ACM Transactions on Database Systems, Vol. 21, No. 2, June 1996, pp.208–237

Tail Recursion Elimination in Deductive Databases

Kenneth A. Ross *
Columbia University
kar@cs.columbia.edu

Abstract

We consider an optimization technique for deductive and relational databases. The optimization technique is an extension of the magic templates rewriting, and it can improve the performance of query evaluation by not materializing the extension of intermediate views. Standard relational techniques, such as unfolding embedded view definitions, do not apply to recursively defined views, and so alternative techniques are necessary. We demonstrate the correctness of our rewriting. We define a class of "nonrepeating" view definitions, and show that for certain queries our rewriting performs at least as well as magic templates on nonrepeating views, and often much better. A syntactically recognizable property, called "weak right-linearity," is proposed. Weak right-linearity is a sufficient condition for nonrepetition, and is more general than right-linearity. Our technique gives the same benefits as right-linear evaluation of right-linear views, while applying to a significantly more general class of views.

1 Introduction

Declarative systems such as relational and deductive databases try to make posing queries simple by letting the users specify *what* they want to compute rather than *how* to compute it. This strategy leads to databases in which it is relatively easy to pose queries. However, since the user does not supply information on how to answer the query, the burden of deciding how to evaluate the query falls on the system.

Thus declarative systems must have powerful optimization techniques available. In large databases they could make the difference between a query being feasible or infeasible. For nonrecursive queries many well-known optimizations for relational databases can be applied. Several techniques for optimizing recursive queries have been recently proposed, including "magic-sets," "right-linear evaluation," "counting methods" and so on. (See [25] for a comprehensive discussion of these and other strategies.)

In this paper we propose an optimization technique that integrates magic sets and a form of tail-recursion elimination significantly more general than right-linear evaluation. By

*This research was performed while the author was at Stanford University, and while visiting the IBM T. J. Watson Research Laboratory. The research conducted at Stanford University was supported under NSF grant IRI-87-22886, a grant from IBM Corporation, and AFOSR contract 88-0266. A preliminary version of this paper appeared at the 1991 conference on Principles of Database Systems [18].

not representing intermediate results of a bottom-up computation, substantial savings over magic sets alone can be realized.

To motivate our technique, consider the following SQL views.

```
Create View V As
Select *
From R, S, W
Where R.A = S.A And S.B = W.B
```

```
Create View W As
Select *
From T, U
Where T.C = U.C
```

View `W` represents a natural join of `T` and `U`, and view `V` represents the natural join of `R` and `S` with `W`. Let us imagine that we can process joins in a pipelined fashion from left to right. In other words, if we want to compute R⋈S⋈T, we can pass the result tuples from R⋈S directly to be joined with `T` without representing the join result R⋈S explicitly in persistent storage. Now suppose that we had a query

```
Select *
From V
Where V.D =7
```

where `D` is an attribute of `R`. One way to execute this query would be as follows: First, apply the selection condition to `R`, and then join it with `S`. Pass the resulting `B` attribute column into a subprocess that computes the corresponding part of the view `W`, storing the result in a relation `W'`. Finally, join the R⋈S result with `W'`.

The problem with the plan above is that it materializes an intermediate result `W'`, probably on persistent storage. This overhead could have been avoided had we processed the query as a join of the four base relations, rather than by starting a separate process for the intermediate view `W`. In a relational system, one can achieve this saving by applying a process of *rewriting*, or *unfolding*. For example, using various relational equivalences, we could have determined that the following view definition is equivalent to the definition of `V` above.

```
Create View V As
Select *
From R, S, T, U
Where R.A = S.A And S.B = T.B And T.C = U.C
```

The advantage of this unfolded definition is that it has no intermediate views to worry about. As long as a view `W` is mentioned only once in the definition of a view `V`, such unfolding can only improve the expected performance of the required joins.

For relational view definitions, an unfolding into a new view definition over just the base relations is usually possible, so that no intermediate view relations need to be materialized. (Exceptions include `Select Distinct` subviews and subviews involving aggregation.) However, for *recursive* view definitions, one cannot always fully unfold a view definition. Consider the following SQL-like definition of a transitive-closure view.

```
Create View TC(Begin,End) As
       Select Source, Destination
       From Link
Union
```

```
Select Source, End
From Link, TC
Where Link.Destination = TC.Begin
```

The problem is that the view `TC` is defined in terms of itself. No matter how much unfolding we do, there will always be a `TC` relation mentioned in the `From` clause of the unfolded view.

Our goal is to avoid materializing intermediate views. For recursive view definitions, unfolding is not a solution, and so we look for other techniques. The new techniques that we develop will also apply to nonrecursive views, and so provide an alternative to explicit unfolding.

Since most of the recent work on optimizing recursive view definitions has taken place in the deductive database research community, we choose to present this paper in the context of a deductive database, where views are written using rules. However, the choice of syntax is not crucial; our results would apply equally well to an SQL-like language, with or without recursion. To see what can go wrong without a tail-recursive evaluation, consider the following example.

Example 1.1: Let P be the set of rules

$$\begin{array}{l} p(X,Z) \leftarrow e(X,Y), p(Y,Z) \\ p(100,X) \leftarrow t(X) \end{array}$$

where e and t are base relations. To make the example more concrete, suppose that $e(X,Y)$ holds when there is a way to get from town X to town Y in some remote region. We number the towns $1,\ldots,100$. $t(X)$ holds when item X is available in the region's capital, which is town number 100. Items are numbered $1\ldots,1000$. Then $p(X,Z)$ holds if it is possible to get from town X to a town that has item Z. (One could imagine other rules like the second rule, for different commercial centers, but for simplicity we do not consider additional rules here.)

Suppose that the t relation is given by $\{t(1), t(2), \cdots, t(1000)\}$, and the e relation is given by $\{e(1,2)$, $e(2,3)$, $\ldots e(n-1,n)$, $e(n,1)\}$. In other words, all items are available in the capital, and the towns are located at various points on a single cyclic one-way road. Suppose our query is ?-$p(1,Z)$, i.e., "Tell me what's available starting from town 1."

Standard techniques such as magic templates would materialize every tuple of the form $p(i,j)$ where $1 \leq i \leq 100$ and $1 \leq j \leq 1000$, thus storing $100,000$ tuples in order to answer the query. It is not necessary to store that many tuples if we apply a tail-recursive evaluation. As we shall see, our tail-recursive evaluation does not materialize $p(i,j)$ when $i > 1$. $p(1,j)$ is materialized for $1 \leq j \leq 1000$. Some additional tuples for some newly-introduced relations may be materialized, but for this example the number of such tuples would be just 100. The total number of materialized tuples is two orders of magnitude smaller than that for the standard approach, and thus represents a significant improvement in performance. □

Magic-sets [2, 3] is a general technique that can, in principle, be applied to any (recursive or nonrecursive) deductive database. By passing binding information from the query itself into the rule evaluation, magic sets restricts the computation to tuples that are in some sense relevant. Magic templates [16] is an extension of magic-sets to handle nonground tuples. Magic-sets and magic-templates materialize portions of all intermediate views, as described in Example 1.1, and so suffer from the problems mentioned above.

Other techniques, such as right-linear evaluation [14, 15] give better performance than magic sets for restricted classes of programs. Common programs such as transitive closures (including Example 1.1) fall into the class of right-linear programs. The right-linear optimization can be thought of as a limited form of tail-recursion elimination. In this paper we propose a form of tail-recursion elimination that applies to a larger class of programs.

Our techniques are motivated by considering a top-down, Prolog-style evaluation method without memoing that does not fully compute "intermediate" predicates. In some cases, Prolog-style evaluation effectively applies tail-recursion elimination. It is this tail-recursion elimination that is exploited for right-linear programs in [14, 15]. We use this observation about top-down evaluation to motivate a bottom-up formalization of tail-recursion elimination. This formalization improves on top-down evaluation in that it remains efficient even when top-down evaluation does not terminate.

The main idea is to introduce a new relation *query*. $query(S, A)$ will hold if S is a required subquery, and A is where the resulting answers should be placed. So for example, $query(p(X), p(X))$ means "pose the query $p(X)$ and store the answers in p" while $query(p(X), q(X))$ means "pose the query $p(X)$ and store the answers in q." By rewriting the rules using the *query* relation, we will be able to avoid unnecessarily storing answers to intermediate subqueries. For Example 1.1, we may derive a tuple $query(p(100, Z), p(1, Z))$ during the evaluation of the rewritten program. This tuple ensures that answers to the subquery ?-$p(100, Z)$ are not explicitly materialized, but are instead passed directly as answers to the original query ?-$p(1, Z)$.

Our rewriting augments magic-templates with a mechanism based on the *query* relation, to take advantage of tail recursion. We define the class of "nonrepeating" programs and show that for nonrepeating programs the augmented version of magic templates does no worse than ordinary magic templates, while it often does much better. We define a sufficient syntactic condition for nonrepetition, which we call "weak right linearity." The class of weakly right linear programs properly includes the class of right-linear programs. Where there is no tail recursion to eliminate, our rewriting reduces to the standard magic-templates rewriting. Thus we are able to evaluate the tail recursive portion of a program using the more efficient technique, while using magic templates for the remainder of the program.

2 Preliminaries

Informally, a *program* is a set of rules defining one or more views. We consider normal programs with function symbols [11] as presented formally below.

Definition 2.1: A *term* is defined inductively as follows:

- A variable or a constant symbol is a term.
- If f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms then $f(t_1, \cdots, t_n)$ is a term.

If p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms then $p(t_1, \cdots, t_n)$ is an *atom*. A *literal* is either an atom or a negated atom. When we write an atom $p(\vec{X})$ it is understood that $\vec{X}$ is a vector of terms, not necessarily variables. □

Definition 2.2: A *normal rule* is a sentence of the form

$$A \leftarrow L_1, \ldots, L_n$$

where A is an atom, and $L_1, \ldots, L_n$ are literals. We refer to A as the *head* of the rule and $L_1, \ldots, L_n$ as the *body* of the rule. Each L_i is a *subgoal* of the rule. All variables are assumed to be universally quantified at the front of the clause, and the commas in the body denote conjunction. If the body of a rule is empty then we may refer to the rule as a *fact*, and omit the "$\leftarrow$" symbol.

A *normal program* is a finite set of normal rules. A *Horn rule* is one with no negative subgoals, and a *Horn program* is one with only Horn rules. A *Datalog* program is a function-free Horn program. □

Logical variables begin with a capital letter; constants, functions, and predicates begin with a lowercase letter. The word *ground* is used as a synonym for "variable-free."

If a predicate is defined only by facts, then we say that the predicate is an *extensional database* (EDB) predicate; otherwise the predicate is an *intensional database* (IDB) predicate.

We consider only Horn programs in this paper. The techniques developed in this paper could be extended to programs with negation by combining with the techniques of [9, 12, 17, 19].

HiLog

It will be convenient in our exposition to use HiLog notation for some meta-predicates [5]. HiLog allows one to have atoms as terms in other atoms. For example, we might write $magic(h(X))$ where h is a predicate symbol rather than a function symbol. The use of HiLog in the present context is not essential. We could rewrite $magic(h(X))$ as $m\_h(X)$, and rewrite $magic(A)$ in terms of several cases for A having various predicate names. However, HiLog enables a more concise and clear presentation.

2.1 SLD Resolution

We present a definition of SLD-resolution from [11], specialized to use a left-to-right "computation rule."

Definition 2.3: A goal is a (negated) conjunction of atoms, written $\leftarrow A_1, \cdots, A_n$. □

Definition 2.4: (SLD-tree) Let P be a program and G a goal. The *SLD-tree* for G with respect to P is defined as follows:

- Each node of the tree is a (possibly empty) goal, together with a computed substitution.
- The root node is G, with empty computed substitution.
- Let $G' = \leftarrow A_1, \cdots, A_n$ $(n \geq 1)$ be a node in the tree. Then G' has a child for each rule r such that the head of r and A_1 are unifiable. Let $H \leftarrow B_1, \cdots, B_m$ be a variant of r using new variables. The child node G'' is

$$\leftarrow (B_1, \cdots, B_m, A_2, \cdots, A_n)\theta$$

where θ is the most general unifier of A_1 and H. The computed substitution at G'' is $\phi\theta$ where ϕ is the computed substitution at G'. We say that each $B_i\theta$ in G'' is a *rule-child* of A_1 in G'. For $i > 1$, if C_i in G_i has rule-child A_i in G', then $A_i\theta$ in G'' is also a rule-child of C_i in G_i.

- Empty nodes have no descendants.

We shall assume without loss of generality that the most general unifiers chosen are idempotent. We say C in G_C is a *rule-ancestor* of D in G_D if there is a sequence $C_0, \ldots, C_k$ of atoms, and a sequence of goals $G_0, \ldots, G_k$ such that $C_0 = C$, $G_0 = G_C$, $C_k = D$, $G_k = G_D$, and for $i = 1, \ldots, k$, C_{i-1} in G_{i-1} is a rule-child of C_i in G_i. Similarly, we can define *rule-descendant.*

A *subrefutation* for the leftmost atom A in a goal G' is a shortest (downward) path in the SLD-tree from G' to a descendant G'' of G' that contains no rule-descendants of A. We associate with the subrefutation the computed substitution at G''.

SLD-resolution is the process of finding the computed substitutions at all empty nodes of an SLD-tree. □

Note the difference between ancestors in the SLD-tree (a relationship between goals) and rule-ancestors (a relationship between atoms in goals).

SLD-resolution is an example of a *top-down* query-processing strategy, also sometimes referred to as *backward chaining.* One starts with the goal to be proved, and repeatedly substitutes the body of a rule for an instance of the head in the goal to be proved. The query succeeds when all the literals in the goal have been proved, i.e., when the resulting goal is empty.

2.2 Magic Templates

An alternative to a top-down query processing strategy is a *bottom-up* strategy, also sometimes referred to as *forward chaining.* One starts with all the information one has, and uses the rules to generate new information. When all of the subgoals of a rule are satisfied, an appropriate instance of the head predicate is inferred. The new information can then be used on the next iteration to generate more information, and so on. The bottom-up evaluation process in which a rule is fired only for instances of the body in which at least one subgoal was newly derived on the previous iteration is called "semi-naive evaluation."

There are a number of reasons why one might prefer a bottom-up strategy to a top-down strategy, and the reader is referred to [25] for a discussion of this issue. However, a bottom-up evaluation of the original rules is likely to be inefficient because it does not take the query into account. A bottom up evaluation of the original program will materialize all predicates defined in the rules, while a query may need to access only a small fraction of the database in order to be answered. Top-down methods use the information in the query by passing variable bindings from the query into intermediate goals.

In order to restrict the bottom-up evaluation of a set of rules, one first rewrites the rules so that they take the query into account. In the rewritten program, only rules that are relevant to the query are ever fired. One rewriting technique developed for this purpose is the "magic templates" rewriting of Ramakrishnan [16]. We now review the magic templates rewriting.

In our presentation we shall assume that subgoals are evaluated from left to right. In the terminology of [16], we use a "full sideways information passing strategy" with a left-to-right order of subgoals.

While the original magic templates transformation introduces a new predicate m_p for each predicate p, we shall use the notation $magic(p(\vec{X}))$ rather than $m\_p(\vec{X})$. Essentially, we view *magic* as a meta-predicate, as in HiLog.

Definition 2.5: (Magic Templates.) For each rule

$$h(\vec{X}) \leftarrow p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n)$$

defining an IDB predicate h in the original program, we generate some rewritten rules.

1. For each IDB predicate p_i in the body we generate the rule

$$magic(p_i(\vec{X}_i)) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1}).$$

2. We also generate the rule

$$h(\vec{X}) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n).$$

Note that EDB predicates are not affected by the rewriting. Finally, if the query is ?-$q(\vec{Y})$, then we add the fact

$$magic(q(\vec{Y}))$$

as a "seed fact." □

Example 2.1: Let P be

$$\begin{array}{l} p(X,Z) \leftarrow e(X,Y), p(Y,Z) \\ p(n,X) \leftarrow t(X) \end{array}$$

where e and t are EDB predicates. This program is a slightly more general version of the program from Example 1.1. Suppose our query is ?-$p(1,Z)$. Then the rewritten program would be

$$\begin{array}{l} magic(p(Y,Z)) \leftarrow magic(p(X,Z)), e(X,Y) \\ p(X,Z) \leftarrow magic(p(X,Z)), e(X,Y), p(Y,Z) \\ p(n,X) \leftarrow magic(p(n,X)), t(X) \\ magic(p(1,Z)) \end{array}$$

We now consider the performance of the rewritten rules on two sample databases. In both cases, suppose that the t relation is given by $\{t(1), t(2), \cdots, t(m)\}$.

(a) Suppose the e relation is given by $\{e(1,n), e(2,n), \ldots e(n,n)\}$. Evaluating the *rewritten* rules gives us $\Theta(m)$ tuples for p, namely $p(1,j)$ and $p(n,j)$ for $1 \leq j \leq m$. In addition, we have a constant number of *magic* tuples. (Evaluating the *original* rules bottom-up would give us $\Theta(nm)$ tuples for p, namely $p(i,j)$ for $1 \leq i \leq n$, $1 \leq j \leq m$.)

(b) Suppose the e relation is given by $\{e(1,2),\ e(2,3),\ \ldots e(n-1,n)\}$. Evaluating the rewritten rules gives us $\Theta(nm)$ tuples for p, namely $p(i,j)$ for $1 \leq i \leq n$, $1 \leq j \leq m$. In addition, $\Theta(n)$ *magic* tuples will be generated. (As we shall see in Section 4, we can improve on this complexity for programs like the one above by using a rewriting strategy incorporating tail recursion elimination.)

□

One point to note about the magic templates rewriting is that it generates *nonground tuples*. For example, the fact $magic(p(1, Z))$ contains a variable Z. Nonground tuples can make bottom-up evaluation of rules more costly because, in general, one needs to perform unification of atoms rather than matching. In some cases one can eliminate the nonground tuples. In the example above, one could replace $magic(p(Y, Z))$ by $magic(p'(Y))$ throughout, where p' is a new unary predicate, without changing the essential behavior of the program.

In other cases, the nonground tuples are essential. An example is a program containing a fact $p(X, X)$ that states that p is true if its two arguments are equal.

2.3 Tail Recursion

Consider the following C code fragment for a recursive binary-search procedure, based on one from [22].

```
int search(int lo, int hi, int val) {
         int k;
         if (lo > hi) return 0;
         k = (lo + hi)/2;
         if (val == List[k]) return 1;
         else if (val < List[k]) return search(lo,k-1,val);
         else if (val > List[k]) return search(k+1,hi,val);
}
```

This procedure searches for the value `val` in the global array `List`, whose elements are in increasing order.

The procedure `search` is *tail recursive* because every recursive call must be the last statement executed in the procedure. A compiler that notices that a procedure is tail recursive can optimize this code by converting the recursive call into iterative code. The optimized code for the procedure above might look like this:

```
int search(int lo, int hi, int val) {
         int k;
L:       if (lo > hi) return 0;
         k = (lo + hi)/2;
         if (val == List[k]) return 1;
         else if (val < List[k]) hi=k-1;
         else if (val > List[k]) lo=k+1;
         goto L;
}
```

The main reason why the iterative code performs better than the recursive code is that new activation records do not have to be stored on the stack for each recursive call. In the iterative code, control returns to the initial call as soon as an element is either found or known to be absent from the array. In the recursive code, once an element is found (or found to be absent) control passes one level up to the calling activation of `search`, returning a boolean value. Control then passes level by level up the activation stack until the initial call is reached. At each level, the intermediate result indicating whether the recursive call returned `1` or `0` has to be stored.

In this paper, we address an analogous form of tail recursion in recursive database rules. In our context, the intermediate results (corresponding to the boolean values in the program above) are not just simple variables, but whole relations. Hence it is particularly important that these recursive calls be optimized so that such relations are not stored or copied more often than necessary.

3 Top-Down Can Beat Bottom-Up

You may be startled by the title of this section, especially if you have seen the paper "Bottom-up Beats Top-down for DATALOG," [24]. However, there is no inconsistency here: the model of top-down computation in [24] represents intermediate answers, whereas the one used in this section does not.

There are cases where a top-down method like SLD-resolution can "beat" bottom-up with magic sets (with or without duplicate elimination) for the following reason: Top down returns the binding relation at the leaves of derivation trees directly, while magic sets computes all the intermediate relations. Consider the following example.

Example 3.1: Let P be the program of Example 2.1(b), i.e.,

$$\begin{array}{l} p(X,Z) \leftarrow e(X,Y), p(Y,Z) \\ p(n,X) \leftarrow t(X) \\ e(1,2) \\ \vdots \\ e(n-1,n) \\ t(1) \\ \vdots \\ t(m) \end{array}$$

SLD-resolution would construct the following derivation tree for the query ?-$p(1,X)$.

$$\begin{array}{ccc} & p(1,X) & \\ & | & \\ & e(1,Y), p(Y,X) & \\ & | & \\ & p(2,X) & \\ & | & \\ & \vdots & \\ & | & \\ & p(n,X) & \\ & | & \\ & t(X) & \\ / & | & \backslash \\ \{X=1\} & \cdots & \{X=m\} \end{array}$$

SLD-resolution effectively uses tail-recursion elimination by returning the answers, i.e., bindings for X, directly from the leaves of the SLD-tree. Without this kind of tail-recursion elimination, the answers would be percolated up the tree through the intermediate nodes until the original query was reached, as in the top-down method of [24].

To find all the answers the amount of work done by SLD-resolution is $\Theta(m+n)$. As we have seen, the magic templates method would compute all the intermediate relations $p(i, X)$ for $i = 2, \ldots n$, each of which is of size m. Magic templates computes $\Theta(mn)$ tuples, and is inferior to SLD-resolution on this example. □

The complexity of the top-down evaluation in the above example is dependent both upon the query and upon the e relation.

Example 3.2: Consider the program from Example 3.1.

(a) If the query was ?-$p(X, Y)$ rather than ?-$p(1, X)$ then the SLD-tree would have size $\Theta(mn + n^2)$ while magic sets would have time complexity $\Theta(mn)$.

(b) If we added the $n - 2$ extra tuples

$$e(1,3), e(2,4), \ldots, e(n-2, n)$$

to the program above, then the SLD-tree for the original query ?-$p(1, X)$ would still be finite, but will have mF_n leaves[1] and hence be of size exponential in n. By contrast, magic sets (with duplicate elimination) would still take $\Theta(mn)$ space and time.

(c) If we added the tuple $e(n, 1)$ we would create a cycle in the e relation. SLD-resolution would not terminate, while magic sets would still take $\Theta(mn)$ space and time. □

Items (b) and (c) from Example 3.2 suggest that SLD-resolution has very limited applicability. However, we will be able to formalize the tail-recursion optimization suggested in Example 3.1 within a bottom-up framework in such a way that duplicate elimination will eliminate these drawbacks.

By integrating tail-recursion optimization with magic templates, we shall obtain the additional benefits of tail-recursion elimination when tail-recursion elimination is beneficial, while retaining the more general benefits of magic templates.

4 Bottom-Up Evaluation

We describe in this section a method to incorporate the tail-recursive aspect of SLD-resolution in a bottom-up framework to take advantage of the tail recursion present in the program. This transformation will allow the tail recursive evaluation of a portion of the program while the remainder of the program is evaluated according to standard magic template techniques. Bry's translation of SLD-resolution [4] explicitly represents the intermediate tuples, and hence does not take advantage of this potential optimization.

[1] Here, F_n is the nth Fibonacci number, which for large n is well approximated by $\alpha\phi^n$ where $\phi \approx 1.618$ and $\alpha \approx 0.447$.

4.1 The Tail-Recursive Rewriting

The first step of the rewriting is to decide which predicates are to be evaluated tail-recursively. In Example 3.1 we would choose p, for example. We label the chosen predicates as "right-recursive." Note that it is not necessary that *every* occurrence of a right-recursive predicate be rightmost in its rule. However, only those occurrences that are rightmost will have the tail-recursion optimization applied. Thus "right-recursive" refers to a predicate which will be evaluated tail-recursively when it appears rightmost in a rule. We shall say that a particular atom is "right-recursive" if it has a right-recursive predicate, and "tail recursive" if it is both right-recursive and rightmost in its rule.

Predicates that are not labeled as right-recursive will be evaluated essentially as before, using magic templates.

The right-recursive predicates will not have corresponding magic tuples. Instead, they shall have corresponding tuples in a binary relation we call *query*. The tuple $query(P, A)$ will mean that we are trying to solve the atom P, and that any answer substitutions found should generate an answer tuple for A, but not necessarily for P. (As with *magic*, we view *query* as a HiLog meta-predicate.) Recall that we allow non-ground tuples, so we could have tuples like $query(p(a, X, Y), q(X))$.

The *query* tuples are the key to the transformation. If one thinks in terms of an SLD-tree, each tuple $query(S, A)$ records a direct link from a subgoal S to a rule-ancestor A of that subgoal. Answers for S and for all intermediate nodes between S and A are not explicitly stored. Solutions to the subgoal S are passed as answers directly to A.

Definition 4.1: (Magic Templates with Right-Recursion.) For each rule

$$h(\vec{X}) \leftarrow p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n)$$

defining an IDB predicate h in the original program, we generate some rewritten rules.

1. For each IDB predicate p_i in the body, if p_i is right-recursive then we have two cases:

   (a) If p_i is rightmost in the rule body (i.e., $i = n$), and h is also right-recursive, then we generate the rewritten rule

   $$query(p_i(\vec{X}_i), A) \leftarrow query(h(\vec{X}), A), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1}).$$

   (b) Otherwise, we generate the rule

   $$query(p_i(\vec{X}_i), p_i(\vec{X}_i)) \leftarrow query(h(\vec{X}), A), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1})$$

   if h is right-recursive, or the following rule if h is not:

   $$query(p_i(\vec{X}_i), p_i(\vec{X}_i)) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1}).$$

   If p_i is not right-recursive, then we generate the rule

   $$magic(p_i(\vec{X}_i)) \leftarrow query(h(\vec{X}), A), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1}).$$

   if h is right-recursive, or the following standard magic templates rule if h is not:

   $$magic(p_i(\vec{X}_i)) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_{i-1}(\vec{X}_{i-1})$$

2. If h is right-recursive then we have two cases:

   (a) If the rightmost subgoal of the rule is not right-recursive, then generate the rule

   $$A \leftarrow query(h(\vec{X}), A), p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n).$$

   (b) If the rightmost subgoal is right-recursive, then no rewritten rule is generated.

   If h is not right-recursive, then generate the standard magic templates rule

   $$h(\vec{X}) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n).$$

Finally, if the query is $q(\vec{Y})$, then we add the fact

$$magic(q(\vec{Y}))$$

as a seed fact if q is not right-recursive, or

$$query(q(\vec{Y}), q(\vec{Y}))$$

if q is right-recursive. □

For rules that do not contain right-recursive predicates, the rewriting above gives the same results as the standard magic templates rewriting of Section 2.2.

We call the the transformation above "magic templates with right-recursion." We denote the result of applying the rewriting to a program P by $MTRR(P)$. Note that item 2(a) generates a HiLog rule, not a normal rule.

Let us see how our rewriting handles the program of Example 3.1.

Example 4.1: We label p as right-recursive. The rewritten version of P from Example 3.1 for the query ?-$p(1, Z)$ is

$$\begin{array}{l} query(p(Y,Z),A) \leftarrow query(p(X,Z),A), e(X,Y) \\ A \leftarrow query(p(n,X),A), t(X) \\ query(p(1,Z), p(1,Z)) \end{array}$$

Given the EDB of Example 3.1, the tuples generated are, in order

$$\begin{array}{l} \{query(p(1,Z),p(1,Z))\} \\ \{query(p(2,Z),p(1,Z))\} \\ \{query(p(3,Z),p(1,Z))\} \\ \quad \vdots \\ \{query(p(n,Z),p(1,Z))\} \\ \{p(1,1), p(1,2), \cdots, p(1,m)\} \end{array}$$

Thus we have reduced the cost of semi-naive bottom-up evaluation to $\Theta(n+m)$ time and space.

Consider also how the rewritten program performs for the modified query and for the modified e relation of Example 3.2. If the query is $p(X, Y)$ as in part (a), then the complexity is $\Theta(mn + n^2)$ as for SLD-resolution, since there will be $\Theta(n^2)$ tuples of the form $query(p(i, Y), p(j, Y))$ generated, for $1 \leq j < i \leq n$. This may be worse than ordinary magic sets, which has time and space complexity $\Theta(mn)$.

On the other hand, if we ask the original query once the additional e tuples are added as in part (b), then as long as duplicate elimination is applied, the complexity is still $\Theta(m+n)$, and not exponential in n like SLD-resolution. In fact, we still get $\Theta(m+n)$ complexity when the e relation has cycles (and has size $\Theta(n)$) as in part (c). $\square$

4.2 Supplementary Predicates

In Example 2.1 one can observe that the conjunction $magic(p(X, Z)), e(X, Y)$ appears in more than one rule. Hence it may be preferable to compute this join once, and refer to the computed result in each of these rules.

A technique for systematically performing such common-subexpression elimination has been proposed in [3] and extended in [20]. The basic idea is to create new relations called "supplementary relations" that hold the result of the conjunction of an initial sequence of the subgoals in the body of a rule.

The supplementary relation technique is easily applied to magic templates, and can also be extended to our rewriting that includes tail-recursion elimination:

Firstly, the arguments of the supplementary predicates for rules with head predicates that are right-recursive will have an extra argument A corresponding to the answer tuple to be generated. Also, for these rules we can omit from the supplementary relations variables in the head that do not appear further to the right in the rule body. Finally, we do not have to generate a supplementary relation for the conjunction of subgoals to the left of the rightmost subgoal if both the head and rightmost subgoal are right-recursive. The tail-recursive evaluation will mean that we won't directly generate tuples for the head predicate, and so the conjunction of all subgoals but the rightmost will be used once rather than twice.

For a version of our transformation that incorporates supplementary predicates see [18].

4.3 Nonrecursive Predicates

While our terminology describes predicates as "right-recursive," it is not necessary that such predicates actually be recursive. The optimization described above can still provide significant benefits for nonrecursive programs. For example, consider the program

$$
\begin{aligned}
p(X) &\leftarrow q(X) \\
q(X) &\leftarrow r(X) \\
q(X) &\leftarrow s(X)
\end{aligned}
$$

in which neither r nor s is recursive in p or q. For a query $p(Y)$, we can save representing the answer set twice (as p and as q) by labeling p and q as right-recursive and applying our transformation. We get the same benefit as unfolding the definition of p in terms of r and s (but not q), as discussed in Section 1.

5 Correctness

We shall show that our method is correct by comparison with SLD-resolution. In the case where there are no right-recursive predicates, our result implies the correctness of magic templates, as shown previously in [16].

We assume that a subset of the IDB predicates of the program have been labeled as "right-recursive." We make use of the concept of a "right ancestor."

Definition 5.1: (Right ancestor). Let a be a selected (i.e., leftmost) atom in a node N of the SLD-tree for an atomic query q. We associate with a a *right ancestor* that is either a itself, or a rule-ancestor of a in N.

- If a is right-recursive, and if a was introduced into a goal by resolving the rule r with a goal node G having selected atom h that is right-recursive, and this occurrence of a is rightmost in r, then the right ancestor of a is the right ancestor of h in G.
- Otherwise, the right ancestor of a is a. □

To simplify the proof below, we shall rewrite the *magic* predicates as *query* predicates, substituting $query(P, P)$ for every occurrence of $magic(P)$ in the head and body of each rule. Again, this does not change the values of any of the other tuples since *query* previously held (and is referenced) only with arguments that have right-recursive predicates. (For rule evaluation, one would prefer the original *magic* relation since it occupies less space.)

Once $MTRR(P)$ has had its *magic* atoms replaced, we denote the result by $MTRR'(P)$.

Theorem 5.1: Let P be a program and let a labeling of right-recursive predicates be given. Let ?-Q be a query, where Q is an IDB atom. Then the following correspondence holds between the SLD tree with root $\leftarrow Q$, and the bottom-up evaluation of $MTRR'(P)$ for the query Q. Let L, M and N be IDB atoms.

	SLD-tree	Evaluation of $MTRR'(P)$
(a)	There is a goal G with leftmost atom L, computed substitution θ, the right ancestor of L is M, and $N = M\theta$.	$query(L, N)$ is inferred.
(b)	There is a goal G with leftmost atom L, a sub-refutation for L with computed substitution θ, the right ancestor of L is L itself, and $N = L\theta$.	N is inferred.

Proof: See Appendix A. ▮

Corollary 5.2: (Correctness) Let P be a Horn program with right-recursive predicates, and let P' be $MTRR(P)$ for the atomic query Q. Then the bottom-up evaluation of P' correctly answers the query Q with respect to the least Herbrand model of P.

Proof: By Theorem 5.1, the soundness and completeness of SLD-resolution [1, 6, 8, 11] with respect to the least Herbrand model, and since $MTRR(P)$ and $MTRR'(P)$ are equivalent for all EDB and IDB predicates from P. ▮

Note that our correctness result does not state that the bottom-up evaluation of $MTRR(P)$ terminates. In general, with recursively applied function symbols, termination is not guaranteed in either SLD-resolution or in the corresponding bottom-up evaluation. However, every correct answer to the query will eventually be generated by the evaluation of $MTRR(P)$. In the case of function-free programs, the evaluation of $MTRR(P)$ is guaranteed to terminate.

6 Nonrepetition

While our transformation may sometimes do worse than magic templates, we can show that it does no worse for the class of nonrepeating programs.

Definition 6.1: (Nonrepetition) Let P be a program, Q a query, and T the SLD-tree for Q with respect to P. We say P is *nonrepeating* with respect to Q if for every pair of leftmost IDB atoms a and a' in nodes N and N' of T, the following condition holds: If θ and θ' are the computed answer substitutions at N and N' respectively, and h and h' are their respective right ancestors, and if a is an instance of a', then the pair $(a, h\theta)$ is an instance of $(a', h'\theta')$. $\Box$

The program of Example 3.1 is nonrepeating for the query ?-$p(1, X)$, as are its modified versions in Example 3.2 parts (b) and (c). (Note that the SLD-tree for Q with respect to P need not be finite in order for P to be nonrepeating with respect to Q.) On the other hand, for the query ?-$p(X, Y)$ as in Example 3.1 part (a), the program is not nonrepeating.

In Section 6.3 we shall show that the modified magic templates method does no worse (and often much better) than ordinary magic templates for nonrepeating programs.

6.1 Right Linearity

So far, the examples where savings over magic sets are achieved have been right linear programs, so that right-linear optimization [14, 15, 25] could have been applied.

Definition 6.2: An *adornment* for an occurrence of a predicate p is a string of "b" and "f" symbols, one for each argument of p, indicating whether the corresponding argument is called bound or free, respectively. An argument is bound if all variables in the argument are bound; otherwise the argument is free.

Given an adornment for the predicate in the head of a rule, the *induced* adornment on a predicate occurrence p in the body of a rule is the adornment in which all variables in bound arguments of the head and in subgoals to the left of p are bound, and all other variables are free. $\Box$

Definition 6.3: (Right Linearity) Let P be a program in which there is a single IDB predicate p, which is labeled as right-recursive. Let α be an adornment for p. Then P is *right-linear* with respect to α if

1. All rules have at most one occurrence of p in the body.

2. If the head adornment is α, and the recursive rules have their subgoals ordered so that the p subgoal is rightmost, then the induced adornment for the p subgoal is α.

3. For each recursive rule, each argument of p that is free according to α is the same variable in both the head and the recursive subgoal, and each of these variables appears only in these two positions. □

When rightmost subgoals of right-recursive predicates satisfy the condition of right-linearity, our transformation is essentially the same as one developed by Sagiv [21] for this special case. The idea of "carrying around" a pointer from a subquery to the "top query" was introduced in a simpler form in [10] and (independently) in [13] for right-linear programs with multiple bindings. It is noted in [25] that Prolog effectively applies tail-recursion optimization for right-linear programs.

6.2 Examples

We shall demonstrate below some examples of nonrepeating programs that are not right linear, for which tail-recursion elimination produces a significant speedup.

Example 6.1: Consider the program

$$\begin{array}{l} p(X, Z_1, Z_2) \leftarrow e(X, Y), p(Y, Z_2, Z_1) \\ p(n, Z_1, Z_2) \leftarrow t(Z_1), q(Z_2) \end{array}$$

This program is not right-linear, since the variables Z_1 and Z_2 are interchanged in the p subgoals in the first rule. It is nonrepeating (for arbitrary EDB relations e, t and q) with respect to a query on p with bound first argument and second and third arguments free. □

Example 6.2: The program

$$\begin{array}{l} p(X, t(Y), Z) \leftarrow e(X, W), p(W, Y, s(Z)) \\ p(X, Y, Z) \leftarrow f(X, Y, Z) \end{array}$$

is nonrepeating (for arbitrary e and f relations) with respect to a query on p with free third argument and first and second arguments bound. This program is not right-linear due to the function symbol around Z in the body of the first rule. □

Example 6.3: Let P be

$$\begin{array}{l} \mathit{leaf}(t(T_1, T_2), s(L)) \leftarrow \mathit{leaf}(T_1, L) \\ \mathit{leaf}(t(T_1, T_2), s(L)) \leftarrow \mathit{leaf}(T_2, L) \\ \mathit{leaf}(L, s(L)) \leftarrow \mathit{atomic}(L) \end{array}$$

Given a binary tree T_0, $\mathit{leaf}(T_0, s^i(L))$ holds for each leaf L of T_0 having depth i.[2] Consider the performance of magic templates with right-recursion for the query ?-$\mathit{leaf}(t_0, L)$, where t_0 is a complete tree of depth d having $n = 2^d - 1$ nodes and $l = 2^{d-1}$ leaves. There will be $\Theta(n)$ *query* tuples. While the size of the tuples themselves add up to give an extra factor of d to the total size, clever management of the tree structure using pointers can reduce this factor to a constant.[3] There will be l *leaf* answer tuples. Thus, semi-naive evaluation of MTRR(P) will take $\Theta(n)$ time and space.

By comparison, magic sets will generate *leaf* tuples at every level of the tree, generating a total of ld *leaf* tuples. Thus magic sets will have time and space complexity $\Theta(n \log n)$. □

[2] We assume all leaves are distinct, so that the structure is indeed a tree and not a directed acyclic graph.

[3] This same observation about pointers could also be used in the regular magic-templates construction, and so it does not affect the comparison made here.

Example 6.4: Let the relation *subtree* encode a binary tree,[4] so that $subtree(t_0, t_1, t_2)$ holds when t_1 is the left subtree of t_0 and t_2 is the right subtree of t_0. (If one subtree is missing, we substitute the constant symbol *null*.) We say that a subtree rooted at t is *balanced* if it has no descendents with exactly one child. Then the program

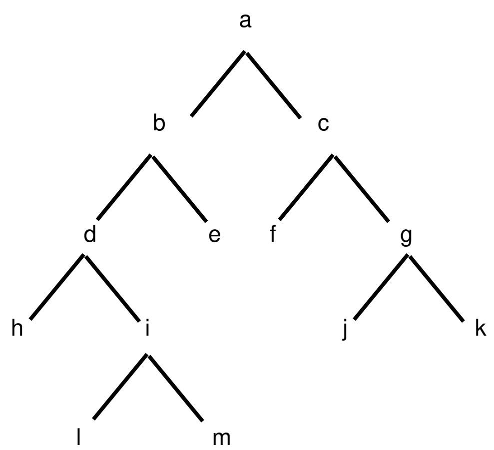
$$
\begin{aligned}
&p(R, X) \leftarrow \mathit{subtree}(R, R_1, R_2), p(R_1, Z), p(R_2, X)\\
&p(R, X) \leftarrow \mathit{atomic}(R), \mathit{values}(R, X)
\end{aligned}
$$

is nonrepeating for a query $p(a, X)$, assuming that *atomic* holds for all leaves with respect to *subtree*, and not for *null*. $p(a, X)$ succeeds if the subtree rooted at a is balanced, and has corresponding rightmost *values* X.

To see how the transformation handles right-recursive predicates appearing rightmost in places, and not rightmost in others, we shall look at this example in further detail. The transformed version of this program, labeling p as right-recursive, and assuming a query ?-$p(a, X)$ is

$$
\begin{aligned}
&\mathit{query}(p(R_1, Z), p(R_1, Z)) \leftarrow \mathit{query}(p(R, X), A), \mathit{subtree}(R, R_1, R_2)\\
&\mathit{query}(p(R_2, X), A) \leftarrow \mathit{query}(p(R, X), A), \mathit{subtree}(R, R_1, R_2), p(R_1, Z)\\
&A \leftarrow \mathit{query}(p(R, X), A), \mathit{atomic}(R), \mathit{values}(R, X)\\
&\mathit{query}(p(a, X), p(a, X))
\end{aligned}
$$

Suppose that the *subtree* relation corresponds to the following tree:

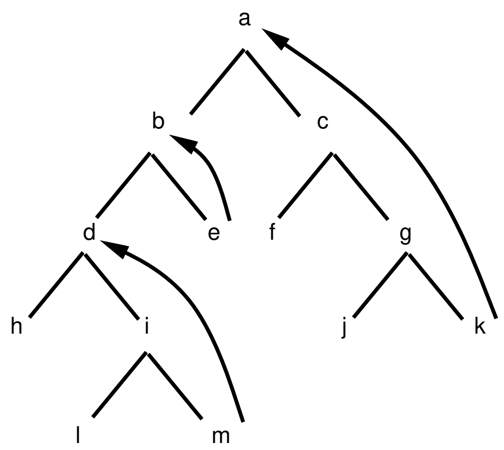


and that for each leaf L, $\mathit{values}(L, L)$ holds. The tuples generated are, in order

$$
\begin{aligned}
&\mathit{query}(p(a, X), p(a, X)),\ \mathit{query}(p(b, X), p(b, X)),\ \mathit{query}(p(d, X), p(d, X)),\\
&\mathit{query}(p(h, X), p(h, X)),\ p(h, h),\ \mathit{query}(p(i, X), p(d, X)),\\
&\mathit{query}(p(l, X), p(l, X)),\ p(l, l),\ \mathit{query}(p(m, X), p(d, X)),\ p(d, m),\\
&\mathit{query}(p(e, X), p(b, X)),\ p(b, e),\ \mathit{query}(p(c, X), p(a, X)),\\
&\mathit{query}(p(f, X), p(f, X)),\ p(f, f),\ \mathit{query}(p(g, X), p(a, X)),\\
&\mathit{query}(p(j, X), p(j, X)),\ p(j, j),\ \mathit{query}(p(k, X), p(a, X)),\ p(a, k).
\end{aligned}
$$

Intermediate answers for the nodes i, m, e, c, g, and k are not represented. Graphically, answers are passed via the arrows in the following diagram:

[4] Again, we assume no common subtrees.



□

6.3 Efficiency

The crucial feature of all the examples above for which top-down does better, is that the rightmost subgoal shares at least one unbound variable with the head. We can effectively say, "All solutions for the free variables in the rightmost subgoal are solutions for the head," *without explicitly representing the solution of the rightmost subgoal.*

Theorem 6.1: (Efficiency) Let P be a program, and let Q be a query on a predicate from P such that P is nonrepeating with respect to Q. Then evaluating the program $P_R = MTRR(P)$, resulting from the magic templates rewriting with right-recursion for P and Q, performs no more inferences than the program P_T generated by magic templates for P and Q.
Proof: By Theorem 5.1, $query(L, N)$ holds when there is a goal G with leftmost atom L, right ancestor M and $N = M\theta$, where θ is the computed substitution at G. The *magic* tuples generated by P_T are either magic tuples generated by P_R, or correspond to first arguments of *query* tuples generated by P_R. The property of nonrepetition guarantees that the extra argument A in the *query* tuples does not affect duplicate elimination in P_R compared with P_T, and hence the number of tuples generated for these predicates. Finally, every time a rule from P_R of the form

$$A \leftarrow query(h(\vec{X}), A), p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n).$$

fires, then a corresponding rule of the form

$$h(\vec{X}) \leftarrow magic(h(\vec{X})), p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n).$$

from P_T also fires. ▮

The evaluation of P_R may compute significantly fewer tuples, since it does not represent intermediate answers for right-recursive predicates. There is, of course, the overhead of having an extra argument in the *query* tuples. At worst, this would add a small constant factor to the overall cost.

In Theorem 6.1 we make the assumption that newly generated atoms are checked against the current information for variance rather than subsumption. There are some subtle effects on relative efficiency that appear when one uses subsumption checking. For a discussion of this issue, see [7, 23].

With the extension proposed above we can do better than "sip-optimal" [16]. Informally, a method is sip-optimal if

- It answers all instances of queries, and
- it generates new queries for every predicate in a rule body whose head matches a query, according to the given "sideways information passing strategy," and
- it infers only the answers and queries necessary by the above items.

Modulo the *magic* tuples, magic templates is sip-optimal [16]. Modulo the *magic* and *query* tuples, our extension is at worst sip-optimal for nonrepeating programs, by Theorem 6.1. We can do better in some cases, for example the program of Example 4.1, because not all intermediate answers need to be represented.

6.4 Weak Right Linearity

We present a syntactically recognizable sufficient condition for a linear program/query pair to be nonrepeating. We call this condition "weak right linearity," since it is satisfied by all right linear programs. The class of weakly right linear programs is more general than the class of right linear programs in three ways:

1. Multiple mutually recursive predicates are permitted.
2. Function symbols are permitted in terms containing free variables.
3. Free variables do not have to be in identical positions in both the head and the right-recursive subgoal.

Definition 6.4: (Weak right linearity) Let P be a range-restricted program, and let $p_1, \ldots, p_n$ be predicates that we label right-recursive. Suppose that at most one such p_i appears in each rule body of P, and that every rule with a right-recursive predicate in the body also has a right-recursive predicate in the head. Let Q be a query on some p_j, with adornment α_j. We say P is weakly right linear with respect to Q if, once the rules of P have their subgoals reordered so that the right-recursive predicate (if present) is rightmost,

1. Each p_i has a unique induced adornment α_i, and
2. If the rightmost subgoal of a rule is right-recursive then every free variable in the head of that rule appears in the head with no enclosing function symbols, and also appears free (and only free) in the rightmost subgoal, according to its induced adornment. □

That right linear programs are weakly right linear is straightforward from the definitions. The programs of Examples 6.1 and 6.2 are weakly right linear but not right linear. The program of Example 6.3 is not weakly right linear due to the presence of the function symbol s in the head of the right-recursive rules. Note that Example 6.3 may not be nonrepeating when the initial "tree" contains repeated subtrees, and is thus a directed acyclic graph. Since the nonrepetition of Example 6.3 depends upon the form of the query, it will be impossible to recognize that it is nonrepeating using a sufficient syntactic condition. A similar observation holds for the nonlinear program of Example 6.4, where nonrepetition depends upon the values in an EDB relation.

Lemma 6.2: Let P be a program, and Q a query. If P is weakly right linear with respect to Q then P is nonrepeating with respect to Q.

Proof: By weak right linearity, all instances of right-recursive predicates will have the same right ancestor, namely the original query, possibly instantiated in different ways. However, since none of the free arguments in the head can have been bound (again, by weak right linearity) the right ancestor will be the uninstantiated query itself, and nonrepetition follows. ∎

Our method yields the same asymptotic performance as the right-linear transformation for right-linear programs, but is more generally applicable. Unlike the right-linear transformation, our transformation introduces nonground tuples. As discussed in Section 2.2, nonground tuples may slow down query evaluation by requiring full unification rather than matching.

7 Conclusions

We have discussed the question of tail-recursion optimization, motivated by the observation that in some cases SLD-resolution can outperform magic sets by not representing intermediate answers explicitly. We have presented a modification of magic templates that takes advantage of tail-recursion elimination. We have demonstrated the correctness of this rewriting, and have shown that this method does at least as well (and often better) than magic templates for the class of nonrepeating programs. We have provided a syntactic sufficient condition for nonrepetition, namely weak right-linearity. The class of weakly-right-linear programs properly includes the class of right-linear programs.

As noted by Sagiv [21], the class of left-linear rules [15] can be generalized in a symmetric way. Left-linear optimization and mixed-linear optimization [15] can then be suitably generalized for programs with rules that commute and are either weakly right-linear or "weakly left-linear."

Acknowledgements

I would like to thank Michael Maher, Shuky Sagiv, S. Sudarshan, Jeff Ullman, and the NAIL! group at Stanford for helpful comments on this work. I am particularly grateful to Catriel Beeri and the anonymous referees, whose numerous and detailed suggestions led to a significant improvement of the presentation of this paper.

A Proof of Theorem 5.1

In what follows we assume that the rule variant used in the top-down computation uses the same variables as the rewritten rules used in the bottom-up computation. We also assume that top-down and bottom-up methods compute the same most general unifiers. These assumptions are not critical, but they simplify the proof by allowing us to ignore variable renaming.

The proof of the left-to-right correspondence is by induction on the depth of the SLD-tree. The proof of the right-to-left correspondence is by induction on the number of iterations of the bottom-up evaluation of $MTRR'(P)$.

($\Rightarrow$) The base case is straightforward: At depth 1 the SLD-tree for Q has one node, namely $\leftarrow Q$, with leftmost atom Q, the identity computed substitution and Q itself as right ancestor. In $MTRR'(P)$, $query(Q, Q)$ holds.

Let us now consider the induction step. Suppose that the implication from left to right holds for the SLD-tree up to depth n. We show that the implication also holds at depth $n+1$.

Let G' be a goal at depth $n+1$ obtained by resolving the goal G of depth n with a variant of rule r. Let G be $\leftarrow D_1, \cdots, D_k$. Let the variant of r used be

$$h \leftarrow b_1, \cdots, b_m$$

where $b_1, \ldots, b_m$ and h are atoms. Suppose the computed substitution at G is ϕ, the most general unifier of h and the leftmost atom D_1 in G is ψ. The computed substitution at G' is then $\theta = \phi\psi$. Since we used a variant of r with new variable names, it follows that $b_1\psi = b_1\theta$ and $h\psi = h\theta$. Let M be the right ancestor of D_1 in G. By the induction hypothesis, $query(D_1, M\phi)$ holds. There are two cases:

$m \geq 1$ (In other words, the body of r is nonempty.) We only need to consider correspondence (a) since no new subrefutations are generated. Then the leftmost atom in G' is $b_1\psi = b_1\theta$. The construction of $MTRR'(P)$ implies that if $m > 1$, or if $m = 1$ and b_1 is not right-recursive, then $query(b_1, b_1)\theta$ holds, by the firing of a rewritten rule of the form

$$query(b_1, b_1) \leftarrow query(h, h)$$

if h is not right-recursive, or if h is right-recursive then by a rule of the form

$$query(b_1, b_1) \leftarrow query(h, A).$$

If $m = 1$ and b_1 is right-recursive, then the right ancestor of $b_1\theta$ in G' is M. In this case, we also know that $query(b_1, M)\theta$ holds by the firing of the rewritten rule of the form

$$query(b_1, A) \leftarrow query(h, A).$$

$m = 0$ (I.e., the body of r is empty.) We consider correspondence (b) first. We have a subrefutation for D_1 with computed substitution θ, and possibly subrefutations for other ancestors of G' also with computed substitution θ. We use a second (inner) induction argument to show that for every ancestor G_A of G' with leftmost atom A that is its own right ancestor, $A\theta$ holds. The induction is on the distance from G' on the chain of ancestors of G'.

Base case: $G_A = G'$, i.e., $A = D_1$. If D_1 is its own right ancestor, then $query(D_1, D_1)$ holds, by the induction hypothesis. Then the rewritten rule

$$h \leftarrow query(h, h)$$

fires if h is not right-recursive, or if h is right recursive then the rule

$$A \leftarrow query(h, A)$$

fires. In either case, $D_1\psi = D_1\theta$ holds.

Induction step: Suppose G_A is an ancestor of G' with A appearing leftmost in G_A, that there is a subrefutation of A with computed substitution θ through G', and that the (inner) induction hypothesis holds for all descendents of G_A on the path to G'. Suppose the right ancestor of A in G_A is A itself. Let G_B be the closest descendent of A along the path to G' such that B is leftmost in G_B, and B is its own right ancestor in G_B. Then by the (inner) induction hypothesis, $B\theta$ holds. Consider the goal G_C that is resolved with a rule r' to generate (a more general instance of) B. Suppose that the variant of r' used is

$$g \leftarrow e_1, \cdots, e_q.$$

Let C be the leftmost atom in G_C. Since we have a subrefutation for A, the subgoal of r' matching B must be rightmost in r'. The right ancestor of C must be A, since the right ancestor relationship is idempotent, and since G_B is the closest descendant of A having leftmost atom that is its own right ancestor. By the (outer) induction hypothesis, $query(C, A\theta_C)$ holds, where θ_C is the computed substitution at G_C, and is more general that θ.

Consider the rewritten rule

$$A \leftarrow query(g, A), e_1, \cdots, e_q.$$

None of $e_1, \ldots, e_{q-1}$ are rightmost in r'. Hence the fact that there are subrefutations for appropriate instances of e_1 to e_{q-1} implies that $e_i\theta_i$ holds for $i = 1, \ldots, q-1$ (where θ_i is defined below), by the (outer) induction hypothesis. For $i = 1, \ldots, q-1$,

$$\theta_i = \theta_{i-1}\psi_i$$

for some substitutions ψ_i, and $\theta_0 = \theta_C$. Now $B\theta = e_q\theta$ holds, and since the substitutions on all other atoms in the rule above are more general than θ, it follows that $A\theta$ is inferred.

We now look at correspondence (a). Consider the status of $G' = \leftarrow D_2\psi, \cdots, D_k\psi$. Let r' now be the rule variant

$$g \leftarrow e_1, \cdots, e_q$$

in which (a more general version of) D_2 was first introduced. Let H be the corresponding goal in which (a more general version of) D_2 first appears, and let θ_0 be the computed substitution at H. Let H_D be the parent of H that is resolved with r', with leftmost atom D and having computed substitution θ_D. By the induction hypothesis, $query(D, M\theta_D)$ holds, where M is the right ancestor of D in H_D. Let $e_1, \cdots, e_j$ be the subgoals to the left of D_2 in r', so that D_2 is an instance of e_{j+1}. Then by G' there is a subrefutation for each e_i with computed substitution θ_i such that for $i = 1, \ldots, j$,

$$\theta_i = \theta_{i-1}\psi_i$$

for some substitutions ψ_i, and θ_j is more general than θ. Since none of the subgoals e_i in r' to the left of D_2 is rightmost in r', each e_i is its own right ancestor and $query(e_i, e_i)\theta_{i-1}$ holds for $i = 1, \ldots, j$ by the induction hypothesis.

By the induction hypothesis we have subrefutations of depth at most n for each $e_i\theta_{i-1}$ for $i = 1, \ldots, j-1$ with computed answer substitutions $\theta_{i-1}\psi_i$, and hence $e_i\theta_i$ holds. The subrefutation of e_j is at depth $n+1$, and $e_j\theta$ holds by correspondence (b) discussed above. Consider the rewritten rule

$$Q_1 \leftarrow Q_2, e_1, \cdots, e_j$$

where Q_1 and Q_2 are as follows. Q_1 is $query(e_{j+1}, A)$ if e_{j+1} and g are both right-recursive and e_{j+1} is rightmost in r'; otherwise Q_1 is $query(e_{j+1}, e_{j+1})$. If g is right-recursive then Q_2 is $query(g, A)$; otherwise Q_2 is $query(g, g)$.

This rule fires with Q_2 unifying with $query(D, M\theta_D)$ to give $Q_1\theta$. If e_{j+1} and g are both right-recursive and e_{j+1} is rightmost in r', then M is the right ancestor of D_2 in G' and $query(e_{j+1}, A)\theta = query(D_2, M\theta)$ is generated. Otherwise, D_2 is its own right ancestor in G' and $query(e_{j+1}, e_{j+1})\theta = query(D_2, D_2)$ is generated.

($\Leftarrow$) The base case is also straightforward: After the first iteration of $MTRR'(P)$ only $query(Q, Q)$ holds. The SLD-tree for Q has root $\leftarrow Q$, with leftmost atom Q, the identity computed substitution and Q itself as right ancestor.

Let us now consider the induction step. Suppose that the implication from right to left holds for iterations up to n. We show that the implication also holds at iteration $n+1$.

We consider correspondence (a) first. Suppose $query(L, N)$ is inferred at stage $n+1$. Then there are two cases:

1. $query(L, N)$ is generated by firing the rewritten rule variant

$$query(b_m, A) \leftarrow query(h, A), b_1, \cdots, b_{m-1}.$$

Let r be the rule variant

$$h \leftarrow b_1, \cdots, b_m.$$

Then L must have a right-recursive predicate, a more general version of L appears rightmost in r, and h is also right-recursive.

Suppose that the matching tuples used in the body are $query(H, B)$, $e_1, \cdots, e_{m-1}$ respectively. Let θ_0 be the most general unifier of $query(H, B)$ and $query(h, A)$. Let θ_i be θ_{i-1} composed with the most general unifier of $b_i\theta_{i-1}$ and e_i, for $i = 1, \cdots, m-1$, and let $\theta = \theta_{m-1}$. Thus $query(L, N) = query(b_m, B)\theta$. $query(H, B)$ holds at stage n. Hence, by the induction hypothesis there is a goal G with leftmost atom H, computed answer substitution ϕ, the right ancestor of H is C, and $B = C\phi$. G can be resolved with r, using most general unifier ψ to get a goal $G_1 = b_1\theta_0, \cdots, b_m\theta_0, \cdots$ where $\theta_0 = \phi\psi$ is the computed substitution at G_1. By the induction hypothesis, since e_1 holds at stage n, $b_1\theta_0$ has a subrefutation from G' with computed substitution θ_1. Consider G_2, the node with computed substitution θ_1 and leftmost atom $b_2\theta_1$, at which the previous subrefutation was generated. Again, by the induction hypothesis, there is a subrefutation for $b_2\theta_1$ with computed substitution θ_2. We proceed in this way until we reach a goal node G_m. The computed substitution at G_m is $\theta_{m-1} = \theta$, and the leftmost atom in G_m is $b_m\theta = L$. The right ancestor of L is C, and $N = C\theta$.

2. $query(L, N)$ is generated by firing the rewritten rule variant of the form

$$query(b_k, b_k) \leftarrow query(h, A), b_1, \cdots, b_{k-1}$$

or of the form

$$query(b_k, b_k) \leftarrow query(h, h), b_1, \cdots, b_{k-1}.$$

Let r be the rule variant

$$h \leftarrow b_1, \cdots, b_m$$

from which the rewritten rule was generated, where $m \geq k$. (If h is right-recursive then the first form of the rewritten rule would have been generated; otherwise, the second form would have been generated.) It must be the case that either L has a predicate that is not right-recursive, or b_k is not rightmost in r, or h is not right-recursive.

Here, (a more general version of) L appears as the kth subgoal of r, and $N = L$. Suppose that the matching tuples used in the body are $query(H, B)$, $e_1, \cdots, e_{m-1}$ respectively. Let θ_0 be the most general unifier of $query(H, B)$ and either $query(h, A)$ or $query(h, h)$ (depending on which form of rewritten rule was used). Let θ_i be θ_{i-1} composed with the most general unifier of $b_i\theta_{i-1}$ and e_i, for $i = 1, \cdots, k-1$, and let $\theta = \theta_{k-1}$. Thus $query(L, N) = query(b_k, b_k)\theta$. $query(H, B)$ holds at stage n. Hence, by the induction hypothesis there is a goal G with leftmost atom H, computed answer substitution ϕ, the right ancestor of H is C, and $B = C\phi$. G can be resolved with r, using most general unifier ψ to get a goal $G_1 = b_1\theta_0, \cdots, b_m\theta_0, \cdots$, where $\theta_0 = \phi\psi$ is the computed substitution at G_1. By the induction hypothesis, since e_1 holds at stage n, $b_1\theta_0$ has a subrefutation from G' with computed substitution θ_1. Consider G_2, the node with computed substitution θ_1 with leftmost atom $b_2\theta_1$, at which the previous subrefutation was generated. Again, by the induction hypothesis, there is a subrefutation for $b_2\theta_1$ with computed substitution θ_2. We proceed in this way until we reach a goal node G_k. The computed substitution at G_k is $\theta_{k-1} = \theta$, and the leftmost atom in G_k is $b_k\theta = L$. The right ancestor of L is L itself.

We now consider correspondence (b). Suppose N is inferred at stage $n+1$.

If the predicate of N is not right-recursive, then N must have been inferred due to the firing of a rewritten rule

$$h \leftarrow query(h, h), b_1, \cdots, b_m.$$

Let r be the rule variant

$$h \leftarrow b_1, \cdots, b_m$$

from which this rewritten rule was generated. Let θ be the substitution generated by the rule firing (as in the proof of correspondence (a) above), so that $h\theta = N$. Each of the conjuncts used in the body must have an appropriate instance that holds at stage n. Using an argument similar to that used for correspondence (a) above, we can find a goal node G_{m+1} such that the following conditions hold: The computed substitution at G_{m+1} is θ, and the leftmost atom in G_{m+1} is not from r. The goal node G corresponding to the *query* tuple used in firing the rewritten rule above has leftmost atom L, say, that is its own right ancestor in G. G_{m+1} is a descendent of G in which all atoms "deriving from" L have been resolved out. Hence there is a subrefutation for L with computed answer substitution θ. Finally, by tracing the individual substitutions generated as in correspondence (a) above, we can shown that $N = L\theta$.

If the predicate of N is right-recursive, then N must have been inferred due to the firing of a rewritten rule

$$A \leftarrow query(h, A), b_1, \cdots, b_m.$$

Let r be the rule variant

$$h \leftarrow b_1, \cdots, b_m$$

from which this rewritten rule was generated. Let θ be the substitution generated by the rule firing (as in the proof of correspondence (a) above), so that $A\theta = N$. Each of the conjuncts used in the body must have an appropriate instance that holds at stage n. Using an argument similar to that used for correspondence (a) above, we can find a goal node G_{m+1} such that the following conditions hold: The computed substitution at G_{m+1} is θ, and the leftmost atom in G_{m+1} is not from r. The goal node G corresponding to the *query* tuple used in firing the rewritten rule above has leftmost atom L, say, with right ancestor C, appearing leftmost in G_C. The right ancestor of C is G_C is C itself. G_{m+1} is a (closest) descendent of G_C in which all rule-descendents of C in G have been resolved out. Hence there is a subrefutation for C with computed answer substitution θ. Finally, by tracing the individual substitutions generated as in correspondence (a) above, we can shown that $N = C\theta$. ∎

References

[1] Apt, K. R., and Van Emden, M. H. Contributions to the theory of logic programming. *JACM 29*, 3 (1982), 841–862.

[2] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems* (1986), pp. 1–15.

[3] Beeri, C., and Ramakrishnan, R. On the power of magic. *Journal of Logic Programming 10* (1991), 255–300. Preliminary version appeared in the 6th ACM Symposium on Principles of Database Systems, 1987.

[4] Bry, F. Query evaluaton in recursive databases: Bottom-up and top-down reconciled. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (1989), Elsevier, New York, pp. 20–39.

[5] Chen, W., Kifer, M., and Warren, D. S. HiLog: A first order semantics for higher-order logic programming constructs. In *Proc. North American Logic Programming Conference* (1989), MIT Press, Cambridge, Massachusetts, pp. 1090–1114.

[6] Clark, K. L. Predicate logic as a computational formalism. Tech. Rep. 79/59, Department of Computing, Imperial College, London, 1979.

[7] Codish, M., Dams, D., and Yardeni, E. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science 124*, 1 (1994), 93–125.

[8] Hill, R. LUSH resolution and its completeness. Tech. Rep. DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[9] Kemp, D., Stuckey, P., and Srivastava, D. Query restricted bottom-up evaluation of normal logic programs. In *Proc. Joint International Conference and Symposium on Logic Programming* (1992), MIT Press, Cambridge, Massachusetts, pp. 288–302.

[10] Kemp, D. B., Ramamohanarao, K., and Somogyi, Z. Right-, left- and multi-linear rule transformations that maintain context information. In *Proceedings of the International Conference on Very Large Databases* (1990), Morgan Kaufman, Palo Alto, pp. 380–391.

[11] Lloyd, J. W. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, New York, 1987.

[12] Morishita, S. An alternating fixpoint tailored to magic programs. In *Proceedings of the Twelfth ACM Conference on Principles of Database Systems* (1993), pp. 123–134.

[13] Mumick, I. S., and Pirahesh, H. Right-linear and overbound queries. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems* (1991), pp. 127–141.

[14] Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D. Argument reduction by factoring. In *Proceedings of the International Conference on Very Large Databases* (1989), Morgan Kaufman, Palo Alto, pp. 173–182.

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

[16] Ramakrishnan, R. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming 11* (1991), 189–216.

[17] Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Controlling the search in bottom-up evaluation. In *Proc. Joint International Conference and Symposium on Logic Programming* (1992), MIT Press, Cambridge, Massachusetts, pp. 273–287.

[18] Ross, K. A. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems* (1991), pp. 92–101.

[19] Ross, K. A. Modular stratification and magic sets for Datalog programs with negation. *Journal of the ACM 41*, 6 (1994), 1216–1266.

[20] Sacca, D., and Zaniolo, C. Implementation of recursive queries for a data language based on pure Horn logic. In *Proceedings of the Fourth International Conference on Logic Programming* (1987), MIT Press, Cambridge, Massachusetts, pp. 104–135.

[21] Sagiv, Y., 1990. (personal communication).

[22] Sethi, R. *Programming Languages, Concepts and Constructs.* Addison-Wesley, Reading, MA, 1989.

[23] Sudarshan, S. *Optimizing Bottom-Up Query Evaluation for Deductive Databases.* PhD thesis, University of Wisconsin, Madison, November 1992. University of Wisconsin Technical Report 1125.

[24] Ullman, J. D. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (1989), pp. 140–149.

[25] Ullman, J. D. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).