Fundamenta Informaticae XX (2009) 1–20
IOS Press

Finitely Recursive Programs: Decidability and Bottom-up Computation*

Francesco Calimeri

Susanna Cozza

Giovambattista Ianni

Nicola Leone

Abstract. The support for function symbols in logic programming under answer set semantics allows to overcome some modeling limitations of traditional Answer Set Programming (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. Lately, the research community is focusing on finding proper subclasses of programs with functions for which decidability of inference is guaranteed. The two major proposals, so far, are finitary programs and finitely-ground programs. These two proposals are somehow complementary: indeed, the former is conceived to allow decidable querying (by means of a top-down evaluation strategy), while the latter supports the computability of answer-sets (by means of 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, which are based on a bottom-up computational model. However, there are also some interesting programs which are suitable for top-down query evaluation; but they do not fall in the class of finitely-ground programs.

In this paper, we focus on disjunctive finitely-recursive positive (DFRP) programs. We design a version of the magic-sets technique for DFRP programs, which ensures query equivalence under both brave and cautious reasoning. We show that, if the input program is DFRP, then its magic-set rewriting is guaranteed to be finitely-ground. Thus, reasoning on DFRP programs turns out to be decidable, and we provide an effective method for its computation on the ASP system DLV.

Keywords: Answer set programming, function symbols, magic sets, knowledge representation.

1. Introduction

Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP) [2, 18, 19, 24, 27], 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 [33, 6, 7, 31, 26, 11, 12, 3, 23]. Supporting function symbols allows to overcome one

*This paper is a significantly extended and revised version of papers appeared in CILC 2009, 24esimo Convegno di Logica Computazionale, and LPNMR 2009, 10th International Conference on Logic Programming and Nonmonotonic Reasoning, pp 71–86.

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 in the general case. The identification of expressive decidable classes of ASP programs with functions is therefore an important task. Two relevant decidable classes of ASP with functions, resulting from alternative approaches (top-down vs bottom-up), are finitary programs [7] and finitely-ground programs [12].

Finitary programs [7] is a class that allows for 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; programs respecting this latter condition are called *finitely recursive* [3]. Moreover, potential sources of inconsistency are limited by requiring that the set of odd-cycles (cycles of recursive calls involving an odd number of negative subgoals) is finite. Thanks to these two restrictions, consistency checking and ground queries are decidable, provided that the atoms involved in odd-cycles are known [8]; while non-ground queries are semi-decidable.

Finitely-ground ($\mathcal{FG}$) programs [12], more recently proposed, can be seen as a "dual" class of finitary programs: the latter is suitable for a top-down evaluation, the former allows for 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 $\mathcal{FG}$ programs can express any computable function, and preserve decidability for ground queries (modulo the availability of odd cyclic atoms for finitary programs). However, answer sets and non-ground queries are computable on $\mathcal{FG}$ programs, while they are not computable on finitary programs. Furthermore, the bottom-up nature of the notion of $\mathcal{FG}$ programs allows an immediate implementation in ASP systems (as ASP instantiators are based on a bottom-up computational model). Indeed, the DLV system [22], for instance, has already been adapted to deal with $\mathcal{FG}$ program by extending its instantiator [13].

Finitary and $\mathcal{FG}$ programs are uncomparable. In particular, the class of $\mathcal{FG}$ programs does not include some programs for which ground querying can be computed in a top-down fashion, like, in particular, $\vee$-free finitely-recursive positive programs. Despite of its simplicity, this latter class includes many significative programs, such as most of the standard predicates for lists manipulation. For instance, the following program, performing the check for membership of an element in a list, is finitely recursive and positive, yet not finitely ground.

$$member(X, [X|Y]). \qquad\qquad member(X, [Y|Z]) \text{ :- } member(X, Z).$$

In this paper, we shed some light on the relationships between finitely recursive and $\mathcal{FG}$ programs, evidencing a sort of "dual" behaviour of the two classes. We show that a suitable magic-set rewriting transforms finitely recursive positive programs into $\mathcal{FG}$ programs. In this way, we devise a strategy for the bottom-up evaluation of finitely-recursive positive programs. Importantly, we effectively deal also with *disjunctive* finitely-recursive positive programs, which were unknown to be decidable so far. In summary, the paper focuses on disjunctive finitely-recursive positive programs (DFRP programs) and queries, providing the following main contribution:

- We design a *magic-sets* rewriting technique for disjunctive programs with functions, which exploits the peculiarities of finitely recursive programs.

- We show that our magic-sets rewriting $RW(Q, P)$ of a (ground) query Q on a DFRP program P enjoys the following properties:

  - for both brave and cautious reasoning, we have that $P \models Q$ iff $RW(Q, P) \models Q$;
  - if Q belongs to the class of finitely recursive queries on P, then $RW(Q, P)$ is finitely ground.
  - the size of $RW(Q, P)$ is linear in the size of the input program;

- We then show that both brave and cautious reasoning on DFRP programs are decidable.

- Importantly, we provide not only a theoretical decidability result for reasoning on DFRP programs, but we also supply a concrete implementation method. Indeed, by applying a light-weight magic-sets rewriting on the input program,any query on a DFRP program can be evaluated by the ASP system DLV, or by any other system supporting $\mathcal{FG}$ programs. Our proposal makes now possible to evaluate, by means of bottom-up techniques, programs featuring 'unsafe' variables in the head of rules, such as, for instance, the program shown before, defining the predicate 'member' for lists. Note that such programs cannot be handled by any of the current bottom-up ASP solvers.

- Besides positive programs, DFRP programs allow for disjunction. In this respect, it is worth mentioning that this proves to be useful when translating description logics to Answer Set Programming. Translations from description logics into disjunctive positive programs are known [20, 32]. However, currently, a cumbersome passage for eliminating function symbols is required. It turns out that this last passage can be avoided when the translated logic program falls in our class. Furthermore, the presence of disjunction significantly increases the declarative nature of the language.

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 describes the magic-sets rewriting technique for function-free programs as already known in the literature, while Section 5 illustrates our adaptation of such technique to the class of DFRP programs (with functions); in Section 6 we present a number of theoretical results; Section 7 discusses related literature, and, eventually, Section 8 draws our conclusions.

2. Motivation

In the $\vee$-free (or-free) case, 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, reasoning with ground queries is decidable (while reasoning is semi-decidable in case of non ground queries). Unfortunately, even if an $\vee$-free 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 2.1. Consider the following program, defining the natural numbers:

   $$nat(0). \qquad\qquad nat(s(X)) \text{ :- } nat(X).$$

   The above program is positive and finitely recursive; hence 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 a 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, standing for infinite instances, which are not admissible by present grounding algorithms.

Example 2.2. As well-known, function symbols enable the possibility of modeling complex data structures such as lists. Let us consider the traditional program used for encoding the *append* predicate:

$$append([\,], L, L). \qquad append([X|X_s], L, [X|Y_s]) \text{ :- } append(X_s, L, Y_s).$$

Ground queries like $append([a], [\,], [a])$ are decidable, although the above program is not safe.

It is worth noting that in this paper we deal also with *disjunctive* finitely recursive programs, which were not even known to be decidable so far, also in the positive case.

Example 2.3. Consider the following program, computing all the possible 2-colorings for an *infinite* chain of nodes and defining coupled nodes as pairs of nodes which are successive and share the same color.

$$\begin{array}{l} color(X, b) \vee color(X, g). \\ coupled(X, next(X), C) \text{ :- } color(X, C), color(next(X), C). \end{array} \tag{1}$$

The above program is positive and finitely recursive; nevertheless, a bottom-up evaluation is unfortunately unfeasible: the first rule, indeed, represents an infinite set of atoms. Simple programs like this constitute a pattern that might occur also in practical problems, such as guessing sequences with given properties in streams of data.

Example 2.4. The following program P_2 defines the comparison operator 'less than' between two natural numbers (the function symbol s represents the successor of a natural number):

$$\begin{array}{l} lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } lessThan(X, Y). \end{array} \tag{2}$$

In this case, bottom-up evaluation is unfeasible, both because of the first rule, and because of the infinite recursion in the second rule.

3. Preliminaries

In this section, we first provide the formal specification of the ASP language fragment we take into consideration (positive disjunctive programs with functions); then, we briefly recall the class of finitely-ground programs [12].

3.1. ASP With Functions

A *term* is either a *simple term* or a *functional term*[1]. 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*.

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*. An atom having p as predicate name is usually referred as $p(\bar{t})$.

A (positive) *disjunctive rule* r is of the form: $\alpha_1 \vee \cdots \vee \alpha_k$:- $\beta_1, \cdots, \beta_n.$, where $k > 0$; $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_n$ are atoms. The disjunction $\alpha_1 \vee \cdots \vee \alpha_k$ is called *head* of r, while the conjunction $\beta_1, \cdots, \beta_n.$ is the *body* of r. We denote by $H(r)$ the set of the head atoms, by $B(r)$ the set of body atoms; we refer to all atoms occurring in a rule with $Atoms(r) = H(r) \cup B(r)$. A rule having precisely one head atom (i.e., $k = 1$ and then $|H(r)| = 1$) is called a *normal rule*. If r is a normal rule with an empty body (i.e., $n = 0$ and then $B(r) = \emptyset$) we

[1] We will use traditional square-bracketed list constructors as shortcut for the representation of lists by means of nested functional terms (see, for instance, [12]).

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. A $\vee$-free program P is a program consisting of normal rules only.

Given a predicate p, a *defining rule* for p is a rule r such that some $p(\bar{t})$ occurs in $H(r)$. If all defining rules of a predicate p are facts, then p is an *EDB predicate*; otherwise p is an *IDB predicate*[2]. The set of all facts of P is denoted by $Facts(P)$; the set of instances of all *EDB* predicates is denoted by $EDB(P)$. A program (a rule, an atom, a term) is *ground* if it contains no variables. A *query* Q is a ground atom.[3]

The most widely accepted semantics for ASP programs is based on the notion of answer-set, proposed in [19] as a generalization of the concept of stable model [18]. Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all 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 atoms obtainable from the atoms of P by replacing variables with elements from U_P. Elements of U_P and B_P are called ground terms and ground atoms, respectively. 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 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, we say that r is satisfied w.r.t. I if some atom appearing in $H(r)$ is true w.r.t. I or some 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 it is *minimal*, i.e., there is no model N for P such that $N \subset M$. The set of all answer sets for P is denoted by $AS(P)$. A program P *bravely entails* (resp., *cautiously entails*) a query Q, denoted by $P \models_b Q$ (resp., $P \models_c Q$) if Q is true in some (resp., all) $M \in AS(P)$.

3.2. Finitely-Ground Programs

The class of finitely-ground ($\mathcal{FG}$) programs [12] constitutes a natural formalization of programs which can be finitely evaluated bottom-up.

Informally, the definition of finitely-ground program relies on the so-called "intelligent instantiation", obtained by means of an operator which is iteratively applied on program's submodules, producing sets of ground rules. In order to properly split a given program P into modules, it is taken in consideration the *Dependency Graph* and the *Component Graph*. The first connects predicate names based on head-bodies dependencies, while the latter connects strongly connected components of the former. Each module corresponds to a strongly connected component (SCC)[4] 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 intelligent instantiation P^γ obtained iteratively, by following the sequence given by γ, has the same answer sets of $grnd(P)$.

For the sake of clarity, we shortly recall here some key concepts introduced in [12], tailoring them to the positive case[5], which is the one herein considered. For complete formal definitions, more details, and examples, we refer the reader to the aforementioned paper. Given a program P, a *component* C is as set of predicates which are strongly connected in the usual predicate dependency graph $\mathcal{G}(P)$ considering head-body dependencies between predicates.

The Component Graph of P, denoted $\mathcal{GC}(P)$, is a directed graph having a node for each strongly connected component of $\mathcal{G}(P)$ and an edge $B \rightarrow A$ if there is a rule $r \in P$ such that there is a predicate $q \in A$ occurring in the head of r and a predicate $p \in B$ occurring in the body of r; A component ordering $\gamma = \langle C_0, \ldots, C_n \rangle$, is a

---

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

[3] 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).

[4] 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.

[5] It is worth noting that the class of programs considered in [12] allows also default negation in the bodies of rules.

total ordering of all components of P s.t., for any C_i, C_j with $i < j$, there is no path from C_j to C_i in $\mathcal{GC}(P)$. A *module* $P(C_i)$ is the set of rules defining predicates in C_i in a program P, excepting those that define also some other predicate belonging to a lower component $C_j, j < i$ in γ.

Definition 3.1. [12] Given a rule r and a set S of ground atoms, an *S-restricted* instance of r is a ground instance r' of r such that $B(r') \subseteq S$. The set of all S-restricted instances of a program P is denoted as $Inst_P(S)$.

Note that, for any $S \subseteq B_P$, $Inst_P(S) \subseteq grnd(P)$. Intuitively, this helps selecting, among all ground instances, those somehow *supported* by a given set S. Some further simplifications can be properly performed by exploiting a modular evaluation of the program that relies on a component ordering.

Definition 3.2. (Adapted from [12]) Given a program P, a component ordering $\langle C_1, \ldots, C_n\rangle$, a set S_i of ground rules for C_i, and a set of ground rules R for the components preceding C_i, the *simplification* $Simpl(S_i, R)$ of S_i w.r.t. R is obtained from S_i by: (i) *deleting* each rule whose head contains some atom $a \in Facts(R)$; (ii) *eliminating* from the remaining rules each atom $a \in B(r)$ s.t. $a \in Facts(R)$.

Assuming that R contains all ground instances obtained from the modules preceding C_i, $Simpl(S_i, R)$ deletes from S_i all rules whose head is certainly already true w.r.t. R, and simplifies the remaining rules by removing from the bodies all atoms true w.r.t. R. We define now the operator Φ, combining $Inst$ and $Simpl$.

Definition 3.3. [12] Given a program P, a component ordering $\langle C_1, \ldots, C_n\rangle$, a component C_i and its corresponding module M, a set X of ground rules of $P(C_i) = M$, and a set R of ground rules belonging only to *EDB* (P) or to modules of components C_j with $j < i$, let $\Phi_{M,R}(X)$ be the transformation defined as :

$$\Phi_{M,R}(X) = Simpl(Inst_M(Heads(R \cup X)), R).$$

Intuitively, $\Phi_{M,R}$ instantiates a given module of P exploiting the instantiation of previous modules, and generates a subset of the theoretical instantiation; importantly, it always admit a least fixpoint $\Phi^{\infty}_{M,R}(\emptyset)$. The *intelligent instantiation* P^γ of P for γ is the last element S_n of the sequence $S_i = S_{i-1} \cup \Phi^{\infty}_{M,S_{i-1}}(\emptyset)$, for $S_0 = EDB(P)$. A nice characterization of the intelligent instantiation can be given in the case of normal ($\vee$-free) programs.

Proposition 3.1. Given a positive normal ($\vee$-free) ground program P and an interpretation I, let $\mathcal{T}_P(I) = \{a \mid a \,\text{:-}\, b_1, \ldots, b_n \in P,\ \{b_1, \ldots, b_n\} \subseteq I\}$ be the traditional *immediate consequence operator* [35]. Then, for any component ordering γ of P, P^γ is a set of facts coinciding with the atoms contained in $\mathcal{T}^{\infty}_P(\emptyset)$.

We provide now the definition of finitely-ground programs, and report next the main result about such class.

Definition 3.4. [12] A program P is *finitely-ground* ($\mathcal{FG}$) if P^γ is finite, for every component ordering γ for P.

Theorem 3.1. (Theorem 3 of [12]) Cautious and brave reasoning over $\mathcal{FG}$ programs are computable. Computability holds even for non-ground queries.

4. The Magic-Sets Technique

In this Section we give some basics on the *magic-sets* technique and its adaptation to the disjunctive case.

4.1. Basic Magic-Sets

It is commonly asserted that the magic-sets method is a strategy for simulating the top-down evaluation. Indeed, and more precisely, the magic sets method does not actually execute a top-down evaluation: the technique tracks down how a top-down evaluation would run on a given program P with a given query Q, and exploits the collected information for building a modified version of P, this latter supposedly more efficient in its evaluation.

This new version of P is obtained by modification and addition of rules. As a result, the computation of Q over P can be narrowed to a smaller ground program containing only atoms and rules which might have some impact in answering Q. A simplistic versions of a bottom-up evaluation algorithm works as follows: given P, we compute $grnd(P)$ (or an equivalent subset thereof) and then apply an algorithm tailored at computing the answer sets of $grnd(P)$. In many applications $grnd(P)$ can be of large size, and, indeed, efforts for restricting $grnd(P)$ to an equivalent and considerably smaller program exist [12, 17, 21].

On the other hand, recall how intuitively a top-down evaluation works: a given query atom Q (the *goal*) is matched against all the rules heads which Q unifies with; when a rule r is matched, all the body atoms of r become in turn *subgoals*. Subgoals are obtained from body atoms by unification with previously matched subgoals, so that the search space of possible matches to subgoals is usually considerably smaller: for instance, for goal $p(a, X)$ and matching rule $p(X, Y)$:- $e(X, Z),\ q(X, Z)$., we obtain a subgoal atom $e(a, Z)$. In turn, $q(X, Z)$ generates a range of subgoals $q(a, z)$ depending on the domain of all values z for which the subgoal $e(a, z)$ is found to be true. In a sense, the range of the first argument of $q(X, Z)$ can be seen as *bound* by the value a, while the range of the second argument Z is bound by the allowed values of Z in e.

One might thus think at tracking the propagation of information due to variable bindings (*sideway information passing*, in the literature), appearing in the query and in subgoals. This information can be valuably exploited in order to generate a program (*the magic program*) which enforces the search space of each subgoal argument to be smaller. Given a program P and a query Q, a magic program is generated: this is added to P, while *magic atoms* are put in rule bodies of P in order to constrain, wherever possible, the domain of subgoal arguments.

We next provide a brief and informal description of the magic-sets rewriting technique, which has originally been defined in [1] for non-disjunctive Datalog (i.e., with no function symbols) queries only. Note that, afterwards, many generalizations have been proposed: the reader is referred to [34] for a detailed presentation. The method is structured in four main phases which are informally illustrated below by example, considering the query $path(X, a)$ on the following program (*edge* is an EBD predicate):

$$path(X, Y) \text{ :- } edge(X, Y). \qquad\qquad path(X, Y) \text{ :- } edge(X, Z), path(Z, Y).$$

1. Adornment Step: First, suitable predicates (*adorned predicates*) with *adornment labels* attached to them, are introduced. Adornments track how variable bindings can be possibly propagated from head atoms to body atoms, and from body atoms to subsequent ones.

In order to ease the reader's understanding, we will assume here that the order of evaluation for body atoms (subgoals) goes from left to right, although different strategies for choosing a subsequent subgoal can be devised.[6]

Values of adornment labels can be *b* and *f*, denoting 'bound' and 'free' respectively, for each argument of an IDB predicate. For instance $path^{fb}$ is a binary predicate which intuitively is a subset of $path$: in particular its second argument is meant to be restricted to a given domain (the *magic set* of $path^{fb}$) which is usually much smaller than the ideal range of $path$ on its second argument (in principle this range could be the full Herbrand universe). Adorned predicates are created by starting from a given query atom Q, adorning it, and then recursively traversing all the bodies of rules in which Q appears. If Q has some constant argument, then this is adorned as *b*. Remaining arguments are adorned as *f*.

Adornment labels are then added to predicate names appearing in rules bodies, for each rule whose head unifies with Q. Intuitively, how labels are chosen depends on the order of evaluation of subgoals within a

[6] We will see next that our simplified adornment strategy does not depend on the chosen ordering for body atoms.

given rule; if a variable appears in a subgoal which has been already taken in consideration, the corresponding argument can be seen as bound: that is, its domain can be determined a priori, once the range of previous subgoals is determined. Determining if an argument adornment should be bound or free can be a sophisticated task as, for instance, described in the *generalized* magic set method [4].

For simplicity, we consider in this informal overview the method of [1]: namely, a body argument is adorned as bound if it is either *a)* a constant, *b)* it shares the same variable with a bound head argument, or *c)* it shares the same variable with an EBD predicate argument appearing in the rule body. An argument is adorned as free otherwise. It is worth noting that, on the other hand, according to the method described in [4], bindings may also be generated by IDB predicates in rule bodies. In particular, an appropriate way of treating sideway information passing, (called *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 of [1] herein adopted uses a particular, predetermined SIPS for all rules.

Example 4.1. Adorning the query $path(X, a)$ generates the adorned predicate $path^{fb}$ and the two adorned rules:

$$path^{fb}(X, Y) \text{ :- } edge(X, Y). \qquad path^{fb}(X, Y) \text{ :- } edge(X, Z), path^{bb}(Z, Y).$$

It is worth noting that adorning a rule may generate new adorned predicates: for instance, in the above example, the new predicate $path^{bb}(Z, Y)$ is generated. Each new adorned predicate can be seen as the counterpart of a family of subgoals in a top-down computation. As such, each new adorned predicate is in turn matched against all the rules whose head unifies with; these latter are adorned with respect to the new adorned predicate, and the process is repeated until no new adorned predicate is generated.

Example 4.2. Processing the adorned predicate $path^{bb}$ we obtain the two adorned rules:

$$path^{bb}(X, Y) \text{ :- } edge(X, Y). \qquad path^{bb}(X, Y) \text{ :- } edge(X, Z), path^{bb}(Z, Y).$$

Take also note that EDB predicates are not subject to adornment.

2. Generation Step: Bound arguments appearing in adorned predicates can be seen as placeholders for arguments which have a domain that can be narrowed. This restricted domain can be computed a priori: a specification of how to compute the domain of bound arguments is given via the so called *magic program.* This latter is constituted of *magic rules*. First, for each adorned predicate p^{α}, having adornment α, we introduce a corresponding *magic* predicate $magic_p^{\alpha}$. The arity of $magic_p^{\alpha}$ will correspond to the number of bound labels appearing in α: that is, free arguments will not appear in magic predicates. Accordingly, from an atom $p^{\alpha}(\bar{t})$ we obtain the *magic* atom $magic(p^{\alpha}(\bar{t})) = magic_p^{\alpha}(\bar{t}')$ where $\bar{t}'$ is obtained from $\bar{t}$ by eliminating all arguments corresponding to an f label in α.

Consider now how to devise rules defining a predicate $magic_p^{\alpha}$. Intuitively, the extension of such a predicate should range only over the values that might have impact on the query at hand when evaluated in a top-down fashion. This means that the extension of $magic_p^{\alpha}$ is constrained: *a)* by the extension of EDB predicates; consider, e.g., the body of a rule containing a EDB atom $e(\bar{t}, \bar{s})$ and an atom $p^{\alpha}(\bar{t}, \bar{u})$: in this case one might add the rule $magic_p^{\alpha}(\bar{t}) \text{ :- } e(\bar{t}, \bar{s})$; and, recursively, *b)* from the extension of other magic predicates from which some bound label has been propagated; consider, e.g., an adorned rule having an atom $q^{\beta}(\bar{t}, \bar{s})$ in its head and $p^{\alpha}(\bar{t}, \bar{u})$ in its body; one might add the rule $magic_p^{\alpha}(\bar{t}) \text{ :- } magic_q^{\beta}(\bar{t})$. [7].

The magic program is thus generated as follows. For each adorned atom A appearing 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 $magic(A)$, and (ii) the body

[7] In general, the extension of $magic_p^{\alpha}$ is built taking into account the extension of all the predicates which caused a bound adornment to appear in α. Indeed in generalized magic sets [4], $magic_p^{\alpha}$ might depend also on some magic predicate $magic_s^{\gamma}$ for s^{γ} and p^{α} sharing a variable within the same body.

of r_m consists of the magic version of the head atom of r_a, followed by all the atoms of r_a which can propagate their variable bindings on A.

Example 4.3. From the overall adorned program obtained in Example 4.1 and 4.2, we generate the following magic program:

$$magic_path^{bb}(Z,Y) \text{ :- } magic_path^{bb}(X,Y), edge(X,Z).$$
$$magic_path^{bb}(Z,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Z).$$

Note how argument labeled as *f* are removed from magic atoms (e.g., $magic_path^{fb}$), and only rules having an adorned predicate in their body are processed and contribute in generating magic rules.

3. Modification Step: The adorned rules are subsequently modified by including magic atoms generated in Step 2 in the rule bodies. This constrains the range of the head variables to the domain of magic set predicates, thus avoiding the inference of facts which cannot contribute to derive 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, the atom magic(H) is inserted in the body of the rule. Adornments, which served the purpose of tracking top-down information propagation, are then removed from all non-magic atoms appearing in r_a.[8]

Example 4.4. For the above adorned program, we obtain:

$$path(X,Y) \text{ :- } magic_path^{bb}(X,Y), edge(X,Y).$$
$$path(X,Y) \text{ :- } magic_path^{bb}(X,Y), edge(X,Z), path(Z,Y).$$
$$path(X,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Y).$$
$$path(X,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Z), path(Z,Y).$$

Note that in our example, the extension of the first argument of $magic_path^{fb}(Y)$ is in any case greater than the extension of $magic_path^{bb}(X,Y)$ on its second argument (see Example 4.3). Thus the fourth rule above subsumes the corresponding second rule, and the third rule subsumes the first. For simplicity, we omit the optimization steps that usually are introduced for eliminating such redundant rules.

4. Processing of the Query: It is eventually necessary to assert some domain information regarding the query at hand. Let the query goal be the adorned IDB atom g^{α}. We generate the so called *magic seed* atom magic(g^{α}). This latter is asserted as a fact. For instance, in our example we generate the *magic fact* $magic_path^{fb}(a)$.

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

Example 4.5. The complete rewriting of our example program is:

$$magic_path^{fb}(a).$$
$$magic_path^{bb}(Z,Y) \text{ :- } magic_path^{bb}(X,Y), edge(X,Z).$$
$$magic_path^{bb}(Z,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Z).$$
$$path(X,Y) \text{ :- } magic_path^{bb}(X,Y), edge(X,Y).$$
$$path(X,Y) \text{ :- } magic_path^{bb}(X,Y),\ edge(X,Z), path(Z,Y).$$
$$path(X,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Y).$$
$$path(X,Y) \text{ :- } magic_path^{fb}(Y), edge(X,Z), path(Z,Y).$$

Here, $magic_path^{fb}(Y)$ represents the set of nodes which can potentially start a path leading to a. Therefore, when answering the query, only these sub-paths will be actually considered in bottom-up computations.

[8] Stripping off the adornments serves mainly for facilitating the equivalence proofs; one may also leave the adornments (also in the query) intact, as it was done in the original definition of magic sets.

4.2. Disjunctive Magic Sets

We now revert our focus back to the language fragment we are interested in, that is, positive disjunctive logic programs allowing function symbols. It is worth noting that the traditional magic set technique does not straightforwardly applies to this class, because of *a)* the presence of disjunction, and *b)* the possibility of having functional terms in combination with disjunction.

We consider first the impact of disjunction. Intuitively, magic predicates capture all the ground atoms which might be relevant for the query Q at hand: if an atom $p(\bar{X})$ is relevant for evaluating Q, and a disjunctive rule r contains $p(\bar{X})$ in the head, also other atoms sharing the same head might have impact on the truth value of $p(\bar{X})$. Thus, one has to investigate also how variable bindings from $p(\bar{X})$ propagate to the head atoms, besides body atoms; this means that in disjunctive rules also head atoms must be adorned, causing however some technical problems. We adapt in the following the approach illustrated in detail in [14], which proved to be satisfactory in this respect. Our adaptation can be summarized as follows:

1. Given a disjunctive rule r to be adorned starting from the head atom $p^{\alpha}(\bar{t})$, each bound label appearing in α can be propagated to other head atoms, besides body atoms. This agrees with the notion of *disjunctive SIPS* as defined in [14]. For simplicity, our binding propagation tracking strategy does not include propagation of bound labels from EDB atoms to head atoms.

2. There is a magic rule for each IDB atom appearing within rules bodies *or* within rule heads. Also, magic rules are not generated starting from disjunctive adorned rules: when magic rules have to be generated for a disjunctive rule r and selected head atom a, we take into consideration the rule r' obtained from r by moving all the head atoms, other than a, into the body. Then, the standard generation step described in Section 4.1 is applied to r'.

3. The Modification step (stage 3 of section 4.1) is applied to the original program, not to the adorned one. Disjunctive rules are modified adding the magic atom $magic(p^{\alpha}(\bar{t}))$ in the body for each head atom $p^{\alpha}(\bar{t})$.

Example 4.6. Consider again program P_1 of Example 2.3:

$$color(X,b) \vee color(X,g).$$
$$coupled(X,next(X),C) \text{ :- } color(X,C), color(next(X),C).$$

Let us consider the query $Q = coupled(1, next(1), g)$. After the adornment phase, all predicates, in this case, have a completely bound adornment, thus obtaining the adorned program

$$color^{bb}(X,b) \vee color^{bb}(X,g).$$
$$coupled^{bbb}(X,next(X),C) \text{ :- } color^{bb}(X,C), color^{bb}(next(X),C).$$

We show now what happens because of the disjunctive rule. It contains two atoms: $color^{bb}(X,b)$ and $color^{bb}(X,g)$, for which a corresponding magic rule has to be generated. First, we consider two intermediate rules:

$$color^{bb}(X,b) \text{ :- } color^{bb}(X,g). \qquad color^{bb}(X,g) \text{ :- } color^{bb}(X,b).$$

From these latter (and not from the originating adorned disjunctive rules) we obtain the two magic rules:

$$magic_color^{bb}(X,g) \text{ :- } magic_color^{bb}(X,b). \qquad magic_color^{bb}(X,b) \text{ :- } magic_color^{bb}(X,g).$$

Eventually, the original, non-adorned, disjunctive rule is modified including both magic atoms corresponding to the two head atoms, obtaining:

$$color(X,b) \vee color(X,g) \text{ :- } magic_color^{bb}(X,b), magic_color^{bb}(X,g).$$

The complete rewritten program consists of the generated and modified rules above, plus what comes out from the query and the second rule (with no difference w.r.t. what previously described for the normal case). Note that in the traditional literature, the modification step is performed on the adorned program. This latter approach has impact on soundness for disjunctive programs. As suggested in [14], we apply modifications to the original program, instead.

As for the impact of the presence of functional terms, adaptations of information passing strategies were already shown in the literature [29]. As shown next, the treatment of functional terms in the context of our main result (treating finitely-recursive queries on positive disjunctive programs) turns out to be much simpler.

5. Magic Sets for Finitely-Recursive Queries

This Section first gives the definition of finitely-recursive queries; then, a suitable version of the magic-sets rewriting technique to DFRP programs is designed.

5.1. Finitely-Recursive Queries

We next provide the definition of finitely-recursive queries and programs.

Definition 5.1. Let P be a program. A ground atom a *depends* on ground atom b with a 1 degree (denoted by $a \geqslant_1 b$) if there is some rule $r \in grnd(P)$ with $a \in H(r)$ and $b \in Atoms(r)$. The reflexive and transitive closure of the "depends on" relation defines dependence on a generic degree i: for any ground atom a, $a \geqslant_0 a$, and if $a \geqslant_1 b$, and $b \geqslant_i c$, then $a \geqslant_{i+1} c$ (a depends on c with a $i+1$ degree). The degree of the dependencies is often omitted, hence simply saying that a *depends on* b ($a \geqslant b$), if $a \geqslant_k b$ holds for some $k \geq 0$.

Definition 5.2. Given a query Q on P, define

$$relAtoms(Q,P) = \{a \in B_P : Q \geqslant a\}$$
$$relRules(Q,P) = \{r \in grnd(P) : a \in H(r) for\ some\ a \in relAtoms(Q,P)\}$$

Then: (i) a query Q is *finitely recursive* on P if $relAtoms(Q,P)$ is finite; (ii) a program P is *finitely recursive* if every query on P is finitely recursive.

The sets $relAtoms(Q,P)$ and $relRules(Q,P)$ are called, respectively, the *relevant atoms* and the *relevant subprogram* of P w.r.t. Q. It is worth noting that, if Q is finitely recursive, then its relevant subprogram is finite.

Example 5.1. Consider the following program:

$$lessThan(X, s(X)).$$
$$lessThan(X, s(Y)) \text{ :- } lessThan(X,Y).$$
$$r(X) \vee t(X) \text{ :- } lessThan(X,Y), q(X).$$
$$q(f(f(0))).$$
$$q(X) \text{ :- } q(f(X)).$$

The program is not finitely recursive (because of rule $q(X) \text{ :- } q(f(X))$. Nevertheless, one may have many finitely-recursive queries on it. All atoms like $lessThan(c_1, c_2)$, for instance, with c_1 and c_2 constant values, are examples of finitely-recursive queries.

5.2. The Rewriting Algorithm

In this section we describe a specialization of the magic-sets algorithm conceived for the case of finitely-recursive queries, which constitutes an elegant and simple adaptation of what described in Sections 4.1 and 4.2. In order to help the reader, we report some informal considerations that led us to the design of such specialization, even if they are not needed in order to prove the soundness of the framework (see Section 6). Note that the below remarks hold no matter of the presence of functional terms in a program.

For a program P and a finitely-recursive ground query Q on it, it is easy to see that:

- The adornment step, described in Section 4.1, recursively selects for adornment all rules relevant which the query depends on. When Q is finitely recursive, a selected rule r cannot have local variables (i.e. variables appearing only in the body of the rules), and each couple of atoms $a, b \in H(r)$ must share the same variables. Indeed, should one of these conditions be violated, $relRules(Q, P)$ would be infinite.
- Query Q is ground; therefore, given the consideration above, adorned predicates can have only bound labels, and adorned rules will contain bound variables only.
- As a consequence of the considerations above, in the generation step, it is no longer necessary to include any other atom in the body of a generated magic rule, apart from the magic version of the considered head atom. Again, this is due to the absence of local variables: this ensures that all the needed bindings are provided through the magic version of the head atom.

The algorithm MS_{FR}, implementing the magic-sets method for finitely-recursive queries and designed taking into account the above consideration, is depicted in Figure 1. MS_{FR} takes in input a positive finitely-recursive program P and a ground query Q, and 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.

A stack S stores all predicates that have 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, and the query predicate is pushed on S. At each step, an element u is removed from S; if u has not been already considered (the auxiliary variable $Done$ tracks the already processed predicate names), all the rules having u in their head are processed one-at-a-time. For each such rule r, let $u(\bar{t})$ be one of the occurrences of predicate u in $H(r)$; if r is not a fact, first a modified version of r is created; moreover, all other predicate names appearing in $H(r)$ and other than u, if some, and all IDB predicates appearing in $B(r)$, are pushed on S. Accordingly, a proper set of magic rules, one per each such pushed values, is generated. In case r is a fact, i.e., its body is empty and it does not contain variables, it is added to the $modifiedRules$ set as it is. Finally, once all the predicates involved in the query evaluation have been processed (thus, S is empty), the algorithm outputs the program $RW(Q, P)$ as the union of all modified rules and generated magic rules.[9] Some rewriting examples are reported next.

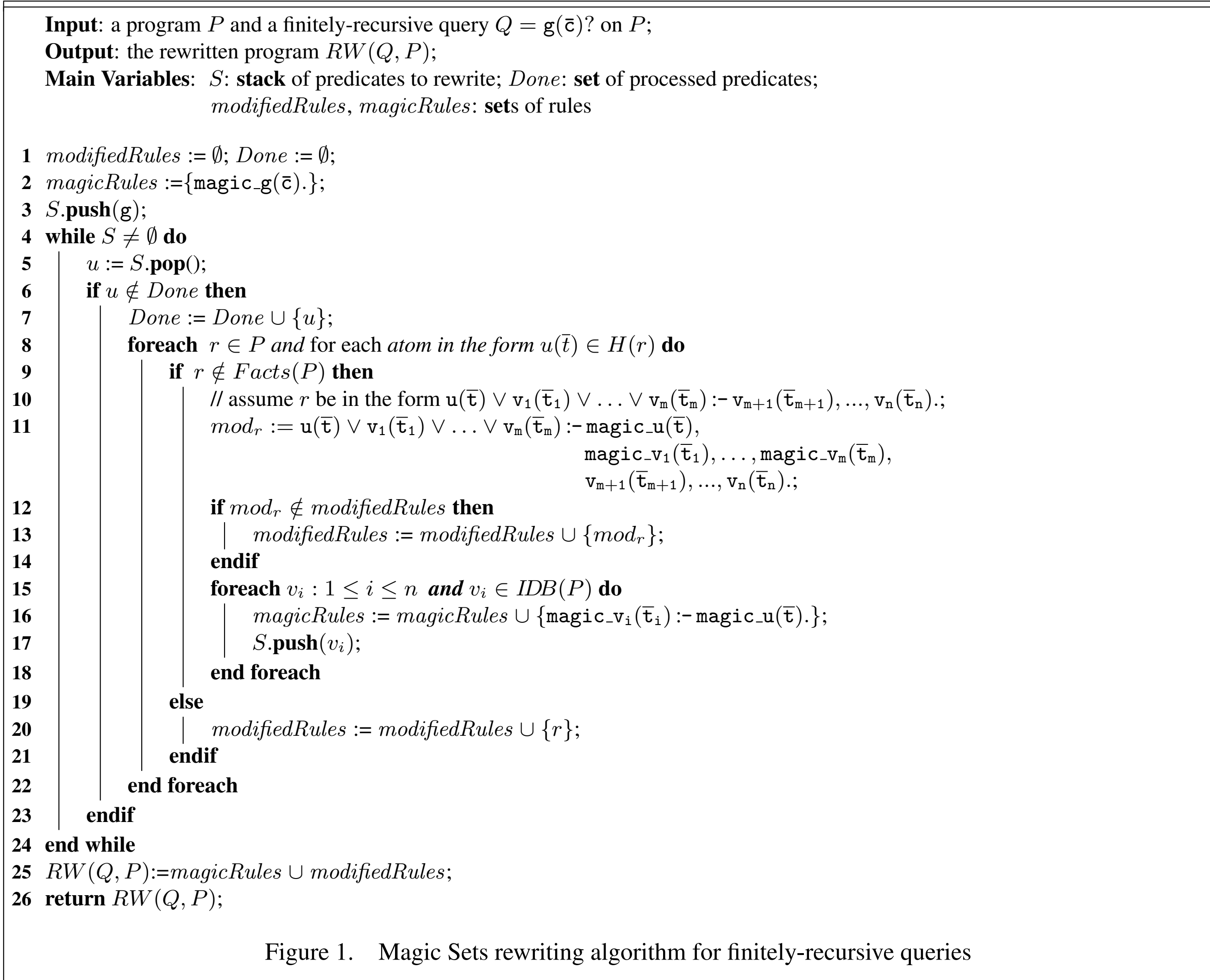
Example 5.2. Let us consider the finitely-recursive query $Q = p(f(g(1)))$ on the following program:

$$p(1). \qquad\qquad p(f(X)) \vee p(g(X)) \text{ :- } p(X).$$

We will depict, step by step, the execution performed by the MS_{FR} algorithm. After the initialization of variables, the first magic fact obtained from the query is generated (*lines* $1-2$):

$$magic_p(f(g(1))).$$

[9]Note that duplicate rules could be generated. Some further care might be taken in order to prevent this, but this is out of the scope of this work; some examples of optimization methods can be found in [14].



Input: a program P and a finitely-recursive query $Q = \mathtt{g}(\bar{\mathtt{c}})?$ on P;
Output: the rewritten program $RW(Q,P)$;
Main Variables: S: **stack** of predicates to rewrite; $Done$: **set** of processed predicates;
$modifiedRules$, $magicRules$: **sets** of rules

```
1  modifiedRules := ∅; Done := ∅;
2  magicRules :={magic_g(c̄).};
3  S.push(g);
4  while S ≠ ∅ do
5      u := S.pop();
6      if u ∉ Done then
7          Done := Done ∪ {u};
8          foreach r ∈ P and for each atom in the form u(t̄) ∈ H(r) do
9              if r ∉ Facts(P) then
10                 // assume r be in the form u(t̄) ∨ v1(t̄1) ∨ ... ∨ vm(t̄m) :- vm+1(t̄m+1), ..., vn(t̄n).;
11                 modr := u(t̄) ∨ v1(t̄1) ∨ ... ∨ vm(t̄m) :- magic_u(t̄),
                                                          magic_v1(t̄1), ..., magic_vm(t̄m),
                                                          vm+1(t̄m+1), ..., vn(t̄n).;
12                 if modr ∉ modifiedRules then
13                     modifiedRules := modifiedRules ∪ {modr};
14                 endif
15                 foreach vi : 1 ≤ i ≤ n and vi ∈ IDB(P) do
16                     magicRules := magicRules ∪ {magic_vi(t̄i) :- magic_u(t̄).};
17                     S.push(vi);
18                 end foreach
19             else
20                 modifiedRules := modifiedRules ∪ {r};
21             endif
22         end foreach
23     endif
24 end while
25 RW(Q,P):=magicRules ∪ modifiedRules;
26 return RW(Q,P);
```

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

The predicate p is then pushed onto the stack S (*line* 3) and the first iteration of the main loop (*line* 4) starts. The predicate p is extracted from S and marked as done (*lines* $5 - 7$). In this case, it is the only predicate to be considered. All rules having p in their head are then processed (*lines* $8 - 22$). The first rule defining p is a fact ($p(1)$.), so the rule is added, with no modification, to $modifiedRules$ (*line* 20). The second rule defining p is a recursive rule. It is worth noting that this rule will be processed twice, because its head contains two occurrences of predicate p (namely, $p(f(X))$ and $p(g(X))$). Thus, the first iteration of the outer *foreach* loop (*lines* $8 - 22$) causes the insertion of the following modified rule, because of $p(f(X))$:

$$p(f(X)) \vee p(g(X)) \text{ :- } magic_p(f(X)), magic_p(g(X)), p(X).$$

the above is added to the $modifiedRules$ set (*lines* $12 - 14$). Then the magic rules below are generated, one for the atom $p(g(X))$ appearing in the head, and one for the atom $p(X)$ appearing in the body, and the p predicate is pushed onto the stack S (*lines* $15 - 18$):

$$magic_p(g(X)) \text{ :- } magic_p(f(X)). \qquad magic_p(X) \text{ :- } magic_p(f(X)).$$

The following iteration of the outer *foreach* loop (due to the atom $p(g(X))$) would cause the same modified rule as above, which thus is not added twice to $ModifiedRules$. The following magic rules are generated, one for

$p(f(X))$ in the head, and one for $p(X)$ in the body, and the p predicate is pushed onto the stack S (*lines* $15-18$):

$$magic_p(f(X)) \text{ :- } magic_p(g(X)). \qquad magic_p(X) \text{ :- } magic_p(g(X)).$$

Then, a second and a third iteration of the main loop start; both immediately ends, as the predicate extracted from the stack S is the already considered predicate p. Finally, S is found empty; no further iterations are needed, and the algorithm outputs the following complete rewritten program:

$$\begin{array}{ll} p(f(X)) \vee p(g(X)) \text{ :- } magic_p(f(X)), magic_p(g(X)), p(X). & p(1). \\ magic_p(f(g(1))). & magic_p(g(X)) \text{ :- } magic_p(f(X)). \\ magic_p(X) \text{ :- } magic_p(f(X)). & magic_p(f(X)) \text{ :- } magic_p(g(X)). \\ magic_p(X) \text{ :- } magic_p(g(X)). & \end{array}$$

Example 5.3. Considering the query $Q = lessThan(s(s(0)), s(0))?$ on the program P_2 of Example 2.4, the algorithm outputs the following rewritten program P_3=$RW(Q, P_2)$:

$$\begin{array}{l} magic_lessThan(s(s(0)), s(0)). \\ magic_lessThan(X, Y) \text{ :- } magic_lessThan(X, s(Y)). \\ lessThan(X, s(X)) \text{ :- } magic_lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } magic_lessThan(X, s(Y)), lessThan(X, Y). \end{array} \tag{3}$$

6. Properties of Disjunctive Finitely-Recursive Programs

In this Section, we first discuss the relationship between DFRP programs and finitely-ground ($\mathcal{FG}$) programs [12]; then, we show some theoretical results about query answering on DFRP programs.

Some preliminary results are grouped in the next Lemma. In order to prove it, we introduce first the notion of partial evaluation of a ground program w.r.t. a set of magic atoms.

Definition 6.1. Given a program P and a set M of ground atoms of the form $magic_a(\bar{t})$ (where $a(\bar{t}) \in B_P$), we denote by $eval(P, M)$ the set of rules obtained from $grnd(P)$ as follows: remove from $grnd(P)$ any rule r such that some atom $magic_a(\bar{t}) \in B(r)$ and $magic_a(\bar{t}) \notin M$; remove any atom of the form $magic_a(\bar{t})$ from the body of the remaining rules.

Moreover, given a query Q on a program P, we denote as $magicRules(Q, P)$ and $modifiedRules(Q, P)$ the final value of the sets $modifiedRules$ and $magicRules$ after the application of MS_{FR} to Q and P.

Lemma 6.1. Let Q be a query on a program P. If Q is finitely recursive, then the following holds:

a. $magicRules(Q, P)$ has a unique answer set M_{mP};

b. $\{a(\bar{t}) \mid magic_a(\bar{t}) \in M_{mP}\} = relAtoms(Q, P)$;

c. M_{mP} is finite;

d. $magicRules(Q, P)$ is finitely ground;

e. $eval(modifiedRules(Q, P), M_{mP}) = relRules(Q, P)$.

Proof:

a. $magicRules(Q, P)$ is positive and normal ($\vee$-free), thus the statement follows.

b. Q is finitely recursive, thus there exists a value k, for which

$$relAtoms(Q, P) = \bigcup_{0 \leq i \leq k} RA_i(Q, P)$$

with $RA_i(Q, P)$ containing each atom a such that Q depends on a at degree i. Also observe that, since $mP = magicRules(Q, P)$ is positive and normal, any iterative bottom-up application of the immediate consequence operator $\mathcal{T}_{mP}$ produces a subset of the unique answer set of mP. The statement can hence be proved by showing that the i-th application of $\mathcal{T}_{mP}(\emptyset)$ on $magicRules(Q, P)$ (denoted by $\mathcal{T}^i_{mP}(\emptyset)$) derives *all and only* the atoms of the form $magic_a(\bar{t})$ s.t. $a(\bar{t}) \in RA_i(Q, P)$.

We prove this by induction. Let $Q = g(\bar{c})$. *(Basis.)* For $i = 0$, the only relevant atom is $g(\bar{c})$ itself, and $\mathcal{T}^0_{mP}(\emptyset) = \{magic_g(\bar{c})\}$. *(Induction.)* Assume that the statement holds for $i - 1$. Then, we first prove that $\mathcal{T}^i_{mP}(\emptyset)$ includes *all* atoms in the form $magic_a(\bar{t})$ s.t. $a(\bar{t}) \in RA_i(Q, P)$. Indeed, if there is an atom $a(\bar{t}) \in RA_i(Q, P)$, then there is some rule $r \in grnd(P)$ s.t. $a(\bar{t}) \in Atoms(r)$ and there is at least an atom $b(\bar{s}) \in H(r)$ s.t. $b(\bar{s}) \in RA_{i-1}(Q, P)$. Note that MS_{FR} builds mP in a way such that in $grnd(mP)$ there exists a rule of the form $magic_a(\bar{t}) \text{ :- } magic_b(\bar{s})$. By induction hypothesis, $magic_b(\bar{s}) \in \mathcal{T}^{i-1}_{mP}(\emptyset)$, thus $magic_a(\bar{t}) \in \mathcal{T}^i_{mP}(\emptyset)$. Analogous considerations give evidence that $\mathcal{T}^i_{mP}(\emptyset)$ contains *only* atoms of the form $magic_a(\bar{t})$ s.t. $a(\bar{t}) \in RA_i(Q, P)$.

c. Q is finitely recursive on P; then $relAtoms(Q, P)$ is finite, and the statement follows from point *(b)*.

d. As shown in the proof of *(b)*, the i-th application of $\mathcal{T}_{mP}(\emptyset)$ derives the atoms of the form $magic_a(\bar{t})$ s.t. Q depends on $a(\bar{t})$ at degree i. Since Q is finitely recursive on P, the number of these atoms is finite, and $\mathcal{T}_{mP}(\emptyset)$ converges finitely. By Proposition 3.1, mP is finitely ground.

e. Given Definition 6.1 and point *(b)* of this Lemma, one can observe that the rules participating in $E = eval(modifiedRules(Q, P), M_{mP})$ can contain in their heads only atoms belonging to $relAtoms(Q, P)$. Indeed, any rule $r \in modifiedRules(Q, P)$ is such that each atom in the form $magic_a(\bar{t})$ has a corresponding atom $a(\bar{t})$ appearing in the head of r;

Also, any atom of the form $magic_a(\bar{t})$ is removed from each rule $r \in E$. Any of such rules clearly belongs to $grnd(P)$ and has an head atom belonging to $relAtoms(Q, P)$, and thus belongs to $relRules(Q, P)$;

On the other hand if one considers a rule $r \in relRules(Q, P)$, similar arguments lead to conclude that since r has an atom $a \in relAtoms(Q, P)$ in its head, a corresponding rule r' must belong to $grnd(modifiedRules(Q, P))$ and then $r \in E$.

□

Theorem 6.1. Given a ground query Q on a program P, if Q is finitely recursive on P, then $RW(Q, P)$ is finitely ground.

Proof:
One can show that $RW(Q, P)$ is finitely ground by the following arguments:

1. Observe that $E = eval(modifiedRules(Q, P), M_{mP})$ is finite, by Lemma 6.1, point *(e)*, and that $magicRules(Q, P)$ is finitely ground by point *(d)* of the same Lemma.

2. Any component ordering $\gamma = \langle C_0, \ldots, C_n \rangle$ of $RW(Q,P)$ can be split into two partitions $\beta = \langle C_0, \ldots, C_k \rangle$ and $\tau = \langle C_{k+1}, \ldots, C_n \rangle$, where β contains only predicates of type $magic_a$, and identifies the module $magicRules(Q,P)$. Since this latter is finitely ground, its intelligent instantiation $(magicRules(Q,P))^{\beta}$ (for short, mP^{β}) is finite.

3. By proposition 3.1, mP^{β} is a set of facts corresponding to the unique model M_{mP} (Lemma 6.1, points *(a)*) of $magicRules(Q,P)$.

4. P^{γ} corresponds to the last step of the sequence $S_i = S_{i-1} \cup \Phi^{\infty}_{M_i,S_{i-1}}(\emptyset)$ for $k < i \leq n$, where $S_k = mP^{\beta}$ and M_i is the program module $P(C_i)$ associated to the component C_i.

5. Consider the finite set of ground rules

$$E' = \{\ v_1(\bar{t}_1) \vee \ldots \vee v_m(\bar{t}_m) \text{ :- } magic_v_1(\bar{t}_1), \ldots, magic_v_m(\bar{t}_m), v_{m+1}(\bar{t}_{m+1}), ..., v_n(\bar{t}_n).\ \textit{s.t.}\ v_1(\bar{t}_1) \vee \ldots \vee v_m(\bar{t}_m) \text{ :- } v_{m+1}(\bar{t}_{m+1}), ..., v_n(\bar{t}_n). \in E\ \}$$

   It can be stated that for any i s.t. $k < i \leq n$, $S_i \subseteq E'$, and and we prove it in the following.

   We know that $mP^{\beta} \subseteq S_{i-1}$, and $\Phi_{M_i,S_{i-1}}(X) = Simpl(Inst_{M_i}(Heads(S_{i-1} \cup X)), S_{i-1})$; furthermore, any rule in $M_i \subseteq \textit{modifiedRules}(Q,P)$ is either a ground fact, or a nonground rule r of the form

$$v_1(\bar{t}_1) \vee \ldots \vee v_m(\bar{t}_m) \text{ :- } magic_v_1(\bar{t}_1), \ldots, magic_v_m(\bar{t}_m), v_{m+1}(\bar{t}_{m+1}), ..., v_n(\bar{t}_n).$$

   If the latter is the case, when $Inst_{M_i}$ is applied to $Heads(S_{i-1} \cup X)$, the only way for instantiating atoms of type $magic_v_i(\bar{t}_i)$ in r is to consider the atom instances contained in $mP^{\beta} = M_{mP}$. Each ground rule coming from the instantiation of such magic atoms is clearly an element of E'. The subsequent application of the $Simpl$ operator, aiming at completing the computation of $\Phi_{M_i,S_{i-1}}(X)$, can possibly delete or modify some of such rules, but no new elements are added to S_i.

Summarizing: for any i s.t. $k < i \leq n$, we proved $S_i \subseteq E'$; this obviously means that also $S_n = P^{\gamma} \subseteq E'$. Since E' is finite, it turns out that P^{γ} is finite, and the statement follows. □

The next theorem provides query equivalence results for the magic-sets rewritten programs.

Theorem 6.2. Given a ground query Q on a program P, if Q is finitely recursive on P, then $P \models_b Q$ (resp., $P \models_c Q$) if and only if $RW(Q,P) \models_b Q$ (resp., $RW(Q,P) \models_c Q$).

Proof:
From Lemma 6.1, point (e), we know that $eval(modifiedRules(Q,P),\ M_{mP}) = relRules(Q,P)$. Thus, we have that $AS(RW(Q,P)) = AS(relRules(Q,P))$. Now, each answer set of $M \in relRules(Q,P)$ can be extended to an answer set of M' of P, while each answer set M' of P contains an answer set of $relRules(Q,P)$ (This can be shown e.g. by known tools such as the splitting theorem [25]). Thus, if we take brave (resp., cautious) reasoning in account, Q can appear in some (resp., all) $M \in AS(P)$ iff Q appears in some (resp., all) answer sets in $AS(relRules(Q,P))$, which, in turn, coincides with $AS(RW(Q,P))$. □

These results are quite relevant; they also imply that all nice properties of $\mathcal{FG}$ programs hold for rewritten finitely-recursive queries too. This includes, in particular, bottom-up computability of answer sets, and hence full decidability of reasoning, as stated from the next Corollary.

Corollary 6.1. Both cautious and brave reasoning on a DFRP program P are decidable.

Proof:
Any query on a finitely-recursive program P is finitely recursive. Hence, the result immediately follows from Theorem 6.1 above and Theorem 3.1. □

Example 6.1. The intelligent instantiation of the program P_3 of Example 5.3 is the following:

$$magic_lessThan(s(s(0)), s(0)).$$
$$magic_lessThan(s(s(0)), 0) :\text{-}\ magic_lessThan(s(s(0)), s(0)).$$

which is finite; hence, as expected, the originating rewritten program is finitely ground. It has the unique finite answer set $\{magic_lessThan(s(s(0)), s(0)), magic_lessThan(s(s(0)), 0)\}$, which is easily computable, thus allowing to answer to the query $Q_{5.3}$ of Example 5.3 with 'no'.

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

Definition 6.2. 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 atom occurrences in P. Note that, if the same atom occurs at two different places in P, it accounts for twice its size.

Example 6.2. The program P_2 in Example 2.4 has size $\|P\| = 8$.

Theorem 6.3. Given a finitely-recursive query Q on a program P, the size of $RW(Q, P)$ is linear in the size of the input P and Q; that is $\|RW(Q, P)\| = O(\|P\| + \|Q\|)$.

Proof:
The program $RW(Q, P)$ consists of the union of 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 head atoms in P (a magic atom per each head atom is added for each rule in P). In the worst case, $\|modifiedRules(Q, P)\|$ can be as large as $2\|P\|$. Thus, $\|modifiedRules(Q, P)\| = O(\|P\|)$.

Let us consider now the $magicRules(Q, P)$ program. 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, even in the worst case, the number of atoms in $magicRules(Q, P)$ is not greater than $2\|P\|$, plus the magic version of the query, which is just an atom.

It is worth noting that, as far as the algorithm is defined, arities of all new predicates are equal or less than the arities of original ones; furthermore, terms are left unchanged. Thus, from the considerations above, the statement $\|RW(Q, P)\| = O(\|P\| + \|Q\|)$ immediately follows. □

7. Related Work

There are many proposals for practically treating functional terms in ASP. These can be shortly classified in

1. *Syntactically restricted fragments*, such as *ω-restricted programs* [33], *λ-restricted programs* [17], *finite-domain programs* [12], *argument-restricted programs* [23], $\mathbb{FDNC}$ *programs* [31], *bidirectional programs* [15], and the proposal of [26]; these approaches introduce syntactic constraints (which can be easily checked at small computational cost) or explicit domain restrictions, thus allowing computability of answer sets and/or decidability of querying;

2. *Semantically restricted fragments*, such as *finitely ground programs* [12], *finitary programs* [6, 7], *disjunctive finitely-recursive programs* [3]; with respect to syntactically restricted fragments, these latter approaches aim at identifying broader classes of programs for which computational tasks, such as querying, enjoy some desirable property, such as decidability. However, recognizing such kind of programs is in general a Turing-complete task.

In particular, the works [3, 6, 12, 31] consider disjunctive programs. Works explicitly focussing on querying for disjunctive programs, and thus more related to ours, are [6] and [3].

The work in [6] studies how to extend finitary programs [7] to preserve decidability for ground querying in the presence of disjunction. To this end, a condition on disjunctive heads is added to the original definition of finitary program [7]. Given a dependency relation which considers only connections between head and body atoms (that is, $a \geqslant b$ iff there exists r such that $a \in H(r)$ and $b \in B(r)$), a disjunctive program P is finitary in the sense of [6] if *(1)* each ground atom in P depends on finitely many other atoms, *(2)* the set S of atoms appearing in odd-negated cycles is finite and *(3)* the set R of atoms a for which there is a rule $r \in P$ in which $a \in max_{\geqslant}(H(r))$ and there is an atom $b \in H(r)$ which is recursive with a and a positively depends on b, is finite [10].

Interestingly, the class of DFRP programs herein defined, which enjoys the decidability of reasoning (as proved in Theorem 6.1), enlarges the positive subclass of disjunctive finitary programs of [6]. Indeed, while all positive finitary programs trivially belong to the class of DFRP programs, the above mentioned third condition is not guaranteed to be fulfilled, although negation is forbidden, as witnessed by the following program:

$$
\begin{array}{ll}
p(X) \vee q(X) \coloneq s(X). & q(X) \coloneq p(X). \\
p(f(X)) \coloneq q(X). & p(1). \\
p(X) \coloneq q(X). &
\end{array}
$$

Baselice et al., in [3], consider instead a redefinition (including disjunction) of finitely-recursive programs, initially introduced in [7] as a super-class of finitary programs allowing function symbols and negation. The authors provide a compactness property result for such programs and some interesting semi-decidability results for cautious ground querying, but no decidability results about positive disjunctive programs. On the contrary, we focus on decidability results for disjunctive finitely-recursive positive programs, also providing an effective strategy for the actual computation of all ground reasoning tasks.

It is also worth mentioning that the introduction of functional terms (or similar constructs) have been studied in several other fields, besides Logic Programming, such as deductive databases (see $\mathcal{LDL}$ [28]); furthermore, studies on computable fragments of logic programs with functions are also related to termination studies of SLD-resolution for Prolog programs (see e.g. [30, 9, 10]).

Some other papers about the magic-set technique [1, 34, 4] are related to the present work as well, for which different extensions and refinements have been proposed. Among the more recent works, an adaptation for soft-stratifiable programs [5], the generalization to the disjunctive case [14] and to Datalog with (possibly unstratified) negation [16] are worth remembering.

8. Conclusions

In this work we have taken in consideration the problem of query answering over disjunctive finitely recursive positive (DFRP) programs, which encompasses many common encoding schemes, such as logic programs with non ground facts, possibly including disjunction. We proved that both brave and cautious reasoning on ground

[10] Given erratum [8], it turns out that both S and R must be known besides being finite.

queries are decidable, and illustrated how DFRP programs can be put in relationship with equivalent finitely-ground programs: this enables accomplishing querying on DFRP programs using standard bottom-up techniques, making viable the usage of such programs with standard ASP solvers. We thus enriched the family of logic programs with function symbols, for which ground query answering can be performed in practice.

References

[1] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J. D.: Magic Sets and Other Strange Ways to Implement Logic Programs, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, Massachusetts, 1986.

[2] Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003, ISBN 0-52181802-8.

[3] Baselice, S., Bonatti, P. A., Criscuolo, G.: On Finitely Recursive Programs, *TPLP*, **9**(2), 2009, 213–238.

[4] Beeri, C., Ramakrishnan, R.: On the Power of Magic, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '87)*, ACM, New York, NY, USA, 1987, ISBN 0-89791-223-3.

[5] Behrend, A.: Soft stratification for magic set based query evaluation in deductive databases, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Diego, CA, USA, 2003.

[6] Bonatti, P. A.: Reasoning with infinite stable models II: Disjunctive programs, *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, 2401, Springer, 2002.

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

[8] Bonatti, P. A.: Erratum to: "Reasoning with infinite stable models", *Artificial Intelligence*, **172**(15), 2008, 1833–1835.

[9] Bossi, A., Cocco, N., Fabris, M.: Norms on Terms and their use in Proving Universal Termination of a Logic Program, *Theoretical Computer Science*, **124**(2), 1994, 297–328.

[10] Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **29**(2), 2007, 10.

[11] Cabalar, P.: Partial Functions and Equality in Answer Set Programming, *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, 5366, Springer, Udine, Italy, December 2008, ISBN 978-3-540-89981-5.

[12] Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation, *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, 5366, Springer, Udine, Italy, December 2008, ISBN 978-3-540-89981-5.

[13] Calimeri, F., Cozza, S., Ianni, G., Leone, N.: `DLV-Complex` homepage, since 2008, `http://www.mat.unical.it/dlv-complex`.

[14] Cumbo, C., Faber, W., Greco, G., Leone, N.: Enhancing the Magic-Set Method for Disjunctive Datalog Programs, *Proceedings of the the 20th International Conference on Logic Programming – ICLP'04*, 3132, 2004.

[15] Eiter, T., Simkus, M.: Bidirectional Answer Set Programs with Function Symbols, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)* (C. Boutilier, Ed.), Pasadena, CA, USA, July 2009.

[16] 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.

[17] Gebser, M., Schaub, T., Thiele, S.: GrinGo : A New Grounder for Answer Set Programming, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07* (C. Baral, G. Brewka, J. Schlipf, Eds.), 4483, Springer Verlag, Tempe, Arizona, May 2007, ISBN 978-3-540-72199-4.

[18] Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming, *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, MIT Press, Cambridge, Mass., 1988.

[19] Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing*, **9**, 1991, 365–385.

[20] Hustadt, U., Motik, B., Sattler, U.: Reducing SHIQ-Description Logic to Disjunctive Datalog Programs, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada*, 2004.

[21] Lefèvre, C., Nicolas, P.: The First Version of a New ASP Solver : ASPeRiX, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)* (E. Erdem, F. Lin, T. Schaub, Eds.), 5753, Springer Verlag, September 2009, ISBN 978-3-642-04237-9.

[22] 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.

[23] Lierler, Y., Lifschitz, V.: One More Decidable Class of Finitely Ground Programs, *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, 5649, Springer, Pasadena, CA, USA, July 2009, ISBN 978-3-642-02845-8.

[24] Lifschitz, V.: Answer Set Planning, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)* (D. D. Schreye, Ed.), The MIT Press, Las Cruces, New Mexico, USA, November 1999.

[25] Lifschitz, V., Turner, H.: Splitting a Logic Program, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)* (P. Van Hentenryck, Ed.), MIT Press, Santa Margherita Ligure, Italy, June 1994.

[26] Lin, F., Wang, Y.: Answer Set Programming with Functions, *Proceedings of Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR2008)*, AAAI Press, Sydney, Australia, September 2008, ISBN 978-1-57735-384-3.

[27] Marek, V. W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm, in: *The Logic Programming Paradigm – A 25-Year Perspective* (K. R. Apt, V. W. Marek, M. Truszczyński, D. S. Warren, Eds.), Springer Verlag, 1999, 375–398.

[28] Naqvi, S., Tsur, S.: *A logical language for data and knowledge bases*, Computer Science Press, Inc., New York, NY, USA, 1989, ISBN 0-7167-8200-6.

[29] Ramakrishnan, R.: Magic Templates: A Spellbinding Approach To Logic Programs, *Journal of Logic Programming*, **11**(3&4), 1991, 189–216.

[30] Schreye, D. D., Decorte, S.: Termination of Logic Programs: The Never-Ending Story, *Journal of Logic Programming*, **19/20**, 1994, 199–260.

[31] Simkus, M., Eiter, T.: FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR2007)*, 4790, Springer, 2007, ISBN 978-3-540-75558-6.

[32] Swift, T.: Deduction in Ontologies via ASP, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)* (V. Lifschitz, I. Niemelä, Eds.), 2923, Springer, Fort Lauderdale, Florida, USA, January 2004, ISBN 3-540-20721-X.

[33] Syrjänen, T.: Omega-Restricted Logic Programs, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer-Verlag, Vienna, Austria, September 2001.

[34] Ullman, J. D.: *Principles of Database and Knowledge Base Systems*, vol. 2, Computer Science Press, 1989.

[35] Van Gelder, A., Ross, K. A., Schlipf, J. S.: The Well-Founded Semantics for General Logic Programs, *Journal of the ACM*, **38**(3), 1991, 620–650.