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

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

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

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

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

Example 1 Consider the logic program $P_1$ below:

$$\begin{align*}
  r_0 & : \text{count}([a, b, c, 0]). \\
  r_1 & : \text{count}(L, I + 1) \leftarrow \text{count}(X[L], I).
\end{align*}$$

The bottom-up evaluation of $P_1$ terminates yielding the set of atoms $\text{count}([a, b, c, 0])$, $\text{count}([b, c, 1])$, $\text{count}([c, 2])$, and $\text{count}([], 3)$. The query goal $\text{count}([], L)$ can be used to retrieve the length $L$ of list $[a, b, c]$.\footnote{Notice that $P_1$ has been written so as to count the number of elements in a list when evaluated in a bottom-up fashion, and therefore differs from the classical formulation relying on a top-down evaluation strategy. However, as shown in Example 2, programs relying on a top-down evaluation strategy can be rewritten into programs whose bottom-up evaluation gives the same result. Our criterion can then be applied to rewritten programs.}

None of the current termination criteria is able to realize that the bottom-up evaluation of $P_1$ terminates. Intuitively, this is because they analyze how the first argument of count affects itself and how the second argument of count affects itself, but miss noticing that the growth of latter is bounded by the reduction of the former. One of the novelties of our criterion is the capability of doing this kind of reasoning—indeed, as shown in Example 8, our criterion is able to realize that the bottom-up evaluation of $P_1$ terminates.

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

Example 2 Consider the following logic program $P_2$:

$$\begin{align*}
  \text{append}([], L, L).
  \text{append}(X[L1], L2, [X[L3]]) & \leftarrow \text{append}(L1, L2, L3).
\end{align*}$$

As an example, the top-down evaluation of $P_2$ w.r.t. the query goal $\text{append}([a, b], [c, d, w])$ gives the concatenation of the lists $[a, b]$ and $[c, d]$. This query cannot be evaluated by standard bottom-up evaluators as the second rule in $P_2$ is not range restricted (because variable $X$ in the rule head does not appear in a positive body literal). However, one might apply rewriting techniques such as magic-set to rewrite $P_2$ into a program whose bottom-up evaluation gives the same result [Beeri and Ramakrishnan, 1991]. In this case, $P_2$ and the
query goal \texttt{append\([\{a, b\}, \{c, d\}, \emptyset]\)} give the following program \(P_2^\prime\):

\[
\begin{align*}
r_0 & : \text{magic\_append\([\{a, b\}, \{c, d\}\])}. \\
r_1 & : \text{magic\_append\([L_1, L_2]\) } \leftarrow \text{magic\_append\([X[L_1], L_2]\)}. \\
r_2 & : \text{append\([\emptyset, L_1]\) } \leftarrow \text{magic\_append\([\emptyset, L]\)}. \\
r_3 & : \text{append\([X[L_1], L_2, \{X[L_3]\}\]) } \leftarrow \text{magic\_append\([X[L_1], L_2]\), append\([L_1, L_2, L_3]\)].}
\end{align*}
\]

Also in the example above, current termination criteria are not able to recognize termination of the bottom-up evaluation of \(P_2^\prime\), while the new criterion introduced in this paper does.

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

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

## 2 Logic Programs with Function Symbols

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

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

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

A rule \(r\) is of the form:

\[A_1 \lor \ldots \lor A_m \leftarrow B_1, \ldots, B_k, \neg C_1, \ldots, \neg C_n\]

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

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

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

**Semantics.** Consider a program \(P\). The Herbrand universe \(H_P\) of \(P\) is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in \(P\). The Herbrand base \(B_P\) of \(P\) is the set of ground atoms which can be built using predicate symbols appearing in \(P\) and ground terms of \(H_P\). A rule \(r'\) is a ground instance of a rule \(r\) in \(P\) if \(r'\) can be obtained from \(r\) by substituting every variable in \(r\) with some ground term in \(H_P\). We use \(ground(r)\) to denote the set of all ground instances of \(r\) and \(ground(P)\) to denote the set of all ground instances of the rules in \(P\), i.e., \(ground(P) = \bigcup_{r \in p} ground(r)\). An interpretation of \(P\) is any subset \(I\) of \(B_P\). The truth value of a ground atom \(A\) w.r.t. \(I\), denoted \(value_1(A)\), is true if \(A \in I\), false otherwise. The truth value of \(\neg A\) w.r.t. \(I\), denoted \(value_1(\neg A)\), is true if \(A \notin I\), false otherwise. The truth value of a conjunction of ground literals \(C = L_1, \ldots, L_n\) w.r.t. \(I\) is \(value_1(C) = \min\{value_1(L_i) \mid 1 \leq i \leq n\}\)—here the ordering false < true holds—whereas the truth value of a disjunction of ground literals \(D = L_1 \lor \ldots \lor L_n\) w.r.t. \(I\) is \(value_1(D) = \max\{value_1(L_i) \mid 1 \leq i \leq n\}\); if \(n = 0\), then \(value_1(C) = \text{true} \) and \(value_1(D) = \text{false}\). A ground rule \(r\) is satisfied by \(I\), denoted \(I \models r\), if \(value_1(head(r)) \geq value_1(body(r))\); we write \(I \not\models r\) if \(r\) is not satisfied by \(I\). Thus, a ground rule \(r\) with empty body is satisfied by \(I\) if

\(^2\)Database facts are not shown in our examples as they are not relevant for the proposed criterion.
value₁(head(r)) = true. An interpretation of \( \mathcal{P} \) is a model of \( \mathcal{P} \) if it satisfies every ground rule in ground(\( \mathcal{P} \)). A model \( M \) of \( \mathcal{P} \) is minimal if no proper subset of \( M \) is a model of \( \mathcal{P} \).

The set of minimal models of \( \mathcal{P} \) is denoted by \( \text{MM}(\mathcal{P}) \).

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

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

3 Termination Analysis Tools

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

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

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

**Example 3** Consider the following program \( \mathcal{P}_3 \):

\[
\begin{align*}
  r_0 & : \text{nat}(X) \leftarrow b(X). \\
  r_1 & : \text{next}(f(X)) \leftarrow \text{nat}(X). \\
  r_2 & : \text{nat}(X) \leftarrow \text{next}(f(X)).
\end{align*}
\]

where \( b \) is a base predicate symbol.

It is easy to see that \( r_0 \) activates \( r_1 \), but not vice versa; \( r_1 \) activates \( r_2 \) and vice versa.

The activation graph of \( \mathcal{P}_3 \) is shown in Figure 1 (left).

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

A variation of the argument graph, called propagation graph, has been proposed in [Lierler and Lifschitz, 2009] to define the class \( \mathcal{AP} \) of \( \Gamma \)-acyclic programs. The main difference is that in the propagation graph only arguments where complex terms may be introduced or propagated are considered.

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

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

\[
\begin{align*}
  d(X, X) &= 0, \\
  d(X, f(t_1, ..., t_m)) &= 1 + \max_{i : t_i \text{ contains } X} d(X, t_i).
\end{align*}
\]

An argument ranking for a program \( \mathcal{P} \) is a partial function \( \phi \) from \( \text{arg}(\mathcal{P}) \) to non-negative integers such that, for every rule \( r \) of \( \mathcal{P} \), every atom \( A \) occurring in the head of \( r \), and every variable \( X \) occurring in a term \( A^1 \), if \( \phi(A^0[i]) \) is defined, then body⁺(\( r \)) contains an atom \( B \) such that \( X \) occurs in a term \( B^1 \), \( \phi(B^0[j]) \) is defined, and the following condition is satisfied:

\[
\phi(A^0[i]) - \phi(B^0[j]) \geq d(X, A^1) - d(X, B^1).
\]

The set of restricted arguments of \( \mathcal{P} \) is \( \mathcal{AR}(\mathcal{P}) = \{ p[i] | p[i] \in \text{arg}(\mathcal{P}) \wedge \exists \phi s.t. \phi(p[i]) \text{ is defined} \} \).

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

Argument restricted programs strictly include \( \omega \)-restricted, \( \lambda \)-restricted, and finite domain programs.

4 Bounded Programs

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

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

We start by introducing a new graph derived from the argument graph by labelling edges with additional information.
the indication of the start edge is not relevant, we will call a cyclic path a cycle. Given a cycle \( \pi \) consisting of \( n \) labelled edges \( e_1, \ldots, e_n \), we can derive \( n \) different cyclic paths starting from each of the \( e_i \)’s—we use \( \tau(\pi) \) to denote the set of such cyclic paths. As an example, if \( \pi \) is a cycle consisting of labelled edges \( e_1, e_2, e_3 \), then \( \tau(\pi) = \{ (e_1,e_2,e_3), (e_2,e_3,e_1), (e_3,e_1,e_2) \} \). Given two cycles \( \pi_1 \) and \( \pi_2 \), we write \( \pi_1 \approx \pi_2 \) iff \( \exists \rho_1 \in \tau(\pi_1) \) and \( \exists \rho_2 \in \tau(\pi_2) \) such that \( \lambda_3(\rho_1) = \lambda_3(\rho_2) \). A cycle is basic if it does not contain two occurrences of the same edge.

We say that a node \( p[i] \) of \( G_L(P) \) depends on a node \( q[j] \) of \( G_L(P) \) iff there is a path from \( q[j] \) to \( p[i] \) in \( G_L(P) \). Moreover, we say that \( p[i] \) depends on a cycle \( \pi \) iff it depends on a node \( q[j] \) appearing in \( \pi \). Clearly, nodes belonging to a cycle \( \pi \) depend on \( \pi \). We say that \( \lambda_2(\rho) = r_1, \ldots, r_m \) denotes a cyclic path in the activation graph \( \Omega(P) \) if \( (r_1, r_2, \ldots, (r_m-1, r_m, (r_m, r_1)) \) are edges of \( \Omega(P) \).

Definition 2 (Active cycle) Given a program \( P \), we say that a cycle \( \pi \) in \( G_L(P) \) is active iff \( \exists \rho \in \tau(\pi) \) such that \( \lambda_2(\rho) \) denotes a cyclic path in the activation graph \( \Omega(P) \).

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

Example 6 Consider the labelled argument graph and the activation graph of program \( P_5 \) (cf. Figure 2). Cycles \( \pi_1 \) and \( \pi_3 \) in Figure 2 are active as \( (r_1, r_2), (r_2, r_1) \) is a cyclic path in \( \Omega(P_5) \). On the contrary, it is easy to check that cycles \( \pi_2 \) and \( \pi_4 \) in Figure 2, are not active.

In the previous example, the labelled edges of the non-active cycle \( \pi_4 \) say that a complex term with function symbol \( f \) might be generated from \( a[2] \) to \( q[2] \) (using rule \( r_3 \)), be propagated from \( q[2] \) to \( a[2] \) (using rule \( r_4 \)), and so on so forth possibly causing the generation of complex terms of unbounded size. However, in order for this to happen, rules \( r_3 \) and \( r_4 \) should activate each other, which is not the case from an analysis of the activation graph. Thus, we can conclude that the generation of unbounded terms cannot really happen.

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

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

---

1. We assume that literals in the head (resp. body) are ordered with the first one being associated with 1, the second one with 2, etc.
• \( S \rightarrow S_1 f_1 S_2 \), \( \forall f_i \in F_P \);
• \( S_1 \rightarrow f_1 S_1 \bar{f} S_1 | \varepsilon \), \( \forall f_i \in F_P \);
• \( S_2 \rightarrow (S_1 | f_i) S_2 | \varepsilon \), \( \forall f_i \in F_P \).

and \( R' \) is the set of production rules:

• \( S \rightarrow f_1 S \bar{f} S | \varepsilon \), \( \forall f_i \in F_P \).

The languages \( L(\Gamma_P) \) and \( L(\Gamma_P') \) are the sets of strings generated by \( \Gamma_P \) and \( \Gamma_P' \), respectively.

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

**Definition 4 (Classification of cycles)** Given a program \( P \) and a cycle \( \pi \) in \( G_L(P) \), we say that \( \pi \) is

• growing if there is \( \rho \in \tau(\pi) \) s.t. \( \lambda_1(\rho) \in L(\Gamma_P) \),
• balanced if there is \( \rho \in \tau(\pi) \) s.t. \( \lambda_1(\rho) \in L(\Gamma_P') \),
• failing otherwise.

Consider the labelled argument graph in Figure 2. Cycles \( \pi_1, \pi_3, \) and \( \pi_4 \) are growing, whereas cycle \( \pi_2 \) is failing. Observe that, in general, a failing cycle is not active, but the vice versa is not true. In fact, cycle \( \pi_4 \) from Figure 2 is not active even if it is not failing.

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

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

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

Observe that base arguments belong to \( \Psi_P(A) \) for any set of arguments \( A \), because they do not depend on cycles.

**Proposition 1** Let \( P \) be a program and \( A \) a set of limited arguments of \( P \). Then, all arguments in \( \Psi_P(A) \) are limited.

**Example 7** Consider program \( P_5 \) of Example 5, whose labelled argument and activation graphs are reported in Figure 2. Notice that basic cycles \( \pi_1 \) and \( \pi_3 \) are growing and active, \( \pi_2 \) is failing and non-active, \( \pi_4 \) is growing and non-active. The basic cycle \( \pi_5 \) composed by \( \pi_1 \) and \( \pi_2 \) (denoted by a dashed line) is failing and non-active, and the basic cycle \( \pi_6 \) composed by \( \pi_3 \) and \( \pi_4 \) (denoted by a dashed line) is growing and non-active. Furthermore, base arguments \( a[1] \) and \( b[1] \) do not depend on any cycle: \( t[1], q[1], s[1] \) depend on \( \pi_1, \pi_2, \) and \( \pi_5; t[2], q[2], s[2] \) depend on \( \pi_3, \pi_4, \) and \( \pi_6 \).

By iteratively applying operator \( \Psi_P \) starting from \( \emptyset \) we have:

(i) \( A_1 = \Psi_P(\emptyset) = \{a[1], b[1]\} \);
(ii) \( A_2 = \Psi_P(A_1) = A_1 \cup \{t[1], q[1], s[1]\} \) as Condition 2 of Definition 5 applies to \( \pi_1 \), and Condition 1 applies to \( \pi_2 \) and \( \pi_5 \);
(iii) \( A_3 = \Psi_P(A_2) = A_2 \cup \{t[2], q[2], s[2]\} \) as Condition 1 applies to \( \pi_4 \) and \( \pi_6 \), and Condition 3 applies to \( \pi_3 \) (in fact, \( \pi_3 \approx \pi_5, \pi_1 \) is not balanced and goes only through nodes in \( A_3 \)).

Therefore, we derive that all arguments are limited.

**Proposition 2** Given a program \( P \) and a set \( A \) of limited arguments of \( P \), then:

1. \( \Psi_P(A) \subseteq \Psi_P^{i+1}(A) \) for \( i \geq 0 \), and
2. there is a finite \( n \) such that \( \Psi_P^n(A) = \Psi_P^\infty(A) \), where \( \Psi_P^0(A) = A \) and \( \Psi_P^{i+1}(A) = \Psi_P(\Psi_P^i(A)) \), for \( i \geq 0 \).

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

**Example 8** Consider program \( P_1 \) of Example 1. To comply with the conditions mentioned at the beginning of this section, \( P_1 \) is rewritten into the following program \( P_8 \):

\[
\begin{align*}
r_0 & : \text{count}(1c(a, 1c(b, 1c(c, nil))), 0). \\
r_1 & : \text{f-count}(X, L, l) \leftarrow \text{count}(1c(X, L, l)). \\
r_2 & : \text{count}(L, s(I, 1, I)) \leftarrow \text{f-count}(X, L, l).
\end{align*}
\]

Here function symbol \( 1c \) denotes the list constructor operator \( \text{"["} \), function symbol \( s \) denotes the sum operator, and \( \text{nil} \)
denotes the empty list “[ ]”. The labelled argument graph is shown in Figure 4, where c and fc stand for count and f-count, respectively. The activation graph is not shown, but it is easy to see that r1 activates r2 and vice versa; thus, both cycles π1 and π2 are active. Furthermore, π1 is failing and π2 is growing. Notice also that c[1], fc[1], and fc[2] depend on π1; c[2] and fc[3] depend on π2.

As π1 is failing, then \( A_1 = \Psi_{P_8}(\emptyset) = \{c[1], fc[1], fc[2]\} \) by Condition 1. Then, \( A_2 = \Psi_{P_8}(A_1) = A_1 \cup \{c[2], fc[3]\} \) by Condition 3 because π1 \( \approx \) π2. π1 is not balanced and goes through arguments c[1] and fc[2] which both belong to A1. It is worth noting that current termination criteria are not able to realize this kind of dependency between π1 and π2. As \( \Psi_{P_8}(A_2) \) gives A2, the fixpoint is reached with all arguments of \( P_8 \) belonging to \( \Psi_{P_8}(\emptyset) \). By Proposition 1, all arguments are limited, thus \( P_8 \) has a finite minimal model.

**Example 9** Consider program \( P_9 \) below derived from program \( P_8 \) of Example 2 by replacing rule \( r_3 \) of \( P_8 \) with the two rules \( r_3 \) and \( r_4 \) below (to satisfy the conditions stated at the beginning of this section):

\[
\begin{align*}
r_0 & : \text{magic_append}([a, b], [c, d]). \\
r_1 & : \text{magic_append}([L1, L2]) \leftarrow \text{magic_append}([X1[L1], L2]). \\
r_2 & : \text{append}([], L1, L2) \leftarrow \text{magic_append}([L1, L2]). \\
r_3 & : \text{f-magic_append}(X1[L1], L2) \leftarrow \text{magic_append}([X1[L1], L2]). \\
r_4 & : \text{append}([X1[L1], L2], [X3[L3]]) \leftarrow \text{f-magic_append}(X1[L1], L2), \text{append}(L1, L2, L3).
\end{align*}
\]

Figure 5 shows the labelled argument graph of \( P_9 \), where ma, a, and fma stand for magic_append, append and f-magic_append, respectively. As π1, π2, and π3 are not growing, then \( \Psi_{P_9}(\emptyset) = \{ma[1], ma[2], fma[1], fma[2], fma[3], a[2]\} \) by Condition 1. Since fma[1] and fma[2] belong to \( \Psi_{P_9}(\emptyset) \), then a[1] \( \in \Psi_{P_9}(\emptyset) \) by Condition 2. As π5 \( \approx \) π4, π5 is not balanced and goes only through a[1] which is in \( \Psi_{P_9}(\emptyset) \), then a[3] belongs to \( \Psi_{P_9}(\emptyset) \) by Condition 3. Since all arguments are in \( \Psi_{P_9}(\emptyset) \), then \( \Psi_{P_9}(\emptyset) = \Psi_{P_9}(\emptyset) \) and the program has a finite minimal model.

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

**Definition 6 (Bounded programs)** The set of bounded arguments of a program \( P \) is \( \Psi_{P}(AR(P)) \). We say that \( P \) is bounded iff all its arguments are bounded. The set of all bounded programs is denoted as \( BP \).

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

**Theorem 2 (Complexity)** The worst-case time complexity of checking whether a program \( P \) belongs to the class \( BP \) is \( O(k + n^3 + n \cdot m \cdot (s + m)) \), where

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

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

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

**Theorem 3** \( T \subseteq BP \subseteq FG \) for \( T \in \{AR, AP\} \).

**Definition 7 (Linear programs)** A program \( P \) is linear if each node in \( G_L(P) \) appears in at most one basic cycle.

For instance, \( P_8 \) and \( P_9 \) of Examples 8 and 9 are linear, while program \( P_5 \) of Example 5 is not linear.

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

5 Conclusions

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


