Efficient Evaluation of Right-, Left-, and Multi-Linear Rules

J. F. Naughton* R. Ramakrishnan† Y. Sagiv‡ J. D. Ullman§

Abstract

We present an algorithm for the efficient evaluation of a useful subset of recursive queries. Like the magic sets tranformation, the algorithm consists of a rewriting phase followed by semi-naive bottom-up evaluation of the resulting rules. We prove that on a wide range of recursions, this algorithm achieves a factor of $O(n)$ speedup over magic sets. Intuitively, the transformations in this algorithm achieve their performance by reducing the arity of the recursive predicates in the transformed rules.

*Department of Computer Science, Princeton University. Work supported by DARPA and ONR contracts NOOO14-85-C-0456 and N00014-85-K-0465, and by NSF Cooperative Agreement DCR-8420948

†Department of Computing Sciences, University of Wisconsin, Madison, Wisconsin. Work supported in part by an IBM Faculty Development award and NSF grant IRI-8804319. Part of this work was done while visiting IBM Almaden Research Center.

‡Dept. of Computer Science, Hebrew University, Jerusalem, Israel. Work supported in part by grant 85-00082 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel. Part of the work of Y. Sagiv was done while visiting Stanford University, where it was supported by NSF grant IRI-87-22886 and Air Force grant AFOSR-88-0266 and a grant of the IBM Corp.

§Stanford University. Work supported by NSF grant IRI-87-22886 and Air Force grant AFOSR-88-0266 and a grant of the IBM Corp.

1 Introduction

A large number of recursion evaluation techniques have been designed with generality in mind, including [BMSU86,KL86,HN84,SZ86,Vie86]. Perhaps the best known is the magic sets transformation [BMSU86,BR87]. The magic sets transformation has the advantage that it is conceptually simple — a rewriting phase followed by semi-naive bottom-up evaluation — and it performs well over a wide range of queries [BR86]. However, on some common and simple recursions, much better algorithms are known.

Other recursion evaluation procedures are motivated by the observation that for useful subsets of recursive queries, there exist high-performance special purpose evaluation algorithms [Agr87,Cha81,HN88,Nau88,Nau87,RHDM86]. While these algorithms outperform magic sets on many queries, the set of queries for which this is so has not been described precisely. Also, because these algorithms do not follow the magic-sets paradigm of a rewriting phase followed by semi-naive bottom-up evaluation, they do not fit as cleanly into a general-purpose logic query processor and it is more difficult to extend them to handle language constructs such as sets, negation, and function symbols.

In this paper we present a rewriting strategy in the spirit of the magic sets transformation that performs much better than magic sets on a useful subset of recursions. Intuitively, the transformations in this rewriting strategy are better than the magic-sets transformation because they have an important feature not found in the magic-sets transformation: the arity of the recursive predicate is smaller in the transformed rules than in the original rules.

We give a syntactic description of the class of recursions to which the new transformations apply and prove that, except in degenerate cases, the new transformations generate strictly fewer facts than the magic-sets transformation.[1]

Under less general conditions, our transformation achieves an $O(n)$ speedup, that is, for every $n > 0$, there is a database of size $O(n)$ such that the new transformations generate $O(n)$ facts as compared to the $O(n^2)$ facts generated by the magic sets transformation. In database applications, where n can be large, this improvement may mean the difference between a feasible and an infeasible computation.

As an example of the power of the transformation presented in this paper, consider the query $t(x_0, Y)$? about the transitive closure. There are three common ways of writing the rules for the transitive closure, t, of a directed graph with arcs given by a database predicate e:

$$\begin{aligned} t(X,Y) &\text{ :- } e(X,Y). \\ t(X,Y) &\text{ :- } e(X,Z),\ t(Z,Y). \end{aligned}$$

or

$$\begin{aligned} t(X,Y) &\text{ :- } e(X,Y). \\ t(X,Y) &\text{ :- } t(X,Z),\ e(Z,Y). \end{aligned}$$

or

$$\begin{aligned} t(X,Y) &\text{ :- } e(X,Y). \\ t(X,Y) &\text{ :- } t(X,Z),\ t(Z,Y). \end{aligned}$$

The transformations presented in this paper will evaluate the query $t(x_0, Y)$ using essentially a linear, breadth-first search from node x_0, no matter which way of defining the closure is chosen. To our knowledge this is the first evaluation technique with this property.

While the transformation presented in this paper achieves an $O(n)$ speedup as compared to the magic-sets transformation, it achieves a much higher speedup as compared to Prolog. There are database for which Prolog (with or without tail recursion elimination) takes exponential time to find all answers to the query $t(x_0, Y)$?, while our algorithm finds all answers in linear time.

For clarity of exposition, we divide the class of rules to which our transformation applies into four categories. In Section 2, we consider *right-linear* recursions, where the recursive subgoal is most naturally placed at the right end of the rule body. Section 3 considers *left-linear* recursions, where the recursive subgoal is naturally placed at the left end, or first in the list of subgoals. Section 4 discusses recursions that have both right- and left-linear rules, while Section 5 discusses recursions in which there may be multi-linear rules, that is, rules that contain both right- and left-linear predicate instances.

We end this section with few basic definitions. In this paper, we consider *Datalog programs,* i.e., programs consisting of Horn-clause rules without function symbols, such that every variable in the head of a rule also appears in the body. An *EDB predicate* is a predicate that appears only in rule bodies, and an *IDB predicate* is a predicate that appears in some rule heads. We consider programs with a single IDB predicate, denoted as p, and we also refer to this predicate as the *recursive predicate.* A rule is *recursive* if it has an occurrence of p in its body; otherwise, it is a *basis* (or *nonrecursive*) rule. For simplicity's sake, we assume that rules do not have constants in their heads. Unless explicitly stated otherwise, we assume that in the head of a rule, no variable appears in more than one column.

2 Right-Linear Recursions

In order to motivate the transformation for right-linear rules, consider the following example where a binary recursive predicate can be replaced by a unary recursive predicate.

Example 2.1 Consider the following ancestor rules:[2]

$$\begin{aligned} r_1 :\ anc(X,Y) &\text{ :- } par(X,Y). \\ r_2 :\ anc(X,Y) &\text{ :- } par(X,Z),\ anc(Z,Y). \end{aligned}$$

If the query is $anc(x_0, Y)$, then the magic-sets transformation produces the following set of rules

$$\begin{aligned} &m\_anc(x_0). \\ m\_anc(Y) &\text{ :- } m\_anc(X),\ par(X,Y). \\ anc(X,Y) &\text{ :- } m\_anc(X),\ par(X,Y). \\ anc(X,Y) &\text{ :- } m\_anc(X),\ par(X,Z), \\ &\qquad anc(Z,Y). \end{aligned}$$

where *m_anc* is the magic predicate whose value is the set of all possible bindings for the first column of *anc*.

Suppose that the relation for *par* is the following set of tuples:

$$\{(x_0,x_1),(x_1,x_2),\ldots,(x_{n-1},x_n)\}$$

Thus, the relation for the magic predicate is

$$\{x_0,x_1,x_2,\ldots,x_n\}$$

[1]The new transformations never generate more facts than magic-sets.

[2]Lowercase letters denote constants and uppercase letters denote variables.

and hence, the set of facts computed for *anc* is

$$\{(x_i, x_j) \mid 0 \le i < j \le n\}$$

The whole evaluation takes $\Omega(n^2)$ time, since $\Omega(n^2)$ facts are derived for *anc*. However, the query is essentially a single-source reachability problem that can be solved in $O(n)$ time on a graph of n edges. In fact, it is sufficient to compute the magic set and use it to select answers from the EDB predicate *par*. That is, all answers are produced by evaluating the following (nonrecursive) rule: [3]

$$answers(Y) \text{ :- } m\_anc(X),\ par(X,Y). \quad (1)$$

□

2.1 Definition of Right-Linear Rules

If we examine again the ancestor rules r_1 and r_2, we can observe why rule r_1 computes all answers. In a top-down evaluation of these rules, each answer is obtained by some number of application of r_2, the recursive rule, followed by one application of r_1, the basis rule. When *anc* is called with the first argument bound and the second argument free, the recursive call to *anc* also has that binding pattern, and moreover, the variable in the free argument of the recursive call (i.e., Y) is the same variable that appears in the free argument of the head. Thus, any values for that variable found by the basis rule are propagated up through all the recursive calls, until they reach the top-level goal, i.e., the original query.

There is another factor that we find essential: the initial binding for the top-level goal is a single value x_0. Hence, we can determine the values for the free argument of *anc* without concern for the matching bound argument. We know that when any value y_0 for the second argument of *anc* reaches the top-level goal, it will be paired with x_0 to make a tuple (x_0, y_0). If we had started at the top-level goal with more than one value in the binding set for the first argument of *anc*, then we would have to propagate complete tuples of *anc* through the recursive calls, in order to keep track of who was an ancestor of whom. In that case, the magic-sets approach would be about as efficient as possible.[4]

We may generalize the ancestor example to a class of rules for which the same evaluation technique works. These rules involve one IDB predicate p. The tuples in m_p, the magic predicate for p, are used with the basis rules, to obtain answer tuples. If the conditions listed below are met, these answer tuples will be all and only the tuples that, when coupled with the constants in the query, are answers to the query itself.

The advantage to evaluating the query this way is that the recursively evaluated predicate, m_p, has fewer arguments than the original recursive predicate, p. Thus, the evaluation of m_p is generally much faster than the evaluation of p itself, as we observed for the ancestor rules.

Suppose we are given a collection of (Datalog) rules with a single IDB predicate, p, and an adorned query goal p^α (the adornment α specifies which arguments are bound and which are free, e.g., p^{bf} means that the first argument of the query is bound and the second is free). We say that these rules are *right-linear* (with respect to a given adornment α) if each recursive rule is of the following form:

$$\begin{aligned} &p(X_1, \ldots, X_m, Y_1, \ldots, Y_n) \text{ :-} \\ &\quad \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k, \\ &\quad p(W_1, \ldots, W_m, Y_1, \ldots, Y_n). \end{aligned}$$

where, without loss of generality, we assume that the m leftmost argument positions of p are the bound ones (according to α), and the following conditions are satisfied:

- $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$ are all subgoals with EDB predicates (i.e., there is only one recursive subgoal),
- the Y_i's are all distinct, and each one appears exactly twice in the rule, i.e., once in the head and once in the recursive subgoal (in the same column as in the head), and
- each W_i either appears in $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$ or is one of $X_1, X_2, \ldots, X_m$.

The above conditions imply the following:

- If we start with the head adornment p^α and the p subgoal is the last one in the body (as shown above), then the adornment of the p subgoal (as produced by sideways information passing [BMSU86,BR87]) is also α.
- An answer to the p subgoal is also an answer to the head goal, because the free arguments of p have the same variables in the head and in the recursive subgoal. Moreover, when an answer to a subgoal is substituted in the body of a recursive rule, it does not generate any other answer (except itself) for the goal in the head.

[3]We assume that by having appropriate indices, this rule can be evaluated in linear time in the size of the *par* relation.

[4]However, if the initial set of bindings were small, we could evaluate individually the set of answers for each binding, and possibly obtain the answer faster than the magic-sets algorithm would.

2.2 Evaluating Right-Linear Datalog Programs

We shall now describe formally how to transform right-linear rules into more efficient rules. In the next subsection, we shall show formally that the transformation of this subsection is more efficient than the magic-sets transformation.

For each recursive rule

$$\begin{aligned} p(X_1,\ldots,X_m,Y_1,\ldots,Y_n) \;\text{:-}\; & \\ & \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k, \\ & p(W_1,\ldots,W_m,Y_1,\ldots,Y_n). \end{aligned}$$

where the m leftmost argument positions of p are assumed, without loss of generality, to be the bound ones, we construct the rule

$$\begin{aligned} m\_p(W_1,\ldots,W_m) \;\text{:-}\; & m\_p(X_1,\ldots,X_m), \\ & \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k. \end{aligned}$$

The predicate *m_p* is just the magic predicate for p, i.e., the set of tuples that appear in the bound arguments of p. To this set of recursive rules, we add the rule

$$m\_p(x_1,\ldots,x_m).$$

where $x_1,\ x_2,\ldots,x_m$ are the constants in the query goal.

Now we create rules for the *answer predicate, a_p*, which is the set of tuples that appear in the free arguments of p. The rules for *a_p* come from the basis (i.e., nonrecursive) rules, since it is the values from the basis that are propagated all the way up to become answers for the query. Thus, for each basis rule

$$p(X_1,\ldots,X_m,Y_1,\ldots,Y_n) \;\text{:-}\; \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k.$$

we construct the rule

$$a\_p(Y_1,\ldots,Y_n) \;\text{:-}\; m\_p(X_1,\ldots,X_m),\ \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k.$$

Finally, the answer to the query can be derived from *a_p* by the rule[5]

$$p(x_1,\ldots,x_m,Y_1,\ldots,Y_n) \;\text{:-}\; a\_p(Y_1,\ldots,Y_n).$$

where $x_1,\ x_2,\ldots,x_m$ are the constants from the query.

Only the new rules have to be evaluated in order to compute the answer to the query. The program consisting of the new rules is referred to as the *reduced* program; the program generated by the magic-sets transformation is referred to as the *magic* program.

[5]In principle, this rule can be combined with the basis rules for *a_p* in order to avoid generating each answer both for *a_p* and *p*. The reason, however, for doing the transformation as described is to make it correct for programs that have both right-linear and left-linear rules also, as discussed in Section 4.

Theorem 2.1 *The reduced program produces the same answer as the original right-linear program, and is (at least) as efficient as the magic program.*

2.3 $O(n)$ Speedup

Under fairly general conditions this rewriting strategy achieves $O(n)$ speedup over Magic Sets; that is, there are EDBs of size $O(n)$ and queries for which the magic program generates $O(n^2)$ facts while the reduced program generates only $O(n)$ facts. The conditions for obtaining $O(n)$ speedup are as follows. The reduced program must have a pair of rules

$$\begin{aligned} m\_p(W_1,\ldots,W_m) \;\text{:-}\; & m\_p(X_1,\ldots,X_m), \\ & \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k. \qquad (2) \\ a\_p(Y_1,\ldots,Y_n) \;\text{:-}\; & m\_p(V_1,\ldots,V_m), \\ & \mathcal{H}_1,\ \ldots,\ \mathcal{H}_q. \qquad (3) \end{aligned}$$

such that the following is true.

1. W_i and W_j $(i,j = 1,\ldots,m)$ are identical if and only if X_i and X_j are identical (i.e., $m\_p(W_1,\ldots,W_m)$ and $m\_p(X_1,\ldots,X_m)$ can be unified without having to equate either a pair of distinct W_i's or a pair of distinct X_i's).
2. W_i and W_j $(i,j = 1,\ldots,m)$ are identical if and only if V_i and V_j are identical.
3. At least one W_j is different from all the X_i's.
4. At least one Y_j is different from all the V_i's.
5. Every $\mathcal{G}_i$ $(i = 1,\ldots,k)$ is an atom for a distinct predicate, and it has (at least) one X_j, such that W_j is different from all the X_i's.
6. Every $\mathcal{H}_i$ $(i = 1,\ldots,q)$ is an atom for a distinct predicate, and it has (at least) one V_j, such that in rule 2, W_j is different from all the X_i's.
7. Each one of the two rules has in its body (at least) one predicate that appears in no other rule.

Theorem 2.2 *Suppose that a reduced right-linear program has a pair of rules satisfying the above conditions. Then for every $n > 0$, there is an EDB of size $O(n)$ and a query, such that the magic program generates $O(n^2)$ facts while the reduced program generates only $O(n)$ facts.*

Without Items 5, 6, and 7, there is still at least an $O(n^2)$ difference between the number of facts generated by the magic program and the number of facts generated by the reduced programs. However, without those items the ratio of these numbers is not guaranteed to be $O(n)$. In the most general case, there is at least an $O(n)$ difference between the two programs unless either all columns of p are bound, or the right-linear program is equivalent to a nonrecursive program.

3 Left-Linear Recursions

Left-linear rules are dual to right-linear rules and can be evaluated in a similar way. The difference, however, is that instead of computing the magic set and using it to select answers, we compute directly the set of all answers.

Example 3.1 Consider the following ancestor rules:

$$\begin{aligned} anc(X,Y) &\ \text{:-}\ par(X,Y). \\ anc(X,Y) &\ \text{:-}\ anc(X,Z),\ par(Z,Y). \end{aligned}$$

If the query is $anc(x_0, Y)$, then the constant from the query can be pushed from the head into the recursive subgoal [AU79], i.e., the rules are equivalent to the following ones:

$$\begin{aligned} anc(x_0,Y) &\ \text{:-}\ par(x_0,Y). \\ anc(x_0,Y) &\ \text{:-}\ anc(x_0,Z),\ par(Z,Y). \end{aligned}$$

In the above rules, the first column of *anc* is redundant, since it is always x_0. Therefore, it can be deleted, i.e., *anc* can be replaced with the answer predicate *a_anc* to obtain the following rules:

$$\begin{aligned} a\_anc(Y) &\ \text{:-}\ par(x_0,Y). \\ a\_anc(Y) &\ \text{:-}\ a\_anc(Z),\ par(Z,Y). \\ anc(x_0,Y) &\ \text{:-}\ a\_anc(Y). \end{aligned}$$

□

3.1 Left-Linear Datalog Programs

We can generalize the above example by defining "left-linear" rules with respect to an adornment α. The essential requirements are that the adornment of the IDB predicate remains α if the recursive subgoal appears first, and that the bound arguments share the same variables in the head and recursive subgoal. Formally, we define a collection of rules with a single IDB predicate p to be *left-linear* with respect to an adornment α for p if the following holds. First, all rules (recursive and nonrecursive) have heads with distinct variables in bound columns. Second, each recursive rule is of the following form:

$$\begin{aligned} &p(X_1,\ldots,X_m,Y_1,\ldots,Y_n)\ \text{:-} \\ &\quad p(X_1,\ldots,X_m,V_1,\ldots,V_n), \\ &\quad \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k. \end{aligned}$$

where without loss of generality we assume that the m leftmost argument positions of p are the bound ones (according to α), and the following conditions are satisfied:

- $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$ are all subgoals of EDB predicates (i.e., there is only one recursive subgoal),
- Each Y_i is different from $X_1, \ldots, X_m$, and similarly, each V_i is different from $X_1, \ldots, X_m$.
- the X_i's do not appear in $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$.

If the last condition is not satisfied (but the first two are satisfied), then the rules are called *pseudo-left-linear*.

Pseudo-left-linear[6] rules can be evaluated by transforming them as follows. For each recursive rule

$$\begin{aligned} &p(X_1,\ldots,X_m,Y_1,\ldots,Y_n)\ \text{:-} \\ &\quad p(X_1,\ldots,X_m,V_1,\ldots,V_n), \\ &\quad \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k. \end{aligned}$$

we construct the rule

$$\begin{aligned} a\_p(Y_1,\ldots,Y_n) &\ \text{:-}\ a\_p(V_1,\ldots,V_n), \\ &\qquad \sigma(\mathcal{G}_1),\ldots,\ \sigma(\mathcal{G}_k). \end{aligned}$$

where σ is the substitution that replaces each X_i with x_i, and $x_1, x_2, \ldots, x_m$ are the bound values in the query goal. Note that if the rule is left-linear (and not just pseudo-left-linear), then the X_i's do not appear in $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$ and the substitution σ is not needed.

For each basis rule

$$p(X_1,\ldots,X_m,Y_1,\ldots,Y_n)\ \text{:-}\ \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k.$$

we construct the rule

$$a\_p(Y_1,\ldots,Y_n)\ \text{:-}\ \sigma(\mathcal{G}_1),\ldots,\ \sigma(\mathcal{G}_k).$$

Finally, we translate *a_p* to *p* by attaching the query constants:

$$p(x_1,\ldots,x_m,Y_1,\ldots,Y_n)\ \text{:-}\ a\_p(Y_1,\ldots,Y_n).$$

Only the new rules have to be evaluated to answer the query.

Theorem 3.1 *The transformed pseudo-left-linear rules produce the same answer as the original rules, and they are no less efficient than the rules constructed by the magic-sets transformation.*

[6] The need to consider both left-linear and pseudo-left-linear rules will become apparent in the next section.

The transformed pseudo-left-linear rules are as efficient as the rules constructed by the magic-sets transformation, but not more efficient. The reason is that the magic set for p has only one tuple consisting of the constants from the query, and therefore, the rules produced by the magic-sets transformation derive the same number of facts as the transformed pseudo-left-linear rules. As will be shown in the next section, however, the above transformation can be used to obtain more efficient rules than those produced by the magic-sets transformation when programs have both right-linear and left-linear (but not pseudo-left-linear) rules.

4 Mixed-Linear Recursions

When a program has both right-linear and left-linear rules, we can transform it to a more efficient program by applying the transformations of the previous two sections to the right-linear and left-linear rules, respectively (the basis rules have to be transformed only according to the transformation for right-linear rules).

Example 4.1 Consider the following program, with a query of the form $p(x_0, Y, Z)$.

$$\begin{array}{lll} r_1: & p(X,Y,Z) & \text{:-}\ \ q(X,Y,Z). \\ r_2: & p(X,Y,Z) & \text{:-}\ \ a(X,A),\ p(A,Y,Z). \\ r_3: & p(X,Y,Z) & \text{:-}\ \ b(Y,B),\ p(X,B,Z). \\ r_4: & p(X,Y,Z) & \text{:-}\ \ c(Z,C),\ p(X,Y,C). \end{array}$$

The adornment for p is thus bff, and with respect to this adornment, rule r_2 is right-linear while rules r_3 and r_4 are left-linear. Hence, we shall rewrite the latter two as follows:

$$\begin{array}{lll} r_3: & p(X,Y,Z) & \text{:-}\ \ p(X,B,Z),\ b(Y,B). \\ r_4: & p(X,Y,Z) & \text{:-}\ \ p(X,Y,C),\ c(Z,C). \end{array}$$

Only the right-linear rule, r_2, produces new bindings for the first argument of p. Thus, for this rule, we construct the corresponding rules for the magic predicate m_p:

$$\begin{array}{l} m\_p(x_0). \\ \ \ m\_p(A)\ \ \text{:-}\ \ m\_p(X),\ a(X,A). \end{array}$$

The left-linear rules, r_3 and r_4, generate new solutions, and so, we construct the corresponding rules for a_p by replacing p with a_p and omitting the first (i.e., bound) argument:

$$\begin{array}{ll} a\_p(Y,Z) & \text{:-}\ \ a\_p(B,Z),\ b(Y,B). \\ a\_p(Y,Z) & \text{:-}\ \ a\_p(Y,C),\ c(Z,C). \end{array}$$

The basis rule is used to initialize the computation of a_p by selecting solutions according to m_p, i.e., as was done in the transformation for right-linear rules:

$$a\_p(Y,Z)\ \ \text{:-}\ \ m\_p(X),\ q(X,Y,Z).$$

Finally, the answer to the query is constructed from a_p:

$$p(x_0,Y,Z)\ \ \text{:-}\ \ a\_p(Y,Z).$$

□

Theorem 4.1 *The transformed right-linear and left-linear rules produce the same answer to the query as the original rule.*

It is not hard to see that the transformed rules are (at least) as efficient as the rules generated by the magic-sets transformation. Moreover, the results about efficiency and speedup in Subsection 2.3 are valid also for programs with both right-linear and left-linear rules.

5 Multi-Linear Rules

The ideas of the previous sections can also be generalized to nonlinear rules, as shown in the next example.

Example 5.1 Consider the ancestor rules

$$\begin{array}{ll} anc(X,Y) & \text{:-}\ \ par(X,Y). \\ anc(X,Y) & \text{:-}\ \ anc(X,Z),\ anc(Z,Y). \end{array}$$

and the query $anc(x_0, Y)$.

The recursive rule has two recursive subgoals. Suppose we unify the head of the recursive rule with the query goal, and so, we get

$$anc(x_0,Y)\ \ \text{:-}\ \ anc(x_0,Z),\ anc(Z,Y).$$

The first subgoal, $anc(x_0, Z)$, has the same constant in the bound column as the query goal. Therefore, every answer to the first subgoal is also an answer to the original query.

The second subgoal, $anc(Z,Y)$, has the same variable as the head in the free column. Therefore, every answer to the second subgoal is also an answer to the original query.

If we continue the recursion and unify the head of the recursive rule with one of the subgoals generated thus far, then it is easy to show by induction that the following principle continues to hold: Every answer to any subgoal generated during top-down expansion is also an answer to the original query. Let a_anc be a unary predicate that contains these answers.

As a result of the above observation, we can transform the above rules in the following way. The recursive rule is transformed in two steps. First, the first subgoal is replaced by the answer predicate:

$$anc(X,Y) \ :\!- \ a\_anc(Z),\ anc(Z,Y).$$

This rule is unsafe, but it is just an intermediate expression in the rewriting process, and will be transformed to a safe rule in the final program. To see that this step is correct, we observe that while the new rule might incorrectly associate answers with some subgoals, the set of answers to the original subgoal is unchanged.

In the second step, we generate the following magic rule by treating $a\_anc(Z)$ as an occurrence of an EDB subgoal.

$$m\_anc(Z) \ :\!- \ a\_anc(Z),\ m\_anc(X).$$

And from the first rule, which is a basis rule, we generate answers:

$$a\_anc(Y) \ :\!- \ m\_anc(X),\ par(X,Y).$$

Thus, the complete transformed program is

$$\begin{array}{rl} m\_anc(x_0). & \\ a\_anc(Y) & :\!- \ m\_anc(X),\ par(X,Y). \\ m\_anc(Z) & :\!- \ a\_anc(Z),\ m\_anc(X). \end{array}$$

Note that in the third rule, the occurrence of $m\_anc(X)$ in the body is redundant, i.e., this rule can be replaced with

$$m\_anc(Z) \ :\!- \ a\_anc(Z).$$

This redundancy can be tested by the method of [Sag88]. □

In general, we can have a program, with a single IDB predicate p and an adorned query goal p^α, such that each recursive rule is either right-linear, left-linear, or multi-linear, where a multi-linear rule is of the following form:

$$\begin{array}{l} p(X_1,\ldots,X_m,Y_1,\ldots,Y_n) \ :\!- \\ \quad \mathcal{G}_1,\ \ldots,\ \mathcal{G}_k, \\ \quad p(W_1,\ldots,W_m,Y_1,\ldots,Y_n). \end{array}$$

where, without loss of generality, we assume that the m leftmost argument positions of p are the bound ones (according to α), and the following conditions are satisfied:

- The X_i are distinct, and can only appear in recursive subgoals (in the same columns as in the rule head.)
- If $\mathcal{G}_i$ ($i = 1,\ldots,k$) is a recursive subgoal, then
  - its first m columns are identical to those of the head, i.e., those columns contain the variables $X_1,\ldots,X_m$, and
  - its last n columns have variables that are not among $X_1,\ldots,X_m$.
- If $\mathcal{G}_i$ ($i = 1,\ldots,k$) is an EDB subgoal, then none of its variables is among $X_1,\ldots,X_m$.
- The Y_i's are all distinct and each one appears exactly twice in the rule, i.e., once in the head and once in the last subgoal (in the same column as in the head).
- Each W_i must appear in $\mathcal{G}_1, \mathcal{G}_2,\ldots,\mathcal{G}_k$ and cannot be one of $X_1, X_2,\ldots,X_m$.

Note that the conditions do not even imply that in a multi-linear rule the adornment of a recursive subgoal (except the last one) must be α. For example, the rule

$$\begin{array}{rl} p(X,Y,Z) & :\!- \ p(X,W,V),\ b(V,U), \\ & \quad\ p(X,W,U),\ p(W,Y,Z). \end{array}$$

is a multi-linear rule, given the adorned query goal p^{bff}.

A program with left-linear, right-linear, and multi-linear rules can be transformed as follows. First, we replace all occurrences of the IDB predicate in each multi-linear rule, except the last occurrence, by the answer predicate a_p (the first m columns of these occurrences are deleted, of course). Second, we transform left-linear rules according to the transformation for these rules, and all other rules are transformed according to the transformation for right-linear rules. For the purpose of transforming a multi-linear rule, we treat occurrences of a_p in that rule as those of EDB predicates and transform that rule as if it were an ordinary right-linear rule. We have the following theorem.

Theorem 5.1 *The transformed program produces the same answer as the original program, and is (at least) as efficient as the magic program.*

6 Conclusion

We have presented a new algorithm for the evaluation of recursive queries. The algorithm has the conceptual simplicity of the magic sets transformation yet performs much better on some common recursions, including the transitive closure (and variants thereof) in its right-linear and nonlinear forms. Because of the similarity in approach to the magic sets tranformation, the algorithm presented here can easily be integrated into systems that use the magic sets transformation for general-purpose recursive query processing. We are currently investigating techniques for extending this algorithm to deal with rules that contain negation, arithmetic operators, and function symbols.

References

[Agr87] Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. In *Proceedings of the IEEE Conference on Data Engineering*, pages 580–590, Los Angeles, Californaia, February 1987.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, 1979.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[BR86] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In *Preprints of Workshop on Foundations of Deductive Databases and Logic Programming*, August 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[Cha81] C. Chang. On the evaluation of queries containing derived relations in a relational data base. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Data Base Theory, Volume 1.* Plenum Press, 1981.

[HN84] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first order databases. *JACM*, 31(1):47–85, 1984.

[HN88] Ramsey W. Haddad and Jeffrey F. Naughton. Counting methods for cyclic relations. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 333–340, Austin, Texas, March 1988.

[KL86] Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive databases. In *Proceedings of the Advanced Database Symposium*, Tokyo, Japan, 1986.

[Nau87] Jeffrey F. Naughton. One sided recursions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 340–348, San Diego, California, March 1987.

[Nau88] Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.

[RHDM86] Arnon Rosenthal, Sandra Heiler, Umeshwar Dayal, and Frank Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., June 1986.

[Sag88] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698, Los Altos, California, 1988. Morgan Kaufmann.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, 1986.