Checking Termination of Logic Programs with Function Symbols through Linear Constraints

Checking Termination of Logic Programs with Function Symbols Through Linear Constraints

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna
{calautti,greco,molinaro,trubitsyna}@dimes.unical.it

DIMES, Università della Calabria, 87036 Rende (CS), Italy

Abstract. Enriching answer set programming with function symbols makes modeling easier, increases the expressive power, and allows us to deal with infinite domains. However, this comes at a cost: common inference tasks become undecidable. To cope with this issue, recent research has focused on finding trade-offs between expressivity and decidability by identifying classes of logic programs that impose limitations on the use of function symbols but guarantee decidability of common inference tasks. Despite the significant body of work in this area, current approaches do not include many simple practical programs whose evaluation terminates. In this paper, we present the novel class of *rule-bounded programs*. While current techniques perform a limited analysis of how terms are propagated from an individual argument to another, our technique is able to perform a more global analysis, thereby overcoming several limitations of current approaches. We also present a further class of *cycle-bounded programs* where groups of rules are analyzed together. We show different results on the correctness and the expressivity of the proposed techniques.

Keywords: Logic programming with function symbols, bottom-up evaluation, program evaluation termination, stable models

1 Introduction

Enriching answer set programming with function symbols has recently seen a surge in interest. Function symbols make modeling easier, increase the expressive power, and allow us to deal with infinite domains. At the same time, this comes at a cost: common inference tasks (e.g., cautious and brave reasoning) become undecidable.

Recent research has focused on identifying classes of logic programs that impose some limitations on the use of function symbols but guarantee decidability of common inference tasks. Efforts in this direction are the class of *finitely-ground* programs [7] and the more general class of *bounded term-size* programs [26]. Finitely-ground programs have a finite number of stable models, each of finite size, whereas bounded term-size (normal) programs have a finite well-founded model. Unfortunately, checking if a logic program is bounded term-size or even finitely-ground is semi-decidable.

Considering the stable model semantics, decidable subclasses of finitely-ground programs have been proposed. These include the classes of *ω-restricted programs* [33], *λ-restricted programs* [14], *finite domain programs* [7], *argument-restricted programs* [21], *safe programs* [19], *Γ-acyclic programs* [19], *mapping-restricted programs* [6], and *bounded programs* [17]. The above techniques, that we call *termination criteria*, provide (decidable) sufficient conditions for a program to be finitely-ground.

Despite the significant body of work in this area, there are still many simple practical programs which are finitely-ground but are not detected by any of the current termination criteria. Below is an example.

Example 1. Consider the following program $\mathcal{P}_1$ implementing the bubble sort algorithm:

$$
\begin{aligned}
r_0 &: \ \mathtt{bub}(\mathtt{L}, [\,], [\,]) \leftarrow \mathtt{input}(\mathtt{L}).\\
r_1 &: \ \mathtt{bub}([\mathtt{Y}|\mathtt{T}], [\mathtt{X}|\mathtt{Cur}], \mathtt{Sol}) \leftarrow \mathtt{bub}([\mathtt{X}|[\mathtt{Y}|\mathtt{T}]], \mathtt{Cur}, \mathtt{Sol}), \mathtt{X} \leq \mathtt{Y}.\\
r_2 &: \ \mathtt{bub}([\mathtt{X}|\mathtt{T}], [\mathtt{Y}|\mathtt{Cur}], \mathtt{Sol}) \leftarrow \mathtt{bub}([\mathtt{X}|[\mathtt{Y}|\mathtt{T}]], \mathtt{Cur}, \mathtt{Sol}), \mathtt{Y} < \mathtt{X}.\\
r_3 &: \ \mathtt{bub}(\mathtt{Cur}, [\,], [\mathtt{X}|\mathtt{Sol}]) \leftarrow \mathtt{bub}([\mathtt{X}|[\,]], \mathtt{Cur}, \mathtt{Sol}).
\end{aligned}
$$

Here `input` is a base predicate symbol whose extension is a fact containing the list we would like to sort. The bottom-up evaluation of this program always terminates for any input list. The ordered list `Sol` can be obtained from the atom $\mathtt{bub}([\,], [\,], \mathtt{Sol})$ in the program's minimal model. □

None of the termination criteria in the literature is able to realize that $\mathcal{P}_1$ is finitely-ground. One problem with them is that when they analyze how terms are propagated from the body to the head of rules, they look at arguments *individually*. For instance, in rule r_1 above, the simple fact that the second argument of `bub` has a size in the head greater than the one in the body prevents several techniques from realizing termination of the bottom-up evaluation of $\mathcal{P}_1$. More general classes such as mapping-restricted and bounded programs are able to do a more complex (yet limited) analysis of how some groups of arguments affect each other. Still, all current termination criteria are not able to realize that in every rule of $\mathcal{P}_1$ the *overall* size of the terms in the head does not increase w.r.t. the *overall* size of the terms in the body. One of the novelties of the technique proposed in this paper is the capability of doing this kind of analysis, thereby identifying finitely-ground programs that none of the current techniques include.

The technique proposed in this paper easily realizes that the bottom-up evaluation of $\mathcal{P}_1$ always terminates for any input list. In fact, our technique can understand that, in every rule, the overall size of the terms in the body does not increase during their propagation to the head, as there is only a simple redistribution of terms. Many practical programs dealing with lists and tree-like structures satisfy this property—below are two examples. However, our technique is not limited only to this kind of programs.

Example 2. Consider the following program $\mathcal{P}_2$ performing a depth-first traversal of an input tree:

$$r_0: \texttt{visit}(\texttt{Tree}, [\,], [\,]) \leftarrow \texttt{input}(\texttt{Tree}).$$
$$r_1: \texttt{visit}(\texttt{Left}, [\texttt{Root}|\texttt{Visited}], [\texttt{Right}|\texttt{ToVisit}]) \leftarrow \texttt{visit}(\texttt{tree}(\texttt{Root}, \texttt{Left}, \texttt{Right}), \texttt{Visited}, \texttt{ToVisit}).$$
$$r_2: \texttt{visit}(\texttt{Next}, \texttt{Visited}, \texttt{ToVisit}) \leftarrow \texttt{visit}(\texttt{null}, \texttt{Visited}, [\texttt{Next}|\texttt{ToVisit}]).$$

Here `input` is a base predicate symbol whose extension contains a tree-like structure represented by means of the ternary function symbol `tree`. The program visits the nodes of the tree and puts them in a list following a depth-first search. The list `L` of visited elements can be obtained from the atom $\texttt{visit}(\texttt{null}, \texttt{L}, [\,])$ in the program's minimal model. For instance, if the input tree is

$$\texttt{input}(\texttt{tree}(\texttt{a}, \texttt{tree}(\texttt{c}, \texttt{null}, \texttt{tree}(\texttt{d}, \texttt{null}, \texttt{null})), \texttt{tree}(\texttt{b}, \texttt{null}, \texttt{null}))).$$

the program produces the list $[\texttt{b}, \texttt{d}, \texttt{c}, \texttt{a}]$ containing the nodes of the tree in opposite order w.r.t. the traversal. □

Also in the case above, even if the program evaluation terminates for every input tree, none of the currently known techniques is able to detect it, while the technique proposed in this paper does.

Example 3. Consider the following program $\mathcal{P}_3$ computing the concatenation of two lists:

$$r_0: \texttt{reverse}(\texttt{L}_1, [\,]) \leftarrow \texttt{input1}(\texttt{L}_1).$$
$$r_1: \texttt{reverse}(\texttt{L}_1, [\texttt{X}|\texttt{L}_2]) \leftarrow \texttt{reverse}([\texttt{X}|\texttt{L}_1], \texttt{L}_2).$$
$$r_2: \texttt{append}(\texttt{L}_1, \texttt{L}_2) \leftarrow \texttt{reverse}([\,], \texttt{L}_1),\ \texttt{input2}(\texttt{L}_2).$$
$$r_3: \texttt{append}(\texttt{L}_1, [\texttt{X}|\texttt{L}_2]) \leftarrow \texttt{append}([\texttt{X}|\texttt{L}_1], \texttt{L}_2).$$

Here `input1` and `input2` are base predicate symbols whose extensions contain two lists $\texttt{L}_1$ and $\texttt{L}_2$ to be concatenated. The result list `L` can be retrieved from the atom $\texttt{append}([\,], \texttt{L})$ in the minimal model of $\mathcal{P}_3$. It is easy to see that the bottom-up evaluation of the program always terminates. □

Contribution. We propose novel techniques for checking if a logic program is finitely-ground. Our techniques overcome several limitations of current approaches being able to perform a more global analysis of how terms are propagated from the body to the head of rules. To this end, we use linear constraints to measure and relate the size of head and body atoms. We first introduce the class of *rule-bounded* programs, which looks at individual rules, and then propose the class of *cycle-bounded* programs, which relies on the analysis of groups of rules. We study the relationship between the proposed classes and current termination criteria.

Organization. Section 2 reports preliminaries on logic programs with function symbols. Section 3 introduces the class of rule-bounded programs. Section 4 presents the class of cycle-bounded programs. Related work and conclusions are reported in Sections 5 and 6, respectively.

2 Preliminaries

This section recalls syntax and the stable model semantics of logic programs with function symbols [15,16,13].

Syntax. We assume to have (pairwise disjoint) infinite sets of *constants*, *logical variables*[1], *predicate symbols*, and *function symbols*. Each predicate and function symbol g is associated with an *arity*, denoted $arity(g)$, which is a non-negative integer for predicate symbols and a positive integer for function symbols.

A *term* is either a constant, a logical variable, or an expression of the form $f(t_1, ..., t_m)$, where f is a function symbol of arity m and $t_1, ..., t_m$ are terms.

An *atom* is of the form $p(t_1, ..., t_n)$, where p is a predicate symbol of arity n and $t_1, ..., t_n$ are terms. A *literal* is 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 $body(r)$. With a slight abuse of notation, we sometimes use $body(r)$ (resp. $head(r)$) to also denote the *set* of literals appearing in the body (resp. head) of r. If $m = 1$, then r is *normal*; in this case, $head(r)$ denotes the head atom. 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). We assume that programs are *range restricted*, i.e., for every rule, every logical variable appears in some positive body literal. W.l.o.g., we also assume that different rules do not share logical variables.

A term (resp. atom, literal, rule, program) is *ground* if no logical variables occur in it. A ground normal rule with an empty body is also called a *fact*.

A predicate symbol p is *defined by* a rule r if p appears in the head of r. Predicate symbols are partitioned into two different classes: *base* predicate symbols, which are defined by facts only, and *derived* predicate symbols, which can be defined by any rule. Facts defining base predicate symbols are called *database facts*.[2]

A *substitution* θ is of the form $\{X_1/t_1, ..., X_n/t_n\}$, where $X_1, ..., X_n$ are distinct logical variables and $t_1, ..., t_n$ are terms. The result of applying θ to an atom (or term) A, denoted $A\theta$, is the atom (or term) obtained from A by simultaneously replacing each occurrence of a logical variable X_i in A with t_i if X_i/t_i belongs to θ. Two atoms A_1 and A_2 *unify* if there exists a substitution θ, called a *unifier* of A_1 and A_2, such that $A_1\theta = A_2\theta$. The *composition* of two substitutions $\theta = \{X_1/t_1, ..., X_n/t_n\}$ and $\vartheta = \{Y_1/u_1, ..., Y_m/u_m\}$, denoted $\theta \circ \vartheta$, is the substitution obtained from the set $\{X_1/t_1\vartheta, ..., X_n/t_n\vartheta, Y_1/u_1, ..., Y_m/u_m\}$ by remov-

[1] Variables appearing in logic programs are called "logical variables" and will be denoted by upper-case letters in order to distinguish them from variables appearing in linear constraints, which are called "integer variables" and will be denoted by lower-case letters.

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

ing every $X_i/t_i\vartheta$ such that $X_i = t_i\vartheta$ and every Y_j/u_j such that $Y_j \in \{X_1, ..., X_n\}$. A substitution θ is *more general* than a substitution ϑ if there exists a substitution η such that $\vartheta = \theta \circ \eta$. A unifier θ of A_1 and A_2 is called a *most general unifier* (mgu) of A_1 and A_2 if it is more general than any other unifier of A_1 and A_2 (indeed, the mgu is unique modulo renaming of logical variables).

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 logical 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. A ground rule r is *satisfied* by I, denoted $I \models r$, if there is a ground literal L in $body(r)$ s.t. $value_I(L) = false$ or there is a ground atom A in $head(r)$ s.t. $value_I(A) = true$. Thus, if the body of r is empty, r is satisfied by I if there is an atom A in $head(r)$ s.t. $value_I(A) = true$. 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 $I \in \mathcal{MM}(\mathcal{P}^I)$. 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., $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$), and $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$ for positive programs. A positive normal program has a unique minimal model.

3 Rule-bounded Programs

In this section, we present *rule-bounded programs*, a class of finitely-ground programs for which checking membership in the class is decidable. Their definition relies on a novel technique which uses linear inequalities to measure terms and atoms' sizes and checks if the size of the head of a rule is always bounded by the size of a mutually recursive body atom (we will formally define what "mutually recursive" means in Definition 2 below).

For ease of presentation, we restrict our attention to positive normal programs. However, our technique can be applied to an arbitrary program $\mathcal{P}$ with disjunction in the head and negation in the body by considering a positive normal program $st(\mathcal{P})$ derived from $\mathcal{P}$ as follows. Every rule $A_1 \vee ... \vee A_m \leftarrow body$ in

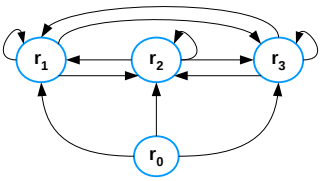

Fig. 1. Firing graph of $\mathcal{P}_1$.

$\mathcal{P}$ is replaced with m positive normal rules of the form $A_i \leftarrow body^+$ $(1 \leq i \leq m)$ where $body^+$ is obtained from $body$ by deleting all negative literals. In fact, as already stated in [19], the minimal model of $st(\mathcal{P})$ contains every stable model of $\mathcal{P}$—whence, finiteness and computability of the minimal model of $st(\mathcal{P})$ implies that $\mathcal{P}$ has a finite number of stable models, each of finite size, which can be computed. In the rest of the paper, a program is understood to be a positive normal program. We start by introducing some preliminary notions.

Definition 1 (Firing graph). *The* firing graph *of a program $\mathcal{P}$, denoted $\Omega(\mathcal{P})$, is a directed graph whose nodes are the rules in $\mathcal{P}$ and such that there is an edge $\langle r, r' \rangle$ if there exist two (not necessarily distinct) rules $r, r' \in \mathcal{P}$ s.t. $head(r)$ and an atom in $body(r')$ unify.* □

Intuitively, an edge $\langle r, r' \rangle$ of $\Omega(\mathcal{P})$ means that rule r may cause rule r' to "fire". The firing graph of program $\mathcal{P}_1$ of Example 1 is depicted in Figure 1. In the definition above, when $r = r'$ we assume that r and r' are two "copies" that do not share any logical variable.

A *strongly connected component* (SCC) of an arbitrary directed graph G is a maximal set $\mathcal{C}$ of nodes of G s.t. every node of $\mathcal{C}$ can be reached from every node of $\mathcal{C}$ (through the edges in G). We say that an SCC $\mathcal{C}$ is *non-trivial* if there exists at least one edge in G between two not necessarily distinct nodes of $\mathcal{C}$. For instance, the firing graph in Figure 1 has two SCCs, $\mathcal{C}_1 = \{r_0\}$ and $\mathcal{C}_2 = \{r_1, r_2, r_3\}$, but only $\mathcal{C}_2$ is non-trivial.

Given a program $\mathcal{P}$ and an SCC $\mathcal{C}$ of $\Omega(\mathcal{P})$, $pred(\mathcal{C})$ denotes the set of predicate symbols defined by the rules in $\mathcal{C}$. We now define when the head atom and a body atom of a rule are mutually recursive.

Definition 2 (Mutually recursive atoms). *Let $\mathcal{P}$ be a program and r a rule in $\mathcal{P}$. The head atom $A = head(r)$ and an atom $B \in body(r)$ are* mutually recursive *if there is a non-trivial SSC $\mathcal{C}$ of $\Omega(\mathcal{P})$ s.t.:*

1. *$\mathcal{C}$ contains r, and*
2. *$\mathcal{C}$ contains a rule r' (not necessarily distinct from r) s.t. $\langle r', r \rangle$ is an edge of $\Omega(\mathcal{P})$ and $head(r')$ unifies with B.* □

In the previous definition, when $r = r'$ we assume that r and r' are two "copies" that do not share any logical variable. Intuitively, the head atom A of a rule r and an atom B in the body of r are mutually recursive when there might

be an actual propagation of terms from B to A (through the application of a sequence of rules). As a very simple example, in the rule $\mathtt{p(f(X))} \leftarrow \mathtt{p(X)}, \mathtt{p(g(X))}$, the first body atom is mutually recursive with the head, while the second one is not as it does not unify with the head atom.

Given a rule r, we use $rbody(r)$ to denote the set of atoms in $body(r)$ which are mutually recursive with $head(r)$. Moreover, we define $srbody(r)$ as the set consisting of every atom in $rbody(r)$ that contains all logical variables appearing in $head(r)$. We say that r is *linear* if $|rbody(r)| \leq 1$. A program $\mathcal{P}$ is *linear* if every rule in $\mathcal{P}$ is linear.

We say that a rule r in an SCC $\mathcal{C}$ of the firing graph is *relevant* if the set of atoms $body(r) \setminus rbody(r)$ does not contain all logical variables in $head(r)$. Roughly speaking, a non-relevant rule will be ignored because its head size is bounded by body atoms which are not mutually recursive with the head (i.e., atoms that do not unify with any rule head or atoms whose predicate symbols are defined by rules in other SCCs). We illustrate the notions introduced so far in the following example.

Example 4. Consider the following program $\mathcal{P}_4$:

$$r_1 : \underbrace{\mathtt{s(f(X),Y)}}_{\text{A}} \leftarrow \underbrace{\mathtt{q(X,f(Y))}}_{\text{B}}, \underbrace{\mathtt{s(Z,f(Y))}}_{\text{C}}.$$
$$r_2 : \underbrace{\mathtt{q(f(U),V)}}_{\text{D}} \leftarrow \underbrace{\mathtt{s(U,f(V))}}_{\text{E}}.$$

The firing graph consists of the edges $\langle r_1, r_1\rangle$, $\langle r_1, r_2\rangle$, $\langle r_2, r_1\rangle$. Thus, there is only one SCC $\mathcal{C} = \{r_1, r_2\}$, which is non-trivial, and $pred(\mathcal{C}) = \{\mathtt{q}, \mathtt{s}\}$. Atoms A and B (resp. A and C, D and E) are mutually recursive. Moreover, $rbody(r_1) = \{B, C\}$, $srbody(r_1) = \{B\}$, $rbody(r_2) = srbody(r_2) = \{E\}$. Both r_1 and r_2 are relevant. □

We use $\mathbb{N}$ to denote the set of natural numbers $\{1, 2, 3, ...\}$ and $\mathbb{N}_0$ to denote the set of natural numbers including the zero. Moreover, $\mathbb{N}^k = \{(v_1, ..., v_k) \mid v_i \in \mathbb{N} \text{ for } 1 \leq i \leq k\}$ and $\mathbb{N}_0^k = \{(v_1, ..., v_k) \mid v_i \in \mathbb{N}_0 \text{ for } 1 \leq i \leq k\}$. Given two k-vectors $\overline{v} = (v_1, ..., v_k)$ and $\overline{w} = (w_1, ..., w_k)$ in $\mathbb{N}_0^k$, we use $\overline{v} \cdot \overline{w}$ to denote the classical scalar product, i.e., $\overline{v} \cdot \overline{w} = \sum_{i=1}^{k} v_i \cdot w_i$.

As mentioned earlier, the basic idea of the proposed technique is to measure the size of terms and atoms in order to check if the rules' head sizes are bounded when propagation occurs. Thus, we introduce the notions of term and atom size.

Definition 3. *Given a rule r and a term t occurring in r, the* size of t w.r.t. r *is recursively defined as follows:*

$$size(t, r) = \begin{cases} 0 & \text{if } t \text{ is either a logical variable not occurring in } head(r) \text{ or a constant;} \\ x & \text{if } t \text{ is a logical variable } X \text{ occurring in } head(r); \\ \sum\limits_{1 \leq i \leq m \,\wedge\, size(t_i, r) \neq 0} (1 + size(t_i, r)) & \text{if } t = f(t_1, ..., t_m). \end{cases}$$

where x is an integer variable. Given an atom $A = p(t_1, ..., t_n)$ in r, the size of A w.r.t. r, *denoted $size(A, r)$, is the n-vector $(size(t_1, r), ..., size(t_n, r))$.* □

In the definition above, every integer variable x intuitively represents the possible sizes that the logical variable X can have during the bottom-up evaluation. Notice that if t is a constant or a logical variable X occurring only in the body, then $size(t, r) = 0$ (in both cases, t does not contribute to the growth of the head). The size of a term of the form $f(t_1, ..., t_m)$ is defined by summing up the size of each t_i having non-zero size, plus 1 (to account for the number of terms of non-zero size which are arguments of f).

Example 5. Consider rule r_1 of program $\mathcal{P}_1$ (see Example 1). Using `lc` to denote the list constructor operator "|", the rule can be rewritten as follows:

$$\texttt{bub(lc(Y,T)},\texttt{lc(X,Cur)},\texttt{Sol)} \leftarrow \texttt{bub(lc(X,lc(Y,T))},\texttt{Cur},\texttt{Sol)},\texttt{X} \leq \texttt{Y}.$$

Let A (resp. B) be the atom in the head (resp. the first atom in the body). Then,

$$\begin{array}{l} size(A, r_1) = ((1 + y) + (1 + t), \quad (1 + x) + (1 + cur), \quad sol) \\ size(B, r_1) = ((1 + x) + [1 + (1 + y) + (1 + t)], \quad cur, \quad sol) \end{array}$$ □

We are now ready to define rule-bounded programs.

Definition 4 (Rule-bounded programs). *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a non-trivial SCC of $\Omega(\mathcal{P})$, and $pred(\mathcal{C}) = \{p_1, ..., p_k\}$. We say that $\mathcal{C}$ is* rule-bounded *if there exist k vectors $\overline{\alpha}_h \in \mathbb{N}^{arity(p_h)}$, $1 \leq h \leq k$, such that for every relevant rule $r \in \mathcal{C}$ with $A = head(r) = p_i(t_1, ..., t_n)$ there exists an atom $B = p_j(u_1, ..., u_m)$ in $srbody(r)$ s.t. the following inequality is satisfied*

$$\overline{\alpha}_j \cdot size(B, r) - \overline{\alpha}_i \cdot size(A, r) \geq 0$$

for every non-negative value of the integer variables in $size(B, r)$ and $size(A, r)$.

We say that $\mathcal{P}$ is rule-bounded *if every non-trivial SCC of $\Omega(\mathcal{P})$ is rule-bounded.* □

Intuitively, for every relevant rule of a non-trivial SCC of $\Omega(\mathcal{P})$, Definition 4 checks if the size of the head atom is bounded by the size of a mutually recursive body atom for all possible sizes the terms can assume. Below is an example of rule-bounded program.

Example 6. Consider again program $\mathcal{P}_4$ of Example 4. Recall that the only non-trivial SCC of $\Omega(\mathcal{P}_4)$ is $\mathcal{C} = \{r_1, r_2\}$, and both r_1 and r_2 are relevant. To determine if the program is rule-bounded we need to check if $\mathcal{C}$ is rule-bounded. Thus, we need to find $\overline{\alpha}_q, \overline{\alpha}_s \in \mathbb{N}^2$ such that there is an atom in $srbody(r_1)$ and an atom in $srbody(r_2)$ which satisfy the two inequalities derived from r_1 and r_2 for all non-negative values of the integer variables therein. Since both $srbody(r_1)$ and $srbody(r_2)$ contain only one element, we have only one choice, namely the one where B is selected for r_1 and E is selected for r_2.

Thus, we need to check if there exist $\overline{\alpha}_q, \overline{\alpha}_s \in \mathbb{N}^2$ s.t. the following linear constraints are satisfied for all non-negative values of the integer variables appearing in them

$$\begin{cases} \overline{\alpha}_q \cdot size(B, r_1) - \overline{\alpha}_s \cdot size(A, r_1) \geq 0 \\ \overline{\alpha}_s \cdot size(E, r_2) - \overline{\alpha}_q \cdot size(D, r_2) \geq 0 \end{cases} \Rightarrow \begin{cases} \overline{\alpha}_q \cdot (x, 1 + y) - \overline{\alpha}_s \cdot (1 + x, y) \geq 0 \\ \overline{\alpha}_s \cdot (u, 1 + v) - \overline{\alpha}_q \cdot (1 + u, v) \geq 0 \end{cases}$$

By expanding the scalar products and isolating every integer variable we obtain:

$$\begin{cases} (\alpha_{q_1} - \alpha_{s_1}) \cdot x + (\alpha_{q_2} - \alpha_{s_2}) \cdot y + (\alpha_{q_2} - \alpha_{s_1}) \geq 0 \\ (\alpha_{s_1} - \alpha_{q_1}) \cdot u + (\alpha_{s_2} - \alpha_{q_2}) \cdot v + (\alpha_{s_2} - \alpha_{q_1}) \geq 0 \end{cases}$$

The previous inequalities must hold for all $x, y, u, v \in \mathbb{N}_0$; it is easy to see that this is the case iff the following system admits a solution:

$$\begin{cases} \alpha_{q_1} - \alpha_{s_1} \geq 0, \quad \alpha_{q_2} - \alpha_{s_2} \geq 0, \quad \alpha_{q_2} - \alpha_{s_1} \geq 0, \\ \alpha_{s_1} - \alpha_{q_1} \geq 0, \quad \alpha_{s_2} - \alpha_{q_2} \geq 0, \quad \alpha_{s_2} - \alpha_{q_1} \geq 0 \end{cases}$$

Since a solution does exist, e.g. $\alpha_{s_1} = \alpha_{s_2} = \alpha_{q_1} = \alpha_{q_2} = 1$ (recall that every α_i must be greater than 0), the SCC $\mathcal{C}$ is rule-bounded, and thus the program is rule-bounded. □

The method to find vectors $\overline{\alpha}_p$ for all $p \in \mathit{pred}(\mathcal{C})$ shown in the previous example can always be applied. That is, we can always isolate the integer variables in the original inequalities and then derive one inequality for each expression that multiplies an integer variable plus the one for the constant term, imposing that all such expressions must be greater than or equal to 0.

It is worth noting that the proposed technique can easily recognize many (finitely-ground) practical programs where terms are simply exchanged from the body to the head of rules (e.g., see Examples 1, 2, and 3).

Example 7. Consider program $\mathcal{P}_1$ of Example 1. Recall that the only non-trivial SCC of $\Omega(\mathcal{P}_1)$ is $\{r_1, r_2, r_3\}$ (see Figure 1) and all rules in it are relevant. Since $|\mathit{srbody}(r_i)| = 1$ for every r_i in the SCC, we have only one set of inequalities, which is the following one (after isolating integer variables):

$$\begin{cases} (\alpha_{b_1} - \alpha_{b_2}) \cdot x_1 + (2\alpha_{b_1} - 2\alpha_{b_2}) \geq 0 \\ (\alpha_{b_1} - \alpha_{b_2}) \cdot y_2 + (2\alpha_{b_1} - 2\alpha_{b_2}) \geq 0 \\ (\alpha_{b_1} - \alpha_{b_3}) \cdot x_3 + (\alpha_{b_2} - \alpha_{b_1}) \cdot \mathit{cur}_3 + (\alpha_{b_1} - 2\alpha_{b_3}) \geq 0 \end{cases}$$

where subscript b stands for predicate symbol `bub`, whereas subscripts associated with integer variables are used to refer to the occurrences of logical variables in different rules (e.g., y_2 is the integer variable associated to the logical variable `Y` in rule r_2). A possible solution is $\overline{\alpha}_b = (2, 2, 1)$ and thus $\mathcal{P}_1$ is rule-bounded.

Considering program $\mathcal{P}_2$ of Example 2, we obtain the following constraints:

$$\begin{cases} (\alpha_{v_1} - \alpha_{v_2}) \cdot \mathit{root}_1 + (\alpha_{v_1} - \alpha_{v_3}) \cdot \mathit{right}_1 + (3\alpha_{v_1} - 2\alpha_{v_2} - 2\alpha_{v_3}) \geq 0 \\ (\alpha_{v_3} - \alpha_{v_1}) \cdot \mathit{next}_2 + 2\alpha_{v_3} \geq 0 \end{cases}$$

where subscript v stands for predicate symbol `visit`. By setting $\overline{\alpha}_v = (2, 1, 2)$, we get positive integer values of $\alpha_{v_1}, \alpha_{v_2}, \alpha_{v_3}$ s.t. the inequalities above are satisfied for all root_1, right_1, $\mathit{next}_2 \in \mathbb{N}_0$. Thus, $\mathcal{P}_2$ is rule-bounded.

The firing graph of program $\mathcal{P}_3$ of Example 3 has two non-trivial SCCs $\mathcal{C}_1 = \{r_1\}$ and $\mathcal{C}_2 = \{r_3\}$. The constraints for $\mathcal{C}_1$ are:

$$\Big\{(\alpha_{r_1} - \alpha_{r_2}) \cdot x_1 + (2\alpha_{r_1} - 2\alpha_{r_2}) \geq 0$$

where subscript r stands for predicate symbol `reverse`. It is easy to see that by choosing any (positive integer) values of α_{r_1} and α_{r_2} such that $\alpha_{r_1} \geq \alpha_{r_2}$, the inequality above holds for all $x_1 \in \mathbb{N}_0$. Likewise, the constraints for $\mathcal{C}_2$ are

$$\Big\{(\alpha_{a_1} - \alpha_{a_2}) \cdot x_3 + (2\alpha_{a_1} - 2\alpha_{a_2}) \geq 0$$

where subscript a stands for predicate symbol `append`. By choosing any (positive integer) values of α_{a_1} and α_{a_2} such that $\alpha_{a_1} \geq \alpha_{a_2}$, the inequality above holds for all $x_3 \in \mathbb{N}_0$. Thus, $\mathcal{P}_3$ is rule-bounded. □

Notice that when checking if an SCC $\mathcal{C}$ is rule-bounded we need to check, for every relevant rule $r \in \mathcal{C}$, if there exists an atom in $srbody(r)$ which satisfies the condition stated in Definition 4. Thus, in the worst case, there are $\prod_{r \in \mathcal{C}} |srbody(r)|$ sets of inequalities for which the condition must be verified. In order to obtain a single set of inequalities for $\mathcal{C}$, the definition might be modified by requiring an inequality *for every* atom in $srbody(r)$. While this variant of Definition 4 would lead to a lower complexity of checking if a program is rule-bounded, the obtained class of rule-bounded programs would be smaller. Clearly, one may also look only at a subset of the $\prod_{r \in \mathcal{C}} |srbody(r)|$ sets of inequalities (e.g., a fixed or polynomial number of them). It is worth noting that in practical cases most of the rules are linear (and thus $|srbody(r)| \leq 1$).

Two key properties of rule-bounded programs are: they are finitely-ground and it is decidable to check whether a given program is rule-bounded.

Theorem 1. *Every rule-bounded program is finitely-ground.* □

Theorem 2. *Checking whether a program is rule-bounded is in NP.* □

It is worth noting that the analysis of the structure of programs is a compile-time operation and the complexity depends on the size of the SCCs, which are usually small.

Theorem 3. *Rule-bounded programs are incomparable with mapping-restricted and bounded programs.* □

Observe that in the previous theorem we have considered only the most general subclasses of finitely-ground programs proposed so far, which generalize previous classes such as argument-restricted programs [21].

4 Cycle-bounded Programs

As saw in the previous section, to determine if a program is rule-bounded we check through linear constraints if the size of the head atom is bounded by the size of a body atom for every relevant rule in a non-trivial SCC of the firing graph (cf. Definition 4). Looking at each rule individually has its limitations, as shown by the following example.

Example 8. Consider the following simple program $\mathcal{P}_8$:

$$\begin{array}{ll} r_1: & \mathtt{p(X,Y)} \quad \leftarrow \mathtt{q(f(X),Y)}. \\ r_2: & \mathtt{q(W,f(Z))} \leftarrow \mathtt{p(W,Z)}. \end{array}$$

It is easy to see that the bottom-up evaluation always terminates, but the program is not rule-bounded. The linear inequalities for the program are (cf. Definition 4):

$$\begin{cases} (\alpha_{q_1} - \alpha_{p_1}) \cdot x + (\alpha_{q_2} - \alpha_{p_2}) \cdot y + \alpha_{q_1} \geq 0 \\ (\alpha_{p_1} - \alpha_{q_1}) \cdot w + (\alpha_{p_2} - \alpha_{q_2}) \cdot z - \alpha_{q_2} \geq 0 \end{cases}$$

It can be easily verified that there are no positive integer values for α_{p_1}, α_{p_2}, α_{q_1}, α_{q_2} such that the inequalities hold for all $x, y, w, z \in \mathbb{N}_0$. The reason is the presence of the expression $-\alpha_{q_2}$ in the second inequality. Intuitively, this is because the size of the head atom increases w.r.t. the size of the body atom in r_2. However, notice that the cycle involving r_1 and r_2 does not increase the overall size of propagated terms. This suggests we can check if an *entire cycle* (rather than each individual rule) propagates terms of bounded size. □

To deal with programs like the one shown in the previous example, we introduce the class of *cycle-bounded programs*, which is able to perform an analysis of how terms propagate through a *group* of rules, rather than looking at rules *individually* as done by the rule-bounded criterion.

Given a program $\mathcal{P}$, a cycle $\pi = \langle r_1, r_2\rangle, \langle r_2, r_3\rangle, ..., \langle r_n, r_1\rangle$ of $\Omega(\mathcal{P})$ is *basic* if every edge does not occur more than once. We say that π is *relevant* if every r_i is relevant, for $1 \leq i \leq n$.

In the following, we first present the cycle-bounded criterion for linear programs and then show how it can be applied to non-linear ones.

Dealing with linear programs. Notice that $rbody(r)$ contains exactly one atom B for every linear rule r in a non-trivial SCC of the firing graph; thus, with a slight abuse of notation, we use $rbody(r)$ to refer to B.

Definition 5 (Linear cycle-bounded programs). *Let $\mathcal{P}$ be a linear program, $\pi = \langle r_1, r_2\rangle, ..., \langle r_n, r_1\rangle$ a basic cycle of $\Omega(\mathcal{P})$, p the predicate symbol defined by r_n, and k the arity of p. Also, let θ_i be an mgu of $head(r_i)$ and $rbody(r_{i+1})$, for $1 \leq i \leq n-1$.*[3] *Given an mgu θ_i ($1 \leq i \leq n-1$) and a pair X/t in θ_i, we define the equality*

$$eq(X/t) = \begin{cases} size(X, r_i) = size(t, r_{i+1}) & \text{if } X \text{ appears in } head(r_i); \\ size(X, r_{i+1}) = size(t, r_i) & \text{if } X \text{ appears in } rbody(r_{i+1}); \end{cases}$$

and define $eq(\theta_i) = \{eq(X/t) \mid X/t \in \theta_i\}$. We say that π is cycle-bounded *if there exists a vector $\overline{\alpha} \in \mathbb{N}^k$ such that the following constraints are satisfied*

$$\{\overline{\alpha} \cdot size(rbody(r_1), r_1) - \overline{\alpha} \cdot size(head(r_n), r_n) \geq 0\} \cup eq(\theta_1) \cup ... \cup eq(\theta_{n-1})$$

[3] Note that such θ_i's always exist by definition of firing graph.

for every non-negative value of the integer variables occurring in the constraints. We say that $\mathcal{P}$ is cycle-bounded *if every relevant basic cycle of $\Omega(\mathcal{P})$ is cycle-bounded.* □

Example 9. Consider again program $\mathcal{P}_8$ of Example 8. The program is clearly linear and $\Omega(\mathcal{P}_8)$ has two relevant basic cycles $\pi_1 = \langle r_1, r_2\rangle, \langle r_2, r_1\rangle$ and $\pi_2 = \langle r_2, r_1\rangle, \langle r_1, r_2\rangle$. To check if π_1 is cycle-bounded we need to check if there exist $\alpha_{q_1}, \alpha_{q_2} \in \mathbb{N}$ s.t. the following constraints are satisfied for all $x, y, w, z \in \mathbb{N}_0$:

$$\begin{cases} \alpha_{q_1} \cdot (x+1) + \alpha_{q_2} \cdot y - \alpha_{q_1} \cdot w - \alpha_{q_2} \cdot (z+1) \geq 0 \\ x = w \\ y = z \end{cases}$$

Notice that the last two equalities above are derived from the mgu $\{\texttt{X/W}, \texttt{Y/Z}\}$ used to unify $head(r_1)$ and $rbody(r_2)$. To check the above condition, we can replace x with w and y with z in the first constraint, thereby obtaining $\alpha_{q_1} - \alpha_{q_2} \geq 0$, which is satisfied for $\alpha_{q_1} \geq \alpha_{q_2}$. Thus, π_1 is cycle-bounded. Likewise, it can be verified that π_2 is cycle-bounded too and thus $\mathcal{P}_8$ is cycle-bounded.

Program $\mathcal{P}_3$ (cf. Example 3) is another linear program that is cycle-bounded. □

Dealing with non-linear programs. The application of the cycle-bounded criterion to arbitrary programs consists in applying the technique to a set of linear programs derived from the original one. Given a rule r, the set of *linear versions* of r is defined as the set of rules $\ell(r) = \{head(r) \leftarrow B \mid B \in rbody(r)\}$. Given a program $\mathcal{P} = \{r_1, ..., r_n\}$, the set of *linear versions* of $\mathcal{P}$ is defined as the set of linear programs $\ell(\mathcal{P}) = \{\{r'_1, ..., r'_n\} \mid r'_i \in \ell(r_i) \text{ for } 1 \leq i \leq n\}$.

Definition 6 (Cycle-bounded programs). *A (possibly non-linear) program $\mathcal{P}$ is* cycle-bounded *if every (linear) program in $\ell(\mathcal{P})$ is cycle-bounded.* □

Like rule-bounded programs, cycle-bounded programs are finitely-ground.

Theorem 4. *Every cycle-bounded program is finitely-ground.* □

Theorem 5. *Checking if a program is cycle-bounded is decidable.* □

Theorem 6. *Cycle-bounded programs are incomparable with rule-bounded, mapping-restricted, and bounded programs.* □

While Theorem 5 establishes decidability of checking if a program is cycle-bounded, we conjecture that the problem is in Π^p_2. Moreover, we point out that cycle-bounded programs are also incomparable with criteria less general than the mapping-restricted and the bounded ones (e.g., argument-restrictedness).

5 Related Work

A significant body of work has been done on termination of logic programs under top-down evaluation [9,36,22,25,8,30,24,28,29,23,5,4,3] and in the area of term

rewriting [37,32,2,11,12]. Termination properties of query evaluation for normal programs under tabling have been studied in [26,27,34].

In this paper, we consider logic programs with function symbols *under the stable model semantics* [15,16] (recall that, as discussed in Section 3, our approach can be applied to programs with disjunction and negation by transforming them into positive normal programs), and thus all the excellent works above cannot be straightforwardly applied to our setting—for a discussion on this see, e.g., [7,1]. In our context, [7] 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*, *λ-restricted programs*, *finite domain programs*, *argument-restricted programs*, *safe programs*, *Γ-acyclic programs*, *mapping-restricted programs*, and *bounded programs*. An adornment-based approach that can be used in conjunction with the techniques above to detect more programs as finitely-ground has been proposed in [18].

Compared with the aforementioned classes, rule- and cycle-bounded programs allow us to perform a more global analysis and identify many practical programs as finitely-ground, such as those where terms in the body are rearranged in the head, which are not included in any of the classes above. We observe that there are also programs which are not rule- or cycle-bounded but are recognized as finitely-ground by some of the aforementioned techniques (see Theorems 3 and 6).

Similar concepts of "term size" have been considered to check termination of logic programs evaluated in a top-down fashion [31], in the context of partial evaluation to provide conditions for strong termination and quasi-termination [35,20], and in the context of tabled resolution [26,27]. These approaches are geared to work under top-down evaluation, looking at how terms are propagated from the head to the body, while our approach is developed to work under bottom-up evaluation, looking at how terms are propagated from the body to the head. This gives rise to significant differences in how the program analysis is carried out, making one approach not applicable in the setting of the other. As a simple example, the rule $\mathtt{p(X)} \leftarrow \mathtt{p(X)}$ leads to a non-terminating top-down evaluation, while it is completely harmless under bottom-up evaluation.

6 Conclusions

Recently, there has been a great deal of interest in enhancing answer set programming with function symbols. Research has focused on identifying classes of logic programs allowing only a limited use of function symbols but guaranteeing decidability of common inference tasks. Despite many excellent techniques that have been proposed in recent years, there are still many terminating practical programs which are not captured by any of the approaches in the literature.

In this paper, we have introduced the novel class of rule-bounded programs, which overcomes different limitations of current approaches by performing a more global analysis of programs, thereby identifying many programs commonly

arising in practice as finitely-ground. We have also introduced the class of cycle-bounded programs where groups of rules are analyzed.

As a direction for future work, we plan to investigate how our techniques can be combined with current termination criteria. Since they look at programs from different standpoints, an interesting issue is to study how they can be integrated so that they can benefit from each other. To this end, an interesting approach would be to plug termination criteria in the generic framework proposed in [10] and study their combination in such a framework. Another intriguing issue would be to analyze the relationships between the notions of safety of [10] and the notions of boundedness used by termination criteria.

References

1. Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *TPLP*, 10(4-6):497–512, 2010.
2. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3. Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. *TPLP*, 9(2):213–238, 2009.
4. Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.
5. Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.
6. Marco Calautti, Sergio Greco, and Irina Trubitsyna. Detecting decidable classes of finitely ground logic programs with function symbols. In *PPDP*, pages 239–250, 2013.
7. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In *ICLP*, pages 407–424, 2008.
8. Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Testing for termination with monotonicity constraints. In *ICLP*, pages 326–340, 2005.
9. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
10. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal safety for answer set programs with external sources. In *AAAI*, 2013.
11. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reas.*, 40(2-3):195–220, 2008.
12. Maria C. F. Ferreira and Hans Zantema. Total termination of term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 7(2):133–162, 1996.
13. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
14. Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.
15. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.
16. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

17. Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Bounded programs: A new decidable class of logic programs with function symbols. In *IJCAI*, pages 926–932, 2013.
18. Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *TPLP*, 13(4-5):737–752, 2013.
19. Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. On the termination of logic programs with function symbols. In *ICLP (Technical Communications)*, pages 323–333, 2012.
20. Michael Leuschel and Germán Vidal. Fast offline partial evaluation of logic programs. *Information and Computation*, 235(0):70–97, 2014.
21. Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *ICLP*, pages 489–493, 2009.
22. Massimo Marchiori. Proving existential termination of normal logic programs. In *Algebraic Methodology and Software Technology*, pages 375–390, 1996.
23. 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.
24. Naoki Nishida and Germán Vidal. Termination of narrowing via termination of rewriting. *Appl. Algebra Eng. Commun. Comput*, 21(3):177–225, 2010.
25. Enno Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):73–116, 2001.
26. Fabrizio Riguzzi and Terrance Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *TPLP*, 13(2):279–302, 2013.
27. Fabrizio Riguzzi and Terrance Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM Transactions on Computational Logic*, 2014.
28. Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Trans. Comput. Log.*, 11(1), 2009.
29. Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, and René Thiemann. Automated termination analysis for logic programs with cut. *TPLP*, 10(4-6):365–381, 2010.
30. Alexander Serebrenik and Danny De Schreye. On termination of meta-programs. *TPLP*, 5(3):355–390, 2005.
31. Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes. In *PODS*, pages 216–226, 1991.
32. Christian Sternagel and Aart Middeldorp. Root-labeling. In *Rewriting Techniques and Applications*, pages 336–350, 2008.
33. Tommi Syrjanen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.
34. Sofie Verbaeten, Danny De Schreye, and Konstantinos F. Sagonas. Termination proofs for logic programs with tabling. *ACM Trans. Comput. Log.*, 2(1):57–92, 2001.
35. Germán Vidal. Quasi-terminating logic programs for ensuring the termination of partial evaluation. In *PEPM*, pages 51–60, 2007.
36. Dean Voets and Danny De Schreye. Non-termination analysis of logic programs with integer arithmetics. *TPLP*, 11(4-5):521–536, 2011.
37. Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1/2):89–105, 1995.