Logic Program Termination Analysis Using Atom Sizes

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

Abstract

Recent years have witnessed a great deal of interest in extending answer set programming with function symbols. Since the evaluation of a program with function symbols might not terminate and checking termination is undecidable, several classes of logic programs have been proposed where the use of function symbols is limited but the program evaluation is guaranteed to terminate.
In this paper, we propose a novel class of logic programs whose evaluation always terminates. The proposed technique identifies terminating programs that are not captured by any of the current approaches. Our technique is based on the idea of measuring the size of terms and atoms to check whether the rule head size is bounded by the body, and performs a more fine-grained analysis than previous work. Rather than adopting an all-or-nothing approach (either we can say that the program is terminating or we cannot say anything), our technique can identify arguments that are "limited" (i.e., where there is no infinite propagation of terms) even when the program is not entirely recognized as terminating. Identifying arguments that are limited can support the user in the problem formulation and help other techniques that use limited arguments as a starting point. Another useful feature of our approach is that it is able to leverage external information about limited arguments. We also provide results on the correctness, the complexity, and the expressivity of our technique.

1 Introduction

Function symbols are widely acknowledged as an important feature in answer set programming as they make modeling easier and increase the language's expressive power. Current solvers provide support for them, but offer only a limited a-priori termination analysis of programs. The main problem is that the evaluation of a program (with function symbols) might not terminate and checking termination is undecidable.

To cope with this issue, recent research has focused on identifying decidable classes of logic programs allowing only a restricted use of function symbols but guaranteeing the program evaluation termination (we will discuss current approaches in the related work subsection).

Most of the work in the literature analyzes programs by looking at how terms are propagated from one individual argument to another. More general approaches such as the mapping-restricted [Calautti *et al.*, 2013] and the bounded [Greco *et al.*, 2013a] techniques are able to perform a more complex (yet limited) analysis of how some groups of arguments affect each other. Recently, [Calautti *et al.*, 2014a] proposed the rule-bounded criterion, which checks if the head size is bounded by the size of a body atom in a rule. However, all current approaches have several limitations that we illustrate in the following example.

Example 1 Consider the following simple program $\mathcal{P}_1$:

$$\mathtt{p}(\mathtt{f}(\mathtt{X},\mathtt{X}),\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}(\mathtt{X},\mathtt{g}(\mathtt{Z}),\mathtt{g}(\mathtt{Y})).$$

The program evaluation always terminates whatever finite set of facts is added to the program—however, all current termination criteria fail to realize this. For instance, the simple fact that the first argument of p has a size in the head greater than the one in the body prevents several techniques from realizing termination of the program evaluation. Also, when comparing the overall size of the head with the body size, current criteria do not succeed in identifying the program as terminating. In contrast, as we will show in the following, our approach performs a more accurate analysis and realizes that the program evaluation always terminates. □

To provide a practical example, below we report a general program which recognizes strings of the language corresponding to an arbitrary $LR(1)$ grammar.

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

$$\begin{aligned}
\mathtt{par}(\mathtt{T},[\mathtt{S1}|[\mathtt{Sym}|[\mathtt{St}|\mathtt{L}]]]) &\leftarrow \mathtt{par}([\mathtt{Sym}|\mathtt{T}],[\mathtt{St}|\mathtt{L}]),\\
&\quad\ \mathtt{act}(\mathtt{St},\mathtt{Sym},\mathtt{shift}(\mathtt{S1})).\\
\mathtt{red}([\mathtt{Sym}|\mathtt{T}],[\mathtt{St}|\mathtt{L}],\mathtt{A},\mathtt{B}) &\leftarrow \mathtt{par}([\mathtt{Sym}|\mathtt{T}],[\mathtt{St}|\mathtt{L}]),\\
&\quad\ \mathtt{act}(\mathtt{St},\mathtt{Sym},\mathtt{reduce}(\mathtt{A},\mathtt{B})).\\
\mathtt{red}(\mathtt{I},\mathtt{L},\mathtt{A},\mathtt{T}) &\leftarrow \mathtt{red}(\mathtt{I},[\mathtt{S}|[\mathtt{X}|\mathtt{L}]],\mathtt{A},[\mathtt{Y}|\mathtt{T}]).\\
\mathtt{par}(\mathtt{I},[\mathtt{S1}|[\mathtt{A}|[\mathtt{St}|\mathtt{L}]]]) &\leftarrow \mathtt{red}(\mathtt{I},[\mathtt{St}|\mathtt{L}],\mathtt{A},[]),\\
&\quad\ \mathtt{act}(\mathtt{St},\mathtt{A},\mathtt{goto}(\mathtt{S1})).
\end{aligned}$$

where we use the classical syntax [H|T] for a list. $LR(1)$ grammars can be encoded in a standard form using an action table defined by facts of the form $\mathtt{act}(\langle\mathtt{state}\rangle,\langle\mathtt{symbol}\rangle,\langle\mathtt{operation}\rangle)$.

Specifically, given the current parsing state $\langle \texttt{state} \rangle$ and a symbol $\langle \texttt{symbol} \rangle$ to be parsed, $\langle \texttt{operation} \rangle$ describes one of the following four parsing operations: $\texttt{shift}(\langle \texttt{newstate} \rangle)$, i.e., the next token is read from the input and pushed to the parsing stack along with the new parsing state $\langle \texttt{newstate} \rangle$; $\texttt{reduce}(\texttt{A}, \texttt{B})$, i.e., there is a production rule $\texttt{A} \rightarrow \texttt{B}$ in the grammar and the top of the parsing stack contains B (according to $\langle \texttt{state} \rangle$), which must be replaced with A; $\texttt{goto}(\langle \texttt{newstate} \rangle)$, i.e., once the reduce operation is complete, the parsing state changes accordingly; accept, i.e., the input string is accepted. The computation starts by providing as input the action table and a fact of the form $\texttt{par}([\texttt{a}_1, ..., \texttt{a}_n, \$], [\texttt{s}_0])$, where $[\texttt{a}_1, ..., \texttt{a}_n, \$]$ is the input string, followed by the "end of string symbol" \$, and $[\texttt{s}_0]$ is the parsing stack containing the initial state $\texttt{s}_0$. The string is accepted iff the program model contains two atoms of the form $\texttt{par}([\$], [\texttt{s}|\texttt{L}])$ and $\texttt{act}(\texttt{s}, \$, \texttt{accept})$. □

Once again, the program above terminates for every finite set of facts—while none of the current approaches is able to realize it, our technique detects the program as terminating.

Related work. A significant body of work has been done on termination of logic programs under top-down evaluation [De Schreye and Decorte, 1994; Marchiori, 1996; Ohlebusch, 2001; Codish *et al.*, 2005; Schneider-Kamp *et al.*, 2009; Nguyen *et al.*, 2007; Bruynooghe *et al.*, 2007; Baselice *et al.*, 2009; Voets and De Schreye, 2011] and in the area of term rewriting [Sternagel and Middeldorp, 2008; Arts and Giesl, 2000; Endrullis *et al.*, 2008]. Termination properties of query evaluation for normal programs under tabling have been studied in [Riguzzi and Swift, 2013; 2014; Verbaeten *et al.*, 2001]. Another approach are $\mathbb{FDNC}$ programs [Eiter and Simkus, 2010], which have infinite answer sets in general, but a finite representation that can be exploited for knowledge compilation and fast query answering.

In this paper, we consider logic programs with function symbols *under the stable model semantics* [Gelfond and Lifschitz, 1988; 1991], and thus all the 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. Since membership in the class is not decidable, decidable subclasses have been proposed: ω*-restricted programs* [Syrjanen, 2001], λ*-restricted programs* [Gebser *et al.*, 2007], *finite domain programs* [Calimeri *et al.*, 2008], *argument-restricted programs* [Lierler and Lifschitz, 2009], *safe* and Γ*-acyclic programs* [Calautti *et al.*, 2014b]. More general classes are *mapping-restricted programs* [Calautti *et al.*, 2013], *bounded programs* [Greco *et al.*, 2013a], and *rule-* and *cycle-bounded programs* [Calautti *et al.*, 2014a]. 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 [Greco *et al.*, 2013b].

Our approach recognizes terminating programs that do not belong to any of the aforementioned classes; also, it can be easily combined with them allowing us to identify more programs (we provide a precise comparison in Section 4).

Concepts of "term size" similar to the one used in this paper have been considered to check termination of logic programs evaluated in a top-down fashion [Sohn and Gelder, 1991], in the context of partial evaluation to provide conditions for strong termination and quasi-termination [Leuschel and Vidal, 2014], and in the context of tabled resolution [Riguzzi and Swift, 2013; 2014]. 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 $\texttt{p}(\texttt{X}) \leftarrow \texttt{p}(\texttt{X})$ leads to a non-terminating top-down evaluation, while it is completely harmless under bottom-up evaluation.

Our work is also related to research done in the database community on termination of the chase procedure, where existential rules are considered (see [Greco *et al.*, 2012; 2011]).

Contribution. We propose a novel class of logic programs with function symbols whose evaluation always terminates. Our technique is based on the idea of measuring the size of terms and atoms using linear constraints to check whether the rule head size is bounded by the body. The proposed approach generalizes previous work along different dimensions.

First, our approach identifies as terminating strictly more programs than the rule-bounded criterion, including programs that are not identified by any of the current approaches in the literature. While the rule-bounded technique looks at the *entire* size of a *single* atom, our technique performs a more accurate analysis by looking at the size of *parts* of *multiple* atoms, overcoming different limitations of the rule-bounded criterion.

Second, in contrast to several current criteria, rather than adopting an "all-or-nothing" approach (either we can say that the program evaluation terminates or we cannot say anything), our technique can identify arguments that are "limited" (i.e., arguments where there is no infinite propagation of terms) even when the program is not entirely recognized. Identifying arguments that are limited can support the user in the problem formulation. Moreover, termination criteria that use limited arguments as a starting point (e.g., the bounded criterion) can take advantage of this information.

Third, our technique can leverage external information about limited arguments for a better understanding of the program evaluation behavior—current approaches such as the argument-restricted and the bounded criteria can provide such sets of limited arguments when they fail to recognize a program. This feature as well as the previous one are an important step towards the combination of termination criteria, enabling different approaches to benefit from each other—this is an issue where little effort has be done so far.

Finally, we provide results on the correctness, the complexity, and the expressivity of the proposed technique.

Organization. In Section 2 we report preliminaries. Section 3 introduces our technique. Section 4 reports results on the complexity and expressivity. Section 5 shows how our technique can be iteratively applied.

2 Preliminaries

This section recalls syntax and the stable model semantics of logic programs with function symbols [Gelfond and Lifschitz, 1988; 1991; Gebser *et al.*, 2012].

Syntax. We assume to have (pairwise disjoint) infinite sets of *logical variables*, *predicate symbols*, and *function symbols*. Logical variables are used in logic programs and are denoted by upper-case letters. To each logical variable X, there corresponds a (unique) *integer variable* x (denoted by the same letter in lower case) which may occur in linear constraints. Each predicate and function symbol g is associated with an *arity*, which is a non-negative integer. Function symbols of arity 0 are called *constants*. A *term* is either a logical variable or an expression of the form $f(t_1, ..., t_m)$, where f is a function symbol of arity $m \geq 0$ 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 $n \geq 0$ and $t_1, ..., t_n$ are terms—we also call the atom a p-atom. We use $pr(A)$ to denote the predicate symbol of an atom A. 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 $body(r)$. 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. 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*.

Let $\mathcal{P}$ be a program. The set of all predicate symbols appearing in $\mathcal{P}$ (resp. appearing in the head of a rule in $\mathcal{P}$) is denoted as $pred(\mathcal{P})$ (resp. $def(\mathcal{P})$). Given a predicate symbol p of arity n, the i-th *argument* of p is an expression of the form $p[i]$, for $1 \leq i \leq n$. The set of all arguments of the predicate symbols in $pred(\mathcal{P})$ is denoted by $args(\mathcal{P})$.

Semantics. Consider a program $\mathcal{P}$. The *Herbrand universe* $H_\mathcal{P}$ of $\mathcal{P}$ is the possibly infinite set of ground terms that can be built using function symbols (and thus also constants) appearing in $\mathcal{P}$. The *Herbrand base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of ground atoms that can be built using predicate symbols appearing in $\mathcal{P}$ and ground terms of $H_\mathcal{P}$.

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 A, denoted $A\theta$, is the atom 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 such that $A_1\theta = A_2\theta$.

A rule (resp. atom) r' is a *ground instance* of a rule (resp. atom) 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 $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$, and $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$ for positive programs. A positive normal program $\mathcal{P}$ has a unique minimal model, which we denote as $\mathcal{MM}(\mathcal{P})$.

Limited programs. Consider a program $\mathcal{P}$. An argument $p[i]$ in $args(\mathcal{P})$ is said to be *limited* iff for every finite set of facts D and for every stable model M of $\mathcal{P} \cup D$, the set $\{t_i \mid p(t_1, ..., t_i, ..., t_n) \in M\}$ is finite. Moreover, $\mathcal{P}$ is said to be *limited* iff every argument in $args(\mathcal{P})$ is limited.

3 Size-Restricted Programs

In this paper we study new conditions under which a positive normal program $\mathcal{P}$ is limited—equivalently, the bottom-up evaluation always terminates for every finite set of facts added to the program. It is worth mentioning that our technique can be used to check if an arbitrary program $\mathcal{P}$ (possibly with disjunction in the head and negation in the body) has a finite number of stable models, each of them has finite size and can be computed. Specifically, it suffices to apply our technique to a positive normal program $st(\mathcal{P})$ derived from $\mathcal{P}$ as follows. Every rule $A_1 \vee ... \vee A_m \leftarrow body$ in $\mathcal{P}$ 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, 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 [Calautti *et al.*, 2014b]. Thus, for ease of presentation, in the rest of the paper a program is understood to be positive and normal.

Also, notice that a (positive normal) program $\mathcal{P}$ is limited iff the program obtained from $\mathcal{P}$ by deleting all its facts is limited. Thus, w.l.o.g., hereafter we assume that every given program $\mathcal{P}$ does not contain facts.

We start by reporting the definition of *firing graph* [Calautti

et al., 2014a], a directed graph that keeps track of whether a rule can trigger another.

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 where there is an edge $\langle r, r' \rangle$ iff 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". In the definition above, w.l.o.g., we assume that different rules do not share logical variables, and 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 a program $\mathcal{P}$ is a maximal set $\mathcal{C}$ of nodes of $\Omega(\mathcal{P})$ s.t. every node of $\mathcal{C}$ can be reached from every node of $\mathcal{C}$ through the edges in $\Omega(\mathcal{P})$—a node always reaches itself.

Given a rule $r \in \mathcal{P}$, we say that the head atom is *mutually recursive* with an atom $B \in body(r)$ if there is an SCC $\mathcal{C}$ of $\mathcal{P}$ containing r and containing a rule r' (possibly equal to r) s.t. $head(r')$ and B unify. The set of all atoms in $body(r)$ that are mutually recursive with $head(r)$ is denoted as $rbody(r)$.

Given a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$, we say that a rule $r \in \mathcal{P}$ is *$\mathcal{A}$-relevant* if $head(r)$ contains at least one variable which does not appear in $body(r) \setminus rbody(r)$ and does not appear in a term t_i of a body atom $p(t_1, ..., t_n)$ such that $p[i] \in \mathcal{A}$. Rules that are not $\mathcal{A}$-relevant will not be considered in the analysis of an SCC (Definition 4) because they cannot infinitely propagate terms (when the SCC is considered in isolation), as all head variables appear in either a body atom which is not mutually recursive with the head or in correspondence of a limited argument. The following example illustrates the aforementioned notions.

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

$$r_1 : \underbrace{\mathtt{p(f(X),Y)}}_{\mathtt{A}} \leftarrow \underbrace{\mathtt{p(X,f(Y))}}_{\mathtt{B}}, \underbrace{\mathtt{b(X,Z)}}_{\mathtt{C}}.$$
$$r_2 : \underbrace{\mathtt{p(X,g(Y))}}_{\mathtt{D}} \leftarrow \underbrace{\mathtt{p(f(X),Y)}}_{\mathtt{E}}.$$

The firing graph of $\mathcal{P}_3$ has the edges $\langle r_1, r_1 \rangle$, $\langle r_2, r_2 \rangle$, and $\langle r_1, r_2 \rangle$. The SCCs of $\mathcal{P}_3$ are $\mathcal{C}_1 = \{r_1\}$ and $\mathcal{C}_2 = \{r_2\}$. Atom `B` is mutually recursive with `A`, and atom `E` is mutually recursive with `D`. Atom `C` is *not* mutually recursive with `A`. Furthermore, given the set of limited arguments $\mathcal{A} = \{\mathtt{p[2]}\}$, rule r_2 is $\mathcal{A}$-relevant, since variable `X` occurring in `D` appears only in the mutually recursive body atom `E` inside argument `p[1]`, which is not in $\mathcal{A}$. Conversely, r_1 is not $\mathcal{A}$-relevant, since variables `X`, `Y` appearing in `A` occur in the body respectively in the non-mutually recursive atom `C` and inside $\mathtt{p[2]} \in \mathcal{A}$ of atom `B`. □

We use $\mathbb{Z}$ to denote the set of all integers and $\mathbb{N}$ to denote the set of all non-negative integers. Given two k-vectors $\overline{v} = (v_1, ..., v_k)$ and $\overline{w} = (w_1, ..., w_k)$ in $\mathbb{Z}^k$, we use $\overline{v} \cdot \overline{w}$ to denote the classical scalar product, that is, $\overline{v} \cdot \overline{w} = \sum_{i=1}^{k} v_i \cdot w_i$. We also use the notation $\overline{v}[i]$ to refer to v_i, for $1 \leq i \leq k$.

Definition 2 (Term/atom size) The *size* of a term t, denoted $size(t)$, is recursively defined as follows:

$$size(t) = \begin{cases} x & \text{if } t \text{ is a logical variable } X. \\ m + \sum_{i=1}^{m} size(t_i) & \text{if } t = f(t_1, ..., t_m). \end{cases}$$

where x is the integer variable corresponding to X. The *size* of an atom $A = p(t_1, ..., t_n)$, denoted as $size(A)$, is the n-vector $(size(t_1), ..., size(t_n))$. □

In the definition above, an integer variable x intuitively represents the possible sizes that the logical variable X can have during the bottom-up evaluation. The size of a term of the form $f(t_1, ..., t_m)$ is defined by summing up the size of its terms t_i plus the arity m of f. Note that the size of every constant is 0.

Example 4 Consider the atom $A = \mathtt{p(a, X, f(X, g(X, Y)))}$. Since $size(\mathtt{a}) = 0$, $size(\mathtt{X}) = x$, and $size(\mathtt{f(X, g(X, Y))}) = 2 + x + (2 + x + y) = 2x + y + 4$, we have that $size(A) = (0, x, 2x + y + 4)$. □

As mentioned before, one of the features of our technique is the capability of leveraging information about arguments that are known to be limited. In order to enable our technique to exploit this kind of information, several notions introduced in the following are defined w.r.t. a set $\mathcal{A}$ of arguments, to be read as the set of arguments that are known to be limited when our criterion is applied to a given program.

Definition 3 (Argument/predicate domain) Given a program $\mathcal{P}$ and a set of arguments $\mathcal{A}$, the *domain* of an argument $p[i] \in args(\mathcal{P})$ w.r.t. $\mathcal{A}$, denoted $\mathbb{D}_{\mathcal{A}}(p[i])$, is $\mathbb{Z}$ if $p[i] \in \mathcal{A}$, and $\mathbb{N}$ otherwise. The *domain* of a predicate symbol p of arity n is $\mathbb{D}_{\mathcal{A}}(p) = \mathbb{D}_{\mathcal{A}}(p[1]) \times \cdots \times \mathbb{D}_{\mathcal{A}}(p[n])$. □

Below we define when an argument is *$\mathcal{A}$-size-restricted* in an SCC of a program—as shown in the following, this ensures that the argument is limited when the SCC is considered in isolation. Then, in Definition 6, we will define how to combine the information coming from all the SCCs in order to determine whether or not an argument is $\mathcal{A}$-size-restricted in the entire program.

Definition 4 (Size-restricted arguments in an SCC) Consider a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$. Let $\mathcal{C}$ be an SCC of $\mathcal{P}$ with $pred(\mathcal{C}) = \{p_1, ..., p_n\}$. We say that an argument $p_i[j]$ of $\mathcal{C}$ is *$\mathcal{A}$-size-restricted in $\mathcal{C}$* iff

1. for every rule $r \in \mathcal{C}$ such that $head(r) = p_i(t_1, ..., t_m)$ the following condition holds: for every variable X occurring in t_j, there exists a term u_k of a body atom $q(u_1, ..., u_{m'})$ s.t. X occurs in u_k and $q[k] \in \mathcal{A}$; or
2. there exist n vectors $\overline{\alpha}_h \in \mathbb{D}_{\mathcal{A}}(p_h)$, $1 \leq h \leq n$, such that for every $\mathcal{A}$-relevant rule $r \in \mathcal{C}$ there exists an atom B in $body(r)$ such that if $pr(head(r)) = p_k$ and $pr(B) = p_l$, then the following conditions hold:
   (a) the constraint
   $$\overline{\alpha}_l \cdot size(B) \geq \overline{\alpha}_k \cdot size(head(r))$$
   is satisfied for every non-negative value of the integer variables in it; and

(b) if $p_k = p_i$ then either $\overline{\alpha}_i[j] \neq 0$ or the constraint

$$\overline{\alpha}_l \cdot size(B) > \overline{\alpha}_i \cdot size(head(r))$$

is satisfied for every non-negative value of the integer variables in it. □

Condition 1 of the definition above simply checks if $p_i[j]$ is $\mathcal{A}$-size-restricted because for every rule of $\mathcal{C}$ having p_i in the head, all variables appearing in correspondence of $p_i[j]$ appear in the body in correspondence of a limited argument.

As for Condition 2, roughly speaking, Definition 4 says that an argument $p_i[j]$ is $\mathcal{A}$-size-restricted in an SCC if, for every (relevant) rule, the size of part of the head is always bounded by the size of part of a body atom, to within a constant factor. When $\overline{\alpha}_i[j] = 0$, a stricter inequality must be satisfied for the rules having p_i in the head. When other coefficients are 0, we are considering only parts of atoms in the analysis—e.g., assuming that $\overline{\alpha}_k[1] = 0$, this means that the first term in every p_k-atom is ignored in the analysis. Notice that only the coefficients associated with limited arguments can assume arbitrary values in $\mathbb{Z}$. We notice that while the rule-bounded criterion allows positive coefficients only, here we allow coefficients to be zero and take negative values (this last case applies to limited arguments only).

Example 5 Consider program $\mathcal{P}_1$ of Example 1, reported below:

$$\mathtt{p(f(X,X),Y,Z)} \leftarrow \mathtt{p(X,g(Z),g(Y))}.$$

Let us consider $\mathcal{A} = \emptyset$. The program has only one SCC $\mathcal{C}$ consisting of the rule above, which is $\mathcal{A}$-relevant. The vector $\overline{\alpha}_p = (0, 1, 1)$ allows us to say that all arguments are $\mathcal{A}$-size-restricted in $\mathcal{C}$. In fact, when arguments $\mathtt{p}[2]$ and $\mathtt{p}[3]$ are considered, Condition 2(a) of Definition 4 holds since

$$(0, 1, 1) \cdot (x, z + 1, y + 1) \geq (0, 1, 1) \cdot (2x + 2, y, z)$$

is satisfied for all non-negative values of the integer variables, and Condition 2(b) is trivially satisfied because both $\overline{\alpha}_p[2]$ and $\overline{\alpha}_p[3]$ are not 0. When argument $\mathtt{p}[1]$ is considered, Condition 2(a) is the same as before and thus is satisfied, and Condition 2(b) holds too since the constraint above with a strict inequality is still satisfied for all non-negative values of the integer variables. □

Example 6 Consider again program $\mathcal{P}_2$ of Example 2, which has only one SCC $\mathcal{C}$ coinciding with $\mathcal{P}_2$ itself. Let us consider $\mathcal{A} = \emptyset$. All rules are $\mathcal{A}$-relevant. We now show that every argument is $\mathcal{A}$-size-restricted in $\mathcal{C}$. In particular, consider the inequalities associated with the rules of $\mathcal{P}_2$ when the `act`-atoms are selected in the body of the first, second, and fourth rule, and the `red`-atom is selected for the third rule:

$$\begin{cases} \overline{\alpha}_{act} \cdot (st, sym, 1{+}s_1) \geq \overline{\alpha}_{par} \cdot (t, 6{+}s_1{+}sym{+}st{+}l) \\ \overline{\alpha}_{act} \cdot (st, sym, 2{+}a{+}b) \geq \overline{\alpha}_{red} \cdot (2{+}sym{+}t, 2{+}st{+}l, a, b) \\ \overline{\alpha}_{red} \cdot (i, 4{+}s{+}x{+}l, a, 2{+}y{+}t) \geq \overline{\alpha}_{red} \cdot (i, l, a, t) \\ \overline{\alpha}_{act} \cdot (st, a, 1{+}s_1) \geq \overline{\alpha}_{par} \cdot (i, 6{+}s_1{+}a{+}st{+}l) \end{cases}$$

By incorporating the vectors $\overline{\alpha}_{act} = (1, 1, 1)$, $\overline{\alpha}_{par} = (0, 0)$, $\overline{\alpha}_{red} = (0, 0, 1, 1)$ into the constraints above, we obtain:

$$\begin{cases} st + sym + s_1 + 1 & \geq & 0 \\ st + sym + a + b + 2 & \geq & a + b \\ a + y + t + 2 & \geq & a + t \\ st + a + s_1 + 1 & \geq & 0 \end{cases}$$

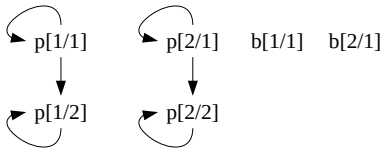

Figure 1: Extended argument graph of $\mathcal{P}_3$.

It is easy to see that the constraints above are satisfied for every $st, sym, s_1, a, b, y, t \in \mathbb{N}$, and thus Condition 2(a) of Definition 4 holds for all arguments. Moreover, since $\overline{\alpha}_{act}[1]$, $\overline{\alpha}_{act}[2]$, $\overline{\alpha}_{act}[3]$, $\overline{\alpha}_{red}[3]$, and $\overline{\alpha}_{red}[4]$ are all different from 0, we can say that arguments $\mathtt{act}[1], \mathtt{act}[2], \mathtt{act}[3], \mathtt{red}[3]$, and $\mathtt{red}[4]$ are $\mathcal{A}$-size-restricted in $\mathcal{C}$, as Condition 2(b) is also satisfied. For arguments $\mathtt{par}[1], \mathtt{par}[2], \mathtt{red}[1]$, and $\mathtt{red}[2]$ (whose coefficients are 0), we have to check if the constraints associated with the rules having predicate symbol `par` (resp. `red`) in the head, namely the first and the last one (resp. the second and third one), are satisfied with a strict inequality. As this is the case, Condition 2(b) holds, and arguments $\mathtt{par}[1]$, $\mathtt{par}[2], \mathtt{red}[1]$, and $\mathtt{red}[2]$ are $\mathcal{A}$-size-restricted in $\mathcal{C}$. □

We now define how to determine if an argument is $\mathcal{A}$-size-restricted in the entire program. This is done by combining the information obtained from the individual analysis of the SCCs. We start by introducing some additional notions.

Given a program $\mathcal{P}$, we assume an arbitrary but fixed numbering $\mathcal{C}_1, ..., \mathcal{C}_n$ of its SCCs. We also define $ex\text{-}args(\mathcal{P})$ as the set $\{p[i/j] \mid \mathcal{C}_j \text{ is an SCC of } \mathcal{P} \text{ and } p[i] \in args(\mathcal{C}_j)\}$. Each element of $ex\text{-}args(\mathcal{P})$ is called an *extended argument* of $\mathcal{P}$. The next tool is called *extended argument graph*—a directed graph keeping track of the propagation of terms between arguments. It is a refinement of the argument graph of [Calimeri *et al.*, 2008] and it leverages the firing graph to perform a component-wise analysis of how terms are propagated between arguments and to get rid of propagation (between arguments) that cannot really occur.

Definition 5 (Extended argument graph) The *extended argument graph* of a program $\mathcal{P}$, denoted $\Delta(\mathcal{P})$, is a directed graph whose set of nodes is $ex\text{-}args(\mathcal{P})$ and where there is an edge $\langle q[j/k], p[i/l] \rangle$ iff

- $k = l$ and there is a rule $r \in \mathcal{C}_k$ such that *(1)* $head(r)$ is a p-atom, *(2)* there is a q-atom B in $body(r)$, *(3)* the i-th term of $head(r)$ and j-th term of B have a common variable, and *(4)* there is a rule $r' \in \mathcal{P}$ such that $head(r')$ and B unify; or
- $k \neq l$ and $p = q$, $i = j$, and there are two rules $r_1 \in \mathcal{C}_k$ and $r_2 \in \mathcal{C}_l$ such that $pr(head(r_1)) = p$ and $\langle r_1, r_2 \rangle$ is an edge of $\Omega(\mathcal{P})$. □

Intuitively, an edge $\langle q[j/k], p[i/l] \rangle$ of $\Delta(\mathcal{P})$ means that there can be a propagation of terms from $q[j]$ in component $\mathcal{C}_k$ to $p[i]$ in component $\mathcal{C}_l$. We say that an extended argument $p[i/l]$ *depends on* an extended argument $q[j/k]$ if there is a path from the latter to the former in $\Delta(\mathcal{P})$.

Example 7 Consider again program $\mathcal{P}_3$ of Example 3. Figure 1 illustrates $\Delta(\mathcal{P}_3)$. □

We are now ready to define when an argument is $\mathcal{A}$-size-restricted in a program.

Definition 6 ($\mathcal{A}$-size-restricted arguments/programs) Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. An argument $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{P}$ if for every SCC $\mathcal{C}_l$ of $\mathcal{P}$ such that $p \in pred(\mathcal{C}_l)$,

1. $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_l$, and
2. $p[i/l]$ depends only on extended arguments $q[j/k]$ such that $q[j]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_k$.

We denote by $\mathcal{R}_{\mathcal{A}}(\mathcal{P})$ the set of all $\mathcal{A}$-size-restricted arguments in $\mathcal{P}$. We say that $\mathcal{P}$ is *$\mathcal{A}$-size-restricted* iff $args(\mathcal{P}) = \mathcal{A} \cup \mathcal{R}_{\mathcal{A}}(\mathcal{P})$. □

Example 8 Consider program $\mathcal{P}_3$ of Example 3, whose extended argument graph is shown in Figure 1 and let $\mathcal{A} = \{\mathtt{p}[2]\}$. Below we show that $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{P}_3$. Since $\mathtt{p} \in pred(\mathcal{C}_1)$ and $\mathtt{p} \in pred(\mathcal{C}_2)$, we first need to check if $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_1$ and $\mathcal{C}_2$. Since $\mathcal{C}_1 = \{r_1\}$ and r_1 is not $\mathcal{A}$-relevant, we can easily conclude that $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_1$. In the case of $\mathcal{C}_2 = \{r_2\}$, where r_2 is $\mathcal{A}$-relevant, we consider the (only) linear constraint associated with r_2, which is $\overline{\alpha}_p \cdot (1+x, y) \geq \overline{\alpha}_p \cdot (x, 1+y)$. Given $\overline{\alpha}_p = (1, 1)$, the constraint is satisfied for all $x, y \in \mathbb{N}$, and since $\overline{\alpha}_p[1] \neq 0$, then $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted also in $\mathcal{C}_2$.

We now just need to check if for every SCC $\mathcal{C}_l$ such that $\mathtt{p} \in pred(\mathcal{C}_l)$, $\mathtt{p}[1/l]$ only depends on extended arguments $\mathtt{q}[j/k]$ such that $\mathtt{q}[j]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_k$. Considering $\mathcal{C}_1$, we have that $\mathtt{p}[1/1]$ depends only on itself (see Figure 1). Concerning $\mathcal{C}_2$, we have that $\mathtt{p}[1/2]$ depends on itself and $\mathtt{p}[1/1]$. Since $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in both $\mathcal{C}_1$ and $\mathcal{C}_2$, we can conclude that $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{P}_3$.

Likewise, it can be easily verified that all other arguments of $\mathcal{P}_3$ are $\mathcal{A}$-size-restricted in $\mathcal{P}_3$ as well. □

Theorem 1 *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Every $\mathcal{A}$-size-restricted argument of $\mathcal{P}$ is limited. If $\mathcal{P}$ is $\mathcal{A}$-size-restricted then it is limited.*

4 Complexity and Expressivity

In this section, we provide results on the complexity and the expressivity of the class of $\mathcal{A}$-size-restricted programs.

We start by showing that checking if an argument is $\mathcal{A}$-size-restricted in an SCC is in NP.

Theorem 2 *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Given an SCC $\mathcal{C}$ of $\mathcal{P}$, checking whether an argument of $\mathcal{C}$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ is in NP.*

From the theorem above, we obtain that checking whether a program is $\mathcal{A}$-size-restricted is in NP.

Theorem 3 *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Checking whether (an argument of) $\mathcal{P}$ is $\mathcal{A}$-size-restricted (in $\mathcal{P}$) is in NP.*

We use AR, BP, RB, and SR to denote, respectively, the set of all argument-restricted [Lierler and Lifschitz, 2009], bounded [Greco *et al.*, 2013a], rule-bounded [Calautti *et al.*, 2014a], and $\emptyset$-size-restricted programs. Moreover, given two sets A and B, we use $A \nparallel B$ as a shorthand for $A \not\subseteq B \wedge B \not\subseteq A$. The following theorem compares our approach with well-known terminating classes previously proposed.

Theorem 4 *$AR \nparallel SR$, $RB \subsetneq SR$, and $BP \nparallel SR$*

Note that our technique strictly generalizes the rule-bounded criterion even when the set of limited arguments $\mathcal{A}$ is empty. Looking at the size of parts of multiple atoms, as opposed to the entire size of a single atom like the rule-bounded criterion does, allows our criterion to include more programs.

By combining our technique with the argument-restricted or bounded criterion we can recognize more limited programs than by using any of them alone. We use $AR + SR$ (resp. $BP + SR$) to denote the set of all $\mathcal{A}$-size-restricted programs where, for each program, $\mathcal{A}$ is the set of its argument-restricted (resp. bounded) arguments.

Corollary 1 $AR \subsetneq AR + SR$, $SR \subsetneq AR + SR$, $BP \subsetneq BP + SR$, $SR \subsetneq BP + SR$.

5 Iterated Criterion

The size-restricted technique presented in the previous section starts from a (possibly empty) set of limited arguments $\mathcal{A}$ and gives as output a new set of limited arguments $\mathcal{A}'$. The question is whether the technique, starting from the resulting set of limited arguments $\mathcal{A}'$, could compute a new set of limited arguments $\mathcal{A}'' \supset \mathcal{A}'$. As shown by the next example, the answer is positive and thus our technique can benefit from an iterative application of itself.

Example 9 Consider the following program $\mathcal{P}_9$.

$$\mathtt{p}(\mathtt{f}(\mathtt{X}), \mathtt{f}(\mathtt{Y})) \quad \leftarrow \quad \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{b}(\mathtt{X}).$$

The program has only one SCC consisting of the rule above. Assume that $\mathcal{A} = \emptyset$. By choosing the first body atom of the rule, we get the following inequality:

$$\overline{\alpha}_p \cdot (x, y) \quad \geq \quad \overline{\alpha}_p \cdot (x{+}1, y{+}1)$$

The vectors $\overline{\alpha}_p = (0, 0)$ and $\overline{\alpha}_b = (1)$ satisfy the conditions of Definition 4. Therefore, the resulting set of $\mathcal{A}$-size-restricted arguments is $\mathcal{A}' = \{\mathtt{b}[1]\}$.

Now, considering $\mathcal{A}'$ as the starting set of limited arguments, we determine that $\mathtt{p}[1]$ is limited too, by Condition 1 of Definition 4. The new set of limited arguments is $\mathcal{A}'' = \mathcal{A}' \cup \{\mathtt{p}[1]\}$. Finally, considering the vectors $\overline{\alpha}_p = (-1, 1)$ (recall that $\mathtt{p}[1] \in \mathcal{A}''$) and $\overline{\alpha}_b = (0)$, the constraint is satisfied for all non-negative values of its integer variables. Then, $\mathtt{p}[2]$ is limited, $\mathcal{A}''' = \mathcal{A}'' \cup \{\mathtt{p}[2]\}$, and hence $\mathcal{P}_9$ is limited. □

Thus, we introduce a simple operator that iteratively applies the size-restricted criterion by using at each iteration the limited arguments derived at previous iterations.

Definition 7 Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. We define the operator $\Psi_{\mathcal{P}}(\mathcal{A}) = \mathcal{A} \cup \mathcal{R}_{\mathcal{A}}(\mathcal{P})$. For $i \geq 1$, we define the i-th iteration of $\Psi_{\mathcal{P}}$ as follows:

$$\Psi^1_{\mathcal{P}}(\mathcal{A}) = \Psi_{\mathcal{P}}(\mathcal{A})$$
$$\Psi^{i+1}_{\mathcal{P}}(\mathcal{A}) = \Psi_{\mathcal{P}}(\Psi^i_{\mathcal{P}}(\mathcal{A})), \quad \text{for } i > 1.$$

Obviously, $\Psi^i_{\mathcal{P}}(\mathcal{A}) \subseteq \Psi^{i+1}_{\mathcal{P}}(\mathcal{A})$ for every $i \geq 1$ and since the number of arguments of $\mathcal{P}$ is finite, then there always exists a finite $n \leq |args(\mathcal{P})|$ such that $\Psi^n_{\mathcal{P}}(\mathcal{A}) = \Psi^{n+1}_{\mathcal{P}}(\mathcal{A})$; we denote $\Psi^n_{\mathcal{P}}(\mathcal{A})$ as $\Psi^\infty_{\mathcal{P}}(\mathcal{A})$.

Corollary 2 *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Every argument in $\Psi^\infty_{\mathcal{P}}(\mathcal{A})$ is limited.*

6 Conclusion

In this paper, we have proposed a novel class of logic programs with function symbols whose bottom-up evaluation always terminates. Our technique identifies programs that are not captured by any of the current approaches and can be combined with them to recognize even more programs.

Interesting directions for future work are to plug termination criteria in the framework proposed in [Eiter *et al.*, 2013] and study their combination in such a framework, and analyze the relationships between the notions of safety of [Eiter *et al.*, 2013] and the notions of limitedness of termination criteria.

References

[Alviano *et al.*, 2010] M. Alviano, W. Faber, and N. Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *TPLP*, 10(4-6):497–512, 2010.

[Arts and Giesl, 2000] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1-2):133–178, 2000.

[Baselice *et al.*, 2009] S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. *TPLP*, 9(2):213–238, 2009.

[Bruynooghe *et al.*, 2007] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM TOPLAS*, 29(2), 2007.

[Calautti *et al.*, 2013] M. Calautti, S. Greco, and I. Trubitsyna. Detecting decidable classes of finitely ground logic programs with function symbols. In *PPDP*, 2013.

[Calautti *et al.*, 2014a] M. Calautti, S. Greco, C. Molinaro, and I. Trubitsyna. Checking termination of logic programs with function symbols through linear constraints. In *RuleML*, pages 97–111, 2014.

[Calautti *et al.*, 2014b] M. Calautti, S. Greco, F. Spezzano, and I. Trubitsyna. Checking termination of bottom-up evaluation of logic programs with function symbols. *TPLP*, 2014.

[Calimeri *et al.*, 2008] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: Theory and implementation. In *ICLP*, pages 407–424, 2008.

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

[De Schreye and Decorte, 1994] D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *JLP*, 19/20:199–260, 1994.

[Eiter and Simkus, 2010] T. Eiter and M. Simkus. Fdnc: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM TOCL*, 11(2), 2010.

[Eiter *et al.*, 2013] T. Eiter, M. Fink, T. Krennwallner, and C. Redl. Liberal safety for answer set programs with external sources. In *AAAI*, 2013.

[Endrullis *et al.*, 2008] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2-3):195–220, 2008.

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

[Gebser *et al.*, 2012] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on AI and ML. Morgan & Claypool Publishers, 2012.

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

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *NGC*, 9(3/4):365–386, 1991.

[Greco *et al.*, 2011] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB*, 4(11):1158–1168, 2011.

[Greco *et al.*, 2012] S. Greco, C. Molinaro, and F. Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[Greco *et al.*, 2013a] S. Greco, C. Molinaro, and I. Trubitsyna. Bounded programs: A new decidable class of logic programs with function symbols. In *IJCAI*, 2013.

[Greco *et al.*, 2013b] S. Greco, C. Molinaro, and I. Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *TPLP*, 13(4-5):737–752, 2013.

[Leuschel and Vidal, 2014] M. Leuschel and G. Vidal. Fast offline partial evaluation of logic programs. *I&C*, 235(0):70–97, 2014.

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

[Marchiori, 1996] M. Marchiori. Proving existential termination of normal logic programs. In *AMST*, 1996.

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

[Ohlebusch, 2001] E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *AAECC*, 12(1/2):73–116, 2001.

[Riguzzi and Swift, 2013] F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *TPLP*, 13(2):279–302, 2013.

[Riguzzi and Swift, 2014] F. Riguzzi and T. Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM TOCL*, 2014.

[Schneider-Kamp *et al.*, 2009] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM TOCL*, 11(1), 2009.

[Sohn and Gelder, 1991] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *ACM PODS*, pages 216–226, 1991.

[Sternagel and Middeldorp, 2008] C. Sternagel and A. Middeldorp. Root-labeling. In *RTA*, pages 336–350, 2008.

[Syrjanen, 2001] T. Syrjanen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.

[Verbaeten *et al.*, 2001] S. Verbaeten, D. De Schreye, and K. F. Sagonas. Termination proofs for logic programs with tabling. *ACM TOCL*, 2(1):57–92, 2001.

[Voets and De Schreye, 2011] D. Voets and D. De Schreye. Non-termination analysis of logic programs with integer arithmetics. *TPLP*, 11(4-5):521–536, 2011.