



# Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols

MARCO CALAUTTI, University of Edinburgh

SERGIO GRECO and IRINA TRUBITSYNA, DIMES, Università della Calabria

In this article, 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 that 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 of 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 to other techniques known in the literature. We also introduce an extension of the proposed approach that 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*;

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. 2017. Detecting Decidable Classes of Finitely Ground Logic Programs with Function Symbols. *ACM Trans. Comput. Logic* 18, 4, Article 28 (November 2017), 42 pages. <https://doi.org/10.1145/3143804>

## 1 INTRODUCTION

Recent developments of answer set solvers have seen significant progress toward providing support for function symbols. The interest in this area is justified by the fact that function symbols make languages more expressive and often make modeling easier and the resulting encodings more readable and concise. The main problem with the introduction of function symbols is that common inference tasks become undecidable.

This article refines and extends results presented at the 15th International Symposium on Principles and Practice of Declarative Programming [Calautti et al. 2013]. Part of this work was contributed by M. Calautti during his doctoral study at DIMES, Università della Calabria.

Authors' addresses: M. Calautti, Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, Scotland, UK; email: mcalautt@inf.ed.ac.uk; S. Greco and I. Trubitsyna, Via P. Bucci, 87036 Rende (CS), Italy; emails: {greco, trubitsyna}@dimes.unical.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2017 ACM 1529-3785/2017/11-ART28 \$15.00

<https://doi.org/10.1145/3143804>

Current techniques analyze how values are propagated among predicate arguments, to detect whether such arguments are *limited*—that is, whether the sets of values that can be associated with these arguments are finite. However, these methods have limited capacity to analyze 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:

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

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 preceding program is that most of the current criteria analyze 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 preceding example, 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 article, we present a new approach for checking termination of the bottom-up evaluation of logic programs with function symbols. The new technique analyzes how function symbols are nested in complex terms to detect larger classes of terminating programs. The preceding analysis is carried out by constructing strings describing how function symbols are nested. To analyze strings of function symbols representing complex terms, the technique proposed here 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  $\mathcal{P}$  also terminates.

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

$$\begin{aligned} \rho_{1.1} : p_1(X) &\leftarrow b_1(X). & \rho_{1.2} : p_2(X) &\leftarrow b_1(X). \\ \rho_{2.1} : q_1(f(X)) &\leftarrow p_1(X), p_2(X). & \rho_{2.2} : q_2(g(X)) &\leftarrow p_1(X), p_2(X). \\ \rho_{3.1} : s_1(X) &\leftarrow q_1(f(X)). & \rho_{3.2} : s_2(X) &\leftarrow q_2(X). \\ \rho_{4.1} : p_1(f(X)) &\leftarrow s_1(X). & \rho_{4.2} : p_2(f(X)) &\leftarrow s_2(X). \end{aligned}$$

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 were techniques proposed to perform such kind of reasoning.

Thus, the proposed technique also analyzes the relationships that may exist among arguments of the same predicate. This allows the recognition of terminating classical logic programs (e.g., the one presented in the following), not recognized by several other techniques.

*Example 1.2.* Consider the following program:

$$\begin{aligned} & \text{reverse}([a, b, c], []). \\ & \text{reverse}(L_1, [X|L_2]) \leftarrow \text{reverse}([X|L_1], L_2). \end{aligned}$$

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  $\text{reverse}_1$  and  $\text{reverse}_2$ , respectively. The specific structure of the two subprograms allow us to establish that the limitedness of  $\text{reverse}_1$  implies that also  $\text{reverse}_2$  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), and Calautti et al. (2015)) cannot be applied. This is due to the fact that in the preceding program, the same function symbol occurs in more than one rule and in the body of rules.

Although we concentrate on positive programs, the technique proposed here 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. 2009a, 2009b, 2010; Nguyen et al. 2007, 2011; Giesl et al. 2014; Bruynooghe et al. 2007; Bonatti 2004; Baselice et al. 2009).

In this article, 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) and Greco et al. (2013a), all of the excellent works mentioned previously 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 (Caroprese et al. 2007; Greco 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 semidecidable, recent research has concentrated on the identification of sufficient conditions, which 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  $\omega$ -*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 a 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  $\lambda$ -*restricted* programs, presented in Gebser et al. (2007). Here, to each predicate  $p$  we assign a natural number  $\lambda(p)$  called

its *rank* that 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 preceding classes. 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 that 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. 2012b; Calautti et al. 2015c) 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 analyze 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 were introduced by Greco et al. (2012). The definition of *acyclic program* relies on the notion of propagation graph, an evolution of the argument graph that 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 labeled argument graph, a directed graph whose edges are labeled 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 labeled argument graph and therefore in the size of the input program.

The class of *rule-bounded* programs has been proposed in Calautti et al. (2015a, 2016a) 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 nondeterministic polynomial-time procedure.

Finally, an orthogonal technique was 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 proposed so far, will be discussed in more detail in Section 3.

A research area related to the one investigated here is—the problem of defining sufficient conditions guaranteeing that the chase fixpoint algorithm terminates, independently of the input

database<sup>1</sup> (Greco and Spezzano 2010; Greco et al. 2011, 2012a, 2015; Grau et al. 2013; Onet 2013; Calautti et al. 2015, 2015b, 2016b; Cali 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 formulas called *tuple-generating dependencies* (TGDs) and *equality-generating dependencies* (EGDs) (Fagin et al. 2005).

Two main classes of TGDs have been 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 again received 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 work 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  $\Sigma$  into an “equivalent” but more informative set  $\Sigma^\alpha$ . Termination of the chase over  $\Sigma^\alpha$  guarantees termination of the chase over  $\Sigma$  (Greco and Spezzano 2010; Greco et al. 2011, 2015).

In Grau et al. (2013), the model-faithful acyclicity (MFA) and model-summarizing acyclicity (MSA) techniques were proposed. The idea is to run the semioblivious (also known as 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  $f(\dots f(\dots)\dots)$ ), 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 article, we present a new semidynamic 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 that take values from finite domains.

<sup>1</sup>This problem is known to be undecidable in general (Gogacz and Marcinkowski 2014).

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) formulas, 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 article 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 formulas. 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 article 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 nonnegative 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  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n \geq 0$  and  $t_1, \dots, t_n$  are terms. An *atom* is of the form  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate symbol of arity  $n \geq 0$  and  $t_1, \dots, t_n$  are terms. A (positive) *rule*  $r$  is of the following form:

$$A \leftarrow B_1, \dots, B_m,$$

where  $m \geq 0$  and  $A, B_1, \dots, B_m$  are atoms.<sup>2</sup> The atom  $A$  is called the *head* of  $r$  and is denoted by  $head(r)$ . The conjunction  $B_1, \dots, B_m$  is called the *body* of  $r$  and is denoted by  $body(r)$ . With a slight abuse of notation, we sometimes use  $body(r)$  to also denote the *set* of atoms appearing in  $body(r)$ .

<sup>2</sup>In this article, we consider positive normal rules—that is, rules where the head consists of a single atom and the body does not contain negated atoms.

We assume that rules are *range restricted*—that is, 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 (respectively, 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 (respectively, rules)  $S$ , we denote by  $\text{arg}(S) = \text{arg}_b(S) \cup \text{arg}_d(S)$  the set of arguments occurring in  $S$ , where  $\text{arg}_b(S)$  (respectively,  $\text{arg}_d(S)$ ) denotes the set of arguments of the base (respectively, 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 is denoted by  $D$ , whereas a finite set of rules defining derived predicates is called *logic program* (or simply *program*) and, by convention, 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  $p < q$ ) 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  $p < q$  and  $q < p$ . Two atoms are mutually recursive if their predicates are mutually recursive. Finally, a rule  $r$  is said to be recursive if  $\text{body}(r)$  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 that 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 that can be built using predicate symbols appearing in  $\mathcal{P}_D$  and ground terms of  $H_{\mathcal{P}_D}$ .

A rule (respectively, atom)  $r'$  is a *ground instance* of a rule (respectively, 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  $\text{ground}(r)$  to denote the set of all ground instances of  $r$  and  $\text{ground}(\mathcal{P})$  to denote the set of all ground instances of the rules in  $\mathcal{P}$  (i.e.,  $\text{ground}(\mathcal{P}) = \cup_{r \in \mathcal{P}} \text{ground}(r)$ ).

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

An interpretation of  $\mathcal{P}_D$  is a *model* of  $\mathcal{P}_D$  if it satisfies every ground rule in  $\text{ground}(\mathcal{P}_D)$ . Clearly, every model of  $\mathcal{P}_D$  contains  $D$ . In this article, we are interested in computing the set of facts that 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  $\text{MM}(\mathcal{P}_D)$ .

The minimum model of  $\mathcal{P}_D$  can be computed via the *immediate consequence operator* of  $\mathcal{P}_D$ , which is a function  $T_{\mathcal{P}_D} : 2^{B_{\mathcal{P}_D}} \rightarrow 2^{B_{\mathcal{P}_D}}$  defined as follows: for every interpretation  $I$ ,  $T_{\mathcal{P}_D}(I) = \{A \mid A \leftarrow B_1, \dots, B_n \in \text{ground}(\mathcal{P}_D) \text{ and } \{B_1, \dots, B_n\} \subseteq I\}$ . The  $i$ -th iteration of  $T_{\mathcal{P}_D}$  ( $i \geq 1$ ) with respect to an interpretation  $I$  is defined as follows:  $T_{\mathcal{P}_D}^1(I) = T_{\mathcal{P}_D}(I)$  and  $T_{\mathcal{P}_D}^i(I) = T_{\mathcal{P}_D}(T_{\mathcal{P}_D}^{i-1}(I))$  for  $i > 1$ . The minimum model of  $\mathcal{P}_D$  coincides with  $T_{\mathcal{P}_D}^\infty(\emptyset)$ . 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 and  $D$  a database, and let  $M = \mathcal{MM}(\mathcal{P}_D)$ . An argument  $p[i] \in \text{arg}(\mathcal{P})$  is said to be *limited* in  $M$  if the set  $\{t_i \mid p(t_1, \dots, t_n) \in \mathcal{MM}(\mathcal{P}_D)\}$  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  $\text{arg}(\mathcal{P})$  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  $T_{\mathcal{P}_D}^n(\emptyset) = T_{\mathcal{P}_D}^\infty(\emptyset)$ .

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

## 3 TERMINATION CRITERIA

As discussed in Section I, 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 analyzing 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—that is, 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  $p(t_1, \dots, t_n)$ ,  $A^0$  denotes the predicate symbol  $p$ , and  $A^i$  denotes term  $t_i$ , for  $1 \leq i \leq n$ . The *depth*  $\text{dept}(X, t)$  of a variable  $X$  in a term  $t$  that contains  $X$  is recursively defined as follows:

$$\begin{aligned} \text{dept}(X, X) &= 0, \\ \text{dept}(X, f(t_1, \dots, t_m)) &= 1 + \max_{i : t_i \text{ contains } X} \text{dept}(X, t_i). \end{aligned}$$

The depth of a term  $t$  is defined as  $\text{dept}(t) = \max\{\text{dept}(X, t) \mid X \text{ occurs in } t\}$ ; analogously, the depth of an atom is defined as  $\text{dept}(p(t_1, \dots, t_n)) = \max\{\text{dept}(X, t_i) \mid i \in [1, n] \wedge X \text{ occurs in } t_i\}$ , 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  $\phi$  from  $\text{arg}(\mathcal{P})$  to nonnegative 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  $\text{body}(r)$  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:

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



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

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

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

We use  $AR(\mathcal{P})$  to denote the set of *restricted arguments* of  $\mathcal{P}$ —that is,  $AR(\mathcal{P}) = \{p[i] \mid p[i] \in \text{arg}(\mathcal{P}) \wedge \exists \phi \text{ s.t. } \phi(p[i]) \text{ is defined}\}$ . A program  $\mathcal{P}$  is said to be *argument restricted* iff  $AR(\mathcal{P}) = \text{arg}(\mathcal{P})$ . For example, program  $\mathcal{P}$  of Example 3.2 is argument restricted.<sup>3</sup> 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:

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

where the input list and initial counter are given via databases of the form  $D = \{\text{input}([a, b, c, \dots], \emptyset)\}$ . It is easy to see that the preceding program is not argument restricted, as there is no finite argument ranking for argument  $\text{count}[2]$ . However, even though the terms propagated in argument  $\text{count}[2]$  “grow” at each application of rule  $r_2$ , the same application of rule  $r_2$  causes the terms propagated in argument  $\text{count}[1]$  to “decrease” their nesting level. This implies that only a finite number of terms might be produced in argument  $\text{count}[2]$ . The bounded technique is able to detect this behavior 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  $\{p(f(g(x))) \leftarrow p(g(x))\}$ , 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 us 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  $\{p(x, x) \leftarrow b(x). \quad p(f(x), x) \leftarrow q(x, x). \quad q(x, y) \leftarrow p(x, y)\}$ , identified as terminating by the technique proposed in this article, 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, 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. 2016a). 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.

<sup>3</sup>It is easy to see that if  $AR(\mathcal{P}) = \text{arg}(\mathcal{P})$ , there exists an argument ranking  $\phi$  that is a total function.

Specifically, the *size of a term*  $t$ , denoted by  $\text{size}(t)$ , 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  $f(t_1, \dots, t_n)$  and its size is  $\text{size}(t) = n + \sum_{i=1}^n \text{size}(t_i)$ . The size of an atom  $A = p(t_1, \dots, t_n)$ , denoted  $\text{size}(A)$ , is the  $n$ -vector  $(\text{size}(t_1), \dots, \text{size}(t_n))$ .

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  $f(t_1, \dots, t_n)$  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 the atom in the head and the atom in the body, respectively. Then  $\text{size}(A) = (t, 1 + n)$  and  $\text{size}(B) = (2 + h + t, n)$ .

*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,  $(\alpha_1, \alpha_2)$ ,  $(2 + h + t, n)$ , and  $(t, 1 + n)$  are vectors, and  $\cdot$  denotes the classical scalar product operator. For the program to be rule bounded, positive integers  $\alpha_1$  and  $\alpha_2$  must exist such that the inequality is satisfied for all possible nonnegative 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.

Membership in the class of rule-bounded programs can be established via a nondeterministic 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, was proposed in Calautti et al. (2015a).

## 4 MAPPING-RESTRICTED PROGRAMS

In this section we assume without loss of generality 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 of 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, which are used to describe the form of atoms derivable during the bottom-up evaluation of a program (Section 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 (Section 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  $\mathcal{P}^u$  (Section 4.3). Finally, in Section 4.4, we compare our criterion to 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  $p[i] \in \text{arg}(\mathcal{P})$  and  $s \in F_{\mathcal{P}}^*$ , where  $F_{\mathcal{P}}$  denotes the alphabet consisting of all function symbols occurring in  $\mathcal{P}$  and  $F_{\mathcal{P}}^*$  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  $\epsilon$  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

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

where  $\text{str}(t)$  is the set of *strings induced by  $t$* , defined as

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

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

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

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

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

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

## 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  $p(t_1, \dots, t_n)$  is derived during the computation of the minimum model of a program  $\mathcal{P}$ , then for each  $t_i$  and every  $s \in \text{str}(t_i)$ , the mapping  $p[i]/s$  belongs to  $\mathcal{U}$ . In other words, 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  $b[i] \in \text{arg}_b(\mathcal{P})$ , and
- (2) for every rule  $r : p(t_1, \dots, t_m) \leftarrow \bigwedge_{k=1}^n p^k(v_1^k, \dots, v_{m_k}^k)$  in  $\mathcal{P}$  and variable  $X$  in  $\text{head}(r)$ , 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  $(v_j^k = g(\dots, X, \dots) \wedge p^k[j]/gs \in \mathcal{U})$ , then
  - if  $t_i = f(\dots X \dots)$ , 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  $\mathcal{P}$ :

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

Although a program  $\mathcal{P}$  might have multiple supported m-sets, it is easy to see that there is a unique supported m-set that is minimal (with respect to 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:

$$\begin{aligned} -\Omega_{\mathcal{P}}^0 &= \{ b[i]/\epsilon \mid b[i] \in \text{arg}_b(\mathcal{P}) \}; \\ -\Omega_{\mathcal{P}}^{n+1} &= \Omega_{\mathcal{P}}^n \cup \{ p[i]/s \mid \exists r : p(t_1, \dots, t_m) \leftarrow \bigwedge_k p^k(v_1^k, \dots, v_{m_k}^k) \in \mathcal{P} \wedge \exists X \text{ in } t_i \wedge \exists s' \in F_{\mathcal{P}}^* \text{ subject to } \forall v_j^k, v_l^h \text{ containing } X, s = \omega(X, t_i, v_j^k, s', \Omega_{\mathcal{P}}^n) = \omega(X, t_i, v_l^h, s', \Omega_{\mathcal{P}}^n) \wedge s \neq \perp \}, \end{aligned}$$

where  $\omega$  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**  $(v_j^k = X \wedge p^k[j]/s' \in \Omega_{\mathcal{P}}^n) \vee (v_j^k = g(\dots X \dots) \wedge p^k[j]/gs' \in \Omega_{\mathcal{P}}^n)$  **then**  
**if**  $t_i = f(\dots X \dots)$  **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

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

The previous example shows that if  $M$  is the minimum model of our program, then the program's minimum supported m-set contains the m-set derived from  $M$ . In other words,  $\mathcal{U}_{\mathcal{P}}^*$  acts as an overapproximation for  $\mathcal{U}_M$ . We will show in the following that given a program  $\mathcal{P}$ , the preceding relation 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  $D^\epsilon = \{ b(\epsilon, \dots, \epsilon) \mid b \text{ is a base predicate of } \mathcal{P} \}$ .

$(\mathcal{U}_M \subseteq \mathcal{U}_{M'})$ . Let  $p[i]/s$  be a mapping in  $\mathcal{U}_M$ . If  $p[i] \in \text{arg}_b(\mathcal{P})$ , then  $s = \epsilon$ , by definition of  $\mathcal{U}_M$ . From the definition of  $D^\epsilon$ , it immediately follows that  $p[i]/\epsilon \in \mathcal{U}_{M'}$  as well. If  $p[i] \in \text{arg}_d(\mathcal{P})$ , then there must be a rule  $r : p(t_1, \dots, t_i, \dots, t_n) \leftarrow \text{body}(r)$  in  $\mathcal{P}$  and a substitution  $\theta$  for all variables occurring in  $r$  such that all atoms in  $\text{body}(r)\theta$  appear in  $M$  (this follows from the definition of the minimum model of  $\mathcal{P}$ ). Let  $\theta'$  be the substitution obtained from  $\theta$  by replacing all constants occurring in  $\theta$  with  $\epsilon$ . Since  $\mathcal{P}$  is a positive, normal program, if the joins in  $\text{body}(r)\theta$  are satisfied, then the joins in  $\text{body}(r)\theta'$  are satisfied as well. Thus, the atoms in  $\text{body}(r)\theta'$  and  $\text{head}(r)\theta'$  must occur in  $M'$ . Then, by construction of  $\theta'$ , 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  $M_n = T_{\mathcal{P} \cup D^\epsilon}^n(\emptyset)$  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  $b[i] \in \text{arg}_b(\mathcal{P})$ . Thus, 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  $T_{\mathcal{P} \cup \mathcal{D}^\epsilon}$ , there must be a rule  $r : p(t_1, \dots, t_i, \dots, t_n) \leftarrow \bigwedge_k q^k(v_1^k, \dots, v_{m_k}^k)$  in  $\mathcal{P}$  and a substitution  $\theta$  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  $r' = r\theta = p(t_1\theta, \dots, t_i\theta, \dots, t_n\theta) \leftarrow \bigwedge_k q^k(v_1^k\theta, \dots, v_{m_k}^k\theta)$  be the ground rule obtained by applying  $\theta$  to  $r$ . Recall that  $s \in str(t_i\theta)$ . Furthermore, by the inductive hypothesis, since each atom in  $body(r)\theta$  belongs to  $M_{n-1}$ , we have that for all  $k, j$  and every  $s' \in str(v_j^k\theta)$ ,  $q^k[j]/s' \in \mathcal{U}_{\mathcal{P}}^*$ . 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  $\theta$  replaces each variable with the same ground term. Let us now consider the set  $S$  of all mappings  $q^k[j]/s' \in \mathcal{U}_{M_{n-1}}$  such that  $s' \in str(v_j^k\theta)$  is obtained by the concatenation of the function symbols in  $v_j^k\theta$  up to a position where  $X$  appears. By the inductive hypothesis,  $S \subseteq \mathcal{U}_{\mathcal{P}}^*$ . Furthermore, the mappings in  $S$  satisfy Item 2 of Definition 4.4 with respect 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  $\mathcal{U}_{\mathcal{P}}^*$ .  $\square$

From the preceding lemma, 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 preceding discussion, 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  $p[i] \in arg(\mathcal{P})$ ,*

- (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), because when  $\mathcal{P} \in \mathcal{MR}$ ,  $MR(\mathcal{P}) = arg(\mathcal{P})$  and every argument of  $\mathcal{P}$  is limited.  $\square$

### 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 us to reduce the termination problem of  $\mathcal{P}$  to 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  $b[i] \in \text{arg}_b(\mathcal{P})$ ,  $\mathcal{P}^u$  contains a fact  $b_i(\epsilon)$ ;
- (2) for every rule  $r = p(t_1, \dots, t_n) \leftarrow \text{body}(r)$  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:

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

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 following program  $\mathcal{P}$  (on the left side), where  $b$  is a base predicate, and the corresponding unary program  $\mathcal{P}^u$  (on the right side), where rules  $\rho_i$  and  $\rho'_i$  are derived from  $r_i$  for  $i \in \{1, 2, 3\}$ :

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

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

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

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

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

The preceding example 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  $\mathcal{P}^u$ . Thus, 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  $h : \text{arg}(\mathcal{P}) \rightarrow \text{arg}(\mathcal{P}^u)$  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  $p[i] \in \text{arg}(\mathcal{P})$  iff  $p_i[1] \in \text{arg}(\mathcal{P}^u)$ . Consider now the bijection  $h$  such that  $h(p[i]) = p_i[1]$ , for every  $p[i] \in \text{arg}(\mathcal{P})$ . 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  $M = T_{\mathcal{P}^u}^\infty(\emptyset)$ ).

**Base ( $n = 1$ ):** Let  $M' = T_{\mathcal{P}^u}^1(\emptyset) = \{b_i(\epsilon) \mid b[i] \in \text{arg}_b(\mathcal{P})\}$ . By construction of  $\mathcal{U}_{M'}$ , we have that  $b_i[1]/\epsilon \in \mathcal{U}_{M'}$ , for every  $b[i] \in \text{arg}_b(\mathcal{P})$ . Recalling that  $b_i[1] = h(b[i])$  and that  $b[i]/\epsilon \in \mathcal{U}_{\mathcal{P}}^*$ , for each  $b[i] \in \text{arg}_b(\mathcal{P})$ , the claim follows.

**Inductive case ( $n > 1$ ):** Let  $M^{n-1} = T_{\mathcal{P}^u}^{n-1}(\emptyset)$ , and assume by the inductive hypothesis that for each  $p[i] \in \text{arg}(\mathcal{P})$  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  $M' = T_{\mathcal{P}^u}^n(\emptyset) = T_{\mathcal{P}^u}(M^{n-1})$ . Now, let  $p[i] \in \text{arg}(\mathcal{P})$  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 the inductive hypothesis. If  $h(p[i])/s \in (\mathcal{U}_{M'} \setminus \mathcal{U}_{M^{n-1}})$ , then there must exist 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 the inductive hypothesis, if  $h(q[j])/str(t) \in \mathcal{U}_{M^{n-1}}$ , then  $q[j]/str(t) \in \mathcal{U}_{\mathcal{P}}^*$ . However, 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  $p[i]/s \in \Omega_{\mathcal{P}}^n$ , 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  $p[i]/s \in \Omega_{\mathcal{P}}^n \setminus \Omega_{\mathcal{P}}^{n-1}$ , with  $n > 0$ . By definition of  $\Omega_{\mathcal{P}}$ , there must be a rule in  $\mathcal{P}$  of the form  $r = p(t_1, \dots, t_m) \leftarrow \bigwedge_k p^k(v_1^k, \dots, v_{m_k}^k)$ , a variable  $X$  in  $t_i$ , and a string  $s'$  such that  $\forall v_j^k, v_l^h$  containing  $X$ ,  $s = \omega(X, t_i, v_j^k, s', \Omega_{\mathcal{P}}^{n-1}) = \omega(X, t_i, v_l^h, s', \Omega_{\mathcal{P}}^{n-1})$  and  $s \neq \perp$ . Looking at function  $\omega$ , the condition

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

can be rewritten by the inductive hypothesis as follows:

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

However, 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  $\omega$  is indeed such that  $p_i[1]/s \in \mathcal{U}_M$ . In other words,  $h(p[i])/s \in \mathcal{U}_M$ .  $\square$

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 to 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 that 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 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  $\mathcal{P}$ :

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

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

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

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  $f$  and  $g$ .

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 as well. Let  $\phi$  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  $s^k = f s^{k-1}$ , where  $s^0 = \epsilon$ . Let  $\mathcal{U}_{\mathcal{P}_f} = \{p[i]/s^k \mid p[i] \in \text{arg}(\mathcal{P}) \wedge 0 \leq k \leq \phi(p[i])\}$ . 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, and thus  $\phi(A^0[i]) \geq \phi(B^0[j]) + \text{dept}(X, A^i) - \text{dept}(X, B^j)$  holds, where  $X, A^i$ , and  $B^j$  are terms occurring in some rule  $r$ , according to Definition 3.1. Furthermore,  $\mathcal{P}_f$  has only one function symbol. Thus, from Item 2, if  $X$  has a mapping to a string  $s^k$  of length  $k$  in  $\text{body}(r)$ , then the mapping  $A^0[i]/s^h$  must belong to  $\mathcal{U}_{\mathcal{P}_f}$ , with  $h = k + \text{dept}(X, A^i) - \text{dept}(X, B^j)$ . By construction of  $\mathcal{U}_{\mathcal{P}_f}$ ,  $k \leq \phi(B^0[j])$ , which implies that  $\phi(A^0[i]) \geq k + \text{dept}(X, A^i) - \text{dept}(X, B^j)$  (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, as  $\phi$  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  $\mathcal{P} \in \mathcal{MR}$ .

To prove the strict inclusion, observe that the program from Example 4.14 is in  $\mathcal{MR}$  but not in  $\mathcal{AR}$ .  $\square$

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:

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

The program is not in  $\mathcal{AR}$ , as there is no argument ranking satisfying the conditions of Definition 3.1. Indeed, assume toward 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:

$$\begin{aligned} (r_2) \quad \phi(q[2]) &\geq \phi(p[2]) \\ (r_3) \quad \phi(s[2]) &> \phi(q[2]) \\ (r_4) \quad \phi(p[2]) &> \phi(s[2]). \end{aligned}$$

Yet combining the preceding inequalities, we obtain that  $\phi(p[2]) > \phi(s[2]) > \phi(q[2]) \geq \phi(p[2])$ . In other words,  $\phi(p[2]) > \phi(p[2])$ .

However, the program is in  $\mathcal{MR}$ , as the minimum model of the following unary program  $\mathcal{P}^u$  is finite:

$$\begin{aligned} \rho_0 &: b_1(\epsilon). \\ \rho_{1.1} &: p_1(X) \leftarrow b_1(X). \\ \rho_{1.2} &: p_2(f(X)) \leftarrow b_1(X). \\ \rho_{2.1} &: q_1(X) \leftarrow p_1(X), p_2(f(X)). \\ \rho_{2.2} &: q_2(f(X)) \leftarrow p_1(X), p_2(f(X)). \\ \rho_{3.1} &: s_1(X) \leftarrow q_1(X). \\ \rho_{3.2} &: s_2(f(Y)) \leftarrow q_2(Y). \\ \rho_{4.1} &: p_1(X) \leftarrow s_1(X). \\ \rho_{4.2} &: p_2(f(Y)) \leftarrow s_2(Y). \end{aligned} \quad \square$$

It is worth noting that the preceding proof 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 in this article, 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 (respectively, rule-bounded) program not belonging to  $\mathcal{MR}$ , and vice versa. Indeed, the program from Example 1.1

is mapping restricted, but neither bounded nor rule bounded, whereas the following program

$$\begin{aligned} p(f(X), Y) &\leftarrow b(X, Y). \\ p(X, f(Y)) &\leftarrow p(f(X), Y). \end{aligned}$$

is both bounded and rule-bounded but not mapping restricted.  $\square$

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  $\mathcal{MR}$  as well.

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  $A_1 \vee \dots \vee A_m \leftarrow body$  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, then the 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  $MM(\mathcal{P}^u)$  is finite is  $PSPACE$ -complete (respectively,  $EXPTIME$ -complete) when the original program contains at most one (respectively, more than one) function symbol. The lower bounds are shown by standard reductions from the halting problem of a  $PSPACE$  (respectively,  $APSPACE$ ) Turing machine, whereas the upper bounds are obtained by reductions to satisfiability of LTL (respectively, CTL) formulas. 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  $\mathcal{P}$  (respectively, an argument  $p[i]$ ) is mapping restricted is tantamount to the problem of deciding whether  $MM(\mathcal{P}^u)$  is finite (respectively,  $p_i[1]$  is limited in  $MM(\mathcal{P}^u)$ ).

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  $\mathcal{P}$  (respectively, database  $D$ ), denoted by  $size(\mathcal{P})$  (respectively,  $size(D)$ ), is defined as the total number of occurrences in  $\mathcal{P}$  (respectively,  $D$ ) 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 with respect to  $size(\mathcal{P})$ .*

**PROOF.** We start by showing that the number of facts and rules in  $\mathcal{P}^u$  is polynomial with respect to  $size(\mathcal{P})$ . Then we show that constructing each fact (respectively, 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 1. 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 1, 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.  $\square$

It is clear from Proposition 4.13 and Lemma 5.1 that the problem of checking whether a program (respectively, an argument  $p[i]$  of  $\mathcal{P}$ ) belongs to  $\mathcal{MR}$  (respectively,  $MR(\mathcal{P})$ ) and the problem of checking whether  $MM(\mathcal{P}^u)$  is finite (respectively,  $p_i[1]$  is limited in  $MM(\mathcal{P}^u)$ ) 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 LTL formulas. See the appendix for more details on LTL.

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

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

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

where  $p_0, \dots, p_n$  are propositions, and  $\zeta_k$ , 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  $\varphi_i$ , 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  $\varphi_i$  encodes the fact that for every state (“always” temporal operator  $G$ ), the specified implication, modeling the behavior of rule  $r_i$ , must hold.

Next we construct the LTL formula  $\varphi$ :

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

The preceding formula states that (i) at the initial state, all database propositions must be true; (ii) all formulas 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  $\varphi$  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  $\varphi$  as follows. Let  $S$  be an infinite sequence of states  $S = s_0, s_1, \dots$  and let  $\psi$  be the mapping function such that

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

It is not difficult to see that the preceding linear structure satisfies  $\varphi$ . Indeed, the depth of all base predicates is 0, and 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 formulas  $\varphi_i$  are satisfied. Finally, if  $d$  is the maximum depth of terms occurring in  $M$ , then 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  $i = 0, 1, \dots$  there always exists at least one ground atom of the form  $p(f^i(\epsilon))$  in  $M$ . In other words, 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  $\varphi$  must be such that for every state  $s$  in  $S$ ,  $\psi$  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 to satisfy the first condition of  $\varphi$ . 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 formulas  $\varphi_i$ , 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  $\varphi$ . Thus,  $\varphi$  is not satisfiable.  $\square$

From the proof of preceding theorem, 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  $\mathcal{MM}(\mathcal{P}^u)$ . In other words, we only need to change the last condition of formula  $\varphi$  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 preceding problem 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);  $Q$  is the set of states, where  $q_0, q_a \in Q$  denote the initial and the accepting states, respectively; and  $\delta$  is the transition function.

Let  $I = a_1, \dots, a_m$  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 without loss of generality 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 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 nonaccepting 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  $cell_i^\alpha(X)$ , stating that at configuration  $X$ , the  $i$ -th cell contains the symbol  $\alpha \in \{0, 1, \perp\}$ . Furthermore, we use atoms of the form  $count_1^{\alpha_1}(X), \dots, count_{n+1}^{\alpha_{n+1}}(X)$  as shorthands for atoms  $cell_{n+1}^{\alpha_1}(X), \dots, cell_{2n+1}^{\alpha_{n+1}}(X)$ . These atoms will represent the counter's bits. Finally, we use the atom  $head_i(X)$  to state that the head of  $TM$  is on the  $i$ -th cell at configuration  $X$  and the atom  $state^q(X)$  to denote that the state at configuration  $X$  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 that will construct the initial configuration of  $TM$ :

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

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

$$cell_i^{\alpha'}(\text{next}(X)) \wedge head_{i+1}(\text{next}(X)) \wedge state^{q'}(\text{next}(X)) \leftarrow state^q(X), head_i(X), cell_i^\alpha(X), count_i^0(X).$$

Such a rule simply states that if at the given configuration  $X$ , the state is  $q$ , the head is on the  $i$ -th cell, which contains symbol  $\alpha$ , and the counter has not reached its maximum value yet (its most significant bit is zero), then at the next configuration  $\text{next}(X)$  (here  $\text{next}$  is a unary function symbol), the  $i$ -th cell will contain the symbol  $\alpha'$ , the head will be on cell  $i + 1$ , and the new state will be  $q'$ .

Next, to propagate all the unmodified cell values at configuration  $X$  to configuration  $\text{next}(X)$ , we also construct the rules of the following form:

$$cell_j^c(\text{next}(X)) \leftarrow state^q(X), cell_i^\alpha(X), head_i(X), count_i^0(X), cell_j^c(X)$$

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  $X$  to configuration  $\text{next}(X)$  has to be

performed, where the content of the  $i$ -th cell is modified, then all other cells containing some value  $c$  at configuration  $X$ , (i.e.,  $\text{cell}_i^c(X)$ ) must still contain such a value at configuration  $\text{next}(X)$ .

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:

$$\text{accept}(X) \leftarrow \text{state}^{\text{qa}}(X), \text{head}_1(X), \text{count}_1^0(X).$$

Note that by construction, every predicate symbol, except for  $b$ , occurs in the head of some of the preceding rules. We thus define  $b$  as a base predicate symbol and all 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  $D = \{b(\epsilon)\}$  contains an *accept*-atom.

Now, we introduce the following additional rule:

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

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  $D = \{b(\epsilon)\}$  is finite. This will show that checking finiteness of  $MM(\mathcal{P} \cup D)$  is *coPSPACE*-hard. It is well-known that *coPSPACE* = *PSPACE*, and from the definition of *coPSPACE*-hardness, we show that checking finiteness of  $MM(\mathcal{P} \cup D)$  is *PSPACE*-hard.  $\square$

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 preceding 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 CTL formulas. Please see the appendix for details on syntax and semantics of CTL.

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

Let  $\text{Prop} = \Pi \cup F$  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  $\mathcal{P}^u$ .

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

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

where  $p_0, \dots, p_n$  are propositions from  $\Pi$ , and if  $t_i = f(X)$ , for  $1 \leq i \leq n$ , then  $\zeta_i$  corresponds to “EN.” and  $B_i = f$ , where  $f$  is a proposition from  $F$ . Otherwise,  $\zeta_i$  is the empty string and  $B_i = \text{true}$ .

Intuitively, the propositions  $p_k$  in  $\varphi_i$ , 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  $f(X)$ , which 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 modeling the ground term in  $A$  if each proposition modeling some ground body atom is associated to the state modeling its ground term. The CTL formula  $\varphi_i$  encodes the fact that in every state (i.e., independently of the value of  $X$ ), the specified implication, modeling the behavior of rule  $r_i$ , must hold.

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

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

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 preceding formula  $\varphi$  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  $t = \epsilon$ ,  $s^\epsilon$  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  $\psi : S \rightarrow 2^{\text{Prop}}$  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  $\varphi$  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 to satisfy the first condition of  $\varphi$ . 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  $\varphi^*$  and the formulas  $\varphi_i$ , 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  $AF.AG.(¬q)$ , occurring in  $\varphi$ . Thus,  $\varphi$  is not satisfiable.  $\square$

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

$$AF.AG.(¬q),$$

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 following problem can be easily obtained from the proof of Theorem 5.3. Indeed, it is well known that  $APSPACE = EXPTIME$ , where *APSPACE* denotes alternating *PSPACE*. Thus, it suffices to encode a *PSPACE* Turing machine via a logic program as already shown in the proof of Theorem 5.3, then implement the acceptance part specifically for *PSPACE* machines with alternation. However, for the sake of completeness, we give the full proof in the following.

**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  $\delta$  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 without loss of generality that every transition in  $\delta$  is of the form  $(q, \alpha) \rightarrow \{(q', \alpha', m'), (q'', \alpha'', m'')\}$ . In other words, the machine  $TM$  may only move its configuration to at most two configurations.

Let  $I = a_1, \dots, a_m$  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 without loss of generality 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 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 nonaccepting configuration or the counter reaches its maximum value (i.e., its first bit is set to 1), then the machine rejects input  $I$ , and 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  $\text{cell}_i^\alpha(X)$ , stating that at configuration  $X$ , the  $i$ -th cell contains the symbol  $\alpha \in \{0, 1, \perp\}$ . Furthermore, we use atoms of the form  $\text{count}_1^{\alpha_1}(X), \dots, \text{count}_{n+1}^{\alpha_{n+1}}(X)$  as shorthands for atoms  $\text{cell}_{n+1}^{\alpha_1}(X), \dots, \text{cell}_{2n+1}^{\alpha_{n+1}}(X)$ . These atoms will represent the counter's bits. Finally, we use the atom  $\text{head}_i(X)$  to state that the head of  $TM$  is on the  $i$ -th cell at configuration  $X$  and the atom  $\text{state}^q(X)$  to denote that the state at configuration  $X$  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 that will construct the initial configuration of  $TM$ :

$$\bigwedge_{i \in [1..m]} \text{cell}_i^{a_i}(X) \wedge \bigwedge_{i \in [m+1..n]} \text{cell}_i^\perp(X) \wedge \bigwedge_{i \in [1..n+1]} \text{count}_i^0(X) \wedge \text{state}^{q_0}(X) \wedge \text{head}_1(X) \leftarrow b(X),$$

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

$$\begin{aligned} \text{cell}_i^\beta(f_1(X)) \wedge \text{head}_{i+1}(f_1(X)) \wedge \text{state}^{q'}(f_1(X)) &\leftarrow \text{state}^q(X), \text{cell}_i^\alpha(X), \text{head}_i(X), \text{count}_1^0(X). \\ \text{cell}_i^\gamma(f_2(X)) \wedge \text{head}_{i-1}(f_2(X)) \wedge \text{state}^{q''}(f_2(X)) &\leftarrow \text{state}^q(X), \text{cell}_i^\alpha(X), \text{head}_i(X), \text{count}_1^0(X). \end{aligned}$$

stating that if at the given configuration  $X$ , the state is  $q$ , the head is on the  $i$ -th cell, which contains symbol  $\alpha$ , and the counter has not reached its maximum value yet, then the machine may move to one of the two allowed configurations  $f_1(X)$  and  $f_2(X)$ , where the  $i$ -th cell will contain symbol  $\beta$  (respectively,  $\gamma$ ), the head will be on cell  $i + 1$  (respectively,  $i - 1$ ), and the new state will be  $q'$  (respectively,  $q''$ ).

Next, to propagate all of the unmodified cell values at configuration  $X$  to the corresponding next configuration, we also construct the rules of the following form:

$$\begin{aligned} \text{cell}_j^c(f_1(X)) &\leftarrow \text{state}^q(X), \text{cell}_i^\alpha(X), \text{head}_i(X), \text{count}_1^0(X), \text{cell}_j^c(X). \\ \text{cell}_j^c(f_2(X)) &\leftarrow \text{state}^q(X), \text{cell}_i^\alpha(X), \text{head}_i(X), \text{count}_1^0(X), \text{cell}_j^c(X). \end{aligned}$$

for every  $1 \leq i \leq 2n + 1$  and for every other tape cell  $1 \leq j \leq 2n + 1$  such that  $j \neq i$  and for every value  $c \in \{0, 1, \perp\}$ . The rules essentially state that if a transition from configuration  $X$  to configuration  $f_1(X)$  (respectively, configuration  $f_2(X)$ ) has to be performed, where the content of the  $i$ -th cell is modified, then all other cells containing some value  $c$  at configuration  $X$  (i.e.,  $\text{cell}_j^c(X)$ ) must still contain such a value at configuration  $f_1(X)$  (respectively,  $f_2(X)$ ).

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  $X$  is an accepting configuration.

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

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

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

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

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

$$\begin{aligned} \text{accepting}(X) &\leftarrow \text{state}^e(X), \text{accepting}(f_1(X)). \\ \text{accepting}(X) &\leftarrow \text{state}^e(X), \text{accepting}(f_2(X)). \end{aligned}$$

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

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

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

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

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  $D = \{b(\epsilon)\}$  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.  $\square$

Additionally in the multiple function symbols case, the proof of the preceding theorem can be used to show *EXPTIME*-hardness for the problem of checking whether an argument (namely,  $\text{accept}[1]$ ) 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 Section 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 Section 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 Section 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 function 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  $b$ :

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

Starting from any database, it is impossible to derive an atom of the form  $p(f(t, t))$ , where  $t$  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 next:

$$\begin{aligned}
 \rho_0 & : b_1(\epsilon). \\
 \rho_1 & : q_1(g(X)) \leftarrow b_1(X). \\
 \rho_{2.1} & : p_1(f(X)) \leftarrow b_1(X). \\
 \rho_{2.2} & : p_1(f(Y)) \leftarrow q_1(Y). \\
 \rho_3 & : t_1(X) \leftarrow p_1(f(X)). \\
 \rho_4 & : t_1(g(X)) \leftarrow t_1(X).
 \end{aligned}$$

Unfortunately,  $\mathcal{P}^u$  does not consider the position of terms inside other terms, transforming rule  $r_3$  into  $\rho_3$ . The fact that argument  $p[1]$  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 that is able to overcome the limitation discussed previously. 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  $t^1, \dots, t^m$ . For instance, considering rule  $r_2 : p(f(X, Y)) \leftarrow b(X), q(Y)$  in Example 6.1, occurrences of variables in the head are denoted by  $X^1$  and  $Y^1$ , whereas body occurrences are denoted by  $X^2$  and  $Y^2$ . 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  $\varphi$  denotes either an atom or a term, is defined recursively as follows:

- $path'(\varphi, t^j) = \epsilon$ , if  $\varphi$  is a constant;
- $path'(\varphi, t^j) = \varphi$ , if  $\varphi$  is a variable;
- $path'(\varphi, t^j) = f_i(path'(t_i, t^j))$ , if  $\varphi = f(t_1, \dots, t_n)$  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  $r = p(f(X, g(c)), X) \leftarrow q(g(X))$ . Let  $X^1, X^2$ , and  $X^3$  be the three occurrences of  $X$  in  $r$  and  $c^1$  be the occurrence of  $c$ . We have that  $path(r, X^1) = p_1(f_1(X))$ ,  $path(r, X^2) = p_2(X)$ ,  $path(r, c^1) = p_1(f_2(g_1(\epsilon)))$ , and  $path(r, X^3) = q_1(g_1(X))$ .

By making use of the notion of path defined earlier, 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 us 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:

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

– 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:

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

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  $\mathcal{P}$ :

$$\begin{aligned} r_1 &: \text{reverse}(\text{cons}(\text{a}, \text{cons}(\text{b}, \text{cons}(\text{c}, \text{null}))), \text{null}). \\ r_2 &: \text{reverse}(\text{L}_1, \text{cons}(\text{X}, \text{L}_2)) \leftarrow \text{reverse}(\text{cons}(\text{X}, \text{L}_1), \text{L}_2). \end{aligned}$$

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

$$\begin{aligned} \rho_{1.1} &: \text{reverse}_1(\text{cons}_1(\epsilon)). \\ \rho_{1.2} &: \text{reverse}_1(\text{cons}_2(\text{cons}_1(\epsilon))). \\ \rho_{1.3} &: \text{reverse}_1(\text{cons}_2(\text{cons}_2(\text{cons}_1(\epsilon)))). \\ \rho_{1.4} &: \text{reverse}_1(\text{cons}_2(\text{cons}_2(\text{cons}_2(\epsilon)))). \\ \rho_{1.5} &: \text{reverse}_2(\epsilon). \\ \rho_{2.1} &: \text{reverse}_1(\text{L}_1) \leftarrow \text{reverse}_1(\text{cons}_2(\text{L}_1)). \\ \rho_{2.2} &: \text{reverse}_2(\text{cons}_1(\text{X})) \leftarrow \text{reverse}_1(\text{cons}_1(\text{X})). \\ \rho_{2.3} &: \text{reverse}_2(\text{cons}_2(\text{L}_2)) \leftarrow \text{reverse}_2(\text{L}_2). \end{aligned}$$

Rules of the form  $\rho_{i,j}$  model the corresponding rule  $r_i$  of  $\mathcal{P}$ . In particular,  $\rho_{1.1}$  through  $\rho_{1.5}$  model the presence of constants `a`, `b`, `c`, and `null` in the ground fact `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  $\rho_{2.1}$  through  $\rho_{2.3}$  model the derivation of variables `L1`, `X`, and `L2` 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:

$$\begin{aligned} \rho_1 &: \text{b}_1(\epsilon). \\ \rho_2 &: \text{q}_1(\text{g}_1(\text{X})) \leftarrow \text{b}_1(\text{X}). \\ \rho_3 &: \text{p}_1(\text{f}_1(\text{X})) \leftarrow \text{b}_1(\text{X}). \\ \rho_4 &: \text{p}_1(\text{f}_2(\text{X})) \leftarrow \text{q}_1(\text{X}). \\ \rho_5 &: \text{t}_1(\text{X}) \leftarrow \text{p}_1(\text{f}_1(\text{X})), \text{p}_1(\text{f}_2(\text{X})). \\ \rho_6 &: \text{t}_1(\text{g}_1(\text{X})) \leftarrow \text{t}_1(\text{X}). \end{aligned}$$

The minimum model of  $\mathcal{P}^v$  is  $M = \{\text{b}_1(\epsilon), \text{q}_1(\text{g}_1(\epsilon)), \text{p}_1(\text{f}_1(\epsilon)), \text{p}_1(\text{f}_2(\text{g}_1(\epsilon)))\}$ . 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  $MM(\mathcal{P}^v)$ .

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

We point out that although the preceding definition 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 believe 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  $p[i] \in arg(\mathcal{P})$ ,*

- (1)  $p[i] \in EMR(\mathcal{P})$  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 1. 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.,  $A \in T_{\mathcal{P}_D}^i(\emptyset)$ ), then for each constant's occurrence  $e^j$  in  $A$ , there is an atom  $path(A, e^j)$  in  $T_{\mathcal{P}^v}^i(\emptyset)$ . 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  $b[k] \in arg_b(\mathcal{P})$ . Suppose now that our statement is true on step  $i$ . We now show that it is true on step  $i + 1$  as well. Suppose that we derive an atom  $A$  by applying  $T_{\mathcal{P}_D}^{i+1}(\emptyset)$  by using rule  $r$  in  $\mathcal{P}$ . This means that all ground atoms in its body belong to  $T_{\mathcal{P}_D}^i(\emptyset)$ . Consequently,  $path(B, c^h)$  is true in  $T_{\mathcal{P}^v}^i(\emptyset)$  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  $T_{\mathcal{P}^v}^{i+1}(\emptyset)$  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  $T_{\mathcal{P}_D}^k(\emptyset)$ . Yet this implies that the corresponding atom  $path(A, e^j)$  has depth  $d$  in  $T_{\mathcal{P}^v}^k(\emptyset)$ . 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 defined previously. 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)  $MR(\mathcal{P}) \subseteq EMR(\mathcal{P})$ ;
- (2)  $\mathcal{P} \in MR \Rightarrow \mathcal{P} \in \mathcal{EMR}$ .

**PROOF.** Assuming constant-free programs, where the nesting level of terms is at most 1, the definition of  $\mathcal{P}^v$  can be simplified as follows:

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

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

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  $T_{\mathcal{P}^v}^k(\emptyset)$  on step  $k$ , there is a corresponding atom  $A'$ , obtained from  $A$  by deleting the subscripts from its function symbols, belonging to  $T_{\mathcal{P}^u}^k(\emptyset)$ .

- (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  $H = \text{path}(A, e^j)$ , where  $e^j$  is the occurrence of some constant in the  $i$ -th argument of  $A$ , having depth  $d$  in  $T_{\mathcal{P}^v}^k(\emptyset)$  on some step  $k$ . From the preceding observation, it follows that  $H' \in T_{\mathcal{P}^u}^k(\emptyset)$ , 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 show that even if a program  $\mathcal{P}$  is not recognized as em-restricted, the set of em-restricted arguments  $\mathcal{EMR}(\mathcal{P})$  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 subprograms  $\mathcal{P}_1^v, \dots, \mathcal{P}_n^v$  (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  $\mathcal{P}_k^v$ , all rules defining  $q$  are contained in a stratum  $\mathcal{P}_j^v$ , with  $j \leq k$ . Finally, a rule  $r$  of some stratum  $\mathcal{P}_k^v$  is an *exit rule* of  $\mathcal{P}_k^v$  if it is not recursive—that is, if  $\text{body}(r)$  contains only base predicates or predicates defined by rules occurring in strata  $\mathcal{P}_j^v$ , 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  $\mathcal{P}_1^v = \{\rho_{1.1}, \rho_{1.2}, \rho_{1.3}, \rho_{1.4}, \rho_{2.1}\}$  and  $\mathcal{P}_2^v = \{\rho_{1.5}, \rho_{2.2}, \rho_{2.3}\}$ , defining the predicate symbols  $\text{reverse}_1$  and  $\text{reverse}_2$ , respectively. Furthermore, all rules of  $\mathcal{P}_1^v$ , except for  $\rho_{2.1}$  are exit rules of  $\mathcal{P}_1^v$ , whereas rules  $\rho_{1.5}, \rho_{2.2}$  are the exit rules of  $\mathcal{P}_2^v$ .

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  $\mathcal{P}_1^v, \dots, \mathcal{P}_n^v$  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:

- $\rho$  is not ground and recursive, and
- for every atom  $B \in \text{body}(\rho)$  that is mutually recursive with  $\text{head}(\rho)$ ,  $\text{dept}(\text{head}(\rho)) < \text{dept}(B)$  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  $\text{reverse}[1]$  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  $\mathcal{P}_1^v, \dots, \mathcal{P}_n^v$  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  $\text{reverse}[2]$  of  $\mathcal{P}$  is steadily restricted. In fact, there exists the predicate  $\text{reverse}_1$ , which is defined in the finitely decreasing stratum  $\mathcal{P}_1^v$ . Furthermore, the exit rules of  $\mathcal{P}_2^v$  (i.e., where  $\text{reverse}_2$  is defined) are  $\rho_{1.5}$  and  $\rho_{2.2}$  and the only predicate occurring in their body is  $\text{reverse}_1$ , whose corresponding argument  $\text{reverse}[1]$  is em-restricted.

Observe that in the preceding example 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  $p(t_1, \dots, t_n)$  derived at step  $k$  using the immediate consequence operator—for instance,  $A \in T_{\mathcal{P}_D}^k(\emptyset)$  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  $\text{path}(A, e^j)$  in  $T_{\mathcal{P}^v}^k(\emptyset)$  and the depth of  $\text{path}(A, e^j)$  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  $T_{\mathcal{P}^v}^k(\emptyset)$  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  $T_{\mathcal{P}^v}^k(\emptyset)$  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  $A \in T_{\mathcal{P}^v}^k(\emptyset)$ :

- (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  $\Delta$  be the upper bound for term growth after the application of one rule in  $\mathcal{P}^v$ . Obviously,  $d$  and  $\Delta$  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 as well.
- (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 nonrecursive 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.  $\square$

As a remark, the definition of steadily restricted arguments along with Proposition 6.14 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 article to determine the termination of different practical programs.

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

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

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

### 6.3 Complexity Results

As stated previously, complexity results for the extended version of our technique can be obtained from results for the original criterion. In other words, 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 with respect to  $size(\mathcal{P})$ ). 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 with respect to  $size(\mathcal{P})$ .*

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 preceding result, consider the following. An atom (respectively, 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  $A = p(t_1, \dots, t_n)$ , 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 with respect to  $size(\mathcal{P})$ , and thus the number of rules in  $\mathcal{P}^v$  is polynomial with respect to  $size(\mathcal{P})$ . 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.  $\square$

In the following, to simplify the complexity analysis, we focus on constant-free programs with nesting level of terms not greater than 1. 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.

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 Theorems 5.2 and 5.3.  $\square$

The preceding proof 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 that 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 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  $f_1(X)$  (respectively,  $f_2(X)$ ) with  $f(X, \text{Dummy})$  (respectively,  $f(\text{Dummy}, X)$ ), where *Dummy* is a new variable not occurring in  $\mathcal{P}'$ . Next, we add an atom  $b(f(\text{Dummy}, \text{Dummy}))$  with base predicate symbol  $b$  to the body of each rule having the variable *Dummy* in its head. As an example, consider the rule of  $\mathcal{P}'$ :

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

The constructed rule using *Dummy* variables is

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

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

$$\begin{aligned} \text{accept}_1(f_1(X)) &\leftarrow \text{accept}_1(X). \\ \text{accept}_1(f_2(\text{Dummy})) &\leftarrow b_1(f_1(\text{Dummy})), b_1(f_2(\text{Dummy})). \end{aligned}$$

The construction described previously 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 *Dummy* (i.e., *Dummy*-rules). Observe that *Dummy*-rules cannot be activated since their body atoms  $b_1(f_1(\text{Dummy})), b_1(f_2(\text{Dummy}))$  contain complex terms inside the base predicate symbol  $b_1$ , but databases may contain only flat atoms. Consequently,  $\mathcal{P}^v$  simulates the computation of an *EXPTIME* Turing machine as well as  $\mathcal{P}'$ .  $\square$

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. Thus, 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.  $\square$

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

## 7 CONCLUSIONS

In this article, 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 that 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 to implement the object model in logics, one unary function symbol is sufficient.

## APPENDIX

### A TEMPORAL LOGICS

#### A.1 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 formulas are either propositions of *Prop* or, if  $\varphi$  and  $\xi$  are well-formed LTL formulas, then  $\neg\varphi, \varphi \vee \xi, \varphi \wedge \xi, \varphi \Rightarrow \xi, G.\varphi, F.\varphi$ , and  $N.\varphi$  are well-formed LTL formulas. The operators  $G, F$ , and  $N$  are the LTL temporal operators denoting the following:

- $G.\varphi$ : Formula  $\varphi$  always holds;
- $F.\varphi$ : Formula  $\varphi$  holds sometime in the future;
- $N.\varphi$ : Formula  $\varphi$  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  $S = s_0, s_1, \dots$  is a possibly infinite sequence of states (i.e., time instants) and  $\psi : \{s_0, s_1, \dots\} \rightarrow 2^{Prop}$  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  $S = s_0, \dots$ , a state  $s_i$  in  $S$  and LTL formulas  $\varphi, \xi$ , 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  $K, s_i \models \phi$  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  $\varphi$  is satisfiable if there exists a linear-time structure  $K = (S, \psi)$ , with  $S = s_0, s_1, \dots$ , 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

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 formulas are either propositions of *Prop* or if  $\varphi$  and  $\psi$  are well-formed CTL formulas, 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 formulas. We consider the CTL temporal operators *AG*, *AF*, *AN*, *EG*, *EF*, and *EN* intuitively denoting the following:

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

Models of CTL formulas 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—for instance, for every state  $s \in S$ , there is a state  $s' \in S$  such that  $sRs'$ . Finally,  $\psi : S \rightarrow 2^{Prop}$  is a function mapping every state to a set of propositions. A *path* in a transition system  $T$  is an infinite sequence  $\pi = \pi_1, \pi_2, \dots$ , 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 formulas  $\varphi$  and  $\xi$ , 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  $T, s \models \phi$  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  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_i \models \varphi$ , for all  $i > 0$ ;
- $T, s \models AF.\varphi$  iff for all paths  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_i \models \varphi$ , for some  $i > 0$ ;
- $T, s \models AN.\varphi$  iff for all paths  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_2 \models \varphi$ ;
- $T, s \models EG.\varphi$  iff there exists a path  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_i \models \varphi$ , for all  $i > 0$ ;
- $T, s \models EF.\varphi$  iff there exists a path  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_i \models \varphi$ , for some  $i > 0$ ;
- $T, s \models EN.\varphi$  iff there exists a path  $\pi = \pi_1, \pi_2, \dots$  with  $\pi_1 = s$ , holds  $T, \pi_2 \models \varphi$ .

We say that a CTL formula  $\varphi$  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:

$$\begin{aligned} -str(t, u^j) &= \epsilon \text{ if } t = u, \\ -str(t, u^j) &= f_i \cdot s \text{ if } t = f(v_i, \dots, v_n) \text{ and } u^j \text{ occurs in } v_i \text{ and } str(v_i, u^j) = s. \end{aligned}$$

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  $\mathcal{P}$ ,  $A = p(t_1, \dots, t_n)$  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. an em-set  $\mathcal{V}_{\mathcal{P}}$  and denote by  $\langle \mathcal{V}_{\mathcal{P}}, \mathcal{A}, X^j \rangle \rightsquigarrow \int$  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  $\langle \mathcal{V}_{\mathcal{P}}, \mathcal{B}, j \rangle \rightsquigarrow \epsilon$  holds, then
  - $\langle \mathcal{V}_{\mathcal{P}}, head(r), t \rangle \rightsquigarrow \epsilon$  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 of its arguments are em-restricted.

### B.1 Example 6.15

Consider the program of Example 6.15:

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

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

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

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

$$\begin{aligned} \rho_{1.1} &: \text{merge}_1(cons_1(\epsilon)). \\ \rho_{1.2} &: \text{merge}_1(cons_2(cons_1(\epsilon))). \\ \rho_{1.3} &: \text{merge}_1(cons_2(cons_2(\epsilon))). \\ \rho_{1.4} &: \text{merge}_2(cons_1(\epsilon)). \end{aligned}$$

$$\begin{aligned}
\rho_{1.5} & \text{merge}_2(\text{cons}_2(\text{cons}_1(\epsilon))). \\
\rho_{1.6} & \text{merge}_2(\text{cons}_2(\text{cons}_2(\text{cons}_1(\epsilon)))). \\
\rho_{1.7} & \text{merge}_2(\text{cons}_2(\text{cons}_2(\text{cons}_2(\epsilon)))). \\
\rho_{1.8} & \text{merge}_3(\epsilon). \\
\rho_{2.1} & \text{merge}_1(L_1) \quad \leftarrow \text{merge}_1(\text{cons}_2(L_1)). \\
\rho_{2.2} & \text{merge}_2(L_2) \quad \leftarrow \text{merge}_2(\text{cons}_2(L_2)). \\
\rho_{2.3} & \text{merge}_3(\text{cons}_1(X)) \quad \leftarrow \text{merge}_1(\text{cons}_1(X)). \\
\rho_{2.4} & \text{merge}_3(\text{cons}_2(\text{cons}_1(Y))) \quad \leftarrow \text{merge}_2(\text{cons}_1(Y)). \\
\rho_{2.5} & \text{merge}_3(\text{cons}_2(\text{cons}_2(L_3))) \quad \leftarrow \text{merge}_3(L_3). \\
\rho_{3.1} & \text{merge}_2(L_2) \quad \leftarrow \text{merge}_2(\text{cons}_2(L_2)), \text{merge}_1(\epsilon). \\
\rho_{3.2} & \text{merge}_3(\text{cons}_1(Y)) \quad \leftarrow \text{merge}_2(\text{cons}_1(Y)), \text{merge}_1(\epsilon). \\
\rho_{3.3} & \text{merge}_3(\text{cons}_2(L_3)) \quad \leftarrow \text{merge}_3(L_3), \text{merge}_1(\epsilon). \\
\rho_{4.1} & \text{merge}_1(L_1) \quad \leftarrow \text{merge}_1(\text{cons}_2(L_1)), \text{merge}_2(\epsilon). \\
\rho_{4.2} & \text{merge}_3(\text{cons}_1(X)) \quad \leftarrow \text{merge}_1(\text{cons}_1(X)), \text{merge}_2(\epsilon). \\
\rho_{4.3} & \text{merge}_3(\text{cons}_2(L_3)) \quad \leftarrow \text{merge}_3(L_3), \text{merge}_2(\epsilon).
\end{aligned}$$

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 1. 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.<sup>4</sup> 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 following algorithm, we shall use substitutions consisting of pairs *variable/term* to remember the (fresh) variables used to replace (complex) terms. The application of a substitution  $\theta = \{U_1/u_1, \dots, U_k/u_k\}$  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**

$F := S$ ;

$\theta := \emptyset$ ;

**foreach** atom  $p(t_1, \dots, t_n) \in S$  and each  $t_i = f(u_1, \dots, u_k)$  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  $u_j$ ;

<sup>4</sup>The equivalence between the rewritten program and the source program can be guaranteed by adding to the database a fact  $b_c(c)$ .

```

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.

**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  $c$  occurring 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  $X_1, \dots, X_m$  be the variables occurring in  $F$ ;

Let  $C = p'(X_1, \dots, X_m)$  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  $X_1, \dots, X_m$  be the variables occurring in  $F$ ;

Let  $C = p'(X_1, \dots, X_m)$  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:

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

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

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

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

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

*First phase (head flattening).*

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

$$r_1 : p(f_1(A), f_2(B, Z), X_c) \leftarrow p_1(A, B, Z, X_c).$$

$$r_2 : p_1(f_2(X, Y), f_2(f_1(X), Y), Z, X_c) \leftarrow q(f_1(f_1(X)), f_2(f_2(f_1(Z), Y), X)), b_c(X_c).$$

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

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

*Second phase (body flattening).*

Step 4: Selecting  $r_4$ , we have that  $\theta = \{D/f_1(X), E/f_2(f_1(Z), Y)\}$ . In  $Flat(r)$ , rule  $r_4$  is replaced by the following rules:

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

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

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

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

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

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  $\max\{0, dept(head(r)) - 1\} + \max\{0, dept(body(r)) - 1\}$ .

## ACKNOWLEDGMENTS

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

## REFERENCES

- Sabrina Baselice, Piero A. Bonatti, and Giovanni Crisculo. 2009. On finitely recursive programs. *Theory and Practice of Logic Programming* 9, 2, 213–238.
- Catriel Beeri and Moshe Y. Vardi. 1984. A proof procedure for data dependencies. *Journal of the ACM* 31, 4, 718–741.
- Piero A. Bonatti. 2004. Reasoning with infinite stable models. *Artificial Intelligence* 156, 1, 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, 10.
- Marco Calautti, Georg Gottlob, and Andreas Pieris. 2015. Chase termination for guarded existential rules. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems (PODS'15)*. 91–103.
- Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2015a. Logic program termination analysis using atom sizes. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*. 2833–2839.
- Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2015b. Rewriting-based check of chase termination. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*.
- Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2016a. Using linear constraints for logic program termination analysis. *Theory and Practice of Logic Programming* 16, 3, 353–377.
- Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2016b. Exploiting equality generating dependencies in checking chase termination. *Proceedings of the VLDB Endowment* 9, 5, 396–407.
- Marco Calautti, Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2015c. Checking termination of bottom-up evaluation of logic programs with function symbols. *Theory and Practice of Logic Programming* 15, 6, 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, New York, NY, 239–250.



- Andrea Cali, Georg Gottlob, and Michael Kifer. 2013. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research* 48, 115–174.
- Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. 2008. Computable functions in ASP: Theory and implementation. In *Proceedings of the International Conference on Logic Programming*. 407–424.
- Luciano Caroprese, Irina Trubitsyna, and Ester Zumpano. 2007. Implementing prioritized reasoning in logic programming. In *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS'07)*, Vol. AIDSS. 94–100.
- Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. 1984. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences* 28, 1, 29–59.
- Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. 2005. Testing for termination with monotonicity constraints. In *Proceedings of the International Conference on Logic Programming*. 326–340.
- Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. 2008. The chase revisited. In *Proceedings of the 27th ACM Symposium on Principles of Database Systems (PODS'08)*. 149–158.
- Alin Deutsch and Val Tannen. 2003. MARS: A system for publishing XML from mixed and redundant storage. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'03)*. 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, 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, 75–93.
- Martin Gebser, Torsten Schaub, and Sven Thiele. 2007. GrinGo: A new grounder for answer set programming. In *Proceedings of the 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. *Journal of the ACM* 38, 3, 620–650.
- Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming. In *Proceedings of the 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, 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 *Proceedings 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 *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP'14)*. 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* 47, 741–808.
- Sergio Greco, Cristian Molinaro, and Francesca Spezzano. 2012a. *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 *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'13)*.
- 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, 737–752.
- Sergio Greco and Francesca Spezzano. 2010. Chase termination: A constraints rewriting approach. *Proceedings of the VLDB Endowment* 3, 1, 93–104.
- Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2011. Stratification criteria and rewriting techniques for checking chase termination. *Proceedings of the VLDB Endowment* 4, 11, 1158–1168.
- Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2012b. On the termination of logic programs with function symbols. In *Proceedings 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, 621–635.
- Sergio Greco, Irina Trubitsyna, and Ester Zumpano. 2007. On the semantics of logic programs with preferences. *Journal of Artificial Intelligence Research* 30, 501–523.
- Yuliya Lierler and Vladimir Lifschitz. 2009. One more decidable class of finitely ground programs. In *Proceedings of the International Conference on Logic Programming*. 489–493.
- Massimo Marchiori. 1996. Proving existential termination of normal logic programs. In *Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science, Vol. 1101. Springer, 375–390.
- Bruno Marnette. 2009. Generalized schema-mappings: From termination to tractability. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'09)*. 13–22.

- Arne Meier, Martin Mundhenk, Michael Thomas, and Heribert Vollmer. 2008. The complexity of satisfiability for fragments of CTL and CTL<sup>\*</sup>. *Electronic Notes on Theoretical Computer Science* 223, 201–213.
- Michael Meier, Michael Schmidt, and Georg Lausen. 2009. On chase termination beyond stratification. arXiv:0906.4228.
- Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. 2007. Termination analysis of logic programs based on dependency graphs. In *Proceedings 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, 33–63.
- Naoki Nishida and Germán Vidal. 2010. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing* 21, 3, 177–225.
- Enno Ohlebusch. 2001. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing* 12, 1–2, 73–116.
- Adrian Onet. 2013. The chase procedure and its applications in data exchange. In *Data Exchange, Integration, and Streams*, P. G. Kolaitis, M. Lenzerini, and N. Schwiekardt (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1–37.
- Peter Schneider-Kamp, Jürgen Giesl, and Manh Thang Nguyen. 2009a. The dependency triple framework for termination of logic programs. In *Proceedings of the 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, Article No. 2.
- 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, 365–381.
- Danny De Schreye and Stefaan Decorte. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming* 19–20, 1, 199–260.
- Alexander Serebrenik and Danny De Schreye. 2005. On termination of meta-programs. *Theory and Practice of Logic Programming* 5, 3, 355–390.
- A. Prasad Sistla and Edmund M. Clarke. 1985. The complexity of propositional linear temporal logics. *Journal of the ACM* 32, 3, 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* 29, 353–389.
- Tommi Syrjanen. 2001. Omega-restricted logic programs. In *Proceedings 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, 521–536.

Received June 2016; revised June 2017; accepted September 2017