Bottom-up Evaluation of Finitely Recursive Queries

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone

Department of Mathematics, University of Calabria, I-87036 Rende (CS), Italy
e-mail: {calimeri, cozza, ianni, leone}@mat.unical.it

Abstract. The support for function symbols in logic programming under Answer Set Programming semantics (ASP) allows to overcome some modeling limitations of traditional ASP systems, such as the inability of handling infinite domains. On the other hand, admitting function symbols in ASP makes inference undecidable in the general case. Thus, the research is lately focusing on finding proper subclasses of ASP programs with functions for which decidability of inference is guaranteed. The two major proposals so far, finitary programs and finitely-ground programs, are complementary, to some extent; indeed, the former are conceived for allowing decidable querying (using a top-down evaluation strategy), while the latter for allowing finite model computation (using a bottom-up evaluation strategy). One of the main advantages of finitely-ground programs is that they can be directly evaluated by current ASP systems. However, many programs lie outside this class, such as, in general, positive finitary programs. Indeed, ground queries over such programs can be easily answered by means of top-down techniques; bottom-up approaches, instead, are, in general, unsuitable. In this work we present a proper adaptation of the magic-sets technique that aims at making query answering over positive finitary (normal) programs feasible by means of bottom-up techniques, i.e., those all current ASP systems rely on.

1 Introduction

Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP) [1–5], evolved significantly during the last decade, and has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. Lately, the ASP community has clearly perceived the strong need to extend ASP by functions, and many relevant contributions have been done in this direction [6–11]. Supporting function symbols allows to overcome one of the major limitation of traditional ASP systems, i.e. the ability of handling finite sets of constants only. On the other hand, admitting function symbols in ASP makes the common inference tasks undecidable or not computable.

Finitary programs [11] is a class of logic programs that allows function symbols yet preserving decidability of ground querying by imposing restrictions both on recursion and on the number of potential sources of inconsistency. Recursion is restricted by requiring each ground atom to depend on finitely many ground atoms; such programs are called *finitely recursive*. Moreover, the number of odd-cycles (cycles of recursive calls involving an odd number of negative subgoals) is required to be finite, so restricting the potential source of inconsistency. Thanks to these two restrictions, consistency checking and ground queries are decidable while nonground queries are semidecidable.

The class of *finitely-ground* ($\mathcal{FG}$) programs [6], recently proposed, can be seen as a "dual" notion of the class of finitary programs. Indeed, while the latter is suitable for a top-down evaluation, the former allows a bottom-up computation. Basically, for each program P in this class, there exists a finite subset P' of its instantiation, called *intelligent instantiation*, having precisely the same answer sets as P. Importantly, such a subset P' is computable for $\mathcal{FG}$ programs. Both finitary programs and finitely-ground programs can express any computable function, and preserve decidability for ground queries. However, answer sets and non-ground queries are computable on finitely-ground programs, while they are not computable on finitary programs. Furthermore, the bottom-up nature of the notion of finitely-ground programs allows an immediate implementation in ASP systems (as ASP instantiators are based on a bottom-up computational model). Indeed, the DLV system [12], for instance, has already been adapted to deal with finitely-ground program by properly extending its instantiator [13].

The two above mentioned classed are not comparable. In particular, the class of finitely-ground programs does not include programs for which ground querying could be trivially decided by following a top-down approach, i.e., positive finitely recursive programs. These are some of the simplest finitary programs; still, they are not, in general, suited for a bottom-up evaluation. However, this class is of clear interest; indeed, it includes many significative programs [1], such as most of the standard predicates on lists. For instance, the following program, performing the check for membership of an element in a list, is finitely recursive and positive.

$$\begin{array}{l} member(X, [X|Y]). \\ member(X, [Y|Z]) \text{ :- } member(X, Z). \end{array} \tag{1}$$

This work aims at overcoming this shortage by finding a strategy that allows a bottom-up evaluation of queries on positive finitely recursive normal logic programs. It is worth noting that a working translation module, actually not considering the peculiar optimization techniques herein presented for finitely recursive queries, is described in [14].

The main contributions of this work can be summarized as follows.

- We design a suitable adaptation of the *magic-sets* rewriting technique for programs with functions.
- We define the class of finitely recursive queries as the queries that can be answered taking into account only a finitely recursive fragment of a program.
- Given a positive finitely recursive program P and a (ground) query Q, let $RW(Q, P)$ denote the logic program obtained by our magic-sets rewriting technique. Then, we present the following main results:
  - $P \models Q$ if and only if $RW(Q, P) \models Q$;
  - given a finitely-recursive query Q on a program P, the size of $RW(Q, P)$ is linear in the size of the input program;
  - given a ground query Q on a program P, if Q is finitely-recursive on P, then $RW(Q, P)$ is finitely-ground.

Thus, the herein presented technique paves the way for the evaluation of finitely recursive queries by the grounder of existing ASP systems, such as DLV and Smodels [15], simply by applying a light-weight rewriting on the (non-ground) input program.

The remainder of the paper is structured as follows. Section 2 motivates our work by means of few significative examples; for the sake of completeness, in Section 3 we report some needed preliminaries; Section 4 illustrates our adaptation of the magic-sets rewriting technique to the class of positive finitary programs; in Section 5 we present a number of theoretical results of the rewritten programs; eventually, Section 6 draws our conclusions and depicts the future work.

[1] Note that, in general, the class of finitely recursive Horn programs strictly includes those Horn programs whose evaluation terminates under standard Prolog SLD resolution.

2 Motivation

Positive finitely-recursive programs might be seen as the simplest subclass of finitary programs. As finitary programs, they enjoy all nice properties of this class. In particular, consistency checking is decidable as well as reasoning with ground queries (while reasoning is semi-decidable in case of non ground queries). Unfortunately, even if a program P is finitely-recursive, it is not suited for the bottom-up evaluation for two main reasons:

1. A bottom-up evaluation of a finitely-recursive program would generate some new terms at each iteration, thus iterating for ever.

   Example 1. Consider the following program P_2 defining the natural numbers:

   $$\begin{array}{l} nat(0). \\ nat(s(X)) \text{ :- } nat(X). \end{array} \tag{2}$$

   The program is positive and finitely-recursive, so every ground query (such as for instance $nat(s(s(s(0))))?$) can be answered in a top-down fashion; but its bottom-up evaluation would iterate for ever, as, for any positive integer n, the n-th iteration would derive the new atom $nat(s^n(0))$.

2. Finitely-recursive programs do not enforce the range of an head variable to be restricted by a body occurrence (i.e., bottom-up safety is not required). A bottom-up evaluation of these unsafe rules would cause the derivation of non-ground facts; such a case is not admissible by present grounding algorithms.

Example 2. Consider the following program P_3, defining the reachability among vertices of a graph:

$$\begin{array}{l} reachable(X, X). \\ reachable(X, Y) \text{ :- } reachable(X, Z), arc(Z, Y). \end{array} \tag{3}$$

The program is positive and finitely-recursive; thus, any ground query can be computed top-down, while a bottom-up evaluation is unfortunately unfeasible: the first iteration would generate $\{reachable(X, X)\}$, representing an infinite set of atoms. In this case, $node(X)$ could be added to the body of the first rule, rendering safe the variable X and then making possible the program bottom-up evaluation. But, this is not always the case, as shown in the next example.

Example 3. The following program P_4 defines the comparison operator 'less than' between two natural numbers (the function symbol s represents the successor of a natural number):

$$\begin{array}{l} lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } lessThan(X, Y). \end{array} \tag{4}$$

The program is positive and finitely-recursive, thus any ground query can be easily answered top-down. For instance, the query $lessThan(s(0), s(s(0)))?$ results true, whereas the query $lessThan(s(s(0)), s(s(0)))?$ is false. Bottom-up evaluation of this program is unfeasible, since the first iteration would generate the set consisting of an infinite number of atoms having the form $\{lessThan(X, s(X))\}$.

3 Preliminaries

This section reports the formal specification of the ASP language with function symbols, followed by some basics on the magic-sets technique. The subclass of ASP programs herein considered are positive normal logic programs (i.e., disjunction and negation are not allowed).

3.1 Syntax

A *term* is either a *simple term* or a *functional term*.[2] A *simple term* is either a constant or a variable. If $t_1 \ldots t_n$ are terms and f is a function symbol (*functor*) of arity n, then $f(t_1, \ldots, t_n)$ is a *functional term*; $t_1 \ldots t_n$ are subterms $f(t_1, \ldots, t_n)$. The subterm relation is assumed to be reflexive and transitive, that is:

- each term is also a subterm of itself;
- if t_1 is a subterm of t_2 and t_2 is subterm of t_3 then t_1 is also a subterm of t_3.

Each predicate p has a fixed arity $k \geq 0$. If $t_1, \ldots, t_k$ are terms and p is a *predicate* of arity k, then $p(t_1, \ldots, t_k)$ is an *atom*. Let A be a set of atoms and p be a predicate. With small abuse of notation we say that $p \in A$ if there is some atom in A with predicate name p.

A *rule* r is of the form:

$$\alpha \;\text{:-}\; \beta_1, \cdots, \beta_n. \tag{5}$$

where $\alpha, \beta_1, \ldots, \beta_n$ are atoms.

Atom α is called *head* of r, while the conjunction $\beta_1, \cdots, \beta_n$, is the *body* of r. We denote the head atom by $H(r)$, and denote by $B(r)$ the set of body atoms. If the body of r is empty (i.e., $n = 0$ and then $B(r) = \emptyset$) we usually omit the " :- " sign; and if it contains no variables, then it is referred to as a *fact*.

An ASP program P is a finite set of rules. The ASP programs considered in this paper are also called *Horn programs*, as negation is forbidden. Horn programs where functional terms are not allowed are generally called *Datalog programs*.

Given a predicate p, a *defining rule* for p is a rule r such that the predicate p occurs in the head atom $H(r)$. If all defining rules of a predicate p are facts, then p is an EDB *predicate*; otherwise p is an IDB *predicate*.[3] The set of all facts of P is denoted by $Facts(P)$; the set of instances of all EDB predicates is denoted by $EDB(P)$ (note that $EDB(P) \subseteq Facts(P)$). The set of all head atoms in P is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$.

A *query* Q is an IDB atom.[4] As usual, a program (a rule, a term, a query) is said to be *ground* if it contains no variables.

3.2 Semantics

Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all (ground) terms that can be built combining constants and functors appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all ground atoms obtainable from the atoms of P by replacing variables with elements from U_P.[5] A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U_P of ground terms. A *ground instance* of a rule r is obtained applying a substitution to r. Given a program P the *instantiation (grounding)* $grnd(P)$ of P is defined as the set of all ground instances of its rules. Given a ground program P, an *interpretation* I for P is a subset of B_P. An atom a is true w.r.t. I if $a \in I$; it is false otherwise. Given a ground rule r,

[2] We will use traditional square-bracketed list constructors as shortcut for the representation of lists by means of nested functional terms (see, for instance, [6]). The usage "à la prolog", or any different, is only a matter of syntactic sugar.

[3] EDB and IDB stand for Extensional Database and Intensional Database, respectively.

[4] Note that this definition of a query is not as restrictive as it may seem, as one can include appropriate rules in the program for expressing unions of conjunctive queries (and more).

[5] With no loss of generality, we assume that constants appearing in any query Q always appear in P. Since we focus on query answering, this allows us to restrict to Herbrand universe/base.

we say that r is satisfied w.r.t. I if its head atom is true w.r.t. I or some body atom appearing in $B(r)$ is false w.r.t. I.

Given a ground program P, we say that I is a *model* of P, iff all rules in $grnd(P)$ are satisfied w.r.t. I. A model M of P is an *answer set* of P if there is no model N for P such that $N \subset M$. Each program P considered in this paper (positive and non-disjunctive) has a unique answer set, which is denoted by $AS(P)$.

3.3 The Magic-Sets Technique

The *magic-sets* method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query. Intuitively, the goal of the magic-sets method is to use the constants appearing in the query to reduce the size of the instantiation by eliminating 'a priori' a number of ground instances of the rules which cannot contribute to the (possible) derivation of the query goal.

This method has originally been defined in [16] for non-disjunctive Datalog (i.e., with no function symbols) queries only. Afterwards, many generalizations have been proposed. In the context of ASP, the generalization to the disjunctive case [17] and to Datalog with (possibly unstratified) negation [18] are worth remembering.

We next provide a brief and informal description of the magic-sets rewriting technique. The reader is referred to [19] for a detailed presentation. The method is structured in four main phases which are informally illustrated below by example, considering the query $path(1,5)$ on the following program P_6:

$$\begin{array}{l} path(X,Y) \;\text{:-}\; edge(X,Y). \\ path(X,Y) \;\text{:-}\; edge(X,Z), path(Z,Y). \end{array} \tag{6}$$

1. Adornment Step: The key idea is to materialize, by suitable adornments, binding information for IDB predicates which would be propagated during a top-down computation. Adornments are strings consisting of the letters *b* and *f*, denoting 'bound' and 'free' respectively, for each argument of an IDB predicate. First, adornments are created for query predicates so that an argument occurring in the query is adorned with the letter b if it is a constant, or with the letter f if it is a variable. The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. It is worth noting that adorning a rule may generate new adorned predicates. Thus, the adornment step is repeated until all adorned predicates have been processed, yielding the *adorned program*.

For simplicity of presentation, we next adopt the "basic" magic-sets method as defined in [16], in which binding information within a rule comes only from the adornment of the head predicate, from EDB predicates in the (positive) rule body, and from constants. In other words, an adornment of type 'b' is induced by a constant, or by a variable occurring either as an argument in a position of type b in the head predicate or in an EDB predicate. On the contrary, in the so-called "generalized" magic-sets method [20], bindings may also be generated by IDB predicates in rule bodies. In particular, an appropriate Sideways Information Passing Strategy (SIPS) has to be specified for each rule, fixing the body ordering and the way in which bindings are generated. In this respect, the basic method uses a particular, predetermined SIPS for all rules.

Example 4. Adorning the query $path(1,5)$ generates the adorned predicate $path^{bb}$ since both arguments are bound, and the adorned program P_7 is:

$$\begin{array}{l} path^{bb}(X,Y) \;\text{:-}\; edge(X,Y). \\ path^{bb}(X,Y) \;\text{:-}\; edge(X,Z), path^{bb}(Z,Y). \end{array} \tag{7}$$

2. Generation Step: The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme and single out the atoms which are relevant for deriving the input query. Let the *magic version* $\text{magic}(p^{\alpha}(\bar{t}))$ for an adorned atom $p^{\alpha}(\bar{t})$ be defined as the atom $magic\_p^{\alpha}(\bar{t}')$, where $\bar{t}'$ is obtained from $\bar{t}$ by eliminating all arguments corresponding to an f label in α, and where $magic\_p^{\alpha}$ is a new predicate symbol obtained by attaching the prefix '$magic\_$' to the predicate symbol p^{α}. Then, for each adorned atom A in the body of an adorned rule r_a, a magic rule r_m is generated such that (i) the head of r_m consists of $\text{magic}(A)$, and (ii) the body of r_m consists of the magic version of the head atom of r_a, followed by all the (EDB) atoms of r_a which can propagate the binding on A.

Example 5. Let us consider the adorned program P_7. First rule does not produce any magic rule, since it does not contain any adorned predicate in its body. Hence, we only generate the following magic rule:

$$magic\_path^{bb}(Z,Y) \text{ :- } magic\_path^{bb}(X,Y), edge(X,Z). \tag{8}$$

3. Modification Step: The adorned rules are subsequently modified by including magic atoms generated in Step 2 in the rule bodies, which limit the range of the head variables avoiding the inference of facts which cannot contribute to deriving the query. The resulting rules are called *modified rules*. Each adorned rule r_a is modified as follows. Let H be the head atom of r_a. Then, atom $\text{magic}(H)$ is inserted in the body of the rule, and the adornments of all non-magic predicates are stripped off.

Example 6. Rules resulting from the modification of the adorned program P_7 are:

$$\begin{array}{l} path(X,Y) \text{ :- } magic\_path^{bb}(X,Y), edge(X,Y). \\ path(X,Y) \text{ :- } magic\_path^{bb}(X,Y),\ edge(X,Z),\ path(Z,Y). \end{array} \tag{9}$$

4. Processing of the Query: Let the query goal be the adorned IDB atom g^{α}. Then, the magic version (also called *magic seed*) is produced as $\text{magic}(g^{\alpha})$ (see step 2 above). For instance, in our example we generate $magic\_path^{bb}(1,5)$.

The complete rewritten program consists of the magic, modified, and query rules.

Example 7. The complete rewriting of our example program is:

$$\begin{array}{l} magic\_path^{bb}(1,5). \\ magic\_path^{bb}(Z,Y) \text{ :- } magic\_path^{bb}(X,Y), edge(X,Z). \\ path(X,Y) \text{ :- } magic\_path^{bb}(X,Y), edge(X,Y). \\ path(X,Y) \text{ :- } magic\_path^{bb}(X,Y),\ edge(X,Z),\ path(Z,Y). \end{array} \tag{10}$$

In this rewriting, $magic\_path^{bb}(X,Y)$ represents the start- and end-nodes of all potential sub-paths of paths from 1 to 5. Therefore, when answering the query, only these sub-paths will be actually considered in bottom-up computations.

4 Rewriting Finitely-Recursive Queries

In this Section the definition of finitely-recursive queries is given first; then, a suitable adaptation of the *magic-sets* rewriting technique for programs with functions is presented, that allows a bottom-up evaluation of such queries over positive programs.

4.1 Finitely-Recursive Queries

Given a ground program P, we say that a ground atom a depends on another ground atom b if there is a rule $r \in grnd(P)$ such that a is the head of r and either $b \in B(r)$ or c depends on b for some atom $c \in B(r)$. A finitely-recursive program [11] is such that *every* ground query on it depends only on a finite set of other ground atoms.

Interestingly, even for some non finitely-recursive program, there may exist a subset of all possible ground queries for which the above mentioned property holds.

Example 8. Consider the following program P_{11}:

$$\begin{array}{l} lessThan(X, s(X)). \\ lessThan(X, s(Y)) \;\text{:-}\; lessThan(X, Y). \\ q(f(f(0))). \\ q(X) \;\text{:-}\; q(f(X)). \\ r(X) \;\text{:-}\; lessThan(X, Y), q(X). \end{array} \tag{11}$$

If the whole program is considered, then it is not finitely-recursive. In fact, atoms such as $q(c)$ or $r(c)$ (where c is a whatsoever constant) depend on an infinite number of other atoms. In the former case, because of a never terminating recursion; in the latter, because of both the local variable Y and the dependency on q atoms. Nevertheless, query atoms having as predicate $lessThan$ continue to depend only on a finite set of other atoms.

More formally, queries depending on a finite set of other atoms can be defined as follows.

Definition 1. Given a program P and a *ground query* Q over P we say that Q is *finitely-recursive* on P if and only if Q depends only on a finite set A of other ground atoms in $grnd(P)$.

In other words, we can say that a ground query Q on a program P is finitely-recursive if and only if the "relevant subprogram" of P for Q ($R_{(P,Q)}$) is positive and finitely-recursive. For instance, considering the program P_{11} of Example 8, all atoms like $lessThan(c_1, c_2)$, where c_1 and c_2 are constant values, are examples of finitely-recursive queries.

4.2 Rewriting Algorithm

We restrict our attention to finitely-recursive queries that cannot be safely bottom-up evaluated (it makes sense to think of a standard bottom-up evaluation for others). In such a case, some steps of the magic-sets technique reported in Section 3.3 can be significantly simplified. In particular, the adornment phase is no longer needed, given that the IDB predicates involved in the query evaluation would have a completely bound adornment. Indeed:

- the query is ground, so it would have a completely bound adornment;
- all rules involved in a top-down evaluation of the query cannot have local variables (i.e. variables appearing only in the body of the rule) since the relevant subprogram is supposed to be finitely-recursive.[6] Thus, starting from a ground query, a complete bound adornment from the head to all the IDB predicates of the body would be propagated.

In the generation step, it is no longer necessary to include any other atom in the body of the generated magic rule, apart from the magic version of the head atom. Again, this is due to the absence of local variables, so that all the needed bindings are provided through the magic version of the head atom.

[6] Indeed, function symbols make the universe infinite, and local variables in a rule would make its head depend on an infinite number of other ground atoms. Local variables could obviously appear in a function-free program, but this could be easily bottom-up evaluated.

```
Input: a program $P$ and a finitely-recursive query $Q = \mathtt{g}(\overline{\mathtt{c}})?$ on $P$
Output: the rewritten program $RW(Q,P)$.
Main Vars: $S$: stack of predicates to rewrite;
    $modifiedRules(Q,P)$, $magicRules(Q,P)$: set of rules;
    $Done$: set of predicates;
begin
 1. $modifiedRules(Q,P)$ := $\emptyset$; $Done$ := $\emptyset$
 2. $magicRules(Q,P)$ := $\{\mathtt{magic\_g}(\overline{\mathtt{c}}).\}$;
 3. $S$.push($\mathtt{g}$);
 4. while $S \neq \emptyset$ do
 5.     $u$ := $S$.pop();
 6.     if $u \notin Done$ then
 7.         $Done$ := $Done \cup \{u\}$;
 8.         for each $r \in P$ : $r$ is a defining rule for $u$ do
 9.             if $B(r) \neq \emptyset$ or $Vars(r) \neq \emptyset$ then
                    // let $r$ be $\mathtt{u}(\overline{\mathtt{t}})$ :- $\mathtt{v_1}(\overline{\mathtt{t_1}}), ..., \mathtt{v_n}(\overline{\mathtt{t_n}})$.
10.                 $modifiedRules(Q,P)$ := $modifiedRules(Q,P) \cup$
                          $\{\mathtt{u}(\overline{\mathtt{t}})$ :- $\mathtt{magic\_u}(\overline{\mathtt{t}}), \mathtt{v_1}(\overline{\mathtt{t_1}}), ..., \mathtt{v_n}(\overline{\mathtt{t_n}}).\}$;
11.                 for each $v_i$ : $v_i \in B(r)$ and $v_i \in IDB(P)$ do
12.                     $magicRules(Q,P)$ := $magicRules(Q,P) \cup$
                              $\{\mathtt{magic\_v_i}(\overline{\mathtt{t_i}})$ :- $\mathtt{magic\_u}(\overline{\mathtt{t}}).\}$;
13.                     $S$.push($v_i$);
14.                 end for
15.             else
16.                 $modifiedRules(Q,P)$ := $modifiedRules(Q,P) \cup r$
17.             end if
18.         end for
19.     end if
20. end while
21. $RW(Q,P)$:=$magicRules(Q,P) \cup modifiedRules(Q,P)$;
22. return $RW(Q,P)$;
end.
```

Fig. 1. Magic Sets rewriting algorithm for finitely-recursive queries

The algorithm MS_{FR} in Figure 1 implements the magic-sets method for finitely-recursive queries. Starting from a program P and a finitely-recursive ground query Q on P, the algorithm outputs a program $RW(Q,P)$ consisting of a set of *modified* and *magic* rules (denoted by $modifiedRules$ and $magicRules$, respectively), which are generated on a rule-by-rule basis. To this end, it exploits a stack S for storing all predicates that are still to be used for propagating the query binding. At first, the set of magic rules is initialized with the magic version of the query (line 2) and the predicate of the query atom is pushed on S (line 3). At each step, an element u is removed from S (line 5). If the predicate u has not been already considered (the auxiliary variable $Done$ is used to check this) (line 6), all the rules defining u are processed one-at-a-time (lines $8-18$). Given one of such rules r, if the body of r is not empty or there is at least a variable in r (line 9, where $Vars(r)$ is used to denote the set of variables occurring in the rule r), then a modified version of r is created (line 10) and a set of magic rules are generated (one for each IDB atom in the body) (lines $11-14$). Moreover, every IDB predicate appearing in the body of r is pushed on the stack S (line 13). In case r is a fact, i.e. its body is empty and there are no variables, it is added to the $modifiedRules$ set as it is (line 16). Finally, once all the predicates involved in the query evaluation have been processed (S is thus empty), the algorithm outputs the program $RW(Q, P)$ obtained as the union of all modified rules and generated magic rules (lines $21 - 22$).

Some rewriting example are reported next.

Example 9. Consider the finitely-recursive query $Q = nat(s(s(0)))$? on the program P_2 of Example 1. For this example, we will depict, step by step, the execution performed by the MS_{FR} algorithm. After the initialization of variables, the algorithm (lines $1-2$) generates the first magic rule deriving from the query:

$$magic\_nat(s(s(0))). \tag{12}$$

The predicate nat is then pushed onto the stack S (line 3) and the first iteration of the cycle (line 4) starts. The predicate nat is extracted from S and is marked as already done (lines $5 - 7$). In this case, this is the only predicate to be considered. All defining rules for nat are then processed (lines $8 - 18$). The first rule defining nat is a fact ($nat(0)$.): both conditions of line 9 are false, so the rule is added to the $ModifiedRules$ set (line 16) unchanged. The second rule defining nat is a recursive rule. First of all, the modified rule:

$$nat(s(X)) \text{ :- } magic\_nat(s(X)), nat(X). \tag{13}$$

is added to the $ModifiedRules$ set (line 10). Then, the following magic rule for the nat atom occurring in the body is generated, and the nat predicate is pushed onto the stack S (lines $11 - 14$):

$$magic\_nat(X) \text{ :- } magic\_nat(s(X)). \tag{14}$$

Then, the second iteration starts but it immediately ends, as the predicate extracted from the stack S is the already considered predicate nat. Finally, S is found empty, and there are no further iterations needed. The algorithm outputs the following complete rewritten program P_{15}=$RW(Q, P)$:

$$\begin{array}{l} magic\_nat(s(s(0))). \\ magic\_nat(X) \text{ :- } magic\_nat(s(X)). \\ nat(0). \\ nat(s(X)) \text{ :- } magic\_nat(s(X)), nat(X). \end{array} \tag{15}$$

Example 10. Considering the query $Q = lessThan(s(s(0)), s(0))$? on the program P_4 of Example 3, the algorithm outputs the following rewritten program P_{16}=$RW(Q, P)$:

$$\begin{array}{l} magic\_lessThan(s(s(0)), s(0)). \\ magic\_lessThan(X, Y) \text{ :- } magic\_lessThan(X, s(Y)). \\ lessThan(X, s(X)) \text{ :- } magic\_lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } magic\_lessThan(X, s(Y)), lessThan(X, Y). \end{array} \tag{16}$$

As we can see in the following, most of the common queries on programs for manipulating lists are finitely-recursive and then can be rewritten using the MS_{FR} algorithm.

Example 11. Let us consider the following program P_{17}, that works on lists:

$$\begin{array}{l} reverse(L, R) \text{ :- } sup\_reverse(L, [\,], R). \\ sup\_reverse([\,], R, R). \\ sup\_reverse([X|T_1], L, R) \text{ :- } sup\_reverse(T_1, [X|L], R). \end{array} \tag{17}$$

For the query $Q = reverse([a, b, c, d], [d, c, b, a])?$, the rewritten program P_{18}=$RW(Q, P)$ is:

$$\begin{array}{ll}
magic\_reverse([a, b, c, d], [d, c, b, a]). & \\
magic\_sup\_reverse(L, [\,], R) \text{ :- } magic\_reverse(L, R). & \\
magic\_sup\_reverse(T_1, [X|L], R) \text{ :- } magic\_sup\_reverse([X|T_1], L, R). & \\
reverse(L, R) \text{ :- } magic\_reverse(L, R), sup\_reverse(L, [\,], R). & (18) \\
sup\_reverse([\,], R, R) \text{ :- } magic\_sup\_reverse([\,], R, R). & \\
sup\_reverse([X|T_1], L, R) \text{ :- } magic\_sup\_reverse([X|T_1], L, R), & \\
\qquad\qquad sup\_reverse(T_1, [X|L], R) &
\end{array}$$

5 Properties of Rewritten Programs

Let $RW(Q, P)$ denote the output of the MS_{FR} algorithm, having as input a program P and a finitely-recursive query Q. Next, we are going to prove some relevant results about the $RW(Q, P)$ program. First of all we give a query equivalence property.

Theorem 1. Given a ground query Q on a program P, if Q is finitely-recursive on P, then $P \models Q$ if and only if $RW(Q, P) \models Q$.

Proof. (*Sketch*) Query equivalence has already been proved for the 'standard' magic-sets technique (see e.g. [19]). The algorithm presented in Section 4.2 differs from the standard one for some aspects but all of them have no consequences on the correctness of the transformation. We next recall the differences against the standard magic-sets technique, and illustrate why the introduced changes do not affect the query equivalence result:

1. Adornment is not performed because the structure of finitely-recursive queries implies that only a completely bound adornment would be derived. Anyway, all the IDB predicates involved in the top-down query evaluation are correctly identified and processed, likewise the standard algorithm does. Both rules modification and magic rules generation are performed considering all processed IDB predicate as having an implicit completely bound adornment.
2. Only the magic version of the head atom is included in the body of each generated magic rule. According to the standard technique, all EDB atoms which can propagate the binding on variables occurring in the currently processed atom should be added to the body. In case of a finitely-recursive query, all variables occurring in the body of a rule necessarily occurs also in its head (no local variables are admitted). Hence, we know 'a priori' that no further atom is needed.
3. The MS_{FR} algorithm acts on a rule-by-rule basis, instead of performing the different phases on the entire program. This approach, adopted also in [17] and [18], allows us to improve efficiency, as each rule of the original program is processed just once. The resulting rewritten program is not affected by this change.

Next, we are going to prove a result about the efficiency of the rewriting algorithm. To this aim, we need to introduce the definition of what we mean for size of a program.

Definition 2. Let P be a (non-ground) logic program. The *size* $\|t\|$ of a term t is 1, if the term is a constant or a variable; the size of a functional term $f(t_1, \ldots, t_n)$ is defined as $1{+}\|t_1\|{+}\ldots, \|t_n\|$. The *size of an atom* is given by the sum of the size of its terms; if the atom has arity 0, size is 1. The *size of the program* P, denoted by $\|P\|$, is the sum of the sizes of all atoms occurring in P. It is worth noting that multiple occurrences of the same atom in P are taken into account.

Example 12. The program P_{17} in Example 11 has size $\|P\| = 18$.

Theorem 2. Given a finitely-recursive query Q on a program P, the size of $RW(Q,P)$ is linear in the size of P and Q. In symbols: $\|RW(Q,P)\| = O(\|P\| + \|Q\|)$. Also, $MS_{FR}(Q,P)$ outputs $RW(Q,P)$ in time linear in the size of P.

Proof. (*Sketch*) The size of the rewritten program may increase w.r.t. the original one, because new atoms ("magic" atoms) and new rules ("magic" rules) are introduced. Nevertheless, a magic atom $magic\_p(\bar{t})$ might be added only as the counterpart of an atom $p(\bar{t})$ already existing in P; it is worth noting that they have exactly the same terms (and thus, the same size). Hence, it is enough to consider the number of atoms in $RW(Q,P)$ w.r.t. the number of atoms in P.

The program $RW(Q,P)$ is obtained as the union of the two sets of rules $modifiedRules(Q,P)$ and $magicRules(Q,P)$. In the worst case, the number of atoms in the first set is given by the number of atoms in P plus as many atoms as the number of rules in P (at most one magic atom is added for each rule in P). Thus, $\|modifiedRules(Q,P)\|$ is definitely $O(\|P\|)$.

Let us consider now the $magicRules(Q,P)$ program. At first, the magic version of the query is added. Then, for each IDB atom occurring in the body of a rule in P, at most one magic rule with exactly two atoms is generated. Then, in the worst case, the number of atoms in $magicRules(Q,P)$ is not greater than $2 \cdot \|P\| + \|Q\|$.

As for the time complexity of MS_{FR}, note that each rule $r \in P$ is processed exactly once. Processing r requires to scan, and process once, all the atoms appearing in it.

Thus, from the considerations above, the statements immediately follows.[7]

5.1 Relationships with Finitely-Ground Programs

We points out next the relationship between finitely-recursive queries and finitely-ground programs [6]. For the sake of clarity, we roughly recall here some key concepts therein introduced;[8] for formal definitions, more details, and examples, we refer the reader to the aforementioned paper.

Some graphs are defined, namely *Dependency Graph* and *Component Graph*, in order to properly split a given program P into modules, each one corresponding to a strongly connected component (SCC)[9] of the dependency graph. An ordering relation is then defined among modules/components: a *component ordering* γ for P is a total ordering such that the instantiation performed one module at a time according to γ is equivalent to the one obtained by processing the whole program altogether. An new operator (Φ) is defined, that properly instantiates a given module of P; such operator exploits the instantiation of previous modules (according to a component ordering) in order to generate a subset of the theoretical instantiation, by adding only those ground rules whose heads have a chance to be true in some answer set. By properly composing consecutive applications of the least fixpoint of Φ to the modules of P according to a component ordering $\gamma = \langle C_1, \ldots, C_n \rangle$, a sequence $S_0 \ldots S_n$ of sets of ground rules is generated, such that many useless rules w.r.t. answer sets computation are dropped. The *intelligent instantiation* P^γ of P for γ is the last element S_n of the sequence above. A program P is *finitely-ground* ($\mathcal{FG}$) if P^γ is finite, for every component ordering γ for P.

We are now ready to introduce an important property of the rewritten programs resulting from the magic-set technique presented in this paper.

[7] Indeed, the final size of the rewritten program actually depends also on the query, since rules that are not relevant for answering are not considered at all; nevertheless, we perform here a worst-case analysis (which is faced when all original rules are relevant).

[8] It is worth noting that the class of programs therein considered allow also default negation in the bodies and disjunction in the heads.

[9] We recall here that a strongly connected component of a directed graph is a maximal subset S of the vertices, such that each vertex in S is reachable from all other vertices in S.

Theorem 3. Given a ground query Q on a program P, if Q is finitely-recursive on P, then $RW(Q,P)$ is finitely-ground.

Proof. (*Sketch*) Let P_{magic} be the set of predicates defined by rules in $magicRules(Q,P)$ and let P_{mod} be the set of predicates defined by $modifiedRules(Q,P)$. We observe that $P_{magic} \cap P_{mod} = \emptyset$ and every component ordering γ for $RW(Q,P)$ (Definition 4 in [6]) will be such that: if $p \in P_{magic}$ and p belongs to the component C_i, then C_i will precede every component C_j featuring a predicate q s.t. $q \in P_{mod}$. This means that, in the modular bottom-up evaluation performed by the intelligent instantiation (Definition 8 in [6]), all rules in $magicRules(Q,P)$ will precede rules in $modifiedRules(Q,P)$.

We know that Q is finitely-recursive, i.e. it depends only on a finite set of other ground atoms in $grnd(P)$. But rules in $magicRules(Q,P)$ are properly built in order to exclusively derive atoms which the query Q depends on. So, starting from the ground query atom Q, each element of the sequence S_i in the intelligent instantiation definition is necessarily finite for modules of $magicRules(Q,P)$. But, magic atoms give binding to all variables occurring in the head of rules in $modifiedRules(Q,P)$, restricting their domain of possible values, so elements S_i will be finite also for all modules of $modifiedRules(Q,P)$. This is enough to prove the statement.

This last result is relevant because it implies that all nice properties of finitely-ground programs hold for rewritten finitely-recursive queries too. This includes, in particular, bottom-up computability of the answer set and hence full decidability of reasoning.

Example 13. If we consider the rewritten program P_{15} of Example 9, we can observe that its resulting intelligent instantiation is finite:

$$\begin{array}{l} magic\_nat(s(s(0))). \\ magic\_nat(s(0)) \text{ :- } magic\_nat(s(s(0))). \\ magic\_nat(0) \text{ :- } magic\_nat(s(0)). \\ nat(0). \\ nat(s(s(0))) \text{ :- } magic\_nat(s(s(0))), nat(s(0)). \\ nat(s(0)) \text{ :- } magic\_nat(s(0)), nat(0). \end{array} \tag{19}$$

The above ground program has the following unique finite answer set: $\{magic\_nat(s(s(0))), magic\_nat(s(0)), magic\_nat(0), nat(0), nat(s(0)), nat(s(s(0)))\}$. The answer to the query Q is then 'yes'.

Example 14. Let us now consider the rewritten program (P_{16}) of Example 10. Also this program, differently from the originating P program, can be safely bottom-up evaluated. Its intelligent instantiation is finite and results to be:

$$\begin{array}{l} magic\_lessThan(s(s(0)), s(0)). \\ magic\_lessThan(s(s(0)), 0) \text{ :- } magic\_lessThan(s(s(0)), s(0)). \end{array} \tag{20}$$

The above ground program has the unique finite answer set $\{magic\_lessThan(s(s(0)), s(0)), magic\_lessThan(s(s(0)), 0)\}$. Thus, the answer to the query Q is 'no'.

6 Conclusions

We presented an adaptation of the magic-sets technique that allows query answering over positive finitary normal programs also by means of standard bottom-up techniques. This allows us to enrich the collection of logic programs with function symbols for which ground query answering can be performed by all current ASP solvers.

Future work will focus on overcoming the limitation of considering only ground queries, by identifying the minimal set of variables required to be bound in order to preserve decidability. Next step will then deal with negation.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
4. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proceedings of the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA, The MIT Press (November 1999) 23–37
5. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S., eds.: The Logic Programming Paradigm – A 25-Year Perspective. Springer Verlag (1999) 375–398
6. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008). Volume 5366 of Lecture Notes in Computer Science., Udine, Italy, Springer (December 2008) 407–424
7. Baselice, S., Bonatti, P.A., Criscuolo, G.: On Finitely Recursive Programs. In: 23rd International Conference on Logic Programming (ICLP-2007). Volume 4670 of LNCS., Springer (2007) 89–103
8. Simkus, M., Eiter, T.: FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In: Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR2007). Volume 4790 of Lecture Notes in Computer Science., Springer (2007) 514–530
9. Lin, F., Wang, Y.: Answer Set Programming with Functions. In: Proceedings of Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR2008), Sydney, Australia, AAAI Press (September 2008) 454–465
10. Syrjänen, T.: Omega-restricted logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Vienna, Austria, Springer-Verlag (September 2001)
11. Bonatti, P.A.: Reasoning with infinite stable models. Artificial Intelligence **156**(1) (2004) 75–111
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (July 2006) 499–562
13. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: `DLV-Complex` homepage (since 2008) http://www.mat.unical.it/dlv-complex.
14. Marano, M., Ianni, G., Ricca, F.: A Magic Set Implementation for Disjunctive Logic Programming with Function Symbols. Submitted to CILC 2009.
15. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In Baral, C., Truszczyński, M., eds.: Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000), Breckenridge, Colorado, USA (April 2000) Online at http://xxx.lanl.gov/abs/cs/0003033v1.
16. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts (1986) 1–15

17. Cumbo, C., Faber, W., Greco, G., Leone, N.: Enhancing the magic-set method for disjunctive datalog programs. In: Proceedings of the the 20th International Conference on Logic Programming – ICLP'04. Volume 3132 of Lecture Notes in Computer Science. (2004) 371–385
18. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. Journal of Computer and System Sciences **73**(4) (2007) 584–609
19. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Volume 2. Computer Science Press (1989)
20. Beeri, C., Ramakrishnan, R.: On the Power of Magic. In: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '87), New York, NY, USA, ACM (1987) 269–284