ISSN: 0128-7680
Pertanika J. Sci. & Technol. 2(2): 121-136 (1994)

A Technique for Transforming Rules in Deductive Databases

Ali Mamat
Department of Computer Science
Faculty of Science and Environmental Studies
Universiti Pertanian Malaysia,
43400 UPM Serdang, Selangor, Malaysia.

Received 15 December 1992

ABSTRAK

Di dalam pangkalan data deduktif, kecekapan penilaian pertanyaan rekursif dianggap sebagai satu matlamat penting. Suatu pendekatan untuk mencapai matlamat ini ialah dengan menggunakan kaedah-kaedah yang menjelmakan penanyaan asal kepada suatu set pertanyaan baru. Satu dari kaedah-kaedah tersebut ialah "Set Magik". Dalam kaedah "Set Magik", pertanyaan yang diungkapkan oleh petua dijelmakan kepada suatu set petua yang dipanggil petua "magik". Kertas ini menunjukkan cara untuk melakukan penjelmaan tersebut dengan menggunakan struktur data graf petua/matlamat. Kelebihan teknik yang digunakan di sini ialah ianya sangat mudah dan jelas.

ABSTRACT

In deductive databases the efficiency of recursive query evaluation is considered as an important goal. One approach to achieving this goal is to use methods that transform the original query into a new set of queries. One such method is magic sets. In the magic sets method, a query expressed by rules is transformed into a set of rules called magic rules. This paper shows how to perform this transformation by using a rule/goal graph data structure. The advantage of the technique used here is that it is very simple and clear.

Keywords: deductive databases, recursive rules, magic sets, rule/goal graph, logic, query

INTRODUCTION

The integration of logic programming and relational database technologies has resulted in a new model for databases called deductive databases. A deductive database consists of facts and rules and from the syntax point of view it is simply a Prolog program. It is claimed that deductive databases can overcome some limitations of relational databases such as recursion.

Deductive database systems extend relational database management systems, in particular, by allowing recursively defined queries to be expressed as logical rules. Rules (clauses) are usually written in the form of

$$A \leftarrow B_1, B_2, \ldots, B_m$$

where A, B_1, ..., B_m are atomic formulas, and $m \geq 0$. Rules of this type are called Horn clauses and the form in which they are written is known as a clausal form. A rule has both a declarative and a procedural (or problem solving) reading. The declarative reading of the above rule is that for all values of the variables in the rule, A is true, if B_1 and B_2 and ... and B_m are true. The procedural reading is that for all values of the variables in the rule, to solve A, B_1 and B_2... and B_m have to be solved.

Suppose we have a relation (or predicate) *manages* (E, M), which means an employee E is managed by a manager M. Let us define another predicate head (E, H). The intention of this predicate is that H is E's manager, or his/her manager's manager, and so on. Relation head can be expressed by rules as follows:

r1 : head(E, M) ← manages(E, M)
r2 : head(E, M) ← manages(E, P), head(P, M)

Suppose that we are given the query that asks all the managers of *ahmad.* With respect to the above rules, this query can how be expressed easily as

?head(ahmad, X)

So, a crucial point is to provide efficient methods for handling recursive queries.

A procedure that answers queries on deductive databases must have several important properties. First, it must return all the answers to a query and then stop. These two properties are known as completeness and termination, respectively. Second, the procedure must be efficient, i.e. it must be able to use only necessary (relevant) facts during a query evaluation. Another property is the complexity of the algorithm. While the procedure is able to restrict the computation to the relevant facts, this restriction should not lead to the more expensive procedure, for instance with an exponential cost.

In the past ten years, various strategies have been proposed to handle recursion in logic queries (Reither 1978; Chang 1981; McKay and Shapiro 1981; Henschen and Naqvi 1984; Ullman 1985; Kifer and Lozinskii 1986; Vieille 1986; Beeri and Ramakrishnan 1987). An excellent survey of the methods can be found in Bancilhon and Ramakrishnan (1986). In general, these methods may be classified in two classes, top-down methods and bottom-up methods. Top-down methods, based in proof theory, start with a query as the top goal and generate a proof tree using the axioms (rules) of the database and an inference rule such as resolution to prove that the query is a logical consequence of the database. Prolog is one such system that uses resolution as an inference rule. These methods have proved to be efficient. However, they suffer from a number of problems, including infinite looping, possible non-termination, and they tend to generate answers to

queries a tuple at a time. On the other hand, bottom-up methods, based on model theory and fixed point semantics, generate all solutions to a query and do not go into an infinite loop. However, they tend to be inefficient. The reason is that bottom-up methods do not make use of ground terms (variable free terms) in the query in the same way that top-down methods do. As a consequence, many irrelevant tuples are generated during a computation. Techniques are thus needed to restrict the generation of irrelevant tuples.

There are two basic approaches which have been pursued to obtain a more direct computation. The first approach seeks to modify the standard bottom-up computation and achieve a more run-oriented solution (Kifer and Lozinskii 1986; Vieille 1986). The second approach seeks to perform a compile-time transformation of the database, based on the given query, into an equivalent form which enables a standard bottom-up computation to focus on relevant tuples. One example of this approach is the magic sets method (Bancilhon *et al.* 1986).

The magic sets method is an algorithm for rewriting logical rules so as to cut down the number of irrelevant facts during bottom-up evaluation. Special sets of values (tuples), called magic sets, are computed so thay they can be used to restrict the computation to the relevant facts only. These sets are determined by propagating bindings in the query top-down evaluation. Thus, in the magic sets method one can find the positive features of both top-down and bottom-up computation.

The magic sets method consists of the following steps:

i) an analysis of the binding propagation behaviour during the top-down phase,
ii) the generation of magic rules from the original one by using the results obtained from (i), and
iii) the execution of magic rules.

The analysis of the binding propagation can be accomplished by using a tool called a rule/goal (Ullman 1985). In this paper we show how to generate magic rules by using the rule/goal graph. The technique is very simple and clear. The transformation algorithm which produces the same magic rules as our technique was given in Ullman (1989), but that algorithm did not make use of the rule/goal graph. These are other algorithms, such as the one in Sacca and Zaniolo (1987), which result in equivalent magic rules but those algorithms are not very clear.

CONCEPT AND TERMINOLOGY

A deductive database (DDB) D consists of facts (extensional axioms), deductive rules (intensional axioms), integrity constraints, and queries. We use Prolog-like notation for the representation of DDB.

A literal is an atomic formula or the negation of an atomic formula. An atomic formula has the form $p(t_1,..., t_n)$ where p is a predicate symbol of arity n, and each t_i is either a constant, a variable or a function. The atomic formula $p(t_1,..., t_n)$ represents a relation between terms $t_1,..., t_n$.

A clause is a disjunction of literals which has the form

$$A_1 \bigvee \cdots \bigvee A_n \bigvee \sim B_1 \bigvee \cdots \bigvee \sim B_m$$

where each A_i and B_j represent an atomic formula. (The symbols, $\bigvee$, $\bigwedge$, and $\sim$ denote *or, and, and not* respectively). Clauses can be written in an equivalent form using implication, and in the Prolog style we have

$$A_1 \bigvee \cdots \bigvee A_n \leftarrow B_1 \bigwedge \cdots \bigwedge B_m$$

All variables in a clause are assumed to be universally quantified. Such a clause is also called a deductive rule, the head of the rule is $A_1 \bigvee \cdots \bigvee A_n$, and the body of the rule is $B_1 \bigwedge \cdots \bigwedge B_m$. A Horn clause is a clause where the head of the rule is restricted to at most one positive literal.

In this paper we are concerned with a DDB as a set of definite Horn clauses, that is there is no negative literal in the body of deductive rules. Furthermore, literals have no functions. This class of deductive databases is known as a datalog database. A goal clause has a null head, and a clause with a null body is an assertion. An assertion is also called a unit clause. A unit clause which contains only ground terms (variable free terms) is called a ground unit clause or fact. A query is a goal clause.

A clause is called recursive if the same predicate symbol appears in both the head and the body. A relation R may be defined recursively by a recursive clause or by mutual recursion where R is defined (in the head) in terms of S in the body, and in turn S itself is defined in terms of R.

Integrity constraints, expressed as rules, represent restrictions that the database must satisfy, play an important role in checking update validity. They are, however, not needed in answering queries over the database.

Certain predicates are defined by the rules, that is they appear as the head of one or more rules; these are called IDB (intensional database) predicates. Other predicates are not defined by rules, but by a stored relation, and they are called EDB (extensional database) predicates. An EDB predicate can only appear in subgoals; an IDB predicate can appear in both heads and subgoals. We assume that IDB and EDB are mutually exclusive.

RULE/GOAL GRAPH

Before going further, it is convenient to introduce the concept of sideways information passing (Beeri and Ramakrishnan 1987). Informally, a sideways information passing describes how bindings (constants assigned to variables) passed to a rule's head are used to evaluate the predicates in the rule's

body. When a sideways information passing is performed, it appears that starting from a predicate name with variables bound in certain positions, we reach the same predicate name with variables bound in other positions. It is thus necessary to distinguish which argument is bound and which is free in a predicate. This leads to the concept of adornment (Ullman 1985).

An adornment or a binding pattern for an n-ary predicate p is a string s of length n of b's and f's, where b stands for bound and f stands for free. The adornment indicates which arguments of p are bound and which are not. If the ith symbol of the adornment is b, then the ith argument if p is bound. If the ith symbol of the adornment is f, then the ith argument of p is free. For a predicate p with an adornment s, we write p^s to denote the adorned predicate for p.

A rule adornment indicates which variables are bound at a point and which are free. The algorithm to decide when a variable becomes bound in a rule

$$H \leftarrow G_1,\ldots,G_k$$

follows from the sideways information passing described earlier.

1. A variable appearing in a bound argument of the rule head is bound before processing any subgoals.
2. A variable is bound after processing subgoal G_i if it was bound before processing G_i or if it appears anywhere in G_i. A rule adornment is denoted by a superscript of the form $[X_1,\ldots, X_m | Y_1, \ldots, Y_n]$, where the X's are bound and the Y's are free. We write $r_{i,j}^{[A|B]}$ to denote that [A|B] is a rule adornment for rule r_i after the consideration of the jth subgoal of r_i.

The binding patterns of predicates in a set of rules and a query can be represented by a finite structure called a rule/goal graph. The concept of a rule/goal graph originally appeared in Ullman (1985) and the rule/goal graph we used follows the one found in Ullman (1989). The rule/goal graph has two types of nodes, namely rule nodes and goal nodes. Goal nodes represent rule adornments.

Let p be the predicate of the query and a be the adornment of that predicate. The process of creating the rule/goal graph starts with the node p^α. From this node, we expand the graph according to the rules given below. As we expand, we add goal nodes and rule nodes. For rule r_s, we use $r_{s.0}$, with an adornment, to represent rule r_s before considering any subgoals, and $r_{s.i}$, with an adornment, to represent rule r_s after considering its first i subgoals. The steps to expand the rule/goal graph are as follows (Ullman 1989):

1. A goal node with an EDB predicate has no successors.
2. A goal node that is an IDB predicate p with an adornment a has successors corresponding to all the rules with head predicate p. If r_s is such a rule, then p^{α} has successor

$$r_{s.0}^{[X_1,\ldots, X_n | Y_1, \ldots, Y_m]}$$

where $X_1, \ldots, X_n$ are all the variables that appear in a argument of r_s's head that is bound according to adornment a, and $Y_1,\ldots, Y_m$ are the other variables of r_s.

3. Consider a rule node $r_{s.i}^{[X_1,\ldots, X_n | Y_1, \ldots, Y_m]}$, $i \geq 0$, and suppose $q(t_1,\ldots, t_k)$ is the i + 1st subgoal of r_s.

   a) One successor of this rule node is a goal node q^p; p is the adornment that makes the jth argument of qbound if all variables appearing in t_j are among the bound variables of the rule so far. Adornment p makes the jth argument free otherwise.
   b) If i + 1 is less than the number of subgoals in rule r_s (i.e. $q(t_1,\ldots,t_k)$ is not the last subgoal of r_s), then node

$$r_{s.i}^{[X_1,\ldots, X_n | Y_1, \ldots, Y_m]}$$

has a second successor, the node

$$r_{s,i+1}^{[X_1,\ldots, X_n, U_1,\ldots, U_j | V_1\ldots,V_h]}$$

where $U_1,\ldots,U_j$ are those variables among $Y_1,\ldots, Y_m$ that appear in $q(t_1,\ldots, t_k)$, and $V_1,\ldots, V_h$ are the remaining variables of the Y's.

Let us consider the same generation rules of *Fig. 1.*

r0 : sg (X, X) ← person (X).

r1 : sg (X, Y) ← par (X, XP), sg (XP, YP), (Y, YP).

Fig. 1. Rules for the same generation

Predicates *person* and *par* are EDB predicates and predicate sg is the only IDB predicate. The intention of predicate *person* (X) is that an individual X is a person. *Par* (X, Y) means Y is the parent of X. The intended meaning of sg (X, Y) is that X and Y are of the same generation. Rule r0 says that every person is of the same generation as her/himself. Rule r1 says that two individuals are of the same generation if their parents are.

Suppose the query is of the form sg^{bf}, that is, given an individual a find all persons who are of the same generation as a. The rule/goal graph constructed by using this query and following the given order of subgoals in the second rule is shown in *Fig. 2.*

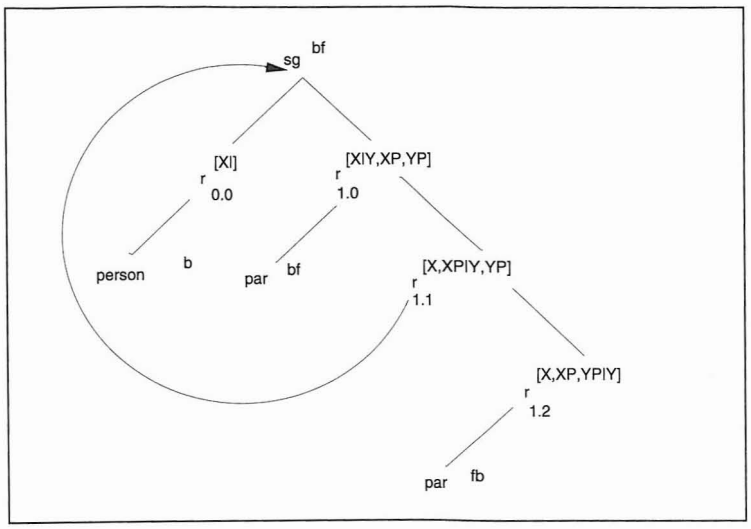

Fig. 2. The rule/goal graph for the same-generation rules

The root has adorned predicate sg^{bf}, which is the predicate in the query with its adornment. The root has two children corresponding to two rules for sg. For each child, the binding on the first argument of sg provides a binding for variable X only. The binding on variable X in rule r_0 causes the argument of the subgoal *person* (X) abound, as shown by the goal node $person^b$, the child node of $r^{[X|]}_{0.0}$.

The other child of the root is

$$r^{[X|Y, XP, YP]}_{1.0}$$

This node has child par^{bf}, representing the first subgoal of r_1. The adornment is bf, because the binding on X provides a binding for the first argument of *par* (X, XP), but not for the second. The other child that the node has is

$$r^{[X, XP|Y, YP]}_{1.1}$$

for which the variable XP becomes bound after considering the first subgoal. This node in turn has two children sg^{bf} and

$$r^{[X, XP, YP|Y]}_{1.2}$$

The former is the same as the root node, thus a new node is not needed. The latter has one and only child par^{fb}, representing the last subgoal of r_1, namely *par*(Y, YP). The first argument of this subgoal is free, while the second one is bound. Since no more nodes can be added, the construction of the rule/goal graph is complete.

THE TRANSFORMATION OF RULES USING A RULE/GOAL GRAPH

In the transformation of rules using magic sets, two new classes of predicates are introduced. These are called magic predicates and supplementary predicates. These predicates compute values that are passed from one predicate to another in the original rules, according to the particular order of execution of the subgoals. The magic predicate for a predicate p, denoted by m.p, has arguments corresponding to the bound arguments of p. The aim of the magic predicates is to compute the sets of bindings that bound arguments of derived predicate (IDB predicate) would gain during a top-down computation. The sets of values computed by magic predicates are called magic sets.

For a rule with k subgoals (predicates), we create supplementary predicates S_0,..., $S_k - 1$. A supplementary predicate has some of the variables of the rule as its arguments. For instance, S_i has arguments corresponding to those variables that are both bound and relevant after considering the first i subgoals. Variables are bound either by appearing in a bound argument of the head or by appearing in one of the first i subgoals. Variables are relevant if they appear either in the head or in the (i + 1)st or a subsequent subgoal. The supplementary predicate S_i is used to pass the bindings obtained so far, to the (i + 1)st subgoal.

Magic rules resulting from the magic sets algorithm of Ullman (1989) can be classified into five groups:

i) Rules for the magic predicates,
ii) Rules for the zeroth supplementary predicates,
iii) Rules for the other supplementary predicates,
iv) Rules for IDB predicates, and
v) The initialisation rule.

We now describe how to generate the above rules by using a rule/goal graph.

A rule/graph can be seen as a tree if we omit all arcs linking rule nodes with goal nodes which appear before the rule nodes, during the construction of the rule/goal graph. A tree can be traversed without going into an infinite loop. We want to show that by visiting each node of the tree using a depth-first search, we can generate all magic rules. Our first task is to remove all the arcs in the rule/goal graph that give rise to loops.

Suppose there is a link between rule node P and goal node Q that appears at a level higher than the level of P. To remove this link, we create a new goal node, say Q′, with the same adorned predicate as Q, and make Q′ a child node to P. Node Q′ does not have any successors. The corresponding tree of a rule/goal graph of *Fig.2* is shown in *Fig.3.*

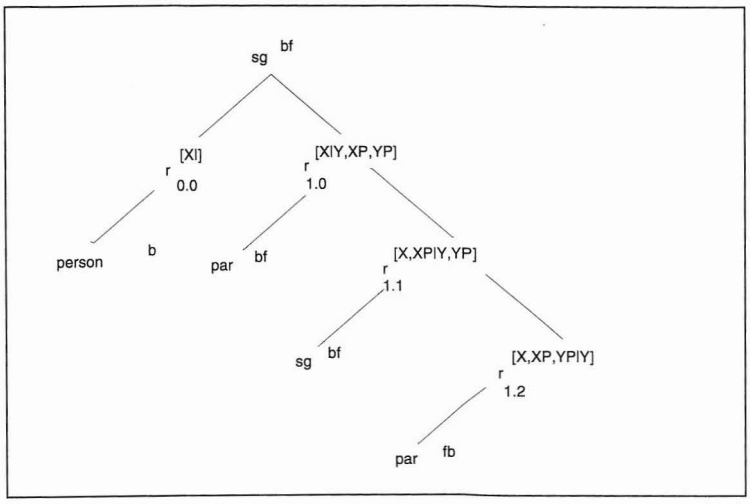

Fig.3. Rule/goal graph for the same-generation rules with the loop removed

In the following, when we refer to a tree we mean a rule/goal graph which has been transformed into a tree.

The next step is to traverse the tree starting from the root. The root is the node which was first created. As we traverse the tree, each time we reach a node, rules will be generated according to that node. For our purpose, nodes of the tree can also be classified into five groups, each of which is associated with a particular type of rule as follows:

a) The root – the initialisation rule,
b) Goal nodes with IDB predicates – rules for magic predicates,
c) Rule nodes $r_{i.0}$ – rules for zero supplementary predicates,
d) Rule nodes r_{ij} where $j \neq 0$ – rules for other supplementary predicates, and
e) Rule nodes with only one successor, i.e. their goal-node child – rules for IDB predicates.

These groups are not necessarily disjoint groups. A node might belong to more than one group. For example, node $r_{0.0}$ of *Fig.3* belongs to group c and e. Before we go further, let us see how we can determine the arguments for the supplementary predicates.

A supplementary predicate associated to jth subgoal of rule r_i is denoted by supi.j. The arguments of supi.j are variables of rule r_i that are both

bound and relevant after considering the first j subgoals. Recall that variables are bound either by appearing in a bound argument of the head or by appearing in one of the first j subgoals. Variables are relevant if they appear either in the head or in the (j + 1)st or a subsequent subgoal.

Let

V_b be a set of bound variables after considering the first j subgoals

V_r be relevant variables

V_h be variables in the head

V_q be variables occurring in (j + 1)st subgoal and the subsequent subgoals.

The supplementary variable Vs for predicate $sup_{i,j}$ is thus

$V_s = V_b \cap V_r = V_b \cap (V_h \cap V_q)$. It can be seen that the arguments for the supplementary predicate $sup_{i.0}$ are the bound variables in the head of rule r_i.

Assume that the query for the rules of *Fig.1* is ?sg(a, X). Let us now traverse the tree of *Fig.3* in a depth-first manner, and generate rules according to the node that has been reached.

1. The root

The first node that we reach is the root, i.e. the goal node sg^{bf}. We create a magic predicate m.sg, where sg is the predicate in the node. The arguments of the predicate are the bound arguments of the query. We thus have the initialisation rule

$$m.sg(a).$$

2. Rule node $r_{0.0}$

We create the rule for zeroth supplementary predicate sup0.0. The arguments of this predicate are bound variables of the node, that is X. The body of the rule is m.sg(X), where sg is the head predicate of r_0 and X is the bound arguments of the head. We thus have rule

$$sup0.0(X) \leftarrow m.sg(X).$$

The next node to be visited is the goal node $person^b$. Because it contains EDB predicate, no rules are created for this node. We then go back to the node $r_{0.0}$ and look for the right node. It happens that no such node exists, and so we create one of the rules for IDB predicates:

$$sg(X,X) \leftarrow sup0.0(X),\ person(X).$$

The head of the resulting rule is the head of rule r0. The first subgoal in the body is the supplementary predicate corresponding to the current node, and the second subgoal is the last subgoal of rule r_0.

We have finished visiting one branch of the root node. We next traverse the right branch of the root.

3. Rule node $r_{1.0}$

The way we create the zeroth supplementary rule for this node is the same as for rule node $r_{0.0}$. But here the rule involved is r_1. Thus,

$$sup1.0(X) \leftarrow m.sg(X)$$

The subsequent node to be visited is the goal node par^{bf}. As explained before, nothing will be done for EDB predicate. A traverse continues through the right child of node $r_{1.0}$.

4. Rule node $r_{1.1}$

This is the first rule node corresponding to other (non-zeroth) supplementary predicates. Before constructing a rule for this node, let us determine the arguments for the supplementary predicate. From the rule node, we know that V_b = (X, XP). Searching through the head of rule r_1 we obtain V_h = (X, Y), and through the second subgoal and the rest we get V_q = (XP, YP, Y). Therefore, the supplementary variables V_s, are given by

$$V_s = (X, XP) \cap ((X, Y) \cup (XP, YP, Y)) = (X, XP).$$

The rule for this node is

$$sup1.1(X, XP) \leftarrow sup1.0(X), par(X, XP)$$

The predicate sup1.0(X) corresponds to the parent node (node $r_{1.0}$) of the node under consideration, and par(X, XP) is the first subgoal of rule r_1. In general, for a predicate supi. j, the second subgoal in the body of the rule defining supi.j is the jth subgoal of the rule r_j. We next consider the left child of node $r_{1.1}$.

5. Subgoal node sg^{bf}

This is the only IDB goal node in the tree. For this node we generate a rule defining magic predicate m.sg. The arguments of this predicate are the bound arguments (determined by adornment bf) of subgoal sg, i.e. the second subgoal of the rule r_1. This subgoal can be identified by the index stored in the parent node (i.e. node $r_{1.1}$). The body of the rule is the supplementary predicate corresponding to the parent node ($r_{1.1}$), defined earlier. Thus, we have the following rule

$$m.sg(XP) \leftarrow sup1.1(X, XP).$$

6. Rule node $r_{1.2}$

Following the right branch of rule node $r_{1.1}$ gives us the last rule node, i.e. node $r_{1.2}$. Like its parent node, we create for this node a rule defining

other supplementary predicates. Variables constituting the argument for the predicates are given by

$$V_s = (X, XP, YP) \cap ((X, Y) \cup (P, Y)) = (X, YP).$$

The resulting rule is

$$sup1.2(X, YP) \leftarrow sup1.1(X, XP), sg(XP, YP).$$

Finally, we also generate a rule defining IDB predicate sg that appears in the head of r_1. The rule is

$$sg(X, Y) \leftarrow sup1.2(X, YP), par(Y, YP).$$

For convenience, we rewrite all magic rules that have been generated as follows:

m.sg(a)
m.sg(X) ← sup1.1(X, XP)
sup0.0(X) ← m.sg(X)
sup1.0(X) ← m.sg(X)
sup1.1(X, XP) ← sup1.0(X), par(X, XP)
sup1.2(X, YP) ← sup1.1(X, XP), sg(XP, YP)
sg(X, X) ← sup0.0(X), person(X)
sg(X, Y) ← sup1.2(X, YP), par(Y, YP)

We have thus seen that all magic rules that should be resulted by the magic sets' transformation algorithm of Ullman (1989) can be generated based on a rule/goal graph. An algorithm in the form of pseudocode for generating magic rules by using the rule/goal graph is given in an appendix.

THE CORRECTNESS AND PERFORMANCE OF THE ALGORITHM

We want to show that our algorithm and Ullman's magic sets' algorithm (Ullman 1989) are equivalent in the sense that both algorithms produce the same magic rules as the results of converting the original database rules. To do so, we have to prove that our algorithm produces only the same magic rules as Ullman's algorithm, and that every magic rule produced by the latter is produced by the former. As mentioned above, magic rules resulting from Ullman's algorithm can be classified into five groups as follows:

(i) Rules for magic predicates
(ii) Rules for zeroth supplementary predicates
(iii) Rules for other supplementary predicates

(iv) Rules for IDB predicates
(v) An initialisation rule

We start with an initialisation rule. There is only one initialisation rule, and this rule is created when the root of the rule/goal graph is processed. Each rule of group (i), rules for magic predicates, is created for each IDB predicate appearing in the body of the rule. Our algorithm creates this rule when it process the IDB goal node of the rule/goal graph. Rules of group (ii) contain a predicate of the form supi.0 (i = 0, 1,..., n − 1 where n is the number of database rules) in their heads. In the rule/goal graph, we have rule nodes of the form $r_{1.0}$. By creating a rule for each node r_i.0(i = 0, 1, ...n − 1) we obtain all the rules of group (ii). Similarly, rules of group (iii)contain predicate supi.j(i − 0,1.,,n − 1 and j = 1,2,...,k − 1 where n is as before and k is the number of subgoals in the rule r_i) in the head. These rules can be generated based on rule nodes $r_{i,j}$ of the rule/goal graph. Finally, rules of group (iv), namely modified original rules, can be obtained by creating one rule for each rule node $r_{i,k-1}$, i = 0, 1,..., n − 1 where n and k are as before.

We have thus demonstrated that all the rules of groups (i) to (v) can be created by our algorithm based on the traversal of the rule/goal graph. The next step is to demonstrate that the algorithm creates only the same magic rules as the algorithm of Ullman. Since the involved nodes of the rule/goal graph correspond to particular rules of groups (i) to (v) above, therefore no other rules can be generated. We have thus showed the equivalence of the two algorithms in terms of their functionality. The details of each rule are omitted as they are obvious.

Besides correctness, another important aspect of an algorithm is its performance. As stated above, converting (or rewriting) rules constitutes only one of three phases in the magic sets method. It is clear that the time taken to convert rules is very short compared to the time required to execute the magic rules. Thus, the performance of the algorithm for converting rules is not a major concern. However, to complete a story we will sketch it in the following. It is also useful to mention that the complexity analysis of the entire magic sets algorithm is not a simple task, and could easily become a subject of a separate paper.

Since our algorithm is based on the traversal of a binary tree (converted rule/goal graph), it is easy to show that the time complexity of the algorithm is O(n log n), where n is the number of nodes in the tree. We have seen that the number of nodes depends on the number of subgoals in database rules. Therefore, n is proportional to the number of subgoals.

CONCLUSION

We have shown how to construct a rule/goal graph for a given query and a set of rules. We have also shown the technique to generate magic rules

by using the resulting rule/goal graph. The analysis of the correctness and performance of our algorithm for generating magic rules has been presented. An interesting characteristic of the algorithm is that it is simple and clear and yet it manages to derive all and correct rules that are required to answer the query.

Another important characteristic of the algorithm is its time complexity. It has been shown that the time complexity of the algorithm is O(nlogn), which is an acceptable time.

REFERENCES

BANCILHON, F. and R. RAMAKRISHNAN. 1986. An amateur's introduction to recursive query processing strategies. In *ACM SIGMOD International Conference on Management of Data*, pp.16-52. New York: ACM.

BANCILHON, F., D. MAIER, Y. SAGIV and J. D. ULLMAN. 1986. Magic sets and other strange ways to implement logic programs. In *Proceedings Fifth ACM Symposium on Principles of Database Systems*, pp.1-15. New York: ACM.

BEERI, C. and R. RAMAKRISHNAN. 1987. On the power of magic. In *Proceedings Sixth ACM Symposium on Principles of Database Systems*, pp.269-283. New York: ACM.

CHANG, C. L. 1981. On the evaluation of queries containing derived relations in relational databases. In *Advances in Databases Theory*, ed. H. Gallaire, J. Minker and J.-M. Nicolas. Vol. I, pp. 235-260. New York: Plenum Press.

HENSCHEN, L.J. and S.A. NAQVI. 1984. On compiling queries in first-order databases. *Journal of ACM* **31(1)**: 47-85.

KIFER, M. and E.L. LOZINSKII. 1986. A framework for an efficient implementation of deductive database systems. In *Proceedings of the Sixth Advanced Database Symposium*, pp. 109-116.

MCKAY, D. P. and S. C. SHAPIRO. 1981. Using active connection graphs for reasoning with recursive rules. In *International Joint Conference on Artificial Intelligence*, pp.368-374.

REITHER, R.1978. On closed world databases. In *Logic and Databases*, ed. H. Gallaire and J. Minker, pp. 55-76. New York: Plenum.

SACCA, D. and C. ZANIOLO. 1987. Implementation of recursive queries for a data language based on pure Horn logic. In *Proceedings Fourth International Conference on Logic Programming*, pp.104-135. Cambridge Massachusetts: MIT Press.

ULLMAN, J. D. 1985. Implementation of logical query languages for databases. *ACM Transactions on Database Systems* **10(3)**: 289-321.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems*, Vol.II. Computer Science Press.

VIEILLE, L. 1986. Recursive axioms in deductive databases. The query/subquery approach. In *Proceedings First International Conference on Expert Database Systems.* pp.179-194.

APPENDIX

Implementation

This section describes the routines for transforming a set of rules into magic rules based on a traversal of a rule/goal graph. During the traversal of the rule/goal graph, we come across two types of nodes, i.e. goal nodes and rule nodes. To manipulate these nodes two routines, namely TraverseSubgoal and ProcessRuleNode, are needed. The former processes goal nodes and the latter processes rule nodes. However, the root node needs a special treatment, although it is one of the goal nodes. The routine that accesses and processes the root constitutes the top-level procedure called TraverseGraph. This procedure is presented as follows:

```
procedure TraverseGraph(root, query);
BEGIN
    /* generate an initialisation rule based on the root */
    print("m.", root ↑predicate);
    /* get bound arguments of the query */
    GetBoundArg(root ↑adorn, query);
    /* access to top-level rule nodes linked to the root */
    entry : = root ↑ptr;
    WHILE entry < > null DO
    BEGIN
        ProcessRuleNode(entry ↑ruleptr, root ↑adorn, null);
        entry : = entry ↑next;
    END
END
```

We assume that a record representing a goal node includes *predicate, adorn,* and *ptr* (pointer) fields. GetBoundArg is a function (routine) to obtain bound arguments of predicate in the query.

ProcessRuleNode

Given a rule node p, an adornment *adorn* for a predicate stored in a goal from which the node p comes out, and a *parent* (rule) node *parent* (exists if the node to be processed has an index i, j, j>0), the procedure *ProcessRuleNode* (p, *adorn, parent*) generates a rule defining a supplementary predicate $sup_{i,j}$, where i.j is an index associated with node p. The following algorithm presented in the form pseudocode defines *ProcessRuleNode* (p, *adorn, parent*).

```
BEGIN
    generate the head of a rule defining sup_ij;
IF j = 0 THEN
    generate the rule's body for supi.0;
ELSE
    generate the rule's body for supi.j, j>0;
IF p ↑golptr < > null THEN
    TraverseSubgoal(p ↑golptr,p);
IF p ↑rulptr < > null THEN
    ProcessRuleNode(p ↑arulptr,null,p);
ELSE
    generate a rule for an IDB predicate;
END
```

The procedure will be called from two places:

a) inside the procedure TraverseSubgoal, described later, and
b) inside the procedure ProcessRuleNode itself.

The call in (a) generates a rule for the zeroth supplementary predicate, and in this case the adornment *adorn* is required to determine the bound arguments of the rule head. The call in (b) generates a rule for the other supplementary predicate, and for this the above *adorn* is irrelevant.

TraverseSubgoal

This function is relevant only for a goal node repreenting an IDB predicate, for which we generate a rule defining a magic predicate. After generating such a rule, we then access and process all other nodes emanating from this goal node. We present TraverseSubgoal (p,parent), where p and parent represent the goal node and its parent (rule) node respectively, in the form of pseudocode as follows:

```
BEGIN
    IF p contains an IDB predicate THEN
        generate a magic rule;
    /* access to top-level rule nodes */
    entry : =  entry ↑ptr;
    WHILE entry < > null DO
        ProcessRuleNode(entry ↑rulptr, p ↑adorn, null);
        entry : = entry ↑next;
    END
END
```