TPLP **19** (5–6): 654–670, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000115

Enhancing Magic Sets with an Application to Ontological Reasoning

MARIO ALVIANO, NICOLA LEONE, PIERFRANCESCO VELTRI and JESSICA ZANGARI
Department of Mathematics and Computer Science, University of Calabria, Italy
(*e-mail:* {alviano,leone,veltri,zangari}@mat.unical.it)

submitted 23 July 2019; accepted 31 July 2019

Abstract

Magic sets are a Datalog to Datalog rewriting technique to optimize query answering. The rewritten program focuses on a portion of the stable model(s) of the input program which is sufficient to answer the given query. However, the rewriting may introduce new recursive definitions, which can involve even negation and aggregations, and may slow down program evaluation. This paper enhances the magic set technique by preventing the creation of (new) recursive definitions in the rewritten program. It turns out that the new version of magic sets is closed for Datalog programs with stratified negation and aggregations, which is very convenient to obtain efficient computation of the stable model of the rewritten program. Moreover, the rewritten program is further optimized by the elimination of subsumed rules and by the efficient handling of the cases where binding propagation is lost. The research was stimulated by a challenge on the exploitation of Datalog/DLV for efficient reasoning on large ontologies. All proposed techniques have been hence implemented in the DLV system, and tested for ontological reasoning, confirming their effectiveness.

KEYWORDS: Datalog; query answering; magic sets; nonmonotonic reasoning; aggregations.

1 Introduction

Datalog is a rule based language for knowledge representation and reasoning suitable for a natural declaration of inductive definitions and ontological reasoning (Eiter et al. 2012). Several extensions to the core language of Datalog exist, among them default negation (Gelder 1989; Gelder et al. 1991; Gelfond and Lifschitz 1991) and aggregates (Simons et al. 2002; Pelov et al. 2007; Liu et al. 2010; Bartholomew et al. 2011; Ferraris 2011; Gelfond and Zhang 2014). Restrictions on the use of these linguistic constructs lead to preserve the existence and uniqueness of the stable model associated with a knowledge base; specifically, such restrictions essentially enforce a stratification on the definitions involving negation and aggregates (Faber et al. 2011). The semantics of the resulting language reached a broad consensus in the knowledge representation and reasoning community, as in fact the notions of *perfect model*, *well-founded model*, and *stable model* coincide for stratified programs (Przymusinski 1989; Gelder et al. 1991).

The stable model of a Datalog program can be constructed bottom-up, starting from facts in the program, and deriving new atoms from rules whose bodies become true. Negation and aggregates are handled by partitioning the input program into different

strata, so that the lowest stratum does not contain negation and aggregates, and each other stratum only negates and aggregates over predicates of lower strata. Such a bottom-up procedure is very efficient for producing the stable model, but it may be by itself inefficient for query answering. In fact, the stable model may contain atoms that are not relevant to answer the given query, and therefore constitute a source of inefficiency for query answering. In contrast, top-down procedures start from the query, and consider bodies of the rules defining the query predicate as subqueries. Hence, the computation focuses on a portion of the stable model that is relevant to answer the query.

The magic sets algorithm is a top-down rewriting of the input program that restricts the range of the object variables so that only the portion of the stable model that is relevant to answer the query is materialized by a bottom-up evaluation of the rewritten program (Bancilhon et al. 1986; Beeri and Ramakrishnan 1991; Balbin et al. 1991; Stuckey and Sudarshan 1994; Alviano et al. 2012). In a nutshell, magic sets introduce rules defining additional atoms, called *magic atoms*, whose intent is to identify relevant atoms to answer the input query, and these magic atoms are added in the bodies of the original rules to restrict the range of the object variables. Without going into much details, consider a typical recursive definition such as the ancestor relation:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

and a query `ancestor(mario,Y)` asking for the ancestors of `mario`. The extension of the `ancestor` relation is likely to contain several tuples that are not linked to `mario`, and are therefore irrelevant to answer the given query. To eliminate such a source of inefficiency, magic sets start with `m#ancestor#bf(mario)`, the *query seed*, which encodes the relevance of the instances of `ancestor(mario,Y)`; note that the first argument of `ancestor` is bound to constant `mario`, while the second argument is associated with a free variable, hence the predicate `m#ancestor#bf` (first argument bound, second argument free). After that, magic sets modify the rules defining the intentional predicate `ancestor`, and introduce *magic rules* for every occurrence of intentional predicates in the bodies of the modified rules. The rewritten program is the following:

```
m#ancestor#bf(mario).
ancestor(X,Y)    :- m#ancestor#bf(X), parent(X,Y).
ancestor(X,Y)    :- m#ancestor#bf(X), parent(X,Z), ancestor(Z,Y).
m#ancestor#bf(Z) :- m#ancestor#bf(X), parent(X,Z).
```

and limits the extension of `ancestor/2` to the tuples that are relevant to answer the given query.

Magic sets are sound and complete for the language considered in this paper (actually, for a broader language; Alviano et al. 2011). However, while on the one hand they are designed to inhibit the source of inefficiency associated with irrelevant atoms, on the other hand they may introduce different sources of inefficiencies, and also produce programs not satisfying the stratification of negation and aggregates. This paper identifies three of such sources of inefficiency, and propose strategies for their inhibition. Specifically, the major source of inefficiency is represented by the possible introduction of recursive definitions in the rewritten program.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

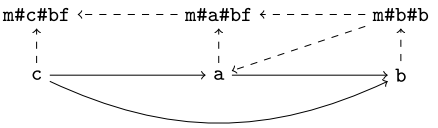

Fig. 1. Dependency graphs (defined in Section 2) of programs Π_1, Π_2, Π_3 (solid arcs) and programs Π'_1, Π'_2, Π'_3 (solid and dashed arcs) from Example 1.1. All arcs have weight 0, possibly with the exception of the arc connecting `a` and `b`, which has weight 1 for programs Π_2, Π'_2, Π_3, and Π'_3.

Example 1.1 (*Magic sets may introduce recursive definitions*)
Consider a query `c(0,Y)` for the following program Π_1:

```
r1 :  a(X,Y) :- edb(X,Y), b(X).
r2 :  b(X)   :- edb(X,Y).
r3 :  c(X,Y) :- a(X,Y), b(Y).
```

and a possible outcome Π'_1 of the magic sets rewriting:

```
r4 :  m#c#bf(0).
r5 :  m#a#bf(X) :- m#c#bf(X).
r6 :  m#b#b (Y) :- m#c#bf(X), a(X,Y).
r7 :  m#b#b (X) :- m#a#bf(X), edb(X,Y).
r8 :  a(X,Y) :- m#a#bf(X), edb(X,Y), b(X).
r9 :  b(X)   :- m#b#b (X), edb(X,Y).
r10 : c(X,Y) :- m#c#bf(X), a(X,Y), b(Y).
```

In particular, rule r_6 is produced while processing rule r_3 with variable `X` bound from the head atom, and considering variable `Y` bound by atom `a(X,Y)`. This is a common strategy, as there is no reason to consider an atom `b(y)` if no instance of `a(X,y)` is first computed. However, as shown in Figure 1, while all definitions in Π_1 are non-recursive, Π'_1 has recursive definitions for `a/2` and `b/1`, which may deteriorate the performance of the subsequent bottom-up evaluation. Following the same strategy, for a program Π_2 comprising r_2, r_3 and

```
r11 :  a(X,Y) :- edb(X,Y), not b(X).
```

the outcome of the magic sets rewriting Π'_2 comprises rules in $\Pi'_1 \setminus \{r_8\}$ and the following rule:

```
r12 :  a(X,Y) :- m#a#bf(X), edb(X,Y), not b(X).
```

Note that Π'_2 is not stratified with respect to negation. Similarly, for Π_3 comprising r_2, r_3 and

```
r13 :  a(X,Y) :- edb(X,Y), #sum{1 : b(X)} = 0.
```

the magic sets rewriting Π'_3 comprises rules in $\Pi'_1 \setminus \{r_8\}$ and the following rule:

```
r14 :  a(X,Y) :- m#a#bf(X), edb(X,Y), #sum{1 : b(X)} = 0.
```

Hence, Π'_3 is not stratified with respect to aggregations. ■

A second source of inefficiency that magic sets may introduce is represented by multiple versions of the original rules when the range of variables cannot be eventually restricted. For example, processing query `a(0)` and the following rule:

```
r15 :  a(X) :- b(X), a(Y), not c(X,Y).
```

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

necessarily leads to the presence of the following rules in the outcome of magic sets:

```
r16 :  m#a#b(0).
r17 :  m#a#f :- m#a#b(X).
r18 :  a(X)  :- m#a#b(X), b(X), a(Y), not c(X,Y).
r19 :  a(X)  :- m#a#f,    b(X), a(Y), not c(X,Y).
```

because variable `Y` is free when `a(Y)` is processed. Hence, in this case all instances of `a/1` in the stable model of the input program are relevant to answer the query in input. Nevertheless, when such a situation occurs, magic sets already produced restricted versions of the original rules, which are likely to decrease the performance of the subsequent bottom-up evaluation of the rewritten program.

The third source of inefficiency identified in this paper is represented by the possible presence of several copies of the same rule in the rewritten program, which is mainly due to different orders of body literals considered during the application of magic sets. While this fact is peculiar of one of the possible implementations of magic sets, it is also an opportunity to address a broader source of inefficiency that may already affect the input program, that is, the presence of *subsumed rules*. In a nutshell, a rule r subsumes another rule r' if the ground instances of r' are included or *less general* than the ground instances of r. For example, `q(X):- p(X,Y)` subsumes `q(X):- p(X,a)`, whose ground instances are among those of the first rule, and also `q(X):- p(X,Y), t(X)`, whose ground instances are less general than those of the first rule.

Summarizing the contributions of this paper, the source of inefficiency associated with the introduction of recursive definitions is inhibited by actively monitoring the dependency graph of the rewritten program, so to avoid the creation of new cycles during the production of magic rules (Section 3.1). The other two sources of inefficiency are instead addressed by processing the outcome of magic sets before executing the bottom-up evaluation. Specifically, if a predicate p is associated with different magic predicates, one of them with all arguments free, the rewritten program is simplified by removing all (useless) rules defining p and whose body contains a magic predicate restricting the range of object variables (Section 3.2). Concerning subsumed rules, they are identified by means of a backtracking algorithm, whose execution is often prevented by a more efficient but incomplete check based on hashed values and bitwise operations (Section 3.3). All the proposed strategies are implemented in DLV (Alviano et al. 2017; Adrian et al. 2018; Leone et al. 2019; Leone et al. 2019), whose magic sets algorithm can be now applied also for programs with stratified aggregates, and assessed empirically on domains involving ontological reasoning (Section 4).

2 Background

Syntax. A *term* is either a constant or an (object) variable. An *atom* has the form $p(\mathbf{t})$, where p is a *predicate* of arity $n \geq 0$, and $\mathbf{t}$ is a list of n terms. For a list $\mathbf{t}$, let $|\mathbf{t}|$ denote the length of $\mathbf{t}$, and $\mathbf{t}_i$ denote the i-th term of $\mathbf{t}$. A *literal* is an atom possibly preceded by the *(default) negation* symbol *not*; atoms are *positive literals*, while atoms preceded by *not* are *negative literals*. An *aggregate* has the form $\#\text{SUM}\{\mathbf{t}' : p(\mathbf{t})\} \odot t$, where $\mathbf{t}, \mathbf{t}'$ are lists of terms, t is a term, and $\odot$ is a comparator in $\{<, \leq, =, \neq, \geq, >\}$. A *rule* has the form

$$\alpha :\!- \ell_1, \ldots, \ell_n, A_1, \ldots, A_m,$$

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

where α is an atom, $n \geq 0$, $m \geq 0$, $\ell_1, \ldots, \ell_n$ are literals, and $A_1, \ldots, A_m$ are aggregates. For such a rule r, define the following notation: $H(r) := \alpha$, the *head* of r; $B(r) := \{\ell_1, \ldots, \ell_n, A_1, \ldots, A_m\}$, the *body* of r; $B^+(r) := \{\ell_i \mid i \in [1..n], \ell_i \text{ is a positive literal}\}$; $B^-(r) := \{\ell_i \mid i \in [1..n], \ell_i \text{ is a negative literal}\}$; $B^A(r) := \{A_i \mid i \in [1..m]\}$. Intuitively, $B(r)$ is interpreted as a conjunction, and we will use α :– $S \wedge S'$ to denote a rule r with $H(r) = \alpha$ and $B(r) = S \cup S'$; abusing of notation, we also permit S and S' to be literals. If $B(r)$ is empty, the symbol :– is usually omitted, and the rule is called a *fact*. A *program* Π is a set of rules. A predicate p occurring in Π is said *extensional* if all rules of Π with p in their heads are facts; otherwise, p is said *intentional*. For any *expression* (atom, literal, aggregate, rule, program) E, let $At(E)$ denote the set of atoms occurring in E. In the following, all programs are assumed to satisfy *safety of rules* and *stratification of negation and aggregates*, defined next.

Safety of rules. A *global variable* of a rule r is a variable X occurring in $H(r)$, $B^+(r)$, $B^-(r)$, or in an aggregate of the form $\#\text{SUM}\{\mathbf{t}' : p(\mathbf{t})\} \odot X$ in $B^A(r)$. All other variables occurring in r are *local variables* (to the aggregates where they occur). An *assignment variable* of a rule r is a variable X such that $B^A(r)$ contains an aggregate of the form $\#\text{SUM}\{\mathbf{t}' : p(\mathbf{t})\} = X$. A global variable X of r is *safe* if X is an assignment variable, or if X occurs in $B^+(r)$. A local variable X in an aggregate $\#\text{SUM}\{\mathbf{t}' : p(\mathbf{t})\} \odot t$ of r is *safe* if X occurs in $\mathbf{t}$. A rule is safe if all of its variables are safe. A program Π satisfies safety of rules if all of its rules are safe. All rules so far are safe; an unsafe rule is, for example, `a(X,Y):- b(X), not c(X,Y), #sum{Z : d(X,Y)} > 0`, as in fact the global variable `Y` and the local variable `Z` are unsafe.

Stratification of negation and aggregates. The *dependency graph* $\mathcal{G}_\Pi$ of a program Π has nodes for each predicate occurring in Π, and a weighted arc from p to q if there is a rule r of Π such that p occurs in $H(r)$, and q occurs in $B(r)$; the arc has weight 1 if q occurs in $B(r) \setminus B^+(r)$, and 0 otherwise. Π satisfies stratification of negation and aggregates if $\mathcal{G}_\Pi$ has no cycle involving arcs of positive weight. Figure 1 shows the dependencies graphs of the programs in Example 1.1.

Semantics. The *universe* U_Π of Π is the set comprising all integers, and the constants occurring in Π. The *base* B_Π of Π is the set of atoms constructible from predicates of Π with constants in U_Π. A *substitution* σ is a mapping from variables to variables and U_Π; for an expression E, let $E\sigma$ be the expression obtained from E by replacing each variable X by $\sigma(X)$. An expression is *ground* if it contains no global variables. Let $ground(\Pi)$ be $\bigcup_{r \in \Pi}\{r\sigma \mid \sigma \text{ is a substitution, and } r\sigma \text{ is ground}\}$. An *interpretation* I is a subset of B_Π. Relation $\models$ is defined as follows: for a ground atom α, $I \models \alpha$ if $\alpha \in I$, and $I \models \mathit{not}\ \alpha$ if $I \not\models \alpha$; for an aggregate $A := \#\text{SUM}\{\mathbf{t}' : p(\mathbf{t})\} \odot t$ occurring in $ground(\Pi)$, $I \models A$ if $\sum_{\mathbf{t}'\sigma : p(\mathbf{t})\sigma \in I} \mathbf{t}'_1\sigma \odot t$; for a ground rule r, $I \models B(r)$ if $I \models \ell$ for all $\ell \in B(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; finally, $I \models ground(\Pi)$ if $I \models r$ for all $r \in ground(\Pi)$. The *(FLP) reduct* of Π with respect to I, denoted Π^I, is the program obtained from Π by removing rules with false bodies, that is, $\Pi^I := \{r \in \Pi \mid I \models B(r)\}$ (Faber et al. 2011). Given a program Π, the *stable model* of Π is the unique interpretation I such that $I \models ground(\Pi)$, and there is no $J \subset I$ such that $J \models ground(\Pi)^I$; let $SM(\Pi)$ denote the stable model of Π. (The stable model of Π can be computed bottom-up as described

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

in the introduction. A formal definition of such a procedure is out of the scope of this paper.)

Example 2.1
Consider the following program in the context of an online shopping site:

```
order(o1). item(o1,i1,20). item(o1,i2,20).
order(o2). cancelled(o2).
total_cost(S) :- order(O), not cancelled(O), #sum{P,I : item(O,I,P)} = S.
```

The stable model of the above program contains facts and `total_cost(40)`, as indeed the only ground rule with true, nonempty body is the following:

```
total_cost(40) :- order(o1), not cancelled(o1), #sum{P,I : item(o1,I,P)} = 40.
```

In particular, note that for $\sigma(\mathtt{O}) \notin \{\mathtt{o1}, \mathtt{o2}\}$ literal $\mathtt{order(O)}\sigma$ is false, for $\sigma(\mathtt{O}) = \mathtt{o2}$ literal `not cancelled(o2)` is false, and for $\sigma(\mathtt{O}) = \mathtt{o1}$ and $\sigma(\mathtt{S}) \neq \mathtt{40}$ the aggregate is false. ■

Queries and magic sets. A query is an atom $q(\mathbf{t})$. Let $answer(q(\mathbf{t}), \Pi)$ be $\{\mathbf{t}\sigma \mid q(\mathbf{t})\sigma \in SM(\Pi)\}$, that is, the answer to the query $q(\mathbf{t})$ over the program Π is the set of ground instances of $q(\mathbf{t})$ in the stable model of Π. The magic sets algorithm aims at transforming program Π into a program Π' such that $answer(q(\mathbf{t}), \Pi) = answer(q(\mathbf{t}), \Pi')$, and $SM(\Pi') \cap At(\Pi) \subseteq SM(\Pi)$; in words, the two programs have the same answer to the query $q(\mathbf{t})$, but the stable model of Π' only contains atoms that link facts to the query. The algorithm relies on *adornments* and *magic atoms* to represent binding information that a top-down evaluation of the query would produce.

Definition 2.1 (*Adornments and magic atoms*)
An adornment for a predicate p of arity k is any string $\mathbf{s}$ of length k over the alphabet $\{b, f\}$. The i-th argument of p is *bound* with respect to $\mathbf{s}$ if $\mathbf{s}_i = b$, and *free* otherwise, for all $i \in [1..k]$. For an atom $p(\mathbf{t})$, let $p^{\mathbf{s}}(\mathbf{t})$ be the (magic) atom $m\#p\#\mathbf{s}(\mathbf{t}')$, where $m\#p\#\mathbf{s}$ is a predicate not occurring in the input program, and $\mathbf{t}'$ contains all terms in $\mathbf{t}$ associated with bound arguments according to $\mathbf{s}$.

Definition 2.2 (*Sideways information passing strategy; SIPS*)
A SIPS for a rule r with respect to an adornment $\mathbf{s}$ for $H(r)$ is a pair $(\prec, bnd)$, where $\prec$ is a strict partial order over $\{H(r)\} \cup B(r)$, and bnd maps $\ell \in \{H(r)\} \cup B(r)$ to the variables of ℓ that are made bound after processing ℓ. Moreover, a SIPS satisfies the following conditions:

- $H(r) \prec \ell$ for all $\ell \in B(r)$ (binding information originates from head atoms);
- $\ell \prec \ell'$ and $\ell \neq H(r)$ implies that either $\ell \in B^+(r)$ or ℓ is an aggregate with assignment (new bindings are created only by positive literals and assignments);
- $bnd(H(r))$ contains the variables of $H(r)$ associated with bound arguments according to $\mathbf{s}$;
- $bnd(\ell) = \emptyset$ if ℓ is a negative literal, or an aggregate without assignment variable;
- $bnd(\ell) \subseteq \{X\}$ if ℓ is an aggregate with assignment variable X.

Example 2.2 (*Magic atoms and SIPS*)
According to Definition 2.1, $\mathtt{c}^{bf}\mathtt{(0,Y)}$ is the magic atom `m#c#bf(0)`. Using the notation introduced in Definition 2.2, the SIPS for r_3 with respect to the adornment bf adopted

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Algorithm 1: MS($Q(\mathbf{T})$: a query atom, Π: a program)

```
1  Let s be such that |s|=|T|, and s_i = b if T_i is a constant, and f otherwise, for all i ∈ [1..|s|];
2  Π' := {Q^s(T).};                          // rewritten program: start with the magic seed
3  S := {⟨Q, s⟩};                            // set of produced adorned predicates
4  D := ∅;                                   // set of processed (or done) adorned predicates
5  while S ≠ D do
6      ⟨q, s⟩ := any element in S \ D;       // select an undone adorned predicate
7      foreach r ∈ Π such that H(r) = q(t) for some list t of terms do
8          Π' := Π' ∪ {q(t) :- q^s(t) ∧ B(r).};   // restrict range of variables
9          Let (≺, bnd) be the SIPS for r with respect to s;
10         foreach ℓ ∈ B(r) such that p(t') ∈ At(ℓ) and p is an intentional predicate of Π do
11             Let s' be such that |s'| = |t'|, and s'_i = b if t'_i is a constant or belongs to
                 bnd(ℓ') for some ℓ' ≺ ℓ, and f otherwise, for all i ∈ [1..|s'|];
12             Π' := Π' ∪ {p^s'(t') :- q^s(t) ∧ {ℓ' ∈ B(r) | ℓ' ≺ ℓ}.};   // add magic rule
13             S := S ∪ {⟨p, s'⟩};           // keep track of produced adorned predicates
14     D := D ∪ {⟨q, s⟩};                    // flag the adorned predicate as done
15 return Π';
```

in Example 1.1 is such that $c(X,Y) \prec a(X,Y) \prec b(Y)$, $bnd(c(X,Y)) = \{X\}$, $\{Y\} \subseteq bnd(a(X,Y)) \subseteq \{X,Y\}$ (i.e., variable `Y` is bound after processing `a(X,Y)`), and $\emptyset \subseteq bnd(b(Y)) \subseteq \{Y\}$ (i.e., whether `Y` is bound after processing `b(Y)` is irrelevant). ■

The magic sets procedure is reported as Algorithm 1. It starts by producing the *magic seed*, obtained from the predicate and the constants in the query. After that, the algorithm processes each produced adorned predicate: each rule defining the predicate is modified so to restrict the range of the head variables to the tuples that are relevant to answer the query; such a relevance is encoded by the magic rules, which are produced for all intentional predicates in the bodies of the modified rules.

Proposition 2.1 (*Theorem 5 of Alviano et al. 2011*)
Let $q(\mathbf{t})$ be a query for a program Π, and Π' be the output of MS$(q(\mathbf{t}), \Pi)$. Thus, $answer(q(\mathbf{t}), \Pi)$ and $answer(q(\mathbf{t}), \Pi')$ are equal.

3 Improved strategies for the magic sets algorithm

The three sources of inefficiency of magic sets that have been identified in the introduction are detailed and addressed in this section.

3.1 Inhibit new cycles

Magic sets may introduce new cycles in the dependency graph of the processed program, as shown in Example 1.1. Such new cycles are due to the binding information passed by body literals to other body literals, and therefore strictly dependent from the adopted SIPS. In fact, new cycles can be inhibited by a drastic restriction on all SIPS $\langle \prec, bnd \rangle$

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Algorithm 2: MS-RS($Q(\mathbf{T})$: a query atom, Π: a program)

```
1  Let s be such that |s|=|T|, and s_i = b if T_i is a constant, and f otherwise, for all i ∈ [1..|s|];
2  Π' := {Q^s(T).};                          // rewritten program: start with the magic seed
3  S := {⟨Q, s⟩};                            // set of produced adorned predicates
4  D := ∅;                                   // set of processed (or done) adorned predicates
5  G := G_Π ∪ {⟨p, m#p⟩ | p is a predicate occurring in Π};          // monitor SCCs
6  while S ≠ D do
7      ⟨q, s⟩ := any element in S \ D;       // select an undone adorned predicate
8      foreach r ∈ Π such that H(r) = q(t) for some list t of terms do
9          Π' := Π' ∪ {q(t) :- q^s(t) ∧ B(r).};          // restrict range of variables
10         Let (≺, bnd) be the SIPS for r with respect to s;
11         foreach ℓ ∈ B(r) such that p(t') ∈ At(ℓ) and p is an intentional predicate of Π do
12             G := G ∪ {⟨m#p, m#q⟩};
13             B := ∅;          // restrict SIPS to preserve strongly connected comp.
14             foreach ℓ' ∈ B(r) such that ℓ' ≺ ℓ and p'(t'') ∈ At(ℓ') do
15                 if {C ∩ At(Π) | C ∈ SCCs(G ∪ {⟨m#p, p'⟩})} = SCCs(G_Π) then
16                     B := B ∪ {ℓ'};   G := G ∪ {⟨m#p, p'⟩};
17             Let s' be such that |s'| = |t'|, and s'_i = b if t'_i is a constant or belongs to bnd(ℓ')
                for some ℓ' ∈ {H(r)} ∪ B such that ℓ' ≺ ℓ, and f otherwise, for all i ∈ [1..|s'|];
18             Π' := Π' ∪ {p^{s'}(t') :- q^s(t) ∧ B.};                    // add magic rule
19             S := S ∪ {⟨p, s'⟩};          // keep track of produced adorned predicates
20     D := D ∪ {⟨q, s⟩};                    // flag the adorned predicate as done
21 return Π';
```

enforcing $\ell \not\prec \ell$' for all ℓ, ℓ' in $B(r)$: this way, all magic rules would contain only magic atoms, and therefore no arc from magic predicates to original predicates would be introduced in the dependency graph. However, the drastic restriction is likely to significantly reduce the benefit of magic sets, as the stronger the restriction on SIPS is, the more atoms are considered relevant to answer a given query. Hence, the goal of this section is to introduce a more relaxed restriction on SIPS, which just prevents the creation of new cycles, but still admit the introduction of new dependencies.

For a graph G and a set of arcs E, let $G \cup E$ denote the graph obtained from G by adding each arc in E. Moreover, let $SCCs(G)$ be the set of *strongly connected components (SCC)* of G, where a SCC of G is a maximal set C of nodes of G such that G contains a path from every $p \in C$ to every $q \in C \setminus \{p\}$. A revised version of magic sets enforcing a restriction on SIPS is shown as Algorithm 2. Note that lines 5 and 12–16 implement a restriction of SIPS guaranteeing that no strongly connected components of $\mathscr{G}_\Pi$ are merged during the application of magic sets. Specifically, a graph G is initialized with the arcs of $\mathscr{G}_\Pi$ and arcs connecting each predicate p with a *representative magic predicate* $m\#p$ (line 5). After that, before creating a new magic rule, elements of $B(r)$ that would cause a change in the strongly connected components of G are discarded (lines 13–16). Graph G is updated with new arcs involving original predicates and representative magic predicates, so that it represents a superset of the graph obtained from $\mathscr{G}_{\Pi'}$ by merging all pairs of nodes of the form $m\#p\#\mathbf{s}$, $m\#p\#\mathbf{s}'$.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Example 3.1
Consider Π_1, query `c(0,Y)`, and SIPS from Example 1.1. Algorithm 2 returns the following program:

```
r20 : m#c#bf(0).
r21 : m#a#bf(X) :- m#c#bf(X).
r22 : m#b#f     :- m#c#bf(X).
r23 : m#b#b (X) :- m#a#bf(X), edb(X,Y).
r24 : a(X,Y) :- m#a#bf(X), edb(X,Y),b(X).
r25 : b(X)   :- m#b#f,     edb(X,Y).
r26 : b(X)   :- m#b#b(X),  edb(X,Y).
r27 : c(X,Y) :- m#c#bf(X), a(X,Y), b(Y).
```

Note that rule r_6 from Example 1.1 is replaced by rule r_{22}, so to avoid the creation of a cycle involving `a` and `b`. Note also that predicate `b` is now associated with two magic predicates, which may reduce the performance of a bottom-up evaluation; this source of inefficiency is addressed in the next section. ■

Theorem 3.1
Let $q(\mathbf{t})$ be a query for a program Π, and Π' be the output of $\mathrm{MS}(q(\mathbf{t}), \Pi)$ with restricted SIPS. Thus, $answer(q(\mathbf{t}), \Pi)$ and $answer(q(\mathbf{t}), \Pi')$ are equal. Moreover, if $C' \in SCCs(\Pi')$, then there is $C \in SCCs(\Pi)$ such that $C' \cap At(\Pi) \subseteq C$.

Proof
Equality of $answer(q(\mathbf{t}), \Pi)$ and $answer(q(\mathbf{t}), \Pi)$ is a consequence of the correctness of magic sets for any choice of SIPS (Proposition 2.1). In fact, the restriction on SIPS applied by algorithm MS-RS still results into SIPS. For $C' \in SCCs(\mathscr{G}_{\Pi'})$, we shall show that there is $C \in SCCs(\mathscr{G}_{\Pi})$ such that $C' \cap At(\Pi) \subseteq C$. Actually, there is $C \in SCCs(G)$ such that $C' \cap At(\Pi) \subseteq C \cap At(\Pi)$. Hence, the claim follows from the fact that $C \cap At(\Pi) \in SCCs(\mathscr{G}_{\Pi})$ is enforced by the condition in line 15 of Algorithm 2. □

An immediate consequence of the above theorem is that magic sets with restricted SIPS are a closed rewriting for the class of programs with stratified negation and aggregations.

3.2 Handle full-free adornments

Adornments containing only fs are produced in presence of predicates whose arguments are all free. In such cases, all of the extension of the predicate in the stable model of the input program is relevant to answer the given query. It turns out that the range of the object variables of all rules defining such predicates cannot be restricted, and indeed the magic sets rewriting includes a copy of these rules with a magic atom obtained from the full-free adornment. Possibly, the magic sets rewriting includes other copies of these rules obtained by different adornments, which can be removed if magic rules are properly modified. Specifically, magic rules associated with predicates for which a full-free adornment has been produced have to become definitions of the magic atom obtained from the full-free adornment. The strategy is summarized in Algorithm 3, and can be efficiently implemented in two steps: a first linear traversal of the program to identify predicates of the form $m\#p\#f\cdots f$ and to flag predicate p; a second linear traversal of the program to remove and rewrite rules with predicate $m\#p\#\mathbf{s}$, for all flagged predicates p.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Algorithm 3: FullFree(Π: a program obtained by executing magic sets)

```
foreach m#p#f···f occurring in Π do
    foreach m#p#s occurring in Π such that s ≠ f···f do
        remove all rules of Π having m#p#s in their bodies;
        replace m#p#s(t) by m#p#f···f in all rule heads of Π;
return Π;
```

Example 3.2

Consider rule r_{15} from the introduction, `a(X):- b(X), a(Y), not c(X,Y)`, and its magic sets rewriting with respect to query `a(0)`:

```
r16 : m#a#b(0).
r17 : m#a#f :- m#a#b(X).
r18 : a(X)  :- m#a#b(X), b(X), a(Y), not c(X,Y).
r19 : a(X)  :- m#a#f,    b(X), a(Y), not c(X,Y).
```

Algorithm 3 removes rules r_{17} and r_{18} because of `m#a#b(X)` in their bodies, and replaces rule r_{16} with the fact `m#a#f`. ■

Theorem 3.2

Let $q(\mathbf{t})$ be a query for a program Π, and Π′ be the output of FullFree(MS($q(\mathbf{t}), \Pi$)). Thus, $answer(q(\mathbf{t}), \Pi)$ and $answer(q(\mathbf{t}), \Pi')$ are equal.

Proof

Let I be $SM(\Pi'')$. The stable model of Π′ is obtained from I by performing the following operation for all $m\#p\#f\cdots f$ occurring in Π″: replace all instances of $m\#p\#\mathbf{s}$ by $m\#p\#f\cdots f$. □

3.3 Efficiently detect subsumed rules

A rule r *subsumes* a rule r', denoted $r \sqsubseteq r'$, if there is a substitution σ such that $H(r)\sigma = H(r')$ and $B(r)\sigma \subseteq B(r')$. Subsumed rules are redundant in the sense that any atom derivable from r' is also derived from r if $r \sqsubseteq r'$; indeed, for any substitution θ and interpretation I such that $B(r')\theta$ is ground and $I \models B(r')\theta$, it holds that $B(r)\sigma\theta$ is ground, $I \models B(r)\sigma\theta$ (because $B(r)\sigma\theta \subseteq B(r')\theta$), and $H(r)\sigma\theta = H(r')\theta$. Hence, $r \sqsubseteq r'$ implies $SM(\Pi) = SM(\Pi \setminus \{r'\})$, and therefore all subsumed rules can be removed from a program before starting its bottom-up evaluation. However, checking subsumption is NP-complete in general, and therefore computationally expensive if ran for all pairs of rules in a program.

Algorithm 4: Subsumption(Π: a program)

```
foreach distinct r, r′ ∈ Π such that hash(r) & hash(r′) = hash(r) do
    if subsumes(r, r′) then remove r′ from Π;
return Π;
```

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Function Subsumes(r, r')

$S := [\langle \text{OneWayUnify}(H(r),\ H(r')),\ B(r)\rangle]$;
while $S \neq \emptyset$ **do**
  $\langle \sigma,\ B\rangle := S.\text{pop}()$;
  if σ *is a function* **then**
    if $B = \emptyset$ **then return** *true*;
    foreach $\ell \in B$ *and* $\ell' \in B(r')$ **do** $S.\text{push}(\langle \sigma \cup \text{OneWayUnify}(\ell,\ \ell'),\ B \setminus \{\ell\}\rangle)$;
return *false*;

Function OneWayUnify(ℓ, ℓ')

if ℓ *and* ℓ' *have different predicates, or are not both positive literals, negative literals, or aggregates* **then return** $\{X \mapsto 0, X \mapsto 1\}$;
Let $\mathbf{t}$ and $\mathbf{t}'$ be the terms in ℓ and ℓ' (for aggregates, symbol : is considered as a constant);
if $|\mathbf{t}| \neq |\mathbf{t}'|$, *or* $\exists i \in [1..|\mathbf{t}|]$ *s.t.* $\mathbf{t}_i$ *is a constant and* $\mathbf{t}_i \neq \mathbf{t}'_i$ **then return** $\{X \mapsto 0, X \mapsto 1\}$;
return $\{\mathbf{t}_i \mapsto \mathbf{t}'_i \mid i \in [1..|\mathbf{t}|], \mathbf{t}_i$ *is a variable*$\}$; `// possibly a function`

The number of performed checks is significantly reduced by means of an hash function that associates each rule with a bit string of fixed length and satisfying the following invariant:

$$\text{if } \mathit{hash}(r)\ \&\ \mathit{hash}(r') \neq \mathit{hash}(r),\ \text{then } r \not\sqsubseteq r' \tag{1}$$

where & is the bitwise AND operator. Specifically, the hash value associated with a rule is designed to be a string of 64 bits computed as follows from the less significant bits of predicate ids and constant ids occurring in the rule: 8 bits for the bitwise OR of predicate ids in $H(r)$ (only one predicate for the language considered in this paper); 8 bits for the bitwise OR of constants ids in $H(r)$; 16 bits for the bitwise OR of predicate ids in $B^+(r) \cup B^A(r)$; 16 bits for the bitwise OR of constant ids in $B^+(r) \cup B^A(r)$; 8 bits for the bitwise OR of predicate ids in $B^-(r)$; 8 bits for the bitwise OR of constants ids in $B^-(r)$.

The idea underlying the above hash function is that all constants and predicates occurring in $H(r)$, $B^+(r) \cup B^A(r)$ and $B^-(r)$ have to also occur in $H(r')$, $B^+(r') \cup B^A(r')$ and $B^-(r')$ in order to have $r \sqsubseteq r'$. The invariant (1) eventually detects pairs of rules not satisfying this property, so to avoid the more expensive backtracking procedure for them. Algorithm 4 summarizes the strategy implemented for removing subsumed rules from programs: when the condition on the hash values is satisfied, use backtracking to build a substitution σ.

Example 3.3
Consider the following rules from the introduction:

r : `q(X) :- p(X,Y).` r' : `q(X) :- p(X,a).` r'' : `q(X) :- p(X,Y), t(X).`

and the following predicate and constant ids: $id(q) = 01, id(p) = 10, id(t) = 11, id(a) = 01$, where for simplicity only 2 bits are used. The hash values of the rules above (using only 2 bits for each portion of the hash value) are the following:

- $\mathit{hash}(r\ ) = 010010000000$;
- $\mathit{hash}(r'\ ) = 010010010000$;
- $\mathit{hash}(r'') = 010011000000$.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Note that $hash(r') \,\&\, hash(r'') = 010010000000 \neq hash(r')$, and in fact $r' \not\sqsubseteq r''$. On the other hand, $hash(r) \,\&\, hash(r') = 010010000000 = hash(r)$, and $r \sqsubseteq r'$. ■

Theorem 3.3
Invariant (1) is satisfied by the proposed hash function.

Proof
Let $hash(r) \,\&\, hash(r') \neq hash(r)$. Hence, there is $i \in [1..64]$ such that $hash(r) = 1$ and $hash(r') = 0$. If $i \in [1..8]$, then predicate ids of $H(r)$ and $H(r')$ disagree on their less significant 8 bits, and therefore they are necessarily different predicates; thus, $r \not\sqsubseteq r'$ holds. If $i \in [9..16]$, then $H(r)$ contains a constant whose $(i-8)$-th less significant bit is 1, while no constant in $H(r')$ has this property; it turns out that $H(r)$ has a constant not occurring in $H(r')$, and therefore $r \not\sqsubseteq r'$ holds also in this case. The remaining cases are similar. □

4 Experiment

The proposed enhancements are implemented in I-DLV 1.1.4, and compared against the performance of the previous magic sets rewriting implemented in I-DLV 1.1.3. Binaries are available at https://github.com/DeMaCS-UNICAL/I-DLV/releases. The experiment comprises synthetic instances from Example 1.1 with facts `edb(0..1000000*size)`, where `size` ranges in [1..10], to show the potential impact of the prevention of new cycles. Additional instances are obtained from LUBM (http://swat.cse.lehigh.edu/projects/lubm/) by generating instances with `50*size` universities, where `size` ranges in [1..20], with the aim to measure the impact of the hashing technique to prevent subsumption checks. All tests were run on a Dell server with 8 CPU Intel Xeon Gold 6140 2.30GHz, RAM 297GB, and HDD 3.29TB 7200rpm. Each test was limited to 1200 seconds of execution time and 250GB of memory consumption.

Concerning the scalability tests, time and memory usage are plotted on Figure 2. For program Π_1, the execution time of the new rewriting is around 62% of the execution time of the previous rewriting on average; similarly, the new rewriting only used around 68% of the memory required by the previous rewriting. These results confirm that the proposed restriction to SIPS may lead computational advantages also for positive programs. The advantage is much more evident for program Π_2, that is, the one for which negative cycles may be introduced by magic sets. Indeed, in this case the new rewriting only needs around 12% of the execution time and around 16% of the memory required by the previous rewriting on average. Finally, concerning program Π_3, the previous rewriting could not be tested because it introduced recursive aggregates, as expected; the new rewriting, instead, performed as for Π_2, with an average execution time of 4.5 seconds and an average memory consumption of 1.4 GB.

As for LUBM, its Datalog encoding consists of 132 rules and 83 predicate names, which become 382 rules and 216 predicate names after running magic sets as implemented in I-DLV 1.1.3. Within I-DLV 1.1.4, instead, the magic sets rewriting comprises 210 rules and 118 predicate names. In fact, several rules and predicate names are removed because of full-free adornments. A few additional rules, precisely 17, are filtered out by the subsumption checks. Within this respect, it is interesting to observe that the number of subsumption checks to perform without the hashing technique presented in Section 3.3 is

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

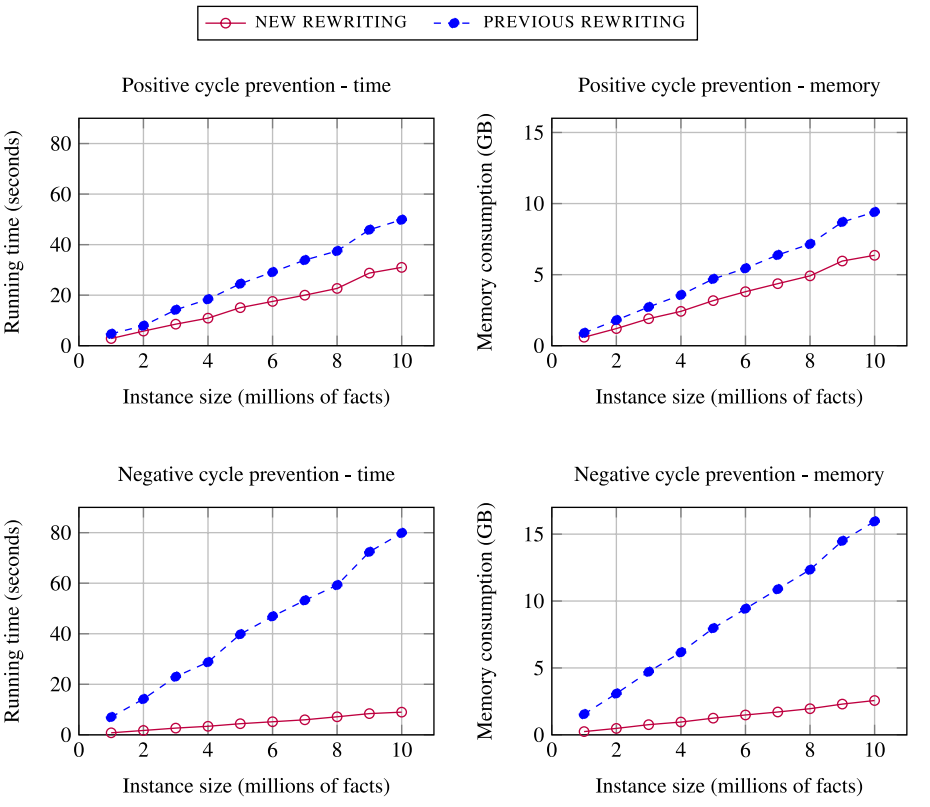

Fig. 2. Scalability results with respect to time (left) and memory (right)

37 600, in contrast to a significantly smaller number of 1 159 checks actually performed; the hashing technique reduced by around 97% the number of subsumption checks. Finally, concerning execution time, both versions scale almost linearly, with a slight advantage of the new magic sets: I-DLV 1.1.3 reported an average execution time of around 529 seconds, with a minimum of around 42 seconds and a maximum of around 1060; I-DLV 1.1.4 reported an average execution time of around 502 seconds, with a minimum of around 40 seconds and a maximum of around 1027.

5 Related work

Magic sets were originally introduced for Datalog programs (Bancilhon et al. 1986), and applied among other contexts to bottom-up analysis of logic programs (Codish and Demoen 1995) and BDD-Based Deductive DataBases (Whaley et al. 2005). Extending the technique to Datalog programs with stratified negation was nontrivial, as the perfect model semantics is not applicable to the rewritten program if recursive negation is introduced by magic sets. Several semantics were considered in the literature to overcome the limitation of perfect model semantics. Among them, some authors defined ad-hoc semantics for rewritten programs (Kerisit and Pugin 1988; Balbin et al. 1991; Behrend 2003), while Kemp et al. (1995) and Ross (1994) considered well-founded semantics, and showed that the well-founded model of any rewritten program obtained from a Datalog program with stratified negation is two-valued.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

A similar semantic issue arises for aggregations (Mumick et al. 1990; Furfaro et al. 2002), as there is no general consensus for recursive aggregates (Alviano and Faber 2018). This fact explains why DLV did not support (dynamic) magic sets (Alviano et al. 2012) for programs with aggregates, even if their correctness was shown also for programs with some form of aggregation (Alviano et al. 2011). In fact, even if techniques to process programs with recursive aggregates are known (Gebser et al. 2009; Alviano et al. 2015; 2016), they are in general less efficient than those for stratified aggregates; for example, *shared aggregate sets* (Alviano et al. 2018) are currently implemented in WASP (Dodaro et al. 2011; Alviano et al. 2019) only in the stratified case.

Magic sets were applied to other extensions of Datalog, in particular to disjunctive Datalog under stable model semantics (Greco 2003; Greco et al. 2005). For disjunctive Datalog, dynamic magic sets push the optimization on all phases of the computation of stable models (Alviano et al. 2012), and are shown to be correct for a semantic class known as *super-coherent* programs (Alviano and Faber 2011; Alviano et al. 2014). The restriction on SIPS applied in Section 3.1 necessarily limits the optimization of dynamic magic sets to the grounding phase, which is anyhow the only computation phase for the language considered in this paper. On the other hand, the restriction on SIPS presented in this paper does not inhibit the application of magic sets to programs characterized by multiple stable models: magic sets would still optimize the grounding of those programs, so that other highly optimized techniques for computing *cautious consequences* of propositional programs can be employed (Alviano et al. 2014; Alviano et al. 2018), among them those based on *unsatisfiable core analysis* (Alviano and Dodaro 2016; Alviano and Dodaro 2017).

6 Conclusion

Magic sets aim at optimizing query answering, but they may introduce recursive definitions that possibly deteriorate the performance of a bottom-up evaluation of the rewritten program. Previous works in the literature noted the problem for programs with stratified negation, and proposed several solutions to the associated semantic issue. By imposing some restriction on SIPS, this paper provides a simple solution to semantic issues arising for programs with stratified negation and aggregations, which also inhibits the creation of new positive recursive definitions (Section 3.1). The role of magic atoms is to restrict the range of variables in the original rules of the processed program. When all arguments of a predicate have to be considered free, a full-free adornment is generated. Any other adornment associated with such a predicate only introduces overhead in the evaluation of the rewritten program. This paper proposes a post-processing of the rewritten program to purge full-free adornments, in contrast to more complex unroll procedures (Section 3.2). Further overhead is associated with subsumed rules. Their identification is nontrivial and addressed by a backtracking algorithm. Even if there are few branching points, actually only if there are multiple occurrences of the same predicate in rule bodies, running the backtracking algorithm for all pairs of rules in the rewritten program is expensive. The hashing technique given in Section 3.3 provides a drastic reduction on the number of checks.

https://doi.org/10.1017/S1471068419000115 Published online by Cambridge University Press

Acknowledgments

This work has been partially supported by MIUR under project "Declarative Reasoning over Streams" (CUP H24I17000080001) – PRIN 2017, by MISE under project "S2BDW" (F/050389/01-03/X32) – "Horizon2020" PON I&C2014-20, by Regione Calabria under project "DLV LargeScale" (CUP J28C17000220006) – POR Calabria 2014-20, and by GNCS-INdAM.

References

Adrian, W. T., Alviano, M., Calimeri, F., Cuteri, B., Dodaro, C., Faber, W., Fuscà, D., Leone, N., Manna, M., Perri, S., Ricca, F., Veltri, P., and Zangari, J. 2018. The ASP system DLV: advancements and applications. *KI 32,* 2-3, 177–179.

Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M., and Ricca, F. 2019. Evaluation of disjunctive programs in WASP. In M. Balduccini, Y. Lierler, and S. Woltran (Eds.), *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, Volume 11481 of *Lecture Notes in Computer Science*, pp. 241–255. Springer.

Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., and Zangari, J. 2017. The ASP system DLV2. In M. Balduccini and T. Janhunen (Eds.), *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, Volume 10377 of *Lecture Notes in Computer Science*, pp. 215–221. Springer.

Alviano, M. and Dodaro, C. 2016. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming 16,* 5-6, 533–551.

Alviano, M. and Dodaro, C. 2017. Unsatisfiable core shrinking for anytime answer set optimization. In C. Sierra (Ed.), *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 4781–4785. ijcai.org.

Alviano, M., Dodaro, C., Järvisalo, M., Maratea, M., and Previti, A. 2018. Cautious reasoning in ASP via minimal models and unsatisfiable cores. *Theory and Practice of Logic Programming 18,* 3-4, 319–336.

Alviano, M., Dodaro, C., and Maratea, M. 2018. Shared aggregate sets in answer set programming. *Theory and Practice of Logic Programming 18,* 3-4, 301–318.

Alviano, M., Dodaro, C., and Ricca, F. 2014. Anytime computation of cautious consequences in answer set programming. *Theory and Practice of Logic Programming 14,* 4-5, 755–770.

Alviano, M. and Faber, W. 2011. Dynamic magic sets and super-coherent answer set programs. *AI Commun. 24,* 2, 125–145.

Alviano, M. and Faber, W. 2018. Aggregates in answer set programming. *KI 32,* 2-3, 119–124.

Alviano, M., Faber, W., and Gebser, M. 2015. Rewriting recursive aggregates in answer set programming: back to monotonicity. *Theory and Practice of Logic Programming 15,* 4-5, 559–573.

Alviano, M., Faber, W., and Gebser, M. 2016. From non-convex aggregates to monotone aggregates in ASP. In S. Kambhampati (Ed.), *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pp. 4100–4194. IJCAI/AAAI Press.

Alviano, M., Faber, W., Greco, G., and Leone, N. 2012. Magic sets for disjunctive datalog programs. *Artif. Intell. 187*, 156–192.

Alviano, M., Faber, W., and Woltran, S. 2014. Complexity of super-coherence problems in ASP. *Theory and Practice of Logic Programming 14,* 3, 339–361.

Alviano, M., Greco, G., and Leone, N. 2011. Dynamic magic sets for programs with monotone recursive aggregates. In J. P. Delgrande and W. Faber (Eds.), *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, Volume 6645 of *Lecture Notes in Computer Science*, pp. 148–160. Springer.

Balbin, I., Port, G. S., Ramamohanarao, K., and Meenakshi, K. 1991. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program. 11,* 3&4, 295–344.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. 1986. Magic sets and other strange ways to implement logic programs. In A. Silberschatz (Ed.), *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pp. 1–15. ACM.

Bartholomew, M., Lee, J., and Meng, Y. 2011. First-order semantics of aggregates in answer set programming via modified circumscription. In *Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011.* AAAI.

Beeri, C. and Ramakrishnan, R. 1991. On the power of magic. *J. Log. Program. 10,* 3&4, 255–299.

Behrend, A. 2003. Soft stratification for magic set based query evaluation in deductive databases. In F. Neven, C. Beeri, and T. Milo (Eds.), *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pp. 102–110. ACM.

Codish, M. and Demoen, B. 1995. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Log. Program. 25,* 3, 249–274.

Dodaro, C., Alviano, M., Faber, W., Leone, N., Ricca, F., and Sirianni, M. 2011. The birth of a WASP: preliminary report on a new ASP solver. In F. Fioravanti (Ed.), *Proceedings of the 26th Italian Conference on Computational Logic, Pescara, Italy, August 31 - September 2, 2011*, Volume 810 of *CEUR Workshop Proceedings*, pp. 99–113. CEUR-WS.org.

Eiter, T., Ortiz, M., Simkus, M., Tran, T., and Xiao, G. 2012. Query rewriting for horn-shiq plus rules. In J. Hoffmann and B. Selman (Eds.), *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell. 175,* 1, 278–298.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log. 12,* 4, 25.

Furfaro, F., Greco, S., Ganguly, S., and Zaniolo, C. 2002. Pushing extrema aggregates to optimize logic queries. *Inf. Syst. 27,* 5, 321–343.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2009. On the implementation of weight constraint rules in conflict-driven ASP solvers. In P. M. Hill and D. S. Warren (Eds.), *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, Volume 5649 of *Lecture Notes in Computer Science*, pp. 250–264. Springer.

Gelder, A. V. 1989. Negation as failure using tight derivations for general logic programs. *J. Log. Program. 6,* 1&2, 109–133.

Gelder, A. V., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38,* 3, 620–650.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4-5, 587–601.

Greco, G., Greco, S., Trubitsyna, I., and Zumpano, E. 2005. Optimization of bound disjunctive queries with constraints. *Theory and Practice of Logic Programming 5,* 6, 713–745.

Greco, S. 2003. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Trans. Knowl. Data Eng. 15,* 2, 368–385.

Kemp, D. B., Srivastava, D., and Stuckey, P. J. 1995. Bottom-up evaluation and query optimization of well-founded models. *Theor. Comput. Sci. 146,* 1&2, 145–184.

Kerisit, J. and Pugin, J. 1988. Efficient query answering on stratified databases. In *FGCS*, pp. 719–726.

Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Cuteri, B., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. 2019. Large scale DLV: preliminary results. In A. Casagrande and E. G. Omodeo (Eds.), *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019.*, Volume 2396 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Cuteri, B., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. 2019. Enhancing DLV for large-scale reasoning. In M. Balduccini, Y. Lierler, and S. Woltran (Eds.), *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, Volume 11481 of *Lecture Notes in Computer Science*, pp. 312–325. Springer.

Liu, L., Pontelli, E., Son, T. C., and Truszczynski, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artif. Intell. 174,* 3-4, 295–315.

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H. Schek (Eds.), *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pp. 264–277. Morgan Kaufmann.

Pelov, N., Denecker, M., and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

Przymusinski, T. C. 1989. On the declarative and procedural semantics of logic programs. *J. Autom. Reasoning 5,* 2, 167–205.

Ross, K. A. 1994. Modular stratification and magic sets for datalog programs with negation. *J. ACM 41,* 6, 1216–1266.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell. 138,* 1-2, 181–234.

Stuckey, P. J. and Sudarshan, S. 1994. Compiling query constraints. In V. Vianu (Ed.), *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota, USA*, pp. 56–67. ACM Press.

Whaley, J., Avots, D., Carbin, M., and Lam, M. S. 2005. Using datalog with binary decision diagrams for program analysis. In K. Yi (Ed.), *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, Volume 3780 of *Lecture Notes in Computer Science*, pp. 97–118. Springer.