In: Computational Logic — CL'2000, Sixth International Conference on Rules and Objects in Databases, Springer, LNCS 1861, 2000, pages 1063-1077.

SLDMagic — The Real Magic (with Applications to Web Queries)*

Stefan Brass

University of Pittsburgh, Dept. of Information Science and Telecommunications,
135 N. Bellefield Ave., Pittsburgh, PA 15260, USA
sbrass@sis.pitt.edu

Abstract. The magic set technique is a standard technique for query evaluation in deductive databases, and its variants are also used in modern commercial database systems like DB2. Numerous improvements of the basic technique have been proposed. However, each of these optimizations makes the transformation more complicated, and combining them in a single system is at least difficult.
In this paper, a new transformation is introduced, which is based on partial evaluation of a bottom-up meta-interpreter for SLD-resolution. In spite of its simplicity, this technique gives us a whole bunch of optimizations for free: For instance, it contains a tail recursion optimization, it transforms non-recursive into non-recursive programs, it can pass arbitary conditions on the parameters to called predicates, and it saves the join necessary to get subquery results back into the calling context. In this way, it helps to integrate many of the previous efforts.
The usefulness of these optimizations is illustrated with example programs querying the World Wide Web.

1 Introduction

Many current developments aim at integrated systems consisting of a programming language and a database management system. For instance, object-oriented database systems combine both functionalities, but also stored procedures and triggers in relational systems go into this direction. Deductive databases offered such an integrated language for a long time. Theoretically, this is very appealing since here a declarative language is also used for the programming part. Declarativity has proven to be very useful in SQL.

Deductive database systems have also become interesting again because they are well suited to process graph-structured data, and the World Wide Web can be seen as a large directed graph of interconnected documents. This view of the WWW is the basis of web query languages, e.g. [KS95,MMM97,HLLS97]. Also, recently proposed data models for semi-structured data and XML [ABS00] as well as the RDF model for Web metadata are graph-structured.

* This paper is a completely rewritten and significally extended version of a paper which appeared in the electronic proceedings of the International Workshop on "Advances in Databases and Information Systems", Moscow, 1996.

One of the biggest problems of deductive databases is still the performance, which is quite far behind other integrated DB/PL systems. It is known that "Bottom-Up [evaluation with magic sets] Beats Top-Down for Datalog" [Ull89a]. However, as noted by Ross [Ros91] (see also [Ull89a]), this does not mean that current deductive databases are at least as efficient as Prolog implementations. This is not even true asymptotically (in O-notation). So let us quickly explain the main difference between magic sets and SLD-resolution (which is the basis of Prolog evaluation). Although both are top-down evaluation methods, and in fact equally goal-directed (see, e.g., [Bra95]), there are important differences.

The magic set method treats predicates (views) like procedures, which are called with a set of bindings for the input (bound) arguments. This input relation is the so-called "magic set". They return a relation for all arguments, such that every returned tuple agrees with one input tuple in the bound arguments. So the result is the semijoin of the magic set and the full extension of the predicate. Of course, the trick is to avoid computing this full extension.

For instance, consider a predicate local_link(From_URL, To_URL, Label) which returns links in the web page From_URL refering to a page To_URL on the same server. An invocation with the first argument bound could look as follows:

From_URL
http://x.edu/
http://y.edu/

$\xrightarrow{\text{local\_link}}$

From_URL	To_URL	Label
http://x.edu/	http://x.edu/a	...
http://x.edu/	http://x.edu/b	...
http://y.edu/	http://y.edu/c	...

In contrast, SLD-resolution works by repeatedly "unfolding" query literals — it replaces the predicate call by the predicate definition. This is what many relational database systems do with view definitions, but in SLD-resolution this is the only computation mechanism and works also with recursive views. Let local_link be defined as follows:

$$\begin{aligned}\mathsf{local\_link}(\mathsf{From\_URL}, \mathsf{To\_URL}, \mathsf{Label}) \leftarrow\ & \mathsf{link}(\mathsf{From\_URL}, \mathsf{To\_URL}, \mathsf{Label}) \wedge \\ & \mathsf{same\_server}(\mathsf{From\_URL}, \mathsf{To\_URL}).\end{aligned}$$

Furthermore, let the query be

$$\begin{aligned}&\mathsf{local\_link}(\text{'}\mathsf{http://www.pitt.edu}\text{'}, \mathsf{URL}, \mathsf{Label}) \wedge \\ &\mathsf{like}(\mathsf{Label}, \text{'}\mathsf{\%Inf\%Sc\%}\text{'}).\end{aligned}$$

SLD-resolution replaces this query by

$$\begin{aligned}&\mathsf{link}(\text{'}\mathsf{http://www.pitt.edu}\text{'}, \mathsf{URL}, \mathsf{Label}) \wedge \\ &\mathsf{same\_server}(\text{'}\mathsf{http://www.pitt.edu}\text{'}, \mathsf{URL}) \wedge \\ &\mathsf{like}(\mathsf{Label}, \text{'}\mathsf{\%Inf\%Sc\%}\text{'}).\end{aligned}$$

In SLD-resolution, there is no explicit procedure call and return. Instead, we always work on complete continuations of the computation. Even if we should choose to evaluate next the calls to link and same_server, the control passes then immediately to like without entering the rule for local_link again. This is essential for tail recursions. Furthermore, we can choose a different evaluation sequence, for instance evaluate the call to like before the call to same_server. This gives us a much bigger optimization potential than the sideways information passing rule of magic sets, which can only locally reorder the body literals within a rule (or decide not to use all available bindings)[1].

Of course, SLD-resolution also has its problems, the most important being the possibility of non-termination. There are tabulation techniques which avoid this [SSW94], but these are essentially equivalent the magic set method. In this paper, we present a new method to combine advantages of bottom-up evaluation and SLD-resolution. The title says that this is the "real magic", because we believe that it was from the beginning the goal of the magic set transformation to combine bottom-up evaluation with Prolog evaluation (i.e. SLD-resolution).

Deductive databases are normally applied when there are large sets of facts which Prolog implementations cannot handle. While we do SLD-resolution as Prolog, we execute it on a bottom-up machine using set-oriented evaluation techniques. Whereas Prolog always does nested loop joins, we can apply merge-joins or hash-joins. Also, we will see that the SLD-resolution selection function, which is not used in Prolog, can be an important means for query optimization.

Our approach is based on the idea of partially evaluating a meta-interpreter. Bry has done this for the standard magic set technique [Bry90], we only start with another meta-interpreter and do a bit more involved partial evaluation. It is fascinating how many optimizations we get for free based on this idea. Such optimizations are known for the magic set method [MFPR90,Ros91,GM92], but integrating them in a single system is at least hard work.

2 Problems of Magic Set Query Evaluation

Let us consider some examples which demonstrate weak points of the standard magic set technique. We will see that an approach based on SLD-resolution can avoid these problems. Specialized solutions to most of these shortcomings have already been developed. Our contribution is an integrated approach which solves all of these problems (and is actually quite simple).

2.1 Tail Recursions

It is a standard task to find all documents which are reachable from a given document via local links (i.e. links refering to documents on the same server). In order to do this, we first define a predicate for the transitive closure (corresponding to $\longrightarrow^*$ in WebSQL [MMM97]):

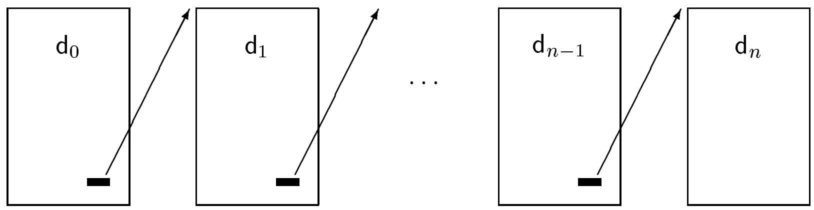
$$\mathsf{local\_reachable}(X, Y) \leftarrow \mathsf{local\_link}(X, Y, \_).$$
$$\mathsf{local\_reachable}(X, Z) \leftarrow \mathsf{local\_link}(X, Y, \_) \wedge \mathsf{local\_reachable}(Y, Z).$$

[1] "Informally, for a rule of a program, a sip represents a decision about the order in which predicates of the rule will be evaluated, and how values for variables are passed from predicates to other predicates during evaluation." [BR91]

Then we can call this predicate with the given start document, say d_0:

$$\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D}).$$

To keep the example simple, let us consider the following hypertext structure:



However, the same problem appears when we have additional links, and this path is only a subgraph. Actually, we could have used any reasonable connection of $n+1$ pages (e.g. a star topology with backlinks).

In order to solve the task, we must only follow the links and output every page we reach. Thus a complexity of $O(n)$ seems reasonable, or $O\big(n * \log(n)\big)$, if we check for cycles due to backlinks. But if we use the magic set technique in this example, the complexity is at least $O(n^2)$. The reason is that this method explicitly represents the results of subqueries. We start with the call $\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D})$, but since there is a link to page d_1, we get the recursive call $\mathsf{local\_reachable}(\mathsf{d}_1, \mathsf{D})$, and so on, for any page of the chain. For each such subquery, the magic set method computes all matching facts which follow from the original program. So we not only get $\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{d}_1), \ldots, \mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{d}_n)$, but also $\mathsf{local\_reachable}(\mathsf{d}_1, \mathsf{d}_2)$, and so on. This is a quadratic number of facts, thus the complexity is at least $O(n^2)$, and probably higher due to join computations and duplicate eliminations.

In contrast, Prolog can process this example in linear time, and this can be understood without looking inside Prolog implementations. The tree of goals (queries) created by SLD-resolution for the above program and data (including the rule for $\mathsf{local\_link}$) contains $8n+5$ nodes, and all nodes have ≤ 3 literals.

However, SLD-resolution will not terminate for cyclic hypertext graphs. But our method of evaluating SLD-resolution bottom-up will compute only a finite number of SLD-goals, and does so in the required time $O\big(n*\log(n)\big)$. In contrast, previous tabulation methods for making SLD-resolution terminate, such as those used in the XSB-system [SSW94], have the same problem as magic sets: They store proven instances of literals in a table, which is already a quadratic number.

The magic set method with tail-recursion optimization developed in [Ros91] and further analyzed in [RS91,SR93] solves the problem. There are also methods for more specific kinds of tail-recursions [NRSU89,Ull89b,KRS90]. However, our method contains such optimizations, and solves many other problems as well.

Current query languages for the web, semistructured data, and XML typically contain path expressions for following edges in the graph. There are specialized algorithms for evaluating these expressions which of course do not have this problem. While path expressions work well for XML, retrieving a page on the web is an expensive operation, so we might need the full power of Datalog to describe as precisely as possible which links we want to follow.

2.2 Nonrecursive Programs

The following predicate computes pages reachable via at most two local links:

$$\mathsf{reach2}(\mathsf{X},\mathsf{Y}) \leftarrow \mathsf{local\_link}(\mathsf{X},\mathsf{Y},\_).$$
$$\mathsf{reach2}(\mathsf{X},\mathsf{Z}) \leftarrow \mathsf{local\_link}(\mathsf{X},\mathsf{Y},\_) \land \mathsf{local\_link}(\mathsf{Y},\mathsf{Z},\_).$$

The program is non-recursive and should be easy to evaluate. However, if we use the magic set technique for the query $\mathsf{reach2}(\mathsf{d}_0,\mathsf{D})$, we get a recursive program. The reason for this problem is that the magic set technique collects all calls to a predicate (with the same "binding pattern") into a single magic predicate. But here, due to the second rule for reach2, the queries for local_link (in the second body literal) depend on solutions for local_link (in the first body literal). And of course, solutions for a predicate always depend conversely on the queries.

In contrast, SLD-resolution treats the two calls to local_link separately, and thus the problem does not occur. Of course, merging calls sometimes can be advantageous, if this helps to avoid recomputations. Therefore our method can be parameterized in such a way that for every body literal either magic sets or SLD-resolution can be chosen.

The methods of [GM92] ensure that non-recursive programs are transformed into non-recursive programs. The basic approach is also to distinguish the two calls. In one of their solutions, they also do some unfolding, and add a "covered subgoal elimination" which we do not (yet) have. Again, the strength of our solution is that it solves different problems at the same time.

2.3 Getting Results Back into the Context of the Caller

Suppose we have a relation my_links(URL, Last_Visited) in which we store our personal collection of most interesting web pages. It is the combination of such local information with a web interface which makes web query languages a useful and powerful tool. Now the following predicate returns those pages which have changed since the time of last visit:

$$\begin{aligned}\mathsf{has\_changed}(\mathsf{URL}) \leftarrow\ & \mathsf{my\_links}(\mathsf{URL},\mathsf{Last\_Visited}) \land \\ & \mathsf{doc\_mtime}(\mathsf{URL},\mathsf{Modif}) \land \\ & \mathsf{Modif} > \mathsf{Last\_Visited}.\end{aligned}$$

When has_changed is called with URL free, the magic set method will evaluate the body literals in the given sequence. So it first accesses the relation my_links. This gives bindings for the two variables URL and Last_Visited. Let us assume that doc_mtime is also an IDB-predicate:

$$\mathsf{doc\_mtime}(\mathsf{URL},\mathsf{Modif}) \leftarrow \mathsf{www\_get}(\mathsf{URL},\mathsf{Title},\mathsf{Modif},\mathsf{Contents}).$$

Then a magic set for doc_mtime is constructed by projecting the bindings for URL and Last_Visited on the variable URL. The predicate doc_mtime returns bindings for URL and Modif. But in the context of the calling rule, all three variables URL,

$$\mathsf{my\_links}(\mathsf{URL}, \mathsf{Last\_Visited}) \ \wedge\ \mathsf{doc\_mtime}(\mathsf{URL}, \mathsf{Modif}) \ \wedge\ \mathsf{Modif} > \mathsf{Last\_Visited}$$

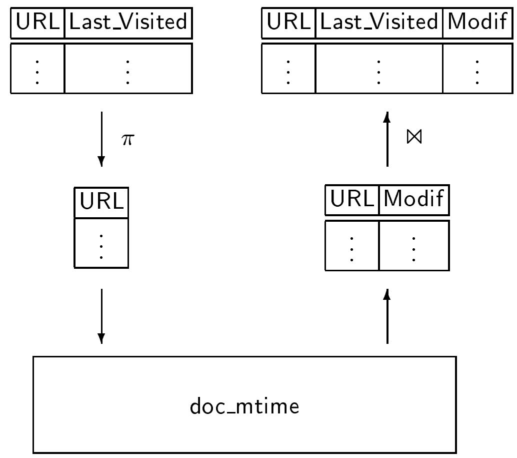


Fig. 1. Projection and Join During Magic Set Evaluation

Last_Visited, and Modif are bound, and all three variables are still needed. Thus, the two relations must be joined on the attribute URL (see Figure 1).

Since a join is expensive, it would have been better not to project the variable Last_Visited away. SLD-resolution always retains the complete bindings for all variables which are still needed: The goals in the SLD tree contain the full continuation of the computation.

Magic sets with supplementary predicates do not solve this problem. This method reduces unnecessary recomputations while evaluating a single rule, but does not change the arguments of the IDB-predicates, as would be required here.

Magic sets could be advantageous to SLD-resolution if e.g. my_links would produce several solutions for a single URL (In the given example, this cannot happen because URL is key). Then the projection could do a duplicate elimination and thereby reduce the input to doc_mtime.

2.4 Passing Conditions on the Parameters to Called Predicates

Suppose that we are again interested in pages from our hotlist which were modified since our last visit, but only from a specific server. Then we use the query

$$\mathsf{has\_changed}(\mathsf{URL}) \wedge \mathsf{on\_server}(\mathsf{URL}, \text{'}\mathsf{www.sis.pitt.edu}\text{'}).$$

Consider now the evaluation of has_changed: While my_links is a locally stored relation, the call to the WWW-interface predicate doc_mtime is very expensive: We have to fetch each page under all URLs stored in my_links, and only later throw out all pages which are not on the given server.

The problem here is that for the magic set technique, the structure of the program in predicates is significant: The "sideways information passing" strategy cannot move literals between predicate boundaries. However, in this case, the optimal sequence would be

$$\begin{aligned}&\mathsf{my\_links}(\mathsf{URL}, \mathsf{Last\_Visited}) \land\\ &\mathsf{on\_server}(\mathsf{URL}, \text{'www.sis.pitt.edu'}) \land\\ &\mathsf{doc\_mtime}(\mathsf{URL}, \mathsf{Modif}) \land\\ &\mathsf{Modif} > \mathsf{Last\_Visited}.\end{aligned}$$

The literals $\mathsf{on\_server}(\mathsf{URL}, \text{'www.sis.pitt.edu'})$ and $\mathsf{Modif} > \mathsf{Last\_Visited}$ are very cheap to evaluate, however, they can only be evaluated after their arguments are bound. And of course, we should try to "fail as early as you can", at least before very expensive literals are evaluated. This corresponds to the classical optimization strategy to push selections as far "down" as possible, and especially evaluate them before expensive joins.

The magic set technique can pass only conditions of the form $\mathsf{X} = \mathsf{const}$ on the parameters to the called predicates. The rectification transformation [Ull89b] was invented to handle conditions of the form $\mathsf{X} = \mathsf{Y}$. The technique of [MFPR90] can pass conditions of the form $\mathsf{X} < \mathsf{const}$. Our method inherits from SLD-resolution the possibility to evaluate arbitrary conditions of the parameters as soon as they become bound. In SLD-resolution this kind of "global optimization" is done by means of the selection function: It decides which literal from the continuation should be evaluated next. In this way, it considerably generalizes the magic set SIP-strategy (at least the reordering part. The SIP-strategy also can decide to use only a subset of the available bindings.)

While we reach the same optimization as [MFPR90], the method of [SR92] has features which would have to be added to our approach. It can move linear arithmetic constraints both from the uses of a predicate into its rules as well as from the definitions towards its uses. Our method breaks up the rule structure so that constraints to be satisfied are visible when we evaluate a predicate. However, it evaluates them only as soon as the become bound, we do not yet check the constraints for consistency. We also do not generate constraints for predicates which would help to detect inconsistencies earlier. On the other hand, our method is not limited to any particular type of constraints. Any predicate which can be evaluated cheaply can act as a constraint.

2.5 Combining Conditions for Index Access

This greater flexibility in the evaluation order is also important for index structures which can evaluate conjunctions of literals. For instance, when we submit a query to a search engine, we should first collect all literals specifying search terms for the same document.

Suppose we have defined a predicate containing the URLs of possible job offers in the Web:

$$\begin{aligned}&\mathsf{job\_offer}(\mathsf{URL}) \leftarrow \mathsf{keyword}(\text{'Job Opportunity'}, \mathsf{URL}).\\ &\mathsf{job\_offer}(\mathsf{URL}) \leftarrow \mathsf{keyword}(\text{'Free Positions'}, \mathsf{URL}).\end{aligned}$$

Here the predicate keyword gives access to a search engine. While this particular definition of the predicate job_offer is very naive, such predicates can be used to represent knowledge about searching in the WWW. The possibility to reuse and share such knowledge is an important issue for future web querying systems.

Now suppose that we are interested only in job offers mentioning "Prolog":

$$\mathsf{job\_offer}(\mathsf{URL}) \wedge \mathsf{keyword}(\text{'Prolog'}, \mathsf{URL}).$$

Then it might be important that when we evaluate the call to job_offer, we already see that there is another call to the predicate keyword. It will certainly be better to combine the search terms, and not to collect first all job offers and then to select those mentioning "Prolog".

In general, index structures often allow to evaluate conjunctions of literals at once. Even a classical B-tree over e.g. the attribute Sal of the relation emp allows us to evaluate a conjunction like $\mathsf{emp}(\mathsf{X}, \mathsf{Y}, \mathsf{Sal}) \wedge \mathsf{Sal} \geq 1000 \wedge \mathsf{Sal} \leq 1500$ in one shot. However, in the source program, these conditions might not be contained in the same rule. Therefore, good query optimization needs the unfolding power of SLD-resolution.

It is sometimes assumed that standard relational query optimization can be done after the magic-set transformation. In our view, this is an error. The result of the transformation prescribes more or less the evaluation order. So many physical parameters (such as the existence of indexes) must already be taken into account when the transformation is done.

3 The Meta-Interpreter

Often, an evaluation method can be explained by presenting an interpreter for it. If this interpreter is written in the language itself, it is called a meta-interpreter. It is a standard exercise in Prolog programming courses to write an interpreter for Prolog in Prolog. However, Bry clarified in [Bry90] that such interpreters depend heavily on the machine model used to execute them. While the standard meta-interpreter runs only on Prolog, Bry developed a meta-interpreter which formalized top-down evaluation, but run itself on a bottom-up machine. He used explicit call and return, and in this way reconstructed the standard magic set transformation. So all we have to do now is to start with a meta-interpreter which describes real SLD-resolution.

3.1 Bottom-Up Execution of SLD-Resolution

We present SLD-goals (nodes of the SLD-tree) by lists of literals which still have to be proven. For instance, the query $\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D})$ gives the root node of the SLD-tree:

$$\mathsf{node}\big([\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D})]\big).$$

The rules of the given program are stored in the form rule(Head, Body), e.g.

$$\mathsf{rule}\big(\mathsf{local\_reachable}(\mathsf{X}, \mathsf{Z}),\ [\mathsf{local\_link}(\mathsf{X}, \mathsf{Y}, \_),\ \mathsf{local\_reachable}(\mathsf{Y}, \mathsf{Z})]\big).$$

Now the SLD-resolution step can be described by means of the following main rule of our meta-interpreter (for simplicity, we have chosen here the "first literal" selection function of Prolog):

$$\begin{aligned}&\mathsf{node}(\mathsf{Child}) \leftarrow\\ &\qquad \mathsf{node}([\mathsf{Lit}|\mathsf{Rest}]) \land\\ &\qquad \mathsf{rule}(\mathsf{Lit},\ \mathsf{Body}) \land\\ &\qquad \mathsf{append}(\mathsf{Body},\ \mathsf{Rest},\ \mathsf{Child}).\end{aligned}$$

Our meta-interpreter will be evaluated bottom-up, so you have to read the rule from right to left: If we insert, e.g., the above node- and rule-facts, we will derive the following node-fact:

$$\mathsf{node}\big([\mathsf{local\_link}(\mathsf{d}_0, \mathsf{Y}, \_),\ \mathsf{local\_reachable}(\mathsf{Y}, \mathsf{D})]\big).$$

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. In addition, it treats derived facts as duplicates if they differ only by a variable renaming from known facts. This is important for the termination and can be easily achieved by normalizing variable names (e.g. $\mathsf{X}_1, \mathsf{X}_2, \ldots$).

There is the small problem that in this way it is difficult to track the binding for the answer variable D. When all literals are proven, we get the empty goal node([ ]), but the answer substitution is lost. We solve this problem by adding to each derived node-fact the current instance of the query. This will be the first argument of the predicate node, the second argument will be the current goal as above. So instead of the above node-fact we really derive

$$\mathsf{node}\big(\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D}),\ [\mathsf{local\_link}(\mathsf{d}_0, \mathsf{Y}, \_),\ \mathsf{local\_reachable}(\mathsf{Y}, \mathsf{D})]\big).$$

This is similar to a rule where the head always remains an instance of the query and we iteratively unfold the body. Since the substitutions are also applied to the first argument, it contains the proven query instance as soon as the goal becomes empty:

$$\mathsf{node}\big(\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{d}_2),\ [\,]\big).$$

The complete meta-interpreter is shown in Figure 2. We assume there that EDB-facts from the database are stored in the predicate db. The distinction between program rules with empty bodies and database facts becomes relevant only later when we do partial evaluation. For simplicity, we assume that the query is a single literal stored in the predicate query. The meta-interpreter can be executed by deductive database systems like CORAL [RSSS94] which allow structured terms and non-ground facts.

Theorem 1 (Relation to SLD-Resolution). *Let the above meta-interpreter be executed on* rule, db, *and* query-*facts corresponding to a program* P, *database* DB, *and query* Q. *Then it computes the goals in the SLD-tree for* $\mathsf{P} \cup \mathsf{DB} \cup \{\leftarrow \mathsf{Q}\}$:

- *For every node* $\mathcal{N}$ *in the SLD-tree with goal* $\leftarrow \mathsf{A}_1 \land \cdots \land \mathsf{A}_n$, *there is a fact* $\mathsf{node}\big(\mathsf{Q}\theta,\ [\mathsf{A}'_1, \ldots, \mathsf{A}'_n]\big)$ *which is derivable from the meta-interpreter and a*

```
/* 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, []).
```

Fig. 2. Bottom-Up Meta-Interpreter for SLD-Resolution

variable-renaming σ *such that* $\mathsf{A}'_i\sigma = \mathsf{A}_i$, $i = 1, \ldots, n$, *and* $\mathsf{Q}\theta\sigma$ *is the result of applying to the query all most general unifiers which SLD-resolution used on the way from the root node to* $\mathcal{N}$.

- *And vice versa, every derivable fact corresponds in this way to (at least) one node in the SLD-tree.*

From the soundness and completeness of SLD-resolution, we directly get the following corollary:

Theorem 2 (Soundness and Completeness). *Let the meta-interpreter be executed on* rule, db, *and* query*-facts corresponding to a program* P, *database* DB, *and query* Q.

- *For every derived fact* $\mathsf{answer}(\mathsf{Q}\theta)$, *the substitution* θ *is a correct answer substitution.*
- *For every correct answer substitution* θ, *there is a derived fact* $\mathsf{answer}(\mathsf{Q}\theta')$ *and a substitution* σ *with* $\theta = \theta'\sigma$.

3.2 Termination

So our meta-interpreter correctly simulates SLD-resolution. As explained in Section 2, this is advantageous for many applications. But do we get in exchange for these advantages also the problem of possible non-termination? The answer is: Often not. 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 $\mathsf{p(X)} \leftarrow \mathsf{p(X)}$ 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 with other kinds of recursions. For such programs, we will later present a combined method which allows to use the "magic set" behaviour (tabulation) for calling some literals.

Definition 1 (Tail-Recursive Program). *A program is at most tail-recursive iff for every rule*

$$\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m,$$

the predicates of B_i, $1 \le i \le m-1$, *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 larger than the class for which the "right recursion optimization" of [Ull89b] is applicable. Most practical programs are covered. The condition ensures that the number of literals in SLD-goals is bounded (assuming the left-to-right selection function).

Theorem 3 (Sufficient Condition for Termination). *Let* P *be an at most tail-recursive program,* DB *a database, and* Q *be a query such that* $\mathsf{P} \cup \mathsf{DB} \cup \{\leftarrow \mathsf{Q}\}$ *is finite and does not contain structured terms. Then the bottom-up evaluation of the above meta-interpreter terminates, i.e. there are only finitely many facts derivable from it (modulo variable renamings).*

3.3 Adding "Magic Set" Behaviour

Because of the problems with general recursive calls, we might be interested to evaluate such literals with the magic set technique. Also, the strength of magic sets is that every predicate is evaluated only once for the same input values. While often the behaviour of SLD-resolution is better, we sometimes might want to table calls and computed results in order to avoid unnecessary recomputations. Fortunately, it is easy to extend the meta-interpreter in such a way that we can choose for every body literal whether it should be evaluated via SLD-resolution or via magic sets.

Let us enclose body literals intended for magic set evaluation into the special predicate call. Then it suffices to add the two rules in Figure 3 to our meta-interpreter. The idea is that we allow SLD-resolution to call itself recursively for evaluating certain literals (like standard SLD-resolution does for negative literals). So we now construct not a single SLD-tree, but one for each recursive call. This explict call and return is the key to understanding the difference between magic sets and SLD-resolution. One can view the two rules also as describing SLD-resolution with tabulation: The first rule enters a predicate call into a table, and the second rule takes solutions from a table in order to solve this literal.

If all IDB-literals are evaluated in subproofs, we get something very similar to magic sets with supplementary predicates: The query-facts correspond to magic facts, answer-facts correspond to derived IDB-facts, and node-facts correspond to facts of the supplementary predicates.

```
/* Set Up Recursive Call (Derive Magic Fact): */
query(Lit) ←
        node(_, [call(Lit)|_]).

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

Fig. 3. Additional Meta-Interpreter Rules for "Magic Set" Behaviour

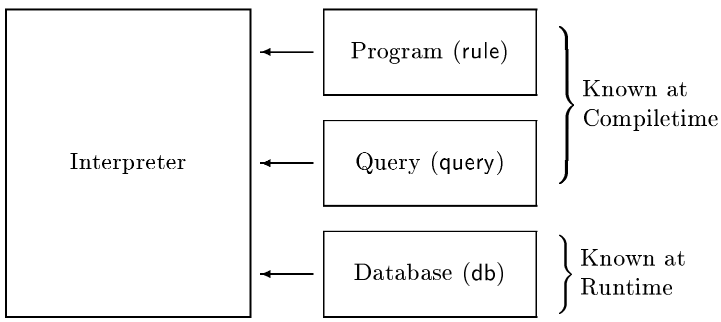


Fig. 4. Inputs of the Meta-Interpreter

4 Partial Evaluation

While the above meta-interpreter can be directly executed (e.g. on CORAL), the use of lists and non-ground facts significantly decreases the performance. It is well known that from an interpreter, one can get a compiler via partial evaluation. Such a compiler will transform a program intended for SLD-evaluation into a program which runs on a bottom-up machine. For BRY's meta-interpreter, it was sufficient to unfold the call to the predicate rule. In our case, partial evaluation becomes a bit more complicated. There are a number of papers which investigate partial evaluation for Prolog (i.e. top-down evaluation), but partial evaluation for bottom-up execution seems to be a new problem.

Our task is to evaluate the meta-interpreter as far as possible, given program and query, but with a yet unknown database (see Figure 4). Especially, we should try to avoid using lists and non-ground facts. This is feasible, since the number of literals in node-facts is bounded as long as the program is at most tail-recursive or other recursions are evaluated via call.

Our main idea is to use conditional facts of the form A ← B to separate what is known at compile-time (A) from what is only at runtime (B). For instance, we might know at compilation time that we can derive facts of the form

$$\mathsf{node}\big(\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D}),\ [\mathsf{local\_reachable}(\mathsf{d}, \mathsf{D})]\big).$$

This corresponds to the situation that we have followed links from the start page d_0 to some page d, and therefore any page D reachable from d is also reachable from d_0. Of course, the possible values for d depend on the data, and are not yet known at compile time. So we would encode this knowledge as

$$\mathsf{node}\big(\mathsf{local\_reachable}(\mathsf{d}_0, \mathsf{D}),\ [\mathsf{local\_reachable}(\mathsf{X}, \mathsf{D})]\big) \ \leftarrow\ \mathsf{p}(\mathsf{X})$$

where p is a new predicate used for the runtime computations.

Let us now explain the partial evaluation in more detail. Basically, we do a standard bottom-up fixpoint computation, but we work now with conditional facts. So we have a set COND of conditional facts which will increase until a fixpoint is reached. An important invariance is that COND will never contain two different conditional facts with the same predicate p in the body. In this way, we can translate facts produced later at runtime (e.g. $\mathsf{p}(\mathsf{d})$) uniquely back to facts of the original program.

We start with the following facts COND (if we want to partially evaluate our meta-interpreter):

- $\mathsf{db}\big(\mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n)\big) \leftarrow \mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n)$ for every EDB-predicate p. Actually, if there are certain small lookup-tables which seldom change, we might be allowed to compile them into our program. In this case we would have a conditional fact $\mathsf{db}\big(\mathsf{p}(\mathsf{c}_1, \ldots, \mathsf{c}_n)\big) \leftarrow \mathsf{true}$ for every row $(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ in the lookup-table.
- $\mathsf{query}(\mathsf{Query}) \leftarrow \mathsf{true}$ for the given query literal Query. If we want to run the query repeatedly with different constants, we can replace them by variables $\mathsf{X}_1, \ldots, \mathsf{X}_n$ and start with $\mathsf{query}(\mathsf{Query}) \leftarrow \mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n)$ instead. When the constants are known at runtime, we would add the corresponding p-fact.
- $\mathsf{rule}\big(\mathsf{A}, [\mathsf{B}_1, \ldots, \mathsf{B}_m]\big) \leftarrow \mathsf{true}$ for each rule $\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_m$ in the given input program.

In addition we have a set PROG of program rules which are the result of the partial evaluation. The set PROG starts out empty.

Now let $\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_m$ be a rule of our meta-interpreter (or whatever program we want to partially evaluate). We choose conditional facts $\mathsf{B}'_i \leftarrow \mathsf{C}_i$ from COND (but with fresh variables) such that there is an mgu θ of $(B_1, \ldots, \mathsf{B}_m)$ and $(\mathsf{B}'_1, \ldots, \mathsf{B}'_m)$. Then the result of the unfolding with respect to the given conditional facts is

$$\mathsf{A}\theta \leftarrow \mathsf{C}_1\theta \wedge \cdots \wedge \mathsf{C}_m\theta.$$

Now the body is already in the right form, but we want to encode also the head via a conditional fact. Let $\mathsf{Y}_1, \ldots, \mathsf{Y}_n$ be those variables which appear in $\mathsf{A}\theta$ and in at least one of the $\mathsf{C}_i\theta$. Then we search COND for a conditional fact of the form $\mathsf{A}\theta \leftarrow \mathsf{p}(\mathsf{Y}_1, \ldots, \mathsf{Y}_n)$ (with any predicate p and the variables possibly renamed). If there is none, we insert this conditional fact with a new predicate p

into COND. Finally, we add the rule $\mathsf{p}(\mathsf{Y}_1,\ldots,\mathsf{Y}_n) \leftarrow \mathsf{C}_1\theta \wedge \cdots \wedge \mathsf{C}_m\theta$ to PROG. Of course, duplicate elimination is needed here: We normalize the variables in such a way that we do not get two rules in COND or PROG which differ only in a renaming of variables.

Some body literals (like the call to append) can already be evaluated fully at compile-time, so there is no need for a matching fact pattern.

When a fixpoint is reached, PROG is the result of the partial evaluation. Each fact derivable from PROG can be translated back into the syntax of the original program by a unique rule from COND.

In the special case of the meta-interpreter, we can guarantee that partial evaluation terminates under the above conditions (all recursions other than tail-recursions are evaluated via call, the input program contains no structured terms, program and database are finite). We also can handle structured terms at least when we move them into the conditional fact bodies which are evaluated at runtime. More research is needed for deciding which function symbols can in general be evaluated during partial evaluation.

5 Conclusions

SQL-3 contains recursion, and current applications like web queries really need it. The main techniques for evaluating recursion are magic sets (with many variants) and SLD-resolution (used in Prolog). In this paper, we have clarified the differences between these two techniques. We have shown that SLD-resolution is often advantageous, and that SLD-resolution can be evaluated in a set-oriented fashion using database techniques. A first prototype implementation of the transformation is available from `http://www2.sis.pitt.edu/~sbrass/sldmagic/`.

It seems that for future performance improvements, we have to look more at the internal data structures. Especially, we want to avoid copying variable values. Our goal is to reach the performance of Prolog systems. This also needs a powerful program analysis to avoid duplicate eliminations.

For simplicity, we have considered only negation-free programs. Adding stratified negation is not difficult, although some care has to be taken to make the output stratified. We are currently working on using our ideas from [BD99,ZBF97] to handle general negation.

References

[ABS00] S. Abiteboul, P. Bunemann, D. Suciu (eds.): *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[BD99] S. Brass, J. Dix: Semantics of (disjunctive) logic programs based on partial evaluation. *The Journal of Logic Programming 40 (1999)*, 1–46.

[BR91] C. Beeri, R. Ramakrishnan: On the power of magic. *The Journal of Logic Programming 10 (1991)*, 255–299.

[Bra95] S. Brass: Magic sets vs. SLD-resolution. In J. Eder, L. A. Kalinichenko (eds.), *Advances in Databases and Information Systems (ADBIS'95)*, 185–203, Springer, 1995.

[Bry90] F. Bry: Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering 5 (1990)*, 289–312.

[GM92] A. Gupta, I. S. Mumick: Magic-sets transformation in nonrecursive systems. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'92)*, 354–367, 1992.

[HLLS97] R. Himmeröder, G. Lausen, B. Ludäscher, C. Schlepphorst: On a declarative semantics for web queries. In *Fifth International Conference on Deductive and Object-Oriented Databases (DOOD'97)*, 1997.

[KRS90] D. B. Kemp, K. Ramamohanarao, Z. Somogyi: Right-, left- and multi-linear rule transformations that maintain context information. In D. McLeod, R. Sacks-Davis, H. Schek (eds.), *Proc. Very Large Data Bases, 16th Int. Conf. (VLDB'90)*, 380–391, Morgan Kaufmann Publishers, 1990.

[KS95] D. Konopnicki, O. Shmueli: W3QS: A query system for the world-wide web. In U. Dayal, P. M. D. Gray, S. Nishio (eds.), *Proc. of the 21st Int. Conf. on Very Large Data Bases, (VLDB'95)*, 54–65, Morgan Kaufmann, 1995.

[MFPR90] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan: Magic conditions. In *Proc. of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, 314–330, 1990.

[MMM97] A. O. Mendelzon, G. Mihaila, T. Milo: Querying the world wide web. *Journal of Digital Libraries 1 (1997)*, 68–88.

[NRSU89] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, J. D. Ullman: Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 235–242, 1989.

[Ros91] 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)*, 92–101, 1991.

[RS91] R. Ramakrishnan, S. Sudarshan: Top-down vs. bottom-up revisited. In V. Saraswat, K. Ueda (eds.), *Proc. of the 1991 Int. Symposium on Logic Programming*, 321–336, MIT Press, 1991.

[RSSS94] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri: The CORAL deductive system. *The VLDB Journal 3 (1994)*, 161–210.

[SR92] D. Srivastava, R. Ramakrishnan: Pushing constraint selections. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'92)*, 301–315, 1992.

[SR93] S. Sudarshan, R. Ramakrishnan: Optimizations of bottom-up evaluation with non-ground terms. In D. Miller (ed.), *Proceedings of the International Logic Programming Symposium (ILPS'93)*, 557–574, MIT Press, 1993.

[SSW94] K. Sagonas, T. Swift, D. S. Warren: XSB as an efficient deductive database engine. In R. T. Snodgrass, M. Winslett (eds.), *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, 442–453, 1994.

[Ull89a] 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)*, 140–149, 1989.

[Ull89b] J. D. Ullman: *Principles of Database and Knowledge-Base Systems, Vol. 2.* Computer Science Press, 1989.

[ZBF97] U. Zukowski, S. Brass, B. Freitag: Improving the alternating fixpoint: The transformation approach. In A. Nerode (ed.), *Proc. of the 4th Int. Conf. on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, 40–59, LNAI 1265, Springer, 1997.