texttt{t}[2], \texttt{q}[2], \texttt{s}[2]\}$ as Condition 1 applies to π_4 and π_6, and Condition 3 applies to π_3 (in fact, $\pi_1 \approx \pi_3$, π_1 is not balanced and goes only through nodes in A_3).
Therefore, we derive that all arguments are limited. □

Notice that in order to guarantee that arguments in $\Psi_{\mathcal{P}}(A)$ are limited, it does not suffice to consider simple cycles only (i.e., cycles going through each node only once), but basic cycles have to be considered. As an example, Figure 3 shows a labelled argument graph with two failing simple cycles composed by edges e_1, e_2, e_5 and e_3, e_4, respectively. However, the basic cycle consisting of edges e_1, e_2, e_3, e_4, e_5 is growing and the generation of infinite terms can take place.

Proposition 2 *Given a program $\mathcal{P}$ and a set A of limited arguments of $\mathcal{P}$, then:*

1. *$\Psi^i_{\mathcal{P}}(A) \subseteq \Psi^{i+1}_{\mathcal{P}}(A)$ for $i \geq 0$, and*
2. *there is a finite n such that $\Psi^n_{\mathcal{P}}(A) = \Psi^{\infty}_{\mathcal{P}}(A)$,*

where $\Psi^0_{\mathcal{P}}(A) = A$ and $\Psi^{i+1}_{\mathcal{P}}(A) = \Psi_{\mathcal{P}}(\Psi^i_{\mathcal{P}}(A))$, for $i \geq 0$.

Observe that $\Psi^i_{\mathcal{P}}(A) \subseteq \Psi^{i+1}_{\mathcal{P}}(A)$, for $i > 0$, for every set of limited arguments A, but the relationship $A \subseteq \Psi_{\mathcal{P}}(A)$ does not always hold if A is an arbitrary non-empty set of arguments of $\mathcal{P}$. Below we report some examples illustrating the computation of $\Psi^{\infty}_{\mathcal{P}}(\emptyset)$ and the conditions of Definition 5.

Example 8 Consider program $\mathcal{P}_1$ of Example 1. To comply with the conditions mentioned at the beginning of this section, $\mathcal{P}_1$ is rewritten into the following program $\mathcal{P}_8$:

$$\begin{array}{ll} r_0: & \texttt{count(lc(a,lc(b,lc(c,nil))),0).} \\ r_1: & \texttt{f-count(X,L,I)} \leftarrow \texttt{count(lc(X,L),I).} \\ r_2: & \texttt{count(L,s(I,1))} \leftarrow \texttt{f-count(X,L,I).} \end{array}$$

Here function symbol `lc` denotes the list constructor operator "|", function symbol `s` denotes the sum operator, and `nil`

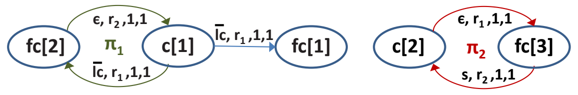

Figure 4: Labelled argument graph of $\mathcal{P}_8$.

denotes the empty list "[ ]". The labelled argument graph is shown in Figure 4, where `c` and `fc` stand for `count` and `f-count`, respectively. The activation graph is not shown, but it is easy to see that r_1 activates r_2 and vice versa; thus, both cycles π_1 and π_2 are active. Furthermore, π_1 is failing and π_2 is growing. Notice also that $\mathtt{c}[1]$, $\mathtt{fc}[1]$, and $\mathtt{fc}[2]$ depend on π_1; $\mathtt{c}[2]$ and $\mathtt{fc}[3]$ depend on π_2.

As π_1 is failing, then $A_1 = \Psi_{\mathcal{P}_8}(\emptyset) = \{\mathtt{c}[1], \mathtt{fc}[1], \mathtt{fc}[2]\}$ by Condition 1. Then, $A_2 = \Psi_{\mathcal{P}_8}(A_1) = A_1 \cup \{\mathtt{c}[2], \mathtt{fc}[3]\}$ by Condition 3 because $\pi_1 \approx \pi_2$, π_1 is not balanced and goes through arguments $\mathtt{c}[1]$ and $\mathtt{fc}[2]$ which both belong to A_1. It is worth noting that current termination criteria are not able to realize this kind of dependency between π_1 and π_2. As $\Psi_{\mathcal{P}_8}(A_2)$ gives A_2, the fixpoint is reached with all arguments of $\mathcal{P}_8$ belonging to $\Psi^{\infty}_{\mathcal{P}_8}(\emptyset)$. By Proposition 1, all arguments are limited, thus $\mathcal{P}_8$ has a finite minimal model. $\square$

Example 9 Consider program $\mathcal{P}_9$ below derived from program $\mathcal{P}'_2$ of Example 2 by replacing rule r_3 of $\mathcal{P}'_2$ with the two rules r_3 and r_4 below (to satisfy the conditions stated at the beginning of this section):

r_0: `magic_append([a, b], [c, d]).`
r_1: `magic_append(L1, L2)` $\leftarrow$ `magic_append([X|L1], L2).`
r_2: `append([ ], L, L)` $\leftarrow$ `magic_append([ ], L).`
r_3: `f-magic_append(X,L1,L2)` $\leftarrow$ `magic_append([X|L1],L2).`
r_4: `append([X|L1], L2, [X|L3])` $\leftarrow$ `f-magic_append(X,L1,L2),`
`append(L1, L2, L3).`

Figure 5 shows the labelled argument graph of $\mathcal{P}_9$, where `ma`, `a`, and `fma` stand for `magic_append`, `append` and `f-magic_append`, respectively. As π_1, π_2, and π_3 are not growing, then $\Psi^1_{\mathcal{P}_9}(\emptyset) = \{\mathtt{ma}[1], \mathtt{ma}[2], \mathtt{fma}[1], \mathtt{fma}[2], \mathtt{fma}[3], \mathtt{a}[2]\}$ by Condition 1. Since $\mathtt{fma}[1]$ and $\mathtt{fma}[2]$ belong to $\Psi^1_{\mathcal{P}_9}(\emptyset)$, then $\mathtt{a}[1] \in \Psi^2_{\mathcal{P}_9}(\emptyset)$ by Condition 2. As $\pi_5 \approx \pi_4$, π_5 is not balanced and goes only through $\mathtt{a}[1]$ which is in $\Psi^2_{\mathcal{P}_9}(\emptyset)$, then $\mathtt{a}[3]$ belongs to $\Psi^3_{\mathcal{P}_9}(\emptyset)$ by Condition 3. Since all arguments are in $\Psi^3_{\mathcal{P}_9}(\emptyset)$, then $\Psi^3_{\mathcal{P}_9}(\emptyset) = \Psi^{\infty}_{\mathcal{P}_9}(\emptyset)$ and the program has a finite minimal model. $\square$

We are now ready to define the class of *bounded programs*. Given a program $\mathcal{P}$, we start with the set of restricted arguments $AR(\mathcal{P})$ (see Section 3), which gives a good and efficiently computable approximation of the set of limited arguments; then, we iteratively apply operator $\Psi_{\mathcal{P}}$ trying to infer more limited arguments. If, eventually, all arguments in $arg(\mathcal{P})$ are determined as limited, then $\mathcal{P}$ is bounded. We do not start from the empty set as there are programs $\mathcal{P}$ s.t. $\Psi^{\infty}_{\mathcal{P}}(\emptyset) \subsetneq \Psi^{\infty}_{\mathcal{P}}(AR(\mathcal{P}))$. Observe also that the relation $A \subseteq \Psi_{\mathcal{P}}(A)$ holds for $A = AR(\mathcal{P})$.

Definition 6 (*Bounded programs*) The set of *bounded* arguments of a program $\mathcal{P}$ is $\Psi^{\infty}_{\mathcal{P}}(AR(\mathcal{P}))$. We say that $\mathcal{P}$ is *bounded* iff all its arguments are bounded. The set of all bounded programs is denoted as $\mathcal{BP}$. $\square$

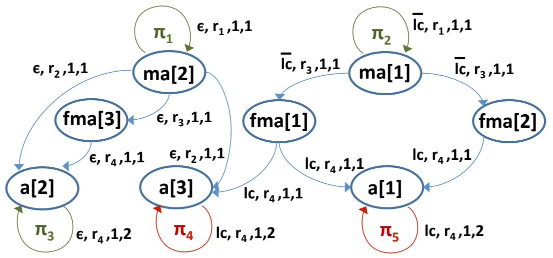

Figure 5: Labelled argument graph of $\mathcal{P}_9$.

Theorem 1 *[**Correctness**] Given a bounded program $\mathcal{P}$, then $\mathcal{SM}(\mathcal{P} \cup D)$ is finite and every $M \in \mathcal{SM}(\mathcal{P} \cup D)$ is finite for any finite set of database facts D.*

Theorem 2 *[**Complexity**] The worst-case time complexity of checking whether a program $\mathcal{P}$ belongs to the class $\mathcal{BP}$ is $O(k + n^3 + n\,m\,l\,(s\,b + m))$, where*

- *k is the cost of building $\Omega(\mathcal{P})$ and $\mathcal{G}_L(\mathcal{P})$,*
- *n is the number of arguments of $\mathcal{P}$,*
- *m is the number of basic cycles in $\mathcal{G}_L(\mathcal{P})$,*
- *l is the maximal length of a basic cycle,*
- *s is the maximal number of distinct variables occurring in a term in the head of a rule,*
- *b is the maximal size of a rule body (considering the number of body atoms and arities of predicate and function symbols).*

In the theorem above, $O(k)$ is polynomial in the size of $\mathcal{P}$; $O(n^3)$ is the complexity of computing the set of restricted arguments. As the number of basic cycles m can be exponential in the size of the program, the overall complexity is exponential. However, in many cases m is "small" (polynomial in the size of $\mathcal{P}$) and the overall complexity becomes polynomial.

The following theorem says that $\mathcal{BP}$ strictly includes $\mathcal{AR}$ and $\mathcal{AP}$ (resp., argument restricted and Γ-acyclic programs), the more general decidable classes in the literature, and is included in the class $\mathcal{FG}$ of finitely ground programs.

Theorem 3 *$\mathcal{T} \subsetneq \mathcal{BP} \subsetneq \mathcal{FG}$, for $\mathcal{T} \in \{\mathcal{AR}, \mathcal{AP}\}$.*

Definition 7 (*Linear programs*) A program $\mathcal{P}$ is *linear* iff each node in $\mathcal{G}_L(\mathcal{P})$ appears in at most one basic cycle. $\square$

For instance, $\mathcal{P}_8$ and $\mathcal{P}_9$ of Examples 8 and 9 are linear, while program $\mathcal{P}_5$ of Example 5 is not linear.

Theorem 4 *Checking if a linear program $\mathcal{P}$ belongs to the class $\mathcal{BP}$ can be solved in polynomial time in the size of $\mathcal{P}$.*

5 Conclusions

Enhancing answer set solvers by providing support for function symbols has attracted a great deal of interest in recent years. As the introduction of function symbols make common inference tasks undecidable, different classes of programs imposing restrictions on the use of function symbols and guaranteeing decidability of common inference tasks have been proposed. In this paper, we have presented a more general class that strictly includes all classes in the literature, along with different results on its correctness and complexity.

References

[Alviano *et al.*, 2010] Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive asp with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10(4-6):497–512, 2010.

[Arts and Giesl, 2000] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[Baselice *et al.*, 2009] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. *Theory and Practice of Logic Programming*, 9(2):213–238, 2009.

[Beeri and Ramakrishnan, 1991] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(1/2/3&4):255–299, 1991.

[Bonatti, 2004] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[Bruynooghe *et al.*, 2007] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[Calimeri *et al.*, 2008] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp: Theory and implementation. In *ICLP*, pages 407–424, 2008.

[Codish *et al.*, 2005] Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Testing for termination with monotonicity constraints. In *ICLP*, pages 326–340, 2005.

[Endrullis *et al.*, 2008] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[Ferreira and Zantema, 1996] Maria C. F. Ferreira and Hans Zantema. Total termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 7(2):133–162, 1996.

[Gebser *et al.*, 2007] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[Greco *et al.*, 2012] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. On the termination of logic programs with function symbols. In *ICLP (Technical Communications)*, pages 323–333, 2012.

[Lierler and Lifschitz, 2009] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *ICLP*, pages 489–493, 2009.

[Marchiori, 1996] Massimo Marchiori. Proving existential termination of normal logic programs. In *Algebraic Methodology and Software Technology*, pages 375–390, 1996.

[Nguyen *et al.*, 2007] Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. Termination analysis of logic programs based on dependency graphs. In *LOPSTR*, pages 8–22, 2007.

[Nishida and Vidal, 2010] Naoki Nishida and Germán Vidal. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):177–225, 2010.

[Ohlebusch, 2001] Enno Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1/2):73–116, 2001.

[Schneider-Kamp *et al.*, 2009a] Peter Schneider-Kamp, Jürgen Giesl, and Manh Thang Nguyen. The dependency triple framework for termination of logic programs. In *LOPSTR*, pages 37–51, 2009.

[Schneider-Kamp *et al.*, 2009b] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), 2009.

[Schneider-Kamp *et al.*, 2010] Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, and René Thiemann. Automated termination analysis for logic programs with cut. *Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010.

[Schreye and Decorte, 1994] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[Serebrenik and De Schreye, 2005] Alexander Serebrenik and Danny De Schreye. On termination of meta-programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[Sternagel and Middeldorp, 2008] Christian Sternagel and Aart Middeldorp. Root-labeling. In *Rewriting Techniques and Applications*, pages 336–350, 2008.

[Syrjänen, 2001] Tommi Syrjänen. Omega-restricted logic programs. In *Logic Programming and Nonmonotonic Reasoning*, pages 267–279, 2001.

[Voets and Schreye, 2011] Dean Voets and Danny De Schreye. Non-termination analysis of logic programs with integer arithmetics. *Theory and Practice of Logic Programming*, 11(4-5):521–536, 2011.

[Zantema, 1995] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1/2):89–105, 1995.