On the Termination of Logic Programs with Function Symbols

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna

DEIS – Università della Calabria, 87036 Rende, Italy
`{greco,fspezzano,irina}@deis.unical.it`

Abstract

Recently there has been an increasing interest in the bottom-up evaluation of the semantics of logic programs with complex terms. The main problem due to the presence of functional symbols in the head of rules is that the corresponding ground program could be infinite and that finiteness of models and termination of the evaluation procedure is not guaranteed. This paper introduces, by deeply analyzing program structure, new decidable criteria, called *safety* and Γ-*acyclicity*, for checking termination of logic programs with function symbols under bottom-up evaluation. These criteria guarantee that stable models are finite and computable, as it is possible to generate a finitely ground program equivalent to the source program. We compare new criteria with other decidable criteria known in the literature and show that the Γ-*acyclicity* criterion is the most general one. We also discuss its application in answering bound queries.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Logic Programming, Function Symbols, Bottom-up Execution, Program Termination, Stable Models

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.323

1 Introduction

Recently there has been an increasing interest in Answer Set Programming (ASP) with function symbols, and more in general on the bottom-up evaluation of the semantics of logic programs with complex terms [1, 2, 3]. Indeed, one of the main limitations of current ASP and datalog systems is the inability (or the limited power) to define programs with complex terms and function symbols [4, 5, 6, 7]. The main problem in extending logic programming under bottom-up evaluation with function symbols is that the corresponding ground program is infinite and that finiteness of models and termination of the evaluation procedure is not guaranteed.

The problem of checking whether the computation of a query terminates has been investigated since the beginning of logic programming. Most of the past work was devoted to the termination of programs under top-down evaluation or for SLD resolution [8, 9], although it also received a significant attention from the deductive database community [10]. The reason was that the only relevant logic programming implemented language was Prolog, which computes answers to queries using a specific SLDNF resolution algorithm. Recently, the attention has been concentrated on semantics which can be naturally computed by means of bottom-up evaluation, such as stable model semantics for programs with possibly unstratified negation, perfect model semantics for programs with stratified negation, and minimum model semantics for positive programs. The following example shows a very simple program which uses function symbols in rule heads and the problem is to decide if the fixpoint computation of the program terminates.

▶ **Example 1.** Consider the following logic program $\mathcal{P}_1$:

$r_1 : \mathtt{p(a, a)}.$
$r_2 : \mathtt{p(f(X), g(X))} \leftarrow \mathtt{p(X, X)}.$

The program has a unique minimal model $M_1 = \{\mathtt{p(a, a)}, \mathtt{p(f(a), g(a))}\}$, which can be computed using the classical bottom-up fixpoint algorithm. Current techniques are not able to establish (in advance, by analyzing the structure of the program) that the fixpoint computation terminates. ☐

The problem, known as *program termination*, (or *query termination*, when we refer to a specific query goal) is, in the general case, undecidable. Therefore, the recent research is investigating the identification of structural criteria that guarantee that the semantics can be computed using, for instance, bottom-up evaluators based on the grounding of programs. This is not a simple task as it is possible to have equivalent queries (i.e. queries computing the same answers, independently from the database) that have different structural properties and very basic changes to the syntax of programs, even without changing the semantics, may significantly alter the structural properties.

Current criteria analyze how values are propagated among predicate arguments, to understand whether the set of values associable with an argument is finite. However, these methods have limited capacity in comprehending finiteness of arguments appearing in recursive rules with function symbols in the head of rules. Considering the previous example, they are not able to understand that rule r_2 can be activated a finite number of times (actually, considering that there is only one exit rule, the recursive rule can be activated at most once).

Related works. As said before, the problem of checking whether the computation of a query terminates has been investigated since the beginning of logic programming.

Most of the past work was devoted to the termination of programs under top-down evaluation or for SLD resolution [8, 9]. The class of *finitary* programs, allowing decidable (ground) query computation using a top-down evaluation, has been proposed in [11]. A program $\mathcal{P}$ is finitary if (1) the number of cycles involving an odd number of negative subgoals is finite, and (2) it is *finitely recursive.* A program $\mathcal{P}$ is finitely recursive if each ground atom depends on finitely many ground atoms [12].

The problem of establishing whether the bottom-up based computation of logic programs terminates received a significant attention since the beginning of deductive databases [10] and recently has received an increasing interest. The class of *finitely ground ($\mathcal{FG}$) programs* has been proposed in [13]. The key property of this class is that stable models (answer sets) are computable. In fact, for each program $\mathcal{P}$ in this class there exists a finite and computable subset of its instantiation (grounding), called *intelligent instantiation*, having precisely the same answer sets as $\mathcal{P}$. As the problem of deciding whether a program is $\mathcal{FG}$ is not decidable, decidable subclasses, such as *finite domain ($\mathcal{FD}$) programs* [13], *ω-restricted programs* [14], *λ-restricted programs* [15] and *argument restricted ($\mathcal{AR}$) programs* [16] have been proposed. The query termination problem for ground query goals has been studied in [17]. Other approaches are the class of $\mathbb{FDNC}$ programs [2], i.e. programs having infinite answer sets in general, but a finite representation that can be exploited for knowledge compilation and fast query answering, and the proposal of [3], where functions are replaced by relations defined over finite domains.

Contribution. We first introduce the concept of *safe arguments* (a restriction of finite domain arguments), by also analyzing how rules may fire each other. As safe arguments can range only on a finite set of values, the instantiation of safe programs (that is, programs whose arguments are all safe) results in a finite ground program. Consequently, safe programs have a finite number of finite stable models and we show that the class of safe programs is decidable. We also show that the class of safe programs extends the class of finite domain programs, but is not comparable with the class of argument restricted programs.

Next we introduce a further criterion, called Γ-acyclicity, which analyzes the role of function symbols used in the program. We introduce the concept of *labelled propagation graph*, representing how complex terms in non-safe (or *affected*) arguments are created and used during bottom-up evaluation. The class of Γ-acyclic programs is defined by only considering affected arguments and cycles spelling strings of an underlying context free language. We show that this class is decidable, strictly extends both classes of safe programs and argument restricted programs and that it has a finite set of finite stable models which can be computed using current ASP systems, by a simple rewriting of the source program.

Finally, we discuss how the new criterion can be used in bound query answering.

Organization. The paper is organized as follows. Section 2 introduces basic notions on logic programming and recalls two main criteria guaranteeing the termination of logic programs with function symbols under bottom-up evaluation. Section 3 presents the class of safe programs. Section 4 introduces the class of Γ-acyclic programs. Section 5 shows how Γ-acyclic programs are rewritten so that their semantics can be computed by current ASP systems. Section 6 discusses bound query answering.

2 Logic programs with function symbols

Syntax. We assume to have infinite sets of constants, variables, predicate symbols and function symbols. Predicate and function symbols have associated a fixed arity. For a predicate p of arity n, we denote by $p[i]$, for $1 \leq i \leq n$, its i-th argument.

A *term* is either a constant, a variable or a complex term of the form $f(t_1, ..., t_m)$, where $t_1, ..., t_m$ are terms and f is a function symbol of arity m; each term t_i, for $1 \leq i \leq m$, is a *subterm* of $f(t_1, ..., t_m)$. The subterm relation is reflexive (each term is subterm of itself) and transitive (if t_i is subterm of t_j and t_j is subterm of t_k, then t_i is subterm of t_k). An *atom* is of the form $p(t_1, ..., t_n)$, where $t_1, ..., t_n$ are terms and p is a predicate symbols of arity n. A *literal* is either a (positive) atom A or its negation $\neg A$. A (*disjunctive*) rule r is a clause of the form:

$$a_1 \vee \cdots \vee a_m \leftarrow b_1, \cdots, b_k, \neg c_1, \cdots, \neg c_n$$

where $m > 0$ $k, n \geq 0$ and $a_1, \cdots, a_m, b_1, \cdots, b_k$, $c_1, \cdots, c_n$ are atoms. The disjunction $a_1 \vee \cdots \vee a_m$ is called the *head* of r and is denoted by $head(r)$ while the conjunction $b_1, \cdots, b_k, \neg c_1, \cdots, \neg c_n$ is called the *body* and is denoted by $body(r)$. If $m = 1$, then r is *normal* (i.e. ∨-free); if $n = 0$, then r is *positive* (i.e. ¬-free); if both $m = 1$ and $n = 0$, then r is *normal and positive.* A *program* $\mathcal{P}$ is a finite set of rules. A term (resp. an atom, a rule or a program) is said to be *ground* if no variables occur in it. A ground normal rule with an empty body is also called *fact.*

With a little abuse of notation we often use the same notation to denote a conjunction of body literals and a set of body literals, that is $body(r)$ is also used to denote the set of literals appearing in the body of r. We also denote the *positive body* of r by $body^+(r) = \{b_1, \ldots, b_k\}$

and the *negative body* of r by $body^{-}(r) = \{c_1, \ldots, c_n\}$. A predicate p depends on a predicate q if there is a rule r such that p appears in the head and q in the body, or there is a predicate s such that p depends on s and s depends on q. A predicate p is said to be recursive if it depends on itself, whereas two predicates p and q are said to be mutually recursive if p depends on q and q depends on p.

Generally, predicate symbols are partitioned into two different classes: extensional (or EDB or base), i.e. defined by the ground facts of a database, and intensional (or IDB or derived), i.e. defined by the rules of the program. The definition of a predicate p consists of all the rules (or facts) having p in the head. A database D consists of all the facts defining EDB predicates, whereas a program $\mathcal{P}$ consists of the rules defining IDB predicates. The program consisting of rules defining IDB predicates and facts defining EDB predicates is denoted by $\mathcal{P}_D$. When there is no ambiguity we shall use the symbol $\mathcal{P}$ to denote the complete set of rules and database facts. Given a set of ground atoms S and an atom $g(t)$, $S[g]$ (resp. $S[g(t)]$) denotes the set of g-tuples (resp. tuples matching $g(t)$) in S. Analogously, for a given set of sets of atoms M we shall use the following notations $M[g] = \{S[g] \mid S \in M\}$ and $M[g(t)] = \{S[g(t)] \mid S \in M\}$. We also assume that programs are *range restricted* [18], i.e. variables appearing in the head or in negated body literals are range restricted, that is they also appear in some positive body literal, and that possible constants in $\mathcal{P}$ are taken from the database domain[1].

Semantics. The Herbrand universe $H_{\mathcal{P}}$ of a program $\mathcal{P}$ is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in $\mathcal{P}$. The Herbrand base $B_{\mathcal{P}}$ of a program $\mathcal{P}$ is the set of ground atoms which can be built using predicate symbols appearing in $\mathcal{P}$ and ground terms of $H_{\mathcal{P}}$. A rule r' is a *ground instance* of a rule r, if r' is obtained from r by replacing every variable in r with some ground term in $H_{\mathcal{P}}$; $ground(\mathcal{P})$ denotes the set of all ground instances of the rules in $\mathcal{P}$. An interpretation of a program $\mathcal{P}$ is any subset of $B_{\mathcal{P}}$. The value of a ground atom L w.r.t. an interpretation I is $value_I(L) = L \in I$, whereas $value_I(\neg L) = L \notin I$. The truth value of a conjunction of ground literals $C = L_1, \ldots, L_n$ is the minimum over the values of L_i, i.e. $value_I(C) = min(\{value_I(L_i) \mid 1 \leq i \leq n\})$, while the value of a disjunction $D = L_1 \vee ... \vee L_n$ is its maximum, i.e. $value_I(D) = max(\{value_I(L_i) \mid 1 \leq i \leq n\})$; if $n = 0$, then $value_I(C) = true$ and $value_I(D) = false$. A ground rule r is *satisfied* by I if $value_I(head(r)) \geq value_I(body(r))$. Thus, a rule r with an empty body is satisfied by I if $value_I(head(r)) = true$. An interpretation M for $\mathcal{P}$ is a model of $\mathcal{P}$ if M satisfies all the rules in $ground(\mathcal{P})$.

The *model-theoretic semantics* for a positive program $\mathcal{P}$ assigns the set of its *minimal models* $\mathcal{MM}(\mathcal{P})$. A model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$. The more general *disjunctive stable model semantics* generalizes stable model semantics previously defined for normal programs [19] and also applies to programs with (unstratified) negation [20].

Let $\mathcal{P}$ be a logic program and let I be an interpretation for $\mathcal{P}$, $\mathcal{P}^I$ denotes the ground positive program derived from $ground(\mathcal{P})$ by (1) removing all the rules that contain a negative literal $\neg a$ in the body and $a \in I$, and (2) removing all the negative literals from the remaining rules. An interpretation I is a (disjunctive) stable model for $\mathcal{P}$ if and only if $I \in \mathcal{MM}(\mathcal{P}^I)$. The set of stable models of $\mathcal{P}$ is denoted by $\mathcal{SM}(\mathcal{P})$. It is well known

[1] Range restricted programs are often called safe programs. We will use the term safe to denote a set of program arguments.

that stable models are minimal models (i.e. $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$) and that for negation-free programs minimal and stable model semantics coincide (i.e. $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$) and that positive normal programs have a unique minimal model.

Finite domain programs. The class of finite domain programs is defined by analyzing the structure of programs and is based on the concept of argument graph.

The *argument graph* $G^A(\mathcal{P})$ of a program $\mathcal{P}$ is a direct graph containing a node for each argument $p[i]$ of an IDB predicate p of $\mathcal{P}$; there is an edge $(q[j], p[i])$ iff there is a rule $r \in \mathcal{P}$ such that: i) an atom $p(\overline{t})$ appears in the head of r; ii) an atom $q(\overline{v})$ appears in $body^+(r)$; iii) $p(\overline{t})$ and $q(\overline{v})$ share the same variable within the i-th and the j-th term, respectively. Given a program $\mathcal{P}$, an argument $p[i]$ is said to be *recursive* if it appears in a cycle of $G^A(\mathcal{P})$.

▶ **Definition 2** ($\mathcal{FD}$ Program [13])**.** Given a program $\mathcal{P}$, the set of *finite-domain arguments ($\mathcal{FD}$ arguments)* of $\mathcal{P}$ is the maximal set $FD(\mathcal{P})$ of arguments of $\mathcal{P}$ such that, for each argument $q[k] \in FD(\mathcal{P})$, every rule r with head predicate q satisfies the following condition. Let t be the term corresponding to argument $q[k]$ in the head of r. Then, either i) t is variable-free, or ii) t is a subterm of (the term of) an $\mathcal{FD}$ argument of a positive body predicate, or iii) every variable appearing in t also appears in (the term of) an $\mathcal{FD}$ argument of a positive body predicate which is not recursive with $q[k]$. If all arguments of the predicates of $\mathcal{P}$ are $\mathcal{FD}$, then $\mathcal{P}$ is said to be an $\mathcal{FD}$ program. □

The main properties of $\mathcal{FD}$ programs are the following: (i) recognizing whether $\mathcal{P}$ is an $\mathcal{FD}$ program is decidable, and (ii) every $\mathcal{FD}$ program is an $\mathcal{FG}$ program. Checking whether a program $\mathcal{P}$ is $\mathcal{FD}$ or not can be done by assuming that all arguments are in $FD(\mathcal{P})$ and eliminating, iteratively, arguments appearing in the head of a rule such that none of the three conditions of Definition 2 holds.

Argument Restricted programs. For any atom $p(t_1, ..., t_n)$, $p(t_1, ..., t_n)^0$ denotes the predicate symbol p, whereas $p(t_1, ..., t_n)^i$, for $1 \leq i \leq n$, denotes its argument term t_i. The depth of a variable X in a term t that contains X, denoted by $d(X, t)$, is defined recursively as follows:

$$d(X,t) = \begin{cases} 0 & if\ t = X \\ 1 + max_{i:t_i\ contains\ X}\, d(X, t_i) & if\ t = f(t_1, ..., t_n) \end{cases}$$

▶ **Definition 3** ($\mathcal{AR}$ Program [16])**.** An argument ranking for a program $\mathcal{P}$ is a function ϕ from arguments to 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 an argument term A^i, $body^+(r)$ contains an atom B such that X occurs in an argument term B^j satisfying the condition

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

A program is argument restricted ($\mathcal{AR}$) if it has an argument ranking. □

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

$r_1 : \mathtt{succ(X, f(X))} \leftarrow \mathtt{nat(X)}.$
$r_2 : \mathtt{nat(0)}.$
$r_3 : \mathtt{nat(Y)} \leftarrow \mathtt{succ(X, Y),\ bounded(Y)}.$

where `bounded` is a base predicate. The argument graph $G^A(\mathcal{P}_4)$ contains the following edges $(nat[1], succ[1])$, $(nat[1], succ[2])$, $(succ[2], nat[1])$ and $(bounded[1], nat[1])$. The program is not finite domain as the argument $succ[2]$ is not finite domain. However, $\mathcal{P}_4$ is argument restricted as it is possible to assign the following consistent ranking to arguments: $\phi(bounded[1]) = \phi(nat[1]) = \phi(succ[1]) = 0$ and $\phi(succ[2]) = 1$. □

The class of argument restricted programs is contained in finitely ground, generalizes the finite domain class and is decidable.

3 Safe programs

In this section we introduce a new criterion guaranteeing that there is a finite instantiation, equivalent to the source program and, therefore, a finite set of finite stable models.

▶ **Definition 5** (Activation Graph). Let $\mathcal{P}$ be a program, the *activation graph* $\Omega(\mathcal{P}) = (\mathcal{P}, E)$ consists of a set of nodes denoting rules and a set of edges E defined as follows: for each pair of rules r and s there is an edge (r, s) from r to s if there is a set of ground facts DB_1 and two matchers θ_1 and θ_2 such that

1. $DB_1 \models body(r)\theta_1 \wedge DB_1 \not\models head(r)\theta_1$ and
2. let $DB_2 = DB_1 \cup head(r)\theta_1$, the following conditions hold:
   - $DB_2 \models body(s)\theta_2 \wedge DB_2 \not\models head(s)\theta_2$ and
   - $DB_1 \not\models body(s)\theta_2 \vee DB_1 \models head(s)\theta_2$. □

▶ **Example 6.** Consider the program $\mathcal{P}_1$ of Example 1 and let $\Omega(\mathcal{P}_1) = (\mathcal{P}_1, E)$ its activation graph. We have that $(r_1, r_2) \in E$, but $(r_2, r_2) \notin E$, as the firing of r_2 cannot fire r_2 again. Clearly, being r_1 a fact, it cannot be fired by another rule. Therefore, $\Omega(\mathcal{P}_1)$ is acyclic. □

▶ **Definition 7** (Safe Function). For any program $\mathcal{P}$, let A be a subset of arguments of $\mathcal{P}$, $\Psi_{\mathcal{P}}(A)$ denotes the set of arguments occurring in $\mathcal{P}$ such that for all rules $r \in \mathcal{P}$ where q appears in the head

1. r does not appear in a cycle of $\Omega(\mathcal{P})$, or
2. let t be the term corresponding to argument $q[k]$, for every variable X appearing in $q[k]$ in the head of r (considering all head occurrences), X also appears in some argument in $body^+(r)$ belonging to A. □

The function $\Psi_{\mathcal{P}}$ is monotonic and, for every set of arguments A occurring in $\mathcal{P}$, the sequence $\Psi_{\mathcal{P}}(A)$, $\Psi^2_{\mathcal{P}}(A)$, ... , $\Psi^i_{\mathcal{P}}(A)$, ... converges in a finite number of steps, that is, there is some finite n such that $\Psi^n_{\mathcal{P}}(A) = \Psi^{n+1}_{\mathcal{P}}(A)$.

▶ **Definition 8** (Safe Arguments). For any program $\mathcal{P}$, $safe(\mathcal{P}) = \Psi^\infty_{\mathcal{P}}(A)$, where $A = FD(\mathcal{P})$ is the set of finite domain arguments of $\mathcal{P}$, denotes the set of safe arguments of $\mathcal{P}$. A program $\mathcal{P}$ is said to be *safe* if all arguments are safe. □

It is worth noting that the starting set to compute safe arguments could be the set of finite domain arguments satisfying condition i) or ii) of Definition 2, that is condition iii) is not necessary to compute safe arguments. We shall denote by $args(\mathcal{P})$ the set of arguments of a program $\mathcal{P}$ and by $aff(\mathcal{P}) = args(\mathcal{P}) - safe(\mathcal{P})$ the set of affected arguments. The class of safe programs will be denoted by $\mathcal{SP}$.

▶ **Example 9.** Consider the program $\mathcal{P}_4$ of Example 4. Although the activation graph is not acyclic (there is a cycle between r_1 and r_3), we have that i) $bounded[1]$, $nat[1]$ and $succ[1]$ are safe as they are finite domain, and ii) $succ[2]$ is safe as the variable X in the first rule appears in a safe body argument. Since all arguments are safe, we have that the program $\mathcal{P}_4$ is safe. □

▶ **Example 10.** The program $\mathcal{P}_1$ of Example 1 is safe, as rule r_2 does not fire itself and the graph $\Omega(\mathcal{P}_1)$ is empty. Moreover, it is not argument-restricted as it is not possible to assign a rank to $p[1]$ and $p[2]$ such that $\phi(p[1]) - \phi(p[j]) \geq d(X, f(X)) - d(X, X)$ and $\phi(p[2]) - \phi(p[j]) \geq d(X, g(X)) - d(X, X)$, with $j = 1, 2$. □

▶ Proposition 11. The problem of deciding whether a program $\mathcal{P}$ is safe is decidable. □

The following theorem states that the class of safe programs i) strictly contains the class of finite domain programs, ii) is not comparable with the class of argument-restricted programs, and iii) is contained in the class of finitely ground programs.

▶ **Theorem 12.** *$\mathcal{FD} \subsetneq \mathcal{SP} \subsetneq \mathcal{FG}$, $\mathcal{AR} \not\subseteq \mathcal{SP}$ and $\mathcal{SP} \not\subseteq \mathcal{AR}$.* □

▶ **Corollary 13.** *For any safe program $\mathcal{P}$, the stable models of $\mathcal{P}$ are finite.* □

From Theorem 12 it also follows that any safe program $\mathcal{P}$ has finitely many stable models and both brave and cautious reasoning over safe programs are computable even for non-ground queries.

4 Exploiting function symbols

In this section we further improve our technique by exploiting the role of function symbols for checking program termination under bottom-up evaluation. We assume that if the same variable X appears in two terms occurring in the head and body of a rule, then one of the two terms must be a subterm of the other and that the nesting level of complex terms is at most one. There is no real restriction in such an assumption as every program could be rewritten into an equivalent program satisfying such a condition. For instance, a rule of the form $p(f(h(X))) \leftarrow q(g(X))$ could be rewritten into the following two rules: $p(f(X)) \leftarrow p'(X)$, $p'(h(X)) \leftarrow p''(X)$ and $p''(X) \leftarrow q(g(X))$.

The following example shows a program admitting finite stable models, but previous criteria, included the safety criterion, are not able to detect it.

▶ **Example 14.** Consider the below program $\mathcal{P}_{14}$:

```
r(f(X)) ← s(X).
q(f(X)) ← r(X).
p(X) ← q(X).
n(X) ← p(g(X)).
s(X) ← n(X).
```

The program is neither safe, as all arguments are affected, nor argument restricted. □

The *(labelled) propagation graph* $\Delta(\mathcal{P})$ is the graph derived from the argument graph $G^A(\mathcal{P})$ by only considering affected arguments and adding labels to arcs.

▶ **Definition 15** (Labelled argument and propagation graphs). Let $\mathcal{P}$ be a program, the *labelled argument graph* $\mathcal{G}^A_L(\mathcal{P}) = (args(\mathcal{P}), E)$, where E is a set of labelled edges defined as follows. For each pair of nodes $p[j], q[i] \in args(\mathcal{P})$ and for every rule $r \in \mathcal{P}$ such that i) there is an atom $q(u) \in body^+(r)$, ii) $head(r) = p(v)$ and iii) the same variable X occurs in both $q[j]$ and $p[i]$, there is an arc $(q[j], p[i], \alpha) \in E$ where

- $\alpha = \epsilon$ if $q[j] = p[i]$ and both arguments contain variables;
- $\alpha = f$ if $q[j] = X$ and $p[i] = f(..., X, ...)$;
- $\alpha = \overline{f}$ if $q[j] = f(..., X, ...)$ and $q[j] = X$.

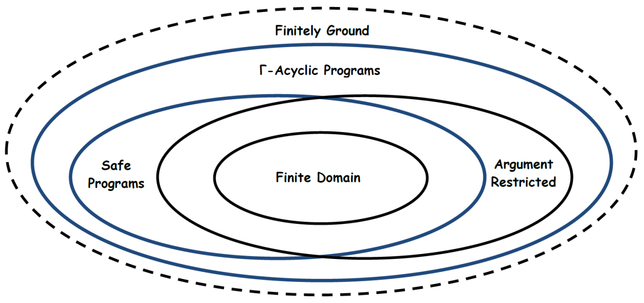

Figure 1 Criteria relationships.

The *(labelled) propagation graph* $\Delta(\mathcal{P})$ is the graph derived from the labelled argument graph $G_L^A(\mathcal{P})$ by only considering affected arguments. □

A path π is a sequence $n_1\,\alpha_1\,n_2\,\alpha_2\,...n_k\,\alpha_k\,n_{k+1}$, where $k \geq 1$ and for each $i \in [1..k]$ (n_i, n_{i+1}, α_i) is an edge of $\Delta(\mathcal{P})$. For any path $\pi = n_1\,\alpha_1\,n_2\,\alpha_2\,...\,n_k\,\alpha_k n_{k+1}$, we denote with $\lambda(\pi)$ the string $\alpha_1 \ldots \alpha_k$.

▶ **Definition 16.** Let $\mathcal{P}$ be a program and let $F = \{f_1, ..., f_m\}$ be the set of function symbols occurring in $\mathcal{P}$. The *grammar* $\Gamma_\mathcal{P}$ is a 4-tuple (N, T, R, S), where $N = \{S, S_1, S_2\}$ is the set of nonterminal symbols, $T = \{f \mid f \in F\} \cup \{\overline{f} \mid f \in F\}$ is the set of terminal symbols, S is the start symbol and R is the set of production rules below defined:

- $S \to S_1\,f_i\,S_2,$ $\quad\forall f_i \in F;$
- $S_1 \to f_i\,S_1\,\overline{f_i}\,S_1 \mid \epsilon,$ $\quad\forall f_i \in F;$
- $S_2 \to (S_1 \mid f_i)\,S_2 \mid \epsilon,$ $\quad\forall f_i \in F.$ □

The language $\mathcal{L}(\Gamma_\mathcal{P})$ is the set of strings generated by $\Gamma_\mathcal{P}$. As $\Gamma_\mathcal{P}$ is context free, the language $\mathcal{L}(\Gamma_\mathcal{P})$ can be recognized by means of a pushdown automaton. Given a grammar $\Gamma = \{N, T, R, S\}$ and a graph G, we say that path π in G spells a string $w \in \mathcal{L}(\Gamma)$ if $\lambda(\pi) = w$.

▶ **Definition 17** (Γ-acyclic Programs)**.** A program $\mathcal{P}$ is said Γ-*acyclic* if there is no cycle in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_\mathcal{P})$. □

Considering previous Example 14, the program $\mathcal{P}_{14}$ is Γ-acyclic, but not safe. Indeed, there is a cycle spelling the strings "$f\,f\,\overline{g}$", "$f\,\overline{g}\,f$" and "$\overline{g}\,f\,f$", but all strings do not belong to the language $\mathcal{L}(\Gamma_{\mathcal{P}_{14}})$. Observe that, in order to correctly recognize a cycle in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_\mathcal{P})$, we have to start from an edge with a positive label f (i.e. starting from an unlabelled edge or from an edge with a label $\overline{f}$ is not useful).

▶ Proposition 18. The problem of deciding whether a program is Γ-acyclic is decidable. □

The below theorem states that the class of acyclic programs strictly contains both classes of safe programs and argument restricted programs and is contained in the class of finitely ground programs.

▶ **Theorem 19.** $\mathcal{SP} \cup \mathcal{AR} \subsetneq \mathcal{AP} \subsetneq \mathcal{FG}$ □

▶ **Corollary 20.** *For any* Γ*-acyclic program* $\mathcal{P}$*, the stable models of* $\mathcal{P}$ *are finite.* □

The relationships among previous criteria and the new ones are reported in Fig. 1.

5 Computing stable models for Γ-acyclic programs

We now show how stable models for Γ-acyclic programs can be computed using current algorithms based on the grounding of programs. The idea is that, considering that positive normal Γ-acyclic programs have a finite minimum model, from a Γ-acyclic program $\mathcal{P}$, we first generate a standard Γ-acyclic program $st(\mathcal{P})$ such that all stable models of $\mathcal{P}$ are contained in the minimum model of $st(\mathcal{P})$ and next we generate a new program $ext(\mathcal{P})$ equivalent to $\mathcal{P}$, such that there is a ground, finite, equivalent version. The computation of the stable models of $ext(\mathcal{P})$ could be carried out by current answer set systems [4, 5, 6].

▶ **Definition 21** (Standard program). Let $\mathcal{P}$ be a logic program, $st(\mathcal{P})$ denote the normal, positive program, called standard version, obtained by replacing i) each disjunctive rule r having m atoms $a_1, ..., a_m$ in the head with m positive rules of the form $a_i \leftarrow body^+(r)$, for $1 \leq i \leq m$, and ii) each derived predicate symbol q with a new derived predicate symbol Q. □

▶ **Example 22.** Consider the program $\mathcal{P}_{22}$ consisting of the two rules

$$\texttt{p(X)} \vee \texttt{q(X)} \leftarrow \texttt{r(X)}, \neg\texttt{a(X)}.$$
$$\texttt{r(X)} \leftarrow \texttt{b(X)}, \neg\texttt{q(X)}.$$

where p, q and r are derived predicates (mutually recursive), whereas a and b are base predicates. The derived standard program $st(\mathcal{P}_{22})$ is:

$$\texttt{P(X)} \leftarrow \texttt{R(X)}.$$
$$\texttt{Q(X)} \leftarrow \texttt{R(X)}.$$
$$\texttt{R(X)} \leftarrow \texttt{b(X)}.$$

▶ **Lemma 23.** *Let $\mathcal{P}$ be a program and let $\mathcal{P}' = st(\mathcal{P}) \cup \{q(\bar{X}) \leftarrow Q(\bar{X}) \mid q \in dpred(\mathcal{P})\}$, where $dpred(\mathcal{P})$ denotes the set of derived predicate symbols in $\mathcal{P}$. For any stable model $M \in \mathcal{SM}(\mathcal{P})$, $M \subseteq \mathcal{MM}(\mathcal{P}')[S_\mathcal{P}]$, where $S_\mathcal{P}$ denotes the set of predicate symbols in $\mathcal{P}$.* □

For any rule r such that $head(r) = q_1(u_1) \vee \cdots \vee q_k(u_k)$, $headconj(r)$ denotes the conjunction $Q_1(u_1), ..., Q_k(u_k)$.

▶ **Definition 24** (Extended program). Let $\mathcal{P}$ be a disjunctive program and let r be a rule of $\mathcal{P}$, then, $ext(r)$ denotes the (disjunctive) extended rule $head(r) \leftarrow headconj(r), body(r)$ obtained by extending the body of r, whereas $ext(\mathcal{P}) = \{ext(r) \mid r \in \mathcal{P}\} \cup st(\mathcal{P})$ denotes the (disjunctive) program obtained by extending the rules of $\mathcal{P}$ and adding (standard) rules defining the new predicates. □

▶ **Example 25.** Consider the program $\mathcal{P}_{22}$ of Example 22. The extended program $ext(\mathcal{P}_{22})$ is as follows:

$$\texttt{p(X)} \vee \texttt{q(X)} \leftarrow \texttt{P(X)}, \texttt{Q(X)}, \texttt{r(X)}, \neg\texttt{a(X)}$$
$$\texttt{r(X)} \leftarrow \texttt{R(X)}, \texttt{b(X)}, \neg\texttt{q(X)}$$

plus the rules in $st(\mathcal{P}_{22})$ showed in Example 22. □

The following theorem states that $\mathcal{P}$ and $ext(\mathcal{P})$ are equivalent w.r.t. the set of predicate symbols in $\mathcal{P}$.

▶ **Theorem 26.** *For every program $\mathcal{P}$, $\mathcal{SM}(\mathcal{P})[S_\mathcal{P}] = \mathcal{SM}(ext(\mathcal{P}))[S_\mathcal{P}]$, where $S_\mathcal{P}$ is the set of predicate symbols occurring in $\mathcal{P}$.* □

6 Bound queries

The bottom-up computation of queries whose related programs are not range-restricted, could not be carried out, as the ground instantiation is infinite. The application of well known rewriting techniques, such as magic-set, may allow bottom-up evaluators to (efficiently) compute bounded queries, by rewriting queries so that the top-down evaluation is emulated [21, 22, 23, 17]. Before presenting our technique, let us introduce some notations.

A query is a pair $Q = \langle q(u_1, .., u_n), \mathcal{P}\rangle$, where $q(u_1, .., u_n)$ is an atom called query goal and $\mathcal{P}$ is a program. An *adornment* of predicate p with arity n is a string $\alpha \in \{b, f\}^*$ such that $|\alpha| = n$. The symbols b and f denote, respectively, bound and free arguments. Given a query $Q = \langle q(u_1, .., u_n), \mathcal{P}\rangle$, $MagicS(Q) = \langle q^\alpha(u_1, .., u_n), MagicS(q(u_1, .., u_n), \mathcal{P})\rangle$ denotes the rewriting of Q, where $MagicS(q(u_1, .., u_n), \mathcal{P})$ denotes the rewriting of rules in $\mathcal{P}$ with respect to the query goal $q(u_1, .., u_n)$ and α is the adornment associated with the query goal.

Since the magic-set rewriting technique has been defined for subclasses of queries (e.g. stratified queries), we assume that our queries are positive[2], although we could consider larger classes with the only necessary condition being that after their rewriting queries must be range restricted.

▶ **Definition 27.** A query $Q = \langle G, \mathcal{P}\rangle$ is said Γ-*acyclic* if either $\mathcal{P}$ or $MagicS(G, \mathcal{P})$ is Γ-acyclic. ◻

It is worth noting that it is possible to have a query $Q{=}\langle G, \mathcal{P}\rangle$ such that $\mathcal{P}$ is Γ-acyclic, but the rewritten program $MagicS(G, \mathcal{P})$ is not Γ-acyclic and vice versa.

▶ **Example 28.** Consider the query $Q = \langle \texttt{p(f(f(a)))}, \mathcal{P}_{28}\rangle$, where $\mathcal{P}_{28}$ is defined below:

```
p(a).
p(f(X))←p(X).
```

$\mathcal{P}_{28}$ is not Γ-acyclic, but if we rewrite the program using the magic-set method, we obtain the Γ-acyclic program:

$\texttt{p}^\texttt{b}\texttt{(a)} \leftarrow \texttt{magic\_p}^\texttt{b}\texttt{(a)}.$ $\qquad$ $\texttt{magic\_p}^\texttt{b}\texttt{(f(f(a)))}.$

$\texttt{p}^\texttt{b}\texttt{(f(X))} \leftarrow \texttt{magic}^\texttt{b}_\texttt{p}\texttt{(f(X))},\ \texttt{p}^\texttt{b}\texttt{(X)}.$ $\qquad$ $\texttt{magic\_p}^\texttt{b}\texttt{(X)} \leftarrow \texttt{magic\_p}^\texttt{b}\texttt{(f(X))}.$

Consider now the query $Q = \langle \texttt{p(a)}, \mathcal{P}'_{28}\rangle$, where $\mathcal{P}'_{28}$ is defined as follows:

```
p(f(f(a))).
p(X)←p(f(X)).
```

The program is Γ-acyclic, but after the magic-set rewriting we obtain the below set of rules:

$\texttt{p}^\texttt{b}\texttt{(f(f(a)))} \leftarrow \texttt{magic\_p}^\texttt{b}\texttt{(f(f(a)))}.$ $\qquad$ $\texttt{magic\_p}^\texttt{b}\texttt{(a)}.$

$\texttt{p}^\texttt{b}\texttt{(X)} \leftarrow \texttt{magic}^\texttt{b}_\texttt{p}\texttt{(X)}, \texttt{p}^\texttt{b}\texttt{(f(X))}.$ $\qquad$ $\texttt{magic\_p}^\texttt{b}\texttt{(f(X))} \leftarrow \texttt{magic\_p}^\texttt{b}\texttt{(X)}.$

which is not Γ-acyclic. ◻

Thus, we propose to first check if the input program is Γ-acyclic and, if it does not satisfy Γ-acyclicity, to check the property on the rewritten program, which is query-equivalent to the original one.

[2] For positive queries we mean queries $\langle G, \mathcal{P}\rangle$ such that $\mathcal{P}$ is positive.

References

1 P. A. Bonatti, "On the decidability of fdnc programs," *Intellig. Artific.*, vol. 5, no. 1, 2011.
2 T. Eiter and M. Simkus, "Fdnc: Decidable nonmonotonic disjunctive logic programs with function symbols," *ACM Trans. Comput. Log.*, vol. 11, no. 2, 2010.
3 F. Lin and Y. Wang, "Answer set programming with functions," in *KR*, pp. 454–465, 2008.
4 N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, K. Koch, S. Perri, and A. Polleres, "The dlv system," in *Jelia*, pp. 537–540, 2002.
5 P. Simons, I. Niemel"a, and T. Soininen, "Extending and implementing the stable model semantics," *Artif. Intell.*, vol. 138, no. 1-2, pp. 181–234, 2002.
6 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *LPNMR*, pp. 260–265, 2007.
7 S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: an interactive tutorial," in *SIGMOD Conference*, pp. 1213–1216, 2011.
8 D. D. Schreye and S. Decorte, "Termination of logic programs: The never-ending story," *J. Log. Program.*, vol. 19/20, pp. 199–260, 1994.
9 D. Voets and D. D. Schreye, "Non-termination analysis of logic programs with integer arithmetics," *TPLP*, vol. 11, no. 4-5, pp. 521–536, 2011.
10 R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli, "A framework for testing safety and effective computability," *J. Comput. Syst. Sci.*, vol. 52, no. 1, pp. 100–124, 1996.
11 P. A. Bonatti, "Reasoning with infinite stable models," *Artif. Intell.*, vol. 156, no. 1, 2004.
12 S. Baselice, P. A. Bonatti, and G. Criscuolo, "On finitely recursive programs," *TPLP*, vol. 9, no. 2, pp. 213–238, 2009.
13 F. Calimeri, S. Cozza, G. Ianni, and N. Leone, "Computable functions in asp: Theory and implementation," in *ICLP*, pp. 407–424, 2008.
14 T. Syrjänen, "Omega-restricted logic programs," in *LPNMR*, pp. 267–279, 2001.
15 M. Gebser, T. Schaub, and S. Thiele, "Gringo : A new grounder for answer set programming," in *LPNMR*, pp. 266–271, 2007.
16 Y. Lierler and V. Lifschitz, "One more decidable class of finitely ground programs," in *ICLP*, pp. 489–493, 2009.
17 M. Alviano, W. Faber, and N. Leone, "Disjunctive asp with functions: Decidable queries and effective computation," *TPLP*, vol. 10, no. 4-6, pp. 497–512, 2010.
18 J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
19 M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *ICLP/SLP*, pp. 1070–1080, 1988.
20 M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Comput.*, vol. 9, no. 3/4, pp. 365–386, 1991.
21 C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 255–299, 1991.
22 S. Greco, "Binding propagation techniques for the optimization of bound disjunctive queries," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 2, pp. 368–385, 2003.
23 G. Greco, S. Greco, I. Trubitsyna, and E. Zumpano, "Optimization of bound disjunctive queries with constraints," *TPLP*, vol. 5, no. 6, pp. 713–745, 2005.