Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols

Citation for published version:
Calautti, M, Greco, S & Trubitsyna, I 2013, Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols. in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming.* PPDP '13, ACM, New York, NY, USA, pp. 239-250.
https://doi.org/10.1145/2505879.2505883

Digital Object Identifier (DOI):
10.1145/2505879.2505883

Published In:
Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming

Detecting decidable classes of finitely ground logic programs with function symbols

Marco Calautti
DIMES, Università della Calabria
87036 Rende (CS), Italy
calautti@dimes.unical.it

Sergio Greco
DIMES, Università della Calabria
87036 Rende (CS), Italy
greco@dimes.unical.it

Irina Trubitsyna
DIMES, Università della Calabria
87036 Rende (CS), Italy
trubitsyna@dimes.unical.it

ABSTRACT

In this paper we propose a new technique for checking whether the bottom-up evaluation of logic programs with function symbols terminates. The technique is based on the definition of *mappings* from arguments to strings of function symbols, representing possible values which could be taken by arguments during the bottom-up evaluation. Such mappings can be computed by transforming the original program into a unary logic program whose termination is decidable. Starting from mappings we can identify *mapping-restricted* arguments, a subset of limited arguments, that is, arguments which can take values from finite domains. The class of mapping-restricted programs, consisting of programs whose arguments are mapping-restricted, is terminating under the bottom-up computation as all its arguments can take values from finite domains. We study the complexity of the presented approach and compare it with other techniques known in the literature. The presented technique is relevant as it individuates as terminating programs not detected by other criteria proposed so far and can be combined with other techniques to further enlarge the class of programs recognized as terminating under the bottom-up evaluation.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Languages

Keywords

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

1. INTRODUCTION

PPDP '13, September 16 - 18 2013, Madrid, Spain Copyright is held by Marco Calautti, Sergio Greco and Irina Trubitsyna. Publication rights licensed to ACM.
ACM 978-1-4503-2154-9/13/09...$15.00.
http://dx.doi.org/10.1145/2505879.2505883.

Recent developments of answer set solvers have seen significant progress towards providing support for function symbols. The interest in this area is justified by the fact that function symbols make languages more expressive and often make modeling easier and the resulting encodings more readable and concise. The main problem with the introduction of function symbols is that common inference tasks become undecidable. A significant body of work has been done on termination of logic programs under top-down evaluation [28, 32, 20, 24, 9, 29, 23, 26, 25, 27, 22, 4, 3, 2] and in the area of chase termination [16, 17, 13]. In this paper, we consider logic programs with function symbols *under the stable model semantics* [11, 12], and thus, as already discussed in [5, 6, 1, 14], all the excellent works above referred cannot straightforwardly be applied to our setting. Considering this context, recent years have witnessed an increasing interest in the problem of identifying logic programs with function symbols for which a finite set of finite stable models exists and can be computed. The class of *finitely ground programs*, guaranteeing the aforementioned property, has been proposed in [5]. Since membership in the class is semi-decidable, recent research has concentrated on the identification of sufficient conditions, that we call *termination criteria*, for a program to be finitely ground. Efforts in this direction are ω*-restricted programs* [30], λ*-restricted programs* [10], *finite domain programs* [5], *argument-restricted programs* [19], *safe programs* [18], Γ*-acyclic programs* [18], and *bounded programs* [14].

Current techniques analyze how values are propagated among predicate arguments, to understand whether such arguments are *limited*, i.e. whether the set of values which can be associated with an argument is finite. However, these methods have limited capacity to analyze the propagation of function symbols during the bottom-up evaluation and they often cannot understand that recursive rules cannot be activated starting from exit rules. Consequently, current techniques are not able to identify as terminating even simple programs whose bottom-up evaluation always terminates. Below is an example.

Example 1. Consider the following program P_1

$$
\begin{array}{lcl}
r_1 : \mathtt{p}(\mathtt{X},\mathtt{f}(\mathtt{X})) & \leftarrow & \mathtt{b}(\mathtt{X}). \\
r_2 : \mathtt{p}(\mathtt{f}(\mathtt{X}),\mathtt{X}) & \leftarrow & \mathtt{b}(\mathtt{X}). \\
r_3 : \mathtt{q}(\mathtt{f}(\mathtt{X}),\mathtt{g}(\mathtt{X})) & \leftarrow & \mathtt{p}(\mathtt{X},\mathtt{X}). \\
r_4 : \mathtt{q}(\mathtt{f}(\mathtt{X}),\mathtt{f}(\mathtt{X})) & \leftarrow & \mathtt{q}(\mathtt{X},\mathtt{X}).
\end{array}
$$

where b is a base predicate, whereas p and q are derived predicates. The program is not recognized as terminating by current criteria that cannot understand that arguments q[1] and q[2] are limited, that is, during the bottom-up evaluation predicate q can take only a limited set of values for both its arguments. Indeed, for all instances of predicate b, rule r_3 cannot be activated as it requires that arguments in the body atom must have the same value. Consequently, the recursive rule r_4 can never be activated and P_1 has a finite minimum model for every possible database instance. □

Thus, in this paper we present a new technique for checking termination of the bottom-up evaluation of logic programs with function symbols. Although we concentrate on positive, normal programs, the technique can be immediately applied to general programs with negation and head disjunction.

Contribution.

We introduce the concept of *mapping* to describe the form of atoms derivable during the bottom-up evaluation of the program and use it to identify *mapping-restricted* arguments, a subset of limited arguments, that is, arguments which can take values from finite domains. We show that mapping-restricted arguments are limited and can be computed by transforming the original program into a unary logic program, belonging to Datalog$_{nS}$, a class of Datalog programs studied in the early 1990s whose finiteness of the minimum model is decidable [8].

We also show that mapping-restricted arguments of the original program correspond to the limited arguments of the transformed program and, using results obtained for Datalog$_{nS}$, we show that their identification is space polynomial in the size of the original program in the presence of just one function symbol.

We discuss the relationship between the class of argument-restricted programs and the class of programs recognized by the new criterion and show that it generalizes previous proposed techniques (e.g. argument-restricted criterion) and is not captured by none of previous techniques.

Finally, we present a modified version of the safe function introduced in [18] to define the classes of safe and Γ-acyclic programs. We show that the refined safe function can be used to further enlarge the set of arguments recognized as limited. We also discuss a combined use of the new technique with other approaches previously proposed.

Organization.

The paper is organized as follows. In Section 2 we cover preliminaries on logic programs with function symbols and Datalog$_{nS}$ programs. Section 3 covers current termination criteria, even though they are not required for understanding the technique proposed in this paper. Then, we present the new technique in Section 4 showing also how it relates to other termination criteria. Finally, we show further improvements and complexity results in Section 5 and Section 6, then we conclude.

2. LOGIC PROGRAMS

Syntax.

We assume to have infinite sets of constants, variables, predicate symbols and function symbols. Predicate and function symbols have associated a fixed arity. Predicates symbols are partitioned into two different classes: *base* (or extensional) and *derived* (or intensional). The arity of a predicate or function symbol g will be denoted by $arity(g)$. For a predicate p of arity n, we denote by $p[i]$, for $1 \leq i \leq n$, its i-th argument.

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

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

where $m > 0$, $k, n \geq 0$ and $a_1, \cdots, a_m, b_1, \cdots, b_k, c_1, \cdots, c_n$ are atoms. The disjunction $a_1 \vee \cdots \vee a_m$ is called the *head* of r and is denoted by $head(r)$ while the conjunction $b_1, \cdots, b_k$, $\neg c_1, \cdots, \neg c_n$ is called the *body* and is denoted by $body(r)$. If $m = 1$, then r is *normal* (i.e. $\vee$-free), whereas if $n = 0$, then r is *positive* (i.e. $\neg$-free). With a little abuse of notation we often use $body(r)$ (resp. $head(r)$) to also denote the set of literals appearing in the body (resp. head) of r. We also denote the *positive body* of r by $body^+(r) = \{b_1, \ldots, b_k\}$ and the *negative body* of r by $body^-(r) = \{c_1, \ldots, c_n\}$.

We assume that rules are *range restricted* [31], i.e. variables appearing in the head or in negated body literals are range restricted, that is they also appear in some positive body literal[1]. A term (resp. an atom, a rule) is said to be *ground* if no variables occur in it. The *depth* of a term t is defined as follows: $depth(t) = 0$ if t is a constant or a variable, and $depth(t) = 1 + \max_{1 \leq i \leq n} \{depth(t_i)\}$ if t is a complex term of the form $f(t_1, \ldots, t_n)$. A ground normal rule with an empty body is also called *fact*. The definition of a predicate symbol p consists of all rules and facts having p in the head. Base predicates are defined by facts, whereas derived predicates are defined by rules. Given a set of rules and facts, we denote with D the database consisting of all facts defining base predicates and with $\mathcal{P}$ the program consisting of all rules defining derived predicates. Without loss of generality, we also assume that constants appearing in rules defining derived predicates also appear in database facts and that complex terms do not appear in database facts. The program consisting of rules defining derived predicates and facts defining base predicates is denoted by $\mathcal{P}_D$ (equal to $\mathcal{P} \cup D$). The set of all arguments of a program $\mathcal{P}$ is denoted by $arg(\mathcal{P})$. The set of *base arguments* of $\mathcal{P}$, i.e. arguments of base predicates of $\mathcal{P}$, is denoted by $arg_b(\mathcal{P})$. Given a program $\mathcal{P}_D$, a predicate p depends on a predicate q if there is a rule r in $\mathcal{P}$ 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.

Semantics.

[1]Range restricted programs are often called safe programs. We will use the term safe to denote further restricted programs.

The Herbrand universe $H_{\mathcal{P}_D}$ of a program $\mathcal{P}_D$ is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in $\mathcal{P}_D$. The Herbrand base $B_{\mathcal{P}_D}$ of a program $\mathcal{P}_D$ is the set of ground atoms which can be built using predicate symbols appearing in $\mathcal{P}_D$ and ground terms of $H_{\mathcal{P}_D}$. A rule r' is a *ground instance* of a rule r, if r' is obtained from r by replacing every variable in r with some ground term in $H_{\mathcal{P}}$; $ground(\mathcal{P})$ denotes the set of all ground instances of the rules in $\mathcal{P}$. An interpretation of a program $\mathcal{P}_D$ is any subset of $B_{\mathcal{P}_D}$. The value of a ground atom L w.r.t. an interpretation I is $value_I(L) = L \in I$, whereas $value_I(\neg L) = L \notin I$. The truth value of a conjunction of ground literals $C = L_1, \ldots, L_n$ is $value_I(C) = true \wedge value_I(L_1) \wedge \cdots \wedge value_I(L_n)$, while the truth value of a disjunction $D = L_1 \vee \cdots \vee L_n$ is $value_I(D) = false \vee value_I(L_1) \vee \cdots \vee value_I(L_n)$, where $true$ and $false$ are built-in truth values such that $false < true$. A ground rule r is *satisfied* by I if $value_I(head(r)) \geq value_I(body(r))$. Thus, a rule r with an empty body is satisfied by I if $value_I(head(r)) = true$. An interpretation M for $\mathcal{P}_D$ is a model of $\mathcal{P}_D$ if M satisfies all the rules in $ground(\mathcal{P}_D)$. Given an interpretation M and a ground rule r, we write $M \models r$ if M satisfies r, and $M \not\models r$ if M does not satisfy r.

The *model-theoretic semantics* for a positive program $\mathcal{P}_D$ assigns the set of its *minimal models* $\mathcal{MM}(\mathcal{P}_D)$. A model M for $\mathcal{P}_D$ is minimal if no proper subset of M is a model for $\mathcal{P}_D$ [21]. The more general *disjunctive stable model semantics* generalizes stable model semantics previously defined for normal programs [11] and also applies to programs with (unstratified) negation [12].

The set of stable models of $\mathcal{P}_D$ is denoted by $\mathcal{SM}(\mathcal{P}_D)$. It is well known that stable models are minimal models (i.e. $\mathcal{SM}(\mathcal{P}_D) \subseteq \mathcal{MM}(\mathcal{P}_D)$) and that for negation-free programs minimal and stable model semantics coincide (i.e. $\mathcal{SM}(\mathcal{P}_D) = \mathcal{MM}(\mathcal{P}_D)$) and that positive normal programs have a unique minimal model, called *minimum model*.

In the presence of function symbols, logic programs may be non terminating, i.e. may have stable models of infinite size. Given a program $\mathcal{P}_D$ and one of its models M, an argument $q[i]$ in $arg(\mathcal{P})$ is said to be *limited in M* iff the set $\{t_i \mid q(t_1, ..., t_i, ..., t_n) \in M\}$ is finite. An argument $q[i]$ in $arg(\mathcal{P})$ is said to be *limited* iff for every finite set of database facts D and for every stable model M of $\mathcal{P}_D$, $q[i]$ is limited in M.

Datalog.

Datalog [31] is the class of function-free logic programs, where predicates are partitioned into base and derived and the only terms are constants or variables, called *data terms*. Different extensions of Datalog have been studied in the literature, including programs with stratified and general negation, programs with disjunctive heads and programs with negation and disjunctive heads. It is well known that the complexity of computing the minimum model for Datalog programs is polynomial in the size of the input databases.

Datalog$_{nS}$ (*Datalog* with n successors), proposed twenty years ago in [8], is an extension of Datalog with a limited use of function symbols capable of representing infinite phenomena like flow of time, state transitions, construction of plans, etc. An example of a Datalog$_{nS}$ program is reported below.

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

$$r: \ \mathtt{meets(T+1, Y)} \ \leftarrow \ \mathtt{follows(X, Y), meets(T, X)}.$$

where $T + 1$ is a shorthand for $+1(T)$ and $+1$ is a function symbol. Rule r schedules the meetings of graduate students with their common advisor, where $\mathtt{meets(t, x)}$ means that x meets her/his advisor in day t. □

The problem of checking the finiteness of the minimum model of Datalog$_{nS}$ programs is decidable [8]. Predicates in Datalog$_{nS}$ can have an arbitrary number of function symbols and they can appear in one fixed argument. Without loss of generality we assume that function symbols are all unary since in Datalog$_{nS}$ every k-ary symbol has $k-1$ positions in which only data terms can occur, and then can be rewritten into many unary function symbols. We also assume that the distinguished argument of a predicate, where function symbols may occur, is the first one. This argument is called *functional*, in addition to usual *data arguments*, and corresponds to a *state* (in Example 2 each *state* represents a particular moment of time), whereas function symbols map a state to another. Predicates containing a functional argument are called functional too. Functional arguments contain *functional terms*, which are built from a distinguished functional constant 0, functional variables and function symbols. Other (data) arguments of an atom can only contain data terms. A data term by itself is not a functional term. This implies that every functional term contains either 0 or a single occurrence of a single functional variable. We also assume that a functional variable in a rule is unique and is denoted by T. For instance, in the program $\mathcal{P}_2$ of Example 2, terms 0, T and $T + 1$ are functional terms, $meets[1]$ is a functional argument, whereas $follows[1], follows[2], meets[2]$ are data arguments.

Other syntactical restrictions of Datalog$_{nS}$ programs hold: i) rules are range restricted, ii) equality and inequality operators are only applied to data terms iii) rule bodies are nonempty, iv) rules do not contain ground terms, and v) functional terms in rules are of depth at most 1.

Datalog$_{1S}$ is a particular subclass of Datalog$_{nS}$ admitting exactly one unary function symbol (+1), so that functional ground terms can simply be seen as numbers representing time. For the sake of presentation, in the following we will briefly review the semantics of Datalog$_{1S}$ programs.

Example 3. Consider the program $\mathcal{P}_3$ obtained from $\mathcal{P}_2$ of Example 2 plus the rule:

$$\mathtt{meets(T, Y)} \leftarrow \mathtt{start(T, Y)}.$$

and the following database D_3:

```
start(0, emma).
follows(emma, kathy).
follows(kathy, emma).
```

The minimal model M_3 of this program is composed by facts

```
follows(emma, kathy)   follows(kathy, emma)
start(0, emma)         meets(0, emma)
```

and the following regularly repeating functional facts:

```
meets(1, kathy)   meets(2, emma)
meets(3, kathy)   meets(4, emma)
meets(5, kathy)   meets(6, emma)
...               ...
```

where 1 is an abbreviation for 0+1, 2 is an abbreviation for (0+1)+1, and so on. □

Let $\mathcal{P}_D$ be a Datalog$_{1S}$ program, M be the model of $\mathcal{P}_D$ and t a ground functional term, the *state* $M[t]$ of M is $M[t] = \{p(\bar{a}) \mid p(t, \bar{a}) \in M\}$; the *snapshot* $M(t)$ of M is $M(t) = \{p(t, \bar{a}) \mid p(t, \bar{a}) \in M\}$; the *data part* M^d of M is the set of all the data facts in M. The *period* of M is a pair (t_1, t_2), where ground functional terms t_1 and t_2 are such that $t_1 < t_2$ and represent the smallest different times with the same state. It has been shown in [8] that $M[t_1 + k] = M[t_2 + k]$ for all $k \geq 0$.

Example 4. Consider the program $\mathcal{P}_3$ and the database D_3 from previous examples. Let M_3 be the minimal model of $\mathcal{P}_3 \cup D_3$. Examples of state, snapshot and data part of M_3 are $M_3[0] = \{\texttt{start(emma)}, \texttt{meets(emma)}\}$, $M_3(0) = \{\texttt{start(0, emma)}, \texttt{meets(0, emma)}\}$ and $M^d = \{\texttt{follows(emma, kathy)}, \texttt{follows(kathy, emma)}\}$. Intuitively, M_3 repeats with period $(1, 3)$, i.e. $M_3[1 + k] = M_3[3 + k]$ for every $k \geq 0$. □

It has been shown in [8] that every Datalog$_{1S}$ program has a "periodic" minimal model and the finiteness of the model of a Datalog$_{1S}$ program $\mathcal{P}_D$ can be checked in polynomial space in the number of facts of $\mathcal{P}_D$.

Activation and argument graphs.

Let $\mathcal{P}$ be a program and r_1, r_2 be (not necessarily distinct) rules of $\mathcal{P}$. We say that r_1 *activates* r_2 iff there exist two ground rules $r'_1 \in ground(r_1)$, $r'_2 \in ground(r_2)$ and a set of ground atoms D such that *(i)* $D \not\models r'_1$, *(ii)* $D \models r'_2$, and *(iii)* $D \cup head(r'_1) \not\models r'_2$. This intuitively means that if D does not satisfy r'_1, D satisfies r'_2, and $head(r'_1)$ is added to D to satisfy r'_1, this causes r'_2 not to be satisfied anymore (and then to be "activated").

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

The *argument graph* of a program $\mathcal{P}$, denoted $G(\mathcal{P})$, is a directed graph whose nodes are $arg(\mathcal{P})$ (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 *i)* an atom $p(t_1, ..., t_n)$ appears in $head(r)$, *ii)* an atom $q(u_1, ..., u_m)$ appears in $body^+(r)$ and *iii)* terms t_i and u_j have a common variable.

In the following we will also consider labelled graphs, i.e. graphs with labelled 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* ρ from a_1 to b_m in a possibly labelled 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 $1 \leq i < m$; if the first and last nodes coincide (i.e., $a_1 = b_m$), then ρ 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*. A cycle is *basic* if it does not contain two occurrences of the same edge. Given a cycle π consisting of n (labelled) edges $e_1, ..., e_n$, we can derive n different cyclic paths starting from each of the e_i's—we use $\tau(\pi)$ to denote the set of such cyclic paths. As an example, if π is a cycle consisting of edges e_1, e_2, e_3, then $\tau(\pi) = \{(e_1, e_2, e_3), (e_2, e_3, e_1), (e_3, e_2, e_1)\}$.

We say that a node $p[i]$ *depends on* a node $q[j]$ in a graph iff there is a path from $q[j]$ to $p[i]$ in that graph. Moreover, we say that $p[i]$ *depends on* a cycle π iff it depends on a node $q[j]$ appearing in π. Clearly, nodes belonging to a cycle π depend on π.

3. TERMINATION CRITERIA

As we discussed in the introduction, the problem of identifying finitely ground logic programs, having a finite set of finite stable models, is semi-decidable. Different decidable criteria proposed in the literature [30, 10, 5, 19, 18, 14] allow to determine the termination of logic programs by considering the propagation of complex terms among arguments of the program. In particular, they detect a subset of limited arguments, so that if all arguments are in the set, program termination is guaranteed. The sets of programs satisfying criterion C define the corresponding class of terminating programs $\mathcal{C}$. In the following we briefly describe some of the known termination criteria proposed in the literature that are relevant to the technique proposed in this paper.

Argument-restricted programs [19]. For every atom A of the form $p(t_1, ..., t_n)$, A^0 denotes the predicate symbol p, and A^i denotes term t_i, for $1 \leq i \leq n$. The *depth* $d(X, t)$ of a variable X in a term t that contains X is recursively defined as follows:

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

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

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

The set of *restricted arguments* of $\mathcal{P}$ is $AR(\mathcal{P}) = \{p[i] \mid p[i] \in arg(\mathcal{P}) \wedge \exists \phi \; s.t. \; \phi(p[i]) \text{ is defined}\}$. A program $\mathcal{P}$ is said to be *argument restricted* iff $AR(\mathcal{P}) = arg(\mathcal{P})$. The class of argument restricted programs is denoted by $\mathcal{AR}$. □

Bounded Programs [14]. The definition of bounded programs relies on the notion of *labelled argument graph*. This graph, denoted $\mathcal{G}_L(\mathcal{P})$, is derived from the argument graph by labelling edges as follows: for each pair of nodes $p[i], q[j] \in arg(\mathcal{P})$ and for every rule $r \in \mathcal{P}$ such that *(i)* an atom $p(t_1, ..., t_n)$ appears in $head(r)$, *(ii)* an atom $q(u_1, ..., u_m)$ appears in $body^+(r)$, *(iii)* terms t_i and u_j have a common variable X, there is an edge $(q[j], p[i], \langle \alpha, r, h, k \rangle)$, where h and k are natural numbers denoting the positions of $p(t_1, ..., t_n)$ in $head(r)$ and $q(u_1, ..., u_m)$ in $body^+(r)$, respectively[2], whereas $\alpha = \epsilon$ if $t_i = u_j$, $\alpha = f$ if $u_j = X$ and $t_i = f(..., X, ...)$, $\alpha = \bar{f}$ if $u_j = f(..., X, ...)$ and $t_i = X$. Moreover, it is also assumed that if the same variable X appears in two terms occurring in the head and body of a rule respectively, then only one of the two terms is a complex term and that the nesting level of complex terms is at most one.

Given a path $\rho = (a_1, b_1, \langle \alpha_1, r_1, h_1, k_1 \rangle), \ldots, (a_m, b_m, \langle \alpha_m, r_m, h_m, k_m \rangle)$, we define $\lambda_1(\rho) = \alpha_1 \ldots \alpha_m$, $\lambda_2(\rho) = r_1, ..., r_m$, and $\lambda_3(\rho) = \langle r_1, h_1, k_1 \rangle \ldots \langle r_m, h_m, k_m \rangle$. Given two cycles π_1 and π_2, we write $\pi_1 \approx \pi_2$ iff $\exists \rho_1 \in \tau(\pi_1)$ and $\exists \rho_2 \in \tau(\pi_2)$ such that $\lambda_3(\rho_1) = \lambda_3(\rho_2)$.

[2]We assume that literals in the head (resp. body) are ordered with the first one being associated with 1, the second one with 2, etc.

Given a program $\mathcal{P}$, we say that a cycle π in $\mathcal{G}_L(\mathcal{P})$ is *active* iff $\exists \rho \in \tau(\pi)$ such that $\lambda_2(\rho) = r_1, ..., r_m$ and $(r_1, r_2), ..., (r_{m-1}, r_m), (r_m, r_1)$ is a cyclic path in the activation graph $\Omega(\mathcal{P})$.

Given a program $\mathcal{P}$ and a path ρ in $\mathcal{G}_L(\mathcal{P})$, we denote with $\hat{\lambda}_1(\rho)$ the string obtained from $\lambda_1(\rho)$ by iteratively eliminating pairs of the form $\gamma\bar{\gamma}$ from the string until the resulting string cannot be further reduced.

Given a program $\mathcal{P}$, a cycle π in $\mathcal{G}_L(\mathcal{P})$ can be classified as follows. We say that π is i) *balanced* if $\exists \rho \in \tau(\pi)$ s.t. $\hat{\lambda}_1(\rho)$ is empty, ii) *growing* if $\exists \rho \in \tau(\pi)$ s.t. $\hat{\lambda}_1(\rho)$ does not contain a symbol of the form $\bar{\gamma}$, iii) *failing* otherwise.

Definition 2. Given a program $\mathcal{P}$, the set of *bounded* arguments $bounded(\mathcal{P})$ is computed by first setting $bounded(\mathcal{P}) = AR(\mathcal{P})$ and next iteratively adding each argument $q[k]$ such that for each basic cycle π in $\mathcal{G}_L(\mathcal{P})$ on which $q[k]$ depends, at least one of the following conditions holds:

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

A program $\mathcal{P}$ is said to be *bounded* if all its arguments are bounded. The class of bounded programs is denoted by $\mathcal{BP}$. □

A relevant aspect that distinguishes this technique from other works, including the presented proposal, is that this technique analyzes how groups of arguments are related to each other. This aspect is illustrated by the below example.

Example 5. Consider the logic program $\mathcal{P}_5$ below:

r_0 : $\mathtt{count([a,b,c],0).}$
r_1 : $\mathtt{count(L, I+1)} \leftarrow \mathtt{count([X|L], I).}$

The bottom-up evaluation of $\mathcal{P}_5$ terminates yielding the set of atoms count([a,b,c],0), count([b,c],1), count([c],2), and count([],3). The query goal count([],L) can be used to retrieve the length L of list [a,b,c].[3] □

Basically, after having established that argument *count*[1] is limited, by analyzing the two cycles involving arguments *count*[1] and *count*[2], respectively, using Condition 3 of Definition 2 it is possible to detect that also argument *count*[2] is limited. Consequently, program $\mathcal{P}_5$ is bounded.

4. MAPPING-RESTRICTED PROGRAMS

In this section, we present a new technique for checking termination of the bottom-up evaluation of logic programs with function symbols. In particular, we denote the concept of *mapping* and use it to describe the form of atoms derivable during the bottom-up evaluation of the program. We next introduce the notion of *mapping-restricted* arguments and show that these arguments are limited. Moreover, the set of *mapping-restricted* arguments of a given program $\mathcal{P}$ can be computed by transforming the original program into a unary Datalog$_{nS}$ program $\mathcal{P}^r$, whose predicates correspond to the arguments of $\mathcal{P}$.

We consider normal positive programs since results obtained for such programs can be easily extended to general disjunctive programs with negation. We assume that i) the nesting level of complex terms is at most one, ii) there are no function symbols appearing in the extensional database, and iii) no constants appear in rules. There is no real restriction in such assumptions as every program and database could be rewritten into an equivalent program satisfying such conditions. For instance, a rule of the form $p(f(h(X))) \leftarrow q(X)$ could be rewritten into two rules: $p(f(X)) \leftarrow p'(X)$, $p'(h(X)) \leftarrow q(X)$. A rule of the form $p(X) \leftarrow b(f(X), X)$, whose extensional database is $b(f(0), 0)$ could be rewritten into the two rules $p(X) \leftarrow b'(Y, X)$ and $b'(f(X), X) \leftarrow b(X, X)$, with the extensional database $b(0, 0)$. Every rule of the form $p(a) \leftarrow body(X)$, where a is a constant, could be rewritten as $p(Y) \leftarrow body(X), p'(Y)$ with the addition of $p'(a)$ to the extensional database.

We start by introducing notations and terminology used hereafter.

Definition 3. Given a program $\mathcal{P}$, an *m-set* $U_\mathcal{P}$ is a set of pairs $p[i]/s$, called *mappings*, such that $p[i] \in arg(\mathcal{P})$ and $s \in F^*_\mathcal{P}$, where i) $F_\mathcal{P}$ denotes the alphabet consisting of all function symbols occurring in $\mathcal{P}$, and ii) $F^*_\mathcal{P}$ denotes the set of all strings plus the empty string denoted by ϵ. □

Intuitively, a pair $p[i]/s$ means that during the bottom-up evaluation of the program, considering all possible databases, argument $p[i]$ could take values whose structure, in terms of nesting of function symbols, is described by s. For instance, let $p(f(g(c_1)), c_2)$ be a ground atom derivable through the bottom-up evaluation of the input program, the mappings for its arguments are $p[1]/fg$ and $p[2]/\epsilon$. Let M be a model of $\mathcal{P} \cup D$, we denote by U_M the m-set derivable from all ground atoms occurring in M.

Given an m-set $U_\mathcal{P}$ and an atom $p(t_1, ..., t_n)$ occurring in $\mathcal{P}$, we say that an occurrence of a variable X in t_i has a *mapping* to a string s in $U_\mathcal{P}$ if $p[i]/s \in U_\mathcal{P} \wedge t_i = X$ or $p[i]/gs \in U_\mathcal{P} \wedge t_i = g(...X...)$. For instance, considering an atom $p(f(X))$ and $U_\mathcal{P} = \{p[1]/fg\}$, an occurrence of X in $f(X)$ has a mapping to a string g in $U_\mathcal{P}$.

Definition 4. Let $\mathcal{P}$ be a program and let $U_\mathcal{P}$ be an m-set. We say that $U_\mathcal{P}$ is *supported* if it can be built iteratively as follows:

1. $q[j]/\epsilon \in U_P$ for every argument $q[j] \in arg_b(\mathcal{P})$, and
2. for every rule $r \in \mathcal{P}$ and for every variable X in r, if all occurrences of variable X in the body of r have a mapping to a string s in $U_\mathcal{P}$, then all occurrences of X in the head of r also have a mapping to s in $U_\mathcal{P}$. □

Intuitively, for any supported $U_\mathcal{P}$ of $\mathcal{P}$, for every database D there exists a model M of $\mathcal{P} \cup D$ such that $U_M \subseteq U_\mathcal{P}$, that

[3]Notice that $\mathcal{P}_5$ has been written so as to count the number of elements in a list when evaluated in a bottom-up fashion, and therefore differs from the classical formulation relying on a top-down evaluation strategy. However, programs relying on a top-down evaluation strategy can be rewritten into programs whose bottom-up evaluation gives the same result.

is $U_{\mathcal{P}}$ is an overestimation of U_M. The number of supported m-sets for a given program $\mathcal{P}$ could be infinite, and there can be supported m-sets of an infinite size.

Given a program $\mathcal{P}$, a supported m-set $U_{\mathcal{P}}$ is *minimal* if there is no supported m-set $U'_{\mathcal{P}}$ such that $U'_{\mathcal{P}} \subset U_{\mathcal{P}}$. It is simple to note that every program $\mathcal{P}$ has a unique supported minimal m-set, called *minimum supported m-set*, denoted in the following by $U^*_{\mathcal{P}}$. The minimum supported m-set can be obtained as the intersection of all supported m-sets of $\mathcal{P}$ and, for any database D, it gives the best overestimation of U_{M^*}, where M^* is the minimum model of $\mathcal{P}_D$ (remember that we assume that D does not contain function symbols).

Example 6. Consider the program $\mathcal{P}_1$ of Example 1 and the database D composed by a fact $\mathtt{b(a)}$. The minimum model of $\mathcal{P}_1 \cup D$ and the corresponding m-set are

$$\begin{array}{lcl} M^* & = & \{\mathtt{b(a)}, \mathtt{p(a,f(a))}, \mathtt{p(f(a),a)}\}, \\ U_{M^*} & = & \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\epsilon\}. \end{array}$$

The minimum supported m-set of this program is

$$\begin{array}{rl} U^*_{\mathcal{P}_1} = & \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon,\ \mathtt{p}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\epsilon, \\ & \mathtt{q}[1]/\mathtt{f}, \mathtt{q}[2]/\mathtt{g}, \mathtt{q}[1]/\mathtt{ff}, \mathtt{q}[2]/\mathtt{gf}\}, \end{array}$$

that is a finite proper superset of U_{M^*}. □

Definition 5. Given a program $\mathcal{P}$, an argument $p[i] \in arg(\mathcal{P})$ is *mapping-restricted* (briefly *m-restricted*) iff $U^*_{\mathcal{P}}$ contains a finite set (possibly empty) of mappings $p[i]/s$. $MR(\mathcal{P})$ denotes the set of all *m-restricted arguments* of $\mathcal{P}$. A program $\mathcal{P}$ is *m-restricted* if $MR(\mathcal{P}) = arg(\mathcal{P})$, i.e. it admits a finite supported m-set. The set of m-restricted programs is denoted by $\mathcal{MR}$. □

From the discussion above it follows that each program whose minimum supported m-set is finite, has a finite minimum model for every database D. Moreover, it can be shown that every m-restricted argument is limited.

Theorem 1. *Every program $\mathcal{P}$ admitting a finite supported m-set is terminating.*

Proof. *(Sketch)* Straightforward from the observation that for any database D, let $M^* = \mathcal{MM}(\mathcal{P}_D)$, $U_{M^*} \subseteq U^*_{\mathcal{P}}$. □

Proposition 1. *Given program $\mathcal{P}$, every m-restricted argument is limited.*

Proof. *(Sketch)* Let $p[i]$ be a m-restricted argument of $\mathcal{P}$ and D be a database. Then, $U^*_{\mathcal{P}}$ contains a finite set of mappings $p[i]/s$. Since $U_{\mathcal{MM}(\mathcal{P}_D)} \subseteq U^*_{\mathcal{P}}$, then $p[i]$ is limited. □

In order to analyze the termination of a given program $\mathcal{P}$, we introduce a transformed program $\mathcal{P}^\tau$ having the following properties:

- the termination of $\mathcal{P}^\tau$ is decidable;
- $U^*_{\mathcal{P}^\tau} = U_{M^*}$, where $M^* = \mathcal{MM}(\mathcal{P}^\tau)$;
- There is a bijection h from $arg(\mathcal{P})$ to $arg(\mathcal{P}^\tau)$ s.t. $h(U^*_{\mathcal{P}}) = U^*_{\mathcal{P}^\tau}$, i.e. $p[i]/s \in U^*_{\mathcal{P}}$ iff $h(p[i])/s \in U^*_{\mathcal{P}^\tau}$.

Definition 6. Let $\mathcal{P}$ be a program. Then $\mathcal{P}^\tau$ denotes the program, where all predicates are unary, derived from $\mathcal{P}$ as follows:

- for every base predicate symbol b with arity n in $\mathcal{P}$, and for every $i \in \{1...n\}$, $\mathcal{P}^\tau$ contains a fact $b_i(0)$;
- for every rule $r = p(t_1, \ldots, t_n) \leftarrow body$ in $\mathcal{P}$, for every variable X occurring in $p(t_1, \ldots, t_n)$, and for every term t_i where X occurs, $\mathcal{P}^\tau$ contains a rule:

$$p_i(t_i^X) \leftarrow \bigwedge_{\substack{q(u_1,\ldots,u_k)\ in\ body \\ \wedge\ X\ occurs\ in\ u_j}} q_j(u_j^X)$$

where t^X denotes the following expression:

$$t^X = \begin{cases} X & \text{if } t = X \\ f(X) & \text{if } t = f(\ldots, X, \ldots). \end{cases}$$

We denote the set of facts defining predicates p_i, where p is a base predicate symbol, by $\hat{\mathcal{D}}^\tau$ and the set of remaining rules by $\hat{\mathcal{P}}^\tau$. □

Example 7. Consider the following program $\mathcal{P}_7$:

$$\begin{array}{lcl} \mathtt{p(X,X)} & \leftarrow & \mathtt{b(X)}. \\ \mathtt{q(f(X),f(X))} & \leftarrow & \mathtt{p(X,X)}. \\ \mathtt{p(f(X),X)} & \leftarrow & \mathtt{q(X,X)}. \end{array}$$

The minimum supported m-set of this program is

$$U^*_{\mathcal{P}_7} = \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\epsilon, \mathtt{q}[1]/\mathtt{f}, \mathtt{q}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{ff},\ \mathtt{p}[2]/\mathtt{f}\}.$$

The transformed unary program $\mathcal{P}_7^\tau$ is:

$$\begin{array}{lcl} \mathtt{b_1(0)}. & & \\ \mathtt{p_1(X)} & \leftarrow & \mathtt{b_1(X)}. \\ \mathtt{p_2(X)} & \leftarrow & \mathtt{b_1(X)}. \\ \mathtt{q_1(f(X))} & \leftarrow & \mathtt{p_1(X), p_2(X)}. \\ \mathtt{q_2(f(X))} & \leftarrow & \mathtt{p_1(X), p_2(X)}. \\ \mathtt{p_1(f(X))} & \leftarrow & \mathtt{q_1(X), q_2(X)}. \\ \mathtt{p_2(X)} & \leftarrow & \mathtt{q_1(X), q_2(X)}. \end{array}$$

The minimum model of $\mathcal{P}_7^\tau$ is $M^* = \{\mathtt{b_1(0)}, \mathtt{p_1(0)}, \mathtt{p_2(0)}, \mathtt{q_1(f(0))}, \mathtt{q_2(f(0))}, \mathtt{p_1(f(f(0)))}, \mathtt{p_2(f(0))}\}$, whereas $U_{M^*} = \{\mathtt{b_1}[1]/\epsilon, \mathtt{p_1}[1]/\epsilon, \mathtt{p_2}[1]/\epsilon, \mathtt{q_1}[1]/\mathtt{f}, \mathtt{q_2}[1]/\mathtt{f}, \mathtt{p_1}[1]/\mathtt{ff}, \mathtt{p_2}[1]/\mathtt{f}\}$. It is easy to see that $U_{M^*} = U^*_{\mathcal{P}_7^\tau}$. □

The following proposition states that for every program $\mathcal{P}$ the m-sets of $\mathcal{P}^\tau$ and $\mathcal{P}$ coincide (up to the bijection h) and are derivable from the minimum model of $\mathcal{P}^\tau$.

Proposition 2. *Let $\mathcal{P}$ be a program and $M^* = \mathcal{MM}(\mathcal{P}^\tau)$ be the minimum model of the transformed program $\mathcal{P}^\tau$, then $U^*_{\mathcal{P}^\tau} = U_{M^*}$ and there is a bijection h s.t. $h(U^*_{\mathcal{P}}) = U^*_{\mathcal{P}^\tau}$.*

Proof. The relation $U^*_{\mathcal{P}^\tau} = U_{M^*}$ is straightforward from the construction of $\mathcal{P}^\tau$. The existence of h follows from Definition 4 of supported m-set and the construction of $\mathcal{P}^\tau$: h is defined as $h(p[i]) = p_i[1]$ for every $p[i] \in arg(\mathcal{P})$. □

Moreover, since $\mathcal{P}^\tau$ is unary and uses only unary function symbols, it is a Datalog$_{nS}$ program. Consequently, the problem of checking the finiteness of its minimum model is decidable [8], implying that the problem of checking the finiteness of $U^*_{\mathcal{P}}$ for a given program $\mathcal{P}$ is decidable as well.

Let us now compare the presented technique with the argument-restricted technique, that generalizes ω-restricted, λ-restricted and finite domain techniques.

Intuitively, the argument-restricted (AR) technique derives the set of restricted arguments estimating the depth of complex terms that can be associated with an argument during the bottom-up evaluation. In particular, it considers the depth of terms in the body and in the head of rules, but it does not test the real possibility to activate a rule starting from a feasible database instance and does not distinguish different function symbols. The new $\mathcal{MR}$ technique overcomes these limitations by introducing the concept of supported m-set, which allows us to describe the form of argument values that are derivable during bottom-up evaluation of the program, starting from any database instance and use this information to simulate the evaluation process. Furthermore, to compute strings associated with head arguments, the current technique also checks that rules can be effectively activated. The following theorem states that the class of m-restricted programs generalizes the class of argument restricted programs.

THEOREM 2. $\mathcal{AR} \subsetneq \mathcal{MR}$

PROOF. Let $\mathcal{P}$ be a program. We denote by $\mathcal{P}_f$ the logic program obtained from $\mathcal{P}$ by replacing every function symbol occurring in $\mathcal{P}$ with the symbol f, admitting that a function symbol does not have fixed arity. Note that $\mathcal{P}$ is argument restricted iff $\mathcal{P}_f$ is argument restricted. Let ϕ be the argument ranking of both $\mathcal{P}$ and $\mathcal{P}_f$. We denote by s^k the string of length k of the form $s^k = f s^{k-1}$, where $s^0 = \epsilon$. Let $U_{\mathcal{P}_f} = \{p[i]/s^k \mid p[i] \in arg(\mathcal{P}) \wedge 0 \leq k \leq \phi(p[i])\}$. Note that such an m-set is a finite supported m-set for $\mathcal{P}_f$. Assume that $\exists p[i]/s \in U^*_{\mathcal{P}}$ such that $|s| > \phi(p[i])$, then, any supported m-set of $\mathcal{P}_f$ would contain a pair $p[i]/s'$ such that $|s'| > \phi(p[i])$, which contradicts the existence of $U_{\mathcal{P}_f}$. Then, $U^*_{\mathcal{P}}$ is finite and $\mathcal{P}$ is in $\mathcal{MR}$. In order to prove the strict inclusion, observe that program P_1 from Example 1 is in $\mathcal{MR}$ but not in $\mathcal{AR}$. □

The inclusion is proper even if the program contains only one function symbol. For instance, program $\mathcal{P}_7$ from Example 7 is in $\mathcal{MR}$ but not in $\mathcal{AR}$.

It is worth noting that although both AR and MR techniques are used to identify decidable subclasses of finitely ground programs, they can also be used to detect, for a given program $\mathcal{P}$, subsets of limited arguments of $\mathcal{P}$. The following proposition states that, given a program $\mathcal{P}$, the set $AR(\mathcal{P})$ of restricted arguments of $\mathcal{P}$ is a subset of the set of $MR(\mathcal{P})$ of m-restricted arguments of $\mathcal{P}$.

PROPOSITION 3. *For any program $\mathcal{P}$, $AR(\mathcal{P}) \subseteq MR(\mathcal{P})$*

PROOF. Straightforward from the proof of Theorem 2. □

Thus, the estimation of the set of limited arguments provided by the MR technique is better than the one provided by the AR technique. Detecting subsets of limited arguments is relevant even when the input program is not recognized as terminating by a given criterion, as in such cases it is possible to combine different techniques to detect the finiteness of the minimum model.

THEOREM 3. *$\mathcal{BP}$ and $\mathcal{MR}$ are not comparable*

PROOF. To prove the theorem it is sufficient to show that i) $\mathcal{BP} \not\subseteq \mathcal{MR}$, that is there is a program belonging to $\mathcal{BP}$ and not belonging to $\mathcal{MR}$, and ii) $\mathcal{MR} \not\subseteq \mathcal{BP}$, that is there is a program belonging to $\mathcal{MR}$ and not belonging to $\mathcal{BP}$. Indeed, P_1 is mapping-restricted, but not bounded, whereas program $\mathcal{P}_5$ is bounded, but not mapping-restricted. □

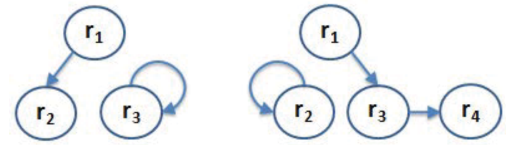

Figure 1: Activation graphs of $\mathcal{P}_8$ (left) and $\mathcal{P}_9$ (right).

5. FURTHER IMPROVEMENTS

As shown in the previous section, the MR criterion generalizes the AR criterion and is uncomparable with other termination criteria so far proposed, including bounded programs. However, the MR technique does not analyze the activation graph and the dependencies among arguments. These aspects are captured by the safe function proposed in [18] to define safe and Γ-acyclic programs and also used in the definition of bounded programs [14]. Consequently, it could be useful to combine the presented technique with other tools and techniques such as the safe function.

Thus, in this section we propose a variation of the safe function which will be used to extend the set of mapping-restricted arguments. While the original safe function proposed in [18] was used to establish termination of the input program, the safe function presented next takes as input a set of limited arguments and returns as output a possibly enlarged set of limited arguments.

DEFINITION 7. For any program $\mathcal{P}$, let A be a subset of $arg(\mathcal{P})$, the *safe function* $\Psi_{\mathcal{P}}(A)$ denotes the set of arguments $q[i]$ occurring in $\mathcal{P}$ such that for all rules $r \in \mathcal{P}$ where q appears in the head

1. r does not depend on a cycle of $\Omega(\mathcal{P})$, i.e. there is no path from some node belonging to a cycle to r, or
2. let t be the term corresponding to argument $q[i]$, for every variable X appearing in t, X also appears in some argument in $body^+(r)$ belonging to A. □

In order to understand the behavior of the safe function, consider the following example.

EXAMPLE 8. Consider the set $A = \{b[1]\}$ and the following program $\mathcal{P}_8$, where b is a base predicate:

$$\begin{array}{lll} \mathtt{r_1: p(X,X)} & \leftarrow & \mathtt{b(X).} \\ \mathtt{r_2: p(f(X),g(X))} & \leftarrow & \mathtt{p(X,X).} \\ \mathtt{r_3: q(f(X))} & \leftarrow & \mathtt{b(X),q(X).} \end{array}$$

The activation graph $\Omega(\mathcal{P}_8)$, shown in Figure 1 (left), has the unique cycle involving r_3. $\Psi_{\mathcal{P}_8}(A)$ contains all arguments of $\mathcal{P}_8$. In fact, $b[1], p[1]$ and $p[2]$ satisfy the first condition of Definition 7, whereas $q[1]$ satisfies the second one. □

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

PROPOSITION 4. *Let $\mathcal{P}$ be a program and let A be a set of limited arguments of $\mathcal{P}$, then all arguments in $\Psi_{\mathcal{P}}(A)$ are also limited.*

PROOF. Let $p[i] \in \Psi_{\mathcal{P}}(A)$, 1) if $p[i]$ satisfies Condition 1 of Definition 7, every rule r where $p[i]$ appears in $head(r)$ does not depend on a cycle in $\Omega(\mathcal{P})$, then r can be activated a finite number of times, thus, r cannot cause $p[i]$ to be non limited; 2) if $p[i]$ satisfies Condition 2 of Definition 7, then $p[i]$ is trivially limited. $\square$

PROPOSITION 5. *Let $\mathcal{P}$ be a program and let A be a set of limited arguments of $\mathcal{P}$, then the sequence $\Psi_{\mathcal{P}}(A), \Psi^2_{\mathcal{P}}(A) \dots \Psi^i_{\mathcal{P}}(A) \dots$ is monotonic and converges in a finite number of steps.*

PROOF. Straightforward from Definition 7 and from observation that $arg(\mathcal{P})$ is finite. $\square$

The next proposition ensures that, the application of the safe function $\Psi_{\mathcal{P}}$ to the set $MR(\mathcal{P})$ of mapping-restricted arguments of a given program $\mathcal{P}$ returns all arguments in $MR(\mathcal{P})$.

PROPOSITION 6. $MR(\mathcal{P}) \subseteq \Psi_{\mathcal{P}}(MR(\mathcal{P}))$*, for any logic program $\mathcal{P}$.*

PROOF. Let $q[i] \in MR(\mathcal{P})$, 1) if $q[i]$ is a base argument, $q[i]$ trivially satisfies Condition 1 of Definition 7; 2) if $q[i]$ is a derived argument, by definition of supported m-set, for every rule r with $q[i]$ in $head(r)$, for every variable X appearing in $q[i]$, X obviously appears in some argument in $body^+(r)$ of $MR(\mathcal{P})$, thus satisfying Condition 2 of Definition 7. $\square$

Propositions 4, 5 and 6 ensure that, once the set $MR(\mathcal{P})$ of limited arguments is computed, a (possibly proper) superset of $MR(\mathcal{P})$ of *limited* arguments can be computed iteratively applying the function $\Psi_{\mathcal{P}}$ starting from the set $MR(\mathcal{P})$ until the fixpoint is reached. Let us now define a new class of terminating programs based on the use of the safe function.

DEFINITION 8. Given a program $\mathcal{P}$, the set $MRS(\mathcal{P}) = \Psi^{\infty}_{\mathcal{P}}(MR(\mathcal{P}))$ denotes the set of *MR-safe arguments* of $\mathcal{P}$. A program $\mathcal{P}$ is said to be *MR-safe* if all arguments are MR-safe. The class of MR-safe programs is denoted by $\mathcal{MRS}$. $\square$

The use of the safe function allows to extend the $\mathcal{MR}$ class of terminating programs. Example 9 illustrates this result and shows that the application of the safe function to the set of mapping-restricted arguments gives better results w.r.t. the set of restricted arguments.

EXAMPLE 9. Consider the following program $\mathcal{P}_9$, where b is a base predicate:

$$\begin{array}{lcl} \mathtt{r_1}: \mathtt{s(f(X),g(X))} & \leftarrow & \mathtt{b(X)}. \\ \mathtt{r_2}: \mathtt{s(f(X),f(X))} & \leftarrow & \mathtt{s(X,X)}. \\ \mathtt{r_3}: \mathtt{q(f(X),h(Y))} & \leftarrow & \mathtt{s(X,g(Y))}. \\ \mathtt{r_4}: \mathtt{q(f(X),l(Y))} & \leftarrow & \mathtt{q(X,h(Y))}. \end{array}$$

The activation graph of $\mathcal{P}_9$ is reported in Figure 1 (right). The set of restricted arguments $AR(\mathcal{P}_9) = \{b[1]\}$, whereas the set of mapping-restricted arguments $MR(\mathcal{P}_9) = \{b[1], s[1], s[2], q[2]\}$. $\mathcal{P}_9$ is not in $\mathcal{MR}$ and even the set of limited arguments obtained by $\Psi^{\infty}_{\mathcal{P}_9}(AR(\mathcal{P})) = \{b[1], q[1], q[2]\} \neq arg(\mathcal{P}_9)$. However, $\Psi^{\infty}_{\mathcal{P}_9}(MR(\mathcal{P})) = arg(\mathcal{P}_9)$, then $\mathcal{P}_9$ is in $\mathcal{MRS}$. $\square$

The following theorems confirm that $\mathcal{MRS}$ strictly extends $\mathcal{MR}$ and show the soundness of the proposed approach.

THEOREM 4. $\mathcal{MR} \subsetneq \mathcal{MRS}$.

PROOF. Inclusion is straightforward from Proposition 5 and Proposition 6. To prove the strong inclusion note that program $\mathcal{P}_9$ is *MR-safe* but not in $\mathcal{MR}$. $\square$

THEOREM 5. *Every MR-safe program $\mathcal{P}$ is terminating.*

PROOF. Straightforward from Definition 7 and Propositions 4, 5 and 6. $\square$

The above results and comments suggest that the *MR* criterion is orthogonal with respect to other criteria. Therefore, it could be used to compute a base set of limited arguments which can be enlarged by next applying other tools such as the safe function. Similarly, starting from the set of limited arguments $MR(\mathcal{P})$, detected by the mapping-restricted technique, it is possible to apply the bounded argument technique to identify a possibly larger set of limited arguments. The combination of the two techniques can be immediately obtained by using in Definition 2, as initial set of limited arguments, the set $MR(\mathcal{P})$ instead of the set $AR(\mathcal{P})$. By denoting the resulting criterion with *MBP* and with $\mathcal{MBP}$ the class of *MR*-bounded programs, we have reason to believe that the class of *MR*-bounded programs generalizes both mapping-restricted and bounded programs.

6. COMPUTATIONAL COMPLEXITY

In this section we will study the computational complexity of the problem of computing the set $MR(\mathcal{P})$ of m-restricted arguments for a given program $\mathcal{P}$. This set gives us an underestimation of the set of limited arguments of $\mathcal{P}$, and, when it coincides with $arg(\mathcal{P})$, the program $\mathcal{P}$ is in $\mathcal{MR}$ and, consequently, is terminating.

From Proposition 2 it follows that the set $MR(\mathcal{P})$ can be computed by first transforming $\mathcal{P}$ into the Datalog$_{nS}$ program $\mathcal{P}^{\tau}$ and next by determining the limited arguments of $\mathcal{MM}(\mathcal{P}^{\tau})$.

Observe that by construction, all predicates of $\mathcal{P}^{\tau}$ are unary and functional, the number of facts in $\hat{\mathcal{D}}^{\tau}$ is equal to the number of base arguments of $\mathcal{P}$, and the number of function symbols in $\mathcal{P}$ and $\mathcal{P}^{\tau}$ coincide.

We consider two different cases on the base of whether the input program $\mathcal{P}$ contains only one or more than one function symbols, that is whether $\mathcal{P}^{\tau}$ is a Datalog$_{1S}$ or a Datalog$_{nS}$ program. Thus, in this section we present an algorithm computing the set of m-restricted arguments for a program $\mathcal{P}$ containing only one function symbol, i.e. $\mathcal{P}^{\tau}$ is a Datalog$_{1S}$ program.

We point out that, as the complexity of checking whether a Datalog$_{nS}$ program terminates may be higher than that of checking termination of a Datalog$_{1S}$ program, we could apply a less expensive (and less general) technique for checking program termintion, by considering a target program $\mathcal{P}^{\tau}$ where all function symbols are replaced by a single function symbol.

We start by introducing some definitions and results used hereafter to define the complexity of our algorithms.

Assuming that simple terms have constant size, the *size* of a program $\mathcal{P}$, denoted by $size(\mathcal{P})$, is bounded by $O(n \cdot p \cdot a_p \cdot a_f)$, where n is the number of rules in the program, p is the maximum number of predicates in the body of rules, a_p is the maximum arity of predicates in the program and a_f

is the maximum arity of function symbols in the program. The *size* of a database D, denoted by $size(D)$, is bounded by $O(d \cdot a_p)$, where d is the number of facts in the database. Finally, the *size* of $\mathcal{P}_D$ is $size(\mathcal{P}_D) = size(\mathcal{P}) + size(D)$.

The following lemma shows the relation between the size of a given program $\mathcal{P}$ and the size of the transformed program $\mathcal{P}^\tau$.

Lemma 1. *Given a program $\mathcal{P}$, $size(\mathcal{P}^\tau) = O(size(\mathcal{P})^2)$.*

Proof. By definition of $\mathcal{P}^\tau$, the number of facts in $\hat{\mathcal{D}}^\tau$ is equal to the number of base arguments of $\mathcal{P}$ and the number of rules in $\hat{\mathcal{P}}^\tau$ is at most $n \cdot a_p \cdot a_f$. Moreover, the maximum number of predicates in the body of rules in $\hat{\mathcal{P}}^\tau$ is $p \cdot a_p$ and the maximum arity of predicates and function symbols of $\hat{\mathcal{P}}^\tau$ is 1. Then, we have that $size(\hat{\mathcal{P}}^\tau) = O((n \cdot a_p \cdot a_f) \cdot (p \cdot a_p)) = O(size(\mathcal{P})^2)$ and $size(\hat{\mathcal{D}}^\tau) = O(size(\mathcal{P}))$, consequently $size(\mathcal{P}^\tau) = O(size(\mathcal{P})^2)$. $\square$

Programs with only one function symbol.

The main function of the algorithm computing the set of m-restricted arguments for programs containing only one function symbol is *ComputeMRestricted*. It takes as input a program $\mathcal{P}$ and returns as output the set of its m-restricted arguments.

Theorem 6. *For any program $\mathcal{P}$,*

$$MR(\mathcal{P}) = ComputeMRestricted(\mathcal{P}). \qquad \square$$

The function starts by computing the transformed program $\mathcal{P}^\tau$ (line 2). Next it computes the period (t_1, t_2) of the model of $\mathcal{P}^\tau$ (lines 3-15). In particular, since $\mathcal{P}^\tau$ is a unary Datalog$_{1S}$ program, the number of states of $\mathcal{P}^\tau$ is bounded by 2^{fsize}, where $fsize$ is the number of predicates in $\mathcal{P}^\tau$. Note that all arguments not limited in $M = \mathcal{MM}(\mathcal{P}^\tau)$ occur in predicates belonging to the states ranging from $M[t_1]$ to $M[t_2]$. Then, the function computes these states and deletes from the output set all the corresponding arguments (lines 16-21).

The computation of a state $M[t]$ of M is done by means of function *ComputeState*. It takes as input the transformation $\mathcal{P}^\tau$ of a program $\mathcal{P}$ and a ground term t and returns as output the state of the model of $\mathcal{P}^\tau$ evaluated in t. Computing a state $M[t]$ is performed by checking whether $\mathcal{P}^\tau \models p(t)$, for every predicate p occurring in $\mathcal{P}^\tau$. Function *Models* is in charge of checking whether $\mathcal{P}^\tau \models p(t)$ and it is a simplified version of the function proposed in [7], specific for unary programs with functional predicates only. This function is based on the following lemma: the notation $\mathcal{P}\{u\}$ denotes the program obtained by replacing every occurrence of the functional variable T in $\mathcal{P}$ with a ground functional term u.

Lemma 2. *[7] Let $\mathcal{P}_D$ be a Datalog$_{1S}$ program, $Q(t, \bar{a})$ a ground atomic query. Then, M is a model of $\mathcal{P}_D \cup \neg Q(t, \bar{a})$ iff the following conditions hold:*

- $D \subseteq M$ and $Q(t, \bar{a}) \notin M(t)$;
- $M(u) \cup M(u+1) \cup M^d \models \mathcal{P}\{u\}$ for any ground functional term u. $\square$

Let us start by presenting the complexity of function *Models*.

Function ComputeMRestricted

input : A positive normal program $\mathcal{P}$.
output: The set $MR(\mathcal{P})$.

1: $MR(\mathcal{P}) := arg(\mathcal{P})$;

`// Constructing` $\mathcal{P}^\tau$ `with only one function symbol.`
2: $\mathcal{P}^\tau :=$ `Compute`$\mathcal{P}^\tau(\mathcal{P})$;

`// Computing the period.`
3: $t_1 := 0$;
4: $t_2 := 1$;
5: **while** *true* **do**
6:     $M[t_2] :=$ `ComputeState`$(\mathcal{P}^\tau, t_2)$;
7:     **for** $t' := t_2$-1 **to** 0 **do**
8:         $M[t'] :=$ `ComputeState`$(\mathcal{P}^\tau, t')$;
9:         **if** $M[t'] = M[t_2]$ **then**
10:             $t_1 = t'$;
11:             **break** while;
12:         **end**
13:     **end**
14:     $t_2 := t_2 + 1$;
15: **end**

`// Finding m-restricted arguments.`
16: $t^* := t_1$;
17: **repeat**
18:     $M[t^*] :=$ `ComputeState`$(\mathcal{P}^\tau, t^*)$;
19:     $MR(\mathcal{P}) := MR(\mathcal{P}) - \{p[i] \mid p_i() \in M[t^*]\}$;
20:     $t^* := t^* + 1$;
21: **until** $t^* = t_2$;
22: **return** $MR(\mathcal{P})$;

Proposition 7. *Let $\mathcal{P}^\tau$ be the transformation of a program $\mathcal{P}$ with one function symbol and $Q(t)$ be a ground atomic query, Function* Models *performs in polynomial space w.r.t. $size(\mathcal{P}^\tau)$ and in polylogarithmic space w.r.t. $depth(t)$.*

Proof. The size of every state of the model M of $\mathcal{P}^\tau$ depends on the number of different ground atoms that can occur in one state; this number, denoted by $fsize$, is polynomial in $size(\mathcal{P}^\tau)$. In a similar way, a snapshot $M(t)$ can be encoded as a pair $(t, M[t])$, requiring polynomial space w.r.t. $size(\mathcal{P}^\tau)$ and logarithmic space w.r.t. *depth*(t) (recall that t is a number that can be encoded in binary).

Function *Models* is a non deterministic algorithm which implements Lemma 2 with some simplifications due to the syntactical form of $\mathcal{P}^\tau$ (all predicates are unary and functional). The application of Lemma 2 consists in verifying whether $\mathcal{P}^\tau \cup \neg Q(t)$ admits a model, that is whether $\mathcal{P}^\tau \models Q(t)$. Moreover, it first guesses the initial snapshot of the minimal Herbrand model of $\mathcal{P}^\tau$. Guessing a snapshot is obviously space polynomial in $size(\mathcal{P}^\tau)$ and logarithmic space in the depth of the given term. Verifying whether CurSnap $\models \hat{\mathcal{D}}^\tau \cup \neg Q(t)$ can be done in polynomial space w.r.t. $size(\mathcal{P}^\tau)$ since it simply needs to check whether $\hat{\mathcal{D}}^\tau \subseteq$ CurSnap $\wedge\ Q(t) \notin$ CurSnap. The cycle in the algorithm performs at most m iterations, which is exponential in $size(\mathcal{P}^\tau)$ but can be encoded in binary, requiring polynomial space. Moreover, the ground functional term v and the ground functional term t appearing in the query Q can be encoded in binary too, requiring a polynomial amount of memory for v in $size(\mathcal{P}^\tau)$ (because $v < m$) and a logarithmic amount of space for t in *depth*(t). Again, at each iteration, guessing the snapshot $M(v+1)$ is space polynomial in $size(\mathcal{P}^\tau)$ and logarithmic space in *depth*$(v+1)$, but since $v < m$, the space for storing $v+1$ is at most polynomial in $size(\mathcal{P}^\tau)$.

Function Models

input : The transformation $\mathcal{P}^\tau$ of a program $\mathcal{P}$.
A ground atomic query $Q(t)$.
output: Truth value of $\mathcal{P}^\tau \models Q(t)$.

$m := 2^{fsize}$;
$v := 0$;
CurSnap := Guess $M(0)$;
NextSnap := *null*;
satisf := CurSnap $\models \hat{\mathcal{D}}^\tau \cup \neg Q(t)$;
while *satisf and* $v < m$ **do**
  NextSnap := Guess $M(v+1)$;
  satisf := CurSnap $\cup$ NextSnap $\models \hat{\mathcal{P}}^\tau\{v\} \cup \neg Q(t)$;
  CurSnap := NextSnap;
  $v := v + 1$;
end
return ***not*** *satisf*;

Function ComputeState

input : The transformation $\mathcal{P}^\tau$ of a program $\mathcal{P}$.
A ground functional term t.
output: The state $M[t]$.

$M[t] := \emptyset$;
foreach *predicate* p **do**
  if `Models` $(\mathcal{P}^\tau, p(t))$ **then**
    $M[t] := M[t] \cup \{p()\}$;
  end
end
return $M[t]$;

Answering to CurSnap $\cup$ NextSnap $\models \hat{\mathcal{P}}^\tau\{v\} \cup \neg Q(t)$ can be done in polynomial space in $size(\mathcal{P}^\tau)$ and polylogarithmic space w.r.t. $depth(t)$. Finally, by Savitch's theorem, every non deterministic space polynomial algorithm can be rewritten into a deterministic one which performs in quadratically more space. $\square$

Since predicates in $\mathcal{P}^\tau$ are unary, the number of different atomic queries to be answered in Function *ComputeState* is polynomial in the size of the program. Then, computing a state has the same complexity of the Function *Models*.

Lemma 3. *Let $\mathcal{P}^\tau$ be the transformation of a program $\mathcal{P}$ with one function symbol. Computing the state of the model of $\mathcal{P}^\tau$ at the given time t requires polynomial space w.r.t. $size(\mathcal{P}^\tau)$ and polylogarithmic space w.r.t. $depth(t)$.*

Proof. Straightforward from previous proposition and considerations. $\square$

We can now present the main complexity result stating that *computing the set of m-restricted arguments of a program $\mathcal{P}$ with one function symbol is space polynomial w.r.t. $size(\mathcal{P})$.*

Theorem 7. *Given a program $\mathcal{P}$ containing only one function symbol, the complexity of computing $MR(\mathcal{P})$ is space polynomial w.r.t. $size(\mathcal{P})$.*

Proof. In Function *ComputeMRestricted*, `Compute`$\mathcal{P}^\tau(\mathcal{P})$ requires polynomial space in $size(\mathcal{P})$, from Lemma 1. The next phase of the algorithm computes the period of the model of $\mathcal{P}^\tau$ which is crucial for finding the m-restricted arguments of $\mathcal{P}$. The whole operation takes at most polynomial space w.r.t. $size(\mathcal{P}^\tau)$ since by Lemma 3 `ComputeState` requires polynomial space in $size(\mathcal{P}^\tau)$ and polylogarithmic space in $depth(t_2)$. Note that $depth(t_2)$ is at most exponential in the maximum size of a state of $\mathcal{P}^\tau$ (i.e. $fsize$), then "polylogarithmic space in $depth(t_2)$" means polynomial space w.r.t. $size(\mathcal{P}^\tau)$. Checking whether $M[t'] = M[t_2]$ requires obviously polynomial space in $size(\mathcal{P}^\tau)$. Finally, storing variables t_1, t_2, t' requires polynomial space in $size(\mathcal{P}^\tau)$. From Lemma 1, the whole phase requires polynomial space w.r.t. $size(\mathcal{P})$. The last phase computes the set $MR(\mathcal{P})$. From the previous considerations, the last phase requires polynomial space w.r.t. $size(\mathcal{P})$ too. $\square$

Corollary 1. *Given a program $\mathcal{P}$, the complexity of checking whether $\mathcal{P} \in \mathcal{MR}$ is space polynomial w.r.t. $size(\mathcal{P})$ if $\mathcal{P}$ contains at most one function symbol.*

Proof. Straightforward from Theorem 7. $\square$

Finally, for the class of MR-safe programs, the complexity of checking whether a given program $\mathcal{P} \in \mathcal{MRS}$ depends on the complexity of computing $MR(\mathcal{P})$ and the complexity of computing the fixpoint of safe function. The safe function can be applied at most $|arg(\mathcal{P})|$ times and needs the construction of the activation graph $\Omega(\mathcal{P})$.

The next proposition introduces a bound on the complexity of computing the activation graph of a program $\mathcal{P}$.

Proposition 8. *For any program $\mathcal{P}$, the activation graph of $\mathcal{P}$ can be constructed in time $O(size(\mathcal{P})^2)$.*

Proof. We denote by m the maximum size of the body of rules in $\mathcal{P}$, i.e. $m = p \cdot a_p \cdot a_f$. Given two rules $r_i, r_j \in \mathcal{P}$, checking whether r_i activates r_j can be done in time $O(m^2)$. In fact, checking whether two atoms unify can be done in time $O(m)$.

In order to check if r_i activates r_j it is sufficient to preliminarly substitute each variable in r_i with a unique dummy constant ξ, obtaining the new ground rule r'_i (time $O(m)$) and then checking *i)* for each atom A in the body of r_j, whether A unifies with $head(r'_i)$ (by computing the mgu θ of A and $head(r'_i)$ and its extension θ', obtained by assigning ξ to variables of r_j not appearing in θ (time $O(m)$)); *ii)* whether $head(r_j)\theta \notin body^+(r'_i)$ (time $O(m)$).

To construct $\Omega(\mathcal{P})$ we have to check, for every possible pair of rules r_i, r_j, whether r_i activates r_j. Since the activation graph has at most n^2 edges, the construction of $\Omega(\mathcal{P})$ can be done in $O(n^2 \cdot m^2)$, i.e. in $O(size(\mathcal{P})^2)$. $\square$

Corollary 2. *Given a program $\mathcal{P}$, the complexity of checking whether $\mathcal{P} \in \mathcal{MRS}$ is space polynomial w.r.t. $size(\mathcal{P})$ if $\mathcal{P}$ contains at most one function symbol.*

Proof. Straightforward from Theorem 7 and Proposition 8. $\square$

Programs with more than one function symbol.

So far we have considered programs with only one functions symbol. As said before, whenever programs contain more than one function symbol, we can perform a less accurate analysis, by replacing all function symbols with a unique symbol, even if they have different arities. The resulting unary program uses only one function symbol. This means that there could be mapping-restricted programs which are not recognized to be in $\mathcal{MR}$.

To enlarge the class of $\mathcal{MR}$ and $\mathcal{MRS}$ programs we can take into account the fact that programs may contain more than one function symbol rewriting them into a Datalog_{nS} program. The counterpart of this growth of expressivity is obviously a greater computational complexity.

Indeed, as the complexity of checking whether a Datalog_{nS} program terminates is exponential, we conjecture that for any program $\mathcal{P}$ with more than one function symbol, the complexity of both computing $MR(\mathcal{P})$ and checking whether $\mathcal{P} \in \mathcal{MR}$ is time exponential w.r.t. $size(\mathcal{P})$. Consequently, as the complexity of computing the safety function is polynomial, we can conjecture that the computational complexity of checking whether $\mathcal{P} \in \mathcal{MRS}$ is time exponential w.r.t. $size(\mathcal{P})$ as well.

7. CONCLUSIONS

In this paper we have presented a new technique for checking whether the bottom-up evaluation of logic programs with function symbols terminates. The technique is based on the definition of mappings from arguments to strings of function symbols representing possible values which could be taken by arguments during the bottom-up evaluation. Such mappings can be computed through the evaluation of a unary program $\mathcal{P}^r$ derived from the input program $\mathcal{P}$. As termination of $\mathcal{P}^r$ is decidable, its fixpoint evaluation gives us a set of limited arguments, here called m-restricted.

We have shown that this technique overcomes previous techniques, such as $\mathcal{AR}$ and is uncomparable with other techniques so far proposed. The technique can be easily combined with other techniques such as safety and the ones recently proposed [18, 14]. Moreover, it is possible to further improve our results by applying the recently proposed orthogonal rewriting based technique [15], that transforms a program into an adorned one. The idea is to apply the termination criteria to the adorned program rather than to the original one, (strictly) enlarging the class of programs recognized as finitely-ground.

Concerning the computational complexity, we point out that termination checking is a compile time operation and the high complexity results are with respect to the size of the program which, usually, is much smaller than the size of the database.

Finally, it would be interesting to investigate algorithms (and complexity results) for the computation of the mapping-restricted arguments of general logic programs. Furthermore, the combination of the presented technique with the bounded criterion is a relevant topic that also deserves a more formal and accurate analysis in terms of the class of logic programs recognized as terminating and in terms of computational complexity.

8. REFERENCES

[1] Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive asp with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10(4-6):497–512, 2010.

[2] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. *Theory and Practice of Logic Programming*, 9(2):213–238, 2009.

[3] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[4] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[5] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp: Theory and implementation. In *International Conference on Logic Programming*, pages 407–424, 2008.

[6] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Enhancing asp by functions: Decidable classes and implementation techniques. In *AAAI Conference on Artificial Intelligence*, 2010.

[7] Jan Chomicki. *A decidable class of logic programs with function symbols*. Manhattan, Kan: Kansas State University, Dept. of Computing and Information Sciences, 1990.

[8] Jan Chomicki and Tomasz Imieliński. Finite representation of infinite query answers. *ACM Trans. Database Syst.*, 18(2):181–223, June 1993.

[9] Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Testing for termination with monotonicity constraints. In *International Conference on Logic Programming*, pages 326–340, 2005.

[10] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 266–271, 2007.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *International Joint Conference and Symposium on Logic Programmin*, pages 1070–1080, 1988.

[12] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[13] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[14] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Checking logic program termination under bottom-up evaluation. In *International Joint Conference on Artificial Intelligence*, 2013.

[15] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *Theory and Practice of Logic Programming*, 2013 (to appear).

[16] Sergio Greco and Francesca Spezzano. Chase termination: A constraints rewriting approach. *Proceeding of the Very Large Data Base Conference*, 3(1):93–104, 2010.

[17] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *Proceeding of the Very Large Data Base Conference*, 4(11):1158–1168, 2011.

[18] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. On the termination of logic programs with function symbols. In *International Conference on Logic Programming (Technical Communications)*,

pages 323–333, 2012.

[19] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *International Conference on Logic Programming*, pages 489–493, 2009.

[20] Massimo Marchiori. Proving existential termination of normal logic programs. In *Algebraic Methodology and Software Technology*, pages 375–390, 1996.

[21] Jack Minker. On indefinite databases and the closed world assumption. In *International Conference on Automated Deduction*, pages 292–308, 1982.

[22] Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. Termination analysis of logic programs based on dependency graphs. In *International Symposium on Logic-based Program Synthesis and Transformation*, pages 8–22, 2007.

[23] Naoki Nishida and Germán Vidal. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):177–225, 2010.

[24] Enno Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1/2):73–116, 2001.

[25] Peter Schneider-Kamp, Jürgen Giesl, and Manh Thang Nguyen. The dependency triple framework for termination of logic programs. In *International Symposium on Logic-based Program Synthesis and Transformation*, pages 37–51, 2009.

[26] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), 2009.

[27] Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, and René Thiemann. Automated termination analysis for logic programs with cut. *Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010.

[28] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[29] Alexander Serebrenik and Danny De Schreye. On termination of meta-programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[30] Tommi Syrjänen. Omega-restricted logic programs. In *Logic Programming and Nonmonotonic Reasoning*, pages 267–279, 2001.

[31] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[32] Dean Voets and Danny De Schreye. Non-termination analysis of logic programs with integer arithmetics. *Theory and Practice of Logic Programming*, 11(4-5):521–536, 2011.