MAGIC FUNCTIONS : A TECHNIQUE TO OPTIMIZE EXTENDED DATALOG RECURSIVE PROGRAMS

Georges GARDARIN

INRIA & University Paris VI
B.P. 105, 78153 Le Chesnay-Cédex (France)
UUCP :...!inria!litp!gg
work sponsored by ISIDE ESPRIT Project and PRC BD3

ABSTRACT

Several methods have been proposed to compile recursive Datalog programs. The most well-known perform a rewriting of rules using MAGIC or PROBLEM predicates in order to push selections before recursion. Rewritten rule systems are generally complex and difficult to translate into optimized relational algebra programs. Moreover, they often generate too many results; thus, the query must be applied to the generated results to eliminate non relevant answers. In this paper, after a survey of the existing compilation techniques which points out their limitations, we develop the **magic function method** introduced in [Gardarin-DeMaindreville86]. It is based on an understanding of the query as a function which maps columns of a relation to other columns. A query against recursive rules is then translated into a fixpoint functional equation. The resolution of this fixpoint equation using Tarski's theorem leads to efficient computation of the query answer. In particular, the derived algorithms push selections through recursion, because selections appear as function arguments. They generate only relevant answers to a given query, **without redundant data computation.** The purpose of this paper is the introduction of a generalized method to obtain and resolve the fixpoint functional equation. The method is general enough to handle non-binary rules, cyclic rules and function symbols. The main advantages of the method are : (1) It directly generates an optimized relational algebra program. (2) It performs a symbolic pre-computation which permits rule redundancy elimination. (3) It fully supports function symbols and range queries.

1. INTRODUCTION

Assuming the reader to be familiar with recursion in deductive databases [Gallaire84, Bancilhon86, Ullman86], we address the problem of evaluating queries referencing rule defined relations. We assume that the rules may include recursive predicates referencing unary, finite and inversible function symbols.

Two types of strategies have been proposed to handle recusive queries. The simplest one is based on query **interpretation.** This approach is mainly derived from backward chaining "à la Prolog" and works in a top-down manner. The derived methods generally push restriction before recursion [Vieille86, Lozinskii85], although certain methods do not work in all cases with function symbols [Gardarin-DeMaindreville85, Kifer-Lozinskii86]. As these methods do not pre-compile the queries, they generate call loops to the DBMS which are rather inefficient. However, a clever optimization of interpreted techniques known as query/sub-query has been developped at ECRC [Vieille86]. In this method, sub-queries and answers are kept in main memory to reduce costs.

Although the limits of the query/sub-query interpreted method are not well known, several researchers claims that the fanciest methods are based on a **compilation** of the query and rules before going to the database. This pre-compilation often rewrites the rule system using "magic" or "problem" predicates. These intermediate predicates simulate the moving up of constants before recursion, in such a way that a semi-naive bottom-up evaluation of the compiled rules presents two interesting features [Bancilhon85] :

(a) No redundant work is performed, that is, tuples are not produced twice using the same rules;

(b)No useless tuples are generated, that is, tuples are eliminated through restrictions as soon as possible.

Two compilation techniques claim to reach objectives (a) and (b) and are the most popular : the magic set approach and the Alexander method. We examine them in the first section of this paper. Using a simple example, we show that the Alexander method is among the most general ones. Also, the magic set approach has recently been extended (Extended supplementary magic sets [Beeri87]) toward a method which is similar to Alexander with a reordering of predicates. This reordering allows the algorithm to perform maximum sideways information passings to evaluate the query. A serious drawback of Alexander or Extended supplementary magic set is the complexity of the transformed rule system which generally, needs further optimizations and simplifications.

The motivations of this paper are three folds : (1) There is a need for a general method to compile a query against rules directly into an optimized relational algebra program performing selections before recursions; (2) Such a method must support rules with function symbols; (3) The method must provide a basic tool for simplifying certain classes of redundant rules. Steps toward such a method have already been taken [Chang81, Henschen84, Gardarin-DeMaindreville86, Ceri86]. The **magic function** approach developed here is a new step toward such an efficient compilation technique. More specifically, we propose a general and formally based method to translate a query and the associated rules into a functional equation. The method applies to any kind of rules which may include unary reversible function symbols.

The paper is organized as follows. The second section is mainly a survey. We summarize the Magic set and Alexander methods. We discuss their power. Then, we recall and precise the formalism presented in [Gardarin-DeMaindreville86] which consists in interpreting relations and queries as magic functions. Magic functions are set valued. In basic form, they map one column of a relation to another column. The fourth section is devoted to the description of the algorithm to translate a query over a set of rules into a fixpoint functional equation of the form $Q(X) = F\{Q(X)\}$, where Q is the query seen as a magic function. This algorithm computes symbolically the query answer in term of functions by performing a resolution of a system of simultaneous equations in the module of variables. The fifth section gives a differential algorithm to translate a magic function fixpoint equation directly into a relational algebra program. Finally, we show that the symbolic computation process allows the system to eliminate certain equivalent rules which are redundant. An example is given. In conclusion, we discuss the limits of the method.

2. A SURVEY OF SOME COMPILATION METHODS

2.1 The magic set method

The magic set approach [Bancilhon86] performs sideways information passing and then rewrite the rules using magic predicates. These predicates correspond to demons which reject useless tuples when applied in forward chaining. The basic magic set method is supported by a rather complex rewriting algorithm. The method does not apply to rules such as the odd ancestors derived from a base relation Parent(young,old) [Bancilhon86] :

(r1) Ancestor(x,y) <-- Parent(x,y)
(r2) Ancestor(x,z) <-- Parent(x,y),Ancestor(y,v),
Ancestor(v,z)

with query such as:
?Ancestor(c,z)

An extended version of the magic set algorithm called the **Generalized magic set** has just been proposed [Beeri87]. The algorithm first requires to build an adorned rule set using maximum sideways information passing (sideways information passing may be portrayed by a SIP graph [Beeri87]). In our case, we obtain :
(r1-a) $Ancestor^{bf}(x,y)$ <-- Parent(x,y)
(r2-a) $Ancestor^{bf}(x,z)$ <-- Parent(x,y),$Ancestor^{bf}(y,v)$,
$Ancestor^{bf}(v,z)$
For each recursive predicate R, a magic predicate MAGIC_R is created whose variables are bound variables in R. Each rule is then modified by the addition of the required magic predicate in its body. The generation of tuples in the magic predicate is given by : (a) the query; (b) rules which model the sideways information passing to the recursive predicate. In our examples, we obtain the rewritten rule system:
(r1-m) :
Magic_$Ancestor^{bf}(x)$, Parent(x,y) --> $Ancestor^{bf}(x,y)$
(r2-m) :
Magic_$Ancestor^{bf}(x)$, Parent(x,y), $Ancestor^{bf}(y,v)$,
$Ancestor^{bf}(v,z)$-->$Ancestor^{bf}(x,z)$
(sip-1) :
Magic_$Ancestor^{bf}(x)$, Parent(x,y) -->Magic_$Ancestor^{bf}(y)$
(sip-2) :
Magic_$Ancestor^{bf}(x)$, Parent(x,y), $Ancestor^{bf}(y,v)$ -->
Magic_$Ancestor^{bf}(v)$
(query) :
Magic_$Ancestor^{bf}(c)$.

The resulting modified rule system is rather complex (mutually recursive) and redundant (condition parts are repeated). The generalized supplementary magic set algorithm [Beeri87] has been proposed to save redundant condition parts : the result is approximately the Alexander method described below.

2.2 The Alexander method

The Alexander method [Rohmer85] consists in rewriting rules in terms of problems and solutions for each recursive predicate. Thus, as the Gordian node was cut in two by Alexander, the Alexander method cuts recursive predicates in two parts: the problem and the solution. The method also starts with an adorned rule set (adorned predicates are written with a boolean adornment of type R^{xxx}, where xxx is a binary vector; we keep here the notation of [Ullman86]). Then, the rewriting of a recursive rule B1,B2,...R,Q1,Q2,... --> R in several optimized rules is performed as follows:
(1) Add the problem predicate PB_R^{xx} to the rule condition.
(2) Propagate linearly the bound variables using sideways information passing up to an occurrence of the recursive predicate; this process is done by scanning the rule predicates, from the condition to the head; to guarantee a good sideways information passing, a reordering of predicates according to a maximum SIP is desirable.
(3) Cut the recusive predicate in two parts: (a) a problem predicate which is generated by the previously scanned condition part; (b) a solution predicate which is used to generate a new rule, with the non scanned part of the rule.
(4) Go on scanning the new rule with the same algorithm (i.e., go to 2).
As the algorithm generates independant rules which may share variables, context predicates are used to transmit variables between rules.

For example, using the definition of odd Ancestor given above with rules (r1) and (r2), we obtain:
(r1-m) :
$Pb\_Ancestor^{bf}(x)$, Parent(x,y) --> $Sol\_Ancestor^{bf}(x,y)$
(sip-1) :
$Pb\_Ancestor^{bf}(x)$, Parent(x,y) -->
$Pb\_Ancestor^{bf}(y)$,Cont1(x,y)
(sip-2) :
Cont1(x,y), $Sol\_Ancestor^{bf}(y,v)$-->
$Pb\_Ancestor^{bf}(v)$,Cont2(x,v)
(r2-m) :
Cont2(x,v), $Sol\_Ancestor^{bf}(v,z)$ -->$Sol\_Ancestor^{bf}(x,z)$
(query) :
$Pb\_Ancestor^{bf}(c)$,$Sol\_Ancestor^{bf}(c,y)$ --> Answer(y)

This set of rules must be applied using semi-naive forward chaining to get the query answer (i.e., Answer(y)). The rewritten rules are not in a very simple form : two intermediate predicates Cont1 and Cont2 are introduced, the rules are mutually recursive. However, the Alexander method appears here to be more successful than the classical magic set method. Indeed, it leads to results similar to those of the generalized supplementary magic set algorithm [Beeri87]. The Alexander method has been operational on PC since 1985. The rule transformer is written in Prolog [Rohmer86].

2.3 On the power of Magic sets and Alexander

Alexander and Magic sets are indeed very similar methods. However, generalized magic sets seems to select the maximum sideways information passing. The selected sideways information passing in Alexander depends on the order of the predicates in the rule body.

Magic sets do not generate rules which produce exactly the query answer. Generally, the generated rules produce more. For example, the above rewritten rules generate all ancestors of "c" as magic set (or problems). Therefore, all parents of an ancestor of "c" are produced in the ancestor relation. This is not correct because the answers are only the odd ancestor of "c". Fortunately, a simple final selection on the generated tuples in the ancestor relation eliminates the indesirable answers. It is indeed difficult to avoid this final selection even with simple linear rules. For example, the rules :

Ancestor(x,y) <-- Parent(x,y)
Ancestor(x,z) <-- Parent(x,y),Ancestor(y,v),Parent(v,z)

still generate the odd ancestors. The magic set for the query ?Ancestor(c,z) is still all ancestors of "c". Thus, the rewritten rules generate all ancestors of "c" in Ancestor, fortunately with their one level descendant as first attribute.

The Alexander method does not do better in the solution predicate; however, the final required selection is included in the rewritten rules: it derives from the query rule which generates the query answer.

Finally, it can be said that magic sets partly fail to generate only useful tuples. This point is often not very clear (see for example theorem 1 of [Bancilhon86]). One main feature of the magic function method is that it does not generate useless tuples in the result; thus, a final selection on the generated relation is not required.

2.4 The Magic function method

In [Gardarin-DeMaindreville86], we introduce a new method based on a functional approach : rules are rewritten as functional equations to compute a query answer. In the current paper which extends this functional approach, the method is called **magic functions.** The intuitive idea of our approach is that a relation instance defines functions; each function maps one set of values in one column to the corresponding set of values in another column. Rules are then rewritten as magic functions. In [Gardarin-DeMaindreville86], the rewriting algorithm is limited to binary predicates with acyclic conditions (i.e., chain rules). In the sequel, a generalization to any predicate with function symbols is going to be presented.

Let us give an intuitive view of the method using the previous example. For the odd ancestors as defined by rules (r1) and (r2) with the query ?Ancestor(c,z), the method leads to the fixpoint equation between the magic functions Ancestor(c) and Parent(c) which respectively maps a set of persons c to their ancestors and parents:

Ancestor(c)=Parent(c)+Ancestor(Ancestor(Parent(c)))

The fixpoint equation is used in a symbolic way to compute the solutions and then to derive the relational algebra program from the symbolic form. In our case, by successive approximations [Tarski's theorem] we derive :

$$Ancestor(c) = Parent(c) + Parent^3(c) + Parent^5(c) + ... + Parent^{2n+1}(c)$$

for some n which gives the fixpoint.
This formula leads to the following program :

Procedure Compute(Ancestor,c);
Begin
Delta := $\pi_{P.2}(\sigma_{P.1 \in c}(PARENT))$;
Ancestor := Delta;
while "Ancestor change" **do**
Delta := $\pi_{P.2}(\sigma_{P.1 \in Delta}(PARENT))$;
Delta := $\pi_{P.2}(\sigma_{P.1 \in Delta}(PARENT))$;
Ancestor := Ancestor $\cup$ Delta;
od;
End.

On this example, the functional approach appears much more simple and efficient than the Magic set or Alexander methods. Also, it is founded on mathematical principles. Unfortunately, the algorithm (based on a graph analysis) given in [Gardarin-DeMaindreville86] which translates rules into functional equations only applies to rules composed with binary predicates; moreover, variables in a rule body should not cycle; thus, the method seems only to apply to binary acyclic rules, also called chain rules. Another drawback of the method is that general symbolic forms are not always easy to compute at compile time. In the sequel, we propose a generalization of the method to handle any kind of rule with function symbols. We also propose a way to use the fixpoint equation which does not require computing a general polynomial form, although keeping the possibility of redundancy elimination. In short, we solve all the problems of the functional method. The resulting extended method, called magic function, is a powerful approach to recursive rule processing.

3. QUERIES AS FUNCTION COMPUTATION

In this section, we recall the precise definition of magic functions and we introduce a few more operations with magic functions.

3.1. Functions defined by a relation

Let R (A1, A2) be a relation in binary form where A1 and A2 are sets of one or more attribute(s). We denote dom (A1) (resp dom(A2)) the domain of A1 (resp. A2), that is in general the cartesian product of the domains of the composing attributes. A given instance of R determines derived functions called magic functions, defined as follows:

Definition 1 : Magic function
A magic function derived from a relation R(A1,A2) is a function which maps a subset of dom(Ai) into the related subset of dom(Aj) ($i \neq j$) according to R.

Indeed, for each possible binary form of a relation R(A1,A2), there exists two magic functions denoted r and r' which may be defined as follows :
(i) Let X = {x1, x2, ... xq} be a subset of dom(A1); r (X) is the subset of dom(A2) defined by
{yk / $\exists$ xp $\in$ X such that R(xp,yk) }.
(ii) Let Y = {y1, y2, ... yq} be a subset of dom(A2); r'(X) is the subset of dom(A1) defined by
{xp / $\exists$ yk $\in$ Y such that R(xp,yk) }.

Clearly, r (X) is obtained by restricting R to those tuples having for A1's value x1 or x2 or ... xn, keeping only the values of A2 as a set :

$$r(X) = \{\Pi_{Aj}(\sigma_{Ai = x1 \text{ or } ... \text{ or } Ai = xn}(R))\}.$$

To allow expression of boolean queries, we may add when necessary a virtual attribute A0 to each relation R. A0 is true for each tuple belonging to R and otherwise false.

A nice property of magic functions is their monotonicity. More precisely, we can demonstrate the following lemma.

Lemma 1 :

Relational functions are monotonic increasing.

Proof :

Let r be a magic function and X1 < X2 subsets of dom(Ai). We have :

$r(X1) = \{yk / \exists \ xp \in X1$ such that $R(xp,yk)\}$.

But, as any element of X1 is a member of X2, we have :

$\{yk / \exists \ xp \in X1$ such that $R(xp,yk)\} \leq \{yk / \exists \ xp \in X2$ such that $R(xp,yk)\}$;

which may be written as : $r(X1) \leq r(X2)$ ♦

3.2. Sum, composition and intersection of magic functions

For convenience and simplicity, the union of two sets is denoted + while the difference is denoted -. We shall use the following classical operations over functions and function results:

(i) The **sum** of two functions having the same domain is defined by:

$$(f+g)(X) = f(X) + g(X).$$

(ii) The **composition** of f and g is possible if the image domain of g is included in the definition domain of f ; it is defined by:

$$(g \circ f)(X) = g(f(X))$$

which is also denoted g.f(X).

(iii) The **intersection** of two functions having the same source and image domain is defined by :

$$(f \cap g)(X) = \{y \mid \exists x \in X : f(x) = g(x) = y\}$$

It is important to see that all introduced operations on magic functions preserve monotonicity.

Lemma 2:

Sum, composition and intersection of magic functions preserve monotonicity.

Proof:

The following equations are obvious to demonstrate:

$$(f+g)(X+Y) = (f+g)(X) + (f+g)(Y)$$
$$(g \circ f)(X+Y) = (g \circ f)(X) + (g \circ f)(Y)$$
$$(f \cap g)(X+Y) = (f \cap g)(X) + (f \cap g)(Y) \quad ♦$$

We could also use the difference operation between two functions defined as :

$$(f - g)(X) = f(X) - g(X)$$

Unfortunately, the difference does not preserve monotonicity.

4. A SYSTEMATIC METHOD TO GENERATE FUNCTIONAL FIXPOINT EQUATIONS

The basis of the magic function method is the rewriting of the rule system as a fixpoint functional equation. In this section, we propose a general approach to get such a fixpoint equation. The method is divided in four steps :

(1) The unification of each rule with the query and rectification of the variables to avoid common variables between rules.

(2) The rewriting of each rule in binary form using a sideways inforation passing strategy;

(3) The translation into a system of equations;

(4) The resolution of the system of equations to get the query fixpoint equation.

We shall illustrate the method with typical examples given below. The database is supposed to be composed of the following relations :

- PARENTS (YOUNG,OLD) abbreviated with predicate letter P;
- B (x1,x2,x3), C (y1,y2,y3) and D (z1,z2,z3).

To illustrate the generality of the method with range queries, we use a query of the form R(c,x), where c may be a constant or a set of constants (for example, c>10 is possible).

Definition of grandfathers :

GP(x,y) <-- P(x,z),P(z,y)
?--GP(c,y)

Quadratic definition of ancestors with functions [Kifer-Lozinskii86] :

A(x,y) <-- P(x,y)
A(f(x),g(y)) <-- A(x,z), A(z,y)
?--A(c,y)

Definition of a cyclic non-binary recursive relation R :

R(x,y,z) <-- B(x,y,z)
R(x,y,z) <-- C(x,t,z),R(t,u,y),D(u,z)
?--R(c,y,z)

4.1. Unification with the query and rectification

This step is simple. Each rule head is unified with the query to evaluate. A straightforward propagation of the query constant(s) is performed in the rule body. From now on, the constant(s) will be considered as a formal parameter. To avoid confusion of variables between different rules, all of them are indexed with the rule number : this process of renaming variables with different names is called rectification.

We illustrate this step with the results of the previous examples :

Definition of grandfathers :
GP(c,y1) <-- P(c,z1),P(z1,y1)

Quadratic definition of ancestors with functions
A(c,y1) <-- P(c,y1)
A(c,y2) <-- A(f'(c),z2), A(z2,g'(y2))
Note that unification implies here the use of the inverse fonction of f and g, denoted f' and g', which are supposed to exist.

Definition of a cyclic non-binary recursive relation R :
R(c,y1,z1) <-- B(c,y1,z1)
R(c,y2,z2) <-- C(c,t2,z2),R(t2,u2,y2),D(u2,z2)

4.2. Rewriting rules in binary form

Each rule is first rewritten in binary form defined as follows.

Definition 2: Binary rule form
Rule form in which:
(i) Variables in the head predicate are divided into two groups, the first one designating the known constants, the other representing the remaining variables.
(ii) Variables in each condition predicate are also divided in two groups, the grouping being done according to a chosen sideway information passing strategy.
(iii) Conditions are added to the rule body using projection functions if constraints need to be kept between variables.

Variables are grouped using a SIP graph, as defined in [Beeri87]. Different choices are possible and will lead to different fixpoint equations. To perform as many selections as possible before recursion, it is desirable to use a complete SIP, that is a SIP which achieves all possible migrations of values between predicates. Finally, after rewriting, the most general form of a rule is :

R(c,x) <-- P1(y,z),P2(t,u),....Pn(v,w),
Ω(c,x,y,z,t,u...,v,w)

where :
- c is a constant or a tuple of constants;
- y,z,t,u,...,v,w are tuple variables or constants; in practice, we denote them with the text string of the domain variable names concatenated;
- Ω(c,x,y,z,t,u...,v,w) is a conjonction of atomic conditions between constants, variables or functions applied to variables, including the projection function.

We illustrate the binary canonical form with the last example which is the only one not in binary form.:
The first rule is simply rewritten as :
R(c,y1z1) <-- B(c,y1z1)
The rewriting of the second rule :
R(c,y2,z2) <-- C(c,t2,z2),R(t2,u2,y2),D(u2,z2)
is done using the following SIP :
$R(c,y2,z2) \rightarrow_{c} C(c,t2,z2)$
$C(c,t2,z2) \rightarrow_{t2} R(t2,u2,y2)$
$R(t2,u2,y2) \rightarrow_{u2} D(u2,z2)$
Thus, we obtain :
R(c,y2z2) <-- C(c,t2z2),R(t2,u2y2),D(u2,z2), $\pi_1(t2z2)=t2$, $\pi_2(t2z2)=\pi_2(y2z2)$, $\pi_2(t2z2)=z2$, $\pi_1(u2y2)=u2$, $\pi_1(y2z2)=\pi_2(u2y2)$
It is important to be sure that the transformation is valid, that is that the reverse transformation remains possible (no condition must be lost).

4.3 Translation of binary form rules into a system of functional equations

Let us assume a binary predicate R defined by a unique rule:

R(c,x) <-- P1(y,z),P2(t,u),....Pn(v,w),
Ω(c,x,y,z,t,u...,v,w)

where P1, P2 ...Pn are binary predicates. Each predicate R, P1, P2,... Pn defines two magic functions as introduced in section 3. The binary form rule may be interpreted as a set of functional equation definitions, as follows (capital variables are set variables) :

if
p1(Y) = Z (or p1'(Z) =Y)
p2(T) = U (or p2'(U) =T)
.....
pn(V) = W (or pn'(W) = V)
Ω(C,X,Y,Z,T,U...,V,W)
then
r(C) = X

Thus, at saturation point of the relation computation, the following functional system of equations must be satisfied (capital letters are used to denote set variables):

p1(Y) = Z (or p1'(Z) =Y)
p2(T) = U (or p2'(U) =T)
.....
pn(V) = W (or pn'(W) = V)
Ω(C,X,Y,Z,T,U...,V,W)
r(C) = X

Let us point out that for each line with an "or" in parenthesis, both equations can be used. Although, in general $p'(p(X)) \neq X$; however, for computing a query answer, we may use $p(X) = Y$ or $Y = p'(X)$ depending on which variable is bound. Thus, when resolving equations, we may use one or the other form : this is due to the fact that we want all answers to queries. Thus, we shall no longer write the two possible forms, but only the one which is required to solve the query and assume that the inverse function of p is p'.

In the case of several rules defining a predicate R, each of them contributes to part of the answer, as follows :

R(c,x1) ← ...
R(c,x2) ← ...
....
R(c,xk) ← ...

As the union of all rule fixpoints must be performed to get the R relation, we must collect all the functional expressions. We have then to compute:

$$r(C) = X1 + X2 + ... + Xk$$

with the equations deriving from the rule bodies. For this purpose, the whole system must be solved in r(C) by successive elimination of the Xi variables, using an algorithm to solve a system of equations.

4.4 Solving the system of equations

Let us now assume a query R(c,y). As seen above, we must evaluate the function r(C). To do this evaluation , we may use the set of equations which is derived from the rules as explained above :

$r(C) = X1 + X2 + Xn$
$p1(Y1) = Z1$
$p2(T1) = U1$
.....
$pn(V1) = W1$
$\Omega(C,X1,Y1,Z1,T1,U1...,V1,W1)$
.....

This system of equations must be solved in r(C). The theory of simultaneous equation solving is well known, so we are not going to develop it here. However, we would like to mention that considering the functions as scalars, the vectors of variables (X1,X2,...) describe a module (i.e., a vector space with a non commutative multiplication of scalars). A general approach to solve equations in such a structure is the elimination of variables by substitution (i.e., Gaussian elimination in a vector space). In general, a unique solution of the form $r(C) = F(C)$ is obtained, where F is an expression of the union, intersection and composition of database functions (i.e., database relations seen as functions). Thus, the answer to the query may be simply computed using the expression F.

Let us give an example using the grandfather definition :
GP(c,y1) <-- P(c,z1),P(z1,y1)
We obtain, in functional form :
$gp(C) = Y1$
$p(C) = Z1$
$p(Z1) = Y1$
This system of three equations with three variables (C is considered as a formal variable) may be solved in C by simple elimination of Y1 and Z1, which entails :

$$gp(C) = p(p(C)).$$

This functional equation tells us that to get the grandfather of a set of persons C = {c}, we must take the parents of the parents of each member of {c}. It is an efficient method to compute the grandfather of a set of persons. The method to construct the functional equation is general enough to deal with recursive rules, function symbols and non-binary predicates.

Certain rules lead to more equations than variables. Let us assume that we get n equations and p variables with $p < n$. Relaxing n - p equations, a first solution of type $r(C)=F1(C)$ may be calculated. Using successively the other equations (which are indeed constraints for F1(C)) and relaxing a previously used equation, other solutions may be evaluated :

$$r(C) = F2(C), ... r(C) = Fn(C).$$

As all equations (which may be seen as integrity constraints on the relation composed of the cartesian product of the variables) must be satisfied, we have :

$$r(C) = F1(C) \cap F2(C) \cap ... \cap Fn(C).$$

On the contrary, certain rules lead to less equations than variables. Such rules do not restrict certain variables enough. Two examples of such rules are :
GP(x,y) <-- P(x,z), P(t,y)
and :
GP(x,t) <-- P(x,y),P(y,z)
The first one leads to a cartesian product. The second one is unsafe because t is not defined in the premises. In the sequel, we shall reject such rules which are dangerous and useless.

4.5. A recursive example with function symbols

The case of recursive rules is not a special one: a recursive rule contributes to a query in functional form as a normal one. However, the generated functional equation appears to be a fixpoint equation [Gardarin-DeMaindreville86]. Due to a lack of space, we cannot treat all given examples here. The reader may find more examples in [Gardarin87]. Let us solve the example of the quadratic definition of ancestors with functions; after unification and rectification, we obtain the binary form :

A(c,y1) <-- P(c,y1)
A(c,y2) <-- A(f'(c),z2), A(z2,g'(y2))

Transforming the above rules in functional equations yields (note that f and g, or their reverse are applied to sets of values):

a(C) = Y1 + Y2
p(C) = Y1
a(f'(C)) = Z2
a(Z2) = g'(Y2)

This is a system of four equations with four variables (C, Y1, Z2, Y2). Solving it in a(C) by elimination of variables Y1, Y2 and Z2 entails :

$$a(C) = p(C) + g(a(a(f'(C))))$$

5. TRANSLATION OF FIXPOINT EQUATIONS INTO RELATIONAL ALGEBRA

5.1 Iterative computation

For a given query Q = R({c},y), the previous method derived a fixpoint functional equation of the form :

$$q = F(q)$$

Applying Tarski's theorem on the q function lattice [Guessarian87], we may solve this equation by successive approximation as follows (ø is the function whose result is always empty) :

$q = F(ø) \cup F(F(ø)) .... \cup F^n(ø)$

Thus, the query answer is given by (we replace q by its real functional form with argument {c}) :

$r(\{c\}) = F(ø)(\{c\}) \cup F(F(ø))(\{c\}) .... \cup F^n(ø)(\{c\})$

and we know that the fixpoint is reached for a certain n where no new data can be produced.

Thus, we can write an iterative program to compute the query answer as follows,where F is the fixpoint equation right member (a text string of symbols), {c} the constant(s) in the query (i.e., the restriction criteria), and R the query answer :

```
Procedure compute(R,F,{c});
   Begin
{Initialize the symbolic form of the answer}
        RSymb := F(ø);
        DeltaSymb := ø;
{Initialize answer with F(ø)(c)}
        RData := RSymb({c});
        DeltaData := ø;
{Compute symbols and data up to fixpoint }
        while "Elements inside DeltaData and
              DeltaSymb change" do
   {Compute next symbolic form = old∪F^n(ø)}
              NewRsymb := Rsymb ∪ F(Rsymb);
   {Compute variations of symbolic form}
              DeltaSymb := NewRsymb - Rsymb;
   {Compute new data generated }
              DeltaData := DeltaSymb({c});
   {Cumulate answers }
              RData := RData ∪ DeltaData;
   {Move to next step }
              Rsymb := NewRsymb;
        od;
   End;
```

This program performs an iterative symbolic generation of the function $F^n(ø)$ from F. At each step, it queries the database to compute $F^n(ø)(\{c\})$; to avoid generating several time the same queries, we eliminate already evaluated symbols in the symbolic functional form of the answer: this explains the difference between the new string of symbols and the old one. The program stops when no new data are generable (or when no new functional expressions, i.e. strings of symbols, are generated: the inclusion of DeltaSymb in the test is a slight optimization which may be omitted). It is important to stop only when all elements included in DeltaSymb({c}) do not change : the stopping test may require to memorize values for each function evaluated in DeltaSymb if one wants to support complex general rules.

For example, with the fixpoint equation :

a(C) = p(C) + g(a(a(f'(C))))

the quadratic ancestor problem, F is set to p+gaaf. At the first iteration, we obtain Rsymb:=p+gppf
At the next iteration, we obtain:

p+gppf+gpgppff+ggppf'gppff, ...

The algorithm successively queries the database to evaluate p(C), then g(p(p(f'(C)))), then g(p(g(p(p(f'(f'(C)))))), then g(g(p(p(f'(g(p(p(f'(f'(C))))))))))....Indeed, the general form does not need to be computed as the program queries

the database to compute Deltadata and stops when no new data appears in the argument of each new applicable function. This is really a good way of mixing symbolics and data computation, although the stopping criteria might be a bit complex. That is possible as any symbol in Rsymb represents a base relation or a base function. Moving back to relational algebra with functions [Zaniolo85], which is simply a re-interpretation of the symbolic formulas, yields a simple program which performs restriction at first: this is due to the fact that constants are the arguments of the magic function strings which are symbolically computed.

5.2 Simplification of general forms by symbolic computations

Indeed, the fixpoint functional equation is a very useful tool which may be used not only to generate relational algebra programs, but also to simplify rule system computation. Let us assume the following system of rules:

A(x,y) <-- P(x,y)
A(x,y) <-- P(x,z),A(z,t),A(t,y)
A(x,y) <-- A(x,z),A(z,t),A(t,y)
with the query :
?--A(c,y).

Magic functions lead to the following fixpoint equation :
$a(C) = p(C) + a(a(p(C))) + a(a(a(C)))$.
A symbolic fixpoint computation yields :
$a(C) = p(C) + p^3(C) + p^5(C) + ... + p^{2n+1}(C)$
which is indeed the form obtained with the two first rules. Thus, magic functions may be used to simplify duplicate computations due to redundant rule systems. Indeed the previous algorithm performs this simplification as, at each iteration step, it eliminates duplicate strings of symbols in the symbolic form of the answer (RSymb). We do not know of any method which is able to perform such rule simplification. This is indeed part of the power of magic functions.

6. CONCLUSION

In this paper, we presented a systematic method for compiling a large class of recursive queries into fixpoint equations. The method is based on a translation of each rule into a system of functional equations using the so-called magic functions. This system is solved in the module space of vector variables. The method is general enough to reject problematic rules. The method also applies to non-recursive queries : in that case, it determines the query answer as a functional equation in which all functions are derived from base relations. In the case of recursion, the fixpoint equations are directly translated into an iterative relational algebra program which is computed in a symbolic way, using Tarski's fixpoint theorem. Although a general polynomial form is not always possible, by mixing symbolic computation and data computation, it appears that the method always performs selection at first and also simplifies query computation programs generated by redundant rules. In other words, a large class of equivalent rules are detected and computed only once.

Finally, it is important to point out that all Datalog programs with unary function symbols can be optimized and translated into functional form using the proposed method. The concerned class of rules includes all the linear or non linear rules, stable or non-stable rules, chain or non chain rules, binary or non-binary rules as defined by other authors. It may also handle mutual recursion as shown in [Gardarin-DeMaindreville86]. A slight problem arises with functions of multiple variables. We think that this problem might be handled with a good choice of variables which would transform n-ary functions into unary ones. A lot of work remains to be done, for example to compare different possible fixpoint equations, to determine the power and limits of a symbolic computation of the query answer, ... Neverthless, fixpoint computation of magic functions is a very powerful method for recursive query compilation and optimization, allowing the compiler to determine certain redundant rules.

REFERENCES AND BIBLIOGRAPHY

[Aho-Ullman79] AHO A.V., ULLMAN J.D. : "Universality of data retrieval languages", Conf. of POPL , San-Antonio, Texas, 1979.

[Bancilhon85] BANCILHON F. : "Naïve evaluation of recursively defined predicates", MCC internal report, 1985.

[Bancilhon86] BANCILHON F., MAIER D., SAGIV Y., ULLMAN J.D. : "Magic sets and other strange ways to implement logic programs", 5th ACM Symposium on Principles of Database Systems, Cambridge, 1986.

[Bancilhon-Rama86] BANCILHON F., RAMAKRISHNAN R. : "An Amateur's Introduction to Recursive Query Processing Strategies", ACM SIGMOD'86, Washington D.C., May 1986.

[Beeri87] BEERI C., RAMAKRISHNAN R. : "On the Power of Magic", MCC Technical Report, Jan. 1987.

[Ceri86] CERI S., GOTTLOB G., LAVAZZA L. : "Translation and Optimization of Logic Queries: The Algebraic Approach" 12th Very Large Data Bases, Kyoto, 1986, Pp; 395-402.

[Chandra82] CHANDRA K.A., HAREL D. : "Horn clauses and the fixpoint query hierarchy", Proc. 1st ACM Symposium on Principles of Database Systems, 1982.

[Chang81] CHANG C. : "On evaluation of queries containing derived relation in a relational database", in [Gallaire81].

[Delobel86] DELOBEL C. : "Bases de Données et Bases de Connaissances : Une Approche Systemique à l'aide d'une Algebre Matricielle des Relations", Journees Francophones, Grenoble, 1986.

[Gallaire81] GALLAIRE H., MINKER J., NICOLAS J.M.: "Advances in database theory", Book, Vol.1, Plenum Press, 1981.

[Gallaire84] GALLAIRE H., MINKER J., NICOLAS J.M.: "Logic and databases : a deductive approach", ACM Computing Surveys, Vol. 16, N° 2, June 1984.

[Gardarin-DeMaindreville85] GARDARIN G., DE MAINDREVILLE C., SIMON E. : "Extending a Relational DBMS towards a Rule Based System: An Approach Using Predicate Transition Nets"
Crete Workshop on DB and AI, June 1985, To appear in a book, Springer-Verlag, Thanos and Schmidt Ed.

[Gardarin-DeMaindreville86] GARDARIN G., DE MAINDREVILLE C. : "Evaluation of Database Recursive Logic Programs as Recurrent Function Series", ACM SIGMOD'86, Washington D.C., May 1986.

[Gardarin-Pucheral86] GARDARIN G., PUCHERAL P. : "Optimization of Generalized Recursive Queries Using Graph Traversal", Internal Report, INRIA 86, Submitted for publication.

[Gardarin87] GARDARIN G : "Les fonctions Magiques : Une Technique pour Optimiser les Règles récursives", Journées Bases de Données PRC BD3, Port-Camargue, France,1987, INRIA Ed.

[Gardarin-Guessarian-De Maindreville] GARDARIN G, GUESSARIAN I., DE MAINDREVILLE C. : "Translation of rules in fixpoint equations", in preparation.

[Guessarian87] GUESSARIAN I. : "Some fixpoint Techniques in Algebraic Structures and Application to Computer Science", to appear in INRIA-MCC Workshop 1987.

[Henschen-Naqvi84] HENSCHEN L.J., NAQVI S.A. : "On compiling queries in recursive first-order databases", JACM, Vol. 31, N° 1, Jan. 1984.

[Lozinskii85] LOZINSKII E.L. : "Evaluation queries in deductive databases by generating subqueries", IJCAI Proc., pp. 173-177, Los Angeles, August 1985.

[Kifer-Lozinskii86] LOZINSKII E., KIFER M. : "Implementing Logic Programs as a Database System", to be published, Data Engineering Conference, 1987.

[Marque-Pucheu83] MARQUE-PUCHEU G. : "Algebraïc structure of answers in a recursive logic database", Rapport Ecole Normale Supérieure, 1983.

[Merrett84] MERRETT T.H. : "Relational Information Systems", Book, Prentice Hall, 1984, Chapter 5.

[Rohmer85] ROHMER J., LESCOEUR R. : "La méthode d'Alexandre : une solution pour traiter les axiomes récursifs dans les bases de données déductives", Rapport de recherche, Bull, DRAL/IA/45.01.1985.

[Sacca-Zaniolo86] SACCA M., ZANIOLO C. : "On the Implementation of a Simple Class of Logic Queries for Databases", 5th ACM Symposium on Principles of Database Systems, Cambridge, 1986.

[Tarski55] TARSKI A. : "A lattice theoretical fixpoint theorem and its applications", Pacific journal of mathematics, N° 5, pp. 285-309, 1955.

[Ullman86] ULLMAN J.D. : "Implementation of logical query languages for Databases", ACM TODS, Vol. 10, N. 3, pp. 289-321,1986.

[Vieille86] VIEILLE L. : "Recursive axioms in deductive databases : the query sub-query approach", Proc. First Intl. Conference on Expert Database Systems, Charleston, 1986.

[Zaniolo86] ZANIOLO C., SACCA M. : "Implementing Recursive Logic Queries with Functions Symbols", Unpublished Manuscript, MCC Report, April 1986.