Performing Inferences over Recursive Data Bases

Shamim A. Naqvi

and

Lawrence J. Henschen

Dept. of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois 60201

Abstract

The research reported in this paper presents a solution to an open problem which arises in systems that use recursive production rules to represent knowledge. The problem can be stated as follows: "Given a recursive definition, how can we derive an equivalent non-recursive program with well-defined termination conditions". Our solution uses connection graphs to first detect occurrences of recursive definitions and then synthesizes a non-recursive program from such a definition.

I. Introduction

In recent years, attention has focused on adding inferential capability to Codd's relational model of data (Codd 1970). This usually takes the form of defining new relations in terms of existing relations in the data base. The defined relations constituting the Intensional Data Base describe general rules about the data whereas explicit facts stored in the data base as base relations comprise the Extensional Data Base. This paper is concerned with the problem of finding a finite inference mechanism for a defined relation.

Reiter (1977) suggests that for non-recursive data bases the essentially logical operations involved in unifying and resolving intensional literals can be taken care of, i.e. "compiled", before receiving queries, leaving only those operations specifically related to information retrieval from the extensional data base.

We propose to extend this idea to the general case by analyzing what resolutions are possible that can lead to answers for a particular kind of query. In the case of recursive axioms this involves finding a pattern of data base retrievals instead of just a single data retrieval as in Reiter (1977).

II. Problem Representation

We shall view a data base as the ground unit clauses of a first order theory without function signs. The words literal and relation will be used interchangeably and all variables are assumed to be universally quantified.

We propose to solve the problem of recursive definitions by using connection graphs like those of Sickel (1976) in which nodes represent intensional axioms and edges connect unifiable literals of opposite signs. A loop is a Potential Recursive Loop (PRL) if the substitutions around the loop are such that the two literals at both ends of the loop are not the same literal and are not in the same instance of the clause partition. Figure 1 shows an example PRL in which E and F are base relations and letters at the end of the alphabet denote variables whereas letters at the start of the alphabet denote constants.

In this case, starting from A(a,x,z,p) and resolving around the loop (separating variables as we go) we eventually come back to clause 1 yielding an ultimate resolvent ¬E1(x,y) A(a,y,z,b) B(y,y') ¬E1(y,y') in which the literal at the end of the cycle, A(a,y,z,b), is a different literal than the one we started the loop with. Two features of this loop traversal are noteworthy. First, the literal E causes data base accesses which provide possibly new values for y. Second, these values of y instantiate x for the next traversal around the loop and also cause data base accesses for F which provides answers to the query.

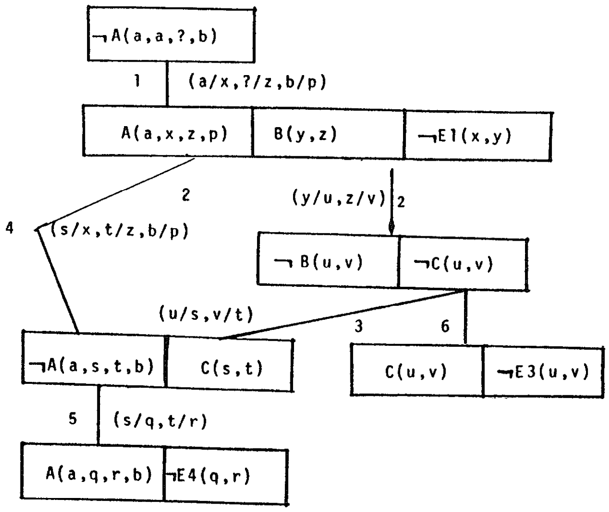

Figure 1. Example PRL

III. Derivation of the Iterative Program

Since non-atomic queries can be decomposed into equivalent atomic queries we shall only consider atomic queries in this paper. Before describing our method of deriving an iterative program for a recursive definition we notice that two kinds of edges exist in a PRL. A cycle edge contributes to the PRL whereas an exit edge provides an exit from the PRL. Extensional literals reached by traversing exit edges are called Exit Extensional Literals and those reached by traversing cycle edges are called Cycle Extensional Literals. For example in Figure 1 edges 2, 3 and 4 are cycle edges and edges 5 and 6 are exit edges; ¬E1(x,y) is a Cycle Extensional Literal whereas ¬E3(u,v) and ¬E4(q,r) are Exit Extensional Literals. We make the following observations about a PRL.

Observation 1 A PRL must have an exit edge, which corresponds to the presence of a basis case for a recursive definition, in order for its clauses to contribute an answer to a query. In figure 1 the basis case is A(a,q,r,b) ¬F(q,r). Notice that a literal having an exit edge has a non-exit edge which contributes to the cycle.

Observation 2 In Horn data bases, if a PRL exists for a literal Q, then a literal ¬Q must exist which provides the closing edge for the PRL.

We represent the defined relations as a connection graph and in a preprocessing step identify all PRLs. A set of answer expressions corresponding to a PRL is derived as follows: We note that the exit edges of Observation 1 above must be connected to cycle literals. Starting from the intensional axiom from which we expect to get a query, we first delete the literal which would resolve against the query. We then resolve around the cycle until we come to an exit edge. At this point the exit literal represents an expression which can be considered as a call to a procedure. This procedure provides one way of obtaining some of the answers. Having derived the expression for the first exit we proceed to successive exits in the same manner. These expressions are called answer expressions. In Figure 1 the answer expressions are ¬E1(x,y) OR ¬E3(y,z) and ¬E1(x,y) OR ¬E4(y,z).

A loop residue is obtained by resolving around the loop, starting from the intensional axiom from which we expect to get a query, and traversing only the cycle edges of the PRL. The ultimate resolvent is of the form

E:= ¬(E1(arg1,..) & ... & Ei(arg1,..))

where the Ei ($i>=0$) are base or defined relations. This expression is called the loop residue. In Figure 1 the loop residue is ¬E1(x,y).

In order to derive a program from a PRL we use an algorithm given in Naqvi (1980). In this section we shall illustrate the working of this algorithm by considering two similar definitions of the ancestor relation. Consider the first definition given below and the corresponding connection graph is shown in Figure 2.

7. ¬ANCESTOR(x,y) ¬FATHER(y,z) ANCESTOR(x,z)

8. ¬MOTHER(u,v) ANCESTOR(u,v)

It is straight-forward to show that with the query ¬ANCESTOR(w,a) we can generate the resolvent

9. ¬ANCESTOR(w,y') ¬FATHER(y',y'')....¬FATHER(y,a)

which corresponds to a left recursive definition of the ancestor relation. In this case the basis statement is used in the end to get the expression ¬MOTHER(w,y') ¬FATHER(y',y'')..¬FATHER(y,a). The data retrieval pattern is to find successive fathers of 'a' and then find a mother. In terms of the connection graph this corresponds to traversing the loop a certain number of times and then taking the exit edge. Examining the PRL we find that z, which is the variable that is expected to be the driver, is replaced in the loop by y. Moreover, z determines y through the extensional evaluation of Father(y,z). This determination occurs within the loop without recourse to the basis statement.

Our algorithm does the above kind of analysis and uses the answer expression derived from the loop and the substitutions from the closing edge of the loop to derive a program for the PRL. For this example it derives the following program fragment:

```
z:=a
ENQUE(q,z) /* q is a queue */
while (q ¬= empty) do
  z:= DEQUE(q)
  x:= ¬(MOTHER(x,y) & FATHER(y,z))
  ENQUE(q,y)
od
```

Now, consider the second definition of the ancestor relation given below and the corresponding connection graph shown in Figure 3.

11. ¬FATHER(x,y) ¬ANCESTOR(y,z) ANCESTOR(x,z)

12. ¬MOTHER(u,v) ANCESTOR(u,v)

Once again, the query ¬ANCESTOR(w,a) and 11 can be shown to generate the resolvent

13. ¬FATHER(x,y) ¬FATHER(y,y)...¬ANCESTOR(y'',a)

In this case, our first answers come from resolving (12) and (13) which corresponds in figure 3 to taking the exit edge to the basis case. Subsequent answers are derived by finding the successive fathers which corresponds to going around the loop a certain number of times. Examining the PRL we find that the values of y derive the next set of answers, x. The expected driver variable z does not participate in this process. Our algorithm uses the resolvent of the basis case and the query to start the loop. The substitutions at the closing edge of the PRL identify the correct variables which drive the loop and serve as place holders for answers. The loop residue derives all the subsequent answers. The program is as shown below.

```
x:=w
z:=a
¬ MOTHER(w,a)
ENQUE(q,w)
while (q ¬ = empty) do
```

```
  y:= DEQUE(q)
  ¬ FATHER(x,y)
  ENQUE(q,x)
od
```

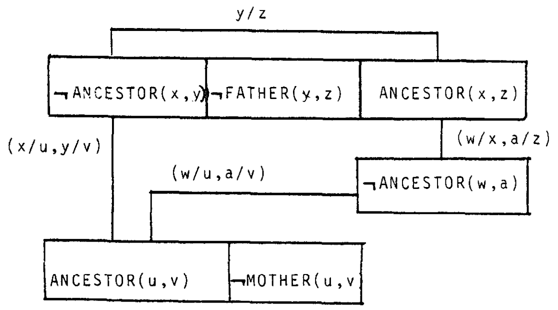

Figure 2. First Definition of Ancestor Relation

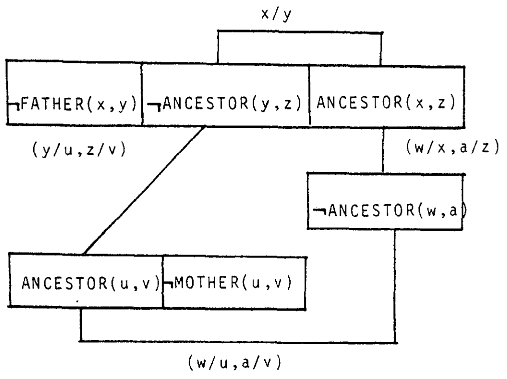

Figure 3. Second Definition of Ancestor Relation

To review then, our algorithm analyzes the PRL of a recursive definition to determine the loop residue, the answer expressions, the resolvent of the query and the basis case and whether the definition is left or right recursive. It then derives a program whose structure corresponds to one of the two program structures outlined above.

It now remains to discuss the termination conditions of the derived programs. Our termination conditions are designed for data bases without function signs. Briefly, we use a queue to store all unique values of the variables, indicated by the loop analysis, during each iteration. Each new iteration dequeues a previously enqueued value to generate some possibly new answers. Since the domain of discourse is finite in the absence of function signs the number of unique answers in the data base is finite. Thus the program queue will ultimately become empty. It should be noted that our technique for the detection of and generating programs for recursive definitions works in the presence of function signs. However, the termination condition does not guarantee finite computations in this case.

V. Summary and Conclusions

We have outlined an algorithm which derives iterative programs for recursively defined relations. The case where a defined relation is mutually recursive with some other definition (e.g. X & R -> R and R & Y -> X) leads to the derivation of mutually recursive programs. Transitive recursive axioms (e.g. ancestor of an ancestor is an ancestor) lead to the derivation of recursive programs. These situations require a fairly complicated control mechanism for execution time invocation of the derived programs. This is discussed in detail in Naqvi (1980) and the algorithm for deriving the programs is also given there. We can show the finiteness and completeness of our method (Naqvi 1980). Although we have considered a first order theory without function signs the method is applicable to data bases containing function signs. The termination condition, however, may not be rigorous in this case. This is an obvious area for further research.

References

Chang, C. L., (1979) "On Evaluation of Queries Containing Derived Relations in a Relational Data Base", Workshop on Formal Bases for Data Bases, Toulouse, France.

Codd, E.F., (1970) "A Relational Model of Data for Large Shared Data Banks", CACM 13, 6, 377-387.

Naqvi, S. (1980) "Deductive Question-Answering in Recursive Rule-Based Systems", Ph.D. Diss., Northwestern University, (in preparation).

Reiter, R., (1977) "An Approach to Deductive Question Answering", BBN Tech. Report no. 3649.

Sickel, S. (1976) "A Search Technique for Clause Interconnectivity Graphs", IEEE Trans. on Computers, Vol. C-25, No. 8.