ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

SLDMagic --- An Improved Magic Set Technique

Stefan Brass

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

SLDMagic — An Improved Magic Set Technique

Stefan Brass

Institut für Informatik, Universität Hannover
Lange Laube 22, D-30159 Hannover, Fed. Rep. Germany
E-mail: sb@informatik.uni-hannover.de

Abstract

We present a rewriting method for Datalog-programs which simulates SLD-resolution more closely than the ordinary “magic set” method does. This is especially advantageous in the case of tail-recursive programs, but already in non-recursive programs we can often save a number of joins. In contrast to the method of ROSS [7], we do not only solve the problem of tail-recursion, but try to simulate SLD-resolution as fully as possible. An especially nice feature of our approach is that we get many other known optimizations “for free” in this way. Based on an idea of BRY [3], our method can be described in an elegant way by means of a meta-interpreter. This also allows to compare the efficiency of SLD-resolution and magic sets within a common framework. We then develop a combined method, which allows to choose the evaluation strategy for every body literal.

Keywords: Deductive Databases, Query Evaluation, Magic Sets, SLD-Resolution.

1 Introduction

It has been noted that bottom-up evaluation of a program rewritten by the “magic set” method can be much less efficient than SLD-resolution if there are tail-recursive rules [7]. This already applies to one of the most often used example programs, namely the transitive closure. For instance, the “ancestor” relation can be described as follows:

$$\begin{aligned} anc(X, Y) &\leftarrow parent(X, Y). \\ anc(X, Z) &\leftarrow parent(X, Y) \wedge anc(Y, Z). \end{aligned}$$

To make the example as simple as possible, let us assume that we know only the father within some ancient dynasty of kings, so the *parent*-relation is one straight line of length n :

$$parent := \{(i-1, i) \mid 1 \leq i \leq n\}.$$

Then the SLD-tree for the query $anc(0, X)$ has $4n + 3$ nodes. With an index over the first argument of *parent*, any reasonable implementation of SLD-resolution should have the running time $O(n * \log(n))$. However, in this particular example, the magic set transformation does not improve the program at all: We can derive $m_{anc}^{bf}(i)$ corresponding to the query $anc(i, X)$ for all $0 \leq i \leq n$. This is intuitively understandable, since when we want to determine the ancestors of person number 0, we have to determine recursively the ancestors of person number 1, and so on. But when we have $m_{anc}^{bf}(i)$, we also get $anc(i, j)$ for all $0 \leq i < j \leq n$ (i.e. the “answer facts” to the above queries). Thus the behaviour is at least $O(n^2)$ (not counting the computations of joins, duplicate elimination, and so on). So the original “magic set” method is not always as efficient as SLD-resolution, not even asymptotically.

This does not conflict with a well-known result of ULLMAN [10], because he has used QRGT-resolution for comparison, not SLD-resolution. The problem is that QRGT-resolution, as well as OLDT-resolution, “materialize” the IDB-facts (lemmas) proven in an SLD-tree. But in case of tail-recursive programs (and only there [2]), a single node in an SLD-tree can complete the proof an unbounded number of lemmas.

This problem has been solved by ROSS, who introduced a variant of magic sets with tail-recursion optimization [7]. The idea was to add to a tail-recursive literal A the literal B , which is really proven when the proof of A is finished. RAMAKRISHNAN and SUDARSHAN have compared in [6, 9] the efficiency of ROSS’s method with Prolog-evaluation,

and at least on an abstract level the result was that this method is as efficient as Prolog. This is also suggested by [2], where it is shown that for programs without tail-recursion magic sets are as efficient as SLD-resolution (using a very high level cost measure).

However, we will show here that it is possible to go further in simulating SLD-resolution. The goal of ROSS was to solve the problem of tail-recursion, our goal is to implement SLD-resolution on top of a deductive database. The tail-recursion optimization is only a consequence of our approach, and we get also a number of other known optimizations “for free”. Furthermore, our method can be described simply and elegantly by adapting BRY’s idea of using a meta-interpreter to derive the “magic set” method [3]. The meta-interpreter given by BRY corresponds to the QRGT-algorithm, therefore he got the standard magic-set transformation. In fact, [3] also contains a meta-interpreter which corresponds more directly to SLD-resolution, but it still implicitly uses the critical tabulation of derived lemmas.

So we present a bottom-up meta-interpreter which corresponds exactly to SLD-resolution. This is in itself simple, but the partial evaluation of this meta-interpreter needs some thoughts.

We then explain the main differences between this implementation of SLD-resolution and magic sets. In short, SLD-resolution gives the called literals more context information, namely, how the proof will continue after the literal is proven. In contrast, magic sets usually need a join to get this context back. The disadvantage of SLD-resolution is that it might happen that the same literal is proven more than once, if it is called in different contexts. This is avoided by magic sets, but they might introduce new recursions by merging different computations. So both methods have advantages of their own, and the real optimum will be some combination.

Thus, we extend the meta-interpreter in such a way that it can do both, either simulate SLD-resolution or magic sets (selectable for every literal by annotations in the program). The idea for combining the two methods is that we get magic sets from SLD-resolution if we evaluate IDB-literals in a subproof, and explicitly get back the answer. Something similar happens with negative literals in SLDNF-resolution: they are also evaluated by a recursive call to SLDNF-resolution. Alternatively, we can say that the extended meta-interpreter describes SLD-resolution with tabulation for some literals.

The currently fastest deductive database system, namely the XSB-system [8], does not use bottom-up evaluation with magic sets, but a variant of SLD-resolution with tabulation. This means that it has the same problems with tail-recursive programs, and yet it is faster than bottom-up evaluation with “magic sets”. So probably any contribution to simulate SLD-resolution more directly could help to make “real” deductive database systems catch up with the XSB-system, and fully utilize the theoretical advantages of bottom-up evaluation.

2 A Bottom-Up Simulation of SLD-Resolution

Our goal in this section is to present a bottom-up meta-interpreter which describes SLD-resolution. We consider only positive Datalog-programs with range-restricted rules.

As usual for meta-interpreters, the rules of the given program are treated as input data, so they are stored in the database. We represent rules as facts about a binary predicate *rule*, where the first argument is the head literal and the second argument is the list of body literals. For instance,

$$anc(X, Z) \leftarrow parent(X, Y) \wedge anc(Y, Z)$$

is stored as

$$rule(anc(X, Z), [parent(X, Y), anc(Y, Z)]).$$

Of course, this fact is neither Datalog (the old predicates are now function symbols) nor range-restricted (it contains variables, but does not have a rule body). Meta-interpreters always need a more general rule language, but this is only temporary, because we will later try to remove this by partial evaluation.

Besides the rules of the program, of course also the facts in the database are input to our meta-interpreter. We assume that the tuples in the database predicates are represented in a common relation *db*, namely in the form

$$db(p(c_1, \dots, c_n)).$$

In this way, the meta-interpreter does not depend on the set of EDB-predicates.

```

/* Initialization (Root Node): */
node(Query, [Query]) ←
    query(Query).

/* SLD-Resolution: */
node(Query, Child) ←
    node(Query, [Lit|Rest]) ∧
    rule(Lit, Body) ∧
    append(Body, Rest, Child).

/* Evaluation of DB-Literal: */
node(Query, Rest) ←
    node(Query, [Lit|Rest]) ∧
    db(Lit).

/* Turn Proven Query into Answer: */
answer(Query) ←
    node(Query, []).

```

Figure 1: Bottom-Up Meta-Interpreter for SLD-Resolution

Finally, the query Q (a single literal) is represented by the fact

```
query(Q).
```

Our goal now is to compute the nodes of the SLD-tree, more precisely, the goals attached to these nodes. But we need of course also the computed substitution for the answer variables. We solve this problem by attaching to each node the given query literal. It shares variables with the goal at the node, and all substitutions are also applied to the query literal. So when we have derived the empty goal, the corresponding instance of the query has been proven. For example, the following *node*-fact might be derivable:

```
node(anc(0, X), [parent(3, X)]).
```

This means that a parent X of person number 3 is also an ancestor of person number 0. There are of course also other possibilities to remember the substitution for the answer variables, but this encoding of the nodes is especially useful when we later have to manage more than one SLD-tree.

The meta-interpreter consists of four rules. First, we need to create the root node of the SLD-tree. Of course, the goal in the root node is the given query literal:

```
node(Query, [Query]) ←
    query(Query).
```

The second rule describes the SLD-resolution step. It is the main rule of the meta-interpreter, which does the “real work”. As usual in SLD-resolution, we replace the first goal literal by the body of a rule with a matching head literal:

```
node(Query, Child) ←
    node(Query, [Lit|Rest]) ∧
    rule(Lit, Body) ∧
    append(Body, Rest, Child).
```

Note that bottom-up evaluation with non-ground facts does the necessary unification, and renames the variables of the used “facts” before that in order to avoid name clashes. The variables in the derived facts are again normalized, so that facts which differ only in the variable names are detected as duplicates. This is important for the termination.

The third rule does the evaluation of EDB-literals by finding matching facts in the database:

$$\begin{aligned} \text{node}(\text{Query}, \text{Rest}) \leftarrow \\ \text{node}(\text{Query}, [\text{Lit}|\text{Rest}]) \wedge \\ \text{db}(\text{Lit}). \end{aligned}$$

The last rule derives the ground instances of the query which are proven once the empty goal is reached:

$$\begin{aligned} \text{answer}(\text{Query}) \leftarrow \\ \text{node}(\text{Query}, []). \end{aligned}$$

The complete meta-interpreter is shown in Figure 1. It computes exactly the nodes in the SLD-tree.

Example 1 This meta-interpreter can be executed by deductive database systems like CORAL¹ which allow structured terms and non-ground facts. For instance, consider again the transitive closure program:

$$\begin{aligned} \text{anc}(X, Y) &\leftarrow \text{parent}(X, Y). \\ \text{anc}(X, Z) &\leftarrow \text{parent}(X, Y) \wedge \text{anc}(Y, Z). \end{aligned}$$

For simplicity, let the *parent*-relation contain the single tuple (0, 1). The SLD-tree for the query $\text{anc}(0, X)$ then looks as follows:

CORAL answers the query “?node(Q, N)” with

```

Q=anc(0, _X0), N=[anc(0, _X0)].
Q=anc(0, _X0), N=[parent(0, _X0)].
Q=anc(0, _X0), N=[parent(0, _X1), anc(_X1, _X0)].
Q=anc(0, 1), N=[].
Q=anc(0, _X0), N=[anc(1, _X0)].
Q=anc(0, _X0), N=[parent(1, _X0)].
Q=anc(0, _X0), N=[parent(1, _X1), anc(_X1, _X0)].

(Number of Answers = 7)

```

The query “?answer(A)” returns “A=anc(0, 1)”. So this approach directly simulates SLD-resolution. $\square$

Since we do not compute the nodes themselves, but only the goals attached to them, the termination behaviour is better than that of SLD-resolution. For instance, the rule $p(X) \leftarrow p(X)$ poses no problem at all, since it does not yield new goals. In general, we can guarantee the termination for all tail-recursive Datalog-programs using only finite database predicates. We do not suggest to simulate SLD-resolution for predicates which are recursive, but not tail-recursive. For programs containing different kinds of recursion, we will later present a combined method which allows to use the “magic set” behaviour (tabulation) for calling some literals.

¹Version 1.2, available from <http://www.cs.wisc.edu/coral/>.

Example 2 But let us have a look at a non tail-recursive program, for instance the “same generation” example:

$$\begin{aligned} sg(X, X) &\leftarrow parent(X, _). \\ sg(X, X) &\leftarrow parent(_, X). \\ sg(X, Y) &\leftarrow parent(X, Xp) \wedge sg(Xp, Yp) \wedge parent(Y, Yp). \end{aligned}$$

Let the query be $sg(a, Y_0)$. If the $parent$ -relation should be cyclic, e.g. consist of the single tuple $parent(a, a)$, longer and longer goals of the form

$$sg(a, Y_n) \wedge parent(Y_{n-1}, Y_n) \wedge \dots \wedge parent(Y_0, Y_1)$$

are constructed.

This problem also occurs with the “counting” method [1]. One can say that this method also tries to simulate SLD-resolution in its limited application area. It only uses the special encoding $sg_down(a, n)$ for the above node of the SLD-tree. In fact, many known optimizations can be understood by the simple idea to simulate SLD-resolution (see below). $\square$

As mentioned above, we can guarantee termination if all recursions are tail-recursions (and the database is finite):

Definition 3 (Tail-Recursive Program)

A program is at most tail-recursive iff for every rule

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

the predicates of B_i , $1 \leq i \leq m - 1$, do not depend on the predicate of A , i.e. no body literal except possibly the last is recursive.

Note that this class of programs is much larger than the class for which the “right recursion optimization” of [11] is applicable. Since the only requirement is that the recursive literal is the last literal, we could of course make the “same generation”-program tail-recursive by moving the recursive body literal to the end of the rule. But for a query of the form $sg(a, Y)$, SLD-resolution becomes very inefficient with this sequence of body literals ($parent$ is called with both arguments free), so we would not want the behaviour of SLD-resolution in this case.

Lemma 4 *Let P be an at most tail-recursive program and Q be a query. Then there is an $n \in \mathbb{N}$, such that for every DB, no goal in the SLD-tree for $P \cup \{\leftarrow Q\}$ wrt DB has more than n literals.*

Proof: In at most tail-recursive programs, a rule can only applied recursively on its last body literal, so iterative recursive rule applications cannot make the stack of “to be proven” literals longer. Therefore, the length (number of literals) of nodes in the SLD-tree is bounded. An upper bound would be the sum of the lengths of all rule bodies. $\square$

So if there are only finitely many possible constants occurring in the goals (e.g. because the program contains only finite database predicates), then the number of possible goals is bounded and we can guarantee termination.

3 Efficiency Comparison

An important difference between SLD-resolution and magic sets is that SLD-resolution passes more context into the sub-computations. For instance, when a predicate p is called in two different rules, then the two computations for p are kept disjoint in SLD-resolution, while they are merged in the magic set method. If the arguments for which p is called are more or less disjoint, then SLD-resolution is superior, because we do not need to filter the appropriate results later by means of a join. If, however, p is called for the same arguments at different places, then SLD-resolution repeats the computation, and magic sets are superior (at least if the computation of p is difficult).

Example 5 Consider for instance the following program computing “grandparent” relation, where we prefer to use the shorthand p for $parent$:

$$\begin{aligned} grandpar(X, Z) &\leftarrow p(X, Y) \wedge p(Y, Z). \\ p(X, Y) &\leftarrow parent(X, Y). \end{aligned}$$

Suppose that the query is $grandpar(a, Z)$. The “magic set” transformation leads to the following program:

$$\begin{aligned} grandpar(X, Z) &\leftarrow m_grandpar^{bf}(X) \wedge p(X, Y) \wedge p(Y, Z). \\ m_p^{bf}(X) &\leftarrow m_grandpar^{bf}(X). \\ m_p^{bf}(Y) &\leftarrow m_grandpar^{bf}(X) \wedge p(X, Y). \\ p(X, Y) &\leftarrow m_p^{bf}(X) \wedge parent(X, Y). \\ m_grandpar^{bf}(a). \end{aligned}$$

(Since there is only one binding pattern for $grandpar$ and p , we suppressed the adornments.) Now if $parent$ does not contain (a, a) , the two rules about m_p^{bf} yield in fact disjoint bindings for p . But these are collected together in one set, and when we get the resulting p -tupels back, we need two joins to match them with the calling body literal. In total, four joins are needed to evaluate this program (and probably even more, see below).

In contrast, SLD-resolution computes the following sequence of goals (encoded in our *node*-facts):

- $node(grandpar(a, Z), [grandpar(a, Z)])$.
- $node(grandpar(a, Z), [p(a, Y) \wedge p(Y, Z)])$.
- $node(grandpar(a, Z), [parent(a, Y) \wedge p(Y, Z)])$.
- For every b with $parent(a, b)$, we get:
 $node(grandpar(a, Z), [p(b, Z)])$.
- $node(grandpar(a, Z), [parent(b, Z)])$.
- For every c with $parent(b, c)$, we get:
 $node(grandpar(a, c), [])$.

So the results of the calls to p appear directly at the place where they are needed, no additional filtering is necessary. Only when $parent$ is evaluated, a join or a selection is needed, which gives two joins in total. We will see later how the intermediate steps can be optimized away.

If $parent$ contained (a, a) , SLD-resolution would repeat the computation of $p(a, Y)$, once with continuation $[p(Y, Z)]$, and once with continuation $[]$. Of course, since the computation of p is easy, this is no problem. But if it would take a long time to compute p , magic sets might be better.

The merging of computations done by the “magic set” method can sometimes introduce recursions in the magically rewritten program, and indeed this happens in the above example (m_p^{bf} depends on p , and p depends on m_p^{bf}). Since the overhead for evaluating recursions is relatively large, p must be rather complicated in order to justify the effort for avoiding its duplicate computation. $\square$

Example 6 Another source of multiple contexts for one goal literal are bindings for variables not occurring in this literal. For instance, consider the rule

$$uncle(X, Y) \leftarrow p(X, Z) \wedge brother(Z, Y),$$

where p is again a computed parent-relation (e.g. the union of *mother* and *father*). Suppose that $uncle$ is called with both arguments bound. The magic set method translates this to:

$$\begin{aligned} uncle(X, Y) &\leftarrow m_uncle^{bb}(X, Y) \wedge p(X, Z) \wedge brother(Z, Y). \\ m_p^{bf}(X) &\leftarrow m_uncle^{bb}(X, Y). \end{aligned}$$

So when p is called, the given values of Y are projected away, but after p is computed, we must join the result again with m_uncle^{bb} in order to get Y back. SLD-resolution retains Y during the computation of p , and thus avoids this join. Of course, there is a tradeoff: If there are many different Y -values for a single X -value, magic sets can be faster.

It is interesting to note that we might get both advantages if we use NF^2 -relations, and group the (X, Y) -tuples by X when we execute p . Such a “nesting” operation is not much more expensive than the projection (with duplicate elimination) done by the magic set method. There seem to be many cases in which NF^2 -relations would allow a more compact representation and fewer copying operations, resulting in a more efficient query evaluation. $\square$

Let us also note that our approach yields the following optimizations as a byproduct:

- Duplicate variables in body literals pose no problem, since we always have the complete information about the structure of the goals. In contrast, the magic set method needs an extra “rectification” phase.
- Values for anonymous variables are never represented explicitly.
- Constants from the rules are pushed into the called rules, corresponding to a kind of “compile time” magic set transformation.
- If the given program is non-recursive, the resulting program is also non-recursive.

Of course, each of these optimizations is already known in the field of deductive databases, but they have been introduced separately. Our common framework can explain the deeper reasons behind them.

4 Adding “Magic Set” Behaviour

So both, magic sets and our bottom-up simulation of SLD-resolution have advantages of their own, and it depends on the application, which one is superior. Therefore, we introduce in this section a parameterized approach which allows to choose between both possibilities on a single literal basis.

In order to simulate the “magic set” behaviour, we allow that SLD-resolution can call itself recursively for evaluating certain literals. Standard SLD-resolution does a similar thing for negative literals. We believe that this idea of starting a subproof and getting back the explicitly represented result is the key to understand the difference between magic sets and the real SLD-resolution.

For our meta-interpreter, we assume that the user has included literals to be evaluated via magic sets in a special predicate $call(\dots)$ (not otherwise used in the program). Another possibility would be to have “table” declarations for the predicates to be evaluated in subproofs. In fact, our meta-interpreter can also be understood as describing a form of SLD-resolution with tabulation. Of course, the best option would be an intelligent optimizer which automatically determines (guesses) where it is better to use SLD-resolution and where “magic sets” are superior.

Our meta-interpreter is generalized by adding two additional rules to handle $call(\dots)$ -literals. First, such a literal creates a new query:

$$\begin{aligned} query(Lit) \leftarrow \\ node(_, [call(Lit)|_]). \end{aligned}$$

So we now construct not a single SLD-tree, but a set of SLD-trees, one for each recursive call. Since every derived $node$ -fact contains the root of its tree as the first argument, we can always identify the tree(s) to which a node belongs.

If the subproof started by the above rule is successful, one or more instances of the query are derived in the $answer$ -relation. Now the following rule gets these results back into the main proof by resolving the $call(\dots)$ -literal with the derived $answer$ -literals:

$$\begin{aligned} node(Query, Rest) \leftarrow \\ node(Query, [call(Lit)|Rest]), \\ answer(Lit). \end{aligned}$$


```

/* Initialization (Root Node): */
node(Query, [Query]) ←
    query(Query).

/* SLD-Resolution: */
node(Query, Child) ←
    node(Query, [Lit|Rest]) ∧
    rule(Lit, Body) ∧
    append(Body, Rest, Child).

/* Set up Recursive Call (Derive Magic Literal): */
query(Lit) ←
    node(−, [call(Lit)|−]).

/* Get Result of Recursive Call: */
node(Query, Rest) ←
    node(Query, [call(Lit)|Rest]),
    answer(Lit).

/* Evaluation of DB Literals: */
node(Query, Rest) ←
    node(Query, [Lit|Rest]),
    db(Lit).

/* Turn Proven Query into Answer: */
answer(Query) ←
    node(Query, []).

```

Figure 2: Meta-Interpreter for SLD-Resolution/Magic Sets

The complete meta-interpreter is shown in Figure 2.

If no IDB-literal is classified as *call*(...), we obviously compute exactly the nodes of the single SLD-tree as in the above meta-interpreter. If all literals are evaluated in subproofs, we get something very similar to BRY's meta-interpreter [3] (only the representation of conjunctions is a little different). In fact, we will argue below that partial evaluation yields exactly the magic set method with supplementary predicates, where *query*-facts correspond to magic facts, *answer*-facts correspond to derived IDB-facts, and *node*-facts correspond to facts of the supplementary predicates.

Note that the first new rule can also be understood as entering a goal literal into a table *query*, and the second rule then allows resolution with proven lemmas from the table *answer*. In this way, our meta-interpreter also describes some form of SLD-resolution with tabulation.

5 Computation of Node Types

The above meta-interpreter can be executed in deductive database systems which allow bottom-up evaluation with non-ground facts. Of course, the use of lists and non-ground facts significantly decreases the performance. If we are given a Datalog-program consisting of range-restricted rules without function symbols, we of course want that the result of the transformation also has these nice properties. The approach suggested by [3] is partial evaluation with respect to the given rules and the query. However, simply unfolding the call to the *rule*-predicate does not help much. The problem is of course the relation *node* — we need more information about the derivable *node*-facts in order to continue the partial evaluation. To this end, we have introduced in [2] the notion of a node type. It results from a

node-fact by abstracting from the occurring constants and the names of the variables.

Definition 7 (Node Type)

Let X_i and C_i ($i \in \mathbb{N}_0$) be disjoint sequences of variables. We will call the C_i parameters. Let $a_1, \dots, a_k$ be the constants appearing in the given program P or query Q . A node type N is a node-literal of the form $node(Q, [A_1, \dots, A_n])$, where

- Q is a literal and the A_i are possibly call($\dots$)-enclosed literals.
- Q and the A_i contain only variables from

$$\{X_0, X_1, \dots\} \cup \{C_0, C_1, \dots\}$$

and constants from $\{a_1, \dots, a_k\}$.

- If X_i appears somewhere in N , then $X_0, \dots, X_{i-1}$ appear already to the left (and the same for C_i).

A node-literal L has type N if there is a mapping θ from the C_i to constants and from the X_i to variables such that $N\theta = L$.

Beside node types, there are also query types and answer types, which are defined completely analogously.

Example 8 Consider again the “transitive closure” program:

$$\begin{aligned} anc(X, Y) &\leftarrow parent(X, Y). \\ anc(X, Z) &\leftarrow parent(X, Y) \wedge anc(Y, Z). \end{aligned}$$

In case of the query $anc(0, X)$, the following node types can appear (for any possible database):

$$\begin{aligned} &node(anc(0, X_0), [anc(0, X_0)]) \\ &node(anc(0, X_0), [parent(0, X_0)]) \\ &node(anc(0, X_0), [parent(0, X_1), anc(X_1, X_0)]) \\ &node(anc(0, C_0), [true]) \\ &node(anc(0, X_0), [anc(C_0, X_0)]) \\ &node(anc(0, X_0), [parent(C_0, X_0)]) \\ &node(anc(0, X_0), [parent(C_0, X_1), anc(X_1, X_0)]) \end{aligned}$$

This set of node types corresponds to the nodes computed in Example 1. However, these seven node types suffice for any possible *parent*-relation, since C_0 can be any ancestor of person number 0.

Of course, it is possible to replace the constant 0 from the query by a parameter. This is usually done in the magic set method, since in this way the program does not have to be recompiled when the same query is posed later with a different constant. However, if we use the constants from the query and the rules, the resulting program can be slightly faster. $\square$

The computation of the node-types can be done by executing the meta-interpreter bottom-up, but with the following differences:

- Since the actual database is unknown at compile time, we use a “generic EDB-fact” $p(C_1, \dots, C_n)$, when we have to resolve with an EDB-literal. Here the parameters C_i act like variables, but if we have to unify a “real variable” X_j and a parameter C_i , we replace X_j by C_i and not vice versa. Thus, we know at what places constants appear.
- The variables in derived literals are normalized to $X_0, X_1, \dots$ and the parameters to $C_0, C_1, \dots$. In this way, we can detect when we have derived the same node type again. It is optionally possible to replace known constants (from the query or the rules) by parameters C_i .

```

Input:      Program  $P$ , Query  $Q$ .
Output:    Node Types  $\mathcal{N}$ , Query Types  $\mathcal{Q}$ ,
              Answer Types  $\mathcal{A}$ .

/* Initialization: */
 $\mathcal{N} := \emptyset$ ; /* No node types so far */
 $\mathcal{A} := \emptyset$ ; /* No answer types so far */
 $\mathcal{Q} := \{normalize\_lit(Q)\}$ ;
 $changed := true$ ;

/* Bottom-Up Computation Loop: */
while  $changed$  do
   $changed := false$ ;

  /* Meta-Interpreter Rule #1: */
  ...

  /* Meta-Interpreter Rule #2: */
  foreach node type  $N_1$  in  $\mathcal{N}$  do
    let  $N_1 = node(Q, [A_1, \dots, A_n])$ ;
    foreach rule  $A \leftarrow B$  in  $P$  do
       $i := max\_var\_index(N_1)$ ;
       $A' \leftarrow B' := fresh\_var(i, A \leftarrow B)$ ;
       $\theta := unify(A_1, A')$ ;
      if  $\theta \neq nil$  then
         $G := append(B', [A_2, \dots, A_n])$ ;
         $N_2 := node(Q\theta, G\theta)$ ;
         $N'_2 := normalize\_node(N_2)$ ;
        if  $N'_2 \notin \mathcal{N}$  then
           $\mathcal{N} := \mathcal{N} \cup \{N'_2\}$ ;
           $changed := true$ ;

  /* Further Meta-Interpreter Rules: */
  ...

```

Figure 3: Algorithm to Compute Node Types

This results in the algorithm of Figure 3. For simplicity, we have only shown how the second rule of the meta-interpreter is used to generate node types, but the application of the other rules is similar. The procedures called in this algorithm do the following:

- *normalize_lit* and *normalize_goal* rename variables and parameters as required in Definition 7.
- *max_var_index* returns one above the maximal index i of a variable X_i occurring in a node type.
- *fresh_var* adds the given offset to all numbers of variables occurring in a rule. This ensures the disjointness of variables before unification.
- *unify* is the standard unification procedure, however variable-to-variable bindings are done in such a way that never a C_i is replaced by an X_j .
- *append* is the list concatenation.

As explained above, it is possible that longer and longer goals occur during SLD-resolution. In that case, our algorithm will not terminate. However, we can guarantee termination if every recursive body literal, which is not the last literal

of the rule, is evaluated in a subproof. (In fact, it suffices if every recursive loop containing a non-tail-recursive literal is “broken” somewhere by requesting a subproof.) As stated in Lemma 4, in this case the number of nodes in a goal occurring in the SLD-tree is bounded (literals evaluated in a subproof are treated like EDB-literals.). But since node types can contain only the finitely many constants occurring the program or the query, and since the variables/parameters are normalized, there is only a finite number of possible node types.

6 Partial Evaluation

When we know the structure of the occurring *node*-, *query*-, and *answer*-literals, it is a simple task to partially evaluate the rules of the meta-interpreter. In fact, the specialized rules can already be constructed during the computation of the node types, since this computation simply simulates the rule application.

In the rewritten program, the node types become predicates with the actual values of the parameters C_i as arguments. In this way, the C_i and X_i generalize the usual b/f-binding patterns. For instance, the node type

$$\text{node}(\text{anc}(0, X_0), [\text{parent}(C_0, X_1), \text{anc}(X_1, X_0)])$$

is used as a predicate of arity one (because it contains one parameter). We denote the predicate corresponding to

$$\text{node}(Q, [A_1, \dots, A_n])$$

by $\text{node}\langle Q : A_1, \dots, A_n \rangle$. Then the fact

$$\text{node}\langle \text{anc}(0, X_0) : \text{parent}(C_0, X_1), \text{anc}(X_1, X_0) \rangle \quad (3)$$

corresponds to the node $\text{parent}(3, Y) \wedge \text{anc}(Y, X)$ in the SLD-tree with root $\text{anc}(0, X)$.

It is actually very simple to modify the program shown in Figure 3 in such a way that it additionally computes the specialized rule instances. All we need is a procedure *node_params* which constructs an argument list consisting of the parameters C_i contained in a given node type².

As an example, the program to partially evaluate the second rule

```
/* SLD-Resolution: */
node(Query, Child) ←
    node(Query, [Lit|Rest]) ∧
    rule(Lit, Body) ∧
    append(Body, Rest, Child).
```

of the meta-interpreter is shown in Figure 4. There are two points worth noting:

- It is important to select the parameters C_1 of the given node type N before the unification with the head literal, and to apply later the unifier θ . Otherwise, if θ replaces a parameter C_i by a constant a from the head literal A' , the argument list will not be correctly constructed.
- The selection of the parameters of the resulting node type must be done before the normalization, otherwise we would loose the connection between head and body of the rule.

Example 9 Let us for example consider the node type

$$\text{node}(p(C_1), [q(C_2, C_3, C_1, X_1)])$$

and the rule (with fresh variables)

$$q(a, X_2, X_2, X_3) \leftarrow r(X_2, X_3).$$

²It has to work also with non-normalized node types, and creates the argument list in the sequence of occurrence, not in the sequence of their indices (of course, every parameter is represented only once in this list).

```

foreach node type  $N_1$  in  $\mathcal{N}$  do
  let  $N_1 = \text{node}(Q, [A_1, \dots, A_n]);$ 
   $\mathcal{C}_1 := \text{node\_params}(N_1);$ 
  foreach rule  $A \leftarrow \mathcal{B}$  in  $P$  do
     $i := \text{max\_var\_index}(N_1);$ 
     $A' \leftarrow \mathcal{B}' := \text{fresh\_var}(i, A \leftarrow \mathcal{B});$ 
     $\theta := \text{unify}(A_1, A');$ 
    if  $\theta \neq \text{nil}$  then
       $G := \text{append}(\mathcal{B}', [A_2, \dots, A_n]);$ 
       $N_2 := \text{node}(Q\theta, G\theta);$ 
       $\mathcal{C}_2 := \text{node\_params}(N_2);$ 
       $N'_2 := \text{normalize\_node}(N_2);$ 
       $\mathcal{R} := \mathcal{R} \cup \{ \text{node}(N'_2)(\mathcal{C}_2) \leftarrow$ 
                            $\text{node}(N_1)(\mathcal{C}_1\theta) \};$ 
    if  $N'_2 \notin \mathcal{N}$  then
       $\mathcal{N} := \mathcal{N} \cup \{N'_2\};$ 
       $\text{changed} := \text{true};$ 

```

Figure 4: Specialization of Meta-Interpreter Rule #2

The argument list of the given node type is (C_1, C_2, C_3) . The unifier θ replaces C_2 by a , X_2 by C_3 , then C_1 by C_3 (or the other way round), and finally X_1 by X_3 (or vice versa). Before normalization, the resulting “node type” is

$$\text{node}(p(C_3), [r(C_3, X_3)]).$$

Its parameter list is (C_3) . After normalization, the resulting node type is

$$\text{node}(p(C_1), [r(C_1, X_1)]).$$

So we finally get the following rule:

$$\text{node}(p(C_1) : r(C_1, X_1)) (C_3) \leftarrow \text{node}(p(C_1) : q(C_2, C_3, C_1, X_1)) (C_3, a, C_3).$$

It is optionally possible to normalize also the argument lists, i.e. to replace as a last step C_3 in the argument lists by C_1 . But this is only cosmetics. $\square$

Example 10 For the transitive closure example, the complete program resulting from our transformation is shown in Figure 5. Obviously, some optimizations are needed:

- The first four rules are propositional and can already be evaluated during the compilation phase, since they do not depend on the given data.
- Rules number 7 and 8 are pure copying rules and can be eliminated. It is always advantageous to unfold a call to a predicate p if this is the only call to p and there is only one rule about p . In the example,

$$\text{node}(anc(0, X_0) : \text{parent}(C_0, X_0))$$

is defined only in rule 7 and called only in rule 9, and the same happens with rules 8 and 10.

```

/* 1 (Fact for given Query): */
query⟨anc(0, X0)⟩.

/* 2 (Root Node): */
node⟨anc(0, X0) : anc(0, X0)⟩ ←
    query⟨anc(0, X0)⟩.

/* 3 (SLD-Resolution with First Rule): */
node⟨anc(0, X0) : parent(0, X0)⟩ ←
    node⟨anc(0, X0) : anc(0, X0)⟩.

/* 4 (SLD-Resolution with Second Rule): */
node⟨anc(0, X0) : parent(0, X1), anc(X1, X0)⟩ ←
    node⟨anc(0, X0) : anc(0, X0)⟩.

/* 5 (Evaluation of DB-Literal): */
node⟨anc(0, C0) : true⟩ (C0) ←
    node⟨anc(0, X0) : parent(0, X0)⟩ ∧
    parent(0, C0).

/* 6 (Evaluation of DB-Literal): */
node⟨anc(0, X0) : anc(C0, X0)⟩ (C0) ←
    node⟨anc(0, X0) : parent(0, X1), anc(X1, X0)⟩ ∧
    parent(0, C0).

/* 7 (SLD-Resolution with First Rule): */
node⟨anc(0, X0) : parent(C0, X0)⟩ (C0) ←
    node⟨anc(0, X0) : anc(C0, X0)⟩ (C0).

/* 8 (SLD-Resolution with Second Rule): */
node⟨anc(0, X0) : parent(C0, X1), anc(X1, X0)⟩ (C0) ←
    node⟨anc(0, X0) : anc(C0, X0)⟩ (C0).

/* 9 (Evaluation of DB-Literal): */
node⟨anc(0, C0) : true⟩ (C1) ←
    node⟨anc(0, X0) : parent(C0, X0)⟩ (C0) ∧
    parent(C0, C1).

/* 10 (Evaluation of DB-Literal): */
node⟨anc(0, X0) : anc(C0, X0)⟩ (C1) ←
    node⟨anc(0, X0) : parent(C0, X1), anc(X1, X0)⟩ (C0) ∧
    parent(C0, C1).

/* 11 (Turn Proven Query into Answer): */
anc(0, C0) ←
    node⟨anc(0, C0) : true⟩ (C0).

```

Figure 5: Rewritten Program (Without Optimizations)

The state after these optimizations is:

```

node(anc(0, C0) : true) (C0) ←           /* 5 */
    parent(0, C0).

node(anc(0, X0) : anc(C0, X0)) (C0) ←     /* 6 */
    parent(0, C0).

node(anc(0, C0) : true) (C1) ←           /* 9 */
    node(anc(0, X0) : anc(C0, X0)) (C0) ∧
    parent(C0, C1).

node(anc(0, X0) : anc(C0, X0)) (C1) ←     /* 10 */
    node(anc(0, X0) : anc(C0, X0)) (C0) ∧
    parent(C0, C1).

anc(0, C0) ←                               /* 11 */
    node(anc(0, C0) : true) (C0).

```

Now the predicates

```
node(anc(0, C0) : true)
```

and

```
node(anc(0, X0) : anc(C0, X0))
```

have identical definitions. We can assume that some form of common subexpression elimination detects this. Let us replace the two synonyms by a predicate *anc₀*.

With the help of these optimizations, we get a very reasonable program for computing the reachable nodes:

```

anc0(C0)    ←  parent(0, C0).           /* 5/6 */
anc0(C1)    ←  anc0(C0) ∧ parent(C0, C1). /* 9/10 */
anc(0, C0)  ←  anc0(C0).               /* 11 */

```

This seems to be even slightly superior to the standard optimized version of this program [11, 4]:

```

m_ancbf(0).
m_ancbf(Z)  ←  m_ancbf(X) ∧ parent(X, Z).
ans(Y)      ←  m_ancbf(X) ∧ parent(X, Y).

```

But even without any optimizations, bottom-up evaluation of the transformed program in Figure 5 is already as efficient as SLD-resolution in the sense that any applicable rule instance corresponds to an edge in the SLD-tree. And since we have seen how much we really still can optimize, there are good chances that bottom-up evaluation really can beat SLD-resolution. $\square$

7 Magic Sets with Supplementary Predicates

Let us now consider the special case that all IDB-literals are evaluated in subproofs.

A small inefficiency in the above meta-interpreter is that we first must turn the query into a goal and only then can apply the rules. In order to improve this, we can treat already *query(Query)* as representing the root node. This is done by replacing the first rule of the meta-interpreter by the following optimized version:

```

/* Initialization (Select Applicable Rules for Query): */
node(Query, Body) ←
    query(Query),
    rule(Query, Body).

```

Now all node types have the form $node(A\theta, [B_i, \dots, B_n]\theta)$, where $A \leftarrow B_1 \wedge \dots \wedge B_n$ is a rule of the given program and θ replaces variables, which are bound in the call to A or in $B_1 \wedge \dots \wedge B_{i-1}$, by parameters (and normalizes the variable names). Note that only values of variables occurring in $B_i \wedge \dots \wedge B_n$ or A are represented by parameters. So the arguments correspond exactly to the arguments of the i -th supplementary predicate of this rule. Furthermore, if we compare the resulting rules with the magic set transformation of [11], we see a striking similarity: The rules for the zeroth supplementary predicates are generated by the first rule of our meta-interpreter, the rules for the other supplementary predicates are generated by the fourth and fifth rule, the rules for the magic predicates are generated by the third rule, and the rules for IDB-predicates correspond to our last rule. To be precise, there are only two very minor differences:

- As presented in [11], there is no n -th supplementary predicate for a rule with n body literals. This would correspond to nodes of the form $node(A\theta, [])$. It is a simple optimization to eliminate it.
- Second, if two rules have the same head A and body rest $B_i \wedge \dots \wedge B_n$, we merge the two supplementary predicates. But this can only improve the performance.

8 Conclusions

This work has tried to clarify the relation between SLD-resolution and magic sets, and has introduced a joint generalization through a meta-interpreter. It was the goal of the magic set transformation to combine all advantages of bottom-up and top-down evaluation (i.e. SLD-resolution). However, this goal was not fully reached, and our approach goes further in this direction.

Our rewriting method allows to decide between SLD-resolution and magic sets for every literal, and choose whatever seems better. If there is only one call of a predicate, or we can see that the arguments are disjoint, or the computation is so simple that it does no harm to repeat it, then we should choose SLD-resolution. Also, the potential savings for tail-recursive literals are so big that they probably outweigh the cost of duplicate computations. On the other hand, recursive calls which are not tail-recursive can only be processed via magic sets.

A first prototype implementation of the computation of node types and the partially evaluated meta-interpreter is available from

`ftp://ftp.informatik.uni-hannover.de/software/sldmagic/sldmagic.html`

An especially nice feature of our method is that it gives many known optimizations “for free”, namely:

- No extra rectification is needed.
- Values for anonymous variables are never explicitly represented.
- Nonrecursive programs are transformed into nonrecursive programs.
- Already at compile-time, constants are pushed “downward” into the called rules.
- Our method pushes also non-equality conditions, such as $X < 20$ into the called literals. These conditions are represented in the “context” or “continuation” of a literal, and a good selection function would evaluate them as soon as X becomes bound. In fact, we should check the consistency of the goals already at compile-time (as far as possible). In the standard magic set transformation, such optimizations are not easy [5].

At the moment, we have either passed the complete context with the called literal or no context at all. This seemed reasonable, since as soon as we have not the complete context, we need a join to get back to the caller. However, incomplete contexts can still help to delimit the computations within the called literal. For instance, we would like to pass the condition $X < 20$ even to a left-recursive literal, where we cannot represent all possible continuations completely.

Let us also note that many deductive database systems have some procedure call mechanism for the predicates, and do not use the pure “magic set” transformation. This leads to some double computations, but avoids the merging of different “calls” to a predicate. Our approach can be understood as going further in this direction.

It seems that future improvements need to look at the internal data structures. We still have considered only a very abstract level and talked about SLD-trees and applicable rule instances. Now the time has come to look at the lower implementation levels. Currently, Prolog implementations are still much faster than deductive databases (for instance, the ECLiPSe Prolog-System is 10 – 20 times faster than the CORAL system, if the problems mentioned above do not occur). This probably can be improved, or else it would be useful to know the theoretical reasons why set-oriented evaluation is inherently slower than tuple-oriented evaluation (what I do not believe).

Acknowledgement

I would like to thank UDO LIPECK for leading me the way to the research presented here and for numerous helpful suggestions on an earlier version of this paper. I also profited from discussions with KLAUS-DIETER SCHEWE, TORSTEN POLLE, and ULRICH ZUKOWSKI.

References

- [1] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. In C. Zaniolo, editor, *Proc. of SIGMOD'86*, pages 16–52, 1986.
- [2] S. Brass. Magic sets vs. SLD-resolution. In J. Eder and L. A. Kalinichenko, editors, *Advances in Databases and Information Systems (ADBIS'95)*, pages 185–203. Springer, 1995.
- [3] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.
- [4] D. B. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. Very Large Data Bases, 16th Int. Conf. (VLDB'90)*, pages 380–391. Morgan Kaufmann Publishers, 1990.
- [5] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proc. of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, pages 314–330, 1990.
- [6] R. Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. In V. Saraswat and K. Ueda, editors, *Proc. of the 1991 Int. Symposium on Logic Programming*, pages 321–336. MIT Press, 1991.
- [7] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proc. of the Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Princ. of Database Systems (PODS'91)*, pages 92–101, 1991.
- [8] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.
- [9] S. Sudarshan. *Optimizing Bottom-Up Query Evaluation for Deductive Databases*. PhD thesis, University of Wisconsin - Madison, 1992.
- [10] J. D. Ullman. Bottom-up beats top-down for Datalog. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pages 140–149, 1989.
- [11] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 2*. Computer Science Press, 1989.