REDUCING LINEAR RECURSION TO TRANSITIVE CLOSURE

Linda Ness

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-25 November 1986

Reducing Linear Recursion to Transitive Closure*

Linda Ness
University of Texas and Carleton College

Abstract

When querying a database using a logic program, one sometimes wants to phrase the query recursively. The question of implementing recursive queries, effectively, then arises, since recursion is expensive to implement. Efficiency of implementation might be improved if the query is rephrased before execution. In this paper, we propose and prove a strategy for evaluating recursive queries defined by an exit rule and a linear recursive rule, expressed as a Horn clause without function symbols. In the strategy, the recursive relation sought is computed, by join and union, from a much simpler recursive relation. The simpler relation is proved to be, in general, the disjoint union of complete simple transitive closure relations. In the special cases that the graph is acylic, or consists of only of components containing cycles, the relation is a single simple transitive closure relation, or is not essentially recursive, since it can be computed by bottom-up evaluation in a bounded number of iterations. The defining rules for the simpler recursive relation, involve one new relation, which would have to be computed and cached. The advantage of this strategy is that it suffices to develop efficient algorithms for computing complete simple transitive closure relations, since the selections have been pushed through. Much of the work on strategies for evaluating linear recursive queries and performance of these strategies focuses on such complete simple transitive closure relations. In fact, for such relations, a simple variation of bottom-up evaluation, called semi-naive evaluation is optimal, in the sense that no inference is made twice.

*This research was partially supported by NSF Grants MCSS-8104017, MCS-8214613, and DCR-8507224 and ONR Contract N00014-86-K-0161.

1.Introduction

When querying a data base using a logic program, one sometimes wants to phrase the query recursively. For example, suppose one represents information about a directed graph in the following two database relations:

p(X,Z) represents the edge relation,

e(X) represents a subset of nodes of the graph.

Let r(X) denote the relation consisting of nodes X, reachable along some path emanating from a node in the subset e. In order to query for the relation r(X), one must phrase the query recursively. A set of rules defining r is:

(1a) r(X) :- e(X)

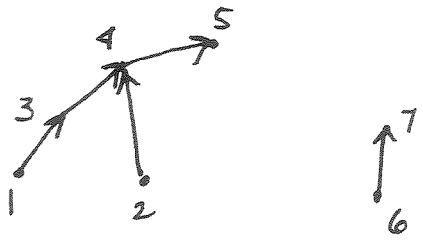
(1b) r(X) :- p(X,Z)r(Z)

The first rule is a base condition, or an exit rule. It means the relation r contains the relation e.

The second rule means a node X is in the relation r if there exists a node Z such that there is an edge from Z to X, and Z is in the relation r. The second rule is an example of a linear recursive rule since there is only one occurrence of the recursive predicate r on the right-hand side of the rule; in addition, it is regular since there is only one nonrecursive predicate.

The predicate on the left hand side of a rule is called the goal predicate.

Example 0: In the directed graph below, the nodes in the relation e are nodes 1 and 2. The relation r consists of nodes 1, 2, 3, 4, 5.



Another interpretation of the relation defined by the above pair of rules can be given if e is interpreted as a

group of people, and p(X,Z) is interpreted as "X is a parent of Z". Then r(X) is the set of ancestors of people in the group e.

Then relation defined by the pair of rules (1) is equivalent to the relation defined by an infinite sequence of rules $r_0, r_1, r_2, \ldots$, where the rule r_i is obtained by applying the recursive rule i times and then applying the exit rule. The first few of these rules are:

r_0 : r(X) :- e(X)

r_1 : r(X) :- p(X,Z1)e(Z1)

r_2 : r(X) :- p(X,Z1)p(Z1,Z2)e(Z2)

r_3 : r(X) :- p(X,Z1)p(Z1,Z2)p(Z2,Z3)e(Z)

.

.

.

The relation may be evaluated in a bottom-up fashion, by evaluating r_i and adding the results to the results of evaluating the earlier rules. We will call the subset of the relation that has been computed in this manner $R_i(X)$, so

$$R_0(X) \quad R_1(X) \quad R_2(X) \quad \cdots .$$

Since the relations e and p are finite, no new tuples will be found after some iteration, say the kth. Thus R_k (X) will be the entire relation. The recursion in the in the first pair of rules (1) is not bounded since one cannot predict a constant k that will work for all relations e and p.

One simple class of recursive relations is that where the relations defined by replacing X and Z in the above rules by ordered sets $X_1, X_2, \ldots X_n$ and $Z_1, Z_2, \ldots Z_n$, where the X_i's and Z_j's are all distinct variables. We call such relations complete simple transitive closure relations. Their defining rules have the form:

$$r(X_1, \ldots, X_n) :- b(X_1, \ldots, X_n)$$

$$r(X_1, \ldots, X_n) :- p(X_1, \ldots, X_n, Z_1, \ldots, Z_n) r(Z_1, \ldots, Z_n)$$

These defining rules are especially simple for two reasons:

a) None of the X_i's in the head of the recursive rule appear in the recursive predicate.

b) None of the X_i's or Z_i's are constants, and there are no repetitions among the X_i's or Z_i's. In other words, the full relation r is sought.

Fundamentally, our main result is: any non-bounded linear recursive relation, is either the disjoint union of complete simple transitive closure relations, or is computed by joining an intermediate relation with such a disjoint union, and then taking the union with a subrelation $R_{d-1}(X)$, for a constant d, that can be determined from the recursive rule.

An example of a complete simple transitive closure relation is provided by the "cousins-at-the-same-generation" query:

$$sg(X,X)$$

$$sg(X_1, X_2) :- p(X_1, Z_1)p(X_2, Z_2)sg(Z_1, Z_2)$$

The recursive rule here is linear, but not regular, since there is more than one non-recursive predicate. If one defines the relation s by

$$s(X_1, X_2, Z_1, Z_2) :- p(X_1, Z_1)p(X_2, Z_2)$$

the relation sg can be defined by the same base condition, and the regular linear recursive rule

$$sg(X_1, X_2) :- s(X_1, X_2, Z_1, Z_2)sg(Z_1, Z_2)$$

Hence, when the "cousins-at-the-same-generation" relation is defined in terms of the relation s, it is an example of a complete simple transitive closure relation.

In [BMSU], Bancilhon et al. discussed strategies for evaluating linear recursive queries. They asked how the counting method and the reverse counting method could be generalized from the "cousins at the same generation example" to arbitrary linear rules. These two strategies both apply well to complete simple transitive closure relations.

Linear recursive rules are, in general, much more complicated, than the linear recursive rules defining complete simple transitive closure relations. Consider the following regular rule:

2) $$r(X_1, X_2, X_3, X_4, X_5) :- p(X_1, X_2, X_3, X_4, X_5, Y_1, Y_2)r(X_2, X_1, Y_1, X_2, Y_2)$$

Here the rule for substituting variables into the recursive predicate is much more complicated. One can easily think of virtually an infinite number of examples of regular, linear recursive rules by just thinking of examples of substitution rules. If the X_i's are distinct variables, the full relation is still sought. One can further complicate the example above by seeking only the subrelation where X_2 has a particular value, say 'John', and where $X_1=X_2$. In other words, the X_i's could be constants or variables, and they need not all be distinct.

Rule 2 can be rewritten much more simply, if we adopt a few conventions. Namely let X denote $X_1, ..., X_5$; let Y denote $Y_1, ..., Y_5$ and let X,Y denote the concatenation of the two sequences above, so

$$X, Y = X_1, ..., X_5, Y_1, ..., Y_5$$

Finally,let S(X,Y) denote X_2, X_1, Y_1, X_2, Y_2, i.e. the argument of the recursive predicate. Then Rule 3 can be rewritten as:

$$r(X) :- p(X,Y) r(S(X,Y))$$

The essential information is the substitution rule S.

In this paper we will confine our attention to the class of queries consisting of an exit rule or base condition of the form

3a) $r(X) :- e(X)$

and a linear recursive rule, which we define to be a rule of the form

3b) $r(X) :- p_1(X,Y) p_2(X,Y) ... p_k(X,Y) r(S(X,Y))$

where X and Y and X,Y denote the sequences specified above, and where S(X,Y) denotes a sequence of length n whose entries are selected from the X_i's and Y_j 's. The X_i's and Y_i's may be variables or constants. No Y_i , that is a variable, can equal an X_j that is a variable. We will say that the X_i variables are distinguished, and the Y_j variables are nondistinguished variables. There may be repetitions among the distinguished variables and among the nondistinguished variables. The rule 6 is regular if k=1, so there is only one non-recursive predicate.

The interpretation of the rules is exactly analagous to that described for 1). The second rule 3b) means that

a tuple X is in the relation r, if there exists a tuple Y such that the concatenation of the tuples X,Y is in relations $p_1, p_2, \cdots p_k$, and the tuple S(X,Y) is in the relation r. The tuples X and Y , of course have to satisfy the specified constant and repetition constraints.

A linear recursive relation defined by a pair of rules in the form of 3a) and 3b) can be evaluated bottom-up using an infinite sequence of rules r_i defined as in the initial example. In order to obtain the rule r_i, the substitution operation S will have to be iterated i times. The essence of the recursion is captured in these iterations. More notation will have to be established to express this sequence of rules. As before we will let $R_i(X)$ denote the subrelation obtained by evaluating the rules $r_0, r_1, ..., r_i$.

Our main results give a new pair of defining rules for the relation. If one decides to do bottom-up evaluation for j-1 steps, followed by a cache of the relation ,$q_j(X,Z)$, defined by the non-recursive predicates in the rule r_j, one can then redefine the linear recursive relation using the subrelation R_{j-1} as the new base relation and the cached relation as the non-recursive predicate in a regular linear recursive rule. The main observation is that a certain choice of j makes the new recursive rule a simple transitive closure rule, if the original argument tuple X in the goal predicate was "standard", in a sense that we will define precisely later. Even if the original argument tuple X is not "standard", the argument tuple of the goal predicate of r_d, which we will denote $S^d(X,Y)$, will be "standard", so the relation $r(S^d(X,Y))$ can be computed using the method above, and then the desired relation r(X) can be computed from $r(S^d(X,Y))$, nonrecursively.

The proper choice of j can be determined from a directed graph determined by the substitution operation S in the linear recursive rule, and this j will be called the diameter of the graph.

It is interesting to note that Naughton [Nau2] found that the same choice of j was useful in specifying a new set of rules defining the same relation, which had fewer, if any, redundant predicates.

The new pair of defining rules we found for a linear recursive relation allowed us to present a strategy for evaluating such relations in which the only recursive relations that need to be computed are complete simple transitive closure relations. As Bancilhon showed [B1], Semi-Naive evaluation, which is an intelligent variant of bottom-up evaluation of these relations results in an optimal strategy for evaluating them, in the sense that the same inference is never made twice. The nonrecursive computations in the strategy, can be

done as efficiently as possible, using sideways information passing, for example. Hence we expect our strategy to be efficient, and feel that we have made significant progress on the problem Bancilhon et al. posed in [BMSU]. Ioannides'performance results in [I2] suggest that our strategy combined with his divide and conquer algoritms for computing the complete simple transitive closures, might also perform efficiently.

A quick perusal of the paper shows that much of it is directed to studying the substitution of variables into the recursive predicate.We did this with great care for several reasons. The graphs describing the substitution of distinguished variables are interesting in their own right and describe all the patterns of recursion. Hence one can easily make up interesting examples of recursion. Our use of a graph to deduce information about the iteration of the substitution operation is certainly not an original idea. The graph we use is simpler than Naughton's argument/variable graph. Ioannides also used a graph, [I1]. Using the adjacency matrix of our substitution graph, we can give a precise algebraic formula for the full substitution operation on distinguished and non-distinguished variables, that Naughton described graphically using weighted edges.

The algebraic formulation immediately leads to a formula for the jth iterate of the full substitution operation which makes its periocidity properties transparent. It allows easy handling of repeated variables and constants, in both the tuple X of arguments of the goal predicate and the tuple Y of arguments in the nonrecursive predicate. Many people have shied away from these more complicated and perhaps, seemingly unnecessary cases. However, we found that the notion of a standard tuple X was essential to our results, and such a tuple in general must contain repeated distinguished variables. Also, since we could handle constants in the argument tuple X as well, we were able to "push selections through".

Once we obtained a concise formula for the substitution operation and understood its properties, the main results were very easy to deduce.

2.The Substitution Graph and Distinguished Substitution Map for a Linear Recursive Rule

The most important information in a linear recursive rule is the formula for substitution of arguments in the goal predicate into the recursive predicate and introduction of new arguments into the remaining positions of the recursive predicate. In the rule

$$r(X) :- p(X,Y) r(S(X,Y))$$

X denotes the argument tuple for the goal predicate; Y represents the tuple of new arguments that are introduced. Then the argument tuple for the non-recursive predicate is the concatenation X,Y and the argument tuple for the recursive predicate S(X,Y) is obtained by applying the formula S for substitution to the argument tuple X,Y . The notation was chosen to suggest viewing S as a (linear) map from tuples to tuples.

We will call the arguments X of the goal predicate distinguished. The arguments Y will be called non-distinguished. We will say that a rule is generic in its distinguished (nondistinguished) arguments, if the distinguished (nondistinguished) arguments are distinct variables.

The substitution map S can be viewed as a combination of a substitution map for the distinguished argument tuple, and a substitution map for the non-distinguished argument tuple. We will denote the substitution map for the distinguished argument tuple by $s^{\sim}$.

More precisely, suppose r is a linear recursive rule, generic in its distinguished arguments, with n distinguished arguments. Define its **distinguished substitution map** by:

$s^{\sim}(X_1, ..., X_n)=(W_1, ..., W_n)$ where

1) $W_i=0$ if no distinguished variable is substituted into the ith position.

2) $W_i=X_j$ if the variable n position i in the goal predicate appears in poston j in the recursive predicate.

Thus $s^{\sim}$ is a (very simple) linear transformation from n-tuples to n-tuples. The matrix for $s^{\sim}$ is an nxn matrix, which has a 1 in the ith row and jth column if $W_i=X_j$, and has zeros elsewhere. The matrix for $s^{\sim}$ is the adjacency matrix for a directed graph, which we will call the substitution graph. In other words we define the **substitution graph** for the rule r to be the directed graph with

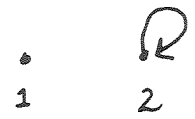
1) a node for each distinguished argument position, labeled by that positon number, and

2) an edge from node i to node j if the variable in position i in the goal predicate is substitued into position j in i the recursive predicate.

Below we give some examples. The graph is the substitution graph for the linear recursive rule; the distinguished substitution map is also given.

Example 1: the recursive rule for transitive closure

$$r(X_1, X_2) :- e(X_1, Y_1) r(Y_1, X_2)$$



$$\begin{bmatrix} 0 \\ X_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

Example 2: the recursive rule for the "cousins at the same generation relation"

$$sg(X_1, X_2) :- p(X_1, Z_1) p(X_2, Z_2) sg(Z_1, Z_2)$$

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

In general, the substitution graph for a complete simple transitive closure relation of arity n consists or n independent nodes and the adjacency matrix is, therefore, the zero matrix.

Recall that evaluating a relation defined by a linear recursive rule and an exit rule, is equivalent to evaluating the rules obtained by iterating the recursive rule i times and then applying the exit rule, for i = 0, 1, 2, ... We denoted these rules r_i. We want to predict the formula for substituting the distinguished arguments into exit predicate positions in these rules. This is equivalent to predicting the substitution map for the recursive rule obtained by iterating the original recursive rule i times. We will call this rule r'_i, and its distinguished substitution map $s^{\sim i}$, so $s^{\sim 1}$ equals the original substitution map $s^{\sim}$. When the recursive rule is iterated, the distinguished argument in position i moves to all positions corresponding to nodes which are successors of node i, or "disappears" if there are no such positions.

The preceding comment implies that the adjacency matrix for $s^{\sim j}$ is the jth power of the adjacency matrix for the original substitution map $s^{\sim}=s^{\sim 1}$. If the graph is acylic, after some number of iterations, say d, the distinguished arguments will all have disappeared. In other words the dth power of the adjacency matrix will be the zero matrix, so in linear algebra terms, $s^{\sim}$ is nilpotent. If, however, the graph consists of several directed cycles of length d_1 and d_2, after any multiple of d_i, i=1 or 2, iterations, the distinguished arguments in positions corresponding to nodes on cycle i, will have returned to their original positons. Thus if d is the least common multiple of d_1 and d_2, after any multiple of d iterations the distinguished arguments will be in their original positions. In this case $s^{\sim}$ is a permutation composed of two disjoint cycles.

Example 3:

$$r(X_1, X_2, X_3, X_4, X_5, X_6) :- p(X_1, X_2, X_3, X_4, X_5, X_6, Y_1) r(X_2, X_1, X_1, X_1, Y_1, X_5)$$

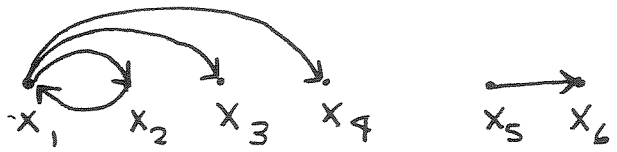
The substitution graph:

The rule obtained by iterating the recursive rule twice is:

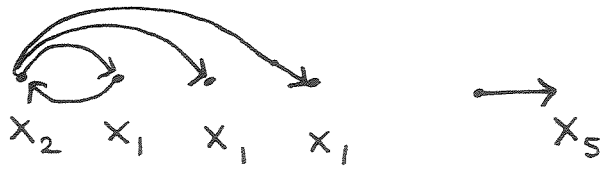
$r(X_{1,\ldots,}X_6)$:–$p(X_1, \ldots, X_6, Y_1)p(X_2, X_1, X_1, X_1, Y_1, X_5, Y_2)r(X_1, X_2, X_2, X_2, Y_2, Y_1)$

The propagation of the variables is illustrated below. The nodes of the substitution graph are labeled by the variables that occupy the corresponding positions:

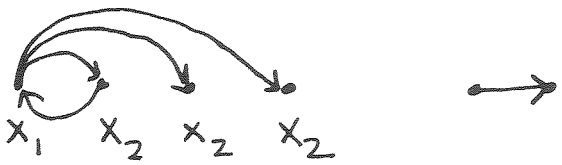
in the goal predicate:



in the recursive predicate of the original recursive rule:



in the recursive predicate of r'_2:



The definitions of the substitution map and distinguished substitution map can easily be extended to linear recursive rules, which are not generic in their distinguished arguments. Let X denote the distinguished argument tuple, in the goal predicate, and let S(X) denote the argument tuple of the recursive predicate.

Definition: Suppose $U=(U_1, ..., U_n)$ and $V=(V_1, ..., V_n)$ are tuples whose arguments are variables or constants. The tuples U and V are **isomorphic** if all of the tuples Y_j have the same pattern of repetition of variabls, and the same constants in the same positions. U is a **specialization** of V, if U is obtained from V, by

making selections, i.e. if U is obtained from V by setting some variable to be a constant, or by setting pair(s) of variables in V equal.

Specialization determines a partial order among argument tuples.

Now, if the argument tuple S(X) is isomorphic to a specialization of X, there is at least one distinguished substitution map s^{-}, which satisfies the definition. If S(X) is not isomorphic to a specialization of X, there are two cases to consider. If X and S(X) have different constants in the same positions, then the recursive relation defined by the rules only consists of the base relation. Otherwise, there is a unique tuple S'(X), which is maximal with respect to the following two properties:

a) S'(X) is a specialization of S(X), and

b) S'(X) is isomorphic to a specialization of X.

Since the relation r(X) is also defined by the original rules where S'(X) replaces S(X), it suffices to consider rules where S(X) is isomorphic to a specialization of X.

Example 4: The pair of recursive rules is:

$$r(X,X,Y,4) :- e(X,X,Y,4)$$
$$r(X,X,Z,4) :- p(X,X,Z,W)r(Z,W,4,X)$$

The same relation is defined by the exit rule and the new recursive rule

$$r(X,X,Z,4) :- p(X,X,Z,W)r(Z,Z,4,4)$$

3.Substitution Graphs and Their Diameter fP

Since the substitution graphs determine all possible patterns for linear recursion, it is useful to clearly understand them. It also provides a simple graphical way to generate "all examples".

Lemma 1: A directed graph with nodes labeled 1,...,n, for some integer n, is a substitution graph for some linear recursive query if and only if every node has at most one immediate predecessor.

Proof: In a linear recursive rule which contains distinct distinguished variables in distinct positions, each distinguished variable which appears in the recursive predicate comes from exactly one position in the goal

predicate. Thus each node in the substitution graph determined by a linear recursive relation has at most one predecessor. Conversely, given a directed graph, with nodes labeled by 1, ..., n, in which each node has at most one predecessor, one can construct a linear recursive rule, with that graph as its substitution graph.

In fact, every node in a substitution graph has at most 1 kth predecessor, for every positive integer k. For by a kth predecessor we mean a node reached by going backwards along k edges.

This characterization determines a convenient restriction on the structure of the connected components of the underlying undirected graph. Since we are considering connectedness properties of the underlying underected graph, one might initially think that such a graph might contain a (undirected) cycle, which, in fact, was

Lemma 2: A connected component of a substitution graph contains at most one (undirected) cycle. Any undirected cycle is, in fact, a directed cycle.

Proof: Suppose the component contains an undirected cycle; if the cycle were not a directed cycle, some node on the cycle would have 2 edges entering it, which is impossible by the characterization of substitution graphs. Now suppose a component contained 2 directed simple cycles. If both cycles contain node j, they must both contain a predecessor of node j. Since there is only one such predecessor, both cycles contain the unique predecessor of node j. Continuing this argument, we see both cycles contain each other. If the 2 cycles do not intersect, there is an undirected path from one to the other, since they are contained in the same component. We may assume the path contains exactly one node n1 and n2 from each cycle. The edges on the path must be directed away from the cycles so that neither n1, nor n2 has more than one entering edge. Hence the path contains some node with 2 entering edges, which again contradicts the fact that each node has at most one predecessor.

Naughton, in [Nau2], stated a result, equivalent to Lemma 2, for his argument/variable graphs.

Lemma 3: A component of a substitution graph is either a rooted tree or the union of a simple directed cycle with 0 or more trees rooted at nodes of the cycle, but otherwise disjoint from the cycle.

Proof: If a component is acyclic in the undirected sense, we must show there is a unique node with no predecessors. If there were 2 such nodes, there would be an undirected path from one node to the other node. The end edges would be directed away from the end nodes. Hence some node on the path would have 2 immediate predecessors, a contradiction. If the component contains a cycle, then deleting the edges in the cycle must leave an acyclic graph by Lemma 2, which as we have just shown is a union of rooted trees. The roots must be on the cycle since the original component was connected.

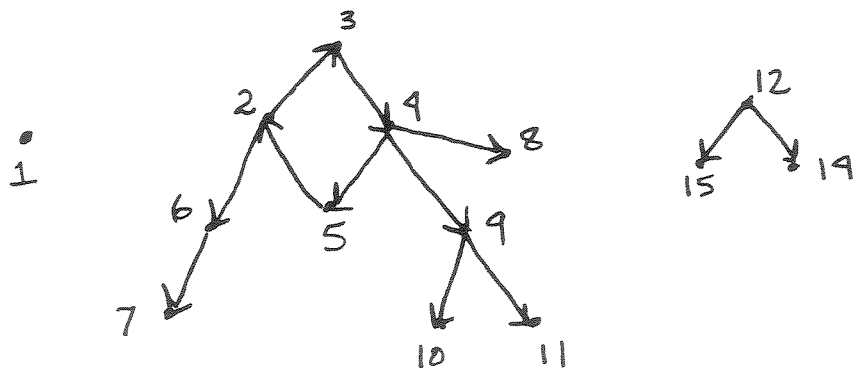
Lemma 3 implies that there are 2 natural subclasses of substitution graphs to consider: namely those with only acyclic components and those consisting of components which contain a cycle. We shall define the latter to be **cyclic substitution graphs**.

Remark: If no distinguished variable is repeated in the recursive predicate, then no node has more than one successor, so the components are either a single node, a nontrivial path, or a simple cycle.

Example 5: The substitution graph for the rule

$$r(X_1, ..., X_{14}) :- p(X, Y) r(Y_1, X_5, X_2, X_3, X_4, X_2, X_6, X_4, X_4, X_9, X_9, Y_2, X_{12}, X_{12})$$

where $X = (X_1, ..., X_{14})$ and $Y = (Y_1, Y_2)$ is:

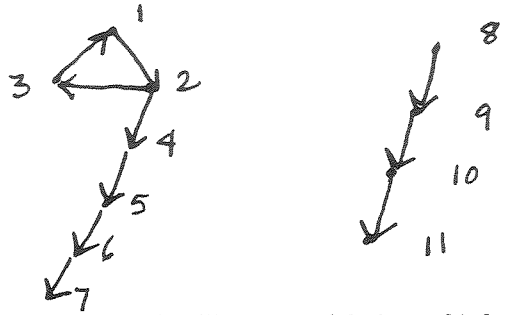


Finally we define the **diameter** of a substitution graph to be the smallest integer d such that

1) d is a multiple of the length of every cycle, and

2) the dth predecessor of every node either is on a cycle or does not exist.

Example 6: The diameter of the substitution graph below is 6.



The diameter of a substitution graph will be a crucial piece of information, If the graph is acyclic, d <= n, otherwise a crude bound on d is: where the product is over primes p less than n, and where the exponent of p gives the largest power of p which is smaller than n.

We are now in a position to formally summarize the discussion of iterations of the substitution mapping at the end of the previous section.

Lemma 4: Assume we are given a linear recursive rule. Let r'_j denote the rule obtained by iterating it j times. Let s~j denote the distinguished substitution map for r_j', j = 1, 2, ... Set $s = s^{1}$. Suppose that the substitution graph is of diameter d and has n nodes.

a) The matrix of $s^{\sim j}$ is the jth power of the matrix for s~.

b) The map $s^{\sim k}$ is defined by $s^{\sim k}(X_1, \ldots, X_n)=(W_1, \ldots, W_n)$ where $W_i=X_j$ if the jth node of the substitution graph is the (unique) kth predecessor of the ith node of the substitution graph.

c) If the substitution graph is acyclic, then $s^{\sim d}$ is the identically zero map.

d) If the substitution graph is cyclic, then

   i) $s^{d+j}=s^{d+j \bmod d}$

   ii) if every component is a cycle $s^{\sim d}=s^{\sim}$.

Proof: Only part i of d needs a proof. The diameter d is defined so that the dth predecessor of a node on a cyclic substitution graph is on a cycle. Since d is a multiple of the length of the cycle, and since the d+jth predecessor is on the same cycle for all j, the d+jth predecessor is the (d+jmod d)th predecessor.

4.The Substitution Map and Standard Argument Tuples

We must now extend the definition of the distinguished substitution map s~ to nondistinguished arguments as well, so that every argument positon in the recursive predicate of the rule r_j', j = 0,1,2,..., obtained by iterating the original recursive rule j times, is filled in a prescribed way.

The argument tuple of the recursive predicate or r_j' is obtained from the original distinguished argument tuple X By iterating the substituton map S j times. The formula for the map S itself is given in the original recursive rule. We will now precisely define the jth iteration, which we will denote S^j .

Each iteration of the substitution map S requires one new argument for each acyclic component of the substituton graph. For by definition of the graph, no distinguished argument is substituted into the argument positions corresponding to nodes which have no predecessors; these nodes are precisely the root nodes of the acyclic components. Order the acyclic components 1, ..., a and let a_i denote the argument position corresponding to the root node of the ith acyclic component. We will denote the tuple of (non-distinguished) arguments introduced in the jth iteraton by $Yj=(Yj_{a_1}, \ldots, Yj_{a_a})$. The ith argument of Yj will be assigned to the position r_i, the position corresponding to the root of the ith acyclic component. Thus it is convenient to define a map u from a-tuples to n-tuples, where n is the number of arguments in the recursive predicate, by

$u(Z_1, ..., Z_a)=(W_1, ..., W_a)$ where

$W_{a_i}=Z_i$ and

$W_j=0$ if some distinguished argument is substituted into the the jth position.

To make the notation cleaner, assume that the original non-distinguished argument tuple Y = Y1. Note that the original **substitution map** S, defined by the original recursive rule, is the vector sum

$$S(X,Y)=s(X)+u(Y).$$

The arguments, in the tuple Yj, introduced in the jth iteration, will, in the next iteration propagate to positions which are successors of the "root positions" into which they were introduced. The arguments in Yj, will in the j+kth iteration move into positions which are kth successor positions of the "root positions" into which they were introduced. After the arguments are introduced, they can be viewed as distinguished arguments, and the distinguished substitution map can be applied. Thus a precise formula for the jth iteration of

S, S^j, is:

$$S(X,Y)=s^j(X)+s^{j-1}(u(Y1))+\cdots+s(u(Y(j-1)))+u(Yj)$$

Here s^j is the jth iterate of the distinguished substitution map and addition is vector addition.

In general, the tuples of non-distinguished arguments Yj which are introduced, need not consist of distinct variables. The arguments can be constants, and variables can be repeated. We will assume that all of the tuples Yj are isomorphic.

Note that the isomorphism class of the tuples Yj of non-distinguished arguments is determined by the linear recursive rule. If the rule is not generic in its distinguished arguments X, there may be several substitution graphs which determine the same tuple S(X,Y). In this case, just choose one of the compatible substitution graphs. Let d denote its diameter. Let S denote the substitution map determined by such a substitution graph.

As the substitution operation S is iterated on a distinguished argument tuple X, the tuples $S^j(X,Y)$ fall into various isomorphism classes. A stable pattern emerges after a finite number of iterations.

Lemma 5: Suppose S is the substitution map determined by a linear recursive rule and a substitution graph of diameter d. Suppose $s^{\sim}$ is the distinguished substitution map determined by the substitution graph.

a) If the substitution graph contains only cyclic components, $S^{j+d}(X)=s^{d+jmodd}(X)$ for j = 0,1,... .

b) If the substitution graph is acyclic all of the argument tuples S^j(X,Y) are isomorphic for j >= d and the argument tuple S^{j+d}(X,Y) has no variables in common with the argument tuple S^j(X,Y) for j = 0,1,... .

c) The argument tuples S^{d+j}(X,Y) and $S^{d+jmodd}$(X,Y) j = 0, 1, ... are isomorphic.

Proof: For part a), note that S = $s^{\sim}$, since there are no non-distinguished variables, if the substitution graph is cyclic. Hence Part d) of Lemma 4 implies part a). If the substitution graph is acyclic, Lemma 4 implies that $s^{\sim}$ to any power, d or greater, is zero, so the definition of S^{k+d} , for k = 0,1,..., reduces to:

$$S^{k+d}(X,Y)=s^{d-1}(u(Y(k+1))+\cdots+u(Y(k+d)).$$

Since each of the tuples u(Y(k+i)), i = 1,...,d, are isomorphic, all of the tuples S^{k+d}, k= 0,1,..., are isomorphic, so part b) is proved. Part c) follows from parts a) and b), since the substitution operation is determined component by component.

Corollary 1: Equate argument positions with the nodes of the substitution graph to which they correspond. Suppose j >= d. After j iterations of the substitution operation S:

a) The positions in an acyclic component contain none of the original distinguished arguments in X. The argument positions in the ith acyclic component at distance k from the root, are all filled with the same argument $Y(j-k)_a$.

b) The positions in a cyclic component contain the distinguished argument which appears in the jth predecessor position, which is on the cycle in that component. If j is a multiple of d all of the positions on the cycle are filled with their original distinguished argument. If j is a multiple of d which is greater than d, all of the positions in the cyclic component contain the same argument that they contained after the dth iteration.

Proof: This follows directly from Lemma 5, the definition of the diameter, and the fact that arguments propagate to successor positions in each iteration of the substitution operation.

Recall that the argument tuple produced by applying the jth iteration of S to the distinguished tuple X and the non-distinguished tuple Y is the argument tuple of the recursive predicate r_j' obtained by iterating the recursive rule j times. Thus Lemma 5 can be viewed as a statement of the periodicity properties of these argument tuples, and as a statement about the standard pattern that emerges after d iterations.

We will need to refer often to this standard pattern of arguments which emerges after no more than d iterations.

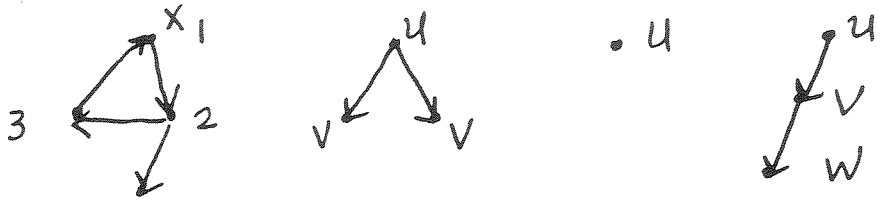
Let $a_1,...,a_a$ denote the argument positions corresponding to nodes of the substitution graph, which are roots of acyclic components. Let $Xa=(Xa_1,...,Xa_a)$. Thus Xa contains the arguments which are replaced by non-distinguished arguments in the substitution operation S.

Definition: An argument tuple X is **standard** for a linear recursive rule if

a) all argument positions, which are at the same distance from a node, contain the same argument, and

b) Xa is isomorphic to Y.

Example 7: If a recursive rule has the following substitution graph, a standard tuple is obtained by filling the argument positions corresponding to the nodes, with the arguments labeling the nodes, as shown.



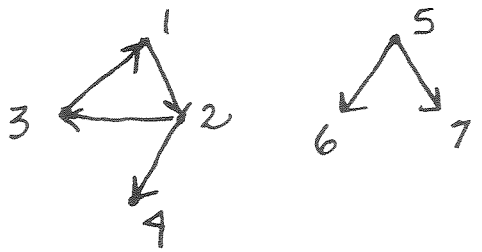
Lemma 6: For any argument tuple X, the argument tuple $S^d(X,Y)$ is standard.

Proof: The lemma follows immediately from the remark above.

The property of standardness is preserved under iterations of the substitution operation; that is, if X is standard for a substitution map S(,Y) determined by a linear recursive rule, then S(X,Y) is also standard. Thus if X is standard, the original linear recursively defined relation can be redefined by a linear reursive rule whose substitution graph contains only paths and cycles. the relations appearing in the new definition are obtained by projections and selections from the original relations.

Example 8: An example of a redefinition which results in a substitution graph consisting of only cycles and paths: Consider the recursive rule

r(X):- p(X,Y)r(S(X)) where $X=(X_1, X_2, X_3, X_3, X_4, X_5, X_5)$ and where S is the substitution operation determined bythe graph below.



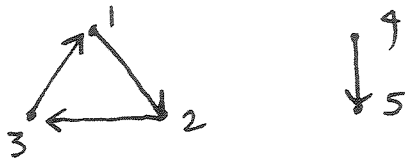
Thus $S(X)=(X_3, X_1, X_2, X_2, Y, X_4, X_4)$. Then X and S(X) are standard. Let $X'=(X_1,\ldots, X_5)$. Define

$$r'(X'):-r(X)$$
$$p'(X',Y):-p(X,Y)$$

Then r' is defined by

$$r'(X'):-p'(X',Y)r'(S'(X'))$$

where S' is the substitution operation determined by the graph:



so $S'(X')=(X_3, X_1, X_2, Y, X_4)$. Note that r' is a projection of r, and r is recoverable from r'.

5.A Strategy for Evaluating Linear Recursive Queries and its Optimality

Suppose the relation r(X) is defined by one exit rule and one linear recursive rule. Then the following steps give a strategy for evaluating the relation r(X). Initially set r(X) to be the empty relation.

Step 1:Construct the edge relation for a substitution graph,and compute its diameter d. Renumber the nodes, and the corresponding argument positions, so that the nodes on components containing a cycle precede nodes on acyclic components. Let $Xc=(X_1, ..., X_c)$ denote the arguments in positions corresponding to nodes on components containing a cycle.Then $Xa=(X_{(c+1)}, ..., X_n)$ denotes arguments in positions coresonding to nodes on acyclic components.

Step 2:If X is a standard argument tuple

Step 2a:

Compute the relation $R_{d-1}(X)$ and add it to the relation r(X), which is being computed. If the substitution graph is cyclic, that is, if every component of the graph contains a cycle, then stop, else continue

Step 2b:

Choose a subset of "acyclic" argument positions, $i_1, ..., i_m$, so that the tuple $X'=(X_{i1}, ..., X_{im})$ contains each variable in Xa exactly once.

Step 2c:

Let Z denote the argument tuple of the rule r'_d, obtained by iterating the recursive rule d times. Compute and cache the relation $q_d(X,Z)$ obtained by computing the join indicated by the nonrecursive predicates in r'_d, and then projecting onto the arguments X and Z.

Step 2d:

If the graph is acylic, so X = Xa, do the following steps. If the graph is not acylic, for each instantiation of the argument tuple Xc, or if the graph is acyclic so Xc is empty and X = Xa, do the following steps.

Step 2d1:

Compute the relation, $R_{d-1}(X')$, which is the projection of the relation $R_{d-1}(X)$.

Step 2d2:

Let $Z'=(Z_{i1}, ..., Z_{im})$. Compute the relation, $q_d(X',Z')$, which is the projection of $q_d(X,Z)$.

Step 2d3:

Compute the relation defined by the rules:

$r'(X'):-R_{d-1}(X')$

$r'(X'):-q_d(X',Z')r(X')$

Step 2d4:

Undo the projection mapping and add the resulting relation to r(X), the relation being computed.

Step 3:If S is not standard, then do

Step 3a:

Compute the relation $r(S^d(X))$, which is defined by the original pair of rules, with $S^d(X)$ substituted for X, using Step 2.

Step 3b:

Compute the relation r(X) from the relation $r(S^d(X))$ using the rules:

$r(X):-R_{d-1}(X)$

$r(X):-q_d(X,Z)r(S^d(X))$

where the tuple Z is empty, if the substitution graph is cyclic.

Thus the only recursive relations that need to be computed are complete simple transitive closure relations. These can be computed by the following **Semi-Naive Evaluation** algorithm, presented in [B1]. This algorithm uses two intermediate relations, New and dr, of the same arity as the relation r(X). Initially r(X) is the empty relation.

dr :- e

repeat

```
        add dr to r
        New(X) :- p(X,Z)dr(Z)
        dr = New - r(X)        /* set difference */
    until dr is empty
```

Bancilhon in [B1] observed the following:

Lemma 7: Semi-Naive Eavluation of complete simple transitive closure relations are optimal in the following sense:

No inference is made twice.

If a tuple is inferred that is already in the relation, it is not used in any more inferences.

As explained in the introduction, if we view the relation p(X,Z) as the edge relation in a directed graph, and if we view the base relation e(X), as a subset of marked nodes of the graph, the simple transitive closure relation computed by the Semi-Naive algorithm is the set of nodes, reachable by some path from a marked node. Since there may be more than one path from a marked node to another node, it is difficult not to infer some tuples twice, but at least they are not used to repeat previous inferences.

Another feature of Semi-Naive Evaluation is: the arity of the intermediate relations is ths same as the arity of the relation sought, and not any larger. The performance results of Bancilhon and Ramakrishnan, [BR], suggest that this significantly improves efficiency of evaluation.

Thus the only recursive linear relations that have to be computed recursively, are complete simple transitive closure relations.

6.Proof of the Strategy

We now prove that the strategy just given computes the relation defined by the original pair of rules.

It will be convenient to use the following notion.

Definition: A relation r(X) is a **simple transitive closure relation** if it can be defined by an exit rule and a regular recursive rule

r(X) :- e(X)

r(X) :- q(X,Z)r(Z)

where the argument tuples X and Z are isomorphic and have no variables in common.

In other words, selections are pushed through in a simple transitive closure relation.

Lemma 8: A simple transitive closure relation can be viewed as a complete simple transitive closure relation.

Proof: Suppose we have a simple transitive closure relation defined by the pair of rules above. Define relations r', e', and q', by projecting out the constant arguments, and enough other arguments to eliminate repetition of variables. Then r',e', and q', satisfy analagous rules, and define a complete simple transitive closure relation.

Theorem: The strategy is correct.

Proof: The main point is to show that the relation r(X) can also be defined using the rules in Step 2, if X is a standard argument tuple, and using the rules in Step 3, otherwise.

Let $q_j(X,Z)$ denote the relation obtained by first computing the join of the nonrecursive predicates in r_j, and then projecting out the non-distinguished arguments which do not appear in the argument tuple of the recursive predicate of r_j. Recall that the argument tuple of the recursive predicate of r_j is $S_j(X,Y)$, where Y is the non-distinguished argument tuple in the recursive predicate of r_1, the original recursive rule. Thus Z denotes the non-distinguished arguments of the tuple $S_j(X,Y)$.

For any j = 1,2, ... the relation r(X) can be defined by the following exit rule and regular linear recursive rule.

$r(X) :- R_{j-1}(X)$

$r(X) :- q_j(X,Z)r(S_j(X,Y))$

One can easily convince oneself, using bottom-up evaluation for both the original set of defining rules and the new pair of defining rules that the new pair of rules defines the same relation as the original set. Furthermore, if the relation $r(S_j(X,Y))$ has been computed, the rules above can be viewed as a nonrecursive way to define r(X) in terms of $r(S_j(X,Y))$.

Thus, the rules in Step 3 are correct, if the rules in Step 2 are correct, since, by Lemma 6, the argument tuple $S^d(X,Y)$ is standard.

Next, suppose that X is a standard tuple. Lemma 4, part d), and Lemma 5, imply that the relation is defined by the rules

$$r(X) :- R_{d-1}(X)$$

$$r(Xc,Xa) :- qd(Xc,Xa,Z)r(Xc,Z)$$

where Z is an argument tuple isomorphic to Xa, which has no variable arguments in common with Xa. Denote the relation determined by the instantiation Kc by r'(Xa), so r'(Xa) :- r(Kc,Xa).

Define $q'_d(Xa,Z) :- q_d(Kc,Xa,Z)$.Define $R'_{d-1}(Xa) :- R_{d-1}(Xc,Xa)$. Then r' is defined by

$$r'(Xa) :- R'_{d-1}(X)$$

$$r'(Xa) :- q'_d(Xa,Z)r'(Z).$$

Thus r' is a simple transitive closure relation. Finally, apply Lemma 8, to obtain the rules in Step 2. This completes the proof that the strategy computes the original relation.

Corollary 1: If the substitution graph is acylic, the relation r(X) is a simple transitive closure relation, and hence can be viewed as a complete simple transitive closure relation.

Proof:By hypothesis, the tuple Xc is empty, so X = Xa. Thus the loop in Step 2 is performed only once. Each time the loop is performed, a simple transitive closure relation is computed, by computing a complete simple transitive closure relation, obtained by projecting the original relation.

Recall that a recursive rule is strongly data independent, if for all possible exit rules, the recursive rule/exit rule pair can be replaced by a fixed, finite set of rules.

The correctness of the strategy implies the

Corollary 2: Assume the substitution graph contains only cyclic components. Then the relation r(X) has a non-recursive definition.

a) $r(X) :- R_{2d-1}(X)$

b) If X is a standard tuple, $r(X)$:–$R_{d-1}(X)$

Hence the relation is strongly data independent.

Proof: Since there are no acylic components, r(X) can be defined by

the rules:

$r(X)$:–$R_{d-1}(X)$

$r(X)$:–$q_d(X)r(X)$

if X is standard. But the second relation will generate no new tuples in a bottom-up evaluation, so the relation is completely defined by the exit rule. In general r(X) can be computed from $r(S^d(X))$ by the rules

$r(X)$:–$R_{d-1}(X)$

$r(X)$:–$q_d(X)r(S^d(X))$.

Since $S^d(X)$ is standard, $r(S^d(X))$=$R_{d-1}(S^d(X))$. Thus the second pair of rules defines $R_{2d-1}(X)$. Thus the relation has a non-recursive definition. Since the substitution graph is the same for all exit rules, the relaton has the same non-recursive definition for all exit rules, and hence is strongly data independent.

The corollary gives a sufficient condition on the substitution graph for strong data independence for linear recursive queries. Naughton in [Nau1] previously obtained a different sufficient condition on the argument/variable graph for strong data independence. His result is more intricate . Prior to Naughton, Sagiv [S], Minker and Nicolas [MN] and Ioannides [I1] obtained test for strong data independence whose hypotheses have more restrictions on the placement and repetition of the distinguished variables in the recursive predicate of the recursive rule body, i.e. restrictions on the substitution rule, and hence on the substitution graph. Our bound when the tuple is standard is new. Ioannides [I1] , previously obtained the other bound.The corollary, for us, was just a by- product of the main result, the correctness of the strategy.

7.The Strategy Applied to An Example

To illustrate the strategy, we apply the strategy to an example query, whose substitution graph contains both an acylic component, and a cyclic component.

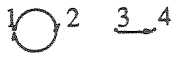
Suppose messages are communicated between transmitters. The transmitters can be in one of a finite number of states or modes. A transmitter in a particular state cannot necessarily send a message to another transmitter, when it is in some state. For example, no transmission is allowed between transmitters which are in the same states. The possible communication channels are stored in the relation p. A tuple in the relation $p(S_1,S_2,T_1,T_2)$ represents the fact that transmitter T_1, when it is in state S_1, can send a message to transmitter T_2, when it is in state S_2; each tuple in the relation p is called a channel. Now suppose that some of the channels are special; perhaps they are heavily used, or blocked, or connect transmitters in different countries. These special channels are stored in a relation e. We want to query for the channels which are connected to a special channel, via a succession of channels in alternating states. Messages transmitted over these channels might fail to be transmitted successfully to their destination, or might have a higher probability of being intercepted. The alternating state requirement is just another constraint on the message transmissions.

The query for the relation $r(S_1,S_2,T_1,T_2)$ of channels, which lead to special channels, through transmitters in alternating states is given below:

$$r(S_1,S_2,T_1,T_2):-e(S_1,S_2,T_1,T_2)$$

$$r(S_1,S_2,T_1,T_2):-p(S_1,S_2,T_1,T_2)p(S_2,S_1,T_2,T_3)r(S_2,S_1,T_2,T_3)$$

The substitution graph is shown below. Note that its diameter is 2, and the state variables correspond to nodes on cyclic components, while the transmitter variables correspond to nodes on acyclic components.



Since the diameter is 2, the relation to be cached, q_2, is just the join of the nonrecursive predicates in the second iteration of the recursive rule. Hence, it is defined by:

$$q_2(S_1,S_2,T_1,\ldots,T_4):-p(S_1,S_2,T_1,T_2)p(S_2,S_1,T_2,T_3)p(S_1,S_2,T_3,T_4)$$

A new set of defining rules for the relation r is:

$$r(S_1,S_2,T_1,T_2):-R_1(S_1,S_2,T_1,T_2)$$

$$r(S_1,S_2,T_1,T_2):-q_2(S_1,S_2,T_1,...,T_4)r(S_1,S_2,T_3,T_4)$$

The new recursive rule is simpler since at least some of the selections can be pushed through to the recursive rule, namely the selections made in the state variables. Selections made in the transmitter variables, can only be pushed through to the nonrecursive predicate.

The strategy for evaluating the full relation is: for each instantiation $S_1=s_1$, $S_2=s_2$, of the state variables, evaluate the resulting subrelation, using the new defining rules. The full relation is the disjoint union of the subrelations.

The same strategy can be used to evaluate subrelations of the full relation specified by setting certain variables to be constants, and possibly repeating other variables, since it is very clear, in the new rules, how far these selections can be pushed through.

7. Conclusion

In this paper, we have proposed a strategy for evaluating a linear recursive relation, defined by one exit rule, and one linear recursive rule. We proved that the strategy computes the correct relation. The main feature of our strategy is: the only recursive relations that need to be computed, are complete simple transitive closure relations. Thus we were able to avoid computing recursive relations, which involved selections, since in our strategy, the selections are "pushed through." Our strategy can handle repeated variables, and constants, in the goal predicate of the recursive rule. Hence, it is quite general.

We took advantage of an underlying periodicity, which appeared when the substitution operaton was iterated. It was helpful to view the substitution operation as a linear transformation, and to understand the relation between the iterates of the substitution operation and the graph, which was the adjacency matrix of the restriction of the substitution operation to distinguished variables. We focused our attention on the substitution operation, rather that on the non- recursive predicates.

We are currently trying to apply these ideas and techniques to develop strategies for relations defined by sets of mutually recursive linear rules, and relations defined by non-linear recursive rules.

8.References

[B1] Bancilhon,F., *On Knowledge Management Systems*, Brodie and Mylopoulos, Editors, Springer Verlag 1986,pp. 165-178.

[BMSU] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1986, pp.1-15.

[BR] Bancilhon, F., R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies, *Proceedings of ACM SIGMOD International Conference on Management of Data* 1986, pp. 16-53.

[I1] Ioannides, Y.E., "Bounded Recursion inDeductive Databases," Technical Report UCB/ERL M85/6, UC Berkeley, February 1985.

[I2] Ioannides,Y.E. "On the Computation of the Transitive Closure of Relational Operators,"Memorandum No. UCB/ERL M86/51, May 1986.

[MN] Minker, J., and J.M. Nicolas, "On Recursive Axioms in Relational Databases," *Information Systems*, 8(1):1-13, 1982.

[Nau1]Naughton, J.,"Data Indipendent Recursion in Deductive Databases," *Proceedigs of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems 1986, pp.267-279.*

[Nau2]Naughton, J.,"Optimizing Function-Free Recursive Inference Rules," preprint.

[S] Sagiv, Y., "On Computing Restricted Projections of Representative Instances," *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principls of Database Systems 1985, pp 171-180.*