Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols

Citation for published version:
Calautti, M, Greco, S & Trubitsyna, I 2017, 'Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols', *ACM Transactions on Computational Logic*, vol. 18, no. 4, pp. 28:1-28:42.
https://doi.org/10.1145/3143804

Digital Object Identifier (DOI):
10.1145/3143804

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM Transactions on Computational Logic

Detecting decidable classes of finitely ground logic programs with function symbols[1]

Marco Calautti[2], School of Informatics, University of Edinburgh,
10 Crichton Street, Edinburgh, United Kingdom

Sergio Greco, DIMES, Università della Calabria,
Via P. Bucci, 87036 Rende (CS), Italy

Irina Trubitsyna, DIMES, Università della Calabria,
Via P. Bucci, 87036 Rende (CS), Italy

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. Starting from mappings, we identify *mapping-restricted* arguments, a subset of limited arguments, namely arguments which take values from finite domains. Mapping-restricted programs, consisting of rules whose arguments are all mapping-restricted, are terminating under the bottom-up computation as all its arguments take values from finite domains. We show that mappings can be computed by transforming the original program into a unary logic program: this allows us to establish decidability of checking if a program is mapping-restricted. We study the complexity of the presented approach and compare it with other techniques known in the literature. We also introduce an extension of the proposed approach which is able to recognize a wider class of logic programs. The presented technique provides a significant improvement as it can detect terminating programs not identified by other criteria proposed so far. Furthermore, it can be combined with other techniques to further enlarge the class of programs recognized as terminating under the bottom-up evaluation.

CCS Concepts: •**Computing methodologies** → *Logic programming and answer set programming;*

General Terms: Knowledge representation, Logic Programming, Temporal Logics.

Additional Key Words and Phrases: Answer set programming, function symbols, bottom-up evaluation, program termination, computational complexity, stable models

ACM Reference Format:
Marco Calautti, Sergio Greco and Irina Trubitsyna, 2016. Detecting decidable classes of finitely ground logic programs with function symbols. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 44 pages.
DOI: 0000001.0000001

1. INTRODUCTION

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 modelling 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.

[1]The paper refines and extends results presented at the 15th International Symposium on Principles and Practice of Declarative Programming [Calautti et al. 2013].
[2]Part of this work has been contributed by the author during his Ph.D. at DIMES, Università della Calabria.

Current techniques analyse how values are propagated among predicate arguments, to detect whether such arguments are *limited*, i.e., whether the sets of values that can be associated with these arguments are finite. However, these methods have limited capacity to analyse the propagation of function symbols during the evaluation and they often cannot recognize that recursive rules cannot be activated indefinitely. Consequently, current techniques are not able to identify as terminating even simple programs whose evaluation always terminates.

The next example shows a case in which the program is terminating but none of the current criteria is able to identify its termination.

Example 1.1. Consider the following program $\mathcal{P}$ where p, q and s denote derived predicates, whereas b is a base predicate defined by a set of ground facts :

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

The program is not recognized as terminating by current criteria, but it is easy to see that recursive rules r_2, r_3 and r_4 cannot be activated indefinitely. Consequently $\mathcal{P}$ has a finite minimum model for every possible database (i.e., set of ground facts defining b). □

The problem with the above program is that most of the current criteria analyse the structure of terms (e.g., depth or size) but do not take into account how different function symbols occur in such terms. Considering the example above, current criteria are not able to understand that rule r_2 cannot be fired by rule r_4, as in each p-atom derived by means of rule r_4 the terms in both arguments cannot be equal.

In this paper we present a new approach for checking termination of the bottom-up evaluation of logic programs with function symbols. The new technique analyses how function symbols are nested in complex terms in order to detect larger classes of terminating programs. The analysis above is carried out by constructing strings describing how function symbols are nested. To analyse strings of function symbols representing complex terms, the technique here proposed rewrites the input program $\mathcal{P}$ into a unary program $\mathcal{P}^u$ where both predicates and functions have a single argument. The termination of $\mathcal{P}^u$ guarantees that also $\mathcal{P}$ terminates.

Example 1.1 (cont.) The unary program derived from the program above is as follows:

$$\begin{array}{llllll} \rho_{1.1} : \mathtt{p}_1\mathtt{(X)} & \leftarrow & \mathtt{b}_1\mathtt{(X)}. & \rho_{1.2} : \mathtt{p}_2\mathtt{(X)} & \leftarrow & \mathtt{b}_1\mathtt{(X)}. \\ \rho_{2.1} : \mathtt{q}_1\mathtt{(f(X))} & \leftarrow & \mathtt{p}_1\mathtt{(X)}, \mathtt{p}_2\mathtt{(X)}. & \rho_{2.2} : \mathtt{q}_2\mathtt{(g(X))} & \leftarrow & \mathtt{p}_1\mathtt{(X)}, \mathtt{p}_2\mathtt{(X)}. \\ \rho_{3.1} : \mathtt{s}_1\mathtt{(X)} & \leftarrow & \mathtt{q}_1\mathtt{(f(X))}. & \rho_{3.2} : \mathtt{s}_2\mathtt{(X)} & \leftarrow & \mathtt{q}_2\mathtt{(X)}. \\ \rho_{4.1} : \mathtt{p}_1\mathtt{(f(X))} & \leftarrow & \mathtt{s}_1\mathtt{(X)}. & \rho_{4.2} : \mathtt{p}_2\mathtt{(f(X))} & \leftarrow & \mathtt{s}_2\mathtt{(X)}. \end{array}$$

where functions and predicates have a unique argument and subscripts on predicates denote an argument position. The termination of the unary program guarantees that the source program terminates as well. □

Other limitations of current approaches include the inability to recognize that particular constraints among arguments may hold. Knowledge of such constraints can help in finding that such arguments are limited. Only very recently, techniques were proposed to perform such kind of reasoning.

Thus the technique proposed also analyses the relationships that may exist among arguments of the same predicate. This allows the recognition of terminating classical logic programs (such as the one presented below), not recognized by several other techniques.

Example 1.2. Consider the following program:

$$\texttt{reverse}([\texttt{a},\texttt{b},\texttt{c}],[\,]).$$
$$\texttt{reverse}(\texttt{L}_1,[\texttt{X}|\texttt{L}_2]) \leftarrow \texttt{reverse}([\texttt{X}|\texttt{L}_1],\texttt{L}_2).$$

Although the size of terms in the second argument of predicate `reverse` increases, the restriction on the first argument guarantees that the program is terminating and, therefore, even the second argument is restricted. □

The peculiarity of the program reported in the previous example is that the corresponding unary program consists of two separate components defining $\texttt{reverse}_1$ and $\texttt{reverse}_2$, respectively. The specific structure of the two subprograms allow us to establish that the limitedness of $\texttt{reverse}_1$ implies that also $\texttt{reverse}_2$ is limited and, therefore, all arguments of the source program are finite.

It is worth noting that terminating criteria introduced for the (semi-)oblivious chase (e.g., those proposed in [Marnette 2009; Greco et al. 2015; Calautti et al. 2015]) cannot be applied. This is due to the fact that in the program above, the same function symbol occurs in more than one rule and in the body of rules.

Although we concentrate on positive programs, the technique here proposed can be immediately applied to general programs with negation and head disjunction.

Related work. A significant body of work has been done on termination of logic programs under top-down evaluation [Schreye and Decorte 1994; Voets and Schreye 2011; Marchiori 1996; Ohlebusch 2001; Codish et al. 2005; Serebrenik and De Schreye 2005; Nishida and Vidal 2010; Schneider-Kamp et al. 2009b; Schneider-Kamp et al. 2009a; Schneider-Kamp et al. 2010; Nguyen et al. 2007; Nguyen et al. 2011; Giesl et al. 2014; Bruynooghe et al. 2007; Bonatti 2004; Baselice et al. 2009].

In this paper, we consider positive logic programs with function symbols under the standard *minimum model semantics* and *bottom-up computation*. Therefore, as already discussed in [Calimeri et al. 2008; Greco et al. 2013a], all the above-mentioned excellent works cannot straightforwardly be applied to our setting. Our results can be trivially extended to disjunctive programs with negation under different *forward chaining-based semantics* (e.g., *minimal model semantics*, *stable model semantics* [Gelfond and Lifschitz 1988; 1991], *well-founded semantics* [Gelder et al. 1991], *minimal founded semantics* [Furfaro et al. 2004]), *preferences-based semantics* [Greco et al. 2007; Caroprese et al. 2007]) and semantics for programs with constraint atoms [Son et al. 2007].

In recent years we have witnessed an increasing interest in the problem of identifying logic programs with function symbols as terminating (i.e., 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 [Calimeri et al. 2008]. 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.

One of the first classes of terminating programs proposed in the literature is the one of ω*-restricted* programs, introduced in [Syrjanen 2001]. The main idea here is to construct a stratification of the predicate symbols, where a predicate p is on an higher level than a predicate q if p is defined in terms of q.

The aforementioned class has been later extended to the class of λ*-restricted* programs, presented in [Gebser et al. 2007]. Here, to each predicate p we assign a natural number $\lambda(p)$ called its *rank* which must be higher than the ranks assigned to predicates occurring in the body of rules defining p.

A more refined approach, which allows the analysis of programs on the argument level, is the *finite domain* technique, proposed in [Calimeri et al. 2008]. This criterion relies on the notion of the *argument graph*, describing the propagation of values among arguments of a logic program. The analysis of cycles in the graph is then used to understand whether values may indefinitely propagate among arguments.

Later on, [Lierler and Lifschitz 2009] introduced the class of *argument-restricted* programs, which strictly includes all the classes above. This class is based on the assignment of a natural number to arguments defining an upper bound on the depth of terms associated with arguments which can be derived during the bottom-up computation. One important aspect of this class is that it is (to the best of our knowledge) one of the most general ones where checking whether a program falls into the class remains tractable.

The class of *safe programs* [Greco et al. 2012; Calautti et al. 2015] is an extension of the class of finite domain programs. Its definition is based on the notions of *argument graph* and *activation graph*, where the latter is used to analyse how rules may trigger each other. The combined use of both the argument and activation graphs allows the identification of terminating programs not included in the class of finite domain programs. A further generalization of this class is the following.

Acyclic programs have been introduced by [Greco et al. 2012]. The definition of acyclic program relies on the notion of propagation graph, an evolution of the argument graph which also takes into account function symbols occurring in the atoms and how rules may trigger each other. Identification of terminating programs is then based on the analysis of cycles and paths in this graph.

A termination criterion more general than the ones presented thus far has been proposed by [Greco et al. 2013a], who introduced the class of *bounded programs*. For a better understanding of how terms are propagated, the technique uses two graphs: *(i)* the *labelled argument graph*, a directed graph whose edges are labelled with useful information on how terms are propagated from the body to the head of rules, and *(ii)* the activation graph. A relevant aspect that distinguishes this criterion from the aforementioned ones is the ability to identify groups of arguments having some correlation during the bottom-up evaluation of the program. This comes at the cost of an increase in complexity, as checking whether a program is bounded is exponential in the size of the labelled argument graph and, therefore, in the size of the input program.

The class of *rule-bounded* programs has been proposed in [Calautti et al. 2016b; 2015a] and relies on the use of linear inequalities to compute the sizes of terms and atoms. The criterion uses this information and checks if the size of the head atom of a (recursive) rule is always bounded by the size of a body atom. Since such a check requires solving a set of integer linear constraints, membership in the class can be established via a non-deterministic polynomial time procedure.

Finally, an orthogonal technique has been presented in [Greco et al. 2013b]. The technique can be used in conjunction with current termination criteria, by actually extending the class of programs recognized as terminating. The technique is based on the rewriting of the source program $\mathcal{P}$ into an "adorned" one $\mathcal{P}^\alpha$. The aim is then to apply termination criteria to $\mathcal{P}^\alpha$ rather than $\mathcal{P}$. The transformation is sound in that if $\mathcal{P}^\alpha$ satisfies a certain termination criterion, then the bottom-up evaluation of $\mathcal{P}$ terminates. However, the size of $\mathcal{P}^\alpha$ is, in the worst case, exponential in the size of $\mathcal{P}$.

The classes of argument-restricted, bounded, and rule-bounded programs, being the more general ones so far proposed, will be discussed in more details in Section 3.

A research area related to the one here investigated is *chase termination*. That is, the problem of defining sufficient conditions guaranteeing that the chase fixpoint algorithm terminates, independently of the input database[3] [Greco and Spezzano 2010;

[3]This problem is known to be undecidable in general [Gogacz and Marcinkowski 2014].

Greco et al. 2011; Greco et al. 2012; Grau et al. 2013; Onet 2013; Greco et al. 2015; Calautti et al. 2015; Calautti et al. 2015b; 2016a; Calì et al. 2013]. The chase algorithm is an important tool used to compute universal models and consistent answers to queries over possibly inconsistent databases (i.e., databases that do not satisfy a given set of integrity constraints). In this context, constraints are defined by specific logical formulae called *tuple generating dependencies* (TGDs) and *equality generating dependencies* (EGDs) [Fagin et al. 2005].

Two main classes of TGDs have been first shown to guarantee the termination of the chase: the class of *full TGDs* [Beeri and Vardi 1984], consisting of TGDs with no existentially quantified variables, and the class of *acyclic inclusion dependencies* [Casanova et al. 1984].

After nearly two decades, the chase received again a lot of attention from the database community, due to its usefulness in several database applications such as data exchange, data integration and consistent query answering.

The class of *weakly acyclic TGDs and EGDs* ($\mathcal{WA}$) has been defined by [Fagin et al. 2005], while at the same time [Deutsch and Tannen 2003] proposed the class of *constraints with stratified-witness*, which is a strict subset of $\mathcal{WA}$. Weak acyclicity strictly includes both sets of full TGDs and acyclic sets of inclusion dependencies.

A first proof of the undecidability of the termination of the chase appeared in [Deutsch et al. 2008]. However, this result only concerns the problem of checking whether the chase terminates over a set of constraints when an instance is given. The same paper proposed the class of *stratified* constraints, for which termination of the chase is guaranteed. Later on, several other researchers started the study of this important problem.

[Meier et al. 2009] proposed the criteria of *safety*, *c-stratification*, *safe restriction*, *inductive restriction* and the *$\mathcal{T}$-hierarchy*, whereas [Marnette 2009] introduced the class of *super-weakly acyclic* TGDs, for which the *termination of the (semi-)oblivious chase* is guaranteed.

Greco et al. proposed both sufficient criteria, such as *local stratification*, and rewriting algorithms consisting in rewriting the original set of constraints Σ into an "equivalent", but more informative set Σ^{α}. Termination of the chase over Σ^{α} guarantees termination of the chase over Σ [Greco and Spezzano 2010; Greco et al. 2011; 2015].

In [Grau et al. 2013], the *model-faithful acyclicity* (MFA) and *model-summarising acyclicity* (MSA) techniques have been proposed. The idea is to run the semi-oblivious (aka skolem) chase and then use some checks to identify a form of cyclic computation. In particular, when a "circular" skolem term is found (i.e., of the form $f(\ldots f(\ldots)\ldots)$), the computation of the chase is forced to stop.

The skolem chase termination problem is closely related to the termination of logic programs, as a set of TGDs can be rewritten into a logic program with function symbols. This is achieved by skolemizing constraints and replacing existentially quantified variables with complex terms and function symbols. However, chase termination techniques can be used only in very restricted cases of the logic programming setting. This follows from the fact that in skolemized constraints, function symbols may appear only in the head of rules and each function symbol may appear in exactly one rule.

Contributions. In this paper we present a new semi-dynamic approach for checking termination of the bottom-up evaluation of logic programs with function symbols.

We introduce the concept of *mapping* to describe the form of atoms derivable during the bottom-up evaluation of a program and use it to identify *mapping-restricted* arguments, a subset of limited arguments, namely those arguments which 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. Decidability results are achieved by reductions to satisfiability of Linear Temporal Logic (LTL) [Sistla and Clarke 1985] and Computation Tree Logic (CTL) [Meier et al. 2008] formulae, for programs with one and more than one function symbol, respectively.

We study the exact complexity of identifying mapping-restricted arguments by showing that the problem is *PSPACE−complete* in the presence of just one function symbol and *EXPTIME−complete* for general programs with more than one function symbol.

We discuss the relationship between the class of mapping-restricted programs and other criteria presented in the literature, showing that it generalizes some well-known techniques (e.g., argument restriction) and is not subsumed by any criterion proposed so far.

Finally, we also propose an improved version of the mapping-restricted criterion, which is able to identify a larger class of practical programs for which the bottom-up evaluation terminates. We show that the overall complexity does not increase in general.

The extended technique also takes into account the fact that by partitioning the rewritten (unary) program into subprograms it is possible to determine whether terms propagated via a particular argument influence the kind of terms propagated in other arguments of the same predicate symbol.

This paper refines and extends results presented in [Calautti et al. 2013]. In particular, Section 6.3 provides alternative proofs for the *PSPACE* and *EXPTIME* upper bounds, which are based on reductions to satisfiability of temporal logic formulae. Furthermore, we also show that such bounds are tight. Section 6 presents various extensions of our technique, which will allow the identification of interesting practical terminating programs, without compromising the complexity bounds.

Organization. The paper is organized as follows. Section 2 introduces preliminaries on logic programs with function symbols. Section 3 recalls termination criteria defined in the literature. Section 4 presents the new technique and discusses how it relates to other termination criteria. Complexity results are shown in Section 5. An extension of the technique, along with a discussion on its complexity is presented in Section 6. Conclusions and further improvements are discussed in Section 7.

2. LOGIC PROGRAMS

2.1. Syntax

We assume to have (pairwise disjoint) infinite sets of *variables*, *predicate symbols*, and *function symbols*. Each predicate and function symbol g is associated with an *arity*, denoted $ar(g)$, which is a non-negative integer. Function symbols of arity 0 are called *constants*. Given a predicate symbol p of arity n, the i-th *argument* of p is an expression of the form $p[i]$, for $1 \leq i \leq n$. A *term* is either a variable, or an expression of the form $f(t_1, \ldots, t_n)$, where f is a function symbol of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. A (positive) *rule* r is of the form:

$$A \leftarrow B_1, \ldots, B_m$$

where $m \geq 0$ and $A, B_1, \ldots, B_m$ are atoms[4]. The atom A is called the *head* of r and is denoted by $head(r)$. The conjunction $B_1, \ldots, B_m$ is called the *body* of r and is denoted

[4]In this paper we consider positive normal rules, that is, rules where the head consists of a single atom and the body does not contain negated atoms.

by $body(r)$. With a slight abuse of notation, we sometimes use $body(r)$ to also denote the *set* of atoms appearing in $body(r)$.

We assume that rules are *range restricted*, i.e., all variables of a rule appear in some body atom. A predicate symbol p is *defined by* a rule r if p appears in the head of r.

A term (resp. atom, rule, set of rules) is *ground* if no variables occur in it, whereas it is said to be *flat* if no function symbols with arity > 0 occur in it. A ground rule with an empty body is called *fact*. A rule having the same atom in the head and in the body is said to be *trivial*, otherwise, it is said to be *nontrivial*.

Predicate symbols are partitioned into *base* and *derived* predicates. Base predicate symbols are defined by ground facts only. For (a set of) atoms (resp. rules) S, we denote by $arg(S) = arg_b(S) \cup arg_d(S)$ the set of arguments occurring in S, where $arg_b(S)$ (resp. $arg_d(S)$) denotes the set of arguments of the base (resp. derived) predicate symbols in S.

A program is a finite set of rules. In particular, a finite set of flat facts defining base predicates is also called *database* and it is denoted by D, whereas a finite set of rules defining derived predicates is called *logic program* (or simply *program*) and, by convention, it is denoted by $\mathcal{P}$ (possibly with subscript). $\mathcal{P}_D = \mathcal{P} \cup D$ denotes a *complete program* containing rules in D and $\mathcal{P}$. Often, whenever there is no ambiguity, we shall use the term program to also denote complete programs.

Given a program $\mathcal{P}$, a predicate p depends on a predicate q (and write $p \prec q$) 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 $p \prec q$ and $q \prec p$. Two atoms are mutually recursive if their predicates are mutually recursive. Finally, a rule r is said to be recursive if $body(r)$ contains an atom that is mutually recursive with the head atom of r.

2.2. Semantics

Consider a program $\mathcal{P}$ and a database D. The *Herbrand universe* $H_{\mathcal{P}_D}$ of $\mathcal{P}_D$ is the possibly infinite set of ground terms which can be built using function symbols (and constants) occurring in $\mathcal{P}_D$. The *Herbrand base* $B_{\mathcal{P}_D}$ of $\mathcal{P}_D$ 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 (resp. atom) r' is a *ground instance* of a rule (resp. atom) r in $\mathcal{P}$ if r' can be obtained from r by substituting every variable in r with some ground terms in $H_{\mathcal{P}_D}$. We use $ground(r)$ to denote the set of all ground instances of r and $ground(\mathcal{P})$ to denote the set of all ground instances of the rules in $\mathcal{P}$, i.e., $ground(\mathcal{P}) = \cup_{r \in \mathcal{P}} ground(r)$.

An *interpretation* of $\mathcal{P}_D$ is any subset I of $B_{\mathcal{P}_D}$. The truth value of a ground atom A w.r.t. I, denoted $value_I(A)$, is $true$ if $A \in I$, otherwise it is $false$. A ground rule r is *satisfied* by I, denoted $I \models r$, if there is a (ground) atom A in $body(r)$ s.t. $value_I(A) = false$ or the (ground) atom A in the head of r is s.t. $value_I(A) = true$.

An interpretation of $\mathcal{P}_D$ is a *model* of $\mathcal{P}_D$ if it satisfies every ground rule in $ground(\mathcal{P}_D)$. Clearly, every model of $\mathcal{P}_D$ contains D. In this paper, we are interested in computing the set of facts which are implied by the rules according to a given semantics (e.g., minimal or stable model semantics).

A model M of $\mathcal{P}_D$ is minimal if no proper subset of M is a model of $\mathcal{P}_D$. Programs have a unique minimal model, also called minimum model, which will be denoted by $\mathcal{MM}(\mathcal{P}_D)$.

The minimum model of $\mathcal{P}_D$ can be computed via the *immediate consequence operator* of $\mathcal{P}_D$ which is a function $T_{\mathcal{P}_D} : 2^{\mathcal{B}_{\mathcal{P}_D}} \rightarrow 2^{\mathcal{B}_{\mathcal{P}_D}}$ defined as follows: for every interpretation I, $T_{\mathcal{P}_D}(I) = \{A \mid A \leftarrow B_1, \ldots, B_n \in ground(\mathcal{P}_D)$ and $\{B_1, \ldots, B_n\} \subseteq I\}$. The i-th iteration of $T_{\mathcal{P}_D}$ ($i \geq 1$) w.r.t. an interpretation I is defined as follows: $T^1_{\mathcal{P}_D}(I) = T_{\mathcal{P}_D}(I)$

and $T^i_{\mathcal{P}_D}(I) = T_{\mathcal{P}_D}(T^{i-1}_{\mathcal{P}_D}(I))$ for $i > 1$. The minimum model of $\mathcal{P}_D$ coincides with $T^\infty_{\mathcal{P}_D}(\emptyset)$. Trivial rules are always satisfied and can be deleted as they do not contribute to compute minimal models.

2.3. Terminating programs

Let $\mathcal{P}$ be a program, D a database and let $M = \mathcal{MM}(\mathcal{P}_D)$. An argument $p[i] \in arg(\mathcal{P})$ is said to be *limited* in M if the set $\{t_i \mid p(t_1, \ldots, t_n) \in \mathcal{MM}(\mathcal{P}_D)\}$ is finite. Furthermore, $p[i]$ is *limited* in $\mathcal{P}$ if for every database D, $p[i]$ is limited in M. Finally, a program $\mathcal{P}$ is *terminating* if all arguments in $arg(\mathcal{P})$ are limited in $\mathcal{P}$.

Equivalently, a program $\mathcal{P}$ is *terminating* if for every database D, the *bottom-up evaluation* of $\mathcal{P}$ starting from the database D terminates. That is, if there is a finite natural number n such that $T^n_{\mathcal{P}_D}(\emptyset) = T^\infty_{\mathcal{P}_D}(\emptyset)$.

In this paper, we study new conditions under which a (positive, normal) program $\mathcal{P}$ is terminating.

3. TERMINATION CRITERIA

As discussed in the introduction, the problem of identifying terminating programs is undecidable in general. Decidable criteria proposed in the literature allow us to determine the termination of programs by analysing their structure with the aim of finding how the propagation of complex terms among arguments of the program occurs. This analysis is usually carried out via graph-based tools describing the propagation of terms between arguments and/or rules, or via linear constraints simulating the propagation of values through the rules. Some of such techniques are meant to detect limited arguments of the program, so that if all arguments are detected, termination is proved. Other techniques apply an “all-or-nothing” approach, i.e., they can just conclude whether the program is terminating. In the following, we briefly describe some of the most general classes of terminating programs proposed in the literature.

Argument-restricted programs. [Lierler and Lifschitz 2009]
The basic idea of the argument-restricted technique is to find a ranking among the arguments of a given program. Intuitively, the rank of an argument represents an estimation of the depth of terms that may occur in it. In particular, let d be the rank assigned to a given argument $p[i]$. Then, d is an upper bound of the depth of terms that may occur in $p[i]$ during the program evaluation.

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

$$\begin{array}{l} dept(X, X) = 0, \\ dept(X, f(t_1, \ldots, t_m)) = 1 + \max\limits_{i \,:\, t_i \text{ contains } X} dept(X, t_i). \end{array}$$

The depth of a term t is defined as $dept(t) = max\{dept(X, t) \mid X \textit{ occurs in } t\}$; analogously, the depth of an atom is defined as $dept(p(t_1, \ldots, t_n)) = max\{dept(X, t_i) \mid i \in [1, n] \wedge X \textit{ occurs in } t_i\}$, whereas the depth of a conjunction of atoms is the maximum depth of the atoms occurring in the conjunction.

Definition 3.1. 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

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

Example 3.2. Consider the following program $\mathcal{P}$:

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

$\mathcal{P}$ has an argument ranking ϕ where $\phi(\mathtt{b[1]}) = 0$, $\phi(\mathtt{p[1]}) = 1$, $\phi(\mathtt{t[1]}) = 2$ and $\phi(\mathtt{s[1]}) = 1$. □

We use $AR(\mathcal{P})$ to denote the set of *restricted arguments* of $\mathcal{P}$, i.e., $AR(\mathcal{P}) = \{p[i] \mid p[i] \in arg(\mathcal{P}) \wedge \exists \phi \ s.t. \ \phi(p[i])$ is defined$\}$. A program $\mathcal{P}$ is said to be *argument-restricted* iff $AR(\mathcal{P}) = arg(\mathcal{P})$. For example, program $\mathcal{P}$ of Example 3.2 is argument-restricted[5]. The class of argument-restricted programs is denoted by $\mathcal{AR}$. Checking whether a program $\mathcal{P}$ is argument-restricted is feasible in polynomial time (cubic in the number of arguments) [Calautti et al. 2015].

Bounded programs. [Greco et al. 2013a]
The class of *bounded programs*, denoted with $\mathcal{BP}$ extends the class of argument-restricted programs, due to the ability to distinguish different function symbols and to the analysis of how terms propagated in the arguments of a program influence each other.

Example 3.3. Consider the following program $\mathcal{P}$ counting the number of elements in a list:

$$\begin{array}{l} r_1 : \mathtt{count(L, N)} \leftarrow \mathtt{input(L, N)}. \\ r_2 : \mathtt{count(T, N+1)} \leftarrow \mathtt{count([H|T], N)}. \end{array}$$

where the input list and initial counter are given via databases of the form $D = \{\mathtt{input([a, b, c, \ldots], 0).}\}$. It is easy to see that the program above is not argument-restricted, as there is no finite argument ranking for argument $\mathtt{count}[2]$. However, even though the terms propagated in argument $\mathtt{count}[2]$ "grow" at each application of rule r_2, the same application of rule r_2 causes the terms propagated in argument $\mathtt{count}[1]$ to "decrease" their nesting level. This implies that only a finite number of terms might be produced in argument $\mathtt{count}[2]$. The bounded technique is able to detect this behaviour and allows us to conclude that program $\mathcal{P}$ is terminating. □

The analysis of how values are propagated is carried out via the *labeled argument graph*. Edges of this graph are labeled with information about the function symbols propagated from rules bodies to head atoms. An analysis of the cycles on this graph is performed, where such labels, denoting the different occurrences of function symbols, are taken into account. For instance, consider the program $\{\mathtt{p(f(g(x))} \leftarrow \mathtt{p(g(X)).}\}$ which is not in $\mathcal{AR}$. As the depth of $p[1]$ increases, the inspection of the occurrences of function symbols in the cycles of the labeled argument graph allows to understand that the rule cannot fire itself. The limitation of this technique is in the fact that such kind of analysis is performed using the labeled argument graph as the only (static) tool. For instance, the simple program $\{\ \mathtt{p(X, X)} \leftarrow \mathtt{b(X).}\ \ \mathtt{p(f(X), X)} \leftarrow \mathtt{q(X, X).}\ \ \mathtt{q(X, Y)} \leftarrow \mathtt{p(X, Y).}\ \}$, identified as terminating by the technique proposed in this paper, does not belong to $\mathcal{BP}$.

Checking whether a program $\mathcal{P}$ is bounded is exponential in the size of the labelled argument graph [Greco et al. 2013a]. However, for a particular class of programs,

[5]It is easy to see that if $AR(\mathcal{P}) = arg(\mathcal{P})$, there exists an argument ranking ϕ which is a total function.

called *linear programs* in [Greco et al. 2013a], which includes different practical programs, checking membership in the class is tractable.

Rule-bounded programs. [Calautti et al. 2016b]
The class of rule-bounded programs, denoted with $\mathcal{RB}$, relies on the use of linear inequalities to measure terms and atoms' sizes and check if the size of the head of a (recursive) rule is always bounded by the size of a body atom.

Specifically, the *size of a term* t, denoted by $size(t)$, is defined as follows: if t is a variable X, its size is represented by an arithmetic variable x. Otherwise, t is a term of the form $f(t_1, \ldots, t_n)$ and its size is $size(t) = n + \sum_{i=1}^{n} size(t_i)$. The *size* of an atom $A = p(t_1, \ldots, t_n)$, denoted $size(A)$, is the n-vector $(size(t_1), \ldots, size(t_n))$.

An arithmetic variable x intuitively represents the possible sizes that the logical variable X can have during the bottom-up evaluation. The size of a term of the form $f(t_1, \ldots, t_n)$ is defined by summing up the size of its terms t_i's plus the arity n of f. For instance, consider rule r_2 of program $\mathcal{P}$ of Example 3.3. Let A and B be, respectively, the atom in the head and the atom in the body. Then, $size(A) = (t, 1 + n)$ and $size(B) = (2 + h + t, n)$.

Example 3.4. The following linear constraint is used to check if the size of the head of r_2 in Example 3.3 is always bounded by the size of the only body atom (rule r_1 is ignored in the analysis as it is harmless):

$$(\alpha_1, \alpha_2) \cdot (2 + h + t, n) \geq (\alpha_1, \alpha_2) \cdot (t, 1 + n)$$

Here (α_1, α_2), $(2 + h + t, n)$, and $(t, 1 + n)$ are vectors and $\cdot$ denotes the classical scalar product operator. In order for the program to be rule-bounded, positive integers α_1 and α_2 must exist such that the inequality is satisfied for all possible non-negative integer values assigned to h, t, and n. A possible solution is $\alpha_1 = 1$ and $\alpha_2 = 1$. Thus the program is rule-bounded. $\Box$

Membership in the class of rule-bounded programs can be established via a non-deterministic polynomial time procedure. Rule-bounded programs are incomparable with argument-restricted and bounded programs. An extension of the class of rule-bounded programs, which is still incomparable with argument-restricted and bounded programs, has been proposed in [Calautti et al. 2015a].

4. MAPPING-RESTRICTED PROGRAMS

In this section we assume, w.l.o.g., that rules are constant-free and the maximum nesting level of terms is 1. There is no real restriction in such an assumption as every program $\mathcal{P}$ 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 the rules $p(f(X)) \leftarrow p'(X)$, $p'(h(X)) \leftarrow q(X)$. A detailed description of how rules are rewritten is presented in Appendix C. We also point out that all the following results extend to arbitrary programs with disjunction in the head and negation in the body.

We start by introducing the notions of *mapping*, *m-set* and *derived m-set*, used to describe the form of atoms derivable during the bottom-up evaluation of a program (Subsection 4.1). Next, we define a set of mappings $\mathcal{U}^*_{\mathcal{P}}$ (called *minimum supported m-set*) providing a way to predict the form of atoms occurring in every model, regardless of the database (Subsection 4.2). The new class of terminating programs, called *mapping-restricted*, is presented. We show that the minimum supported m-set of a program $\mathcal{P}$ can also be constructed by transforming $\mathcal{P}$ into a unary program $\mathcal{P}^u$ so that $\mathcal{U}^*_{\mathcal{P}}$ can be obtained from the minimum model of $\mathcal{P}^u$ (Subsection 4.3). Finally, in Subsection 4.4 we compare our criterion with the ones presented in Section 3.

4.1. M-set

Definition 4.1 (*Mapping and m-set*). Let $\mathcal{P}$ be a program. A *mapping* is a pair $p[i]/s$ 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 Kleene closure on $F_{\mathcal{P}}$. An *m-set* of $\mathcal{P}$ is a set of mappings of $\mathcal{P}$. □

Intuitively, a mapping $p[i]/s$ is used to describe the fact that during the bottom-up evaluation of a program, considering all possible databases, argument $p[i]$ could take values whose structure, in terms of nesting of function symbols, is described by the string 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 $p[2]/\epsilon$, where the symbol ϵ denotes the empty string.

The next step is to characterize the models of a program $\mathcal{P}$ and database D by means of m-sets.

Definition 4.2 (*Derived m-set*). Let $\mathcal{P}$ be a program, D a database and M a model of $\mathcal{P}_D$. The *m-set derived from* M is the m-set of $\mathcal{P}$ defined as:

$$\mathcal{U}_M = \{p[i]/s \mid p(t_1, \ldots, t_n) \in M \ \wedge \ s \in str(t_i)\}$$

where $str(t)$ is the set of *strings induced by* t defined as:

$$str(t) = \begin{cases} \{\epsilon\} \text{ if } t \text{ is a simple term,} \\ \{f \cdot s \mid t = f(u_1, \ldots, u_k) \ \wedge \ s \in str(u_j) \ \wedge \ 1 \leq j \leq k\} \text{ otherwise} \end{cases}$$ □

Whenever $str(t) = \{s\}$ is a singleton, we also denote the element s by $str(t)$.

Example 4.3. Consider the database $D = \{\mathtt{b(a).}\}$ and the following program $\mathcal{P}$:

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

The minimum model M of $\mathcal{P}_D$ and the corresponding derived m-set are:

$$\begin{array}{lcllll} M & = & \{ \ \mathtt{b(a)}, & \mathtt{p(a, f(a))}, & \mathtt{p(f(a), a)} & \}, \\ \mathcal{U}_M & = & \{ \ \mathtt{b[1]}/\epsilon, & \mathtt{p[1]}/\epsilon, \mathtt{p[2]/f}, & \mathtt{p[1]/f}, \mathtt{p[2]}/\epsilon & \}. \end{array}$$ □

4.2. Minimum supported m-set

In the following, we show how to construct a particular m-set, called supported, that characterizes the whole set of minimum models of a program $\mathcal{P}$, regardless of the database. Intuitively, a supported m-set $\mathcal{U}$ is such that whenever an atom $p(t_1, \ldots, t_n)$ is derived during the computation of the minimum model of a program $\mathcal{P}$, then for each t_i and every $s \in str(t_i)$, the mapping $p[i]/s$ belongs to $\mathcal{U}$. That is, for each t_i there must exist strings that agree with t_i.

Definition 4.4 (*Supported m-set*). Let $\mathcal{P}$ be a program and let $\mathcal{U}$ be an m-set of $\mathcal{P}$. We say that $\mathcal{U}$ is *supported* if:

(1) $b[i]/\epsilon \in \mathcal{U}$ for every argument $b[i] \in arg_b(\mathcal{P})$, and

(2) for every rule $r\ :\ p(t_1,\ldots,t_m)\ \leftarrow\ \bigwedge_{k=1}^{n} p^k(v_1^k,\ldots,v_{m_k}^k)$ in $\mathcal{P}$ and variable X in $head(r)$, if there exists a string s such that for every body term v_j^k containing X, either $(v_j^k = X \wedge p^k[j]/s \in \mathcal{U})$ or $(v_j^k = g(\ldots,X,\ldots) \wedge p^k[j]/gs \in \mathcal{U})$, then:
— if $t_i = f(\ldots X \ldots)$, then $p[i]/fs \in \mathcal{U}$, whereas
— if $t_i = X$, then $p[i]/s \in \mathcal{U}$. □

As shown by the next example, a program might have multiple supported m-sets.

Example 4.5. Considering the program $\mathcal{P}$ of Example 4.3, the following m-sets are supported m-sets of $\mathcal{P}$:

$$\begin{aligned}\mathcal{U}^1 &= \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\epsilon, \mathtt{q}[1]/\mathtt{f}, \mathtt{q}[2]/\mathtt{g}, \mathtt{q}[1]/\mathtt{ff}, \mathtt{q}[2]/\mathtt{gf}\}\\ \mathcal{U}^2 &= \mathcal{U}^1 \cup \{\ \mathtt{p}[1]/\mathtt{g}, \mathtt{p}[2]/\mathtt{g}, \mathtt{q}[1]/\mathtt{fg}, \mathtt{q}[2]/\mathtt{gg}\}\end{aligned}$$ □

Although a program $\mathcal{P}$ might have multiple supported m-sets, it is easy to see that there is a unique supported m-set which is minimal (w.r.t. set inclusion). We refer to this m-set as the *unique minimal* (or simply *minimum*) supported m-set of $\mathcal{P}$, and denote it by $\mathcal{U}_{\mathcal{P}}^*$. In the case of Example 4.5, the minimum supported m-set coincides with $\mathcal{U}^1$.

Definition 4.4 implicitly introduces a monotonic operator allowing us to compute the minimum supported m-set of a program $\mathcal{P}$. The operator, denoted by $\Omega_{\mathcal{P}}$, is defined as follows:

— $\Omega_{\mathcal{P}}^0 = \{\ b[i]/\epsilon \mid b[i] \in arg_b(\mathcal{P})\ \}$;
— $\Omega_{\mathcal{P}}^{n+1} = \Omega_{\mathcal{P}}^n \cup \{\ p[i]/s \mid \exists r : p(t_1,\ldots,t_m) \leftarrow \bigwedge_k p^k(v_1^k,\ldots,v_{m_k}^k)\ \in\ \mathcal{P} \wedge \exists X$ in $t_i \wedge \exists s' \in F_{\mathcal{P}}^*$ s.t. $\forall v_j^k, v_l^h$ containing $X, s{=}\omega(X,t_i,v_j^k,s',\Omega_{\mathcal{P}}^n){=}\omega(X,t_i,v_l^h,s',\Omega_{\mathcal{P}}^n) \wedge s \neq\perp\}$,

where ω is the function returning either a string or the symbol $\perp$ defined as follows.

```
**Function** $\omega(X,t_i,v_j^k,s',\Omega_{\mathcal{P}}^n)$:
**begin**
    **if** $(v_j^k = X \wedge p^k[j]/s' \in \Omega_{\mathcal{P}}^n) \vee (v_j^k = g(\ldots X \ldots) \wedge p^k[j]/gs' \in \Omega_{\mathcal{P}}^n)$ **then**
        **if** $t_i = f(\ldots X \ldots)$ **then return** $fs'$;
        **if** $t_i = X$ **then return** $s'$;
    **else return** $\perp$;
**end**.
```

Clearly the operator $\Omega_{\mathcal{P}}$ is monotone, because $\Omega_{\mathcal{P}}^n \subseteq \Omega_{\mathcal{P}}^{n+1}$, and $\Omega_{\mathcal{P}}^\infty = \mathcal{U}_{\mathcal{P}}^*$. As we will show in the next example, the minimum supported m-set of a program is closely related to the model of the program itself.

Example 4.6. Consider the program $\mathcal{P}$ and database D of Example 4.3. The m-set derived from $M = \mathcal{MM}(\mathcal{P}_D)$ and the minimum supported m-set of $\mathcal{P}$ are:

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

The previous example shows that if M is the minimum model of our program, the program's minimum supported m-set contains the m-set derived from M. That is, $\mathcal{U}_{\mathcal{P}}^*$ acts as an over-approximation for $\mathcal{U}_M$. We will show in the following that given a program $\mathcal{P}$, the relation above holds regardless of the chosen database.

LEMMA 4.7. *Given a program $\mathcal{P}$, then $\mathcal{U}_M \subseteq \mathcal{U}_{\mathcal{P}}^*$ for every database D, where $M = \mathcal{MM}(\mathcal{P}_D)$.*

PROOF. To prove the lemma, we first show that $\mathcal{U}_M \subseteq \mathcal{U}_{M'}$ and then show that $\mathcal{U}_{M'} \subseteq \mathcal{U}^*_{\mathcal{P}}$, where $M' = \mathcal{MM}(\mathcal{P} \cup D^{\epsilon})$ and $D^{\epsilon} = \{b(\epsilon, \ldots, \epsilon) \mid b$ is a base predicate of $\mathcal{P}\}$.

($\mathcal{U}_M \subseteq \mathcal{U}_{M'}$). Let $p[i]/s$ be a mapping in $\mathcal{U}_M$. If $p[i] \in arg_b(\mathcal{P})$, then $s = \epsilon$, by definition of $\mathcal{U}_M$. From the definition of D^{ϵ} it immediately follows that $p[i]/\epsilon \in \mathcal{U}_{M'}$ as well. If $p[i] \in arg_d(\mathcal{P})$, then there must be a rule $r : p(t_1, \ldots, t_i, \ldots, t_n) \leftarrow body(r)$ in $\mathcal{P}$ and a substitution θ for all variables occurring in r, such that all atoms in $body(r)\theta$ appear in M (this follows from the definition of the minimum model of $\mathcal{P}$). Let θ' be the substitution obtained from θ by replacing all constants occurring in θ with ϵ. Since $\mathcal{P}$ is a *positive, normal* program, if the joins in $body(r)\theta$ are satisfied, the joins in $body(r)\theta'$ are satisfied as well. Thus, the atoms in $body(r)\theta'$ and $head(r)\theta'$ must occur in M'. Then, by construction of θ', we have $p[i]/s \in \mathcal{U}_{M'}$.

($\mathcal{U}_{M'} \subseteq \mathcal{U}^*_{\mathcal{P}}$). We show this part of the lemma by induction on the steps of the immediate consequence operator $T_{\mathcal{P} \cup D^{\epsilon}}$ needed to construct M'. In particular, let $M_n = T^n_{\mathcal{P} \cup D^{\epsilon}}(\emptyset)$ for $n > 0$. We show that for each $n > 0$, if $p[i]/s \in \mathcal{U}_{M_n}$, then $p[i]/s \in \mathcal{U}^*_{\mathcal{P}}$.

Base ($n = 1$): All mappings $b[i]/s \in \mathcal{U}_{M_1}$ are such that $b[i] \in arg_b(\mathcal{P})$. So, by definition of $\mathcal{U}_{M_1}$ and $\mathcal{U}^*_{\mathcal{P}}$, the claim follows.

Inductive case ($n > 1$): Let $p[i]/s \in \mathcal{U}_{M_n \setminus M_{n-1}}$. By definition of $T_{\mathcal{P} \cup D^{\epsilon}}$, there must be a rule $r : p(t_1, \ldots, t_i, \ldots, t_n) \leftarrow \bigwedge_k q^k(v^k_1, \ldots, v^k_{m_k})$ in $\mathcal{P}$ and a substitution θ for all variables occurring in r, such that all atoms in $body(r)\theta$ appear in M_{n-1} and $s \in str(t_i\theta)$. Let $r' = r\theta = p(t_1\theta, \ldots, t_i\theta, \ldots, t_n\theta) \leftarrow \bigwedge_k q^k(v^k_1\theta, \ldots, v^k_{m_k}\theta)$ be the ground rule obtained by applying θ to r. Recall that $s \in str(t_i\theta)$. Furthermore, by inductive hypothesis, since each atom in $body(r)\theta$ belongs to M_{n-1}, we have that for all k, j and every $s' \in str(v^k_j\theta)$, $q^k[j]/s' \in \mathcal{U}^*_{\mathcal{P}}$. Assume that s is the string in $str(t_i\theta)$ obtained by the concatenation of all function symbols in $t_i\theta$ up to a position in which some variable X occurs in t_i. Furthermore, note that θ replaces each variable with the same ground term. Let us now consider the set S of all the mappings $q^k[j]/s' \in \mathcal{U}_{M_{n-1}}$ such that $s' \in str(v^k_j\theta)$ is obtained by the concatenation of the function symbols in $v^k_j\theta$, up to a position where X appears. By inductive hypothesis, $S \subseteq \mathcal{U}^*_{\mathcal{P}}$. Furthermore, the mappings in S satisfy Item 2 of Definition 4.4, w.r.t. to r and the variable X. Thus, since s is obtained by concatenating function symbols in $t_i\theta$ up to some occurrence of X, from the definition of $\mathcal{U}^*_{\mathcal{P}}$, we conclude that $p[i]/s$ must belong to $\mathcal{U}^*_{\mathcal{P}}$. □

From the lemma above, it follows that each program whose minimum supported m-set is finite has a finite minimum model for every database D. With this lemma in hand, we are ready to define our class of terminating programs.

Definition 4.8 (*Mapping-restricted arguments*). An argument $p[i]$ occurring in a program $\mathcal{P}$ is said to be *mapping-restricted* (or simply *m-restricted*) iff the set $\{p[i]/s \mid p[i]/s \in \mathcal{U}^*_{\mathcal{P}}\}$ is finite. The set of all *m-restricted arguments* of $\mathcal{P}$ is denoted by $MR(\mathcal{P})$. □

Definition 4.9 (*Mapping-restricted programs*). A program $\mathcal{P}$ is said to be *m-restricted* iff $MR(\mathcal{P}) = arg(\mathcal{P})$, i.e., $\mathcal{U}^*_{\mathcal{P}}$ is finite. The set of all m-restricted programs is denoted by $\mathcal{MR}$. □

It is clear, from the discussion above, that every program belonging to $\mathcal{MR}$ has a finite minimum model, and thus is terminating. Furthermore, we can show that every m-restricted argument is also limited.

THEOREM 4.10. *Given a program $\mathcal{P}$ and an argument $p[i] \in arg(\mathcal{P})$, then*

(*1*) $p[i] \in MR(\mathcal{P})$ *implies that* $p[i]$ *is limited in* $\mathcal{P}$*;*
(*2*) $\mathcal{P} \in \mathcal{MR}$ *implies that* $\mathcal{P}$ *is terminating.*

PROOF. Let $\mathcal{U}^*_{\mathcal{P}}$ be the minimum supported m-set of $\mathcal{P}$.
(1) If $p[i] \in MR(\mathcal{P})$, then the set $S_{p[i]} = \{p[i]/s \mid p[i]/s \in \mathcal{U}^*_{\mathcal{P}}\}$ is finite. From Lemma 4.7, for every database D, if $M = \mathcal{MM}(\mathcal{P}_D)$, then $\mathcal{U}_M \subseteq \mathcal{U}^*_{\mathcal{P}}$. Then, the set $S'_{p[i]} = \{p[i]/s \mid p[i]/s \in \mathcal{U}_M\}$ is such that $S'_{p[i]} \subseteq S_{p[i]}$. Yet this implies that the set $\{t_i \mid p(t_1, \dots, t_n) \in M\}$ is finite as well. Then, $p[i]$ is limited in $\mathcal{P}$.
(2) This straightforwardly follows from Item (1), since when $\mathcal{P} \in \mathcal{MR}$, $MR(\mathcal{P}) = arg(\mathcal{P})$ and every argument of $\mathcal{P}$ is limited. □

4.3. Unary transformation

In this section, we discuss an alternative way of defining the minimum supported m-set of a program $\mathcal{P}$. In particular, we show how to encode the operator $\Omega_{\mathcal{P}}$ as the immediate consequence operator $T_{\mathcal{P}^u}$ of a unary program $\mathcal{P}^u$ derived from $\mathcal{P}$. Such a correspondence will help to better understand the notion of (minimum) supported m-set and, more importantly, it allows to reduce the termination problem of $\mathcal{P}$ to the the termination problem of $\mathcal{P}^u$. The latter problem is decidable and provides a convenient tool for discussing the complexity results presented in Section 5.

Definition 4.11 (*Unary transformation*). Let $\mathcal{P}$ be a program. We define $\mathcal{P}^u$ as the unary program consisting of the rules derived from $\mathcal{P}$ as follows:

(1) for every argument $b[i] \in arg_b(\mathcal{P})$, $\mathcal{P}^u$ contains a fact $b_i(\epsilon)$;
(2) for every rule $r = p(t_1, \dots, t_n) \leftarrow body(r)$ in $\mathcal{P}$, for every variable X occurring in $p(t_1, \dots, t_n)$, and for every term t_i where X occurs, we add the following rule in $\mathcal{P}^u$ if it is nontrivial:

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

where t^X is defined as follows:

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

Example 4.12. Consider the program $\mathcal{P}$ below (on the left side), where b is a base predicate, and the corresponding unary program $\mathcal{P}^u$ (on the right side), where rules ρ_i and ρ'_i are derived from $\mathtt{r_i}$ for $i \in \{1, 2, 3\}$.

$$\mathcal{P} = \begin{cases} \mathtt{r_1}: & \mathtt{p(X,X)} & \leftarrow \mathtt{b(X)}. \\ \mathtt{r_2}: & \mathtt{q(f(X),f(X))} & \leftarrow \mathtt{p(X,X)}. \\ \mathtt{r_3}: & \mathtt{p(f(X),X)} & \leftarrow \mathtt{q(X,X)}. \end{cases} \qquad \mathcal{P}^u = \begin{cases} \rho_0: & \mathtt{b_1}(\epsilon). & \\ \rho_1: & \mathtt{p_1(X)} & \leftarrow \mathtt{b_1(X)}. \\ \rho'_1: & \mathtt{p_2(X)} & \leftarrow \mathtt{b_1(X)}. \\ \rho_2: & \mathtt{q_1(f(X))} & \leftarrow \mathtt{p_1(X), p_2(X)}. \\ \rho'_2: & \mathtt{q_2(f(X))} & \leftarrow \mathtt{p_1(X), p_2(X)}. \\ \rho_3: & \mathtt{p_1(f(X))} & \leftarrow \mathtt{q_1(X), q_2(X)}. \\ \rho'_3: & \mathtt{p_2(X)} & \leftarrow \mathtt{q_1(X), q_2(X)}. \end{cases}$$

The minimum model M of $\mathcal{P}^u$ is:

$$M = \{\mathtt{b_1}(\epsilon), \mathtt{p_1}(\epsilon), \mathtt{p_2}(\epsilon), \mathtt{q_1}(\mathtt{f}(\epsilon)), \mathtt{q_2}(\mathtt{f}(\epsilon)), \mathtt{p_1}(\mathtt{f}(\mathtt{f}(\epsilon))), \mathtt{p_2}(\mathtt{f}(\epsilon))\}$$

Furthermore, the minimum supported m-set of $\mathcal{P}$ and the m-set derived from M are:

$$\begin{array}{rcl} \mathcal{U}^*_{\mathcal{P}} & = & \{\mathtt{b}[1]/\epsilon,\ \ \mathtt{p}[1]/\epsilon,\ \ \mathtt{p}[2]/\epsilon,\ \ \mathtt{q}[1]/\mathtt{f},\ \ \mathtt{q}[2]/\mathtt{f},\ \ \mathtt{p}[1]/\mathtt{ff},\ \ \mathtt{p}[2]/\mathtt{f}\} \\ \mathcal{U}_M & = & \{\mathtt{b}_1[1]/\epsilon,\ \mathtt{p}_1[1]/\epsilon,\ \mathtt{p}_2[1]/\epsilon,\ \mathtt{q}_1[1]/\mathtt{f},\ \mathtt{q}_2[1]/\mathtt{f},\ \mathtt{p}_1[1]/\mathtt{ff},\ \mathtt{p}_2[1]/\mathtt{f}\} \end{array}$$ □

The example above shows that there is a direct correspondence between the minimum supported m-set of $\mathcal{P}$ and the m-set derived from the minimum model of the unary program $\mathcal{P}^u$. In turn, this shows that there is a one-to-one correspondence between $\mathcal{U}^*_{\mathcal{P}}$ and the minimum model M of $\mathcal{P}^u$. So, it is clear that if it is possible to compute M, then also $\mathcal{U}^*_{\mathcal{P}}$ can be constructed.

In the following, we show that such a useful correspondence holds for every program.

PROPOSITION 4.13. *Let $\mathcal{P}$ be a program and let $M = \mathcal{MM}(\mathcal{P}^u)$ be the minimum model of the unary program $\mathcal{P}^u$. Then, there exists a bijection $h : arg(\mathcal{P}) \to arg(\mathcal{P}^u)$ such that $p[i]/s \in \mathcal{U}^*_{\mathcal{P}}$ iff $h(p[i])/s \in \mathcal{U}_M$.*

PROOF. First of all, please note that $p[i] \in arg(\mathcal{P})$ iff $p_i[1] \in arg(\mathcal{P}^u)$. Consider now the bijection h such that $h(p[i]) = p_i[1]$, for every $p[i] \in arg(\mathcal{P})$. We start by showing the "if" part of the proposition, by induction on the steps of the immediate consequence operator $T_{\mathcal{P}^u}$ used to construct M (recall that $M = T^{\infty}_{\mathcal{P}^u}(\emptyset)$).

Base ($n = 1$):
Let $M' = T^1_{\mathcal{P}^u}(\emptyset) = \{b_i(\epsilon) \mid b[i] \in arg_b(\mathcal{P})\}$. By construction of $\mathcal{U}_{M'}$, we have that $b_i[1]/\epsilon \in \mathcal{U}_{M'}$, for every $b[i] \in arg_b(\mathcal{P})$. Recalling that $b_i[1] = h(b[i])$ and that $b[i]/\epsilon \in \mathcal{U}^*_{\mathcal{P}}$, for each $b[i] \in arg_b(\mathcal{P})$, the claim follows.

Inductive case ($n > 1$):
Let $M^{n-1} = T^{n-1}_{\mathcal{P}^u}(\emptyset)$ and assume by inductive hypothesis that for each $p[i] \in arg(\mathcal{P})$ and string s, if $h(p[i])/s \in \mathcal{U}_{M^{n-1}}$, then $p[i]/s \in \mathcal{U}^*_{\mathcal{P}}$. Furthermore, let $M' = T^n_{\mathcal{P}^u}(\emptyset) = T_{\mathcal{P}^u}(M^{n-1})$. Now, let $p[i] \in arg(\mathcal{P})$ be such that $h(p[i])/s \in \mathcal{U}_{M'}$, with $s = f_1 \cdots f_m$. If $h(p[i])/s \in \mathcal{U}_{M^{n-1}}$ then the claim follows by inductive hypothesis. If $h(p[i])/s \in (\mathcal{U}_{M'} \setminus \mathcal{U}_{M^{n-1}})$, then, there must exists a ground atom A of the form $p_i(f_1(f_2(\dots f_m(\epsilon))))$ in $M' \setminus M^{n-1}$. Since $M' = T_{\mathcal{P}^u}(M^{n-1})$, from the definition of $T_{\mathcal{P}^u}$, there is a ground instance of a rule in $\mathcal{P}^u$ of the form $A \leftarrow B_1, \dots, B_k$, such that $k > 0$ and such that the atoms $\{B_1, \dots, B_k\} \subseteq M^{n-1}$. From the definition of $\mathcal{U}_{M^{n-1}}$, for each atom $q_j(t) \in \{B_1, \dots, B_k\}$, the mapping $h(q[j])/str(t) \in \mathcal{U}_{M^{n-1}}$, where $str(t)$ is the (only) string induced by the term t. Furthermore, by inductive hypothesis, if $h(q[j])/str(t) \in \mathcal{U}_{M^{n-1}}$, then $q[j]/str(t) \in \mathcal{U}^*_{\mathcal{P}}$. But, from the definition of supported m-set, there must also exist a mapping of the form $p[i]/s$ in $\mathcal{U}^*_{\mathcal{P}}$.

We show the "only if" part of the proposition via an induction on the steps of the operator $\Omega_{\mathcal{P}}$ needed to construct the minimum supported m-set $\mathcal{U}^*_{\mathcal{P}}$ of $\mathcal{P}$. In particular, we show that at each step n, if $p[i]/s \in \Omega^n_{\mathcal{P}}$, then $h(p[i])/s \in \mathcal{U}_M$.

For the step $n = 0$, the claim follows by definition of $\Omega_{\mathcal{P}}$ and $\mathcal{U}_M$. Now, let $p[i]/s \in \Omega^n_{\mathcal{P}} \setminus \Omega^{n-1}_{\mathcal{P}}$, with $n > 0$. By definition of $\Omega_{\mathcal{P}}$, there must be a rule in $\mathcal{P}$ of the form $r = p(t_1, \dots, t_m) \leftarrow \bigwedge_k p^k(v^k_1, \dots, v^k_{m_k})$, a variable X in t_i and a string s' such that $\forall v^k_j, v^h_l$ containing X, $s = \omega(X, t_i, v^k_j, s', \Omega^{n-1}_{\mathcal{P}}) = \omega(X, t_i, v^h_l, s', \Omega^{n-1}_{\mathcal{P}})$ and $s \neq \perp$. Looking at function ω, the condition:

$$(v^k_j = X \wedge p^k[j]/s' \in \Omega^{n-1}_{\mathcal{P}}) \vee (v^k_j = g(\dots X \dots) \wedge p^k[j]/gs' \in \Omega^{n-1}_{\mathcal{P}})$$

can be rewritten by inductive hypothesis to the following:

$$(v_j^k = X \wedge p_j^k[1]/s' \in \mathcal{U}_M) \vee (v_j^k = g(\ldots X \ldots) \wedge p_j^k[1]/gs' \in \mathcal{U}_M)$$

But this implies, by definition of $\mathcal{P}^u$ and by definition of the immediate consequence operator $T_{\mathcal{P}^u}$ that the string s returned by the function ω is indeed such that $p_i[1]/s \in \mathcal{U}_M$. That is, $h(p[i])/s \in \mathcal{U}_M$. □

Therefore, the minimum supported m-set of a program $\mathcal{P}$ can be obtained by computing the minimum model of $\mathcal{P}^u$, that is $\mathcal{U}_{\mathcal{P}}^* = \{p[i]/s \mid p_i[1]/s \in \mathcal{U}_{\mathcal{MM}(\mathcal{P}^u)}\} = \{p[i]/str(t) \mid p_i(t) \in \mathcal{MM}(\mathcal{P}^u)\}$.

4.4. Comparison with other approaches

In this section we study the relationship between the class of mapping-restricted programs and other classes of terminating programs proposed in the literature. We will focus on the classes of *argument-restricted*, *bounded* and *rule-bounded* programs introduced in Section 3. This study allows us to conclude that $\mathcal{MR}$ strictly includes several decidable classes defined in the literature (which are included in $\mathcal{AR}$). Furthermore, to the best of our knowledge, there is no decidable class already defined in the literature which includes $\mathcal{MR}$.

We start by showing that the class of argument-restricted programs is strictly contained in the class of mapping-restricted programs. Argument-restricted programs are identified by constructing a ranking over the arguments of a given program $\mathcal{P}$. Such rankings represent the maximum depth that terms may have during the bottom-up evaluation of $\mathcal{P}$, for any given database. The reason why such a class is contained in $\mathcal{MR}$ is that mapping-restricted programs are identified by finding the *structure* that terms may have inside the arguments. This is achieved by constructing, for each argument, strings of function symbols, which can represent both the depth of the term (via the string length) and the function symbols composing the term itself.

Example 4.14. Consider the following program $\mathcal{P}$:

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

where b is a base predicate symbol. It is easy to show that there is no argument ranking for the arguments $\mathtt{p}[1]$ and $\mathtt{p}[2]$. This is because, according to Definition 3.1, the presence of rule r_2 in the program will enforce the constraint $\phi(\mathtt{p[i]}) > \phi(\mathtt{p[i]})$, for $i = 1, 2$. On the other hand, we can construct the minimum supported m-set of $\mathcal{P}$ using the operator $\Omega_{\mathcal{P}}$. First construct $\Omega_{\mathcal{P}}^0 = \{\mathtt{b}[1]/\epsilon\}$. Then, by looking at rule r_1, construct $\Omega_{\mathcal{P}}^1 = \Omega_{\mathcal{P}}^0 \cup \{\mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\epsilon\}$. Then, using the previous two mappings and rule r_2, we obtain $\Omega_{\mathcal{P}}^2 = \Omega_{\mathcal{P}}^1 \cup \{\mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\mathtt{g}\}$. Since there are no pairs of mappings $\mathtt{p}[1]/\mathtt{s}_1$ and $\mathtt{p}[2]/\mathtt{s}_2$ in $\Omega_{\mathcal{P}}^2 \setminus \Omega_{\mathcal{P}}^1$ such that $s_1 = s_2$, we obtain the finite minimum supported m-set:

$$\mathcal{U}_{\mathcal{P}}^* = \Omega_{\mathcal{P}}^2 = \Omega_{\mathcal{P}}^\infty = \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\epsilon, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\mathtt{g}\}$$

Since $\mathcal{U}_{\mathcal{P}}^*$ is finite, we conclude that $\mathcal{P} \in \mathcal{MR}$. Observe also that the construction of $\mathcal{U}_{\mathcal{P}}^*$ terminates because the mapping-restricted technique is able to distinguish between the symbols $\mathtt{f}$ and $\mathtt{g}$. □

In the following we formally prove that argument-restricted programs are strictly contained in the class of mapping-restricted programs.

THEOREM 4.15. $\mathcal{AR} \subsetneq \mathcal{MR}$.

PROOF. Let $\mathcal{P}$ be an argument-restricted 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. Moreover, if $\mathcal{U}^*_{\mathcal{P}_f}$ is finite, then $\mathcal{U}^*_{\mathcal{P}}$ is finite too. Let ϕ be the argument ranking (which is a total function) of both $\mathcal{P}$ and $\mathcal{P}_f$. We denote by s^k the string of length k of the form $s^k = fs^{k-1}$, where $s^0 = \epsilon$. Let $\mathcal{U}_{\mathcal{P}_f} = \{p[i]/s^k \mid p[i] \in arg(\mathcal{P}) \wedge 0 \leq k \leq \phi(p[i])\}$. Note that such an m-set is a finite supported m-set for $\mathcal{P}_f$.

In fact, $\mathcal{U}_{\mathcal{P}_f}$ trivially satisfies Item 1 of Definition 4.4. Regarding Item 2, recall that $\mathcal{P}_f$ is argument-restricted, thus $\phi(A^0[i]) \geq \phi(B^0[j]) + dept(X, A^i) - dept(X, B^j)$ holds, where X, A^i and B^j are terms occuring in some rule r, according to Definition 3.1. Furthermore, $\mathcal{P}_f$ has only one function symbol. So, from Item 2, if X has a mapping to a string s^k of length k in $body(r)$, the mapping $A^0[i]/s^h$ must belong to $\mathcal{U}_{\mathcal{P}_f}$, with $h = k + dept(X, A^i) - dept(X, B^j)$. By construction of $\mathcal{U}_{\mathcal{P}_f}$, $k \leq \phi(B^0[j])$ which implies $\phi(A^0[i]) \geq k + dept(X, A^i) - dept(X, B^j)$ (i.e., $\phi(A^0[i]) \geq h$). Since $A^0[i]/s^l \in \mathcal{U}_{\mathcal{P}_f}$, for every $l \leq \phi(A^0[i])$ and since $\phi(A^0[i]) \geq h$, we conclude that $A^0[i]/s^h \in \mathcal{U}_{\mathcal{P}_f}$.

It is worth noting that $\mathcal{U}_{\mathcal{P}_f}$ is not necessarily minimum, since ϕ is not guaranteed to map arguments to the smallest possible integers. However, the minimum supported m-set $\mathcal{U}^*_{\mathcal{P}_f}$ is a subset of every supported m-set, then $\mathcal{U}^*_{\mathcal{P}_f} \subseteq \mathcal{U}_{\mathcal{P}_f}$. Thus, $\mathcal{U}^*_{\mathcal{P}_f}$ is finite, implying that $\mathcal{U}^*_{\mathcal{P}}$ is finite and $\mathcal{P} \in \mathcal{MR}$.

In order to prove the strict inclusion, observe that the program from Example 4.14 is in $\mathcal{MR}$ but not in $\mathcal{AR}$. □

Interestingly, $\mathcal{AR}$ is strictly contained in $\mathcal{MR}$ even if we consider programs with only one function symbol.

THEOREM 4.16. *$\mathcal{AR} \subsetneq \mathcal{MR}$ even for programs admitting only one function symbol.*

PROOF. The weak inclusion derives from Theorem 4.15. To prove strict containment, it is sufficient to show that there is a program $\mathcal{P}$ containing only one function symbol such that $\mathcal{P} \in \mathcal{MR}$ and $\mathcal{P} \notin \mathcal{AR}$. Such a program is as follows:

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

The program is not in $\mathcal{AR}$ as there is no argument ranking satisfying the conditions of Definition 3.1. Indeed, assume towards a contradiction that $\mathcal{P} \in \mathcal{AR}$. Then, by definition of argument ranking, from rules r_2, r_3 and r_4, the following inequalities must be satisfied:

$$\begin{array}{ll}
(r_2) & \phi(\mathtt{q[2]}) \geq \phi(\mathtt{p[2]}) \\
(r_3) & \phi(\mathtt{s[2]}) > \phi(\mathtt{q[2]}) \\
(r_4) & \phi(\mathtt{p[2]}) > \phi(\mathtt{s[2]})
\end{array}$$

However, combining the inequalities above, we obtain that $\phi(\mathtt{p[2]}) > \phi(\mathtt{s[2]}) > \phi(\mathtt{q[2]}) \geq \phi(\mathtt{p[2]})$. That is, $\phi(\mathtt{p[2]}) > \phi(\mathtt{p[2]})$.

On the other hand, the program is in $\mathcal{MR}$, as the minimum model of the unary program $\mathcal{P}^u$ below is finite:

$$\begin{array}{lll}
\rho_0 & : \mathtt{b}_1(\epsilon). & \\
\rho_{1.1} & : \mathtt{p}_1(\mathtt{X}) & \leftarrow \mathtt{b}_1(\mathtt{X}). \\
\rho_{1.2} & : \mathtt{p}_2(\mathtt{f}(\mathtt{X})) & \leftarrow \mathtt{b}_1(\mathtt{X}). \\
\rho_{2.1} & : \mathtt{q}_1(\mathtt{X}) & \leftarrow \mathtt{p}_1(\mathtt{X}), \mathtt{p}_2(\mathtt{f}(\mathtt{X})). \\
\rho_{2.2} & : \mathtt{q}_2(\mathtt{f}(\mathtt{X})) & \leftarrow \mathtt{p}_1(\mathtt{X}), \mathtt{p}_2(\mathtt{f}(\mathtt{X})). \\
\rho_{3.1} & : \mathtt{s}_1(\mathtt{X}) & \leftarrow \mathtt{q}_1(\mathtt{X}). \\
\rho_{3.2} & : \mathtt{s}_2(\mathtt{f}(\mathtt{Y})) & \leftarrow \mathtt{q}_2(\mathtt{Y}). \\
\rho_{4.1} & : \mathtt{p}_1(\mathtt{X}) & \leftarrow \mathtt{s}_1(\mathtt{X}). \\
\rho_{4.2} & : \mathtt{p}_2(\mathtt{f}(\mathtt{Y})) & \leftarrow \mathtt{s}_2(\mathtt{Y}).
\end{array}$$

□

It is worth noting that the proof above guarantees that the containment remains strict even if the only function symbol is unary.

Observe that both argument and mapping-restricted techniques are also able to detect, for a given program $\mathcal{P}$, a subset of the limited arguments of $\mathcal{P}$. As we will show later on in this paper, even if a program is not detected as terminating, the capacity to identify a large subset of its limited arguments is still an important task. This is due to the fact that one can devise techniques (see Section 6.2) able to eventually identify the whole set of arguments as limited, when an initial set of limited arguments is given.

The following corollary states that the set of limited arguments identified by our approach is always better than the one recognized by the argument-restricted technique.

COROLLARY 4.17. *For any program $\mathcal{P}$, $AR(\mathcal{P}) \subseteq MR(\mathcal{P})$.*

PROOF. Straightforward from the proof of Theorem 4.15. □

We conclude this section by showing that the class of mapping-restricted programs is incomparable to the classes of bounded and rule-bounded programs.

THEOREM 4.18. *$\mathcal{MR}$ is incomparable with $\mathcal{BP}$ and $\mathcal{RB}$.*

PROOF. To prove the theorem it is sufficient to show that there is a bounded (resp. rule-bounded) program not belonging to $\mathcal{MR}$, and viceversa. Indeed, the program from Example 1.1 is mapping-restricted, but neither bounded nor rule-bounded, whereas the following program:

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

is both bounded and rule-bounded but not mapping-restricted. □

Regarding the extension of the class of rule-bounded programs, proposed in [Calautti et al. 2015a], this class strictly extends $\mathcal{RB}$ and is incomparable with $\mathcal{AR}$. Consequently, it is incomparable with $\mathcal{MR}$ too.

It is worth noting that although we have restricted our attention to normal, positive programs the technique can be applied to an arbitrary program $\mathcal{P}$ with disjunction in the head and negation in the body by considering a positive program $st(\mathcal{P})$ derived from $\mathcal{P}$ as follows. Every rule $A_1 \vee \cdots \vee A_m \leftarrow body$ in $\mathcal{P}$ is replaced with m positive rules of the form $A_i \leftarrow body^+$ $(1 \leq i \leq m)$ where $body^+$ is obtained from $body$ by deleting all negative literals. For instance, considering general programs under stable model semantics, as already stated in [Greco et al. 2012], the minimal model of $st(\mathcal{P})$ contains every stable model of $\mathcal{P}$. Thus, if one can show that $st(\mathcal{P})$ is terminating, finiteness and computability of the stable models of $\mathcal{P}$ is guaranteed as well.

5. DECIDABILITY AND COMPLEXITY RESULTS

In this section, we show that the problem of checking whether (an argument of) a program $\mathcal{P}$ is mapping-restricted is decidable. In particular, we make use of the relation between our problem and the problem of checking the finiteness of the minimum model of $\mathcal{P}^u$. Thus, we focus our attention on the unary program $\mathcal{P}^u$. We prove that checking if $\mathcal{MM}(\mathcal{P}^u)$ is finite is *PSPACE−complete* (resp. *EXPTIME−complete*) when the original program contains at most one (resp. more than one) function symbol. The lower bounds are shown by standard reductions from the halting problem of a *PSPACE* (resp. *APSPACE*) Turing machine, whereas the upper bounds are obtained by reductions to satisfiability of Linear Temporal Logic (resp. Computation Tree Logic) formulae. We follow the assumption of Section 4, stating that rules are constant-free and the maximum nesting level of terms in $\mathcal{P}$ is 1.

With Proposition 4.13 in place, it is clear that the problem of deciding whether a program $\mathcal{P}$ (resp. an argument $p[i]$) is mapping-restricted is tantamount to the problem of deciding whether $\mathcal{MM}(\mathcal{P}^u)$ is finite (resp. $p_i[1]$ is limited in $\mathcal{MM}(\mathcal{P}^u)$).

In order to properly discuss the complexity of the aforementioned problems, we start by defining the notion of size of a program and then show the relationship between the size of a given program $\mathcal{P}$ and its corresponding unary program $\mathcal{P}^u$.

The *size* of a program $\mathcal{P}$ (resp. database D), denoted by $size(\mathcal{P})$ (resp. $size(D)$), is defined as the total number of occurrences in $\mathcal{P}$ (resp. D) of predicate symbols, function symbols, variables and constants. The *size* of a program along with its database $\mathcal{P}_D$ is $size(\mathcal{P}_D) = size(\mathcal{P}) + size(D)$.

LEMMA 5.1. *Given a program $\mathcal{P}$, the unary program $\mathcal{P}^u$ is computable in polynomial time w.r.t. $size(\mathcal{P})$.*

PROOF. We start by showing that the number of facts and rules in $\mathcal{P}^u$ is polynomial w.r.t. $size(\mathcal{P})$. Then we will show that constructing each fact (resp. rule) of $\mathcal{P}^u$ is feasible in polynomial time. These two results will immediately give the desired complexity result. In the following we denote by D^u the set of facts occurring in $\mathcal{P}^u$ and by $\mathcal{T}^u$ the set of remaining rules, i.e., $\mathcal{P}^u \setminus D^u$. We also denote by n_r the number of rules in $\mathcal{P}$, b_p the maximum number of predicates in the body of rules of $\mathcal{P}$, a_p the maximum arity of predicates in $\mathcal{P}$ and a_f the maximum arity of function symbols in $\mathcal{P}$.

By definition of $\mathcal{P}^u$, the number of facts in D^u is equal to the number of base arguments of $\mathcal{P}$, and the maximum arity of predicates in D^u is one. Regarding $\mathcal{T}^u$, the rules in it are at most $n_r \cdot a_p \cdot a_f$. Furthermore, the maximum number of predicates in the body of rules in $\mathcal{T}^u$ is $b_p \cdot a_p$ and the maximum arity of predicates and function symbols of $\mathcal{T}^u$ is one. Since the nesting level for function symbols is assumed to be at most one, we have that the total number of symbols (predicates, functions and variables) occurring in $\mathcal{T}^u$ is bounded by $O((n_r \cdot a_p \cdot a_f) \cdot (b_p \cdot a_p))$. Regarding the construction of $\mathcal{P}^u$, computing one fact of D^u is a constant time operation since facts of D^u are all of the form $b_i(\epsilon)$, where b_i corresponds to a base argument $b[i]$ of $\mathcal{P}$. Given a rule r in $\mathcal{P}$, the construction of one rule of $\mathcal{T}^u$ starting from r just requires a linear scanning of r from left to right by picking a variable occurrence in the head and then extracting from $body(r)$ the needed atoms containing the same variable. □

It is clear from Proposition 4.13 and Lemma 5.1 above that the problem of checking whether a program (resp. an argument $p[i]$ of) $\mathcal{P}$ belongs to $\mathcal{MR}$ (resp. $MR(\mathcal{P})$) and the problem of checking whether $\mathcal{MM}(\mathcal{P}^u)$ is finite (resp. $p_i[1]$ is limited in $\mathcal{MM}(\mathcal{P}^u)$) are not only equivalent but also *polynomial time* equivalent. This means that complexity results for the latter problem can immediately be applied to the former (and vice versa), when such results involve complexity classes closed under polynomial time many-one reductions.

In the following, we show the exact complexity of the aforementioned problems. In particular, we show how the complexity changes on the basis of whether $\mathcal{P}$ contains at most one or more than one function symbol.

THEOREM 5.2. *Let $\mathcal{P}$ be a program with at most one function symbol. Checking whether the minimum model of $\mathcal{P}^u$ is finite is in PSPACE.*

PROOF. Recall that $\mathcal{P}^u$ is a positive normal logic program having only unary predicate symbols, and rules contain exactly one variable, say X. In the following we denote by D^u the set of facts occurring in $\mathcal{P}^u$ and by $\mathcal{T}^u$ the set of remaining rules, i.e., $\mathcal{P}^u \setminus D^u$. By our assumption, at most one function symbol may occur in $\mathcal{T}^u$. To show the desired upper-bound, we reduce the problem of checking whether the minimum model of a program of the form of $\mathcal{P}^u$ is finite to the satisfiability problem of Linear Temporal Logic (LTL) formulae. See the appendix for more details on LTL.

Let $M = \mathcal{MM}(\mathcal{P}^u)$ be the minimum model of $\mathcal{P}^u$. We now construct a LTL formula φ, starting from $\mathcal{P}^u$, such that φ is satisfiable iff M is finite. Predicates and terms in $\mathcal{P}^u$ are modelled by means of propositions and LTL temporal operators, respectively. In particular, atoms in $\mathcal{P}^u$ of the form $p_i(X)$ are modelled by means of propositions p_i and the depth of terms of atoms in M will be modelled by the states of the linear-time structure.

Firstly, for every rule $r_i : p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ occurring in $\mathcal{T}^u$, we construct

$$\varphi_i : \; G.(\zeta_1.p_1 \wedge \ldots \wedge \zeta_n.p_n \Rightarrow \zeta_0.p_0)$$

where $p_0, \ldots, p_n$ are propositions and ζ_k, where $k \in [1..n]$, coincides with the "next" temporal operator N if $t_k = f(X)$, or the empty string otherwise. Recalling that every rule r_i has just one variable X, the propositions p_k in φ_i, corresponding to atoms of the form $p_k(X)$ in r_i, belong to the same state, say s (corresponding to the depth of X). The propositions corresponding to atoms of the form $p_k(f(X))$ belong to the state next to s. The LTL formula φ_i encodes the fact that for every state ("always" temporal operator G), the specified implication, modelling the behaviour of rule r_i, must hold.

Next we construct the LTL formula φ:

$$\bigwedge_{b \in pred(D^u)} b \;\wedge\; \bigwedge_{r_i \in \mathcal{T}^u} \varphi_i \;\wedge\; F.G.(\bigwedge_{p \in pred(\mathcal{T}^u)} \neg p)$$

The formula above states that (i) at the initial state, all database propositions must be true; (ii) all formulae simulating $\mathcal{T}^u$ must be satisfied in every state, and (iii) there must exist a state ("sometime" temporal operator F) from which all propositions remain false. This last check is also known as the "persistence check".

Intuitively, the first two conditions guarantee that the associations "proposition/predicate - state/depth", simulating the atoms in $\mathcal{MM}(\mathcal{P}^u)$, must satisfy the depth relation established by rules in $\mathcal{P}^u$. The latter one ensures the finiteness of M.

We now formally prove that M is finite iff the formula φ is satisfiable.
($\Rightarrow$) Assume that M is finite. Please note that the only ground terms occurring in M are of the form $f^i(\epsilon)$, for some $i \geq 0$. Then, we construct a linear structure $K = (S, \psi)$ for the formula φ as follows. Let S be an infinite sequence of states $S = s_0, s_1, \ldots$ and let ψ be the mapping function such that:

$$\psi(s_i) = \{p \mid p(f^i(\epsilon)) \in M\}, \text{ for } i = 0, 1, \ldots$$

It is not difficult to see that the above linear structure satisfies φ. Indeed, the depth of all base predicates is 0, thus the corresponding propositions are true in the initial

state, and only in it. Since all ground rules from $\mathcal{T}^u$ are satisfied by M, the depth relations among atoms described by the corresponding LTL formulae φ_i are satisfied. Finally, if d is the maximum depth of terms occurring in M, there exists a state s_{d+1} starting from which all propositions are false.
($\Leftarrow$) Assume now that M is infinite. Then, since base atoms are flat and the maximum nesting level of function symbols in $\mathcal{T}^u$ is 1, for every $i = 0, 1, \ldots$ there always exists at least one ground atom of the form $p(f^i(\epsilon))$ in M. That is, whatever ground term $f^i(\epsilon)$, of an arbitrary depth i we may consider, there must be an atom in M containing such a term. From this consideration, it follows that every linear structure $K = (S, \psi)$ satisfying φ must be such that for every state s in S, ψ is such that $\psi(s) \neq \emptyset$. Thus, there are infinitely many states where at least one proposition is true. Indeed, $\psi(s_0)$ must contain (at least) all propositions corresponding to base predicates in order to satisfy the first condition of φ. Moreover, it can be easily shown by induction, that $\psi(s_d)$ must contain all propositions p such that atoms of the form $p(t)$, where t has depth d, occur in M. This is because formulae φ_i, for all rules $r_i \in \mathcal{T}^u$ used to derive these atoms, must be satisfied.

But this would immediately imply that K cannot satisfy the formula $F.G.(\neg q)$, occurring in φ. Thus, φ is not satisfiable. □

From the proof of theorem above, we can easily show that an almost identical reduction can be adopted for the problem of checking whether a given argument $q[1]$ of $\mathcal{P}^u$ (recall that $\mathcal{P}^u$ is unary) is limited in $\mathcal{MM}(\mathcal{P}^u)$. That is, we only need to change the last condition of formula φ to the following one:

$$F.G.(\neg q)$$

where only the proposition q must remain false.

In the following, we show that the previous upper bounds are also tight.

THEOREM 5.3. *Let $\mathcal{P}$ be a program with at most one function symbol. Checking whether the minimum model of $\mathcal{P}^u$ is finite is PSPACE$-$hard.*

PROOF. To show the hardness result, we show that it holds for the subclass of programs $\mathcal{P}$ with only unary predicate symbols and with at most one function symbol,which is unary. In particular, note that for such kinds of programs, the corresponding unary program $\mathcal{P}^u = \mathcal{T}^u \cup D^u$, where D^u denotes the set of facts and $\mathcal{T}^u$ all other rules in $\mathcal{P}^u$, is such that $\mathcal{T}^u$ coincides with $\mathcal{P}$ (modulo predicate symbols renaming). Thus, from Proposition 4.13, the problem of checking whether a program $\mathcal{P}$ is m-restricted is tantamount to the problem of checking whether the minimum model of $\mathcal{P} \cup D$ is finite, where the database D contains, for every base predicate symbol b of $\mathcal{P}$, an atom of the form $b(\epsilon)$.

We prove the hardness of the problem above by reduction of the acceptance problem of a *PSPACE* turing machine $TM = (\{0, 1, \perp\}, Q, q_0, q_a, \delta)$, where $\{0, 1, \perp\}$ denotes the tape alphabet ($\perp$ is the blank symbol), Q the set of states, where $q_0, q_a \in Q$ denote the initial and the accepting states respectively, and δ is the transition function.

Let $I = a_1, \ldots, a_m$ be an input string for TM and let $n = poly(m)$ be the (polynomially many) number of tape cells used by TM during its computation. We assume w.l.o.g. that such n cells are extended with other $n + 1$ cells, where the first n cells of TM's tape will be used as worktape, which will also contain the input string I at the initial configuration. The remaining $n + 1$ cells will be used to store a counter. This counter will be incremented whenever the machine TM changes its configuration. Since TM is a *PSPACE* machine, it is easy to encode such a counter in binary with $n + 1$ bits, in order to count the exponentially many configurations. We also assume that the head of TM never attempts to move outside these $2n+1$ cells. Finally, we assume that TM accepts input I if the head is on the first tape cell and the state is the accepting state q_a. When TM halts in a non accepting state or the counter reaches its maximum value (i.e., its first bit is set to 1), then the machine rejects input I.

Let us now define the unary predicate symbols needed to construct a positive normal program simulating the computation of TM with input I.

To each i-th tape cell of TM, where $1 \leq i \leq 2n+1$, we associate the atom $\mathtt{cell}_\mathtt{i}^{\alpha}(\mathtt{X})$, stating that at configuration $\mathtt{X}$, the i-th cell contains the symbol $\alpha \in \{\mathtt{0}, \mathtt{1}, \perp\}$. Furthermore, we use atoms of the form $\mathtt{count}_\mathtt{1}^{\alpha_1}(\mathtt{X}), \ldots, \mathtt{count}_\mathtt{n+1}^{\alpha_{n+1}}(\mathtt{X})$ as shorthands for atoms $\mathtt{cell}_\mathtt{n+1}^{\alpha_1}(\mathtt{X}), \ldots, \mathtt{cell}_\mathtt{2n+1}^{\alpha_{n+1}}(\mathtt{X})$. These atoms will represent the counter's bits. Finally, we use the atom $\mathtt{head_i}(\mathtt{X})$ to state that the head of TM is on the i-th cell at configuration X and the atom $\mathtt{state}^\mathtt{q}(\mathtt{X})$ to denote that the state at configuration $\mathtt{X}$ is q, for every state $q \in Q$.

We now show how the simulation of TM with input I works. We first introduce a rule which will construct the initial configuration of TM:

$$\bigwedge_{\mathtt{i}\in[\mathtt{1..m}]} \mathtt{cell}_\mathtt{i}^{\mathtt{a_i}}(\mathtt{X}) \wedge \bigwedge_{\mathtt{i}\in[\mathtt{m+1..n}]} \mathtt{cell}_\mathtt{i}^{\perp}(\mathtt{X}) \wedge \bigwedge_{\mathtt{i}\in[\mathtt{1..n+1}]} \mathtt{count}_\mathtt{i}^{\mathtt{0}}(\mathtt{X}) \wedge \mathtt{state}^{\mathtt{q_0}}(\mathtt{X}) \wedge \mathtt{head_1}(\mathtt{X}) \leftarrow \mathtt{b}(\mathtt{X}).$$

where $\mathtt{b}$ is a base predicate symbol whose database extension will contain the initial configuration. The first m cells are filled with the symbols of input I, the remaining $n-m$ cells are filled with blanks and the counter's bits are initially set to zeroes. Furthermore, at the initial configuration, we force the current state to be q_0 and the head's position to be 1.

Now, for every transition rule $(q, \alpha) \rightarrow (q', \alpha', \mathbf{r}) \in \delta$, where $\mathbf{r}$ denotes the fact that the head of TM moves to the right, we first construct the following rule, for every tape cell position $1 \leq i \leq 2n+1$:

$$\begin{array}{l}\mathtt{cell}_\mathtt{i}^{\alpha'}(\mathtt{next}(\mathtt{X})) \ \wedge\ \mathtt{head_{i+1}}(\mathtt{next}(\mathtt{X})) \ \wedge\ \mathtt{state}^{\mathtt{q}'}(\mathtt{next}(\mathtt{X})) \leftarrow \\ \qquad\qquad\qquad \mathtt{state}^\mathtt{q}(\mathtt{X}),\ \mathtt{head_i}(\mathtt{X}),\ \mathtt{cell}_\mathtt{i}^{\alpha}(\mathtt{X}),\ \mathtt{count}_\mathtt{1}^{\mathtt{0}}(\mathtt{X}).\end{array}$$

Such a rule simply states that if at the given configuration $\mathtt{X}$, the state is q, the head is on the i-th cell, which contains symbol α and the counter has not reached its maximum value yet (its most significant bit is zero), then at the next configuration $\mathtt{next}(\mathtt{X})$ (here $\mathtt{next}$ is a unary function symbol), the i-th cell will contain the symbol α', the head will be on cell $i+1$ and the new state will be q'.
Next, in order to propagate all the unmodified cell values at configuration $\mathtt{X}$ to configuration $\mathtt{next}(\mathtt{X})$, we also construct the rules of the form below:

$$\mathtt{cell}_\mathtt{j}^{\mathtt{c}}(\mathtt{next}(\mathtt{X})) \leftarrow \mathtt{state}^\mathtt{q}(\mathtt{X}), \mathtt{cell}_\mathtt{i}^{\alpha}(\mathtt{X}), \mathtt{head_i}(\mathtt{X}), \mathtt{count}_\mathtt{1}^{\mathtt{0}}(\mathtt{X}), \mathtt{cell}_\mathtt{j}^{\mathtt{c}}(\mathtt{X}).$$

for every other tape cell $1 \leq j \leq 2n+1$ such that $j \neq i$ and for every value $c \in \{0, 1, \perp\}$. These rules essentially state that if a transition from configuration $\mathtt{X}$ to configuration $\mathtt{next}(\mathtt{X})$ has to be performed, where the content of the i-th cell is modified, then all other cells containing some value c at configuration $\mathtt{X}$, (i.e., $\mathtt{cell}_\mathtt{j}^{\mathtt{c}}(\mathtt{X})$) must still contain such a value at configuration $\mathtt{next}(\mathtt{X})$.

We will construct similar rules for the transition rules whose head moves to the left.

Finally, we encode the acceptance of TM with the following rule:

$$\mathtt{accept}(\mathtt{X}) \leftarrow \mathtt{state}^{\mathtt{q_a}}(\mathtt{X}),\ \mathtt{head_1}(\mathtt{X}),\ \mathtt{count}_\mathtt{1}^{\mathtt{0}}(\mathtt{X}).$$

Note that by construction, every predicate symbol, except for $\mathtt{b}$, occurs in the head of some of the rules above. We thus define $\mathtt{b}$ as base predicate symbol and all the other predicate symbols as derived. It is not difficult to see that the machine TM accepts its input I if and only if the model of such a program with database $D = \{\mathtt{b}(\epsilon)\}$ contains an *accept*-atom.

Now, we introduce the following additional rule:

$$\mathtt{accept(next(X))} \leftarrow \mathtt{accept(X)}.$$

With the addition of this rule, we can immediately show that *TM* accepts input I iff the minimum model of the obtained program and the database $D = \{\mathtt{b}(\epsilon)\}$ is finite. This will show that checking finiteness of $\mathcal{MM}(\mathcal{P} \cup D)$ is *coPSPACE−hard*. It is well-known that *coPSPACE = PSPACE*, and from the definition of *coPSPACE−hardness*, we show that checking finiteness of $\mathcal{MM}(\mathcal{P} \cup D)$ is *PSPACE−hard*. □

It is worth mentioning that the same reduction used in the previous theorem easily shows that the complexity of checking whether an argument (namely, $accept[1]$) is m-restricted is *PSPACE−hard* as well.

From the above theorems and considerations, Lemma 5.1 and Proposition 4.13, we finally obtain the main complexity result for programs with at most one function symbol.

COROLLARY 5.4. *Given a program $\mathcal{P}$ with at most one function symbol, the complexity of checking whether (an argument of) $\mathcal{P}$ is m-restricted is PSPACE−complete.* □

We now consider programs with an arbitrary number of function symbols. In particular, we show that for programs of this form, the computational complexity increases.

THEOREM 5.5. *Let $\mathcal{P}$ be a program with more than one function symbol. Checking whether the minimum model of $\mathcal{P}^u$ is finite is in EXPTIME.*

PROOF. We denote with D^u the set of facts occurring in $\mathcal{P}^u$ and $\mathcal{T}^u$ denotes the set of remaining rules, i.e., $\mathcal{P}^u \setminus D^u$. The difference between this proof and the one of Theorem 5.2 is that we reduce our problem to the satisfiability problem of Computation Tree Logic (CTL) formulae. Please see the Appendix for details on syntax and semantics of CTL.

Let $M = \mathcal{MM}(\mathcal{P}^u)$ be the minimum model of $\mathcal{P}^u$. We construct a CTL formula φ starting from $\mathcal{P}^u$, such that φ is satisfiable iff M is finite. Predicate symbols and function symbols in $\mathcal{P}^u$ are modelled by means of propositions of the target CTL formula, whereas terms in $\mathcal{P}^u$ are modelled by the use of CTL temporal operators. Furthermore, ground terms inside the atoms of M and their subterm relation are modelled by the states and transition relation of the underlying transition system, respectively.

Let $Prop = \Pi \cup F$ be the set of CTL propositions such that $p \in \Pi$ iff p is a predicate symbol in $\mathcal{P}^u$ and $f \in F$ iff f is a function symbol in $\mathcal{P}^u$.

Firstly, for every rule $r_i : p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ occurring in $\mathcal{T}^u$, we construct

$$\varphi_i : \ AG.(\zeta_1(B_1 \wedge p_1) \wedge \ldots \wedge \zeta_n(B_n \wedge p_n) \Rightarrow \zeta_0(B_0 \wedge p_0))$$

where $p_0, \ldots, p_n$ are propositions from Π, and if $t_i = f(X)$, for $1 \leq i \leq m$, then ζ_i corresponds to "$EN.$" and $B_i = f$, where f is a proposition from F. Otherwise, ζ_i is the empty string and $B_i = true$.

Intuitively, the propositions p_k in φ_i, corresponding to atoms of the form $p_k(X)$ in r_i, belong to the same state, corresponding to X, say s. The propositions corresponding to atoms of the form $p_k(f(X))$ belong to the state corresponding to $f(X)$, that is next to s following the branch associated to f. Observe that if the depth of atoms corresponding to the state s is d, then the depth of all atoms corresponding to some state next to s is $d+1$. A ground instance of r_i describes how the head ground atom A is derived starting from the ground atoms corresponding to body atoms. The corresponding CTL formula ensures that the proposition p_0 is associated to the state modelling the ground term in A if each proposition modelling some ground body atom is associated to the state modelling its ground term. The CTL formula φ_i encodes the fact that in every state (i.e independently of the value of X), the specified implication, modelling the behaviour of rule r_i, must hold.

Next, we define the CTL formula φ^* guaranteeing that whenever two ground atoms have the same (ground) term, the corresponding propositions must be true in the same state. In particular, φ^* is the following conjunction:

$$\bigwedge_{p_1,p_2\in\Pi \,\wedge\, f\in F} AG.(EN.(p_1 \wedge f \wedge \neg p_2) \Rightarrow \neg EN.(p_2 \wedge f))$$

In other words, let s be any state (AG) corresponding to a ground term u in M and let s' be a state next to s corresponding to the term $t = f(u)$. We do not allow two propositions corresponding to the same ground term t to be associated to different states in the underlying transition system. This ensures that each term corresponds to exactly one state.

Finally, we construct the CTL formula φ:

$$\bigwedge_{b\in pred(D^u)} b \;\wedge \bigwedge_{r_i\in\mathcal{T}^u} \varphi_i \;\wedge\; \varphi^* \;\wedge\; AF.AG.(\bigwedge_{p\in pred(\mathcal{T}^u)} \neg p)$$

Intuitively, the first three conditions guarantee that the associations "proposition/predicate", "state/ground terms" and "transition relation/subterm relation", simulating the atoms in M, must satisfy the "subterm relation" established by rules in $\mathcal{P}^u$. The latter one ensures the finiteness of M, which is similar to the "persistence check" shown for the one function symbols case. This formula essentially states that whatever path we consider on the underlying transition system, there is a time instant from which all propositions will never become true again. We now show that M is finite iff the formula φ above is satisfiable.

($\Rightarrow$) Assume that M is finite and let G be the set of ground terms occurring in the atoms of M. We construct the following transition system $T = (S, s^{\epsilon}, R, \psi)$. The set S of states is defined as $S = \{s^t \mid t \in G\} \cup \{s_e\}$. Intuitively, for every term t occurring in M we construct a state s^t. In particular, when $t = \epsilon$, s^{ϵ} is the state of S denoting the initial state of the transition system. Finally, the state s_e denotes the "empty" state in which no propositions might occur. In fact, the function $\psi : S \to 2^{Prop}$ is defined as follows:

— $\psi(s^{\epsilon}) = \{p \mid p(\epsilon) \in M\}$;
— $\psi(s^t) = \{p \mid p(t) \in M\} \cup \{f\}$, for each $s^t \in S$ where $t = f(u)$;
— $\psi(s_e) = \emptyset$.

Finally, R is the serial transition relation such that:

— for every $s^t, s^u \in S$, where $u = f(t)$, $s^t R s^u$ holds;
— for every $s^t \in S$ such that there is no $s^u \in S$ with $u = f(t)$, $s^t R s_e$ holds;
— $s_e R s_e$ holds.

($\Leftarrow$) Assume now that a transition system $T = (S, s_0, R, \psi)$, such that $T, s_0 \models \varphi$, exists, but M is infinite. Let $Tree(T)$ be the tree-like structure describing T. Note that since database atoms are flat and the maximum nesting level of function symbols in $\mathcal{T}^u$ is 1, for every $d \geq 0$, there is always at least one ground atom in M, whose term has depth d. From this consideration, it follows that the transition system T satisfying φ must satisfy the following property: for each $d \geq 0$ there exists a state s belonging to level d in $Tree(T)$. Indeed, $\psi(s_0)$ must contain (at least) all propositions corresponding to base predicates in order to satisfy the first condition of φ. Moreover, it can be easily shown by induction on d, that for every atom $p(t)$ in M, where t has depth d, there exists a state s at level d of $Tree(T)$ such that $p \in \psi(s)$. This is because the conjunction φ^* and the formulae φ_i, for all rules $r_i \in \mathcal{T}^u$ used to derive $p(t)$, must be satisfied.

However, this would immediately imply that T cannot satisfy the formula $AF.AG.(\neg q)$, occurring in φ. Thus, φ is not satisfiable. □

In the same spirit of the one function symbol case, we can exploit the proof of the theorem above to provide an *EXPTIME* procedure for checking whether an argument $q[1]$ is limited in $\mathcal{MM}(\mathcal{P}^u)$. That is, we only need to change the last condition of formula φ to the following one:

$$AF.AG.(\neg q)$$

stating that in every path, there exists a state s such that in all paths starting from s, q remains false.

As we show in the next theorem, the *EXPTIME* upper bound is optimal. We point out that a proof for the problem below can be easily obtained from the proof of Theorem 5.3. Indeed, it is well-known that *APSPACE* = *EXPTIME*, where *APSPACE* denotes alternating *PSPACE*. So, it suffices to encode a *PSPACE* Turing machine via a logic program as already shown in the proof of Theorem 5.3, and then implement the acceptance part specifically for *PSPACE* machines with alternation. However, for the sake of completeness, we give the full proof below.

THEOREM 5.6. *Let $\mathcal{P}$ be a program with more than one function symbol. Checking whether the minimum model of $\mathcal{P}^u$ is limited is EXPTIME−hard.*

PROOF. We prove the desired hardness result by reduction of the acceptance problem of an *APSPACE* turing machine $TM = (\{0,1,\perp\}, Q, q_0, q_a, \delta)$, where $\{0,1,\perp\}$ denotes the tape alphabet ($\perp$ is the blank symbol), Q the set of states, where $q_0, q_a \in Q$ denote the initial and the accepting states respectively, and δ is the transition function. Furthermore, the set Q is partitioned into two sets Q_u and Q_e, denoting universal and existential states respectively. We can assume w.l.o.g. that every transition in δ is of the form $(q,\alpha) \rightarrow \{(q',\alpha',m'),(q'',\alpha'',m'')\}$. That is, the machine TM may only move its configuration to at most two configurations.

Let $I = a_1,\ldots,a_m$ be an input string for TM and let $n = poly(m)$ be the (polynomialy many) number of tape cells used by TM during its computation. We assume w.l.o.g. that such n cells are extended with other $n+1$ cells, where the first n cells of TM's tape will be used as worktape, which will also contain the input string I at the initial configuration. The remaining $n+1$ cells will be used to store a counter. This counter will be incremented whenever the machine TM changes its configuration. Since TM is an *APSPACE* machine, it is easy to encode such a counter in binary with $n+1$ bits, in order to count the exponentially many configurations. We also assume that the head of TM never attempts to move outside these $2n+1$ cells. When TM halts in a non accepting configuration or the counter reaches its maximum value (i.e., its first bit is set to 1), then the machine rejects input I, otherwise the machine accepts input I.

We now start by constructing a positive normal program, denoted by $\mathcal{P}'$, simulating the computation of TM with input I. Let us first define the unary predicate symbols of $\mathcal{P}'$.

To each i-th tape cell of TM, where $1 \leq i \leq 2n+1$, we associate the atom $\mathtt{cell}_{\mathtt{i}}^{\alpha}(\mathtt{X})$, stating that at configuration X, the i-th cell contains the symbol $\alpha \in \{0,1,\perp\}$. Furthermore, we use atoms of the form $\mathtt{count}_{1}^{\alpha_1}(\mathtt{X}),\ldots,\mathtt{count}_{\mathtt{n+1}}^{\alpha_{\mathtt{n}+1}}(\mathtt{X})$ as shorthands for atoms $\mathtt{cell}_{\mathtt{n+1}}^{\alpha_1}(\mathtt{X}),\ldots,\mathtt{cell}_{\mathtt{2n+1}}^{\alpha_{\mathtt{n+1}}}(\mathtt{X})$. These atoms will represent the counter's bits. Finally, we use the atom $\mathtt{head}_{\mathtt{i}}(\mathtt{X})$ to state that the head of TM is on the i-th cell at configuration

$\mathtt{X}$ and the atom $\mathtt{state^q(X)}$ to denote that the state at configuration $\mathtt{X}$ is q, for every state $q \in Q$.

We now show how the simulation of TM with input I works. We first introduce a rule which will construct the initial configuration of TM:

$$\bigwedge_{\mathtt{i}\in[\mathtt{1..m}]} \mathtt{cell_i^{a_i}(X)} \wedge \bigwedge_{\mathtt{i}\in[\mathtt{m+1..n}]} \mathtt{cell_i^{\perp}(X)} \wedge \bigwedge_{\mathtt{i}\in[\mathtt{1..n+1}]} \mathtt{count_i^0(X)} \wedge \mathtt{state^{q_0}(X)} \wedge \mathtt{head_1(X)} \leftarrow \mathtt{b(X)}.$$

where the first m cells are filled with the symbols of input I, the remaining $n-m$ cells are filled with blanks and the counter's bits are initially set to zeroes.

Now, for every transition rule $(q, \alpha) \to \{(q', \beta, \to), (q'', \gamma, \leftarrow)\} \in \delta$, we first construct, for every $1 \le i \le 2n+1$ the following rules:

$$\mathtt{cell_i^{\beta}(f_1(X))} \wedge \mathtt{head_{i+1}(f_1(X))} \wedge \mathtt{state^{q'}(f_1(X))} \leftarrow \mathtt{state^q(X)}, \mathtt{cell_i^{\alpha}(X)}, \mathtt{head_i(X)}, \mathtt{count_1^0(X)}.$$
$$\mathtt{cell_i^{\gamma}(f_2(X))} \wedge \mathtt{head_{i-1}(f_2(X))} \wedge \mathtt{state^{q''}(f_2(X))} \leftarrow \mathtt{state^q(X)}, \mathtt{cell_i^{\alpha}(X)}, \mathtt{head_i(X)}, \mathtt{count_1^0(X)}.$$

stating that if at the given configuration $\mathtt{X}$, the state is q, the head is on the i-th cell, which contains symbol α and the counter has not reached its maximum value yet, then the machine may move to one of the two allowed configurations $\mathtt{f_1(X)}$ and $\mathtt{f_2(X)}$, where the i-th cell will contain symbol β (resp., γ), the head will be on cell $i+1$ (resp., $i-1$) and the new state will be q' (resp., q'').
Next, in order to propagate all the unmodified cell values at configuration $\mathtt{X}$ to the corresponding next configuration, we also construct the rules of the form below:

$$\mathtt{cell_j^c(f_1(X))} \leftarrow \mathtt{state^q(X)}, \mathtt{cell_i^{\alpha}(X)}, \mathtt{head_i(X)}, \mathtt{count_1^0(X)}, \mathtt{cell_j^c(X)}.$$
$$\mathtt{cell_j^c(f_2(X))} \leftarrow \mathtt{state^q(X)}, \mathtt{cell_i^{\alpha}(X)}, \mathtt{head_i(X)}, \mathtt{count_1^0(X)}, \mathtt{cell_j^c(X)}.$$

for every $1 \le i \le 2n+1$ and for every other tape cell $1 \le j \le 2n+1$ such that $j \ne i$ and for every value $c \in \{0, 1, \perp\}$. The rules essentially state that if a transition from configuration $\mathtt{X}$ to configuration $\mathtt{f_1(X)}$ (resp., configuration $\mathtt{f_2(X)}$) has to be performed, where the content of the i-th cell is modified, then, all other cells containing some value c at configuration $\mathtt{X}$, (i.e., $\mathtt{cell_j^c(X)}$) must still contain such a value at configuration $\mathtt{f_1(X)}$ (resp., $\mathtt{f_2(X)}$).
We can construct almost identical rules for the case where the head moves to different directions for both configurations.

The main difference with the one function symbol case is how the acceptance of TM with input I is encoded via normal positive rules. To do so, we recursively define whether a configuration $\mathtt{X}$ is an accepting configuration.

Clearly, the configuration where the current state is q_a is an accepting configuration:

$$\mathtt{accepting(X)} \leftarrow \mathtt{state^{q_a}(X)}.$$

Furthermore, for every universal state $u \in Q_u$, if some configuration $\mathtt{X}$ has state u, and the next two configurations $\mathtt{f_1(X)}$ and $\mathtt{f_2(X)}$ are accepting configurations, then $\mathtt{X}$ is accepting as well:

$$\mathtt{accepting(X)} \leftarrow \mathtt{state^u(X)}, \mathtt{accepting(f_1(X))}, \mathtt{accepting(f_2(X))}.$$

Similarly, if $e \in Q_e$ is an existential state and some configuration $\mathtt{X}$ has state e and one of the next two configurations is accepting, then $\mathtt{X}$ is accepting:

$$\mathtt{accepting(X)} \leftarrow \mathtt{state^e(X)}, \mathtt{accepting(f_1(X))}.$$
$$\mathtt{accepting(X)} \leftarrow \mathtt{state^e(X)}, \mathtt{accepting(f_2(X))}.$$

Finally, the machine accepts if the initial configuration is an accepting configuration:

$$\mathtt{accept(X)} \leftarrow \mathtt{accepting(X)}, \mathtt{b(X)}.$$

As for the one function symbol case, we also add the rule:

$$\mathtt{accept}(\mathtt{f}_1(\mathtt{X})) \leftarrow \mathtt{accept}(\mathtt{X}).$$

With the addition of this rule, we can immediately show that TM accepts input I iff the minimum model of the obtained program and the database $D = \{\mathtt{b}(\epsilon)\}$ is finite. This will show that checking whether the minimum model of a program of the form of $\mathcal{P}^u$ with at most two function symbols is finite is $coAPSPACE-hard$ and thus $coEXPTIME-hard$. Since, $coEXPTIME = EXPTIME$, and from the definition of $coEXPTIME-hardness$, we show that the problem is $EXPTIME-hard$. □

Also in the multiple function symbols case, the proof of the theorem above can be used to show $EXPTIME-hardness$ for the problem of checking whether an argument (namely, $accept[1]$) is limited in $\mathcal{MM}(\mathcal{P}^u)$.

We conclude this section by providing the desired complexity result for checking m-restrictedness in the case of programs allowing more than one function symbol.

COROLLARY 5.7. *Given a program $\mathcal{P}$ with more than one function symbol, the complexity of checking whether (an argument of) $\mathcal{P}$ is m-restricted is $EXPTIME-complete$.* □

6. EXTENDING MAPPING-RESTRICTION

In this section, we present an extension of the technique proposed in Section 4. This extension allows arbitrary nesting and constants in programs, preventing the proliferation of rules that would make the meaning of programs hard to understand. In Subsection 6.1 we first refine the analysis of the structure that terms may have during the bottom-up evaluation of a program. As a second improvement, in Subsection 6.2 we show that the terms propagated in some argument $p[i]$ are strongly related to the terms propagated in other arguments of p, when the so-called "steadily restricted" condition holds. Finally, in Subsection 6.3, we show that the new technique remains $EXPTIME-complete$ in the general case. However, $PSPACE-completeness$ is guaranteed only in the case that at most one fuction symbol is allowed and such a symbol is unary.

6.1. Argument position inside complex terms

Most termination criteria, including the m-restricted technique, do not take into account the position in which simple terms may occur inside atoms and rules. The possibility to distinguish whether a variable occurs in a particular position within a complex term, gives additional information on how values are propagated between arguments during the bottom-up evaluation of the program. The following example clarifies the importance of considering the position of terms inside other terms.

Example 6.1. Consider the program $\mathcal{P}$, where the only base predicate is $\mathtt{b}$:

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

Starting from any database, it is impossible to derive an atom of the form $\mathtt{p}(\mathtt{f}(\mathtt{t},\mathtt{t}))$, where $\mathtt{t}$ is a ground term. Consequently, rule r_3 and the recursive rule r_4 cannot be activated. Thus, $\mathcal{P}$ is terminating. The corresponding program $\mathcal{P}^u$, used to compute the minimum supported m-set is reported below:

$$\begin{array}{lll}
\rho_0 & : \mathtt{b}_1(\epsilon). & \\
\rho_1 & : \mathtt{q}_1(\mathtt{g}(\mathtt{X})) & \leftarrow \ \mathtt{b}_1(\mathtt{X}). \\
\rho_{2.1} & : \mathtt{p}_1(\mathtt{f}(\mathtt{X})) & \leftarrow \ \mathtt{b}_1(\mathtt{X}). \\
\rho_{2.2} & : \mathtt{p}_1(\mathtt{f}(\mathtt{Y})) & \leftarrow \ \mathtt{q}_1(\mathtt{Y}). \\
\rho_3 & : \mathtt{t}_1(\mathtt{X}) & \leftarrow \ \mathtt{p}_1(\mathtt{f}(\mathtt{X})). \\
\rho_4 & : \mathtt{t}_1(\mathtt{g}(\mathtt{X})) & \leftarrow \ \mathtt{t}_1(\mathtt{X}).
\end{array}$$

Unfortunately, $\mathcal{P}^u$ does not consider the position of terms inside other terms, transforming rule r_3 into ρ_3. The fact that argument $\mathtt{p}[1]$ has a complex term containing a variable occurring twice is lost. The minimum model of $\mathcal{P}^u$ is infinite and thus the m-restricted criterion does not detect the termination of $\mathcal{P}$. □

We propose an extension of the m-restricted technique which is able to overcome the limitation discussed above. We assume that the database contains flat facts only, whereas programs may contain constants and ground facts (with complex terms), and there is no limitation on the maximum nesting level of terms.

The analysis of programs is now performed via an extended version of the unary program $\mathcal{P}^u$. Let us start with some preliminary notations.

We assume a fixed linear order for each simple term occurring in a rule; the different m occurrences of a simple term t are denoted by $t^1, \ldots, t^m$. For instance, considering rule $r_2 : \mathtt{p}(\mathtt{f}(\mathtt{X},\mathtt{Y})) \leftarrow \mathtt{b}(\mathtt{X}), \mathtt{q}(\mathtt{Y})$ in Example 6.1, occurrences of variables in the head are denoted $\mathtt{X}^1$ and $\mathtt{Y}^1$ whereas body occurrences are denoted by $\mathtt{X}^2$ and $\mathtt{Y}^2$. The next definition allows us to build, given a rule r and an occurrence t^j of a simple term t in r, a unary atom containing t, where each predicate or function symbol has associated a subscript denoting the position of t inside the atom or (complex) term.

Definition 6.2. Let t^j be an occurrence of a simple term t in a rule r. We denote with $path(r, t^j)$ the atom defined as:

$$path(r, t^j) = path'(A, t^j)$$

where A is the atom of r where t^j occurs. $path'(\varphi, t^j)$, where φ denotes either an atom or a term, is defined recursively as follows:

— $path'(\varphi, t^j) = \epsilon$, if φ is a constant;
— $path'(\varphi, t^j) = \varphi$, if φ is a variable;
— $path'(\varphi, t^j) = f_i(path'(t_i, t^j))$, if $\varphi = f(t_1, \ldots, t_n)$ where f is either a predicate or a function symbol and t_i is the term where t^j occurs. □

For example, consider the rule $r = \mathtt{p}(\mathtt{f}(\mathtt{X}, \mathtt{g}(\mathtt{c})), \mathtt{X}) \leftarrow \mathtt{q}(\mathtt{g}(\mathtt{X}))$. Let $\mathtt{X}^1, \mathtt{X}^2$ and $\mathtt{X}^3$ be the three occurrences of $\mathtt{X}$ in r and $\mathtt{c}^1$ be the occurrence of $\mathtt{c}$. We have that $path(r, \mathtt{X}^1) = \mathtt{p}_1(\mathtt{f}_1(\mathtt{X}))$, $path(r, \mathtt{X}^2) = \mathtt{p}_2(\mathtt{X})$, $path(r, \mathtt{c}^1) = \mathtt{p}_1(\mathtt{f}_2(\mathtt{g}_1(\epsilon)))$ and $path(r, \mathtt{X}^3) = \mathtt{q}_1(\mathtt{g}_1(\mathtt{X}))$.

By making use of the notion of path defined above, we now show how to construct, starting from a program $\mathcal{P}$, a unary program $\mathcal{P}^v$ where subscripts are also associated to function symbols. This allows to track down the position of simple terms inside complex terms.

Definition 6.3. Given a program $\mathcal{P}$, $\mathcal{P}^v$ denotes the *unary program* consisting of the rules derived from $\mathcal{P}$ as follows:

— for every argument $b[i] \in arg_b(\mathcal{P})$, $\mathcal{P}^v$ contains a fact $b_i(\epsilon)$;

— for every rule $r \in \mathcal{P}$ and every occurrence of a constant e^j in $head(r)$ we add the following rule in $\mathcal{P}^v$ if it is nontrivial:

$$path(r, e^j) \leftarrow \bigwedge_{c^h occurs\ in\ body(r)} path(r, c^h)$$

— for every rule $r \in \mathcal{P}$ and every variable occurrence X^j in $head(r)$ we add the following rule in $\mathcal{P}^v$ if it is nontrivial:

$$path(r, X^j) \leftarrow \bigwedge_{X^k occurs\ in\ body(r)} path(r, X^k) \wedge \bigwedge_{c^h occurs\ in\ body(r)} path(r, c^h)$$

where c^h denotes an occurrence of constant c in the body of r. □

Example 6.4. Consider the program of Example 1.2. Using `cons` to denote the list constructor function symbol and `null` to denote the constant corresponding to the empty list, we can rewrite the program of Example 1.2 to the following program $\mathcal{P}$:

$$r_1 : \mathtt{reverse(cons(a, cons(b, cons(c, null))), null).}$$
$$r_2 : \mathtt{reverse(L_1, cons(X, L_2))} \leftarrow \mathtt{reverse(cons(X, L_1), L_2).}$$

The corresponding unary program $\mathcal{P}^v$ is reported below:

$$\rho_{1.1} : \mathtt{reverse_1(cons_1(}\epsilon\mathtt{)).}$$
$$\rho_{1.2} : \mathtt{reverse_1(cons_2(cons_1(}\epsilon\mathtt{))).}$$
$$\rho_{1.3} : \mathtt{reverse_1(cons_2(cons_2(cons_1(}\epsilon\mathtt{)))).}$$
$$\rho_{1.4} : \mathtt{reverse_1(cons_2(cons_2(cons_2(}\epsilon\mathtt{)))).}$$
$$\rho_{1.5} : \mathtt{reverse_2(}\epsilon\mathtt{).}$$

$$\rho_{2.1} : \mathtt{reverse_1(L_1)} \leftarrow \mathtt{reverse_1(cons_2(L_1)).}$$
$$\rho_{2.2} : \mathtt{reverse_2(cons_1(X))} \leftarrow \mathtt{reverse_1(cons_1(X)).}$$
$$\rho_{2.3} : \mathtt{reverse_2(cons_2(L_2))} \leftarrow \mathtt{reverse_2(L_2).}$$

Rules of the form $\rho_{i.j}$ model the corresponding rule r_i of $\mathcal{P}$. In particular, $\rho_{1.1} - \rho_{1.5}$ model the presence of constants `a`, `b`, `c` and `null` in the ground fact $\mathtt{reverse(cons(a, cons(b, cons(c, null))), null)}$. Since `null` occurs twice, two rules $\rho_{1.4}$ and $\rho_{1.5}$ describing its occurrences are created. Rules $\rho_{2.1} - \rho_{2.3}$ model the derivation of variables $\mathtt{L_1}$, `X` and $\mathtt{L_2}$ in r_2. □

Example 6.5. As another example, consider the program $\mathcal{P}$ of Example 6.1. The corresponding unary program $\mathcal{P}^v$ is as follows:

$$\rho_1 : \mathtt{b_1(}\epsilon\mathtt{).}$$
$$\rho_2 : \mathtt{q_1(g_1(X))} \leftarrow \mathtt{b_1(X).}$$
$$\rho_3 : \mathtt{p_1(f_1(X))} \leftarrow \mathtt{b_1(X).}$$
$$\rho_4 : \mathtt{p_1(f_2(X))} \leftarrow \mathtt{q_1(X).}$$
$$\rho_5 : \mathtt{t_1(X)} \leftarrow \mathtt{p_1(f_1(X)),\ p_1(f_2(X)).}$$
$$\rho_6 : \mathtt{t_1(g_1(X))} \leftarrow \mathtt{t_1(X).}$$

The minimum model of $\mathcal{P}^v$ is $M = \{\mathtt{b_1(}\epsilon\mathtt{), q_1(g_1(}\epsilon\mathtt{)), p_1(f_1(}\epsilon\mathtt{)), p_1(f_2(g_1(}\epsilon\mathtt{)))}\}$. Observe that M is finite, whereas the minimum model of $\mathcal{P}^u$ reported in Example 6.1 is infinite. □

The new version of the derived unary program can be used to extend the m-restricted technique. In particular, we define the class of *extended m-restricted* programs/arguments (*em-restricted* for short) and show the correctness of our approach.

Definition 6.6 (*Extended m-restricted programs*). An argument $p[i]$ occurring in a program $\mathcal{P}$ is said to be *em-restricted* iff $p_i[1]$ is limited in $\mathcal{MM}(\mathcal{P}^v)$.

The set of all *em-restricted arguments* of $\mathcal{P}$ is denoted by $\mathit{EMR}(\mathcal{P})$ and $\mathcal{P}$ is said to be *em-restricted* iff $\mathit{EMR}(\mathcal{P}) = \mathit{arg}(\mathcal{P})$, i.e., $\mathcal{MM}(\mathcal{P}^v)$ is finite. The set of em-restricted programs is denoted by $\mathcal{EMR}$. □

We point out that although the definition above relies on the construction of $\mathcal{P}^v$, it is still possible to define em-restricted programs/arguments via more declarative definitions, as in Definitions 4.2 and 4.4 of Section 4. However, we think that such definitions would be too involved and do not add much to the proposed approach. Thus, for the sake of clarity, we provide such alternative definitions in Appendix B.

We are now ready to present our main correctness results.

THEOREM 6.7. *Given a program $\mathcal{P}$ and an argument $p[i] \in \mathit{arg}(\mathcal{P})$, then*

(*1*) *$p[i] \in \mathit{EMR}(\mathcal{P})$ implies that $p[i]$ is limited in $\mathcal{P}$;*
(*2*) *$\mathcal{P} \in \mathcal{EMR}$ implies that $\mathcal{P}$ is terminating.*

PROOF. To simplify the discussion, we focus on constant-free programs having terms with nesting level at most one. We start by proving the following. For every program $\mathcal{P}$ and database D, if a (ground) atom A is derived using the immediate consequence operator at step i, i.e., $A \in T^i_{\mathcal{P}_D}(\emptyset)$, then for each constant's occurrence e^j in A, there is an atom $path(A, e^j)$ in $T^i_{\mathcal{P}^v}(\emptyset)$. Furthermore, the depth of $path(A, e^j)$ coincides with the depth of the occurrence e^j in A.

First observe that D is composed by flat facts only, whereas $\mathcal{P}^v$ contains a fact $b_k(\epsilon)$ for every $b[k] \in \mathit{arg}_b(\mathcal{P})$. Suppose now that our statement is true on step i. We now show that it is true on step $i+1$ too. Suppose that we derive an atom A by applying $T^{i+1}_{\mathcal{P}_D}(\emptyset)$ by using rule r in $\mathcal{P}$. This means that all ground atoms in its body belong to $T^i_{\mathcal{P}_D}(\emptyset)$. Consequently, $path(B, c^h)$ is true in $T^i_{\mathcal{P}^v}(\emptyset)$ for every constant's occurrence c^h in B for every body atom B of r. Consequently, all rules derived from r in the unary program $\mathcal{P}^v$ can be applied and $path(A, e^j)$ is true in $T^{i+1}_{\mathcal{P}^v}(\emptyset)$ for every constant's occurrence e^j in A. Observe that the depth of $path(A, e^j)$ coincides with the depth of the occurrence e^j in A by definition.

Now, with this result in hand, we prove the theorem.

(1) Suppose that there is an atom A whose argument $p[i]$ is em-restricted, but not limited in $\mathcal{P}$. Then, for any depth d there is a step, say k, such that some constant's occurrence e^j in the i-th argument of A has depth k in $T^k_{\mathcal{P}_D}(\emptyset)$. Yet this implies that the corresponding atom $path(A, e^j)$ has depth d in $T^k_{\mathcal{P}^v}(\emptyset)$. Consequently, $p_i[1]$ is not limited in the minimal model of $\mathcal{P}^v$ and $p[i]$ cannot be em-restricted. This leads to the desired contradiction.

(2) This follows from Item 1 and from the fact that a program is terminating iff every argument is limited. □

The revised technique extends the one previously defined. In fact, as shown in the proof of the following theorem, the new approach recognizes larger sets of limited arguments and terminating programs.

THEOREM 6.8. *For every program $\mathcal{P}$:*

(*1*) *$\mathit{MR}(\mathcal{P}) \subseteq \mathit{EMR}(\mathcal{P})$;*
(*2*) *$\mathcal{P} \in \mathcal{MR} \Rightarrow \mathcal{P} \in \mathcal{EMR}$.*

PROOF. Assuming constant-free programs, where the nesting level of terms is at most one, the definition of $\mathcal{P}^v$ can be simplified as follows:

— for every argument $b[i] \in arg_b(\mathcal{P})$, $\mathcal{P}^v$ contains a fact $b_i(\epsilon)$;
— for every rule $r \in \mathcal{P}$ and every variable occurrence X^j in $head(r)$, $\mathcal{P}^v$ contains a rule:

$$path(head(r), X^j) \leftarrow \bigwedge_{X^j occurs\ in\ body(r)} path(r, X^j)$$

if it is not trivial.

Observe also that the program $\mathcal{P}^u$ can be obtained from $\mathcal{P}^v$ by first deleting subscripts from the function symbols, then deleting the eventual duplicate body atoms/rules, and, at the end, by deleting trivial rules. This means that for every ground atom A derived by the immediate consequence operator $T^k_{\mathcal{P}^v}(\emptyset)$ on step k, there is a corresponding atom A', obtained from A by deleting the subscripts from its function symbols, belonging to $T^k_{\mathcal{P}^u}(\emptyset)$.

(1) Consider an atom A of $\mathcal{P}$ and suppose that its argument $p[i]$ is m-restricted, but not em-restricted. Since $p[i]$ is not em-restricted, then for any depth d there is an atom $H = path(A, e^j)$, where e^j is the occurrence of some constant in the i-th argument of A, having depth d in $T^k_{\mathcal{P}^v}(\emptyset)$ on some step k. From the observation above, it follows that $H' \in T^k_{\mathcal{P}^u}(\emptyset)$, where H' is obtained from H by deleting the subscripts from its function symbols and describes the same constant occurrence e^j in the i-th argument of A. Consequently, $p_i[1]$ is not limited in the minimum model of $\mathcal{P}^u$ and $p[i]$ cannot be m-restricted. This leads to the desired contradiction.

(2) The weaker inclusion $\mathcal{MR} \subseteq \mathcal{EMR}$ follows from Item 1, whereas the stronger one follows from the fact that the program $\mathcal{P}$ of Example 6.1 is em-restricted but non m-restricted. □

6.2. Steadily restricted arguments

In this section we will show that even if a program $\mathcal{P}$ is not recognized as em-restricted, the set of em-restricted arguments $EMR(\mathcal{P})$ can be profitably exploited to compute a larger set of limited arguments of $\mathcal{P}$. In particular, we show that if an em-restricted argument $p[i]$ enjoys the so-called “steadily restricted” property, then any other argument of the predicate symbol p is guaranteed to be limited.

Let $\mathcal{P}$ be a program. A *stratification* of $\mathcal{P}^v$ is a partition of $\mathcal{P}^v$ into sub-programs $\mathcal{P}^v_1, \ldots, \mathcal{P}^v_n$ (called strata) where for each predicate p of $\mathcal{P}^v$, all rules defining p are contained in a stratum, the recursive rules occurring in the same stratum are mutually recursive, and for every $1 \leq k \leq n$, if a predicate q occurs in the body of a rule in $\mathcal{P}^v_k$, all rules defining q are contained in a stratum $\mathcal{P}^v_j$, with $j \leq k$. Finally, a rule r of some stratum $\mathcal{P}^v_k$ is an *exit rule* of $\mathcal{P}^v_k$ if it is not recursive, i.e., if $body(r)$ contains only base predicates or predicates defined by rules occurring in strata $\mathcal{P}^v_j$, with $j < k$.

Example 6.9. Consider the program $\mathcal{P}^v$ of Example 6.4. We have that $\mathcal{P}^v$ can be partitioned into the strata $\mathcal{P}^v_1 = \{\rho_{1.1}, \rho_{1.2}, \rho_{1.3}, \rho_{1.4}, \rho_{2.1}\}$ and $\mathcal{P}^v_2 = \{\rho_{1.5}, \rho_{2.2}, \rho_{2.3}\}$, defining the predicate symbols `reverse`$_1$ and `reverse`$_2$ respectively. Furthermore, all rules of $\mathcal{P}^v_1$, except for $\rho_{2.1}$ are exit rules of $\mathcal{P}^v_1$, whereas rules $\rho_{1.5}$, $\rho_{2.2}$ are the exit rules of $\mathcal{P}^v_2$. □

We are now ready to define the notion of steadily restricted arguments. Recall that for every argument $p[i]$ of $\mathcal{P}$, there exists a corresponding predicate symbol p_i in the unary program $\mathcal{P}^v$ and vice versa.

Definition 6.10 (*Finitely decreasing stratum*). Let $\mathcal{P}$ be a program and let $\mathcal{P}_1^v, \ldots, \mathcal{P}_n^v$ be a stratification of $\mathcal{P}^v$. We say that a stratum $\mathcal{P}_k^v$ is *finitely decreasing* if for each predicate p_i in $\mathcal{P}_k^v$, the argument $p[i]$ of $\mathcal{P}$ is em-restricted and for every rule $\rho \in \mathcal{P}_k^v$ derived from a recursive rule of $\mathcal{P}$, the following conditions hold:

— ρ is not ground and recursive, and
— for every atom $B \in body(\rho)$ which is mutually recursive with $head(\rho)$, $dept(head(\rho)) < dept(B)$ holds. □

Example 6.11. Let $\mathcal{P}$ be the program of Example 1.2 and $\mathcal{P}^v$ be the corresponding unary program shown in Example 6.4. Considering the stratification of $\mathcal{P}^v$, $\mathcal{P}_1^v = \{\rho_{1.1}, \rho_{1.2}, \rho_{1.3}, \rho_{1.4}, \rho_{2.1}\}$ and $\mathcal{P}_2^v = \{\rho_{1.5}, \rho_{2.2}, \rho_{2.3}\}$, we have that $\mathcal{P}_1^v$ is finitely decreasing. This is because the argument $\texttt{reverse}[1]$ is em-restricted and the only rule in $\mathcal{P}_1^v$ derived from a recursive rule of $\mathcal{P}$ (namely $\rho_{2.1}$) is not ground, recursive and the depth of the body atom is strictly greater than the depth of the head atom. □

Intuitively, the recursive rules occurring in the finitely decreasing stratum propagate a value among them and the depth of this value decreases during the propagation. The propagation of a value among the recursive rules is guaranteed because they are not ground. Indeed, every rule in $\mathcal{P}^v$ has just one variable, and every not ground rule in $\mathcal{P}^v$ has a variable both in its head and in its body by construction of $\mathcal{P}^v$. The depth of the propagated value decreases since for every recursive rule the depth of the head atom is strictly smaller than the depth of any body atom.

Definition 6.12 (*Steadily restricted argument*). Let $\mathcal{P}$ be a program and $\mathcal{P}_1^v, \ldots, \mathcal{P}_n^v$ be a stratification of $\mathcal{P}^v$. An argument $p[i]$ of $\mathcal{P}$ is said to be *steadily restricted* if:

(1) there exists a predicate p_j (possibly equal to p_i) defined in a finitely decreasing stratum of $\mathcal{P}^v$, and
(2) let $\mathcal{P}_k^v$ be the stratum where p_i is defined, for every predicate q_j occurring in the body of some exit rule of $\mathcal{P}_k^v$, $q[j]$ is em-restricted. □

Clearly, in Item (2) we could have considered all arguments recognized as limited by some criterion.

Example 6.13. Consider again the program $\mathcal{P}$ of Example 1.2 and the transformed program $\mathcal{P}^v$ of Example 6.4, with stratification $\mathcal{P}_1^v = \{\rho_{1.1}, \rho_{1.2}, \rho_{1.3}, \rho_{1.4}, \rho_{2.1}\}$ and $\mathcal{P}_2^v = \{\rho_{1.5}, \rho_{2.2}, \rho_{2.3}\}$. We have that the argument $\texttt{reverse}[2]$ of $\mathcal{P}$ is steadily restricted. In fact, there exists the predicate $\texttt{reverse}_1$, which is defined in the finitely decreasing stratum $\mathcal{P}_1^v$. Furthermore, the exit rules of $\mathcal{P}_2^v$ (i.e., where $\texttt{reverse}_2$ is defined) are $\rho_{1.5}$ and $\rho_{2.2}$ and the only predicate occurring in their body is $\texttt{reverse}_1$, whose corresponding argument $\texttt{reverse}[1]$ is em-restricted. □

Observe that in the example above the stratification of $\mathcal{P}^v$ was unique. In the general case, $\mathcal{P}^v$ may have different stratifications. However, since the recursive rules occurring in the same stratum must be mutually recursive (see the definition of stratification) the choice of the particular stratification does not influence the "steadily restricted" property of an argument of $\mathcal{P}$. Thus, any of them can be chosen in an arbitrary way.

PROPOSITION 6.14. *Given a program $\mathcal{P}$, every steadily restricted argument of $\mathcal{P}$ is also limited in $\mathcal{P}$.*

PROOF. Let $\mathcal{P}$ be a program and $p[i]$ be a *steadily restricted* argument of $\mathcal{P}$ and D be a database. Consider now a ground atom A of the form $p(t_1, \ldots, t_n)$ derived at step k using the immediate consequence operator, i.e., $A \in T^k_{\mathcal{P}_D}(\emptyset)$ using rule r of $\mathcal{P}$.

As shown in the proof of Theorem 6.7, for each constant's occurrence e^j in A, there is an atom $path(A, e^j)$ in $T^k_{\mathcal{P}^v}(\emptyset)$ and the depth of $path(A, e^j)$ coincides with the depth of the occurrence e^j in A. (Result 1)

Since t_i may be a complex term, different constant occurrences may be present in it, and for each of them there is a ground atom of the form $p_i(\dots)$ in $T^k_{\mathcal{P}^v}(\emptyset)$ in $\mathcal{P}^v$, defined by means of a corresponding rule in $\mathcal{P}^v$. We denote by $Atoms(r, A, i)$ the set of such p_i-atoms in $T^k_{\mathcal{P}^v}(\emptyset)$ and denote by $Rules(r, A, i)$, the rules of $\mathcal{P}^v$ used to derive the atoms in $Atoms(r, A, i)$.

It follows from Result 1 that the depth of t_i coincides with the maximum depth of the atoms in $Atoms(r, A, i)$. (Result 2)

Let us now consider the rule r used to derive $A \in T^k_{\mathcal{P}_D}(\emptyset)$.

1. If r is not recursive, then the corresponding rules in $Rules(r, A, i)$ are the exit rules of the stratum where p_i is defined, and all body atoms of rules in $Rules(r, A, i)$ have limited arguments. Let d be the upper bound of the depth of terms occurring in limited arguments of $\mathcal{MM}(\mathcal{P}^v)$ and Δ be the upper bound for term growth after the application of one rule in $\mathcal{P}^v$. Obviously, d and Δ are finite and $dept(B) < d + \Delta$ for every $B \in Atoms(r, A, i)$. Thus, from Result 2, $dept(t_i)$ cannot be unbounded too.

2. If r is recursive, then there exists a $p[j]$, such that p_j is defined in finitely decreasing stratum of $\mathcal{P}^v$. We assumed that A was obtained on step k using the immediate consequence operator. Suppose that the next step of the immediate consequence operator that uses rule r is k_1 and let us call $A_1(u_1, \dots, u_n)$ the atom derived by means of this application of r.

It follows from Result 2, that $dept(t_j)$ and $dept(u_j)$ coincide with the maximum depth of atoms in $Atoms(r, A, j)$ and $Atoms(r, A_1, j)$, respectively.

Since p_j is defined in a finitely decreasing stratum of $\mathcal{P}^v$, the depth of atoms in $Atoms(r, A_1, j)$ must be smaller than the depth of the corresponding atoms in $Atoms(r, A, j)$. Consequently, $dept(u_j) < dept(t_j)$. This reasoning can be extended to the further application of r. Since we know that $p[j]$ is em-restricted, we conclude that r can be applied only a finite number of times.

Since, (1) each non-recursive rule defining predicate p in $\mathcal{P}$ cannot be used to generate a term of unbounded depth in the i-th argument of A and (2) each recursive rule defining predicate p in $\mathcal{P}$ can be applied only a finite number of times, $p[i]$ must be limited. □

As a remark, the definition of steadily restricted arguments along with Proposition 6.14 above allow us to establish a connection between different arguments of the same predicate symbol. Furthermore, we can profitably exploit such a result together with the techniques defined in this paper in order to determine the termination of different practical programs.

Example 6.15. Consider the following program $\mathcal{P}$ merging two lists:

$$
\begin{aligned}
r_1 &: \texttt{merge}([\texttt{a},\texttt{b}],[\texttt{c},\texttt{d},\texttt{e}],[\,]).\\
r_2 &: \texttt{merge}(\texttt{L}_1,\texttt{L}_2,[\texttt{X}|[\texttt{Y}|\texttt{L}_3]]) \leftarrow \texttt{merge}([\texttt{X}|\texttt{L}_1],[\texttt{Y}|\texttt{L}_2],\texttt{L}_3)\\
r_3 &: \texttt{merge}([\,],\texttt{L}_2,[\texttt{Y}|\texttt{L}_3]) \leftarrow \texttt{merge}([\,],[\texttt{Y}|\texttt{L}_2],\texttt{L}_3)\\
r_4 &: \texttt{merge}(\texttt{L}_1,[\,],[\texttt{X}|\texttt{L}_3]) \leftarrow \texttt{merge}([\texttt{X}|\texttt{L}_1],[\,],\texttt{L}_3)
\end{aligned}
$$

The arguments merge[1] and merge[2] are em-restricted, whereas merge[3] is steadily restricted. Indeed, the program $\mathcal{P}^v$ (reported in Appendix B) consists of two strata $\mathcal{P}^v_1$ and $\mathcal{P}^v_2$ defining, respectively, $\{\texttt{merge}_1, \texttt{merge}_2\}$ and $\texttt{merge}_3$. Furthermore, $\mathcal{P}^v_2$ depends on the finitely decreasing stratum $\mathcal{P}^v_1$. The exit rules of $\mathcal{P}^v_2$ are $\rho_{2.3}$, $\rho_{2.4}$, $\rho_{3.2}$ and $\rho_{4.2}$. The body predicates of these rules are $\texttt{merge}_1$ and $\texttt{merge}_2$ that correspond to em-restricted arguments of $\mathcal{P}$. Therefore, the program $\mathcal{P}$ is terminating. □

6.3. Complexity results

As previously stated, complexity results for the extended version of our technique can be obtained from results for the original criterion. That is, we first need to show that the construction of $\mathcal{P}^v$, given a program $\mathcal{P}$, is negligible (i.e., it is feasible in polynomial time w.r.t. $size(\mathcal{P})$). Once such a result is shown, we can use the results introduced in Section 5 to show the complexity of computing the set of em-restricted arguments of $\mathcal{P}$.

LEMMA 6.16. *Given a program $\mathcal{P}$, the unary program $\mathcal{P}^v$ is computable in polynomial time w.r.t. $size(\mathcal{P})$.*

PROOF. We start by showing that constructing the atom $path(r, e^j)$ for some rule $r \in \mathcal{P}$ and some simple term occurrence e^j in r is a linear time procedure. To show the above result consider the following. An atom (resp. term) A occurring in $\mathcal{P}$ can be seen as a tree. In particular we use $tree(A)$ to denote the following tree: if A is a simple term, $tree(A)$ is the singleton tree where the only node is labeled with A. Otherwise, let $A = p(t_1, \ldots, t_n)$, where p is either a predicate or a function symbol, then $tree(A)$ is the tree having the root node labeled with p and whose children are $tree(t_i)$, for $1 \leq i \leq n$. Please note that given a simple term e occurring in A, every occurrence e^j of e in A, for some j is a leaf of $tree(A)$. Thus, constructing $path(r, e^j)$, for some rule $r \in \mathcal{P}$ is equivalent to the problem of traversing $tree(A)$, where A is the atom of r in which e^j occurs, in a depth first fashion, until a leaf labeled with e is found for the j-th time. The path that leads from the root to this leaf describes the atom $path(r, e^j)$. Tree traversal is a linear time procedure, and thus the construction of $path(r, e^j)$ is linear time as well. We are now ready to show our result. From Definition 6.3, $\mathcal{P}^v$ contains the following rules: a linear number of facts, each one constructible in constant time and a rule constructed for every rule $r \in \mathcal{P}$ and simple term occurrence e^j in r. From the definition of $size(\mathcal{P})$, the number of rules of $\mathcal{P}$ and the number of simple terms occurrences are linear w.r.t. $size(\mathcal{P})$, thus the number of rules in $\mathcal{P}^v$ is polynomial w.r.t. $size(\mathcal{P})$. Furthermore, since the construction of "$path$" atoms is linear, the construction of all rules of $\mathcal{P}^v$ is a polynomial time procedure as well. □

In the following, to simplify the complexity analysis, we focus on constant-free programs with nesting level of terms not greater than one. We show that for programs having at most one function symbol and such a symbol is unary, checking em-restrictedness is $PSPACE-complete$, whereas in all other cases, the problem is $EXPTIME-complete$.

Intuitively, the *PSPACE* bounds for the unary function symbol case hold because for such programs $\mathcal{P}$, $\mathcal{P}^v$ belongs to the same class of programs to which the simpler $\mathcal{P}^u$ belongs to.

THEOREM 6.17. *Given a program $\mathcal{P}$, checking whether a program $\mathcal{P}$ is em-restricted is $PSPACE-complete$ if $\mathcal{P}$ admits at most one function symbol and this symbol is unary.*

PROOF. Observe that when a program $\mathcal{P}$ has only one function symbol and such a symbol is unary, the corresponding program $\mathcal{P}^v$ contains only one function symbol of the form f_1. Indeed, $\mathcal{P}^v$ coincides with $\mathcal{P}^u$, when the subscript is removed from f_1. Thus, we immediately prove the claim from Theorem 5.2 and Theorem 5.3. □

The proof above easily shows that also checking whether an argument of $\mathcal{P}$ is em-restricted is $PSPACE-complete$, when $\mathcal{P}$ has only one function symbol which is unary.

Suppose now that $\mathcal{P}$ admits one or more function symbols of any arity. Regarding the *EXPTIME* upper bound, we exploit the proof shown for Theorem 5.5. For the hardness result, we cannot directly make use of previous results. This follows from the fact that when $\mathcal{P}$ contains only one function symbol of any arity, the corresponding program $\mathcal{P}^v$ might not be of the form needed by the hardness proof given for Theorem 5.6.

Therefore, we first prove the following lemma.

LEMMA 6.18. *Given a program $\mathcal{P}$, checking whether (an argument of) $\mathcal{P}$ is em-restricted is $EXPTIME-hard$ even if $\mathcal{P}$ admits at most one function symbol and this symbol is binary.*

PROOF. We need to show that there exists a program $\mathcal{P}$ with only one function symbol where this symbol is binary such that $\mathcal{P}^v$ is expressive enough to simulate the computation of an *EXPTIME* Turing machine.

To this aim, we proceed as follows. Let $\mathcal{P}'$ be the unary program shown in the proof of Theorem 5.6 simulating the computation of an *EXPTIME* Turing machine. We show how to construct a program $\mathcal{P}$, having only one function symbol (which is binary), whose unary program $\mathcal{P}^v$ "almost" coincides with $\mathcal{P}'$. This will immediately show that there actually exists a program with one function symbol and this symbol is binary, such that checking whether such a program is em-restricted is $EXPTIME-hard$.

In particular, the construction of $\mathcal{P}$ can be performed as follows. We modify $\mathcal{P}'$ by substituting each occurrence of $\mathtt{f_1(X)}$ (resp. $\mathtt{f_2(X)}$) with $\mathtt{f(X, Dummy)}$ (resp. $\mathtt{f(Dummy, X)}$), where $\mathtt{Dummy}$ is a new variable not occurring in $\mathcal{P}'$. Next, we add an atom $\mathtt{b(f(Dummy, Dummy))}$ with base predicate symbol b to the body of each rule having the variable $\mathtt{Dummy}$ in its head. As an example, consider the rule of $\mathcal{P}'$:

$$\mathtt{accept(f_1(X))} \leftarrow \mathtt{accept(X)}.$$

The constructed rule using Dummy variables is:

$$\mathtt{accept(f(X, Dummy))} \leftarrow \mathtt{accept(X), b(f(Dummy, Dummy))}.$$

This rule will have two corresponding rules in $\mathcal{P}^v$:

$$\mathtt{accept_1(f_1(X))} \leftarrow \mathtt{accept_1(X)}.$$
$$\mathtt{accept_1(f_2(Dummy))} \leftarrow \mathtt{b_1(f_1(Dummy)), b_1(f_2(Dummy))}.$$

The construction described above leads to the program $\mathcal{P}$ that has exactly one function symbol where this symbol is binary. Its unary program $\mathcal{P}^v$ has the same rules as $\mathcal{P}'$ (up to predicate symbols renaming) plus the rules with variable $\mathtt{Dummy}$ (i.e., $\mathtt{Dummy}$-rules). Observe that $\mathtt{Dummy}$-rules cannot be activated since their body atoms $\mathtt{b_1(f_1(Dummy)), b_1(f_2(Dummy))}$ contain complex terms inside the base predicate symbol $\mathtt{b_1}$, but databases my contain only flat atoms. Consequently, $\mathcal{P}^v$ simulates the computation of an *EXPTIME* Turing machine as well as $\mathcal{P}'$. □

With the previous lemma in place, we can finally show the complexity of checking em-restrictedness in the general case.

THEOREM 6.19. *Given a program $\mathcal{P}$, checking whether a program $\mathcal{P}$ is em-restricted is $EXPTIME-complete$.*

PROOF. The *EXPTIME* upper bound follows from the proof shown for Theorem 5.5. Regarding the $EXPTIME-hardness$, If $\mathcal{P}$ contains more than one function symbol of arbitrary arity, em-restricted programs strictly contain the m-restricted ones. So the lower bound immediately follows. If $\mathcal{P}$ contains only one function symbol that is at least binary, the hardness follows from Lemma 6.18. □

The above complexity result also holds for the problem of checking whether an argument of $\mathcal{P}$ is em-restricted.

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 $\mathcal{P}^u$ derived from the input program $\mathcal{P}$. As finiteness of the minimum model of $\mathcal{P}^u$ is decidable, its evaluation gives us a set of limited arguments of the original program, called m-restricted.

Precise complexity results and relative expressive power are also discussed. In particular, we have shown that the complexity is $EXPTIME\!-\!complete$ in general and $PSPACE\!-\!complete$ when only one function symbol is allowed. Furthermore, our technique generalizes previous approaches, such as $\mathcal{AR}$, but is incomparable with other recently proposed techniques. The technique can be easily combined with other techniques such as Adornment rewriting [Greco et al. 2013b], a recently proposed orthogonal method that transforms a program into an adorned equivalent 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 terminating.

Then, we defined an extended version of the proposed approach that is able to further enlarge the class of programs identified as terminating. Complexity results of the extension have been provided as well, by modifications of the proofs for the original technique. 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 is usually much smaller than the size of the database. Furthermore, although we have studied the general case where programs allow different function symbols, we point out that in several contexts where function symbols are used to model stratification of rules, temporal phenomena or implement the object model in logics, just one unary function symbol is needed.

Aknowledgements. The authors would like to thank Cristian Molinaro and the anonymous referees for the useful comments.

REFERENCES

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

Catriel Beeri and Moshe Y. Vardi. 1984. A Proof Procedure for Data Dependencies. *J. ACM* 31, 4 (1984), 718–741.

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

Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems* 29, 2 (2007), 10.

Marco Calautti, Georg Gottlob, and Andreas Pieris. 2015. Chase Termination for Guarded Existential Rules. In *Proc. of the 34th ACM Symposium on Principles of Database Systems (PODS)*. 91–103.

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2015a. Logic Program Termination Analysis Using Atom Sizes. In *Proc. of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*. 2833–2839.

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2015b. Rewriting-based Check of Chase Termination. In *Proc. of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*.

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2016a. Exploiting Equality Generating Dependencies in Checking Chase Termination. *Proc. of the VLDB Endowment* 9, 5 (2016), 396–407.

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2016b. Using Linear Constraints for Logic Program Termination Analysis. *Theory and Practice of Logic Programming* 16, 3 (2016), 353–377.

Marco Calautti, Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2015. Checking termination of bottom-up evaluation of logic programs with function symbols. *Theory and Practice of Logic Programming* 15, 6 (2015), 854–889.

Marco Calautti, Sergio Greco, and Irina Trubitsyna. 2013. Detecting decidable classes of finitely ground logic programs with function symbols. In *Principles and Practice of Declarative Programming*. ACM, 239–250.

Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *Journal of Artificial Intelligence Research (JAIR)* 48 (2013), 115–174.

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. 2008. Computable Functions in ASP: Theory and Implementation. In *Proc. International Conference on Logic Programming*. 407–424.

Luciano Caroprese, Irina Trubitsyna, and Ester Zumpano. 2007. Implementing Prioritized Reasoning in Logic Programming. In *ICEIS 2007 - Proc. Ninth International Conference on Enterprise Information Systems, Volume AIDSS*. 94–100.

Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. 1984. Inclusion Dependencies and Their Interaction with Functional Dependencies. *J. Comput. System Sci.* 28, 1 (1984), 29–59.

Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. 2005. Testing for Termination with Monotonicity Constraints. In *Proc. International Conference on Logic Programming*. 326–340.

Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. 2008. The chase revisited. In *Proc. of the 27th ACM Symposium on Principles of database systems (PODS)*. 149–158.

Alin Deutsch and Val Tannen. 2003. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*. 201–212.

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005), 89–124.

Filippo Furfaro, Gianluigi Greco, and Sergio Greco. 2004. Minimal founded semantics for disjunctive logic programs and deductive databases. *Theory and Practice of Logic Programming* 4, 1-2 (2004), 75–93.

Martin Gebser, Torsten Schaub, and Sven Thiele. 2007. GrinGo : A New Grounder for Answer Set Programming. In *Proc. International Conference on Logic Programming and Nonmonotonic Reasoning*. 266–271.

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM* 38, 3 (1991), 620–650.

Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proc. International Joint Conference and Symposium on Logic Programming*. 1070–1080.

Michael Gelfond and Vladimir Lifschitz. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9, 3/4 (1991), 365–386.

Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with AProVE. In *Proc. of the 7th International Joint Conference on Automated Reasoning*. 184–191.

T. Gogacz and J. Marcinkowski. 2014. All-Instances Termination of Chase is Undecidable. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*. 293–304.

Bernardo Cuenca Grau, Ian Horrocks, Markus Krotzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. 2013. Acyclicity Notions for Existential Rules and Their Application to Query Answering in Ontologies. *Journal of Artificial Intelligence Research (JAIR)* 47 (2013), 741–808.

Sergio Greco, Cristian Molinaro, and Francesca Spezzano. 2012. *Incomplete Data and Data Dependencies in Relational Databases*. Morgan & Claypool Publishers.

Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2013a. Checking Logic Program Termination Under Bottom-Up Evaluation. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2013b. Logic Programming with Function Symbols: Checking Termination of Bottom-up Evaluation Through Program Adornments. *Theory and Practice of Logic Programming* 13, 4-5 (2013), 737–752.

Sergio Greco and Francesca Spezzano. 2010. Chase Termination: A Constraints Rewriting Approach. *Proc. of the VLDB Endowment* 3, 1 (2010), 93–104.

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2011. Stratification Criteria and Rewriting Techniques for Checking Chase Termination. *Proc. of the VLDB Endowment* 4, 11 (2011), 1158–1168.

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2012. On the Termination of Logic Programs with Function Symbols. In *Proc. of the International Conference on Logic Programming (Technical Communications)*. 323–333.

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2015. Checking Chase Termination: Cyclicity Analysis and Rewriting Techniques. *IEEE Transactions on Knowledge Data Engineering* 27, 3 (2015), 621–635.

Sergio Greco, Irina Trubitsyna, and Ester Zumpano. 2007. On the Semantics of Logic Programs with Preferences. *Journal of Artificial Intelligence Research (JAIR)* 30 (2007), 501–523.

Yuliya Lierler and Vladimir Lifschitz. 2009. One More Decidable Class of Finitely Ground Programs. In *Proc. International Conference on Logic Programming*. 489–493.

Massimo Marchiori. 1996. Proving Existential Termination of Normal Logic Programs. In *Algebraic Methodology and Software Technology*. 375–390.

Bruno Marnette. 2009. Generalized schema-mappings: from termination to tractability. In *PODS*. 13–22.

Arne Meier, Martin Mundhenk, Michael Thomas, and Heribert Vollmer. 2008. The Complexity of Satisfiability for Fragments of CTL and CTL*. *Electronic Notes on Theoretical Computer Science* 223 (2008), 201–213.

Michael Meier, Michael Schmidt, and Georg Lausen. 2009. On Chase Termination Beyond Stratification. *CoRR* abs/0906.4228 (2009).

Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. 2007. Termination Analysis of Logic Programs Based on Dependency Graphs. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation*. 8–22.

Manh Thang Nguyen, Danny De Schreye, Jürgen Giesl, and Peter Schneider-Kamp. 2011. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming* 11, 1 (2011), 33–63.

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

Enno Ohlebusch. 2001. Termination of Logic Programs: Transformational Methods Revisited. *Applicable Algebra in Engineering, Communication and Computing* 12, 1/2 (2001), 73–116.

Adrian Onet. 2013. The Chase Procedure and its Applications in Data Exchange. In *Data Exchange, Integration, and Streams*. 1–37.

Peter Schneider-Kamp, Jürgen Giesl, and Manh Thang Nguyen. 2009a. The Dependency Triple Framework for Termination of Logic Programs. In *Proc. International Symposium on Logic-based Program Synthesis and Transformation*. 37–51.

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

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

Danny De Schreye and Stefaan Decorte. 1994. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming* 19/20 (1994), 199–260.

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

A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM* 32, 3 (1985), 733–749.

Tran Cao Son, Enrico Pontelli, and Phan Huy Tu. 2007. Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. *Journal of Artificial Intelligence Research (JAIR)* 29 (2007), 353–389.

Tommi Syrjanen. 2001. Omega-Restricted Logic Programs. In *Proc. of the International Conference on Logic Programming and Nonmonotonic Reasoning*. 267–279.

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

A. TEMPORAL LOGICS

A.1. Linear temporal logic

Linear Temporal Logic (LTL) is an extension of propositional logic with the notion of time. In particular, in LTL we assume to have a countably infinite set $Prop$ of propositions, the set of standard logic connectives $\neg, \wedge, \vee, \Rightarrow$, and a set of temporal operators. *Well-formed* LTL formulae are either propositions of $Prop$ or, if φ and ξ are well-formed LTL formulae, then $\neg\varphi$, $\varphi \vee \xi$, $\varphi \wedge \xi$, $\varphi \Rightarrow \xi$, $G.\varphi$, $F.\varphi$ and $N.\varphi$, are well-formed LTL formulae. The operators G, F and N are the LTL temporal operators denoting the following:

— $G.\varphi$: formula φ always holds;
— $F.\varphi$: formula φ holds sometime in the future;
— $N.\varphi$: formula φ holds at the next instant.

The semantics of LTL is defined in terms of linear-time structures. A linear-time structure is a pair $K = (S, \psi)$, where $S = s_0, s_1, \ldots$ is a possibly infinite sequence of states (i.e., time instants) and $\psi : \{s_0, s_1, \ldots\} \rightarrow 2^{Prop}$ is a function mapping each state to a set of propositions, i.e., it tells us what propositions are true at a given state.

Given a linear-time structure $K = (S, \psi)$ where $S = s_0, \ldots$, a state s_i in S and LTL formulae φ,ξ, the satisfiability relation $\models$ is inductively defined as follows:

— $K, s_i \models p$ iff $p \in \psi(s_i)$, where $p \in Prop$;
— $K, s_i \models \neg\varphi$ iff $K, s_i \not\models \varphi$;
— $K, s_i \models \phi \wedge \xi$ iff $K, s_i \models \varphi$ and $K, s_i \models \xi$;
— $K, s_i \models \varphi \vee \xi$ iff $K, s_i \models \varphi$ or $K, s_i \models \xi$;
— $K, s_i \models \varphi \Rightarrow \xi$ iff $K, s_i \models \neg\phi$ or $K, s_i \models \xi$;
— $K, s_i \models N.\varphi$ iff $K, s_{i+1} \models \varphi$;
— $K, s_i \models G.\varphi$ iff for every $j \geq i$, $K, s_j \models \varphi$;
— $K, s_i \models F.\varphi$ iff $\exists j \geq i$ such that $K, s_j \models \varphi$

We say that an LTL formula φ is satisfiable if there exists a linear-time structure $K = (S, \psi)$, with $S = s_0, s_1, \ldots$, such that $K, s_0 \models \varphi$.

The problem of checking whether an LTL formula is satisfiable is $PSPACE-complete$ [Meier et al. 2008].

A.2. Computation tree logic

Computation Tree Logic (CTL) is another extension of propositional logic with the notion of time, where the underlying models might encode multiple execution paths, rather than only one. Formally, given an infinite set $Prop$ of propositions and logic connectives as shown for LTL, well-formed CTL formulae are either propositions of $Prop$ or if φ and ψ are well-formed CTL formulae, then $\neg\varphi$, $\varphi \vee \xi$, $\varphi \wedge \xi$, $\varphi \Rightarrow \xi$, $AG.\varphi$, $AF.\varphi$, $AN.\varphi$, $EG.\varphi$, $EF.\varphi$ and $EN.\varphi$ are well-formed CTL formulae. We consider the CTL temporal operators AG, AF, AN, EG, EF and EN intuitively denoting the following:

— $AG.\phi$: in all execution paths, formula ϕ always holds;
— $AF.\phi$: in all execution paths, formula ϕ holds sometime in the future;
— $AN.\phi$: in all execution paths, formula ϕ holds at the next instant;
— $EG.\phi$: there exists an execution path such that, formula ϕ always holds;
— $EF.\phi$: there exists an execution path such that, formula ϕ holds sometime in the future;
— $EN.\phi$: there exists an execution path such that, formula ϕ holds at the next instant.

Models of CTL formulae denote multiple execution paths, encoded as all possible paths over a given transition system. A *transition system* is a quadruple $T = (S, s_0, R, \psi)$, where S is a set of states, $s_0 \in S$ is the *initial state* and $R \subseteq S \times S$ is a *serial* transition relation, i.e., for every state $s \in S$, there is a state $s' \in S$ such that sRs'. Finally, $\psi : S \to 2^{Prop}$ is a function mapping every state to a set of propositions. A *path* in a transition system T is an infinite sequence $\pi = \pi_1, \pi_2, \ldots,$ of states such that $x_i R x_{i+1}$ for all $i > 0$. Intuitively, the set S of states and the relation R of T together define a graph, where each state s is a node and edges are defined by R. A path in T simply denotes a path in the corresponding graph. Thus, transition systems can also be seen as a tree-like structure, rooted in s_0, obtained by the unwinding of the aforementioned graph. We denote such a tree with $Tree(T)$. Every path in $Tree(T)$ corresponds to a path in T.

Given a transition system $T = (S, s_0, R, \psi)$, a state s in S and CTL formulae φ and ξ, the satisfiability relation $\models$ is inductively defined as follows:

— $T, s \models p$ iff $p \in \psi(s)$, where $p \in Prop$;
— $T, s \models \neg\varphi$ iff $T, s \not\models \varphi$;
— $T, s \models \phi \wedge \xi$ iff $T, s \models \varphi$ and $T, s \models \xi$;
— $T, s \models \varphi \vee \xi$ iff $T, s \models \varphi$ or $T, s \models \xi$;
— $T, s \models \varphi \Rightarrow \xi$ iff $T, s \models \neg\phi$ or $T, s \models \xi$;
— $T, s \models AG.\varphi$ iff for all paths $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_i \models \varphi$, for all $i > 0$;
— $T, s \models AF.\varphi$ iff for all paths $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_i \models \varphi$, for some $i > 0$;
— $T, s \models AN.\varphi$ iff for all paths $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_2 \models \varphi$;
— $T, s \models EG.\varphi$ iff there exists a path $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_i \models \varphi$, for all $i > 0$;
— $T, s \models EF.\varphi$ iff there exists a path $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_i \models \varphi$, for some $i > 0$;
— $T, s \models EN.\varphi$ iff there exists a path $\pi = \pi_1, \pi_2, \ldots$ with $\pi_1 = s$, holds $T, \pi_2 \models \varphi$.

We say that a CTL formula φ is satisfiable if there exists a transition system $T = (S, s_0, R, \psi)$ such that $T, s_0 \models \varphi$.

The problem of checking satisfiability of a CTL formula is $EXPTIME{-}complete$ even when we restrict to only three of the presented operators [Meier et al. 2008].

B. EM-RESTRICTED PROGRAMS: ALTERNATIVE DEFINITIONS AND EXAMPLES

Given a simple term (i.e., a constant or a variable) occurrence u^j in a term t, we denote by $str(t, u^j)$ the string of function symbols used to reach u^j and defined as follows:

— $str(t, u^j) = \epsilon$ if $t = u$
— $str(t, u^j) = f_i \cdot s$ if $t = f(v_i, \ldots, v_n)$ and u^j occurs in v_i and $str(v_i, u^j) = s$.

Let $\mathcal{P}$ be a program, an *extended mapping set* (*em-set*) $\mathcal{V}_\mathcal{P}$ for $\mathcal{P}$ is a set of pairs $p[i]/s$ such that $p[i] \in arg(\mathcal{P})$ and $s \in FN^*_\mathcal{P}$, where $FN_\mathcal{P}$ denotes the alphabet consisting of symbols of the form f_j such that f occurs in $\mathcal{P}$ and $j \in [1, ar(f)] \cap \mathcal{N}$.

Let $\mathcal{V}_\mathcal{P}$ be an em-set of $\mathcal{P}$, $A = p(t_1, \ldots, t_n)$ an atom occurring in $\mathcal{P}$ and t_i the term containing an occurrence X^j of a variable X, we say that X^j occurring in A is mapped to $s \in FN^*_\mathcal{P}$ w.r.t. a em-set $\mathcal{V}_\mathcal{P}$, and denote by $\langle \mathcal{V}_\mathcal{P}, A, X^j \rangle \leadsto s$ if $p[i]/str(t_i, X^j) \cdot s \in \mathcal{V}_\mathcal{P}$.

Definition B.1 (*Supported em-set*). Let $\mathcal{P}$ be a program and let $\mathcal{V}_\mathcal{P}$ be an em-set of $\mathcal{P}$. We say that $\mathcal{V}_\mathcal{P}$ is *supported* if:

(1) $p[i]/\epsilon \in \mathcal{V}_\mathcal{P}$ for every argument $p[i] \in arg_b(\mathcal{P})$, and
(2) for every rule $r \in \mathcal{P}$ and every simple term t in $head(r)$, if for every atom $B \in body(r)$ and for every occurrence of a constant c^j in B the mapping $\langle \mathcal{V}_\mathcal{P}, B, c^j \rangle \leadsto \epsilon$ holds, then:
— $\langle \mathcal{V}_\mathcal{P}, head(r), t \rangle \leadsto \epsilon$ holds if t is a constant;

— $\langle \mathcal{V}_\mathcal{P}, head(r), t\rangle \rightsquigarrow s$ holds if t is a variable and s is a string such that for every body atom B of r and every occurrence of a term t^j in B the mapping $\langle \mathcal{V}_\mathcal{P}, B, t^j\rangle \rightsquigarrow s$ holds. □

The minimum supported em-set of $\mathcal{P}$ is denoted by $\mathcal{V}^*_\mathcal{P}$.

Definition B.2. Let $\mathcal{P}$ be a program and let $\mathcal{V}^*_\mathcal{P}$ the minimum supported em-set of $\mathcal{P}$. We say that an argument $p[i]$ of $\mathcal{P}$ is *em-restricted* iff the set $\{p[i]/s \mid p[i]/s \in \mathcal{V}^*_\mathcal{P}\}$ is finite. Finally, $\mathcal{P}$ is em-restricted if all its arguments are em-restricted. □

B.1. Example 6.15

Consider the program of Example 6.15

$$\begin{array}{lll}
r_1: & \mathtt{merge([a,b],[c,d,e],[\,])}. & \\
r_2: & \mathtt{merge(L_1,L_2,[X|[Y|L_3]])} & \leftarrow \mathtt{merge([X|L_1],[Y|L_2],L_3)}. \\
r_3: & \mathtt{merge([\,],L_2,[Y|L_3])} & \leftarrow \mathtt{merge([\,],[Y|L_2],L_3)}. \\
r_4: & \mathtt{merge(L_1,[\,],[X|L_3])} & \leftarrow \mathtt{merge([X|L_1],[\,],L_3)}.
\end{array}$$

By denoting the binary list constructor operator with `cons` and the empty list with the constant `null` we can rewrite into the below set of rules $\mathcal{P}$:

$$\begin{array}{lll}
r_1: & \mathtt{merge(cons(a,cons(b,null)),cons(c,cons(d,cons(e,null))),null)}. & \\
r_2: & \mathtt{merge(L_1,L_2,cons(X,cons(Y,L_3)))} & \leftarrow \mathtt{merge(cons(X,L_1),cons(Y,L_2),L_3)}. \\
r_3: & \mathtt{merge(null,L_2,cons(Y,L_3))} & \leftarrow \mathtt{merge(null,cons(Y,L_2),L_3)}. \\
r_4: & \mathtt{merge(L_1,null,cons(X,L_3))} & \leftarrow \mathtt{merge(cons(X,L_1),null,L_3)}.
\end{array}$$

The derived unary program $\mathcal{P}^v$ is as follows:

$$\begin{array}{ll}
\rho_{1.1} & \mathtt{merge_1(cons_1(}\epsilon\mathtt{))}. \\
\rho_{1.2} & \mathtt{merge_1(cons_2(cons_1(}\epsilon\mathtt{)))}. \\
\rho_{1.3} & \mathtt{merge_1(cons_2(cons_2(}\epsilon\mathtt{)))}. \\
\rho_{1.4} & \mathtt{merge_2(cons_1(}\epsilon\mathtt{))}. \\
\rho_{1.5} & \mathtt{merge_2(cons_2(cons_1(}\epsilon\mathtt{)))}. \\
\rho_{1.6} & \mathtt{merge_2(cons_2(cons_2(cons_1(}\epsilon\mathtt{))))}. \\
\rho_{1.7} & \mathtt{merge_2(cons_2(cons_2(cons_2(}\epsilon\mathtt{))))}. \\
\rho_{1.8} & \mathtt{merge_3(}\epsilon\mathtt{)}.
\end{array}$$

$$\begin{array}{lll}
\rho_{2.1} & \mathtt{merge_1(L_1)} & \leftarrow \mathtt{merge_1(cons_2(L_1))}. \\
\rho_{2.2} & \mathtt{merge_2(L_2)} & \leftarrow \mathtt{merge_2(cons_2(L_2))}. \\
\rho_{2.3} & \mathtt{merge_3(cons_1(X))} & \leftarrow \mathtt{merge_1(cons_1(X))}. \\
\rho_{2.4} & \mathtt{merge_3(cons_2(cons_1(Y)))} & \leftarrow \mathtt{merge_2(cons_1(Y))}. \\
\rho_{2.5} & \mathtt{merge_3(cons_2(cons_2(L_3)))} & \leftarrow \mathtt{merge_3(L_3)}.
\end{array}$$

$$\begin{array}{lll}
\rho_{3.1} & \mathtt{merge_2(L_2)} & \leftarrow \mathtt{merge_2(cons_2(L_2)),\ merge_1(}\epsilon\mathtt{)}. \\
\rho_{3.2} & \mathtt{merge_3(cons_1(Y))} & \leftarrow \mathtt{merge_2(cons_1(Y)),\ merge_1(}\epsilon\mathtt{)}. \\
\rho_{3.3} & \mathtt{merge_3(cons_2(L_3))} & \leftarrow \mathtt{merge_3(L_3),\ merge_1(}\epsilon\mathtt{)}.
\end{array}$$

$$\begin{array}{lll}
\rho_{4.1} & \mathtt{merge_1(L_1)} & \leftarrow \mathtt{merge_1(cons_2(L_1)),\ merge_2(}\epsilon\mathtt{)}. \\
\rho_{4.2} & \mathtt{merge_3(cons_1(X))} & \leftarrow \mathtt{merge_1(cons_1(X)),\ merge_2(}\epsilon\mathtt{)}. \\
\rho_{4.3} & \mathtt{merge_3(cons_2(L_3))} & \leftarrow \mathtt{merge_3(L_3),\ merge_2(}\epsilon\mathtt{)}.
\end{array}$$

where each rule $\rho_{i.j}$ is derived from rule r_i. □

C. RULES REWRITING

In this appendix, we formally present how rules can be rewritten so that constants do not appear inside rules and the maximum level of nesting of terms is one. More specifically, for every rule r, first each constant c occurring in r is replaced with a fresh variable X_c and an atom $b_c(X_c)$ is added to the body of r, where b_c is a new base predicate symbol[6]. Then, r is rewritten into a set of rules $Flat(r)$ such that the maximum depth of each term is 1. For every program $\mathcal{P}$, $Flat(\mathcal{P}) = \cup_{r \in \mathcal{P}} Flat(r)$ denotes the set of rewritten rules derived from rules in $\mathcal{P}$.

In the algorithm below, we shall use substitutions consisting of pairs $variable/term$ to remember the (fresh) variables used to replace (complex) terms. The application of a substitution $\theta = \{U_1/u_1, \ldots, U_k/u_k\}$ to an atom A, denoted as $A\theta$, is the atom obtained from A by replacing each variable U_i with the term u_i, for $1 \leq i \leq k$.

We start by presenting the function *FlattenAtoms*, getting as input a set of atoms and returning a set of atoms and a substitution. In particular, this function returns a set of atoms obtained from the input one by replacing complex terms at the second level with fresh variables. A substitution mapping each new variable to the corresponding complex term is returned as well.

For example, given the set of atoms $S = \{p(f(f(X)),\ g(X, g(f(Y), Z)))\}$, the output would be the set $F = \{p(f(A),\ g(X, B))\}$ and the substitution $\theta = \{A/f(X),\ B/g(f(Y), Z)\}$.

```
Function FlattenAtoms;
input:  A set of atoms $S$;
output: A set of flattened atoms $F$ and a substitution $\theta$;
begin
    $F := S$;
    $\theta := \emptyset$;
    for each atom $p(t_1, \ldots, t_n) \in S$ and each $t_i = f(u_1, \ldots, u_k)$ with $dept(t_i) > 1$ do
        replace each complex term $u_j$ with a fresh variable $U_j$ in $F$;
        add $U_j/u_j$ to $\theta$ for each complex term $u_j$;
    end
    return $(F, \theta)$;
end.
```

We are now ready to present the rule rewriting algorithm. In essence, given a rule, the following algorithm first makes the rule constant-free and then flattens, one level at the time, all atoms occurring in the head and the body.

[6]The equivalence between the rewritten program and the source program can be guaranteed by adding to the database a fact $b_c(c)$.

```
Algorithm Rule rewriting;
input:  A rule $r$;
output: A set of rules $Flat(r)$;
begin
  // Constants removal
  Let $b_c$ be a new base predicate symbol, for each constant $c$;
  add the atom $b_c(X_c)$ to $body(r)$, for each constant $c$ occuring in $r$;
  replace all occurrences of a constant $c$ in $r$ with a new variable $X_c$;

  $Flat(r) := \{r\}$;

  // head flattening
  while $\exists r' \in Flat(r)$ with $dept(head(r')) > 1$ do
    Let $(F, \theta) =$ FlattenAtoms($head(r')$);
    Let $X_1, \ldots, X_m$ be the variables occurring in $F$;
    Let $C = p'(X_1, \ldots, X_m)$ be an atom with a fresh predicate symbol $p'$;
    replace $r'$ with the rules $F \leftarrow C$ and $C\theta \leftarrow body(r')$ in $Flat(r)$;
  end

  // body flattening
  while $\exists r' \in Flat(r)$ with $dept(body(r')) > 1$ do
    Let $(F, \theta) =$ FlattenAtoms($body(r')$);
    Let $X_1, \ldots, X_m$ be the variables occurring in $F$;
    Let $C = p'(X_1, \ldots, X_m)$ be an atom with a fresh predicate symbol $p'$;
    replace $r'$ with the rules $C \leftarrow F$ and $head(r') \leftarrow C\theta$ in $Flat(r)$;
  end

  return $Flat(r)$;
end.
```

Example C.1. Consider the following rule:

$$r: \ \mathtt{p(f_1(f_2(X,Y)),f_2(f_2(f_1(X),Y),Z),c) \leftarrow q(f_1(f_1(X)),f_2(f_2(f_1(Z),Y),X))}.$$

Step 0: All constants are removed from r, obtaining the rule:

$$r: \ \mathtt{p(f_1(f_2(X,Y)),f_2(f_2(f_1(X),Y),Z),X_c) \leftarrow q(f_1(f_1(X)),f_2(f_2(f_1(Z),Y),X)),b_c(X_c)}.$$

Step 1: $Flat(r)$ contains the rule:

$$r: \ \mathtt{p(f_1(f_2(X,Y)),f_2(f_2(f_1(X),Y),Z),X_c) \leftarrow q(f_1(f_1(X)),f_2(f_2(f_1(Z),Y),X)),b_c(X_c)}.$$

First phase (head flattening).
Step 2: Selecting r we have that $\theta = \{\mathtt{A/f_2(X,Y),B/f_2(f_1(X),Y)}\}$. In $Flat(r)$, rule r is replaced by the rules:

$$r_1: \ \mathtt{p(f_1(A),f_2(B,Z),X_c) \leftarrow p_1(A,B,Z,X_c)}.$$
$$r_2: \ \mathtt{p_1(f_2(X,Y),f_2(f_1(X),Y),Z,X_c) \leftarrow q(f_1(f_1(X)),f_2(f_2(f_1(Z),Y),X)),b_c(X_c)}.$$

Step 3: Selecting r_2 we have that $\theta = \{\mathtt{C/f_1(X)}\}$. In $Flat(r)$, rule r_2 is replaced by the rules:

$$r_3: \ \mathtt{p_1(f_2(X,Y),f_2(C,Y),Z,X_c) \leftarrow p_2(X,Y,C,Z,X_c)}.$$
$$r_4: \ \mathtt{p_2(X,Y,f_1(X),Z,X_c) \leftarrow q(f_1(f_1(X)),f_2(f_2(f_1(Z),Y),X)),b_c(X_c)}.$$

Second phase (body flattening).
Step 4: Selecting r_4 we have that $\theta = \{\mathtt{D/f_1(X),E/f_2(f_1(Z),Y)}\}$. In $Flat(r)$, rule r_4 is

replaced by the rules:

$$\begin{array}{l} r_5: \ \mathtt{q_1(D,E,X,X_c) \leftarrow q(f_1(D),f_2(E,X)),b_c(X_c).} \\ r_6: \ \mathtt{p_2(X,Y,f_1(X),Z,X_c) \leftarrow q_1(f_1(X),f_2(f_1(Z),Y),X,X_c).} \end{array}$$

Step 5: Selecting r_6 we have that $\theta = \{\mathtt{F/f_1(Z)}\}$. In $Flat(r)$, rule r_6 is replaced by the rules:

$$\begin{array}{l} r_7: \ \mathtt{q_2(X,F,Y,X_c) \leftarrow q_1(f_1(X),f_2(F,Y),X,X_c).} \\ r_8: \ \mathtt{p_2(X,Y,f_1(X),Z,X_c) \leftarrow q_2(X,f_1(Z),Y,X_c).} \end{array}$$

The final set of rules $Flat(r)$ is as follows:

$$\begin{array}{l} r_1: \ \mathtt{p(f_1(A),f_2(B,Z),X_c) \leftarrow p_1(A,B,Z,X_c).} \\ r_3: \ \mathtt{p_1(f_2(X,Y),f_2(C,Y),Z,X_c) \leftarrow p_2(X,Y,C,Z,X_c).} \\ r_8: \ \mathtt{p_2(X,Y,f_1(X),Z,X_c) \leftarrow q_2(X,f_1(Z),Y,X_c).} \\ r_7: \ \mathtt{q_2(X,F,Y,X_c) \leftarrow q_1(f_1(X),f_2(F,Y),X,X_c).} \\ r_5: \ \mathtt{q_1(D,E,X,X_c) \leftarrow q(f_1(D),f_2(E,X)),b_c(X_c).} \end{array}$$

□

It is worth noting that, at each step, $Flat(r)$ contains only one rule which may need to be rewritten. Furthermore, the number of steps needed to flatten a rule r is equal to $\max\{0,\ dept(head(r)) - 1\} + \max\{0,\ dept(body(r)) - 1\}$.