Checking termination of bottom-up evaluation of logic programs with function symbols

Citation for published version:
Calautti, M, Greco, S, Spezzano, F & Trubitsyna, I 2015, 'Checking termination of bottom-up evaluation of logic programs with function symbols' Theory and Practice of Logic Programming, vol. 15, no. 6, pp. 854–889. DOI: 10.1017/S1471068414000623

Digital Object Identifier (DOI):
10.1017/S1471068414000623

Published In:
Theory and Practice of Logic Programming

TLP **00** (0): 1–36, 2014.
doi:10.1017/S1471068414000623

*Checking termination of bottom-up evaluation of logic programs with function symbols**

MARCO CALAUTTI, SERGIO GRECO, FRANCESCA SPEZZANO and IRINA TRUBITSYNA Q1

DIMES, Università della Calabria, 87036 Rende (CS), Italy
(*e-mail:* {`calautti,greco,fspezzano,trubitsyna`}`@dimes.unical.it`)

submitted 7 December 2012; revised 26 March 2014; accepted 29 June 2014

Abstract

Recently, there has been an increasing interest in the bottom-up evaluation of the semantics of logic programs with complex terms. The presence of function symbols in the program may render the ground instantiation infinite, and finiteness of models and termination of the evaluation procedure, in the general case, are not guaranteed anymore. Since the program termination problem is undecidable in the general case, several decidable criteria (called program termination criteria) have been recently proposed. However, current conditions are not able to identify even simple programs, whose bottom-up execution always terminates. The paper introduces new decidable criteria for checking termination of logic programs with function symbols under bottom-up evaluation, by deeply analyzing the program structure. First, we analyze the propagation of complex terms among arguments by means of the extended version of the argument graph called *propagation graph*. The resulting criterion, called Γ-*acyclicity*, generalizes most of the decidable criteria proposed so far. Next, we study how rules may activate each other and define a more powerful criterion, called *safety*. This criterion uses the so-called *safety function* able to analyze how rules may activate each other and how the presence of some arguments in a rule limits its activation. We also study the application of the proposed criteria to bound queries and show that the safety criterion is well-suited to identify relevant classes of programs and bound queries. Finally, we propose a hierarchy of classes of terminating programs, called k-*safety*, where the k-safe class strictly includes the (k-1)-safe class.

KEYWORDS: Logic programming with function symbols, bottom-up execution, program termination, stable models

1 Introduction

Recently, there has been an increasing interest in the bottom-up evaluation of the semantics of logic programs with complex terms. Although logic languages under stable model semantics have enough expressive power to express problems in the second level of the polynomial hierarchy, in some cases function symbols make

* This work refines and extends results from the conference paper (Greco *et al.* 2012).

languages compact and more understandable. For instance, several problems can be naturally expressed using list and set constructors, and arithmetic operators. The presence of function symbols in the program may render the ground instantiation infinite, and finiteness of models and termination of the evaluation procedure, in the general case, are not guaranteed anymore. Since the program termination problem is undecidable in the general case, several decidable sufficient conditions (called *program termination criteria*) have been recently proposed.

The program termination problem has received a significant attention since the beginning of logic programming and deductive databases (Krishnamurthy *et al.* 1996) and has recently received an increasing interest. A considerable body of work has been done on termination of logic programs under top-down evaluation (Schreye and Decorte 1994; Marchiori 1996; Ohlebusch 2001; Bonatti 2004; Codish *et al.* 2005; Serebrenik and De Schreye 2005; Bruynooghe *et al.* 2007; Nguyen *et al.* 2007; Baselice *et al.* 2009; Schneider-Kamp *et al.* 2009a; Schneider-Kamp *et al.* 2009b; Nishida and Vidal 2010; Schneider-Kamp *et al.* 2010; Ströder *et al.* 2010; Voets and Schreye 2010; Brockschmidt *et al.* 2012; Liang and Kifer 2013). In this context, the class of *finitary* programs, allowing decidable (ground) query computation using a top-down evaluation, has been proposed in (Bonatti 2004; Baselice *et al.* 2009). Moreover, there are other research areas, such as these of term rewriting (Zantema 1995; Ferreira and Zantema 1996; Arts and Giesl 2000; Sternagel and Middeldorp 2008; Endrullis *et al.* 2008) and chase termination (Fagin *et al.* 2005; Marnette 2009; Meier *et al.* 2009; Greco and Spezzano 2010; Greco *et al.* 2011), whose results can be of interest to the logic program termination context.

In this paper, we consider logic programs with function symbols *under the stable model semantics* (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) and thus, all the excellent works mentioned above cannot be straightforwardly applied to our setting. Indeed, the goal of top-down termination analysis is to detect, for a given program and query goal, sufficient conditions guaranteeing that the resolution algorithm terminates. On the other side, the aim of the bottom-up termination analysis is to guarantee the existence of an equivalent finite ground instantiation of the input program. Furthermore, as stated in (Schreye and Decorte 1994), even restricting our attention to the top-down approach, the termination of logic programs strictly depends on the selection and search rules used in the resolution algorithm. Considering the different aspects of term rewriting and termination of logic programs, we address readers to (Schreye and Decorte 1994) (pp. 204–207).

In this framework, the class of *finitely ground programs* ($\mathcal{FG}$) has been proposed in (Calimeri *et al.* 2008). The key property of this class is that stable models (answer sets) are computable as 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 $\mathcal{P}$. Since the problem of deciding whether a program is in $\mathcal{FG}$ is not decidable, decidable subclasses, such as *finite domain programs* (Calimeri *et al.* 2008), ω-*restricted programs* (Syrjänen 2001), λ-*restricted programs* (Gebser *et al.* 2007b), and the most general one, *argument-restricted programs* (Lierler and Lifschitz 2009), have been proposed.

Current techniques analyze how values are propagated among predicate arguments to detect whether a given argument is *limited*, i.e. whether the set of values which can be associated with the argument, also called *active domain*, is finite. However, these methods have limited capacity in comprehending that arguments are limited in the case where different function symbols appear in the recursive rules. Even the argument-restricted criterion, which is one the most general criteria, fails in such cases.

Thus, we propose a new technique, called Γ-acyclicity, whose aim is to improve the argument-restricted criterion without changing the (polynomial) time complexity of the argument-restricted criterion. This technique makes use of the so-called *propagation graph*, that represents the propagation of values among arguments and the construction of complex terms during the program evaluation.

Furthermore, since many practical programs are not recognized by current termination criteria, including the Γ-acyclicity criterion, we propose an even more general technique, called *safety*, which also analyzes how rules activate each other. The new technique allows us to recognize as terminating many classical programs, still guaranteeing polynomial time complexity.

Example 1
Consider the following program P_1 computing the length of a list stored in a fact of the form input(L):

$$
\begin{aligned}
\mathtt{r}_0 &: \ \mathtt{list(L)} \leftarrow \mathtt{input(L)}.\\
\mathtt{r}_1 &: \ \mathtt{list(L)} \leftarrow \mathtt{list([X|L])}.\\
\mathtt{r}_2 &: \ \mathtt{count([\,],0)}.\\
\mathtt{r}_3 &: \ \mathtt{count([X|L],I+1)} \leftarrow \mathtt{list([X|L])},\ \mathtt{count(L,I)}.
\end{aligned}
$$

where input is a base predicate defined by only one fact of the form input([a, b, ...]). □

The safety technique, proposed in this paper, allows us to understand that P_1 is finitely ground and, therefore, terminating under the bottom-up evaluation.

Contribution.

- We first refine the method proposed in (Lierler and Lifschitz 2009) by introducing the set of restricted arguments and we show that the complexity of finding such arguments is polynomial in the size of the given program.
- We then introduce the class of Γ-acyclic programs, that strictly extends the class of argument-restricted programs. Its definition is based on a particular graph, called propagation graph, representing how complex terms in non-restricted arguments are created and used during the bottom-up evaluation. We also show that the complexity of checking whether a program is Γ-acyclic is polynomial in the size of the given program.
- Next we introduce the *safety function* whose iterative application, starting from the set of Γ-acyclic arguments, allows us to derive a larger set of limited arguments, by analyzing how rules may be activated. In particular, we define the *activation graph* that represents how rules may activate each other and

design conditions detecting rules whose activation cannot cause their head arguments to be non-limited.

- Since new criteria are defined for normal logic programs without negation, we extend their application to the case of disjunctive logic programs with negative literals and show that the computation of stable models can be performed using current ASP systems, by a simple rewriting of the source program.
- We propose the application of the new criteria to bound queries and show that the safety criterion is well suited to identify relevant classes of programs and bound queries.
- As a further improvement, we introduce the notion of *active paths* of length k and show its applicability in the termination analysis. In particular, we generalize the safety criterion and show that the k-*safety* criteria define a hierarchy of terminating criteria for logic programs with function symbols.
- Complexity results for the proposed techniques are also presented. More specifically, we show that the complexity of deciding whether a program $\mathscr{P}$ is Γ-acyclic or safe is polynomial in the size of $\mathscr{P}$, whereas the complexity of deciding whether a program is k-safe, with $k > 1$ is exponential.

A preliminary version of this paper has been presented at the 28th International Conference on Logic Programming (Greco *et al.* 2012). Although the concepts of Γ-acyclic program and safe program have been introduced in the conference paper, the definitions contained in the current version are different. Moreover, most of the theoretical results and all complexity results contained in this paper as well as the definition of k-safe program are new.

Organization. The paper is organized as follows. Section 2 introduces basic notions on logic programming with function symbols. Section 3 presents the argument-restriction criterion. In Section 4 the propagation of complex terms among arguments is investigated and the class of Γ-acyclic programs is defined. Section 5 analyzes how rules activate each other and introduces the *safety* criterion. In Section 6 the applicability of the safety criterion to (partially) ground queries is discussed. Section 7 presents further improvements extending the safety criterion. Finally, in Section 8 the application of termination criteria to general disjunctive programs with negated literals is presented.

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 g is associated with a fixed *arity*, denoted by $ar(g)$, which is a non-negative integer for predicate symbols and a natural number for function symbols.

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 $1 \leqslant i \leqslant m$; 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. The depth $d(u,t)$ of a simple term u in a term t that contains u is recursively defined as follows:

$$d(u,u) = 0,$$
$$d(u, f(t_1, ..., t_m)) = 1 + \max_{i \,:\, t_i \text{ contains } u} d(u, t_i).$$

The *depth of term* t, denoted by $d(t)$, is the maximal depth of all simple terms occurring in t.

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:

$$A_1 \vee ... \vee A_m \leftarrow B_1, ..., B_k, \neg C_1, ..., \neg C_n$$

where $m > 0$, $k \geqslant 0$, $n \geqslant 0$, and $A_1, ..., A_m, B_1, ..., B_k$, C_1, $..., C_n$ are atoms. The disjunction $A_1 \vee ... \vee A_m$ is called the *head* of r and is denoted by head(r); the conjunction $B_1, ..., B_k, \neg C_1, ..., \neg C_n$ is called the *body* of r and is denoted by body(r). The *positive* (resp. *negative*) *body* of r is the conjunction $B_1, ..., B_k$ (resp. $\neg C_1, ..., \neg C_n$) 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$^-(r)$) to also 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*. If a rule r is both normal and positive, then it is *standard*.

A *program* is a finite set of rules. A program is *normal* (resp. *positive*, *standard*) if every rule in it is normal (resp. positive, standard). 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*. For any atom A (resp. set of atoms, rule), $var(A)$ denotes the set of variables occurring in A.

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 $\mathscr{P}$ consists of all rules in $\mathscr{P}$ with p in the head. Predicate symbols are partitioned into two different classes: *base* predicate symbols, whose definition can contain only facts (called *database facts*), and *derived* predicate symbols, whose definition can contain any rule. Database facts are not shown in our examples as they are not relevant for the proposed criteria.

Given a program $\mathscr{P}$, a predicate p depends on a predicate q if there is a rule r in $\mathscr{P}$ 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. A rule r is said to be *recursive* if its body contains a predicate symbol mutually recursive with a predicate symbol in the head. Given a rule r, $r$body(r) denotes the set of body atoms whose predicate symbols are mutually recursive with the predicate symbol of an atom in the head. We say that r is *linear* if $|r\text{body}(r)| \leqslant 1$. We say that a recursive rule r defining a predicate p is *strongly* linear if it is linear, the recursive predicate symbol appearing in the body is p and there are no other recursive rules defining p. A predicate symbol p is said to be linear (resp. strongly linear) if all recursive rules defining p are linear (resp. strongly linear).

A *substitution* is a finite set of pairs $\theta = \{X_1/t_1, ..., X_n/t_n\}$ where $t_1, ..., t_n$ are terms and $X_1, ..., X_n$ are distinct variables not occurring in $t_1, \ldots, t_n$. If $\theta = \{X_1/t_1, ..., X_n/t_n\}$ is a substitution and T is a term or an atom, then $T\theta$ is the term or atom obtained from T by simultaneously replacing each occurrence of X_i in T by t_i ($1 \leqslant i \leqslant n$)—$T\theta$ is called an *instance* of T. Given a set S of terms (or atoms), $S\theta = \{T\theta \mid T \in S\}$. A substitution θ is a *unifier* for a finite set of terms (or atoms) S if $S\theta$ is a singleton. We say that a set of terms (or atoms) S *unifies* if there exists a unifier θ for S. Given two substitutions, $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\vartheta = \{Y_1/u_1, \ldots, Y_m/u_m\}$, their *composition*, denoted by $\theta \circ \vartheta$, is the substitution obtained from the set $\{X_1/t_1\vartheta, \ldots, X_n/t_n\vartheta, Y_1/u_1, \ldots, Y_m/u_m\}$ by removing every $X_i/t_i\vartheta$ such that $X_i = t_i\vartheta$ and every Y_j/u_j such that $Y_j \in \{X_1, \ldots, X_n\}$. A substitution θ is *more general* than a substitution ϑ if there exists a substitution η such that $\vartheta = \theta \circ \eta$. A unifier θ for a set S of terms (or atoms) is called a *most general unifier* (mgu) for S if it is more general than any other unifier for S. The mgu is unique modulo renaming of variables.

Semantics. Let $\mathscr{P}$ be a program. The *Herbrand universe* $H_{\mathscr{P}}$ of $\mathscr{P}$ is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in $\mathscr{P}$. The *Herbrand base* $B_{\mathscr{P}}$ of $\mathscr{P}$ is the set of ground atoms which can be built using predicate symbols appearing in $\mathscr{P}$ and ground terms of $H_{\mathscr{P}}$. A rule r' is a *ground instance* of a rule r in $\mathscr{P}$ if r' can be obtained from r by substituting every variable in r with some ground term in $H_{\mathscr{P}}$. We use ground(r) to denote the set of all ground instances of r and ground($\mathscr{P}$) to denote the set of all ground instances of the rules in $\mathscr{P}$, i.e. ground($\mathscr{P}$) $= \cup_{r \in \mathscr{P}}$ground($r$). An *interpretation* of $\mathscr{P}$ is any subset I of $B_{\mathscr{P}}$. The truth value of a ground atom A w.r.t. I, denoted by $\text{value}_I(A)$, is *true* if $A \in I$, *false* otherwise. The truth value of $\neg A$ w.r.t. I, denoted by $\text{value}_I(\neg A)$, is *true* if $A \notin I$, *false* otherwise. The truth value of a conjunction of ground literals $C = L_1, ..., L_n$ w.r.t. I is $\text{value}_I(C) = \min(\{\text{value}_I(L_i) \mid 1 \leqslant i \leqslant n\})$—here the ordering *false* $<$ *true* holds—whereas the truth value of a disjunction of ground literals $D = L_1 \vee ... \vee L_n$ w.r.t. I is $\text{value}_I(D) = \max(\{\text{value}_I(L_i) \mid 1 \leqslant i \leqslant n\})$; if $n = 0$, then $\text{value}_I(C) = \textit{true}$ and $\text{value}_I(D) = \textit{false}$. A ground rule r is *satisfied* by I, denoted by $I \models r$, if $\text{value}_I(\text{head}(r)) \geqslant \text{value}_I(\text{body}(r))$; 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 $\text{value}_I(\text{head}(r)) = \textit{true}$. An interpretation of $\mathscr{P}$ is a *model* of $\mathscr{P}$ if it satisfies every ground rule in *ground*($\mathscr{P}$). A model M of $\mathscr{P}$ is minimal if no proper subset of M is a model of $\mathscr{P}$. The set of minimal models of $\mathscr{P}$ is denoted by $\mathscr{M}\mathscr{M}(\mathscr{P})$.

Given an interpretation I of $\mathscr{P}$, let $\mathscr{P}^I$ denote the ground positive program derived from *ground*($\mathscr{P}$) 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 $\mathscr{P}$ if and only if $I \in \mathscr{M}\mathscr{M}(\mathscr{P}^I)$ (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991). The set of stable models of $\mathscr{P}$ is denoted by $\mathscr{S}\mathscr{M}(\mathscr{P})$. It is well known that stable models are minimal models (i.e.

$\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$). Furthermore, minimal and stable model semantics coincide for positive programs (i.e. $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$). A standard program has a unique minimal model, called *minimum model*.

Given a set of ground atoms S and a predicate g (resp. an atom A), $S[g]$ (resp. $S[A]$) denotes the set of g-atoms (resp. ground atoms unifying with A) in S. Analogously, for a given set M of sets of ground atoms, we shall use the following notations $M[g] = \{S[g] \mid S \in M\}$ and $M[A] = \{S[A] \mid S \in M\}$. Given a set of ground atoms S, and a set G of predicates symbols, then $S[G] = \cup_{g \in G} S[g]$.

Argument graph. Given an n-ary predicate p, $p[i]$ denotes the ith argument of p, for $1 \leqslant i \leqslant n$. 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 $\mathcal{P}$ is denoted by $args(\mathcal{P})$; analogously, $args_b(\mathcal{P})$ and $args_d(\mathcal{P})$ denote the sets of all base and derived arguments, respectively.

For any program $\mathcal{P}$ and n-ary predicate p occurring in $\mathcal{P}$, an argument $p[i]$, with $1 \leqslant i \leqslant n$, is associated with the set of values it can take during the evaluation; this domain, called *active domain* of $p[i]$, is denoted by $AD(p[i]) = \{t_i | p(t_1, \ldots, t_n) \in M \wedge M \in \mathcal{SM}(\mathcal{P})\}$. An argument $p[i]$ is said to be *limited* iff $AD(p[i])$ is finite.

The *argument graph* of a program $\mathcal{P}$, denoted by $G(\mathcal{P})$, is a directed graph whose nodes are $args(\mathcal{P})$ (i.e. the arguments of $\mathcal{P}$), and there is an edge from $q[j]$ to $p[i]$, denoted by $(q[j], p[i])$, iff there is a rule $r \in \mathcal{P}$ such that:

1. an atom $p(t_1, ..., t_n)$ appears in head(r),
2. an atom $q(u_1, ..., u_m)$ appears in body$^+(r)$, and
3. terms t_i and u_j have a common variable.

Consider, for instance, program P_1 of Example 1. $G(P_1) = (args(P_1), E)$, where $args(P_1) = \{\texttt{input[1]}, \texttt{list[1]}, \texttt{count[1]}, \texttt{count[2]}\}$, whereas, considering the occurrences of variables in the rules of P_1 we have that $E = \{$`(input[1],list[1])`, `(list[1],list[1])`, `(list[1],count[1])`, `(count[1],count[1])`, `(count[2],count[2])`$\}$.

Labeled directed graphs. In the following we will also consider labeled directed graphs, i.e. directed graphs with labeled edges. In this case we represent an edge from a to b as a triple (a, b, l), where l denotes the label.

A *path* π from a_1 to b_m in a possibly labeled directed graph is a non-empty sequence $(a_1, b_1, l_1), \ldots, (a_m, b_m, l_m)$ of its edges s.t. $b_i = a_{i+1}$ for all $1 \leqslant i < m$; if the first and last nodes coincide (i.e. $a_1 = b_m$), then π is called a *cyclic path*. In the case where the indication of the starting edge is not relevant, we will call a cyclic path a *cycle*.

We say that a node *a depends on* a node b in a graph iff there is a path from b to a in that graph. Moreover, we say that *a depends on* a cycle π iff it depends on a node b appearing in π. Clearly, nodes belonging to cycle π depend on π.

3 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.

An *argument ranking* for a program $\mathcal{P}$ is a partial function ϕ from $args(\mathcal{P})$ to non-negative integers, called *ranks*, such that, for every rule r of $\mathcal{P}$, every atom $p(t_1, \ldots, t_n)$ occurring in the head of r, and every variable X occurring in a term t_i, if $\phi(p[i])$ is defined, then $\text{body}^+(r)$ contains an atom $q(u_1, \ldots, u_m)$ such that X occurs in a term u_j, $\phi(q[j])$ is defined, and the following condition is satisfied:

$$\phi(p[i]) - \phi(q[j]) \geqslant d(X, t_i) - d(X, u_j). \tag{1}$$

A program $\mathcal{P}$ is said to be *argument-restricted* if it has an argument ranking assigning ranks to all arguments of $\mathcal{P}$.

Example 2
Consider the following program P_2, where $\mathtt{b}$ is a base predicate:

$$\begin{aligned} r_1 &: \mathtt{p(f(X))} \leftarrow \mathtt{p(X), b(X)}.\\ r_2 &: \mathtt{t(f(X))} \leftarrow \mathtt{p(X)}.\\ r_3 &: \mathtt{s(X)} \leftarrow \mathtt{t(f(X))}. \end{aligned}$$

This program has an argument ranking ϕ, where $\phi(\mathtt{b[1]})= 0$, $\phi(\mathtt{p[1]})= 1$, $\phi(\mathtt{t[1]})= 2$, and $\phi(\mathtt{s[1]})= 1$. Consequently, P_2 is argument-restricted. □

Intuitively, the rank of an argument is an estimation of the depth of terms that may occur in it. In particular, let d_1 be the rank assigned to a given argument $p[i]$ and let d_2 be the maximal depth of terms occurring in the database facts. Then $d_1 + d_2$ gives an upper bound of the depth of terms that may occur in $p[i]$ during the program evaluation. Different argument rankings may satisfy condition (1). A function assigning minimum ranks to arguments is denoted by $\phi_{\min}$.

Minimum ranking. We define a monotone operator Ω that takes as input a function ϕ over arguments and gives as output a function over arguments that gives an upper bound of the depth of terms.

More specifically, we define $\Omega(\phi)(p[i])$ as

$$\max(\ \max\{D(p(t_1, \ldots, t_n), r, i, X) \mid r \in \mathcal{P} \wedge p(t_1, \ldots, t_n) \in \text{head}(r) \wedge X \text{ occurs in } t_i\}, 0)$$

where $D(p(t_1, \ldots, t_n), r, i, X)$ is defined as

$$\min\{d(X, t_i) - d(X, u_j) + \phi(q[j]) \mid q(u_1, \ldots, u_m) \in \text{body}^+(r) \wedge\ X \text{ occurs in } u_j\}.$$

In order to compute $\phi_{\min}$ we compute the fixpoint of Ω starting from the function ϕ_0 that assigns 0 to all arguments. In particular, we have:

$$\begin{aligned} &\phi_0(p[i]) = 0;\\ &\phi_k(p[i]) = \Omega(\phi_{k-1})(p[i]) = \Omega^k(\phi_0)(p[i]). \end{aligned}$$

The function $\phi_{\min}$ is defined as follows:

$$\phi_{\min}(p[i]) = \begin{cases} \Omega^k(\phi_0)(p[i]) & \textit{if } \exists k \text{ (finite) s.t. } \Omega^k(\phi_0)(p[i]) = \Omega^\infty(\phi_0)(p[i]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We denote the set of *restricted arguments* of $\mathcal{P}$ as $AR(\mathcal{P}) = \{p[i] \mid p[i] \in args(\mathcal{P}) \wedge \phi_{\min}(p[i]) \text{ is defined}\}$. Clearly, from definition of $\phi_{\min}$, it follows that all restricted arguments are limited. Observe that $\mathcal{P}$ is *argument-restricted* iff $AR(\mathcal{P}) = args(\mathcal{P})$.

Example 3
Consider again program P_2 from Example 2. The following table shows the first four iterations of Ω starting from the base ranking function ϕ_0:

	ϕ_0	$\phi_1 = \Omega(\phi_0)$	$\phi_2 = \Omega(\phi_1)$	$\phi_3 = \Omega(\phi_2)$	$\phi_4 = \Omega(\phi_3)$
`b[1]`	0	0	0	0	0
`p[1]`	0	1	1	1	1
`t[1]`	0	1	2	2	2
`s[1]`	0	0	0	1	1

Since $\Omega(\phi_3) = \Omega(\phi_2)$, further applications of Ω provide the same result. Consequently, $\phi_{\min}$ coincides with ϕ_3 and defines ranks for all arguments of P_2. □

Let $M = |args(\mathscr{P})| \times d_{\max}$, where $d_{\max}$ is the largest depth of terms occurring in the heads of rules of $\mathscr{P}$. One can determine whether $\mathscr{P}$ is argument-restricted by iterating Ω starting from ϕ_0 until:

(1) one of the values of $\Omega^k(\phi_0)$ exceeds M, in such a case $\mathscr{P}$ is not argument-restricted;
(2) $\Omega^{k+1}(\phi_0) = \Omega^k(\phi_0)$, in such a case $\phi_{\min}$ coincides with ϕ_k, $\phi_{\min}$ is total, and $\mathscr{P}$ is argument-restricted.

Observe that if the program is not argument-restricted the first condition is verified with $k \leqslant M \times |args(\mathscr{P})| \leqslant M^2$, as at each iteration the value assigned to at least one argument is changed. Thus, the problem of deciding whether a given program $\mathscr{P}$ is argument-restricted is in *P Time*. In the following section we will show that the computation of restricted arguments can be done in polynomial time also when $\mathscr{P}$ is not argument-restricted (see Proposition 1).

4 Γ-acyclic programs

In this section we exploit the role of function symbols for checking program termination under bottom-up evaluation. Starting from this section, we will consider standard logic programs. Only in Section 8 we will refer to general programs, as it discusses how termination criteria defined for standard programs can be applied to general disjunctive logic programs with negative literals. We also assume that if the same variable X appears in two terms occurring in the head and body of a rule respectively, then at most one of the two terms is a complex term and that the nesting level of complex terms is at most one. As we will see in Section 8, there is no real restriction in such an assumption as every program could be rewritten into an equivalent program satisfying such a condition.

The following example shows a program admitting a finite minimum model, but the argument-restricted criterion is not able to detect it. Intuitively, the definition of argument-restricted programs does not take into account the possible presence of different function symbols in the program that may prohibit the propagation of values in some rules and, consequently, guarantee the termination of the bottom-up computation.

Example 4
Consider the following program P_4:

$$\begin{aligned} r_0 &: \mathtt{s(X)} \leftarrow \mathtt{b(X)}.\\ r_1 &: \mathtt{r(f(X))} \leftarrow \mathtt{s(X)}.\\ r_2 &: \mathtt{q(f(X))} \leftarrow \mathtt{r(X)}.\\ r_3 &: \mathtt{s(X)} \leftarrow \mathtt{q(g(X))}. \end{aligned}$$

where b is a base predicate symbol. The program is not argument-restricted since the argument ranking function $\phi_{\min}$ cannot assign any value to r[1], q[1], and s[1]. However the bottom-up computation always terminates, independently from the database instance. □

In order to represent the propagation of values among arguments, we introduce the concept of *labeled argument graphs*. Intuitively, it is an extension of the argument graph where each edge has a label describing how the term propagated from one argument to another changes. Arguments that are not dependent on a cycle can propagate a finite number of values and, therefore, are limited.

Since the active domain of limited arguments is finite, we can delete edges ending in the corresponding nodes from the labeled argument graph. Then, the resulting graph, called *propagation graph*, is deeply analyzed to identify further limited arguments.

Definition 1 (*Labeled argument graph*)
Let $\mathscr{P}$ be a program. The *labeled argument graph* $\mathscr{G}_L(\mathscr{P})$ is a labeled directed graph $(args(\mathscr{P}), E)$ where E is a set of labeled edges defined as follows. For each pair of nodes $p[i], q[j] \in args(\mathscr{P})$ such that there is a rule r with $\text{head}(r) = p(v_1, \ldots, v_n)$, $q(u_1, \ldots, u_m) \in \text{body}(r)$, and terms u_j and v_i have a common variable X, there is an edge $(q[j], p[i], \alpha) \in E$ such that

- $\alpha = \epsilon$ if $u_j = v_i = X$,
- $\alpha = f$ if $u_j = X$ and $v_i = f(..., X, ...)$,
- $\alpha = \overline{f}$ if $u_j = f(..., X, ...)$ and $v_i = X$. □

In the definition above, the symbol ϵ denotes the empty label which concatenated to a string does not modify the string itself, that is, for any string s, $s\epsilon = \epsilon s = s$.

The labeled argument graph of program P_4 is shown in Figure 1 (left). The edges of this graph represent how the propagation of values occurs. For instance, edge $(\mathtt{b[1]}, \mathtt{s[1]}, \epsilon)$ states that a term t is propagated without changes from b[1] to s[1] if rule r_0 is applied; analogously, edge $(\mathtt{s[1]}, \mathtt{r[1]}, \mathtt{f})$ states that starting from a term t in s[1] we obtain f(t) in r[1] if rule r_1 is applied, whereas edge $(\mathtt{q[1]}, \mathtt{s[1]}, \overline{\mathtt{g}})$ states that starting from a term g(t) in q[1] we obtain t in s[1] if rule r_3 is applied.

Given a path π in $\mathscr{G}_L(\mathscr{P})$ of the form $(a_1, b_1, \alpha_1), \ldots, (a_m, b_m, \alpha_m)$, we denote with $\lambda(\pi)$ the string $\alpha_1 \ldots \alpha_m$. We say that π *spells a string* w if $\lambda(\pi) = w$. Intuitively, the string $\lambda(\pi)$ describes a sequence of function symbols used to compose and decompose complex terms during the propagation of values among the arguments in π.

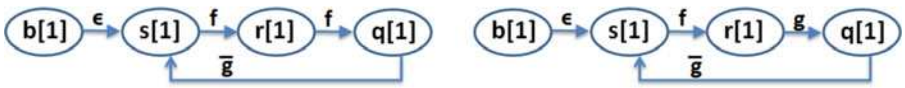

Fig. 1. (Colour online) Labeled argument graphs of programs P_4 (left) and P_5 (right).

Example 5

Consider program P_5 derived from program P_4 of Example 4 by replacing rule r_2 with the rule $\mathtt{q(g(X))} \leftarrow \mathtt{r(X)}$. The labeled argument graph $\mathscr{G}_L(P_5)$ is reported in Figure 1 (right). Considering the cyclic path $\pi = (\mathtt{s[1]},\mathtt{r[1]},\mathtt{f})$, $(\mathtt{r[1]},\mathtt{q[1]},\mathtt{g})$, $(\mathtt{q[1]},\mathtt{s[1]},\overline{\mathtt{g}})$, $\lambda(\pi) = \mathtt{fg}\overline{\mathtt{g}}$ represents the fact that starting from a term t in s[1] we may obtain the term f(t) in r[1], then we may obtain term g(f(t)) in q[1], and term f(t) in s[1], and so on. Since we may obtain a larger term in s[1], the arguments depending on this cyclic path may not be limited.

Consider now program P_4, whose labeled argument graph is shown in Figure 1 (left), and the cyclic path $\pi' = (\mathtt{s[1]},\mathtt{r[1]},\mathtt{f})$, $(\mathtt{r[1]},\mathtt{q[1]},\mathtt{f})$, $(\mathtt{q[1]},\mathtt{s[1]},\overline{\mathtt{g}})$. Observe that starting from a term t in s[1] we may obtain term f(t) in r[1] (rule r_1), then we may obtain term f(f(t)) in q[1] (rule r_2). At this point the propagation in this cyclic path terminates since the head atom of rule r_2 containing term f(X) cannot match with the body atom of rule r_3 containing term g(X). The string $\lambda(\pi') = \mathtt{ff}\overline{\mathtt{g}}$ represents the propagation described above. Observe that for this program all arguments are limited. □

Let π be a path from $p[i]$ to $q[j]$ in the labeled argument graph. Let $\hat{\lambda}(\pi)$ be the string obtained from $\lambda(\pi)$ by iteratively eliminating pairs of the form $\alpha\overline{\alpha}$ until the resulting string cannot be further reduced. If $\hat{\lambda}(\pi) = \epsilon$, then starting from a term t in $p[i]$ we obtain the same term t in $q[j]$. Consequently, if $\hat{\lambda}(\pi)$ is a non-empty sequence of function symbols $f_{i_1}, f_{i_2}, \ldots, f_{i_k}$, then starting from a term t in $p[i]$ we may obtain a larger term in $q[j]$. For instance, if $k = 2$ and f_{i_1} and f_{i_2} are of arity one, we may obtain $f_{i_2}(f_{i_1}(t))$ in $q[j]$. Based on this intuition we introduce now a grammar $\Gamma_{\mathscr{P}}$ in order to distinguish the sequences of function symbols used to compose and decompose complex terms in a program $\mathscr{P}$, such that starting from a given term we obtain a larger term.

Given a program $\mathscr{P}$, we denote with $F_{\mathscr{P}} = \{f_1, ..., f_m\}$ the set of function symbols occurring in $\mathscr{P}$, whereas $\overline{F}_{\mathscr{P}} = \{\overline{f} \mid f \in F_{\mathscr{P}}\}$ and $T_{\mathscr{P}} = F_{\mathscr{P}} \cup \overline{F}_{\mathscr{P}}$.

Definition 2

Let $\mathscr{P}$ be a program, the *grammar* $\Gamma_{\mathscr{P}}$ is a 4-tuple $(N, T_{\mathscr{P}}, R, S)$, where $N = \{S, S_1, S_2\}$ is the set of non-terminal symbols, S is the start symbol, and R is the set of production rules defined below:

1. $S \rightarrow S_1\, f_i\, S_2, \qquad \forall f_i \in F_{\mathscr{P}};$
2. $S_1 \rightarrow f_i\, S_1\, \overline{f}_i\, S_1 \mid \epsilon, \qquad \forall f_i \in F_{\mathscr{P}};$
3. $S_2 \rightarrow S_1\, S_2 \mid f_i\, S_2 \mid \epsilon, \qquad \forall f_i \in F_{\mathscr{P}}.$ □

The language $\mathscr{L}(\Gamma_{\mathscr{P}})$ is the set of strings generated by $\Gamma_{\mathscr{P}}$.

Example 6
Let $F_{\mathscr{P}} = \{\mathtt{f},\mathtt{g},\mathtt{h}\}$ be the set of function symbols occurring in a program $\mathscr{P}$. Then strings $\mathtt{f}$, $\mathtt{fg\bar{g}}$, $\mathtt{g\bar{g}f}$, $\mathtt{fg\bar{g}h\bar{h}}$, $\mathtt{fhg\bar{g}\bar{h}}$ belong to $\mathscr{L}(\Gamma_{\mathscr{P}})$ and represent, assuming that $\mathtt{f}$ is a unary function symbol, different ways to obtain term $\mathtt{f(t)}$ starting from term $\mathtt{t}$. □

Note that only if a path π spells a string $w \in \mathscr{L}(\Gamma_{\mathscr{P}})$, then starting from a given term in the first node of π we may obtain a larger term in the last node of π. Moreover, if this path is cyclic, then the arguments depending on it may not be limited. On the other hand, all arguments not depending on a cyclic path π spelling a string $w \in \mathscr{L}(\Gamma_{\mathscr{P}})$ are limited.

Given a program $\mathscr{P}$ and a set of arguments $\mathscr{S}$ recognized as limited by a specific criterion, the *propagation graph* of $\mathscr{P}$ w.r.t. $\mathscr{S}$, denoted by $\Delta(\mathscr{P},\mathscr{S})$, consists of the subgraph derived from $\mathscr{G}_L(\mathscr{P})$ by deleting edges ending in a node of $\mathscr{S}$. Although we can consider any set $\mathscr{S}$ of limited arguments, in the following we assume $\mathscr{S} = AR(\mathscr{P})$ and, for the simplicity of notation, we denote $\Delta(\mathscr{P}, AR(\mathscr{P}))$ as $\Delta(\mathscr{P})$. Even if more general termination criteria have been defined in the literature, here we consider the *AR* criterion since it is the most general among those so far proposed having polynomial time complexity.

Definition 3 (Γ-*acyclic arguments and* Γ-*acyclic programs*)
Given a program $\mathscr{P}$, the set of its Γ-*acyclic arguments*, denoted by $\Gamma A(\mathscr{P})$, consists of all arguments of $\mathscr{P}$ not depending on a cyclic path in $\Delta(\mathscr{P})$ spelling a string of $\mathscr{L}(\Gamma_{\mathscr{P}})$. A program $\mathscr{P}$ is called Γ-*acyclic* if $\Gamma A(\mathscr{P}) = args(\mathscr{P})$, i.e. if there is no cyclic path in $\Delta(\mathscr{P})$ spelling a string of $\mathscr{L}(\Gamma_{\mathscr{P}})$. We denote the class of Γ-acyclic programs $\Gamma\mathscr{A}$. □

Clearly, $AR(\mathscr{P}) \subseteq \Gamma A(\mathscr{P})$, i.e. the set of restricted arguments is contained in the set of Γ-acyclic arguments. As a consequence, the set of argument-restricted programs is a subset of the set of Γ-acyclic programs. Moreover, the containment is strict, as there exist programs that are Γ-acyclic, but not argument-restricted. For instance, program P_4 from Example 4 is Γ-acyclic, but not argument-restricted. Indeed, all cyclic paths in $\Delta(P_4)$ do not spell strings belonging to the language $\mathscr{L}(\Gamma_{P_4})$.

The importance of considering the propagation graph instead of the labeled argument graph in Definition 3 is shown in the following example.

Example 7
Consider program P_7 below obtained from P_4 by adding rules r_4 and r_5.

$$\begin{aligned}
r_0 &: \mathtt{s(X)} \leftarrow \mathtt{b(X)}.\\
r_1 &: \mathtt{r(f(X))} \leftarrow \mathtt{s(X)}.\\
r_2 &: \mathtt{q(f(X))} \leftarrow \mathtt{r(X)}.\\
r_3 &: \mathtt{s(X)} \leftarrow \mathtt{q(g(X))}.\\
r_4 &: \mathtt{n(f(X))} \leftarrow \mathtt{s(X)}, \mathtt{b(X)}.\\
r_5 &: \mathtt{s(X)} \leftarrow \mathtt{n(X)}.
\end{aligned}$$

The corresponding labeled argument graph $\mathscr{G}_L(P_7)$ and propagation graph $\Delta(P_7)$ are reported in Figure 2. Observe that arguments $\mathtt{n[1]}$ and $\mathtt{s[1]}$ are involved in

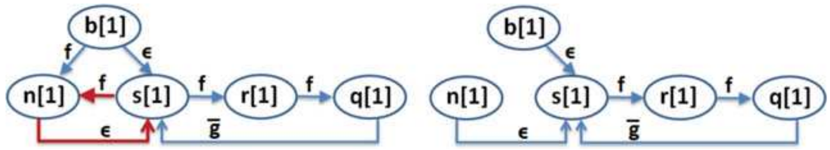

Fig. 2. (Colour online) Labeled argument graph (left) and propagation graph (right) of program P_7.

the red cycle in the labeled argument graph $\mathscr{G}_L(P_7)$ spelling a string of $\mathscr{L}(\Gamma_{P_7})$. At the same time this cycle is not present in the propagation graph $\Delta(P_7)$ since $AR(P_7) = \{\mathtt{b}[1], \mathtt{n}[1]\}$ and the program is Γ-acyclic. □

Theorem 1

Given a program $\mathscr{P}$,

1. all arguments in $\Gamma A(\mathscr{P})$ are limited;
2. if $\mathscr{P}$ is Γ-acyclic, then $\mathscr{P}$ is finitely ground.

Proof

(1) As previously recalled, arguments in $AR(\mathscr{P})$ are limited. Let us now show that all arguments in $\Gamma A(\mathscr{P}) \setminus AR(\mathscr{P})$ are limited too. Suppose by contradiction that $q[k] \in \Gamma A(\mathscr{P}) \setminus AR(\mathscr{P})$ is not limited. Observe that depth of terms that may occur in $q[k]$ depends on the paths in the propagation graph $\Delta(\mathscr{P})$ that ends in $q[k]$. In particular, this depth may be infinite only if there is a path π from an argument $p[i]$ to $q[k]$ (not necessarily distinct from $p[i]$), such that $\hat{\lambda}(\pi)$ is a string of an infinite length composed by symbols in F_P. But this is possible only if this path contains a cycle spelling a string in $\mathscr{L}(\Gamma_{\mathscr{P}})$. Thus we obtain contradiction with Definition 3.

(2) From the previous proof, it follows that every argument in the Γ-acyclic program can take values only from a finite domain. Consequently, the set of all possible ground terms derived during the grounding process is finite and every Γ-acyclic program is finitely ground.

□

From the previous theorem we can also conclude that all Γ-acyclic programs admit a finite minimum model, as this is a property of finitely ground programs.

We conclude by observing that since the language $\mathscr{L}(\Gamma_{\mathscr{P}})$ is context-free, the analysis of paths spelling strings in $\mathscr{L}(\Gamma_{\mathscr{P}})$ can be carried out using pushdown automata.

As $\Gamma_{\mathscr{P}}$ is context free, the language $\mathscr{L}(\Gamma_{\mathscr{P}})$ can be recognized by means of a pushdown automaton $M = (\{q_0, q_F\}, T_{\mathscr{P}}, \Lambda, \delta, q_0, Z_0, \{q_F\})$, where q_0 is the initial state, q_F is the final state, $\Lambda = \{Z_0\} \cup \{F_i | f_i \in F_{\mathscr{P}}\}$ is the stack alphabet, Z_0 is the initial stack symbol, and δ is the transition function defined as follows:

1. $\delta(q_0, f_i, Z_0) = (q_F, F_i Z_0), \quad \forall f_i \in F_{\mathscr{P}},$
2. $\delta(q_F, f_i, F_j) = (q_F, F_i F_j), \quad \forall f_i \in F_{\mathscr{P}},$

3. $\delta(q_F, \overline{f}_j, F_j) = (q_F, \epsilon), \qquad \forall f_i \in F_{\mathscr{P}}$.

The input string is recognized if after having scanned the entire string the automaton is in state q_F and the stack contains at least one symbol F_i.

A path π is called:

- *increasing*, if $\hat{\lambda}(\pi) \in \mathscr{L}(\Gamma_{\mathscr{P}})$,
- *flat*, if $\hat{\lambda}(\pi) = \epsilon$,
- *failing*, otherwise.

It is worth noting that $\lambda(\pi) \in \mathscr{L}(\Gamma_{\mathscr{P}})$ iff $\hat{\lambda}(\pi) \in \mathscr{L}(\Gamma_{\mathscr{P}})$ as function $\hat{\lambda}$ emulates the pushdown automaton used to recognize $\mathscr{L}(\Gamma_{\mathscr{P}})$. More specifically, for any path π and relative string $\lambda(\pi)$ we have that:

- if π is increasing, then the pushdown automaton recognizes the string $\lambda(\pi)$ in state q_F and the stack contains a sequence of symbols corresponding to the symbols in $\hat{\lambda}(\pi)$ plus the initial stack symbol Z_0;
- if π is flat, then the pushdown automaton does not recognize the string $\lambda(\pi)$; moreover, the entire input string is scanned, but the stack contains only the symbol Z_0;
- if $\hat{\lambda}(\pi)$ is failing, then the pushdown automaton does not recognize the string $\lambda(\pi)$ as it goes in an error state.

Complexity. Concerning the complexity of checking whether a program is Γ-acyclic, we first introduce definitions and results that will be used hereafter. We start by introducing the notion of size of a logic program.

We assume that simple terms have constant size and, therefore, the size of a complex term $f(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are simple terms, is bounded by $O(k)$. Analogously, the size of an atom $p(t_1, \ldots, t_n)$ is given by the sum of the sizes of the t_i's, whereas the size of a conjunction of atoms (resp. rule, program) is given by the sum of the sizes of its atoms. That is, we identify for a program $\mathscr{P}$ the following parameters: n_r is the number of rules of $\mathscr{P}$, n_b is the maximum number of atoms in the body of rules of $\mathscr{P}$, a_p is the maximum arity of predicate symbols occurring in $\mathscr{P}$, and a_f is the maximum arity of function symbols occurring in $\mathscr{P}$. We assume that the size of $\mathscr{P}$, denoted by $size(\mathscr{P})$, is bounded by $O(n_r \times n_b \times a_p \times a_f)$. Finally, since checking whether a program is terminating requires to read the program, we assume that the program has been already scanned and stored using suitable data structures. Thus, all the complexity results presented in the rest of the paper do not take into account the cost of scanning and storing the input program. We first introduce a tighter bound for the complexity of computing $AR(\mathscr{P})$.

Proposition 1

For any program $\mathscr{P}$, the time complexity of computing $AR(\mathscr{P})$ is bounded by $O(|args(\mathscr{P})|^3)$.

Proof

Assume that $n = |args(\mathscr{P})|$ is the total number of arguments of $\mathscr{P}$. First, it is important to observe the connection between the behavior of operator Ω and the

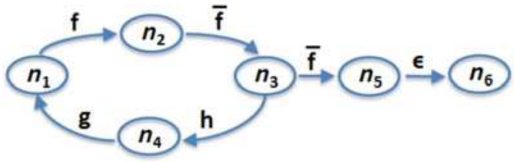

Fig. 3. (Colour online) Propagation graph $\Delta(\mathscr{P})$.

structure of the labeled argument graph $\mathscr{G}_L(\mathscr{P})$. In particular, if the applications of the operator Ω change the rank of an argument $q[i]$ from 0 to k, then there is a path from an argument to $q[i]$ in $\mathscr{G}_L(\mathscr{P})$, where the number of edges labeled with some positive function symbol minus the number of edges labeled with some negative function symbol is at least k. Given a cycle in a labeled argument graph, let us call it *affected* if the number of edges labeled with some positive function symbol is greater than the number of edges labeled with some negative function symbol.

If an argument is not restricted, it is involved in or depends on an affected cycle. On the other hand, if after an application of Ω the rank assigned to an argument exceeds n, this argument is not restricted (Lierler and Lifschitz 2009). Recall that we are assuming that $d_{\max} = 1$ and, therefore, $M = n \times d_{\max} = n$.

Now let us show that after $2n^2 + n$ iterations of Ω all not restricted arguments exceed rank n. Consider an affected cycle and suppose that it contains k arguments, whereas the number of arguments depending on this cycle, but not belonging to it is m. Obviously, $k + m \leqslant n$. All arguments involved in this cycle change their rank by at least one after k iterations of Ω. Thus their ranks will be greater than $n + m$ after $(n + m + 1) \times k$ iterations. The arguments depending on this cycle, but not belonging to it, need at most another m iterations to reach the rank greater than n. Thus all unrestricted arguments exceed the rank n in $(n+m+1) \times k+m$ iterations of Ω. Since $(n + m + 1) \times k + m = nk + mk + (k + m) \leqslant 2n^2 + n$, the restricted arguments are those that at step $2n^2 + n$ do not exceed rank n. It follows that the complexity of computing $AR(\mathscr{P})$ is bounded by $O(n^3)$ because we have to do $O(n^2)$ iterations and, for each iteration we have to check the rank of n arguments. □

In order to study the complexity of computing Γ-acyclic arguments of a program we introduce a directed (not labeled) graph obtained from the propagation graph.

Definition 4 (*Reduction of* $\Delta(\mathscr{P})$)
Given a program $\mathscr{P}$, the *reduction* of $\Delta(\mathscr{P})$ is a directed graph $\Delta_R(\mathscr{P})$ whose nodes are the arguments of $\mathscr{P}$ and there is an edge $(p[i], q[j])$ in $\Delta_R(\mathscr{P})$ iff there is a path π from $p[i]$ to $q[j]$ in $\Delta(\mathscr{P})$ such that $\hat{\lambda}(\pi) \in F_{\mathscr{P}}$. □

The reduction $\Delta_R(\mathscr{P})$ of the propagation graph $\Delta(\mathscr{P})$ from Figure 3 is shown in Figure 4. It is simple to note that for each path in $\Delta(\mathscr{P})$ from node $p[i]$ to node $q[j]$ spelling a string of $\mathscr{L}(\Gamma_{\mathscr{P}})$ there exists a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathscr{P})$ and vice versa. As shown in the lemma below, this property always holds.

Lemma 1
Given a program $\mathscr{P}$ and arguments $p[i], q[j] \in args(\mathscr{P})$, there exists a path in $\Delta(\mathscr{P})$ from $p[i]$ to $q[j]$ spelling a string of $\mathscr{L}(\Gamma_{\mathscr{P}})$ iff there is a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathscr{P})$.

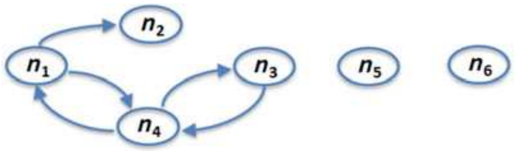

Fig. 4. (Colour online) Reduction $\Delta_R(\mathscr{P})$ of propagation graph $\Delta(\mathscr{P})$

Fig. 4-Colour online, B/W in print

Proof

($\Rightarrow$) Suppose there is a path π from $p[i]$ to $q[j]$ in $\Delta(\mathscr{P})$ such that $\lambda(\pi) \in \mathscr{L}(\Gamma_{\mathscr{P}})$. Then $\hat{\lambda}(\pi)$ is a non-empty string, say $f_1, \ldots, f_k$, where $f_i \in F_{\mathscr{P}}$ for $i \in [1, \ldots, k]$. Consequently, π can be seen as a sequence of subpaths $\pi_1, \ldots, \pi_k$, such that $\hat{\lambda}(\pi_i) = f_i$ for $i \in [1, \ldots, k]$. Thus, from the definition of the reduction of $\Delta(\mathscr{P})$, there is a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathscr{P})$ whose length is equal to $|\hat{\lambda}(\pi)|$.
($\Leftarrow$) Suppose there is a path $(n_1, n_2) \ldots (n_k, n_{k+1})$ from n_1 to n_{k+1} in $\Delta_R(\mathscr{P})$. From the definition of the reduction of $\Delta(\mathscr{P})$, for each edge (n_i, n_{i+1}) there is a path, say π_i, from n_i to n_{i+1} in $\Delta(\mathscr{P})$ such that $\hat{\lambda}(\pi_i) \in F_{\mathscr{P}}$. Consequently, there is a path from n_1 to n_{k+1} in $\Delta(\mathscr{P})$, obtained as a sequence of paths $\pi_1, \ldots, \pi_k$ whose string is simply $\lambda(\pi_1), \ldots, \lambda(\pi_k)$. Since $\hat{\lambda}(\pi_i) \in F_{\mathscr{P}}$ implies that $\lambda(\pi_i) \in \mathscr{L}(\Gamma_{\mathscr{P}})$, for every $1 \leqslant i \leqslant k$, we have that $\lambda(\pi_1), \ldots, \lambda(\pi_k)$ belongs also to $\mathscr{L}(\Gamma_{\mathscr{P}})$. $\square$

Proposition 2

Given a program $\mathscr{P}$, the time complexity of computing the reduction $\Delta_R(\mathscr{P})$ is bounded by $O(|args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$.

Proof

The construction of $\Delta_R(\mathscr{P})$ can be performed as follows. First, we compute all the paths π in $\Delta(\mathscr{P})$ such that $|\hat{\lambda}(\pi)| \leqslant 1$. To do so, we use a slight variation of the Floyd–Warshall's transitive closure of $\Delta(\mathscr{P})$ which is defined by the following recursive formula. Assume that each node of $\Delta(\mathscr{P})$ is numbered from 1 to $n = |args(\mathscr{P})|$, then we denote with $\mathrm{path}(i, j, \alpha, k)$ the existence of a path π from node i to node j in $\Delta(\mathscr{P})$ such that $\hat{\lambda}(\pi) = \alpha$, $|\alpha| \leqslant 1$ and π may go only through nodes in $\{1, \ldots, k\}$ (except for i and j).

The set of atoms $\mathrm{path}(i, j, \alpha, k)$, for all values $1 \leqslant i, j \leqslant n$, can be derived iteratively as follows:

- (base case: $k = 0$) $\mathrm{path}(i, j, \alpha, 0)$ holds if there is an edge (i, j, α) in $\Delta(\mathscr{P})$,
- (inductive case: $0 < k \leqslant n$) $\mathrm{path}(i, j, \alpha, k)$ holds if
  - — $\mathrm{path}(i, j, \alpha, k-1)$ holds, or
  - — $\mathrm{path}(i, k, \alpha_1, k-1)$ and $\mathrm{path}(k, j, \alpha_2, k-1)$ hold, $\alpha = \alpha_1\alpha_2$ and $|\alpha| \leqslant 1$.

Note that in order to compute all the possible atoms $\mathrm{path}(i, j, \alpha, k)$, we need to first initialize every base atom $\mathrm{path}(i, j, \alpha, 0)$ with cost bounded by $O(n^2 \times |F_{\mathscr{P}}|)$, as this is the upper bound for the number of edges in $\Delta(\mathscr{P})$. Then, for every $1 \leqslant k \leqslant n$, we need to compute all paths, $\mathrm{path}(i, j, \alpha, k)$, thus requiring a cost bounded by $O(n^3 \times |F_{\mathscr{P}}|)$ operations. The whole procedure will require $O(n^3 \times |F_{\mathscr{P}}|)$ operations. Since we have computed all possible paths π in $\Delta(\mathscr{P})$ such that $|\hat{\lambda}(\pi)| \leqslant 1$, we can obtain all the edges (i, j) of $\Delta_R(\mathscr{P})$ (according to Definition 4) by simply selecting the atoms path(i, j, α, k) with $\alpha \in F_{\mathscr{P}}$, whose cost is bounded by $O(n^2 \times |F_{\mathscr{P}}|)$. Then, the time complexity of constructing $\Delta_R(\mathscr{P})$ is $O(n^3 \times |F_{\mathscr{P}}|)$. □

Theorem 2
The complexity of deciding whether a program $\mathscr{P}$ is Γ-acyclic is bounded by $O(|args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$.

Proof
Assume that $n = |args(\mathscr{P})|$ is the total number of arguments of $\mathscr{P}$. To check whether $\mathscr{P}$ is Γ-acyclic it is sufficient to first compute the set of restricted arguments $AR(\mathscr{P})$ which requires time $O(n^3)$ from Proposition 1. Then, we need to construct the propagation graph $\Delta(\mathscr{P})$, for which the maximum number of edges is $n^2 \times (|F_{\mathscr{P}}| + |\overline{F}_{\mathscr{P}}| + 1)$, then it can be constructed in time $O(n^2 \times |F_{\mathscr{P}}|)$ (recall that we are not taking into account the cost of scanning and storing the program). Moreover, starting from $\Delta(\mathscr{P})$, we need to construct $\Delta_R(\mathscr{P})$, which requires time $O(n^3 \times |F_{\mathscr{P}}|)$ (*cf.* Proposition 2) and then, following Lemma 1, we need to check whether $\Delta_R(\mathscr{P})$ is acyclic. Verifying whether $\Delta_R(\mathscr{P})$ is acyclic can be done by means of a simple traversal of $\Delta_R(\mathscr{P})$ and checking if a node is visited more than once. The complexity of a depth-first traversal of a graph is well known to be $O(|E|)$ where E is the set of edges of the graph. Since the maximum number of edges of $\Delta_R(\mathscr{P})$ is by definition $n^2 \times |F_{\mathscr{P}}|$, the traversal of $\Delta_R(\mathscr{P})$ can be done in time $O(n^2 \times |F_{\mathscr{P}}|)$. Thus, the whole time complexity is still bounded by $O(n^3 \times |F_{\mathscr{P}}|)$. □

Corollary 1
For any program $\mathscr{P}$, the time complexity of computing $\Gamma A(\mathscr{P})$ is bounded by $O(|args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$.

Proof
Straightforward from the proof of Theorem 2. □

As shown in the previous theorem, the time complexity of checking whether a program $\mathscr{P}$ is Γ-acyclic is bounded by $O(|args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$, which is strictly related to the complexity of checking whether a program is argument-restricted, which is $O(|args(\mathscr{P})|^3)$. In fact, the new proposed criterion performs a more accurate analysis on how terms are propagated from the body to the head of rules by taking into account the function symbols occurring in such terms. Moreover, if a logic program $\mathscr{P}$ has only one function symbol, the time complexity of checking whether $\mathscr{P}$ is Γ-acyclic is the same as the one required to check if it is argument-restricted.

5 Safe programs

The Γ-acyclicity termination criterion presents some limitations, since it is not able to detect when a rule can be activated only a finite number of times during the bottom-up evaluation of the program. The next example shows a simple terminating program which is not recognized by the Γ-acyclicity termination criterion.

Example 8
Consider the following logic program P_8:

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

where b is base predicate. As the program is standard, it has a (finite) unique minimal model, which can be derived using the classical bottom-up fixpoint computation algorithm. Moreover, independently from the set of base facts defining b, the minimum model of P_8 is finite and its computation terminates. □

Observe that the rules of program P_8 can be activated at most n times, where n is the cardinality of the active domain of the base predicate b. Indeed, the recursive rule r_2 cannot activate itself since the newly generated atom is of the form $\mathtt{p(f(\cdot),g(\cdot))}$ and does not unify with its body.

As another example consider the recursive rule $\mathtt{q(f(X))} \leftarrow \mathtt{q(X),t(X)}$ and the strongly linear rule $\mathtt{p(f(X),g(Y))} \leftarrow \mathtt{p(X,Y),t(X)}$ where t[1] is a limited argument. The activation of these rules is limited by the cardinality of the active domain of t[1].

Thus, in this section, in order to define a more general termination criterion we introduce the *safety* function which, by detecting rules that can be executed only a finite number of times, derives a larger set of limited arguments of the program. We start by analyzing how rules may activate each other.

Definition 5 (*Activation graph*)
Let $\mathscr{P}$ be a program and let r_1 and r_2 be (not necessarily distinct) rules of $\mathscr{P}$. We say that r_1 *activates* r_2 iff head(r_1) and an atom in body(r_2) unify. The *activation graph* $\Sigma(\mathscr{P}) = (\mathscr{P}, E)$ consists of the set of nodes denoting the rules of $\mathscr{P}$ and the set of edges (r_i, r_j), with $r_i, r_j \in \mathscr{P}$, such that r_i activates r_j. □

Example 9
Consider program P_8 of Example 8. The activation graph of this program contains two nodes r_1 and r_2 and an edge from r_1 to r_2. Rule r_1 activates rule r_2 as the head atom $\mathtt{p(X,X)}$ of r_1 unifies with the body atom $\mathtt{p(X,X)}$ of r_2. Intuitively, this means that the execution of the first rule may cause the second rule to be activated. In fact, the execution of r_1 starting from the database instance $D = \{\mathtt{b(a)}\}$ produces the new atom $\mathtt{p(a,a)}$. The presence of this atom allows the second rule to be activated, since the body of r_2 can be made true by means of the atom $\mathtt{p(a,a)}$, producing the new atom $\mathtt{p(f(a),g(a))}$. It is worth noting that the second rule cannot activate itself since head(r_2) does not unify with the atom $\mathtt{p(X,X)}$ in body(r_2). □

The activation graph shows how rules may activate each other, and, consequently, the possibility to propagate values from one rule to another. Clearly, the active domain of an argument $p[i]$ can be infinite only if p is the head predicate of a rule that may be activated an infinite number of times. A rule may be activated an infinite number of times only if it depends on a cycle of the activation graph. Therefore, a rule not depending on a cycle can only propagate a finite number of values into its head arguments. Another important aspect is the structure of rules and the presence of limited arguments in their body and head atoms. As discussed at the beginning

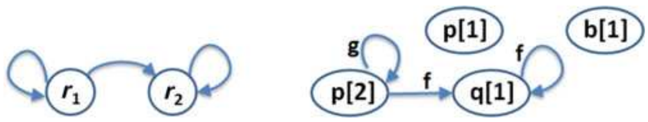

Fig. 5. (Colour online) Activation (left) and propagation (right) graphs of program P_{10}.

Fig. 5-Colour online, B/W in print

of this section, rules $\mathtt{q(f(X))} \leftarrow \mathtt{q(X), t(X)}$ and $\mathtt{p(f(X), g(Y))} \leftarrow \mathtt{p(X, Y), t(X)}$, where $\mathtt{t[1]}$ is a limited argument, can be activated only a finite number of times. In fact, as variable X in both rules can be substituted only by values taken from the active domain of t[1], which is finite, the active domains of q[1] and p[1] are finite as well, i.e. q[1] and p[1] are limited arguments. Since q[1] is limited, the first rule can be applied only a finite number of times. In the second rule we have predicate p of arity two in the head, and we know that p[1] is a limited argument. Since the second rule is strongly linear, the domains of both head arguments p[1] and p[2] grow together each time this rule is applied. Consequently, the active domain of p[2] must be finite as well as the active domain of p[1] and this rule can be applied only a finite number of times.

We now introduce the notion of *limited term*, that will be used to define a function, called *safety function*, that takes as input a set of limited arguments and derives a new set of limited arguments in $\mathscr{P}$.

Definition 6 (*Limited terms*)
Given a rule $r = q(t_1, \ldots, t_m) \leftarrow \text{body}(r) \in \mathscr{P}$ and a set A of limited arguments, we say that t_i is *limited* in r (or r limits t_i) w.r.t. A if one of the following conditions holds:

1. every variable X appearing in t_i also appears in an argument in body(r) belonging to A, or
2. r is a strongly linear rule such that:
   (a) for every atom $p(u_1, ..., u_n) \in \text{head}(r) \cup r\text{body}(r)$, all terms $u_1, ..., u_n$ are either simple or complex;
   (b) $\text{var}(\text{head}(r)) = \text{var}(r\text{body}(r))$,
   (c) there is an argument $q[j] \in A$. □

Definition 7 (*Safety function*)
For any program $\mathscr{P}$, let A be a set of limited arguments of $\mathscr{P}$ and let $\Sigma(\mathscr{P})$ be the activation graph of $\mathscr{P}$. The *safety function* $\Psi(A)$ denotes the set of arguments $q[i] \in \mathit{args}(\mathscr{P})$ such that for all rules $r = q(t_1, \ldots, t_m) \leftarrow \text{body}(r) \in \mathscr{P}$, either r does not depend on a cycle π of $\Sigma(\mathscr{P})$ or t_i is limited in r w.r.t. A. □

Example 10
Consider the following program P_{10}:

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

where b is base predicate. Let $A = \Gamma A(\mathscr{P}) = \{\mathtt{b[1]}, \mathtt{p[1]}\}$. The activation and the propagation graphs of this program are reported in Figure 5. The application of the safety function to the set of limited arguments A gives $\Psi(A) = \{\mathtt{b[1]}, \mathtt{p[1]}, \mathtt{p[2]}\}$. Indeed:

- $\mathtt{b[1]} \in \Psi(A)$ since `b` is a base predicate which does not appear in the head of any rule; consequently all the rules with `b` in the head (i.e. the empty set) trivially satisfy the conditions of Definition 7.
- $\mathtt{p[1]} \in \Psi(A)$ because the unique rule with `p` in the head (i.e. r_1) satisfies the first condition of Definition 6, that is, r_1 limits the term `f(X)` w.r.t. A in the head of rule r_1 corresponding to argument `p[1]`.
- Since r_1 is strongly linear and the second condition of Definition 6 is satisfied, $\mathtt{p[2]} \in \Psi(A)$ as well. □

The following proposition shows that the safety function can be used to derive further limited arguments.

Proposition 3
Let $\mathscr{P}$ be a program and let A be a set of limited arguments of $\mathscr{P}$. Then, all arguments in $\Psi(A)$ are also limited.

Proof
Consider an argument $q[i] \in \Psi(A)$, then for every rule $r = q(t_1, \ldots, t_n) \leftarrow \text{body}(r)$ either r does not depend on a cycle of $\Sigma(\mathscr{P})$ or t_i is limited in r w.r.t. A.

Clearly, if r does not depend on a cycle of $\Sigma(\mathscr{P})$, it can be activated a finite number of times as it is not 'effectively recursive' and does not depend on rules which are effectively recursive.

Moreover, if t_i is limited in r w.r.t. A, we have that either:

(1) The first condition of Definition 6 is satisfied (i.e. every variable X appearing in t_i also appears in an argument in body(r) belonging to A). This means that variables in t_i can be replaced by a finite number of values.
(2) The second condition of Definition 6 is satisfied. Let $p(t_1, ..., t_n) = \text{head}(r)$, the condition that all terms $t_1, ..., t_n$ must be simple or complex guarantees that, if terms in head(r) grow, then they grow all together (Conditions 2.a and 2.b). Moreover, if the growth of a term t_j is blocked (Condition 2.c), the growth of all terms (including t_i) is blocked too.
Therefore, if one of the two conditions is satisfied for all rules defining q, the active domain of $q[i]$ is finite. □

Unfortunately, as shown in the following example, the relationship $A \subseteq \Psi(A)$ does not always hold for a generic set of arguments A, even if the arguments in A are limited.

Example 11
Consider the following program P_{11}:

$$\begin{aligned}
r_1 &: \mathtt{p(f(X),Y)} \leftarrow \mathtt{q(X), r(Y)}.\\
r_2 &: \mathtt{q(X)} \leftarrow \mathtt{p(X,Y)}.\\
r_3 &: \mathtt{t(Y)} \leftarrow \mathtt{r(Y)}.\\
r_4 &: \mathtt{s(Y)} \leftarrow \mathtt{t(Y)}.\\
r_5 &: \mathtt{r(Y)} \leftarrow \mathtt{s(Y)}.
\end{aligned}$$

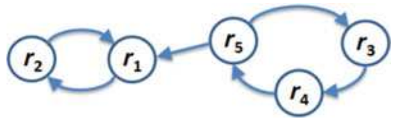

Fig. 6. (Colour online) Activation graph of program P_{11}.

Fig. 6-Colour online, B/W in print

Its activation graph $\Sigma(P_{11})$ is shown in Figure 6, whereas the set of restricted arguments is $AR(P_{11}) = \Gamma A(P_{11}) = \{\mathtt{r[1]}, \mathtt{t[1]}, \mathtt{s[1]}, \mathtt{p[2]}\}$. Considering the set $A = \{\mathtt{p[2]}\}$, we have that the safety function $\Psi(\{\mathtt{p[2]}\}) = \emptyset$. Therefore, the relation $A \subseteq \Psi(A)$ does not hold for $A = \{\mathtt{p[2]}\}$.

Moreover, regarding the set $A' = \Gamma A(P_{11}) = \{\mathtt{r[1]}, \mathtt{t[1]}, \mathtt{s[1]}, \mathtt{p[2]}\}$, we have $\Psi(A') = \{\mathtt{r[1]}, \mathtt{t[1]}, \mathtt{s[1]}, \mathtt{p[2]}\} = A'$, i.e. the relation $A' \subseteq \Psi(A')$ holds. □

The following proposition states that if we consider the set A of Γ-acyclic arguments of a given program $\mathscr{P}$, the relation $A \subseteq \Psi(A)$ holds.

Proposition 4

For any logic program $\mathscr{P}$:

1. $\Gamma A(\mathscr{P}) \subseteq \Psi(\Gamma A(\mathscr{P}))$;
2. $\Psi^i(\Gamma A(\mathscr{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathscr{P}))$ for $i > 0$.

Proof

(1) Suppose that $q[k] \in \Gamma A(\mathscr{P})$. Then $q[k] \in AR(\mathscr{P})$ or $q[k]$ does not depend on a cycle in $\Delta(\mathscr{P})$ spelling a string of $\mathscr{L}(\Gamma_{\mathscr{P}})$. In both cases $q[k]$ can depend only on arguments in $\Gamma A(\mathscr{P})$. If $q[k]$ does not depend on any argument, then it does not appear in the head of any rule and, consequently, $q[k] \in \Psi(\Gamma A(\mathscr{P}))$. Otherwise, the first condition of Definition 6 is satisfied and $q[k] \in \Psi(\Gamma A(\mathscr{P}))$.

(2) We prove that $\Psi^i(\Gamma A(\mathscr{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathscr{P}))$ for $i > 0$ by induction. We start by showing that $\Psi^i(\Gamma A(\mathscr{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathscr{P}))$ for $i = 1$, i.e. that the relation $\Psi(\Gamma A(\mathscr{P})) \subseteq \Psi(\Psi(\Gamma A(\mathscr{P})))$ holds. In order to show this relation we must show that for every argument $q[k] \in \mathscr{P}$ if $q[k] \in \Psi(\Gamma A(\mathscr{P}))$, then $q[k] \in \Psi(\Psi(\Gamma A(\mathscr{P}))$. Consider $q[k] \in \Psi(\Gamma A(\mathscr{P}))$. Then, $q[k]$ satisfies Definition 7 w.r.t. $A = \Gamma A(\mathscr{P})$. From comma one of this proof it follows that $\Gamma A(\mathscr{P}) \subseteq \Psi(\Gamma A(\mathscr{P}))$, consequently $q[k]$ satisfies Definition 7 w.r.t. $A = \Psi(\Gamma A(\mathscr{P}))$ too and so, $q[k] \in \Psi(\Psi(\Gamma A(\mathscr{P})))$. Suppose that $\Psi^k(\Gamma A(\mathscr{P})) \subseteq \Psi^{k+1}(\Gamma A(\mathscr{P}))$ for $k > 0$. In order to show that $\Psi^{k+1}(\Gamma A(\mathscr{P})) \subseteq \Psi^{k+2}(\Gamma A(\mathscr{P}))$ we must show that for every argument $q[k] \in \mathscr{P}$ if $q[k] \in \Psi^{k+1}(\Gamma A(\mathscr{P}))$, then $q[k] \in \Psi^{k+2}(\Gamma A(\mathscr{P}))$. Consider $q[k] \in \Psi^{k+1}(\Gamma A(\mathscr{P}))$. Then $q[k]$ satisfies Definition 7 w.r.t. $A = \Psi^k(\Gamma A(\mathscr{P}))$. Since $\Psi^k(\Gamma A(\mathscr{P})) \subseteq \Psi^{k+1}(\Gamma A(\mathscr{P}))$, $q[k]$ satisfies Definition 7 w.r.t. $A = \Psi^{k+1}(\Gamma A(\mathscr{P}))$ too. Consequently, $q[k] \in \Psi^{k+2}(\Gamma A(\mathscr{P}))$.

□

Observe that we can prove in a similar way that $AR(\mathscr{P}) \subseteq \Psi(AR(\mathscr{P}))$ and that $\Psi^i(AR(\mathscr{P})) \subseteq \Psi^{i+1}(AR(\mathscr{P}))$ for $i > 0$.

Definition 8 (*Safe arguments and safe programs*)
For any program $\mathscr{P}$, safe$(\mathscr{P}) = \Psi^{\infty}(\Gamma A(\mathscr{P}))$ denotes the set of *safe arguments* of $\mathscr{P}$. A program $\mathscr{P}$ is said to be *safe* if all arguments are safe. The class of safe programs will be denoted by $\mathscr{SP}$. □

Clearly, for any set of arguments $A \subseteq \Gamma A(\mathscr{P})$, $\Psi^i(A) \subseteq \Psi^i(\Gamma A(\mathscr{P}))$. Moreover, as shown in Proposition 4, when the starting set is $\Gamma A(\mathscr{P})$, the sequence $\Gamma A(\mathscr{P}), \Psi(\Gamma A(\mathscr{P})), \Psi^2(\Gamma A(\mathscr{P})), \ldots$ is monotone and there is a finite $n = O(|args(\mathscr{P})|)$ such that $\Psi^n(\Gamma A(\mathscr{P})) = \Psi^{\infty}(\Gamma A(\mathscr{P}))$. We can also define the inflactionary version of Ψ as $\hat{\Psi}(A) = A \cup \Psi(A)$, obtaining that $\hat{\Psi}^i(\Gamma A(\mathscr{P})) = \Psi^i(\Gamma A(\mathscr{P}))$, for all natural numbers i. The introduction of the inflactionary version guarantees that the sequence $A, \hat{\Psi}(A), \hat{\Psi}^2(A), \ldots$ is monotone for every set A of limited arguments. This would allow us to derive a (possibly) larger set of limited arguments starting from any set of limited arguments.

Example 12
Consider again program P_8 of Example 8. Although $AR(P_8) = \emptyset$, the program P_8 is safe as $\Sigma(P_8)$ is acyclic.

Consider now the program P_{10} of Example 10. As already shown in Example 10, the first application of the safety function to the set of Γ-acyclic arguments of P_{10} gives $\Psi(\Gamma A(P_{10})) = \{\mathtt{b[1]}, \mathtt{p[1]}, \mathtt{p[2]}\}$. The application of the safety function to the obtained set gives $\Psi(\Psi(\Gamma A(P_{10}))) = \{\mathtt{b[1]}, \mathtt{p[1]}, \mathtt{p[2]}, \mathtt{q[1]}\}$. In fact, in the unique rule defining q, term f(Y), corresponding to the argument q[1], is limited in r w.r.t. $\{\mathtt{b[1]}, \mathtt{p[1]}, \mathtt{p[2]}\}$ (i.e. the variable Y appears in body(r) in a term corresponding to argument p[2] and argument p[2], belonging to the input set, is limited). At this point, all arguments of P_{10} belong to the resulting set. Thus, safe$(P_{10}) = args(P_{10})$, and we have that program P_{10} is safe. □

We now show results on the expressivity of the class $\mathscr{SP}$ of safe programs.

Theorem 3
The class $\mathscr{SP}$ of safe programs strictly includes the class $\Gamma\mathscr{A}$ of Γ-acyclic programs and is strictly contained in the class $\mathscr{FG}$ of finitely ground programs.

Proof
($\Gamma\mathscr{A} \subsetneq \mathscr{SP}$). From Proposition 4 it follows that $\Gamma\mathscr{A} \subseteq \mathscr{SP}$. Moreover, $\Gamma\mathscr{A} \subsetneq \mathscr{SP}$ as program P_{10} is safe but not Γ-acyclic.
($\mathscr{SP} \subsetneq \mathscr{FG}$). From Proposition 3 it follows that every argument in the safe program can take values only from a finite domain. Consequently, the set of all possible ground terms derived during the grounding process is finite and the program is finitely ground. Moreover, we have that the program P_{16} of Example 16 is finitely ground, but not safe. □

As a consequence of Theorem 3, every safe program admits a finite minimum model.

Complexity. We start by introducing a bound on the complexity of constructing the activation graph.

Proposition 5

For any program $\mathscr{P}$, the activation graph $\Sigma(\mathscr{P})$ can be constructed in time $O(n_r^2 \times n_b \times (a_p \times a_f)^2)$, where n_r is the number of rules of $\mathscr{P}$, n_b is the maximum number of body atoms in a rule, a_p is the maximum arity of predicate symbols and a_f is the maximum arity of function symbols.

Proof

To check whether a rule r_i activates a rule r_j we have to determine if an atom B in body(r_j) unifies with the head-atom A of r_i. This can be done in time $O(n_b \times u)$, where u is the cost of deciding whether two atoms unify, which is quadratic in the size of the two atoms (Venturini Zilli 1975), that is $u = O((a_p \times a_f)^2)$ as the size of atoms is bounded by $a_p \times a_f$ (recall that the maximum depth of terms is 1). In order to construct the activation graph we have to consider all pairs of rules and for each pair we have to check if the first rule activates the second one. Therefore, the global complexity is $O(n_r^2 \times n_b \times u) = O(n_r^2 \times n_b \times (a_p \times a_f)^2)$. □

We recall that given two atoms A and B, the size of a mgu θ for $\{A, B\}$ can be, in the worst case, exponential in the size of A and B, but the complexity of deciding whether a unifier for A and B exists is quadratic in the size of A and B (Venturini Zilli 1975).

Proposition 6

The complexity of deciding whether a program $\mathscr{P}$ is safe is $O((\text{size}(\mathscr{P}))^2 + |args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$.

Proof

The construction of the activation graph $\Sigma(\mathscr{P})$ can be done in time $O(n_r^2 \times n_b \times (a_p \times a_f)^2)$, where n_r is the number of rules of $\mathscr{P}$, n_b is the maximum number of body atoms in a rule, a_p is the maximum arity of predicate symbols, and a_f is the maximum arity of function symbols (*cf.* Proposition 5).

The complexity of computing $\Gamma A(\mathscr{P})$ is bounded by $O(|args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$ (*cf.* Theorem 2).

From Definition 7 and Proposition 4 it follows that the sequence $\Gamma A(\mathscr{P})$, $\Psi(\Gamma A(\mathscr{P}))$, $\Psi^2(\Gamma A(\mathscr{P}))$, ... is monotone and converges in a finite number of steps bounded by the cardinality of the set $args(\mathscr{P})$. The complexity of determining rules not depending on cycles in the activation graph $\Sigma(\mathscr{P})$ is bounded by $O(n_r^2)$, as it can be done by means of a depth-first traversal of $\Sigma(\mathscr{P})$, which is linear in the number of its edges. Since checking whether the conditions of Definition 6 hold for all arguments in $\mathscr{P}$ is in $O(\text{size}(\mathscr{P}))$, checking such conditions for at most $|args(\mathscr{P})|$ steps is $O(|args(\mathscr{P})| \times \text{size}(\mathscr{P}))$. Thus, the complexity of checking all the conditions of Definition 7 for all steps is $O(n_r^2 + |args(\mathscr{P})| \times \text{size}(\mathscr{P}))$.

Since, $n_r^2 \times n_b \times (a_p \times a_f)^2 = O((\text{size}(\mathscr{P}))^2)$, $|args(\mathscr{P})| = O(\text{size}(\mathscr{P}))$ and $n_r^2 = O((\text{size}(\mathscr{P}))^2)$, the complexity of deciding whether $\mathscr{P}$ is safe is $O((\text{size}(\mathscr{P}))^2 + |args(\mathscr{P})|^3 \times |F_{\mathscr{P}}|)$. □

6 Bound queries and examples

In this section we consider the extension of our framework to queries. This is an important aspect as in many cases, the answer to a query is finite, although the models may have infinite cardinality. This happens very often when the query goal contains ground terms.

6.1 Bound queries

Rewriting techniques, such as magic-set, allow bottom-up evaluators to efficiently compute (partially) ground queries, that is queries whose query goal contains ground terms. These techniques rewrite queries (consisting of a query goal and a program) such that the top-down evaluation is emulated (Beeri and Ramakrishnan 1991; Greco 2003; Greco *et al.* 2005; Alviano *et al.* 2010). Labelling techniques similar to magic-set have been also studied in the context of term rewriting (Zantema 1995). Before presenting the rewriting technique, let us introduce some notations.

A *query* is a pair $Q = \langle q(u_1,..,u_n),\mathscr{P}\rangle$, where $q(u_1,..,u_n)$ is an atom called *query goal* and $\mathscr{P}$ is a program. We recall that an *adornment* of a predicate symbol p with arity n is a string $\alpha \in \{b,f\}^*$ such that $|\alpha| = n$[1]. The symbols b and f denote, respectively, *bound* and *free* arguments. Given a query $Q = \langle q(u_1,..,u_n),\mathscr{P}\rangle$, $\text{Magic}S(Q) = \langle q^\alpha(u_1,..,u_n), \text{Magic}S(q(u_1,..,u_n),\mathscr{P})\rangle$ indicates the rewriting of Q, where $\text{Magic}S(q(u_1,..,u_n),\mathscr{P})$ denotes the rewriting of rules in $\mathscr{P}$ w.r.t. the query goal $q(u_1,..,u_n)$ and α is the adornment associated with the query goal.

We assume that our queries $\langle G,\mathscr{P}\rangle$ are positive, as the rewriting technique is here applied to $\langle G, st(\mathscr{P})\rangle$ to generate the positive program which is used to restrict the source program (see Section 8).

Definition 9
A query $Q = \langle G,\mathscr{P}\rangle$ is *safe* if $\mathscr{P}$ or $\text{Magic}S(G,\mathscr{P})$ is safe. □

It is worth noting that it is possible to have a query $Q=\langle G,\mathscr{P}\rangle$ such that $\mathscr{P}$ is safe, but the rewritten program $\text{Magic}S(G,\mathscr{P})$ is not safe and vice versa.

Example 13
Consider the query $Q = \langle \mathtt{p(f(f(a)))}, P_{13}\rangle$, where P_{13} is defined below:

$$\begin{array}{l}\mathtt{p(a).}\\ \mathtt{p(f(X)) \leftarrow p(X).}\end{array}$$

P_{13} is not safe, but if we rewrite the program using the magic-set method, we obtain the safe program:

$$\begin{array}{l}\mathtt{magic\_p^b(f(f(a))).}\\ \mathtt{magic\_p^b(X) \leftarrow magic\_p^b(f(X)).}\\ \mathtt{p^b(a) \leftarrow magic\_p^b(a).}\\ \mathtt{p^b(f(X)) \leftarrow magic\_p^b(f(X)),\ p^b(X).}\end{array}$$

[1] Adornments of predicates, introduced to optimize the bottom-up computation of logic queries, are similar to *mode of usage* defined in logic programming to describe how the arguments of a predicate p must be restricted when an atom with predicate symbol p is called.

Consider now the query $Q = \langle \mathtt{p(a)}, \mathscr{P}'_{13} \rangle$, where $\mathscr{P}'_{13}$ is defined as follows:

$$\begin{array}{l} \mathtt{p(f(f(a)))}. \\ \mathtt{p(X)} \leftarrow \mathtt{p(f(X))}. \end{array}$$

The program is safe, but after the magic-set rewriting we obtain the following program:

$$\begin{array}{l} \mathtt{magic\_p^b(a)}. \\ \mathtt{magic\_p^b(f(X))} \leftarrow \mathtt{magic\_p^b(X)}. \\ \mathtt{p^b(f(f(a)))} \leftarrow \mathtt{magic\_p^b(f(f(a)))}. \\ \mathtt{p^b(X)} \leftarrow \mathtt{magic\_p^b(X), p^b(f(X))}. \end{array}$$

which is not recognized as safe because it is not terminating. □

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

We recall that for each predicate symbol p with arity n, the number of adorned predicates $p^{\alpha_1 \ldots \alpha_n}$ could be exponential and bounded by $O(2^n)$. However, in practical cases only few adornments are generated for each predicate symbol. Indeed, rewriting techniques are well consolidated and widely used to compute bound queries.

6.2 Examples

Let us now consider the application of the technique described above to some practical examples. Since each predicate in the rewritten query has a unique adornment, we shall omit them.

Example 14
Consider the query $\langle \mathtt{reverse([a,b,c,d],L)}, P_{14} \rangle$, where P_{14} is defined by the following rules:

$$\begin{array}{l} r_0 : \mathtt{reverse([\,],[\,])}. \\ r_1 : \mathtt{reverse([X|Y],[X|Z])} \leftarrow \mathtt{reverse(Y,Z)}. \end{array}$$

The equivalent program P'_{14}, rewritten to be computed by means of a bottom-up evaluator, is:

$$\begin{array}{l} \rho_0 : \mathtt{m\_reverse([a,b,c,d])}. \\ \rho_1 : \mathtt{m\_reverse(Y)} \leftarrow \mathtt{m\_reverse([X|Y])}. \\ \rho_2 : \mathtt{reverse([\,],[\,])} \leftarrow \mathtt{m\_reverse([\,])}. \\ \rho_3 : \mathtt{reverse([X|Y],[X|Z])} \leftarrow \mathtt{m\_reverse([X|Y]), reverse(Y,Z)}. \end{array}$$

Observe that P'_{14} is not argument-restricted. In order to check Γ-acyclicity and safety criteria, we have to rewrite rule ρ_3 having complex terms in both the head and the body. Thus we add an additional predicate b1 defined by rule ρ_4 and replace ρ_3 by ρ'_3.

$$\begin{array}{l} \rho'_3 : \mathtt{reverse([X|Y],[X|Z])} \leftarrow \mathtt{b1(X,Y,Z)}. \\ \rho_4 : \mathtt{b1(X,Y,Z)} \leftarrow \mathtt{m\_reverse([X|Y]), reverse(Y,Z)}. \end{array}$$

The obtained program, denoted P''_{14}, is safe but not Γ-acyclic. □

Example 15
Consider the query $\langle \mathtt{length([a,b,c,d],L)}, P_{15}\rangle$, where P_{15} is defined by the following rules:

$$r_0 : \mathtt{length([\,],0).}$$
$$r_1 : \mathtt{length([X|T],I+1)} \leftarrow \mathtt{length(T,I).}$$

The equivalent program P'_{15}, is rewritten to be computed by means of a bottom-up evaluator as follows[2] :

$$\rho_0 : \mathtt{m\_length([a,b,c,d]).}$$
$$\rho_1 : \mathtt{m\_length(T)} \leftarrow \mathtt{m\_length([X|T]).}$$
$$\rho_2 : \mathtt{length([\,],0)} \leftarrow \mathtt{m\_length([\,]).}$$
$$\rho_3 : \mathtt{length([X|T],I+1)} \leftarrow \mathtt{m\_length([X|T]),\ length(T,I).}$$

Also in this case, it is necessary to split rule ρ_3 into two rules to avoid having function symbols in both the head and the body, as shown below:

$$\rho'_3 : \mathtt{length([X|T],I+1)} \leftarrow \mathtt{b1(X,T,I).}$$
$$\rho_4 : \mathtt{b1(X,T,I)} \leftarrow \mathtt{m\_length1(X,T),\ length(T,I).}$$

The obtained program, denoted P''_{15}, is safe but not Γ-acyclic. □

We conclude this section pointing out that the queries in the two examples above are not recognized as terminating by most of the previously proposed techniques, including *AR*. We also observe that many programs follow the structure of programs presented in the examples above. For instance, programs whose aim is the verification of a given property on the elements of a given list, have the following structure:

$$\mathtt{verify([\,],[\,]).}$$
$$\mathtt{verify([X|L_1],[Y|L_2])} \leftarrow \mathtt{property(X,Y), verify(L_1,L_2).}$$

Consequently, queries having a ground argument in the query goal are terminating.

7 Further improvements

The safety criterion can be improved further as it is not able to detect that in the activation graph, there may be cyclic paths that are not effective or can only be activated a finite number of times. The next example shows a program which is finitely ground, but recognized as terminating by the safety criterion.

Example 16
Consider the following logic program P_{16} obtained from P_8 by adding an auxiliary predicate q:

$$r_1 : \mathtt{p(X,X)} \leftarrow \mathtt{b(X).}$$
$$r_2 : \mathtt{q(f(X),g(X))} \leftarrow \mathtt{p(X,X).}$$
$$r_3 : \mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Y).}$$

P_{16} is equivalent to P_8 w.r.t. predicate p. □

[2] Observe that program P'_{15} is equivalent to program P_1 presented in the Introduction, assuming that the base predicate `input` is defined by a fact `input([a,b,c,d])`.

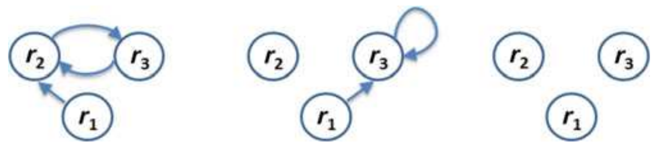

Fig. 7. (Colour online) k-restricted activation graphs: $\Sigma_1(P_{16})$ (left), $\Sigma_2(P_{16})$ (center), $\Sigma_3(P_{16})$ (right).

Fig. 7-Colour online, B/W in print

Although the activation graph $\Sigma(P_{16})$ contains a cycle, the rules occurring in the cycle cannot be activated an infinite number of times. Therefore, in this section we introduce the notion of *active paths* and extend the definitions of activation graphs and safe programs.

Definition 10 (*Active path*)
Let $\mathscr{P}$ be a program and $k \geqslant 1$ be a natural number. The path $(r_1, r_2), \ldots, (r_k, r_{k+1})$ is an *active path* in the activation graph $\Sigma(\mathscr{P})$ iff there is a set of unifiers $\theta_1, \ldots, \theta_k$, such that

- head(r_1) unifies with an atom from body(r_2) with unifier θ_1;
- head$(r_i)\theta_{i-1}$ unifies with an atom from body(r_{i+1}) with unifier θ_i for $i \in [2, ..., k]$.

We write $r_1 \overset{k}{\rightsquigarrow} r_{k+1}$ if there is an active path of length k from r_1 to r_{k+1} in $\Sigma(\mathscr{P})$. □

Intuitively, $(r_1, r_2), \ldots, (r_k, r_{k+1})$ is an active path if r_1 transitively activates rule r_{k+1}, that is if the head of r_1 unifies with some body atom of r_2 with mgu θ_1, then the head of the rule $r_2\theta_1$ unifies with some body atom of r_3 with mgu θ_2, then the head of the rule $r_3\theta_2$ unifies with some body atom of r_4 with mgu θ_3, and so on until the head of the rule $r_k\theta_{k-1}$ unifies with some body atom of r_{k+1} with mgu θ_k.

Definition 11 (*k-restricted activation graph*)
Let $\mathscr{P}$ be a program and $k \geqslant 1$ be a natural number, the *k-restricted activation graph* $\Sigma_k(\mathscr{P}) = (\mathscr{P}, E)$ consists of a set of nodes denoting the rules of $\mathscr{P}$ and a set of edges E defined as follows: there is an edge (r_i, r_j) from r_i to r_j iff $r_i \overset{k}{\rightsquigarrow} r_j$, i.e. iff there is an active path of length k from r_i to r_j. □

Example 17
The k-restricted activation graphs for the program of Example 16, with $k \in [1, ..., 3]$, are reported in Figure 7. □

Obviously, the activation graph presented in Definition 5 is 1-restricted. We next extend the definition of safe function by referring to k-restricted activation graphs, instead of the (1-restricted) activation graph.

Definition 12 (*k-safety function*)
For any program $\mathscr{P}$ and natural number $k \geqslant 1$, let A be a set of limited arguments of $\mathscr{P}$. The *k-safety function* $\Psi_k(A)$ denotes the set of arguments $q[i] \in args(\mathscr{P})$ such that for all rules $r = q(t_1, \ldots, t_m) \leftarrow$ body$(r) \in \mathscr{P}$, either r does not depend on a cycle π of $\Sigma_j(\mathscr{P})$, for some $1 \leqslant j \leqslant k$, or t_i is limited in r w.r.t. A. □

Observe that the *k-safety function* Ψ_k is defined as a natural extension of the safety function Ψ by considering all the j-restricted activation graphs, for $1 \leqslant j \leqslant k$. Note that the 1-restricted activation graph coincides with the standard activation graph and, consequently, Ψ_1 coincides with Ψ.

Definition 13 (*k-safe arguments*)
For any program $\mathcal{P}$, $\mathrm{safe}_k(\mathcal{P}) = \Psi_k^{\infty}(\Gamma A(\mathcal{P}))$ denotes the set of *k-safe* arguments of $\mathcal{P}$. A program $\mathcal{P}$ is said to be *k-safe* if all arguments are k-safe. □

Example 18
Consider again the logic program P_{16} from Example 16. $\Sigma_2(P_{16})$ contains the unique cycle (r_3, r_3); consequently, $\mathtt{q[1]}$ and $\mathtt{q[2]}$ appearing only in the head of rule r_2 are 2-safe. By applying iteratively operator Ψ_2 to the set of limited arguments $\{\mathtt{b[1]}, \mathtt{q[1]}, \mathtt{q[2]}\}$, we derive that also $\mathtt{p[1]}$ and $\mathtt{p[2]}$ are 2-safe. Since $\mathrm{safe}_2(P_{16}) = args(P_{16})$, we have that P_{16} is 2-safe. Observe also that $\Sigma_3(P_{16})$ does not contain any edge and, therefore, all arguments are 3-safe. □

For any natural number $k > 0$, $\mathcal{SP}_k$ denotes the class of k-safe logic programs, that is the set of programs $\mathcal{P}$ such that $\mathrm{safe}_k(\mathcal{P}) = args(\mathcal{P})$. The following proposition states that the classes of k-safe programs define a hierarchy where $\mathcal{SP}_k \subsetneq \mathcal{SP}_{k+1}$.

Proposition 7
The class $\mathcal{SP}_{k+1}$ of $(k+1)$-safe programs strictly extends the class $\mathcal{SP}_k$ of k-safe programs, for any $k \geqslant 1$.

Proof
($\mathcal{SP}_k \subseteq \mathcal{SP}_{k+1}$) It follows straightforwardly from the definition of k-safe function.
($\mathcal{SP}_k \neq \mathcal{SP}_{k+1}$) To show that the containment is strict, consider the program P_{16} from Example 16 for $k = 1$ and the following program $\mathcal{P}_k$ for $k > 1$:

$$\begin{aligned}
r_0:&\quad \mathtt{q_1(f(X), g(X)) \leftarrow p(X, X).}\\
r_1:&\quad \mathtt{q_2(X, Y) \leftarrow q_1(X, Y).}\\
\dots&\\
r_{k-1}:&\quad \mathtt{q_k(X, Y) \leftarrow q_{k-1}(X, Y).}\\
r_k:&\quad \mathtt{p(X, Y) \leftarrow q_k(X, Y).}
\end{aligned}$$

It is easy to see that $\mathcal{P}_k$ is in $\mathcal{SP}_{k+1}$, but not in $\mathcal{SP}_k$. □

Recall that the minimal model of a standard program $\mathcal{P}$ can be characterized in terms of the classical immediate consequence operator $\mathcal{T}_{\mathcal{P}}$ defined as follows. Given a set I of ground atoms, then

$$\mathcal{T}_{\mathcal{P}}(I) = \{A\theta \mid \exists r : A \leftarrow A_1, \dots, A_n \in \mathcal{P} \text{ and } \exists \theta \text{ s.t. } A_i\theta \in I \text{ for every } 1 \leqslant i \leqslant n\}$$

where θ is a substitution replacing variables with constants. Thus, $\mathcal{T}_{\mathcal{P}}$ takes as input a set of ground atoms and returns as output a set of ground atoms; clearly, $\mathcal{T}_{\mathcal{P}}$ is monotonic. The ith iteration of $\mathcal{T}_{\mathcal{P}}$ ($i \geqslant 1$) is defined as follows: $\mathcal{T}_{\mathcal{P}}^1(I) = \mathcal{T}_{\mathcal{P}}(I)$ and $\mathcal{T}_{\mathcal{P}}^i(I) = \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}^{i-1}(I))$ for $i > 1$. It is well known that the minimum model of $\mathcal{P}$ is equal to the fixed point $\mathcal{T}_{\mathcal{P}}^{\infty}(\emptyset)$.
A rule r is *fired at run-time* with a substitution θ at step i if $\mathrm{head}(r)\theta \in T_{\mathcal{P}}^i(\emptyset) - T_{\mathcal{P}}^{i-1}(\emptyset)$. Moreover, we say that r *is fired (at run-time) by a rule s* if r is fired with a

substitution θ at step i, s is fired with a substitution σ at step $i-1$, and head$(s)\sigma \in$ body$(r)\theta$. Let $\mathscr{P}$ be a program whose minimum model is $M = \mathscr{M}\mathscr{M}(\mathscr{P}) = T^{\infty}_{\mathscr{P}}(\emptyset)$, $M[\![r]\!]$ denotes the set of facts which have been inferred during the application of the immediate consequence operator using rule r, that is the set of facts head$(r)\theta$ such that, for some natural number i, head$(r)\theta \in T^{i}_{\mathscr{P}}(\emptyset) - T^{i-1}_{\mathscr{P}}(\emptyset)$; $M[\![r]\!]$ if infinite iff r is fired an infinite number of times. Clearly, if a rule s fires at run-time a rule r, then the activation graph contains an edge (s, r). An *active sequence* of rules is a sequence of rules $r_1, \ldots, r_n$ such that r_i fires at run-time rule r_{i+1} for $i \in [1, ..., n-1]$.

Theorem 4

Let $\mathscr{P}$ be a logic program and let r be a rule of $\mathscr{P}$. If $M[\![r]\!]$ is infinite, then, for every natural number k, r depends on a cycle of $\Sigma_k(\mathscr{P})$.

Proof

Let n_r be the number of rules of $\mathscr{P}$ and let $N = n_r \times k$. If $M[\![r]\!]$ is infinite we have that there is an active sequence of rules $r'_0, r'_1, \ldots, r'_i, \ldots, r'_N$ such that r'_N coincides with r. This means that

$$r'_0 \overset{k}{\rightsquigarrow} r'_k,\ r'_k \overset{k}{\rightsquigarrow} r'_{2k},\ \ldots,\ r'_{j\times k} \overset{k}{\rightsquigarrow} r'_{(j+1)\times k},\ \ldots,\ r'_{(n_r-1)\times k} \overset{k}{\rightsquigarrow} r'_N,$$

i.e. that the k-restricted activation graph $\Sigma_k(\mathscr{P})$ contains path $\pi = (r'_0, r'_k)$, $(r'_k, r'_{2k}), \ldots, (r'_{j\times k}, r'_{(j+1)\times k}), \ldots, (r'_{(n_r-1)\times k}, r)$. Observe that the number of rules involved in π is $n_r + 1$ and is greater than the number of rules of $\mathscr{P}$. Consequently, there is a rule occurring more than once in π, i.e. π contains a cycle. Therefore, r depends on a cycle of $\Sigma_k(\mathscr{P})$. □

As shown in Example 18, in some cases the analysis of the k-restricted activation graph is enough to determine the termination of a program. Indeed, let cyclic$R(\Sigma_k(\mathscr{P}))$ be the set of rules r in $\mathscr{P}$ s.t. r depends on a cycle in $\Sigma_k(\mathscr{P})$, the following results hold.

Corollary 2

A program $\mathscr{P}$ is terminating if $\forall r \in \mathscr{P}$, $\exists k$ s.t. $r \notin$ cyclic$R(\Sigma_k(\mathscr{P}))$.

Proof

Straightforward from Theorem 4. □

Obviously, if there is a k such that for all rules $r \in \mathscr{P}$ $r \notin$ cyclic$R(\Sigma_k(\mathscr{P}))$, $\mathscr{P}$ is terminating. We conclude this section showing that the improvements here discussed increase the complexity of the technique which is not polynomial anymore.

Proposition 8

For any program $\mathscr{P}$ and natural number $k > 1$, the activation graph $\Sigma_k(\mathscr{P})$ can be constructed in time exponential in the size of $\mathscr{P}$ and k.

Proof

Let $(r_1, r_2) \cdots (r_k, r_{k+1})$ be an active path of length k in $\Sigma(\mathscr{P})$. Consider a pair (r_i, r_{i+1}) and two unifying atoms $A_i =$ head(r_i) and $B_{i+1} \in$ body(r_{i+1}) (with $1 \leqslant i \leqslant k$), the size of an mgu θ for A_i and B_{i+1}, represented in the standard way (*cf.* Section 2), can be exponential in the size of the two atoms. Clearly, the size of $A_i\theta$ and $B_{i+1}\theta$

can also be exponential. Consequently, the size of $A_{i+1}\theta$ which is used for the next step, can grow exponentially as well. Moreover, since in the computation of an active path of length k we apply k mgu's, the size of terms can grow exponentially with k. □

Observe that for the computation of the 1-restricted argument graph it is sufficient to determine if two atoms unify (without computing the mgu), whereas for the computation of the k-restricted argument graphs, with $k > 1$, it is necessary to construct all the mgu's and to apply them to atoms.

8 Computing stable models for disjunctive programs

In this section we discuss how termination criteria, defined for standard programs, can be applied to general disjunctive logic programs. First, observe that we have assumed that whenever the same variable X appears in two terms occurring, respectively, in the head and body of a rule, at most one of the two terms is a complex term 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 r' of the form

$$\mathtt{p(f(g(X)),h(Y,Z))} \leftarrow \mathtt{p(f(X),Y)},\ \mathtt{q(h(g(X),l(Z)))}$$

is rewritten into the set of 'flatten' rules below:

$$\begin{array}{lcl}
\mathtt{p(f(A),h(Y,Z))} & \leftarrow & \mathtt{b_1(A,Y,Z)} \\
\mathtt{b_1(g(X),Y,Z)} & \leftarrow & \mathtt{b_2(X,Y,Z)} \\
\mathtt{b_2(X,Y,Z)} & \leftarrow & \mathtt{b_3(X,Y,g(X),l(Z))} \\
\mathtt{b_3(X,Y,B,C)} & \leftarrow & \mathtt{p(f(X),Y)},\ \mathtt{q(h(B,C))}
\end{array}$$

where $\mathtt{b_1}, \mathtt{b_2}$ and $\mathtt{b_3}$ are new predicate symbols, whereas $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$ are new variables introduced to flat terms with depth greater than 1.

More specifically, let $d(p(t_1,\dots,t_n)) = \max\{d(t_1),\dots,d(t_n)\}$ be the depth of atom $p(t_1,\dots,t_n)$ and $d(A_1,\dots,A_n) = \max\{d(A_1),\dots,d(A_n)\}$ be the depth of a conjunction of atoms $A_1,\dots,A_n$, for each standard rule r we generate a set of 'flatten' rules, denoted by $\mathrm{flat}(r)$ whose cardinality is bounded by $O(d(\mathrm{head}(r)) + d(\mathrm{body}(r))$. Therefore, given a standard program $\mathscr{P}$, the number of rules of the rewritten program is polynomial in the size of $\mathscr{P}$ and bounded by

$$O\left(\sum_{r\in\mathscr{P}} d(\mathrm{head}(r)) + d(\mathrm{body}(r))\right).$$

Concerning the number of arguments in the rewritten program, for a given rule r we denote with $nl(r,h,i)$ (resp. $nl(r,b,i)$) the number of occurrences of function symbols occurring at the same nesting level i in the head (resp. body) of r and with $nf(r) = \max\{nl(r,t,i) \mid t \in \{h,b\} \wedge i > 1\}$. For instance, considering the above rule r', we have that $nl(r',h,1) = 2$ (function symbols $\mathtt{f}$ and $\mathtt{h}$ occur at nesting level 1 in the head), $nl(r',h,2) = 1$ (function symbol $\mathtt{g}$ occurs at nesting level 2 in the head), $nl(r',b,1) = 2$ (function symbols $\mathtt{f}$ and $\mathtt{h}$ occur at nesting level 1 in the head), $nl(r', b, 2) = 2$ (function symbols g and l occur at nesting level 2 in the head). Consequently, $nf(r') = 2$.

The rewriting of the source program results in a 'flattened' program with $|\text{flat}(r)| - 1$ new predicate symbols. The arity of every new predicate in flat(r) is bounded by $|\text{var}(r)| + nf(r)$. Therefore, the global number of arguments in the flattened program is bounded by

$$O\left(args(\mathscr{P}) + \sum_{r \in \mathscr{P}} \left(\; |\text{var}(r)| + nf(r) \; \right) \right).$$

The termination of a disjunctive program $\mathscr{P}$ with negative literals can be determined by rewriting it into a standard logic program $st(\mathscr{P})$ such that every stable model of $\mathscr{P}$ is contained in the (unique) minimum model of $st(\mathscr{P})$, and then by checking $st(\mathscr{P})$ for termination.

Definition 14 (*Standard version*)
Given a program $\mathscr{P}$, $st(\mathscr{P})$ denotes the standard program, called *standard version*, obtained by replacing every disjunctive rule $r = a_1 \vee \cdots \vee a_m \leftarrow \text{body}(r)$ with m standard rules of the form $a_i \leftarrow \text{body}^+(r)$, for $1 \leqslant i \leqslant m$.

Moreover, we denote with $ST(\mathscr{P})$ the program derived from $st(\mathscr{P})$ by replacing every derived predicate symbol q with a new derived predicate symbol Q. □

The number of rules in the standard program $st(\mathscr{P})$ is equal to $\sum_{r \in \mathscr{P}} |\text{head}(r)|$, where $|\text{head}(r)|$ denotes the number of atoms in the head of r.

Example 19
Consider program P_{19} consisting of the two rules

$$\begin{array}{l} \mathtt{p(X)} \vee \mathtt{q(X)} \leftarrow \mathtt{r(X)}, \neg \mathtt{a(X)}. \\ \mathtt{r(X)} \leftarrow \mathtt{b(X)}, \neg \mathtt{q(X)}. \end{array}$$

where p, q and r are derived (mutually recursive) predicates, whereas a and b are base predicates. The derived standard program $st(P_{19})$ is as follows:

$$\begin{array}{l} \mathtt{p(X)} \leftarrow \mathtt{r(X)}. \\ \mathtt{q(X)} \leftarrow \mathtt{r(X)}. \\ \mathtt{r(X)} \leftarrow \mathtt{b(X)}. \end{array}$$

□

Lemma 2
For every program $\mathscr{P}$, every stable model $M \in \mathscr{SM}(\mathscr{P})$ is contained in the minimum model $\mathscr{MM}(st(\mathscr{P}))$.

Proof
From the definition of stable models we have that every $M \in \mathscr{SM}(\mathscr{P})$ is the minimal model of the ground positive program $\mathscr{P}^M$. Consider now the standard program $\mathscr{P}'$ derived from $\mathscr{P}^M$ by replacing every ground disjunctive rule $r = a_1 \vee \cdots \vee a_n \leftarrow \text{body}(r)$ with m ground normal rules $a_i \leftarrow \text{body}(r)$. Clearly, $M \subseteq \mathscr{MM}(\mathscr{P}')$. Moreover, since $\mathscr{P}' \subseteq st(\mathscr{P})$, we have that $\mathscr{MM}(\mathscr{P}') \subseteq \mathscr{MM}(st(\mathscr{P}))$. Therefore, $M \subseteq \mathscr{MM}(st(\mathscr{P}))$. □

The above lemma implies that for any logic program $\mathscr{P}$, if $st(\mathscr{P})$ is finitely ground we can restrict the Herbrand base and only consider head (ground) atoms $q(t)$ such that $q(t) \in \mathscr{MM}(st(\mathscr{P}))$. This means that, after having computed the minimum model of $st(\mathscr{P})$, we can derive a finite ground instantiation of $\mathscr{P}$, equivalent to the original program, by considering only ground atoms contained in $\mathscr{MM}(st(\mathscr{P}))$.

We next show how the original program $\mathscr{P}$ can be rewritten so that, after having computed $\mathscr{MM}(st(\mathscr{P}))$, every grounder tool easily generates an equivalent finitely ground program. The idea consists in generating, for any disjunctive program $\mathscr{P}$ such that $st(\mathscr{P})$ satisfies some termination criterion (e.g. safety), a new equivalent program $\mathrm{ext}(\mathscr{P})$. The computation of the stable models of $\mathrm{ext}(\mathscr{P})$ can be carried out by considering the finite ground instantiation of $\mathrm{ext}(\mathscr{P})$ (Leone *et al.* 2002; Simons *et al.* 2002; Gebser *et al.* 2007a).

For any disjunctive rule $r = q_1(u_1) \vee \cdots \vee q_k(u_k) \leftarrow \mathrm{body}(r)$, the conjunction of atoms $Q_1(u_1), ..., Q_k(u_k)$ will be denoted by $\mathrm{headconj}(r)$.

Definition 15 (*Extended program*)
Let $\mathscr{P}$ be a disjunctive program and let r be a rule of $\mathscr{P}$, then, $\mathrm{ext}(r)$ denotes the (disjunctive) *extended rule* $\mathrm{head}(r) \leftarrow \mathrm{headconj}(r), \mathrm{body}(r)$ obtained by extending the body of r, whereas $\mathrm{ext}(\mathscr{P}) = \{\mathrm{ext}(r) \mid r \in \mathscr{P}\} \cup ST(\mathscr{P})$ denotes the (disjunctive) *extended program* obtained by extending the rules of $\mathscr{P}$ and adding (standard) rules defining the new predicates. □

Example 20
Consider the program P_{19} of Example 19. The extended program $\mathrm{ext}(P_{19})$ is as follows:

$$\begin{array}{l}
\mathtt{p(X)} \vee \mathtt{q(X)} \leftarrow \mathtt{P(X)}, \mathtt{Q(X)}, \mathtt{r(X)}, \neg\mathtt{a(X)}.\\
\mathtt{r(X)} \leftarrow \mathtt{R(X)}, \mathtt{b(X)}, \neg\mathtt{q(X)}.\\
\mathtt{P(X)} \leftarrow \mathtt{R(X)}.\\
\mathtt{Q(X)} \leftarrow \mathtt{R(X)}.\\
\mathtt{R(X)} \leftarrow \mathtt{b(X)}.
\end{array}$$
□

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

Theorem 5
For every program $\mathscr{P}$, $\mathscr{SM}(\mathscr{P})[S_{\mathscr{P}}] = \mathscr{SM}(\mathrm{ext}(\mathscr{P}))[S_{\mathscr{P}}]$, where $S_{\mathscr{P}}$ is the set of predicate symbols occurring in $\mathscr{P}$.

Proof
First, we recall that $ST(\mathscr{P}) \subseteq \mathrm{ext}(\mathscr{P})$ and assume that N is the minimum model of $ST(\mathscr{P})$, i.e. $N = \mathscr{MM}(ST(\mathscr{P}))$.

- We first show that for each $S \in \mathscr{SM}(\mathrm{ext}(\mathscr{P}))$, $M = S - N$ is a stable model for $\mathscr{P}$, that is $M \in \mathscr{SM}(\mathscr{P})$.
  Let us consider the ground program $\mathscr{P}''$ obtained from $\mathrm{ext}(\mathscr{P})^S$ by first deleting every ground rule $r = \mathrm{head}(r) \leftarrow \mathrm{headconj}(r), \mathrm{body}(r)$ such that $N \not\models \mathrm{headconj}(r)$ and then by removing from the remaining rules, the conjunction $\mathrm{headconj}(r)$. Observe that the sets of minimal models for $\mathrm{ext}(\mathscr{P})^S$

and $\mathscr{P}''$ coincide, i.e. $\mathscr{M}\mathscr{M}(\text{ext}(\mathscr{P})^S) = \mathscr{M}\mathscr{M}(\mathscr{P}'')$. Indeed, for every r in $\text{ext}(\mathscr{P})^S$, if $N \not\models \text{headconj}(r)$, then the body of r is false and thus r can be removed as it does not contribute to infer head atoms. On the other hand, if $N \models \text{headconj}(r)$, the conjunction $\text{headconj}(r)$ is trivially true, and can be safely deleted from the body of r.

Therefore, $M \cup N \in \mathscr{M}\mathscr{M}(\mathscr{P}'')$. Moreover, since $\mathscr{P}'' = (\mathscr{P} \cup ST(\mathscr{P}))^S = \mathscr{P}^M \cup ST(\mathscr{P})^N$, we have that $M \in \mathscr{M}\mathscr{M}(\mathscr{P}^M)$, that is $M \in \mathscr{S}\mathscr{M}(\mathscr{P})$.

- We now show that for each $M \in \mathscr{S}\mathscr{M}(\mathscr{P})$, $(M \cup N) \in \mathscr{S}\mathscr{M}(\text{ext}(\mathscr{P}))$.
  Let us assume that $S = M \cup N$. Since $M \in \mathscr{M}\mathscr{M}(\mathscr{P}^M)$ we have that $S \in \mathscr{S}\mathscr{M}(\mathscr{P} \cup ST(\mathscr{P}))$, that is $S \in \mathscr{M}\mathscr{M}((\mathscr{P} \cup ST(\mathscr{P}))^S)$. Consider the ground program $\mathscr{P}'$ derived from $(\mathscr{P} \cup ST(\mathscr{P}))^S$ by replacing every rule disjunctive $r = \text{head}(r) \leftarrow \text{body}(r)$ such that $M \models \text{body}(r)$ with $\text{ext}(r) = \text{head}(r) \leftarrow \text{headconj}(r), \text{body}(r)$. Also in this case we have that $\mathscr{M}\mathscr{M}(\mathscr{P} \cup ST(\mathscr{P}))^S) = \mathscr{M}\mathscr{M}(\mathscr{P}')$ as $S \models \text{body}(r)$ iff $S \models \text{body}(\text{ext}(r))$. This, means that S is a stable model for $\text{ext}(\mathscr{P})$.

$\square$

9 Conclusion

In this paper we have proposed a new approach for checking, on the basis of structural properties, termination of the bottom-up evaluation of logic programs with function symbols. We have first proposed a technique, called Γ-*acyclicity*, extending the class of argument-restricted programs by analyzing the propagation of complex terms among arguments using an extended version of the argument graph. Next, we have proposed a further extension, called *safety*, which also analyzes how rules can activate each other (using the activation graph) and how the presence of some arguments in a rule limits its activation. We have also studied the application of the techniques to partially ground queries and have proposed further improvements which generalize the safety criterion through the introduction of a hierarchy of classes of terminating programs, called k-*safety*, where each k-safe class strictly includes the (k-1)-safe class.

Although our results have been defined for standard programs, we have shown that they can also be applied to disjunctive programs with negative literals, by simply rewriting the source programs. The semantics of the rewritten program is ‘equivalent’ to the semantics of the source one and can be computed by current answer set systems. Even though our framework refers to the model theoretic semantics, we believe that the results presented here go beyond the ASP community and could be of interest also for the (tabled) logic programming community (e.g. tabled Prolog community).

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

Alviano, M., Faber, W. and Leone, N. 2010. Disjunctive asp with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming 10,* 4–6, 497–512.

Arts, T. and Giesl, J. 2000. Termination of term rewriting using dependency pairs. *Theoretical Computer Science 236,* 1–2, 133–178.

Baselice, S., Bonatti, P. A. and Criscuolo, G. 2009. On finitely recursive programs. *Theory and Practice of Logic Programming 9,* 2, 213–238.

Beeri, C. and Ramakrishnan, R. 1991. On the power of magic. *Journal of Logic Programming 10,* 1–4, 255–299.

Bonatti, P. A. 2004. Reasoning with infinite stable models. *Artificial Intelligence 156,* 1, 75–111.

Brockschmidt, M., Musiol, R., Otto, C. and Giesl, J. 2012. Automated termination proofs for java programs with cyclic data. In *Proc. of 24th International Conference on Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer-Verlag, Berlin, 105–122.

Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S. and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29,* 2.

Calimeri, F., Cozza, S., Ianni, G. and Leone, N. 2008. Computable functions in ASP: Theory and implementation. In *Proc. of 24th International Conference on Logic Programming (ICLP)*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, Berlin, 407–424.

Codish, M., Lagoon, V. and Stuckey, P. J. 2005. Testing for termination with monotonicity constraints. In *Proc. of 21st International Conference on Logic Programming (ICLP)*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, Berlin, 326–340.

Endrullis, J., Waldmann, J. and Zantema, H. 2008. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning 40,* 2–3, 195–220.

Fagin, R., Kolaitis, P. G., Miller, R. J. and Popa, L. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science 336,* 1, 89–124.

Ferreira, M. C. F. and Zantema, H. 1996. Total termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing 7,* 2, 133–162.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007a. *clasp* : A conflict-driven answer set solver. In *Proc. of 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer-Verlag, Berlin, 260–265.

Gebser, M., Schaub, T. and Thiele, S. 2007b. Gringo : A new grounder for answer set programming. In Proc. of 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer-Verlag, Berlin, 266–271.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of 5th International Conference and Symposium on Logic Programming (ICLP/SLP)*, R. A. Kowalski and K. A. Bowen, Eds. Series in Logic Programming, vol. 2. MIT Press, Cambridge, MA, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Greco, G., Greco, S., Trubitsyna, I. and Zumpano, E. 2005. Optimization of bound disjunctive queries with constraints. *Theory and Practice of Logic Programming 5,* 6, 713–745.

Greco, S. 2003. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Transactions on Knowledge and Data Engineering 15,* 2, 368–385.

Greco, S. and Spezzano, F. 2010. Chase termination: A constraints rewriting approach. *Proceedings of the VLDB Endowment (PVLDB) 3,* 1, 93–104.

Greco, S., Spezzano, F. and Trubitsyna, I. 2011. Stratification criteria and rewriting techniques for checking chase termination. *Proceedings of the VLDB Endowment (PVLDB) 4,* 11, 1158–1168.

Greco, S., Spezzano, F. and Trubitsyna, I. 2012. On the termination of logic programs with function symbols. In *Proc. of International Conference on Logic Programming (Technical Communications).* 323–333. **Q2**

Krishnamurthy, R., Ramakrishnan, R. and Shmueli, O. 1996. A framework for testing safety and effective computability. *Journal of Computer and System Sciences 52,* 1, 100–124.

Leone, N., Pfeifer, G., Faber, W., Calimeri, F., Dell'Armi, T., Eiter, T., Gottlob, G., Ianni, G., Ielpa, G., Koch, K., Perri, S. and Polleres, A. 2002. The DLV system. In *Proc. of 8th European Conference on Logics in Artificial Intelligence (JELIA)*, S. Flesca, S. Greco, G. Ianni and N. Leonne, Eds. Lecture Notes in Computer Science, vol. 2424. Springer-Verlag, Berlin, 537–540.

Liang, S. and Kifer, M. 2013. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming 13,* 4–5, 705–719.

Lierler, Y. and Lifschitz, V. 2009. One more decidable class of finitely ground programs. In *Proc. of 25th International Conference on Logic Programming (ICLP)*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, Berlin, 489–493.

Marchiori, M. 1996. Proving existential termination of normal logic programs. In *Proc. of 5th International Conference on Algebraic Methodology and Software Technology*, M. Wirsing and M. Nivat, Eds. Lecture Notes in Computer Science, vol. 1101. Springer-Verlag, Berlin, 375–390.

Marnette, B. 2009. Generalized schema-mappings: From termination to tractability. In *Proc. of 28th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS).* ACM, New York, NY, 13–22.

Meier, M., Schmidt, M. and Lausen, G. 2009. On chase termination beyond stratification. *CoRR abs/0906.4228.*

Nguyen, M. T., Giesl, J., Schneider-Kamp, P. and Schreye, D. D. 2007. Termination analysis of logic programs based on dependency graphs. In *Proc. of 17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR*), A. King, Ed. Lecture Notes in Computer Science, vol. 4915. Springer-Verlag, Berlin, 8–22.

Nishida, N. and Vidal, G. 2010. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing 21,* 3, 177–225.

Ohlebusch, E. 2001. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing 12,* 1/2, 73–116.

Schneider-Kamp, P., Giesl, J. and Nguyen, M. T. 2009b. The dependency triple framework for termination of logic programs. In *Proc. of 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR*), D. D. Schreye, Ed. Lecture Notes in Computer Science, vol. 6037. Springer-Verlag, Berlin, 37–51.

Schneider-Kamp, P., Giesl, J., Serebrenik, A. and Thiemann, R. 2009a. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic 11,* 1.

Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A. and Thiemann, R. 2010. Automated termination analysis for logic programs with cut. *Theory and Practice of Logic Programming 10,* 4–6, 365–381.

Schreye, D. D. and Decorte, S. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19/20*, 199–260.

Serebrenik, A. and De Schreye, D. 2005. On termination of meta-programs. *Theory and Practice of Logic Programming 5,* 3, 355–390.

Simons, P., Niemela, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2, 181–234.

Sternagel, C. and Middeldorp, A. 2008. Root-labeling. In *Proc. of 19th International Conference on Rewriting Techniques and Applications*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 5117. Springer-Verlag, Berlin, 336–350.

Ströder, T., Schneider-Kamp, P. and Giesl, J. 2010. Dependency triples for improving termination analysis of logic programs with cut. In *Proc. of 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, M. Alpuente, Ed. Lecture Notes in Computer Science, vol. 6564. Springer-Verlag, Berlin, 184–199.

Syrjänen, T. 2001. Omega-restricted logic programs. In *Proc. of 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, T. Eiter, W. Faber and M. L. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer-Verlag, Berlin, 267–279.

Venturini Zilli, M. 1975. Complexity of the unification algorithm for first-order expressions. *CALCOLO 12,* 4, 361–371.

Voets, D. and Schreye, D. D. 2010. Non-termination analysis of logic programs using types. In *Proc. of 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, M. Alpuente, Ed. Lecture Notes in Computer Science, vol. 6564. Springer-Verlag, Berlin, 133–148.

Zantema, H. 1995. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae 24,* 1/2, 89–105.