TRANSLATION AND OPTIMIZATION OF LOGIC QUERIES: THE ALGEBRAIC APPROACH

S. Ceri ('), G. Gottlob ("), L. Lavazza (^)

(') Dipartimento di Elettronica, Politecnico di Milano, Italy
(") Istituto per la Matematica Applicata del CNR, Genova, Italy
(^) TXT-Techint Software e Telematica, Milano, Italy

ABSTRACT

This paper presents an algebraic approach to translation and optimization of logic queries. We first develop a syntax directed translation from rules of function-free logic programs to algebraic equations; then we show solution methods for independent equations and for systems of interdependent equations. Such solutions define the operational and fixpoint semantics of function-free logic programs and queries. We also present algebraic optimization methods for "top-down" and "bottom-up" strategies; the former are useful if no initial binding is provided with the query, while the latter are useful if some arguments of the query are bound to constant values.

1. INTRODUCTION

In recent times, the combination of relational databases and logic programming (LP) has become a popular argument of research. The application of LP as query language of a relational database entails a relevant enrichment of the expressive power of traditional query languages; hence the database community looks at LP as a promising approach for posing complex (e.g. recursive or deductive) queries. At the same time, databases provide the technology for processing large collections of data in an efficient way, hence solving many of the problems posed by LP applications when they manage large amounts of information.

The efficient implementation of logic queries has been discussed in many recent papers, including [Bancilhon86, Chandra82, Ceri86 Henshen84, Marque-Pucheu84, Sacca'86, Ullman85]. Major research directions are:

a. Designing "pure" LP languages, e.g. languages which do not incorporate procedural features, such as the dependency of the computation from the order of clauses and the use of special predicates. This trend has marked the adoption of "pure" Horn clauses as "standard" LP language, and consequently a certain resistence to Prolog.

b. Determining how function-free Horn Clauses can be efficiently executed. New formal models, such as "rule-goal" graphs, describe binding propagation among clauses. Several rules for "capturing" nodes of graphs have been defined; capturing a node is equivalent to evaluating the information associated to the corresponding clause or rule, deduced from the database.

c. Determining how LP programs can be made more efficient, by operating transformations from LP to LP. Tecniques such as the "magic set", the "counting" and the "Eager" methods have been developed to this purpose.

We have focused our attention on the use of relational algebra at work on the same problems. While the idea of using relational algebra for executing logic queries is not new (see, among others, [Aho 79] and [Marque-Pucheu 84]), the major contribution of this paper is to give a systematic overview of how traditional algebra, extended by a closure operator, can be applied to solve logic queries.

2. MODELS FOR LOGIC PROGRAMMING AND RELATIONAL DATABASES

In this section, we give our definition and interpretation of function-free logic programs and queries; then, we introduce positive algebra extended with the closure operator.

2.1 Function-free logic programs

A function-free logic program (FFLP) is a set of definite, function-free Horn clauses (i.e. clauses that contain exactly one positive literal); we use a Prolog-like syntax for clauses, which have their positive literal on the LHS and zero or more (negative) literals on the RHS, in conjunctive form. For instance, the following is a FFLP:

```
S(a,b).
R(c,b).
P(X):- P(Y), R(X,Y).
P(X):- S(Y,X).
Q(X,Y):- P(X), R(X,Z), S(Y,Z).
```

We can better interpret a FFLP within the framework of databases if we consider terms appearing in the LHS of clauses as either database relations or computed relations. Ground clauses of the former are stored in an extensional database EDB; ground clauses of the latter are evaluated by executing the FFLP; thus, each computed relation appears as LHS of one or more clause. In the above example, R and S are database relations, P and Q are computed relations; the first two clauses are equivalent to assigning an instance to relations R and S, and in general will not be present in FFLPs but will be stored in the EDB.

We generalize the class of database relations to include all those terms for which we have a function available for evaluating arithmetic predicates. For instance, the relation PLUS(X,Y,Z)

Proceedings of the Twelfth International Conference on Very Large Data Bases **Kyoto, August, 1986**

links all representable integers X, Y, and Z such that X+Y=Z. Further, we can extend Horn clauses to include special terms such as equality between variables (e.g., X=Y), since we assume that the equality test be available, and hence we can represent it through a special relation EQ(X,Y). We assume that the domain and co-domain of functions be finite.

Each variable X of a clause of a FFLP program is associated to a finite range Rg(X). Ranges of variables of database relations are limited to the values existing in the instance of that relation or to the domain and co-domain of functions; all variables of computed relations are limited to a unique finite universe U which includes all individuals possibly occurring in the DB.

An input goal to a FFLP is a clause consisting of a single literal, for instance:

```
?-Q(a,X).
```

The solution of an input goal P(t1,...,tn) is the set of all ground instances of P(t1,...,tn) which are logical consequences of the FFLP; for example:

```
Sol(FFLP, Q(a,X)) = {Q(a,c) | (c ∈ U) ∧
                     (FFLP U EDB => Q(a,c))}
```

If the input goal is a literal with all places bound to constant values, then the solution space is reduced to the answers "yes" or "no":

```
Sol(FFLP, P(c)) = "yes" if (c ∈ U)) ∧
                          (FFLP U EDB => P(c)),
                  "no" otherwise.
```

2.2. Positive algebra with closure

We assume that the reader is familiar with the relational model and algebra as from [Ullman82]. Positive relational algebra (RA+) includes as primitive operators selection (σ), projection(π), cartesian product(x), join($\bowtie$), semi-join($\ltimes$), and union (U); noticeably, it does not include the difference operator. We extend projection to include π_{ϕ} E (projection on the empty set) as an operator which can be applied to an algebraic expression E; π_{ϕ} E returns "yes" if E $\neq \phi$, "no" if E $= \phi$. ϕ denotes the empty relation of suitable degree.

The closure operator (0^X) is applied to an expression E(X) of a (variable) relation X, provided that the schema of the result of E(X) is the same as the schema of X. The language obtained by extending RA+ with the closure operator is denoted as ERA+. The operational definition of the closure operator is given by the following program A_0 which computes $0^X E(X)$:

```
| A0(E(X)):
|   S <- φ
|   REPEAT
|     Y <- E(S)
|     S <- S U Y
|   UNTIL Y=S
|   RETURN S
```

The fixpoint semantics of the closure operator is also well defined. Let P denote the set of all possible relations having the same domain as X:

```
P = powerset (Dm(X))
```

Then, P is a complete lattice under the union and intersection operations, having as minimal element the empty relation and as maximal element Dm(X).

Expressions of RA+ are monotone: (i.e. X $\subseteq$ Y => E(X) $\subseteq$ E(Y)), hence algorithm A_0 always terminates after a finite number of iterations, and 0^X E(X) is the minimal solution (i.e. the least fixpoint) of the equation X=E(X):

$$0^X E(X) = \min \{ X \in P \mid X = E(X) \}$$

3. SYNTAX DIRECTED TRANSLATION FROM FUNCTION - FREE LOGIC PROGRAMS TO EXTENDED RELATIONAL ALGEBRA

A syntax-directed translation algorithm maps each clause of a FFLP into a correspondent disequation of RA+. Disequations are subsequently interpreted as equations under the Closed World Assumption (CWA) and solved using the closure operation.

3.1 Translation of individual clauses

Let a generic Horn clause of a FFLP have the following structure:

R: $P(\alpha_1,\ldots \alpha_n) :- Q_1(\beta_1,\ldots \beta_k), \ldots, Q_h(\beta_s,\ldots \beta_m)$

Then, the translation associates to R an algebraic disequation:

$$Expr(Q_1, \ldots Q_n) \subseteq P$$

where P is a computed relation correspondent to predicate P, and similarly Q_i are either computed or database relations correspondent to predicates Q_j. The specification of the algorithm for the syntax directed translation requires defining some useful notation and two rewriting functions.

- occurs(α_i,RHS), of sort boolean, is "true" if the term α_i belongs to the RHS, "false" otherwise.
- corr(i) denotes the function returning, for each α_i of the LHS, the index j of the leftmost variable β_j of the RHS such that $\beta_j = \alpha_i$, if such a variable exists.
- const(α_i) of sort boolean, is true if α_i is a constant, false otherwise; const(β_j) is similarly defined.
- var(x_i), of sort boolean, is defined as "not const(x_i)".
- newvar(x) is a procedure returning a new variable name at each invocation.
- Let E denote a string, x and y denote symbols. Then E<x,y> denotes a new string in which the first occurence of x is substituted by y.

The syntax directed translation of R into an algebraic disequation of RA+ is defined through two recursive rules which apply to the LHS and RHS of R respectively.

The first rule deals with three special cases:

a. bindings to constant values in the LHS;

b. multiple occurrences of the same variable in the LHS;

c. existence of a variable of the LHS that does not occur in the RHS.

Each such special case is reconducted, by suitable string transformations, to an equivalent case in which all positions of the LHS are bound to distinct variables. Then, the second rule is applied to the RHS; it simply generates all selection conditions due to constant bindings or replicated variables within the RHS, and then builds a cartesian product with all database or computed relations corresponding to terms of the RHS.

After applying the second rule, equivalence - preserving transformations can be applied to the resulting expression in RA+ to transform cartesian products into joins and to propagate selections to their operands, in the conventional way (see [Ullman82]). The notation $X_{i=1..k} Q_i$ indicates the cartesian product of relations $Q_1, \ldots Q_k$.

rule 1.

```
T(R)=if ∃ i : const(α_i)    /*a*/
     then
      newvar(x)
      return T(LHS<α_i,x>:-RHS, EQ(x,α_i))
     elsif ∃ i,j : α_i=α_j, i<j   /*b*/
      then
       newvar(x)
       return T(LHS<α_i,x>:-RHS, EQ(x,α_j)
     elsif ∃ i: var(α_i) ∧ not occurs(α_i,RHS) /*c*/
       then return T(LHS:-RHS, Rg(α_i))
     else return Π_corr(1),...corr(n) T'(RHS)
     end if;
```

rule 2.

```
T'(R) = if ∃ i : const(β_i) /* a */
         then
           newvar(x)
           return σ_i=β_i T'(RHS<β_i,x>)
         elsif ∃ i,j : β_i=β_j, i<j   /* b */
            then
              newvar(x)
              return σ_i=j T'(RHS<β_i,x>)
         else return (X_i=1..k Q_i)
        end if;
```

Example. The following is a systematic application of the translation algorithm.

P(X,X,Z):-S(X,Y)R(Y,a,Z)

1. by T, case (b)
T(P(N1,X,Z):-S(X,Y),R(Y,a,Z),EQ(N1,X))

2. by T, last recursive call
$\Pi_{6,1,5}$ T' (S(X,Y),R(Y,a,Z),EQ(N1,X))

3. by T', cases (a) and (b)
$\Pi_{6,1,5} \sigma_{4=a \wedge 2=3 \wedge 1=7}$ T'(S(N2,N3),R(Y,N4,Z),EQ(N1,X))

4. by T', last recursive call
$\Pi_{6,1,5} \sigma_{4=a \wedge 2=3 \wedge 1=7}$ (S x R x EQ)

5. After pushing selection and join conditions
$\Pi_{6,1,5}$ ((S $\bowtie_{2=1}$ ($\sigma_{2=a}$R)) $\bowtie_{1=1}$ EQ)

6. Final disequation:
$\Pi_{6,1,5}$ ((S $\bowtie_{2=1}$ ($\sigma_{2=a}$R)) $\bowtie_{1=1}$ EQ) $\subseteq$ P

3.2. Transformations from disequations to equations in RA+

By effect of the translation rules explained above, we can turn each FFLP into a set of algebraic disequations. For instance, the last 3 clauses of the FFLP in Section 2.1 generate:

Π_2 (P $\bowtie_{1=2}$ R) $\subseteq$ P

Π_2 S $\subseteq$ P

$\Pi_{1,4}$ ((P $\bowtie_{1=1}$ R) $\bowtie_{3=2}$ S) $\subseteq$ Q

The Closed World Assumption (CWA, [Reiter78]) enables us to turn these disequations into equations. By the CWA, all facts which cannot be deduced by the application of the FFLP to the database are false; hence, no fact about a generic term P can be proved other than through existing disequations; hence the union of all RHS of disequations gives the algebraic equation required to compute P. The above example generates the system of equations:

P = Π_1 (P $\bowtie_{1=2}$ R) U (Π_2 S)

Q = $\Pi_{1,4}$ ((P $\bowtie_{1=1}$ R) $\bowtie_{3=2}$ S)

Marque-Pucheu et al. [Marque_Pucheu84] show a transformation from logic programs into equations which does not directly use relational algebra.

3.3. Solution of independent equations

Solving a system of equations is easier if each equation is independent from the others (or can be made independent by suitable substitutions); an equation is independent when it does not contain computed relations in the RHS other than the one in the LHS. For the solution of independent equations, two cases are given:

a. The RHS contains only database relations; in this case, the solution of the equation is simply given by the evaluation of the expression in RA+. This happens when rules for the computed relation in the FFLP are nonrecursive.

b. The RHS contains one or more occurrences of the computed relation CR; in this case, the definition of the closure operator as fixpoint of algebraic equations enables us to build the solution as follows:

sol(CR) = 0^{CR} RHS(CR)

In the above example (Sect. 3.2), the first equation is independent, and can be solved as:

sol(P) = 0^P (Π_2(P $\bowtie_{1=2}$ R) U (Π_2 S))

The second equation for Q depends on P, however we can suspend its evaluation until we have solved the equation for P. Then we consider P as a fixed (i.e. database) relation, and the second equation falls in case (a) above:

sol(Q) = $\Pi_{1,4}$ ((sol(P) $\bowtie_{1=1}$ R) $\bowtie_{3=2}$ S)

We can now interpret input goals as suitable expressions on the algebraic solutions; for instance:

?-P(X). <=> sol(P)

?-P(a). <=> $\Pi_\phi \sigma_{1=a}$ sol(P)

?-Q(a,X). <=> $\sigma_{1=a}$ sol(Q)

?-Q(a,b). <=> $\Pi_\phi \sigma_{1=a \wedge 2=b}$ sol(Q)

3.4. Reduction by substitutions

We have seen that by suitable substitutions it is sometimes possible to reduce a system of mutually interdependent equations to a system of equations which are independently resolvable, when a certain order of evaluation is observed. We call these systems of equations reducible by substitutions. Let us first define the problem more formally. We start with a given set S of equations of the following form:

S: R1 = E1[R1,...Rn]
R2 = E2[R1,...Rn]
.....
Rn = En[R1,...Rn]

where each Ri is a distinct relational variable and each Ei is an expression of RA^+, which involves some (not necessarily all) variables R1,...Rn; we call Ei the defining part of Ri.

A substitution consists in the replacement of some variable Ri by its defining part Ei in the RHSs of all equations of S. A resolution of Rj consists of a series of substitutions which generate an equation Rj=E' that does not contain any variable different from Rj. Clearly, if Rj has a resolution, then its value can be computed either by evaluation of an expression of constant relations (if Rj does not appear in E') or by the application of the closure operator to E', yielding 0^{Rj} E'(Rj).

After successful resolution, Rj can be marked as a known constant relation and we can eliminate the defining equation for Rj from the original system of equations S. This step is called the elimination of Rj from S. A set of equations is reducible by substitutions iff it can be transformed to the empty set by successive resolutions and eliminations. Note that it is useless to substitute a variable which occurs in its own defining part; we shall forbid such substitutions.

It is clear that the particular form of each expression Ei in S as well as the constant relations that appear in Ei are not relevant for determining whether S is reducible by substitutions; the only relevant information is the mutual interdependency of relational variables in S. The information on variable interdependency is most appropriately represented in the dependency graph G=<N,E> defined as follows:

N(G)= {R1,...,Rn}
E(G)= {<Ri,Rj>|Rj occurs in Ei}.

Note that loops of the form <Ri,Ri> may occur in E(G); in this case we call Ri a looping node.

It is easy to see that substituting a variable Ri in S exactly corresponds to dropping the node Ri from G and linking all the predecessors of Ri to all successors of Ri by new edges. By analogy, we call such a process the substitution of node Ri in G. Since we have forbidden to substitute variables Ri that occur in their own defining part, we forbid to substitute looping nodes in a dependency graph.

If, after a series of node substitutions, the node Rj has no outgoing edges to any nodes different from Rj, we say that we have resolved Rj. After a successful resolution of Rj we can eliminate node Rj and all its incident and outgoing edges from the original graph G, yielding a new (and simpler) graph G'=G-{Rj}. If, by successive resolution and elimination of nodes, it is possible to get an empty graph, we say that G is reducible by substitutions.

Lemma 3.1. Any acyclic connection graph G is reducible by substitutions.

Theorem 3.1. A strongly connected graph G is reducible by substitution iff it contains a node K such that G-{K} is acyclic.

Theorem 3.2. A graph G is reducible by substitution iff its strongly connected components are each reducible by substitutions.

Proofs of Lemma 3.1 and of Theorems 3.1 and 3.2 are presented in [Camerini86]. Based on Theorems 3.1 and 3.2, we build algorithm REDUCE which determines whether a graph G is reducible by substitutions; in the positive case, the algorithm outputs a series of substitutions and resolutions (in correct order) to reduce G.

The algorithm requires introducing a Reduction Graph $\hat{G}(N,E)$, built from the dependency graph G by identifying all strongly connected components of G.

$N(\hat{G})$ are the connected components of G;
$E(\hat{G})$ are the edges between connected components of G, defined in the obvious way.

Clearly $\hat{G}$ is acyclic; we call bottom nodes of $\hat{G}$ all nodes which have no outgoing edges.

ALGORITHM "REDUCE"
INPUT: Dependency Graph G
OUTPUT: Sequence of substitutions or resolutions

1. identify strong connection components G1...Gk of G.
2. for each component Gi find a node Ki in Gi such that Gi-{Ki} is acyclic; if Ki cannot be found then stop with output "irreducible".
3. build the reduction graph $\hat{G}$.
4. if $\hat{G}$ is empty then stop.
5. for any bottom node Gi of $\hat{G}$ do:
   for each node Rj in Gi-{Ki}
      output "substitute Rj";
   output "resolve Ki";
   apply REDUCE to Gi-{Ki};
   remove Gi from $\hat{G}$.
6. goto 4.

It is easy to see that the REDUCE algorithm is polynomial in the size of its input (i.e. the dependency graph G): the identification of the strong connection components (step 1) as well as the acyclicity test (step 2) are well known polynomial problems; furthermore it is not hard to see that steps 4-6 are repeated at most ||N(G)|| times.

Example. Consider the following system S of equations in RA+:

$R1 = C1 \cup \pi_{l1}(R1 \bowtie_{p1} R6) \cup R5$

$R2 = R5 \cup R4$

$R3 = \pi_{l2}(R4 \bowtie_{p2} R3) \cup C2$

$R4 = R3 \bowtie_{p3} C4$

$R5 = \pi_{l3}(R1 \bowtie_{p4} R6) \cup C3$

$R6 = (R1 \bowtie_{p5} C5) \cup C6$

Figure 1 shows the reduction graph for S.

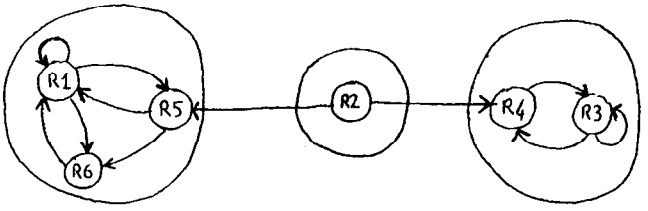

Fig. 1. Dependency graph and reduction graph for a system of equations

The output produced by the REDUCE algorithm is as follows:

SUBSTITUTE R4; RESOLVE R3; RESOLVE R4;
SUBSTITUTE R5; SUBSTITUTE R6; RESOLVE R1;
RESOLVE R6; RESOLVE R5; RESOLVE R2.

The sequence of solutions is produced as follows:

$R3 = \Pi_{l2}((R3 \bowtie_{p3} C4) \bowtie_{p2} R3) \cup C2$

$R4 = R3 \bowtie_{p3} C4$

$R1 = \Pi_{l1}(R1 \bowtie_{p1} ((R1 \bowtie_{p5} C5) \cup C6) \cup$
$\Pi_{l3}(R1 \bowtie_{p4} ((R1 \bowtie_{p5} C5) \cup C6)) \cup C1 \cup C3$

$R6 = (R1 \bowtie_{p5} C5) \cup C6$

$R5 = \Pi_{l3}((R1 \bowtie_{p4} R6) \cup C3)$

$R2 = R5 \cup R4$

3.5. Resolution of mutually dependent equations

Some systems of equations cannot be reduced by substitutions; in such cases we have to use other solution methods. There are two approaches for solving nonreducible systems. In the first approach, the single relation variables Ri are combined to one super-variable R, for instance by use of cartesian products. Then the original system of equations can be rewritten as R=E[R] and solved by applying the closure operator. Two different variants of this method are described in [Chandra82] and [Ceri86].

The second approach computes the solution directly from the original system of equations by initially setting $Ri=\phi$ for i=1...n and then sucessively computing Ri:=Ei[R1,..,Rn], until the value of each Ri remains unchanged:

ALGORITHM A

```
| FOR i:=1 TO n DO Ri:=ϕ;
| REPEAT
|   cond:=true;
|   FOR i:=1 TO n DO Si:=Ri;
|   FOR i:=1 TO n DO
|      BEGIN
|      Ri:= Ei[S1,...,Sn];
|      IF Ri <> Si THEN cond:=false;
|      END;
| UNTIL cond;
| FOR i:=1 TO n DO OUTPUT(Ri).
```

We can optimize Algorithm A by using at each step the recently computed values for R1,...,Ri for the computation of the new value of Ri+1, instead of using the old values, i.e., S1,...,Si:

ALGORITHM B

```
| FOR i:=1 TO n DO Ri:=ϕ;
| REPEAT
|   cond:=true;
|   FOR i:=1 TO n DO
|     BEGIN
|     S:= Ri;
|       Ri:= Ei[R1,...,Rn];
|      IF Ri <> S THEN cond:=false;
|      END;
| UNTIL cond;
| FOR i:=1 TO n DO OUTPUT(Ri).
```

Algorithms A and B have well-known correspondents in the field of numerical analysis: Algorithm A corresponds to the Jacobi algorithm for the iterative solution of systems of equations, while algorithm B corresponds to the Gauss-Seidel algorithm.

3.6. Conclusion of Section 3

In this section we have given rules for transforming FFLPs into systems of equations in RA+ and we have shown how these systems can be solved by use of the closure operator. By this process we have defined both the fixpoint semantics and the operational semantics for FFLPs and queries operating on an extensional database EDB.

Though outside the scope of this paper, it can be seen that our semantics correspond exactly to the semantics for logic programs defined by Van Emden and Kovalski [VanEmden76]; due to the limitedness of ranges for individual variables and to the absence of function symbols, the Herbrand universe of any FFLP is finite.

4. ALGEBRAIC APPROACH TO THE OPTIMIZATION OF LOGIC PROGRAMS

We turn now to the optimization of expressions in ERA+. Execution strategies presented in the previous section suffer from two major disadvantages:

a. The algorithm which computes the closure operation is not too efficient.

b. In the computation: $\sigma_F 0^X E(X)$, conditions of the logical query are not used, because the selection condition is not pushed inside $0^X E(X)$.

Proofs of theorems in this section are quite simple and can be found in [Ceri86].

4.1. Efficient computation of the closure operator

Algorithm A_0 presented in section 2.2 is not very efficient because several partial unions are needed and because E(S) has to be evaluated several times for the same tuples; consider now the more efficient program A_1:

```
| A1(E(X)):
```

```
|   S <- φ
|   REPEAT
|     Y <- S
|     S <- E(S)
|   UNTIL Y=S
| RETURN S
```

Theorem 4.1. If E(X) is an expression in RA+, then $A_0(E(X))$ and $A_1(E(X))$ are equivalent programs.

Note that Theorem 4.1 does not hold if E(X) is not in RA+; for instance, if E(X)=K-X, $K \neq \phi$, then A_1 never halts while A_0 terminates returning K as result.

Program A_1 eliminates the first source of inefficiency, i.e. the computation of unions within the program A_0, but it does not eliminate the second source, i.e. the computation of E(S) several times for the same tuple. We now develop two algorithms which do not have this burdensome property.

Definition 4.1. An expression E(X) over RA+ is linear if it holds:

$\forall X,Y \subseteq Rg(E) : E(X \cup Y) = E(X) \cup E(Y)$

We can now present algorithm A_2:

```
| A2(E(X)):
|   S <- φ; D <- φ
|   REPEAT
|     D <- E(D) - S
|     S <- S U D
|   UNTIL D = φ
|   RETURN S
```

Theorem 4.2. If E(X) is linear, then $A_2(E(X))$ is equivalent to $A_0(E(X))$.

The advantage of algorithm A_2 compared to algorithm A_1 is that the size of D_i (the "difference term" produced at each iteration) is smaller with respect to the size of S_i (the "accumulation" term).

Theorem 4.3. Each expression E(X) can be rewritten in a canonical form:

$$\cup_{i=1..k} \pi_{\alpha_i} \sigma_{\beta_i} (X^{\gamma i} \times C_i) \cup C_0$$

where $X^{\gamma i}$ represents the cartesian product of γi terms X and C_i are constant (database) relations.

Let can denote the transformation in RA+ from an expression E(X) to its canonical form. Let degree(E(x)) denote the maximal γi in can(E(X)) Efficient algorithms can be developed depending on the degree of an expression; we show, in particular, algorithms developed for degree(E(X))=2. Note that if degree(E(X))=1 then the expression is linear, hence A_2 can be applied.

Let E'(X,Y) be the expression obtained from E(X) by replacing each cartesian product X x X with X x Y. Then, we can build algorithm A_3.

```
| A3(E(X)):
|   S <- φ; D <- φ
|   REPEAT
|     T1 <- E'(S,D)
|     T2 <- E'(D,S)
|     T3 <- E(D)
|     D  <- (T1 U T2 U T3) - S
|     S  <- S U D
|   UNTIL D = φ
|   RETURN S
```

Theorem 4.4. If the degree of E(X) is 2, then algorithm A_3 is equivalent to A_1.

The advantage of algorithm A_3 compared to algorithm A_1 is that we never compute the term E'(S,S), which might be large. We further notice that the program can be simplified by omitting the evaluation of T2 if the expression E'(X,Y) is commutative. Algorithm A_3 can be generalized for a generic i.

Example: Nonlinear ancestor program

FFLP: anc(x,y):- par(x,y).
anc(x,y):- anc(x,z), anc(z,y).

DISEQUATIONS: PAR $\subseteq$ ANC
$\pi_{1,4}$(ANC $\bowtie_{2=1}$ ANC) $\subseteq$ ANC

EQUATION: ANC = PAR $\cup \pi_{1,4}$ (ANC $\bowtie_{2=1}$ ANC)

SOLUTION:
sol(ANC) = 0^{ANC}(PAR $\cup \pi_{1,4}$(ANC $\bowtie_{2=1}$ ANC))

The degree of E(ANC) is 2. Assuming acyclicity of the PAR relation, algorithm A_3 produces at each iteration i the pair of ancestors corresponding to the 2i-th and 2i+1-th generations; term T2 should not be evaluated, as the expression is clearly commutative. Acyclicity of the PAR relation is not required by algorithm A_3.

4.2. Pushing selection conditions into linear expressions

Aho and Ullman [Aho79] indicate a method for optimizing expressions of the type: σ_F 0^X(E(X)). We briefly outline their method by one example. The linear expression for the set of all ancestors of an individual "a" is:

$$ANC = \sigma_{1=a}\ 0^X\ (\pi_{1,4}(X \bowtie_{2=1} PAR) \cup PAR)$$

It holds:

$$\sigma_{1=a} X = \sigma_{1=a}\ (\pi_{1,4}(X \bowtie_{2=1} PAR) \cup PAR)$$

By applying associativity and distributivity:

$$\sigma_{1=a} X = (\pi_{1,4}(\sigma_{1=a} X \bowtie_{2=1} PAR) \cup \sigma_{1=a} PAR)$$

By introducing the variable Y for $\sigma_{1=a} X$ we get:

$$Y = (\pi_{1,4}(Y \bowtie_{2=1} PAR) \cup \sigma_{1=a} PAR)$$

Thus we have:

$$ANC = 0^Y\ (\pi_{1,4}(Y \bowtie_{2=1} PAR) \cup \sigma_{1=a} PAR)$$

This formula can now be evaluated using algorithm A_2. Unfortunately, this method applies only when the selection can be pushed directly to the variable X in E(X). In the rest of this section we show, on the ground of examples, how other optimizations are possible. Consider the terms:

(4.1) $D_0 = E(\phi)$
$D_{n+1} = E(D_n) - E(\phi)$

Considering algorithm A_2, it is easy to verify that, for linear expressions E(X), the following equation holds:

$$0^X E(X) = \cup_{i=0,\ldots,\infty} D_i.$$

This formulation for E(X) is attractive because we can compute terms so that σ_F is pushed into each D_i, and never compute the full terms D_i:

(4.2) $\sigma_F\ 0^X E(X) = \cup_{i=0,\ldots,\infty} \sigma_F D_i.$

Starting from this general form, we can see how algebraic manipulations produce the same effect as techniques such as the "magic set" and "magic counting". We use a well-known example: the search of same generation cousins. Let us produce the transformation from FFLP to ERA+ for this example:

FFLP: sg(X,X).
sg(X,Y):- par(X,X1),sg(X1,Y1),par(Y,Y1).

DISEQUATIONS: EQ $\subseteq$ SG

$$\Pi_{1,5}((PAR \bowtie_{2=1} SG) \bowtie_{4=2} PAR) \subseteq SG$$

EQUATION: $SG = EQ \cup \Pi_{1,5}((PAR \bowtie_{2=1} SG) \bowtie_{4=2} PAR)$

SOLUTION:
$sol(SG) = 0^{SG}(EQ \cup \Pi_{1,5}((PAR \bowtie_{2=1} SG) \bowtie_{4=2} PAR))$

Notice that the expression computing SG is linear. We introduce the "composition" operation, as in [Aho 79], to denote the following expression (where R and S are binary relations):

$$R \circ S = \Pi_{1,4}\, R \bowtie_{2=1} S$$

Then, denoting as RAP the relation obtained by exchanging the order of attributes in PAR (RAP = $\Pi_{2,1}$ PAR), we have:

$$sol(SG) = 0^{SG} EQ \cup (PAR \circ (SG \circ RAP))$$

Note that the composition operation is associative, hence:

$$(X \circ Y) \circ Z = X \circ (Y \circ Z) = X \circ Y \circ Z$$

Further, we indicate as X^i a chain of i-1 applications of the composition to a binary relation X:

$$X^i = X_1 \circ X_2 \circ \ldots X_i$$

Terms D_i defined by the system (4.1) are:

$D_0 = E(\phi) = EQ$
$D_{i+1} = E(D_i) - E(\phi) = PAR^i \circ EQ \circ RAP^i$

Consider the query in FFLP:

?- sg(a,X).

corresponding to the expression in ERA+:

$$\sigma_{1=a}\, sol(SG)$$

By propagating selections to the terms D_i in the right side, we obtain:

$$D_i = \sigma_{1=a}\, PAR^i \circ EQ \circ RAP^i$$

Let us compare the terms D_i and D_{i+1}: we denote as reducing common subexpression(s) R_i the largest common subexpression(s) of D_i, D_{i+1} which includes selection condition(s); in this case,

$$R_i = \sigma_{1=a}\, PAR^i$$

It is possible to pre-determine a subset of the relation PAR which contains all relevant tuples for the computation of SG; this is done by evaluating the "magic" set M of all elements that can appear in the second column of terms R_i; this is the set of all ancestors of "a".

$PAR_0 = \phi$
$i=0$
while R_i is not empty do
begin
$PAR_M = PAR_M \cup R_i$
$i = i+1$
end
$M = \Pi_2\, PAR_M$

Consider now the semi-join reduction of PAR:

$$PAR' = PAR \ltimes_{2=1} M.$$

It is easy to see that only tuples of this relation give a contribution to terms D_i; we can then write:

$$D_i = \sigma_{1=a}\, PAR'^i \circ EQ \circ RAP^i$$

By using equation (4.2), we deduce that:

$\sigma_{1=a}\, 0^{SG}\, (EQ \cup (PAR \circ SG \circ RAP)) =$
$\cup_{i=0,\ldots,\infty}\, \sigma_{1=a}\, PAR^i \circ EQ \circ RAP^i =$
$\cup_{i=0,\ldots,\infty}\, \sigma_{1=a}\, PAR'^i \circ EQ \circ RAP^i =$
$\sigma_{1=a}\, 0^{SG}\, (EQ \cup (PAR' \circ SG \circ RAP))$.

We can then apply algorithm A_2 to solve this simplified problem. Note that M is itself obtained as the application of the closure to a simple expression, as follows:

$$M = \Pi_2\, 0^X\, ((X \circ PAR) \cup \sigma_{1=a}\, PAR)$$

Assuming acyclic data, we can easily show the algebraic equivalent of the "magic counting" method. Let us first simplify each term D_i by eliminating the EQ relation and propagating equality conditions:

$$D_{i+1} = PAR^i \circ RAP^i$$

With acyclic data, there cannot be replicated tuples in the union of terms R_i; hence, there is an i such that $R_i = \phi$ for $j>i$. But if $R_i = \phi$, then also $D_i = \phi$. Hence we can evaluate SG using the LHS of equation (4.2) by the following algorithm:

$SG = \phi$
$R = \sigma_{1=a}\, PAR$
$i=0$
while $R \neq \phi$ do
begin
$R = R \circ PAR$
$SG = SG \cup (R \circ RAP^i)$
$i=i+1$
end

Algebraic transformations of this section are easily generalized for a query with two bindings:

?-sg(a,b).

corresponding to the expression:

$$\Pi_\phi\, \sigma_{1=a \wedge 2=b}\, sol\,(SG).$$

We omit details of derivations, and show the final results:

(1) Using magic sets:

$$\Pi_\phi\, \sigma_{1=a \wedge 2=b}\, 0^{SG}\, (EQ \cup (PAR' \circ SG \circ RAP'))$$

with RAP' being the semi-join reduction of RAP by the magic set produced by the selection $\sigma_{1=b}$ RAP.

(2) Using magic counting, we produce the program:

(2) Using magic counting, we produce the program:

```
answer=no
R1 = σ_{1=a} PAR
R2 = σ_{2=b} PAR
if R1 o R2 ≠ φ then answer = yes
while ((R1 ≠ φ) and (R2 ≠ φ) and (answer = no))
    do
  begin
  R1 = R1 o PAR
  R2 = PAR o R2
  if R1 o R2 ≠ φ then answer = yes
  end
output answer
```

Notice that with this program we compute iteratively two terms, each obtained from one binding condition. The computation is halted as soon as either of the two terms is empty or D_i = R1 o R2 produces one tuple.

5. CONCLUSIONS

This paper has presented a systematic approach to the algebraic treatment of logic queries; we have shown a syntax directed translation from FFLP to algebraic equations and then shown how equations or systems of equations can be solved and how individual equations can be optimized.

Several problems considered in this paper need further improvements:

a. The proposed solution method for systems of equations could be improved by propagating bindings from one equation to another.

b. Efficient algorithms presented for expressions of degree 1 and 2 can be generalized to expressions of any degree.

c. Further investigation is needed to fully understand how the algebraic approach compares with the "magic set" and "magic counting" methods.

d. Another noticeable direction of research has as goal the treatment of Horn clauses including function symbols. The necessary counterpart on the database side is the extension of the relational model and languages to model complex objects (e.g., non-1NF relations).

Acknowledgement

This research was supported by the Esprit project n.432 Meteor (An integrated formal approach to industrial software developement). We like to thank Stefano Crespi-Reghizzi, who has stimulated our work and provided useful comments on a first draft of the paper, Letizia Tanca, who has suggested us an improvement for the solution of systems of equations, and Paolo Camerini, who has helped proving the theorems in Section 3.4.

REFERENCES

[Aho79] A.V. Aho, J.D. Ullman, "Universality of data retrieval languages", Sixth ACM Symp. on principles of programming languages, San Antonio Jan. 1979.

[Apt82] K.R. Apt, M.H. VanEmden, "Contributions to the theory of logic programming", ACM Journal, 29:3, p841-862, 1982.

[Bancilhon86] F. Bancilhon, D. Maier, Y. Sagiv, J.D. Ulmann, "Magic sets and other strange ways to implement logic programs", Proc. ACM-PODS, Cambridge (MA), March 24-26 1986.

[Camerini86] P. Camerini, S. Ceri, G. Gottlob, L. Lavazza: "A note on the solution of mutually dependent equations by variable substitution", Dipartimento di Elettronica, Internal Report, 1986.

[Ceri85] S. Ceri, S. Crespi-Reghizzi, L. Lavazza: "Extended Relational Algebra (ERA): Data structures and operations", Rep. n. Meteor/T2/TXT/1, June 1985.

[Ceri86] S. Ceri, G. Gottlob, L. Lavazza, "Translation and optimization of logic queries: The algebraic approach", Dipartimento di Elettronica, Politecnico di Milano, Rep. n.86-004.

[Chandra82] A.K. Chandra, D. Harel, "Horn clauses and the fixpoint hierarchy", in Proc. ACM-PODS, pp. 158-162, 1982.

[Henshen 84] L.J. Henshen and S.A. Navqi, "On compiling queries in recursive first-order databases", ACM Journal, 22:4, pp.47-85, 1984.

[Marque_Pucheu84] G. Marque-Pucheu, J. Martin-Gallausiaux, G. Jomier, "Interfacing Prolog and relationl DBMS" New applications of DBs, Academic Press 1984.

[Reiter78] R. Reiter, "On closed world databases", in Logic and databases, Plenum Press, New York, 1978.

[Sacca'86] D. Sacca', C. Zaniolo, "On the Implementation of a Simple Class of Logic Queries for Databases", Proc. ACM-PODS, Cambridge (MA), March 24-26 1986.

[Ullman82] J. Ullman, "Principles of database systems", Computer Science Press, Rockville, Md., 1982.

[Ullman85] J. Ullman, "Implementation of logical query languages for databases", ACM-TODS, 10:3, pp. 289-321, 1985.

[VanEmden76] M. H. Van Emden, R. Kovalski, "The semantics of predicate logic as a programming language", ACM Journal, 23:4, pp.733-742, 1976.