A Magic Set Implementation for Disjunctive Logic Programming with Function Symbols

Marco Marano, Francesco Ricca, and Giovambattista Ianni

Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy.
mmarano@deis.unical.it,ianni@mat.unical.it, ricca@mat.unical.it

Abstract. The Magic Sets rewriting technique (MS) consists in rewriting a logic program P with respect to a query Q in such a way that, the bottom-up evaluation of the rewritten program simulates the top-down evaluation of Q in the original program. In this way, only a restricted part of the ground program of P is evaluated, thus obtaining valuable efficiency improvements. Many extensions and modifications of the base technique have been proposed in literature, but most of them are confined to the Datalog realm. In this paper we present an implementation of the Magic Set rewriting technique that is applicable to positive disjunctive logic programs with function symbols under the answer set semantics. We show how the base technique has to be modified when both disjunction in the rule heads and function symbols occur contemporarily in the input program. Finally, we describe our implementation of the technique, which is able to produce magic programs compatible with DLV syntax.

1 Introduction

The Magic Sets rewriting technique takes a significant place in the literature about logic programming and deductive database systems, since its early definition in [1]. Given a logic program P and a query Q over its vocabulary, this technique consists in rewriting P with respect to Q, by adding some predicates and some newly created rules: these latter are introduced in order to simulate the top-down computation of the program. By using Magic Sets it is possible to reduce the amount of unnecessary computation, due to portions of the ground version of P which cannot alter the answer to Q, but are however evaluated if a pure bottom-up scheme is used. Many extensions and modifications of the base technique have been proposed in literature, aimed at improving or extending it to more specific cases. Among them, we mention here the extensions to disjunctive logic programs in [2, 6], and the one realized for programs with (possibly unstratified) negation in [3]. In this paper we focus our attention on positive disjunctive logic programs with function symbols, applying the magic set technique to this kind of programs. Some particular issues arise when considering this language, due to the presence of function symbols along with disjunction. The main contributions of this work are: (i) we extend the magic set technique to the case of positive disjunctive programs with function symbols by devising an appropriate transformation algorithm; (ii) we give an implementation of the algorithm, and we show how it works by example.

In the following, we first recall both syntax and stable model semantics of disjunctive logic programs; then, we present by means of an example the classic Magic Sets transformation for Datalog programs, followed by the detailed description of our "magic" algorithm; some notes regarding the implementation and comments about future work conclude the paper.

2 DLP with function symbols

Let $\mathcal{C}$ be a denumerable set of distinguished constant and function symbols. Let $\mathcal{X}$ be a set of variables. Let $\mathcal{P}$ be a set of predicate symbols. We conventionally denote variables with uppercase first letter, while constants will be denoted by lowercase first letter. A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant symbol from $\mathcal{C}$ or a variable from $\mathcal{X}$. A *functional term* is of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are *terms* and f is a function symbol of arity n.
An *atom* is of the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are *terms*, and $p \in \mathcal{P}$ represents the *predicate name* of the atom.
A *program* is a set of *rules* of the form $a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_m$ where $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ are atoms, and $n \geq 0$, $m \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the head of r, denoted by $H(r)$, while $b_1, \ldots, b_m$ is a conjunction denoted as $B(r)$ (the body of r). A rule with empty body will be called *fact*. A predicate appearing only in rule bodies and in facts is referred to as *EDB predicate*, otherwise as *IDB predicate*. In the following, we will assume to deal with *safe* programs, that is, programs in which each variable appearing in a rule r appears in at least one atom in $B(r)$.
Given the program P, the *Herbrand Universe* U_P is the set of ground terms which can be constructed using the symbols of $\mathcal{C}$ appearing in P. A *substitution* for a rule r of P is a mapping θ from the set of variables of r to U_P. We denote $r\theta$ as the *ground* rule obtained by substituting variable occurring in r with elements of U_P according to θ. A ground rule contains only ground atoms; the set of all possible ground atoms that can be constructed combining predicates of P and terms in U_P is usually referred to as *Herbrand Base* (B_P). We denote by $grnd(r)$ the set of ground rules obtained by applying all the possible substitutions to r. Given a plain program P, its ground version $grnd(P)$ is the union of all the sets $grnd(r)$ for $r \in P$.
An *interpretation* for P is a set of ground atoms, that is, an interpretation is a subset $I \subseteq B_P$. We define the following entailment notion with respect to an interpretation I.
For a a ground atom: $I \models a$ iff $a \in I$; For $a_1, \ldots, a_n$ ground atoms:
$I \models a_1, \ldots, a_n$ iff $I \models a_i$, for each $1 \leq i \leq n$; $I \models a_1 \vee \cdots \vee a_n$ iff $I \models a_i$ for at least one i, $1 \leq i \leq n$. For a rule r: $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$;
A *model* for P is an interpretation M for P such that every rule $r \in grnd(P)$ is such that $M \models r$. A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M. The set of all minimal models for P is denoted by $\mathrm{MM}(P)$.
An interpretation I for a program P is an *answer set* for P if $I \in \mathrm{MM}(P)$ (i. e., I is a minimal model for the positive program P). The set of all answer sets for P is denoted by $ans(P)$. We say that $P \models a$ for an atom a, if $M \models a$ for all $M \in ans(P)$.

3 Informal Overview

The *Magic Sets* rewriting technique consists of a simulation of the top-down evaluation of a query Q by modifying an original program P and producing a rewritten program $M(P, Q)$ which comprises additional rules, and updates to the original ones. $M(P, Q)$ is conceived in order to reduce computation to what is actually relevant for answering the query. In fact, $grnd(P)$ contains, in general, many ground rules that have no impact in answering Q as they are related to atoms which Q does not depends on. In general, it is expected that $grnd(M(P, Q))$ has smaller size than $grnd(P)$.
The original magic sets method was first described in [1] for the case of Datalog, i. e. logic programs without function symbols. Following work considered the presence of functional terms, yet not explicitly taking disjunction also into account (see e.g. []).

Concerning the stable model semantics, it is known how to apply this rewriting technique to Datalog Programs with disjunction [2, 6] and also (with some restricting assumption) to unstratified programs [3].
To give an intuition about the general magic set technique for Datalog programs, we can consider the following (traditional) example. Let us consider the query $Q = path(1, 5)?$ on the following program P_1:

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

As a first step, head predicates are "adorned". Basically, we simulate the top-down computation and annotate the way how the variable bindings are propagated from the head atom to body atoms. Each rule of the input program is replaced by an "adorned" one in which the name of each predicate is modified by appending the binding information. Given an IDB predicate, we denote a bound argument with the b letter, while a free one is labeled with f. For instance $path^{bf}$ is a predicate which is in principle a subset of $path$: in particular its first argument is restricted to a set of values (the *magic set* of $path^{bf}$) which is usually much smaller than the range of $path$ on its first argument. The adornment process starts from the query Q. This latter is adorned in a very simple manner: all constants in the query become bound, all variables are marked as free (we obtain in this case the predicate $path^{bb}$). Adornment is propagated to rules' heads in which $path$ appears, and subsequently from the head to the body. If a new adorned predicate is created (as it is present in the head or the body of the rule), this is processed in turn in the same way of the original adorned query, until no more adorned predicates have to be processed. SIPs (Sideways Information Passing Strategies) are used in order to establish the adornment policy: for space reasons we refer the reader to [7], for a detailed explanation about SIPs. In our example, the arguments of the given query are both constants,and thus bound; we will build the adorned program according to $path^{bb}$:

Note that EDB predicates are excluded from adornment. The next step of the transformation consists in generating magic rules starting from the adorned program. These rules define *magic predicates*. A *magic predicate* defines the allowed range of values for bound arguments of a predicate. We start from the head of the rule.
Given an adorned head atom $a(\mathbf{t})$, we obtain the set of terms $\mathbf{t}'$, derived from $\mathbf{t}$ by removing all the terms corresponding to free arguments, and generate the magic atom $magic_a(\mathbf{t}')$. Then, for each atom b in the body, we create its magic version $magic_b(\dots)$. Subsequently, we generate a magic rule having $magic_b$ in the head and $magic_a$ in the body, followed by all the atoms of the adorned rule which can propagate the binding.
The third step consists of the modification of the adorned rules. In this step we add to the bodies of the rules the magic atoms which have been generated in the previous step. For each rule with head h, an according magic atom $magic_h$ is inserted in the body of the rule.

$magic\_path^{bb}(1, 5).\ magic\_path^{bb}(Z, Y) \text{ :- } magic\_path^{bb}(X, Y), edge(X, Z).$
$path(X, Y) \text{ :- } magic\_path^{bb}(X, Y), edge(X, Y).$
$path(X, Y) \text{ :- } magic\_path^{bb}(X, Y),\ edge(X, Z),\ path(Z, Y).$

Finally, in the last step the query is processed by adding a magic fact $magic\_q\_ad$ if q is the query and ad its adornment; In our example we add $magic\_path^{bb}(1, 5)$.
The resulting program is then evaluated w.r.t. the query.

4 Magic Sets for DLP with Function Symbols

In this section we present a new version of the standard Magic Set technique presented above, which works on *disjunctive* programs with *function symbols*. The algorithm is sketched in Figure 1. The main procedure is called **magify**.

```
Program magify(Program P, Query Q)
{
    Program M(P,Q)=∅;
        if(Q.isconjunctive())
        {
            Rule R',Query Q';
            (Q',R') = normalizeQuery(Q);
            P.addRule(R');
            Q=Q';
        }
    Program AP = createAdornedProgram(P,Q);
    Program MR = createMagicRules(AP);
    Program MP = addMagicAtoms(P);
    Fact MF = createMagicFact(Q);
    M(P,Q)=removeAdornments(MP∪MR∪MF);
    return M(P,Q);
}
```

Fig. 1. Function createAdornedProgram

```
Program createAdornedProgram( Program P,
Query Q )
{
    Stack S = ∅; Program AP = ∅;
    S.push(createAdornedVersionOf(Q));
    while(S. size > 0)
        {
            Atom x= S.pop();
            for(Rule r in P)
            Rules adornedRules = adornRule(r,x);
            AP.add(adornedRules);setDone(X);
            for(Rule ar in adornedRules)
                for(Atom a in ar)
                    if(!done(a)) S.push(a);
        }
    return AP;
}
```

Fig. 2. Function createAdorned-Program

The function **magify** takes a program P and a query Q as input, and applies the Magic Sets Transformation, generating $M(P,Q)$ (the *magified program*). **magify** is made of other subprocedures, detailed in the following. Let us assume it is given the query: $Q_2 = a(f(1))?$ and the program P_2:

$$r1 : a(X) \vee b(X) \text{ :- } c(X), e(X).\ \ r2 : c(f(X)) \text{ :- } c(X).\ \ r3 : e(1).\ \ r4 : c(1).$$

When a query is conjunctive, it is transformed into a rule, having in the head a new atom which contains all the variables from the atoms in the original query. The original query is replaced by a new one which consists of the head of the newly created rule. This procedure is performed by the function **normalizeQuery(Query Q)**.
The next step consists of creating the adorned program **AP**, by means of the function **createAdornedProgram(Program P, Query Q)**, reported in Figure 2. A stack S is used in order to keep the atoms scheduled for adornment. The query is adorned using the function **createAdornedVersionOf** and pushed in S at first. The main cycle pops out from S a given atom a and accordingly adorns each rule having in the head an atom whose name matches with it. When a certain adornment is generated for the first time for a predicate, this is pushed into S, in order to be processed. The algorithm iterates until S is empty.
The adornment of each rule is actually performed by the inner function **adornRule**(r, x) which returns a set of adorned rules according to the labels of x, to be added to the adorned program. If x is not in the head of r, **adornRule** returns an empty set. More in detail, the output of **adornRule** contains a set R' of adorned rules for each atom $x' \in H(r)$ which unifies with x. Each $r' \in R'$ is built according to the following strategy: per each $x' \in H(r)$ which unifies with x, x' is labelled according to x, then such labelling is propagated to $B(r)$, according to a SIP [7]. Successively, adornments are propagates from

$B(r)$ to the remaining head atoms. Moreover, from the obtained adorned disjunctive rule r', corresponding to x', we obtain $|H(r)|-1$ auxiliary rules obtained by leaving in the head only one atom $x \in H(r) \setminus \{x'\}$ and having $B(r) \cup (H(r) \setminus x)$ as body. The obtained set of auxiliary rules in AP will not take part in the final program $M(P,Q)$, but will be further processed in order to obtain the set of magic rules MR. In turn, magic rules are created, according with the traditional strategy, by calling the **CreateMagicRules** function. In our example, we get first from rule $r1$ and $r2$, the adorned versions $r1' : a^b(X) \vee b^b(X) \text{:-} c^b(X), e(X)$ and $r2' : c^b(f(X)) \text{:-} c^b(X)$ then **createMagicRules(Program P)** obtains from $r1'$ and $r2'$ the corresponding magic rules; and from $r1'$ we get the two rules: $a^b(X) :- c^b(X), e(X), b^b(X).$ $b^b(X) :- c^b(X), e(X), a^b(X).$, while $r2'$ is left unchanged. Now the function **createMagicRules** simply applies the normal Magic-Set strategy to these intermediate rules, as seen in previous section. In our example we obtain:

$magic\_c^b(X) :- magic\_a^b(X), e(X), b^b(X).$ $magic\_c^b(X) :- magic\_b^b(X), e(X), a^b(X).$
$magic\_c^b(X) :- magic\_c^b(f(X)).$

The third rule has been obtained by applying the algorithm for the non disjunctive case. Now, the function **addMagicAtoms**(P) is called, which returns a version of **P** including magic predicates within the body of each rule of **P**. In this simple step, for each atom in the head of the rule the corresponding magic atoms are added in the body. Successively, a magic atom from the query is generated by the function **createMagicFact(Query Q))** to be added to the final output. In our example we get: $magic\_a^b(f(1))$. Finally, the function call **removeAdornments**$(MP \cup MR \cup MF)$ removes all adornments from the non-magic predicates. This is necessary as stated in [2]. The final output for our example is the following:

$magic\_c^b(X) :- magic\_a^b(X), e(X), b^b(X).$ $magic\_c^b(X) :- magic\_b^b(X), e(X), a^b(X).$
$magic\_c^b(X) :- magic\_c^b(f(X)).$ $magic\_a^b(f(1)).$
$a(X) \vee b(X) :- c(X), e(X), magic\_a^b(X), magic\_b^b(X).$ $c(f(X)) :- c(X), magic\_c^b(f(X)).$

It must be noted here that two aspects of the class of programs we are treating, disjunction and the presence of function symbols, need a particular treatment. In particular:

Disjunction requires modifications on the adornment strategy. Let r be a rule of the form:

$$r1 : h_1(t_1) \vee \ldots \vee h_n(t_n) :- b_1(p_1), \ldots, b_m(p_m).$$

If we adorn the rule w.r.t. the atom $h_i(t_i)$, also other head atoms have to be taken into consideration, because they can contain variables which are actually important for the evaluation. The function acts as follows:(i) the atom $h_i(t_i)$ is adorned w.r.t. the query; (ii) the body is adorned w.r.t. the adornments of $h_i(t_i)$ by using a suitable SIP; (iii) other head atoms $h_1(t_1) \vee \ldots \vee h_{i-1}(t_{i-1}) \vee h_{i+1}(t_{i+1}) \vee \ldots \vee h_n(t_n)$ are adorned w.r.t. patterns found in the body.
In fact, it has been shown in [6] that if we want to keep the algorithm sound, other head predicates cannot propagate bindings, but can only receive them. In this case bindings are propagated from the body to the remaining head atoms.

Function symbols have impact on the choice of the labelling for arguments: Given an atom a(. . . , t, . . .) for t a functional term t , the corresponding argument of a is labelled as bound iff all the subterms of t are set as bound at the moment of adornment of a.

Remark. Our transformation applies to programs with function symbols, thus, in general, an evaluation of the $M(P,Q)$ is not guaranteed to terminate. However, there are language restrictions that ensures termination, for instance see [9].

5 Implementation Notes and Future Work

The prototype has been implemented in the Java programming language as a preprocessor able to generate a magified program $M(P,Q)$ compatible with the DLV input format [4] from a given program P and a, possibly conjunctive, query Q. The system uses a new Library, called DLVParser, which contains a full framework of classes useful for both the parsing and the manipulation of a Disjunctive Logic Programs in standard syntax.
Design patterns have been used, in order to keep the system flexible and easily extensible. In particular, the *S*trategy pattern has been used for allowing the implementation of multiple SIPs, so that the user of the API of our system is allowed to define his own strategy. To define a new SIP, only a few methods have to be implemented. We have implemented a default SIP, which mimics the propagation of bindings in the Prolog SLD resolution. Inclusion of other constructs such as negation and constraints are forthcoming.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J. D. Magic Sets and Other Strange Ways to Implement Logic Programs. In*PODS'86*,1986.
2. Cumbo C., Faber W., Greco G., Leone N. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *ICLP'04* pages 371-385,2004.
3. Faber W., Greco G., Leone N. Magic Sets and their Application to Data Integration. In *ICDT 2005* pages 306-320, 2005.
4. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.
5. Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
6. Greco, S. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE TKDE 15 (2), 368-285*, 2003.
7. Beeri, C., Ramakrishnan, R. On the power of magic. *JLP 10 (1,2,3&4), 255-299*,1991.
8. Ramakrishnan, R. Magic Templates: A Spellbinding Approach To Logic Programs. *JLP 11 (3&4), 189-216* , 1991.
9. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni and Nicola Leone. Bottom-up Evaluation of Finitely Recursive Queries. *CILC09.*, 2009.