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,irina.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 $P_1$:

$$p(f(X, X), Y, Z) ← p(X, g(Z), g(Y)).$$

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 $P_2$:

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

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 $act((state), (symbol), (operation))$.

2833
Specifically, given the current parsing state \( \langle \text{state} \rangle \) and a symbol (\( \text{symbol} \)) to be parsed, \( \langle \text{operation} \rangle \) describes one of the following four parsing operations: shift(\( \text{newstate} \)), i.e., the next token is read from the input and pushed to the parsing stack along with the new parsing state \( \langle \text{newstate} \rangle \); reduce(\( A, B \)), i.e., there is a production rule \( A \rightarrow B \) in the grammar and the top of the parsing stack contains \( B \) (according to \( \langle \text{state} \rangle \)), which must be replaced with \( A \); goto(\( \text{newstate} \)), 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 \( \text{par}(\{a_1, \ldots, a_n, 0\}, \{s_0\}) \), where \( \{a_1, \ldots, a_n, 0\} \) is the input string, followed by the “end of string symbol” \( \$ \), and \( \{s_0\} \) is the parsing stack containing the initial state \( s_0 \). The string is accepted iff the program model contains two atoms of the form \( \text{par}(\{\$\}, \{s\}L) \) and \( \text{act}(s, \$, \text{accept}) \). □

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 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: \( \omega \)-restricted programs [Syriyanov, 2001], \( \lambda \)-restricted programs [Gebser et al., 2007], finite domain programs [Calimeri et al., 2008], argument-restricted programs [Lierler and Lifschitz, 2009], safe and \( \Gamma \)-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 \( p(X) \leftarrow p(X) \) 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 been 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.
\section{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
\[
A_{1} \lor ... \lor A_{m} \leftarrow B_{1}, ..., B_{k}, \neg C_{1}, ..., \neg C_{n}
\]
where \( m > 0 \), \( k \geq 0 \), \( n \geq 0 \), and \( A_{1}, A_{n}, B_{1}, ..., B_{k}, C_{1}, ..., C_{n} \) are atoms. The disjunction \( A_{1} \lor ... \lor A_{m} \) 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 \( \theta \) 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 \( \theta \) 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 \( \theta \). Two atoms \( A_{1} \) and \( A_{2} \) unify if there exists a substitution \( \theta \) 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., \( ground(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} ground(r) \).

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 \( A \not\in I \), false otherwise. A ground rule \( r \) is satisfied by \( I \), denoted \( I \models r \), if there is a ground literal 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 \( \mathcal{M} \) of \( \mathcal{P} \) is minimal if no proper subset of \( \mathcal{M} \) is a model of \( \mathcal{P} \). The set of minimal models of \( \mathcal{P} \) is denoted by \( MM(\mathcal{P}) \). 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 \( I \in MM(\mathcal{P}^{I}) \). The set of stable models of \( \mathcal{P} \) is denoted by \( SM(\mathcal{P}) \). It is well known that \( SM(\mathcal{P}) \subseteq MM(\mathcal{P}) \), and \( SM(\mathcal{P}) = MM(\mathcal{P}) \) for positive programs. A positive normal program \( \mathcal{P} \) has a unique minimal model, which we denote as \( MM(\mathcal{P}) \).

**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.

\section{Size-Limited 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 \( A_{1} \lor ... \lor A_{m} \leftarrow body \) 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 $(r, r')$ iff there exist two (not necessarily distinct) rules $r, r' \in \mathcal{P}$ s.t. $\text{head}(r)$ and an atom in $\text{body}(r')$ unify. □

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

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 $\text{head}(r)$ contains at least one variable which does not appear in $\text{body}(r) \setminus \text{rbody}(r)$ 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 $\mathcal{P}_3$:

\[
\begin{align*}
   r_1 & : p(f(X), Y) & \rightarrow & & p(X, f(Y)), b(X, Z), \\
   r_2 & : p(X, g(Y)) & \rightarrow & & p(f(X), Y).
\end{align*}
\]

The firing graph of $\mathcal{P}_3$ has the edges $(r_1, r_1)$, $(r_2, r_2)$, and $(r_1, r_2)$. 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 $\mathcal{A} = \{p[2]\}$, 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 $C$ occur in the body respectively in the non-mutually recursive atom $C$ and inside $p[2] \in \mathcal{A}$ 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 $\mathbf{v} = (v_1, ..., v_k)$ and $\mathbf{w} = (w_1, ..., w_k)$ in $\mathbb{Z}^k$, we use $\mathbf{v} \cdot \mathbf{w}$ to denote the classical scalar product, that is, $\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{k} v_i \cdot w_i$. We also use the notation $\mathbf{v}[i]$ to refer to $v_i$, for $1 \leq i \leq k$.

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

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

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

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

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 $p[i] \in \text{args}(\mathcal{P})$ w.r.t. $\mathcal{A}$, denoted $\text{D}_\mathcal{A}(p[i])$, is $\mathbb{Z}$ if $p[i] \in \mathcal{A}$, and $\mathbb{N}$ otherwise. The domain of a predicate symbol $p$ of arity $n$ is $\text{D}_\mathcal{A}(p) = \text{D}_\mathcal{A}(p[1]) \times \cdots \times \text{D}_\mathcal{A}(p[n])$. □

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 $\mathbb{C}$ be an SCC of $\mathcal{P}$ with $\text{pred}(\mathbb{C}) = \{p_1, ..., p_n\}$. We say that an argument $p_i[j]$ of $\mathbb{C}$ is $\mathcal{A}$-size-restricted in $\mathbb{C}$ iff

1. for every rule $r \in \mathbb{C}$ such that $\text{head}(r) = p_i(t_1, ..., t_m)$ 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 $\text{size}(t_k) \leq \text{D}_\mathcal{A}(p_i[j])$;
2. there exist $n$ vectors $\mathbf{s}_h \in \text{D}_\mathcal{A}(p_h), 1 \leq h \leq n$, such that for every $\mathcal{A}$-relevant rule $r \in \mathbb{C}$ there exists an atom $B$ in $\text{body}(r)$ such that if $\text{pr}(\text{head}(r)) = p_k$ and $\text{pr}(B) = p_i$, then the following conditions hold:
   - (a) the constraint
     \[
     \mathbf{s}_i \cdot \text{size}(B) \geq \mathbf{s}_k \cdot \text{size}(\text{head}(r))
     \]
     is satisfied for every non-negative value of the integer variables in it; and
that an argument \(p_i[j]\) in \(A\)-size-restricted because for every rule of \(C\) 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 \(A\)-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 \(\pi_i[j] = 0\), 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 \(\pi_k[1] = 0\), 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 \(\pi\). 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 \(P_1\) of Example 1, reported below:

\[
p(f(X, Y), X, Y) \leftarrow p(X, g(Z), g(Y)).
\]

Let us consider \(A = \emptyset\). The program has only one SCC \(C\) consisting of the rule above, which is \(A\)-relevant. The vector \(\pi_p = (0, 1, 1)\) allows us to say that all arguments are \(A\)-size-restricted in \(C\). In fact, when arguments \(p[2]\) and \(p[3]\) 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 \(\pi_p[2]\) and \(\pi_p[3]\) are not 0. When argument \(p[1]\) 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 \(P_2\) of Example 2, which has only one SCC \(C\) coinciding with \(P_2\) itself. Let us consider \(A = \emptyset\). All rules are \(A\)-relevant. We now show that every argument is \(A\)-size-restricted in \(C\). In particular, consider the inequalities associated with the rules of \(P_2\) 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:

\[
\begin{align*}
\pi_{act}(st, sym, 1 + s_1) &\geq \pi_{par}(6 + s_1 + sym + st + l) \\
\pi_{act}(st, sym, 2 + a + b) &\geq \pi_{red}(2 + sym + l, 2 + st + l, a, b) \\
\pi_{red}(i, 4 + s + x + l, a, 2 + y + t) &\geq \pi_{red}(i, l, a, t) \\
\pi_{act}(st, a, 1 + s_1) &\geq \pi_{par}(6 + s_1 + a + st + l)
\end{align*}
\]

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

\[
\begin{align*}
st + sym + s_1 + 1 &\geq 0 \\
st + sym + a + b + 2 &\geq a + b \\
a + y + t + 2 &\geq a + t \\
st + a + s_1 + 1 &\geq 0
\end{align*}
\]

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 \(\pi_{act}[1], \pi_{act}[2], \pi_{act}[3], \pi_{red}[3]\), and \(\pi_{red}[4]\) are all different from 0, we can say that arguments \(act[1], act[2], act[3], red[3]\), and \(red[4]\) are \(A\)-size-restricted in \(C\), as Condition 2(b) is also satisfied. For arguments \(par[1], par[2], red[1],\) and \(red[2]\) (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 \(par[1], par[2], red[1],\) and \(red[2]\) are \(A\)-size-restricted in \(C\).

We now define how to determine if an argument is \(A\)-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 \(P\), we assume an arbitrary but fixed numbered \(C_1, \ldots, C_n\) of its SCCs. We also define \(ex-args(P)\) as the set \{\(p[i]/j\) | \(C_j\) is an SCC of \(P\) and \(p[i] \in \text{args}(C_j)\}\). Each element of \(ex-args(P)\) is called an extended argument of \(P\). 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 \(P\), denoted \(\Delta(P)\), is a directed graph whose set of nodes is \(ex-args(P)\) and where there is an edge \((q[j]/k), p[i]/l)\) iff

- \(k = l\) and there is a rule \(r \in C_k\) 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 \(r' \in P\) such that \(head(r')\) and \(B\) unify; or
- \(k \neq l\) and \(p = q, i = j,\) and there are two rules \(r_1 \in C_k\) and \(r_2 \in C_l\) such that \(pr(head(r_1)) = p\) and \((r_1, r_2)\) is an edge of \(\Omega(P)\).

Intuitively, an edge \((q[j]/k), p[i]/l)\) of \(\Delta(P)\) means that there can be a propagation of terms from \(q[j]\) in component \(C_k\) to \(p[i]\) in component \(C_l\). 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 \(\Delta(P)\).

**Example 7** Consider again program \(P_3\) of Example 3. Figure 1 illustrates \(\Delta(P_3)\).

![Figure 1: Extended argument graph of \(P_3\).](image-url)
We are now ready to define when an argument is \(A\)-size-restricted in a program.

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

1. \(p[j]\) is \(A\)-size-restricted in \(C\), and
2. \(p[j/k]\) depends only on extended arguments \(q[j/k]\) such that \(q[j]\) is \(A\)-size-restricted in \(C\).

We denote by \(R_A(P)\) the set of all \(A\)-size-restricted arguments in \(P\). We say that \(P\) is \(A\)-size-restricted iff \(\text{args}(P) = A \cup R_A(P)\). \(\square\)

**Example 8** Consider program \(P_3\) of Example 3, whose extended argument graph is shown in Figure 1 and let \(A = \{p[2]\}\). Below we show that \(p[1]\) is \(A\)-size-restricted in \(P_3\).

Since \(P \in \text{pred}(C_1)\) and \(p \in \text{pred}(C_2)\), we first need to check if \(p[1]\) is \(A\)-size-restricted in \(C_1\) and \(C_2\). Since \(C_1 = \{r_1\}\) and \(r_1\) is not \(A\)-relevant, we can easily conclude that \(p[1]\) is \(A\)-size-restricted in \(C_1\). In the case of \(C_2 = \{r_2\}\), where \(r_2\) is \(A\)-relevant, we consider the (only) linear constraint associated with \(r_2\), which is \(\pi_{p} \cdot (1 + x, y) \geq \pi_{p} \cdot (x, 1 + y)\). Given \(\pi_{p} = (1, 1)\), the constraint is satisfied for all \(x, y \in \mathbb{N}\), and since \(\pi_{p}[1] \neq 0\), then \(p[1]\) is \(A\)-size-restricted also in \(C_2\).

We now just need to check if for every SCC \(C\) such that \(p \in \text{pred}(C)\), \(p[1/l]\) only depends on extended arguments \(q[j/k]\) such that \(q[j]\) is \(A\)-size-restricted in \(C\). Considering \(C_1\), we have that \(p[1/1]\) depends only on itself (see Figure 1). Concerning \(C_2\), we have that \(p[1/2]\) depends on itself and \(p[1/1]\). Since \(p[1]\) is \(A\)-size-restricted in both \(C_1\) and \(C_2\), we can conclude that \(p[1]\) is \(A\)-size-restricted in \(P_3\).

Likewise, it can be easily verified that all other arguments of \(P_3\) are \(A\)-size-restricted in \(P_3\) as well. \(\square\)

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

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

From the theorem above, we obtain that checking whether a program is \(A\)-size-restricted is in \(NP\).

**Theorem 3** Let \(P\) be a program and \(A\) be a set of limited arguments of \(P\). Checking whether (an argument of) \(P\) is \(A\)-size-restricted (in \(P\)) 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 \(\phi\)-size-restricted programs. Moreover, given two sets \(A\) and \(B\), we use \(A \upharpoonright B\) as a shorthand for \(A \subseteq B \wedge B \subseteq A\).

The following theorem compares our approach with well-known terminating classes previously proposed.

**Theorem 4** \(AR \upharpoonright SR, RB \subseteq SR,\) and \(BP \upharpoonright SR\)

Note that our technique strictly generalizes the rule-bounded criterion even when the set of limited arguments \(A\) 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 \(A\)-size-restricted programs where, for each program, \(A\) is the set of its argument-restricted (resp. bounded) arguments.

**Corollary 1** \(AR \subseteq AR + SR, SR \subseteq AR + SR,\) \(BP \subseteq BP + SR, SR \subseteq BP + SR\).

5 Iterated Criterion

The size-restricted technique presented in the previous section starts from a (possibly empty) set of limited arguments \(A\) and gives as output a new set of limited arguments \(A'\). The question is whether the technique, starting from the resulting set of limited arguments \(A'\), could compute a new set of limited arguments \(A'' \supset A'\). 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 \(P_3\).

\[p(f(X), f(Y)) \leftarrow p(X, Y), b(X).\]

The program has only one SCC consisting of the rule above.

To obtain the first body atom of the rule, we get the following inequality:

\[\pi_{p} \cdot (x, y) \geq \pi_{p} \cdot (x+1,y+1)\]

The vectors \(\pi_{p} = (0,0)\) and \(\pi_{b} = (1)\) satisfy the conditions of Definition 4. Therefore, the resulting set of \(A\)-size-restricted arguments is \(A' = \{b[1]\}\).

Now, considering \(A'\) as the starting set of limited arguments, we determine that \(p[1]\) is limited too, by Condition 1 of Definition 4. The new set of limited arguments is \(A'' = A' \cup \{p[1]\}\). Finally, considering the vectors \(\pi_{p} = (-1, 1)\) (recall that \(p[1] \in A''\)) and \(\pi_{b} = (0)\), the constraint is satisfied for all non-negative values of its integer variables. Then, \(p[2]\) is limited, \(A''' = A'' \cup \{p[2]\}\), and hence \(P_3\) is limited. \(\square\)

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 \(P\) be a program and \(A\) be a set of limited arguments of \(P\). We define the operator \(\Psi_{\phi}(A) = A \cup R_{A}(P)\).

For \(i \geq 1\), we define the \(i\)-th iteration of \(\Psi_{\phi}\) as follows:

\[\Psi_{\phi}^{i}(A) = \Psi_{\phi}(A) \Psi_{\phi}^{i+1}(A) = \Psi_{\phi}(\Psi_{\phi}(A)), \quad \text{for } i > 1.\]

Obviously, \(\Psi_{\phi}^{i}(A) \subseteq \Psi_{\phi}^{i+1}(A)\) for every \(i \geq 1\) and since the number of arguments of \(P\) is finite, then there always exists a finite \(n \leq |\text{args}(P)|\) such that \(\Psi_{\phi}^{n}(A) = \Psi_{\phi}^{n+1}(A)\); we denote \(\Psi_{\phi}^{n}(A)\) as \(\Psi_{\phi}^{\infty}(A)\).

**Corollary 2** Let \(P\) be a program and \(A\) be a set of limited arguments of \(P\). Every argument in \(\Psi_{\phi}^{\infty}(A)\) 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