ICOT Technical Report: TR-183

TR-183

Compiling Horn Clause Queries in Deductive. Databases: A Horn Clause Transformation Approach

by
N. Miyazaki (Oki Electric Industry Co. Ltd.)
H. Yokota and H. Itoh (ICOT)

June. 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Compiling Horn Clause Queries in Deductive Databases: A Horn Clause Transformation Approach

Nobuyoshi Miyazaki,
(Oki Electric Industry Co. Ltd.)

Haruo Yokota* and Hidenori Itoh
(Institute for New Generation Computer Technology)

*(H. Yokota is currently at Fujitsu Laboratories Ltd.)

June 1986
Revised, October 1986

Abstract

One way to construct a deductive database system is to combine a deductive component with a relational database management system. The deductive component accepts Horn clause queries and compiles them into relational operations. A method to compile queries is proposed in this paper. The method first transforms queries to their equivalent normal forms by partial evaluation and then compiles them into iterative relational operations. By transforming queries, we eliminate intermediate predicates and we can handle queries more efficiently in subsequent processing. This transformation is called Horn Clause Transformation (HCT). HCT may be used as a preprocessing for any other query processing methods. After HCT, queries are processed by simultaneous least fixed point operations. We show that complex mutual recursions as well as simple recursions can be reduced to simple iterations. The compilation algorithm for queries that use "not" predicate to express negation as failure, termination condition of compiled programs, and some ways to improve the performance of resulted programs are also presented.

1. Introduction

Deductive database system has first-order logic as its theoretical basis. The database consists of a finite set of constants and a set of first-order clauses [Gallaire84]. We discuss here only definite deductive databases whose clauses are restricted to definite clauses (Horn clauses). It is easily shown that deductive database systems are relationally complete if they are augmented with negation as failure capability under the closed world assumption [Codd72], [Clark78], [Reiter78a], [Kunufuji82]. A deductive database system is one of the candidates for the kernel of large scale knowledge base system in the Fifth Generation Computer Systems Project. We have investigated this subject from the beginning of the project [Kunufuji82], [Miyazaki82], [Yokota84], [MurakamiK84], [Yokota86a], and the work reported here is a part of the efforts to develop a distributed knowledge base system [Itoh86].

Reiter [Reiter78b] proposed a method to design deductive database systems by combining deductive components with relational database management systems (RDBMS). If the databases are large, this method provides a way to construct an efficient system, because we can use existing systems or apply known techniques to manage fast access paths to data and to optimize the query processing. In this approach, deductive components first process the queries and generate programs of relational operations. If the generation of these programs are separated their execution, this process is called compilation. The compilation is fairly straightforward for non-recursive queries, but handling recursive queries is more difficult because of the termination condition problem, i.e., the processing may result in an infinite loop. If the query involves only one recursion, the answer can be obtained by computing the least fixed points (or transitive closure) of the recursive expression [Aho79], and we can easily design the deductive component using theorem provers. One example of such systems based on deferred evaluation method is reported in [Yokota84] [MurakamiK84]. Similar methods are discussed in [Chakravarthy82], [Ioannidis86], [Valduriez86].

Several methods have been proposed to handle more complex queries. Henschen and Naqvi [Henscen84] proposed a method using connection graphs. This method can handle complex queries in principle, but compiled programs are mutually recursive programs which are difficult to implement if queries have

mutual recursion in them. A setting evaluation method was proposed to handle recursions by converting queries to iterative instance enumeration [Yokota86a]. Handling complex recursive queries is easier in this method because the enumeration process is not recursive. The advantage of this method is that compilation is easy because detection of the recursion is not necessary. However, this fact is also a disadvantage because even non-recursive queries require iterative enumeration. The evaluation of extensional database is also not efficient because it has to compute unnecessary intermediate results. Related topics are also discussed in [Chakravarthy81], [McKay81], [Lozinski85], [Ullman85], [Han86].

Another important problem to design deductive database systems is how to realize the negation as failure capability [Clark78],[Chandra85]. Although the system reported in [Yokota84] has this capability for some special cases, the way to design compilation method for complex queries is not known. Vielle [Vielle86] proposed a method to handle Horn clause queries not using compilation, and discussed the way to handle "not" in complex queries.

In this paper, we propose a method to compile deductive queries to programs of relational operations. Our approach is to transform complex queries to simpler equivalent forms before actual compilation. By doing this, queries can be processed more easily and efficiently in subsequent processing. This transformation process can be used as a preprocess of any query evaluation method proposed in other literatures.

After simplification of queries, we convert queries to relational operations. We can handle any recursive queries by reducing the problem to simultaneous least fixed point operation, which can be realized by simple iterative programs. We also discuss the way to handle negation as failure as well as its termination condition. The characteristics of the method are summarized as follows.

(1) It reduces complex recursions to simple recursions and then handles them by iteration.

(2) It converts non-recursive queries to non-iterative programs of relational operations.

(3) Evaluation of non-recursive intermediate expressions is not necessary.

(4) It can handle negation as failure ("not" predicates).

Following topics are presented in subsequent sections. Section 2 shows the outline of the method. Transformation method of Horn clause queries to simpler forms is discussed in section 3. The conversion of a query to relational operations and execution of the converted result are described in section 4. The way to improve efficiency of the algorithm is discussed in section 5.

2. Outline of the compilation

A deductive database consists of a set of Horn clauses. A set of ground unit clauses is called as extensional database (EDB) and stored in a relational database. The mapping between these clauses and relations is a well known one-to-one correspondence between a fact and a tuple based on first-order logic [Gallaire84]. For instance, a fact parent(taro,jiro) corresponds to a tuple <taro,jiro> in parent relation. Other clauses (rules) belong to intentional database (IDB). A query consists of a goal clause and a set of rules. Thus, we use rules given in a query and those in IDB to answer the query. These rules are combined together during or before compilation. Because IDB includes rules that are not related to a query, rules necessary for the query must be extracted first. Once the extraction is done, we do not have to distinguish these two set of rules. Therefore, these two rule sets are not distinguished in most part of this paper. Horn clause queries may be ad hoc queries asked by human users or embedded in a program [Miyazaki82] [Miyazaki86].

In our model, a deductive database system consists of a deductive component and a relational database management system (RDBMS). These two components are sometimes called as an intentional processor and an extensional processor. The deductive component accepts a query and compiles it into a program that includes a set of relational queries. Then, it computes the result of the query by executing the compiled program using RDBMS.

The query processing algorithm is divided into three stages.

(1) Horn clause transformation (HCT):

The system analyses a given query and detects recursive predicates. It also transforms the query to an equivalent extensional normal form. The extensional normal form is defined as a set of clauses whose body contain only recursive predicates, extensional predicates, and comparisons. Extensional predicates are special predicates that indicate that corresponding facts are stored in an extensional database. Extraction of necessary clauses are also done by HCT.

(2) Generation of a program containing relational queries:

The resulted program is iterative if the query includes recursive predicates, and it is non-iterative if the query does not involve recursion.

(3) Execution of the compiled program.

The basic algorithm of HCT and its extension to more general cases are discussed in Section 3. Generation of the program of relational operations is similar to that in [Yokota86a], and we discuss its basic algorithm and the termination condition of the resulted program in following sections.

3 Horn Clause Transformation

3.1 Basic ideas

We assume that the rules given in a query and IDB are combined together before transformation begins. In other word, rules are treated as if all of them are given in a query. Horn clause transformation (HCT) is an algorithm that detects recursions in a Horn clause query and transforms the query to an equivalent extensional normal form that contains only certain types of clauses. HCT is done by deferring the evaluation of predicates that require the extensional database or that are recursive during the evaluation of the query by theorem provers such as Prolog. HCT can be done either depth first like Prolog

or breadth first like retrieval by unification proposed in [Yokota86b] [MurakamiM85] [Morita86]. If there are rules not related to the query, HCT extracts only necessary rules from the rule set.

We illustrate the principle of HCT first by an example.

[Example1]

```
:- ancestor(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
parent(X,Y)   :- father(X,Y).
parent(X,Y)   :- mother(X,Y).
father(X,Y)   :- edb(father(X,Y)).
mother(X,Y)   :- edb(mother(X,Y)).
```

Here, edb is an extensional predicate that indicates that the corresponding facts are stored in EDB. An AND/OR graph expansion of the query is shown in Figure 1. It is a partial AND/OR tree expansion, and is an AND/OR graph because there is an implicit edge between "ancestor" nodes to express recursion.

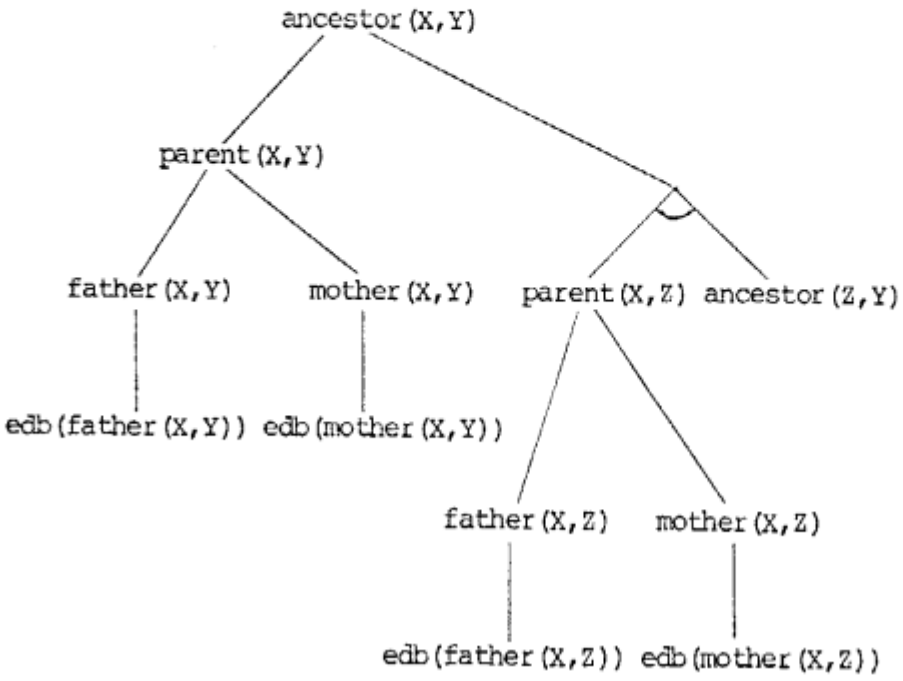

Figure 1 AND/OR expansion of ancestor

If we find a predicate that is same as one of its ancestors during the expansion, we do not expand it further but memorize it as a recursive predicate. Extensional predicates and comparison are not evaluated too. Other predicates are expanded. Thus, AND/OR expansion of Figure 1 is equivalent to that in Figure 2.

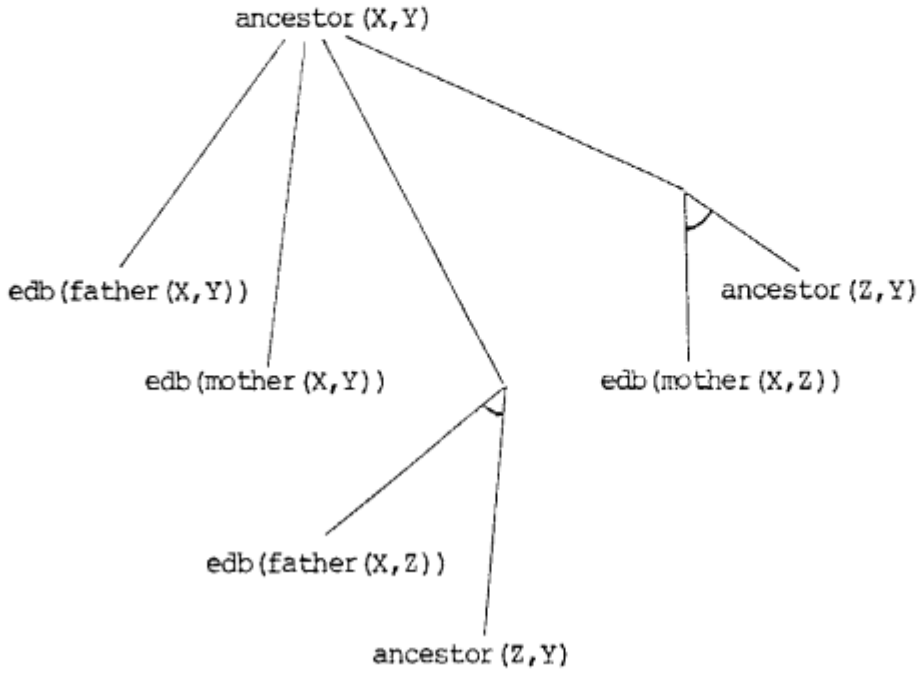

Figure 2 Equivalent AND/OR graph to Figure 1.

A set of Horn clauses which corresponds to Figure 2 is as follows. This is the result of HCT, and ancestor is detected as a recursive predicate.

```
[Result of HCT]
     :- ancestor(X,Y).
     ancestor(X,Y) :- edb(father(X,Y)).
     ancestor(X,Y) :- edb(mother(X,Y)).
     ancestor(X,Y) :- edb(father(X,Z)),ancestor(Z,Y).
     ancestor(X,Y) :- edb(mother(X,Z)),ancestor(Z,Y).
```

We can consider above set of clauses as a query equivalent to the original query. Because the transformed query does not include intermediate predicate such as "parent" and we know whether the query is recursive or not, it is easy to convert it to a program with relational operations as discussed in Section 4.

HCT handles queries by partial evaluation and can be regarded as a variation of deferred evaluation algorithm. It detects and stops the expansion of recursive predicates during the AND/OR expansion of a query. If a predicate that corresponds to an OR node is expanded, its parent node becomes an OR node instead of the expanded (eliminated) node. Therefore, it is an algorithm that moves OR nodes nearer to the root. It handles AND nodes in similar way.

3.2 Basic Algorithm of Horn Clause Transformation

The realization of HCT can be done by depth first way or breadth first way. Because we can use backtrack capability of a logic programming language, the implementation by depth first would be easier than the one by breadth first. However, the depth first way is not suitable to explain the algorithm because of backtracking. Therefore, we show here the basic breadth first algorithm.

To simplify the basic algorithm we restrict the syntax of clauses as follows. The expansion of HCT to relax these restrictions is discussed in Section 3.3.

(1) Predicates that appear in bodies and the goal are either extensional, comparison, or those which appear in heads of clauses.

(2) Extensional predicate has only one argument that is a function to indicate corresponding relation. The arguments of this function must be variables. Further, the arguments of predicates other than comparison are also variables.

(3) Each variable in arguments of head predicate must appear as arguments in the body of the same clause.

(4) There are no clauses without body part, i.e., all unit clauses are assumed to be in EDB.

The first restriction is introduced to simplify the discussion. We later discuss the way to handle a second-order predicate "not" for negation as failure capability. The second restriction is to guarantee that all related clauses are inspected when the AND/OR tree expansion is halted by the detection of recursive predicates. Because this restriction is too strong, the way to relax it is also discussed in later section. The third and fourth restrictions prevent (part of the) result being obtained without referring EDB.

The syntax of clauses is similar to the one in DEC-10 Prolog. We use a kind of pseudo-clauses in the form [head :- body| edb-body| predecessors| recursive-prd.] to express temporary result in the process of computation. Here, "head" and "body" correspond to ordinary head and body respectively, "edb-body" is a transformed expression to be evaluated using EDB, "predecessors" is a set of predicates that have been deleted from the body during AND/OR tree expansion, and "recursive-prd" is a set of predicates that are recursive.

First, we introduce a partial evaluation algorithm for breadth first expansion. More detailed description is found in Appendix A.

```
Procedure BFPE(goal, IDB, Temp-rules, Subgoals);
    /* Breadth First Partial Expansion */
    /* Input: goal, IDB(a set of rules in the query) */
    /* Output: Temp-rules   */
    /* Input & Output: Subgoals */
  begin;
    recursive-prd := Subgoals;
    /* Expansion of the root */
     Find rules in IDB that unify to goal and
       construct a set of pseudo-clauses;
    /* expansion of nodes other than root */
     repeat until no change occurs;
       for all pseudo-clauses do;
         change body of pseudo-clause by moving
           predicate not to be expanded to edb-body;
           /* edb, comparison and recursive ones are not expanded. */
         if predicate is found recursive add it to recursive-prd:
```

```
        /* It is recursive if it is found in predecessors. */
      end for;
     /* expand nodes */
      for every pair of T in the set of pseudo-clauses
         and R in IDB do;
       if the left most predicate of the body in T
         unifies head of R then construct
         new pseudo-clause from them;
      end for;
    end repeat;
    Subgoals := recursive-prd;
  end BFPE;
```

The Horn clause transformation is done by using BFPE as follows.

```
Procedure HCT(goal, IDB, Transformed-rules);
  begin;
    /* detects the recursive predicates */
      recursive-prd := {};
      call BFPE(goal, IDB, Temp-rules, recursive-prd);
    /* check if the resut is already obtained */
      if recursive-prd = {} or goal is the only element
        of it then do;
          Transformed-rules := {[head :- edb-body] of Temp-rules};
      end else do;
    /* reprocess if the query is complex */
         for goal and every element of recursive-prd R do;
          call BFPE(R, IDB, Temp-rules, recursive-prd);
          Transformed-rules := ++ {[head :- edb-body] of Temp-rules};
            /* "++" means add new elements */
        end for;
      end else;
    end HCT;
```

Because the basic algorithm for Horn clause transformation stops expansion when it detects recursive predicates, and because the number of the clauses in IDB (query) is finite, the algorithm always halts for any query. The algorithm just simulates the breadth first way of first-order theorem provers except that it defers the evaluation of recursive predicates and other special predicates. Therefore, the result of the Horn clause transformation is equivalent to the original query.

The basic algorithm consists of recursion detective phase and transformation phase. The reason for this configuration is that the first phase itself is not sufficient if there are recursive predicates other than the goal. If the goal is only recursive predicate in the query, the first phase generates required result.

An example that requires both phases is shown below. The arguments of predicates are not shown because it is not essential in this example.

[Example2]

```
 :- a.
 a :- edb(g).
 a :- b, edb(c).
 b :- edb(h).
 b :- d, edb(e).
 d :- edb(f),b,a.
```

There are three mutually recursive predicates "a", "b" and "d". The result of the first BFPE call is as follows.

```
:- a.
a :- edb(g).
a :- edb(h),edb(c).
a :- edb(f),b,a,edb(e),edb(c).
```

Two recursive predicates "a" and "b" is found, and "d" is eliminated. Recursive predicates that are not self recursive may be eliminated by transformation. Self recursive predicate is defined as a predicate that includes itself in clauses defining it. The above result is almost equivalent

to the original query, but it is not evaluable with EDB because the definition of "b" is not shown in the transformed form. Therefore, we use procedure BFPE again for the goal "a" and subgoal "b". Note that recursive-prd is set to {a,b} when BFPE is called. The result of the second phase is as follows.

```
:- a.
a :- edb(g).
a :- b,edb(c).
b :- edb(h).
b :- edb(f),b,a,edb(e).
```

Thus, we obtain an extensional normal form for the query.

3.3 The Property of Horn Clause Transformation

Main features of HCT are as follows.

(1) HCT extracts necessary rules to answer a query from rules in the query and IDB. IDB may be very large and only a fraction of it is used to answer a query. The extraction should be done at early stage for efficiency reasons, and HCT does it during transformation process.

(2) HCT transforms a rule set of a query to an equivalent extensional normal form. This means elimination of non-recursive predicates that define virtual relations.

(3) HCT eliminates some of the recursive predicates that are not self recursive in original form.

The processing of a complex query is a lot more time consuming than the simple one, although we can handle any recursions as shown in Section 4 and [Yokota86a] [Vielle86]. Therefore, transforming a query into an extensional normal form and eliminating some of the recursive predicates reduce the time to evaluate a query. The time saving effect is very large when mutually recursive query is transformed to a simple recursive one, because the latter can be evaluated efficiently as discussed in section 5.

It is interesting to see the effect of HCT in terms of connection graph (CG) used in [Henscen84]. A CG of a query is a directed graph where each node represents a clause and each edge represents unifiability between a pair of clauses. An edge is directed from a predicate in body to a unifiable predicate in head. A potential recursive loop (PRL) is a cycle in CG. An edge leading from a cycle out of it is called exit edge. The AND/OR expansion of the query corresponds to traversing its CG. Roughly speaking, HCT transforms a PRL of any length to the one with length 1 unless it intersects another PRL. An existing edge leading to a node that defines non-recursive intermediate predicate is transformed to the one or more edges leading to extensional predicates.

3.4 The Extension of Horn Clause Transformation

The restriction of the query form introduced in the previous section is too strong if we want a deductive database system that is truely improvement over the relational database system. Therefore, we discuss how to relax these restrictions in this section. First we discuss the way to handle negation as failure capability, and then we allow the use of constants in arguments of predicates. There are ways to relax other restrictions, but these two capabilities sufficiently broaden the function of the system.

(1) Introducing second-order predicate "not"

To realize negation as failure capability, we introduce a second-order predicate "not". The "not" roughly has same meaning as DEC-10 Prolog. If we allow "not" in a deductive database system, it is relationally complete [Codd72], [Kunufuji82].

The handling of "not" is similar to recursive predicates. The evaluation algorithm moves "not" from the body to edb-body in a pseudo-clause. At the same time, it memorizes content of "not" as a subgoal that must be expanded later. A recursive predicate is already expanded when the algorithm detects it, because the detection is done by finding same predicate to its ancestor node during expansion. On the other hand, the content of "not" is not expanded when the algorithm finds it. Therefore, the content of "not" must be expanded later. The detection phase of the HCT must call BFPE repeatedly if there are "not" predicates in the query.

An example of the use of "not" is shown below.

[Example 3]

```
:- a.
a :- edb(b).
a :- edb(c),a,not(d).
d :- e.
e :- f.
e :- f,e.
f :- edb(f).
```

We find that "a" is recursive and "d" is in "not" by the first call of BFPE. Therefore, we expand "d" by calling BFPE, and find "e" is recursive. Thus, we expand "a", "d", and "e" in transformation phase and get the following result.

```
:- a.
a: - edb(b).
a :- edb(c),a,not(d).
d :- e.
e :- edb(f).
e :- edb(f),e.
```

The algorithm with the above extension halts because there are only finite number of clauses.

(2) Use of constants in arguments

By allowing constants or instantiated variables in the arguments of a goal, we can improve the capability of the system. We can also simplify the clauses by allowing constants as arguments of usual predicates, although we can specify the equivalent query without this capability. The clause [a(X,Y) :- b(X,Z,jiro), c(Z,taro).] is more elegant than [a(X,Y) :- b(X,Z,Z1), c(Z,Z2), Z1=jiro, Z2=taro).]. The use of constants in heads is also sometimes convenient.

Constants in a goal can be handled by converting it to variables before the HCT and reconverting it to constants after transformation. Constants in other predicates can be handled by similar way. But this method may expand clauses that are not really used in the usual theorem prover, and generate unnecessary clauses that do not produce any answer.

The other method to handle constants is to change the detection phase. If a recursive predicate is found then we call BFPE repeatedly until its most general form is found.

The following is an example of constants in query.

[Example 4]

```
:- ancestor(taro,X).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
parent(X,Y)   :- father(X,Y).
parent(X,Y)   :- mother(X,Y).
father(X,Y)   :- edb(father(X,Y)).
mother(X,Y)   :- edb(mother(X,Y)).
```

[Result of HCT]

```
:- ancestor(taro,Y).
ancestor(X,Y) :- edb(father(X,Y).
ancestor(X,Y) :- edb(mother(X,Y).
ancestor(X,Y) :- edb(father(X,Z)),ancestor(Z,Y).
ancestor(X,Y) :- edb(mother(X,Z)),ancestor(Z,Y).
```

The head of resulted clauses is ancestor(X,Y) rather than ancestor(taro,X) because the latter is not the most general form in the expansion.

4 Compiling Queries into Relational Operations

The transformed queries are compiled into programs of relational operations, and they are evaluated using EDB. The conversion process is similar to the one used in [Yokota86a]. Note that we have detected recursions and eliminated intermediate predicates before conversion, although most of the discussions in Section 4 and 5 can be applied to original queries. Unit resolution is used in [Yokota86a] to prove the equivalence of the iterative instance enumeration to the original query. Similar proof can be given for our algorithm. But we use relational calculus to explain the conversion so as to show the correspondence of two expressions. We assume at first that there are no constants in goal and head predicate.

4.1 Semi-Domain-Relational-Calculus and Semi-Relational-Algebra

4.1.1 Semi-Relational-Calculus

If the bodies of clauses consist only of extensional predicates and comparisons, the heads are predicates that match the goal. Therefore, there is the following correspondence between a Horn clause and domain relational calculus.

r :- q1,q2,.....,qn.

<---->

∀(variables not in qi)

∃(variables of qi)(r v ~q1 v,....,v ~qn)

<---->

{r(t1,...,tk)|∃(variables in q_i)(q1 ∧ ,...,∧qn ∧ C_F}

C_F is a conditional expression for the variables and constants.

If there are more than one clause with same head predicate, the corresponding expression in relational calculus is "OR"ed. In usual domain calculus, all relations that appear in expression must be real relations in EDB. In our semi-domain-relational-calculus, the expression may include not only real relations but also virtual relations that are defined in other expressions. Thus, a query is expressed by a set of expressions that define virtual

relations. If these expressions involve no recursions, the difference between usual calculus and semi-calculus is nothing but mere notation. We allow recursive expressions in semi-relational-calculus.

4.1.2 Semi-Relational-Algebra and Conversion Rules

A query in semi-relational-calculus has an equivalent semi-relational-algebra expressions. Semi-relational-algebra expressions can be obtained by expressing each calculus expression by relational algebraic way. It is extended algebra because we allow recursive expressions in semi-algebra.

A query expressed in Horn clauses can be converted to semi-relational-algebra by following conversion rules. These rules are used to covert each clause to semi-algebraic expression. The expression corresponding to a query is obtained by union of converted expressions that correspond same head. Constants and instantiated variables in goal are regarded as a selection operator over the result in this subsection.

(1) Conversion of clauses without "not"

The conversion of clauses that do not involve "not" predicate is done by following conversion rules.

```
  r :- q1,q2,...,qn,p1,p2,...,pj.
--->
  r = π(σF(q1*q2*...*qn))                (I)
```

Here, p1,p2,...,pj are comparison operators, q1,q2,...,qn are either extensional predicates or recursive predicates. They are also used to express relations corresponding to predicates with same names. "σ_F" is a selection operator that corresponds comparison predicates and equivalent variables in the arguments of predicates. "π" is a projection operator which corresponds the mapping of arguments in body to those in head. "*" denotes Cartesian product of relations.

The conversion rule (I) is a general formula, and the converted expression is not in efficient form if it is directly evaluated because of Cartesian product. Therefore, we should distribute conditions in selection operator over relations and use join operations to evaluate the result.

(2) Conversion of "not" predicate.

We have to restrict the syntax of Horn clause if we allow "not" in it. The reason of this restriction is that some of the clauses such as r(t) :- not(g(t)) is not convertible to safe expressions in relational calculus and do not have corresponding algebraic expressions. The restriction is not very strong in practice, because most expressions excluded by the restriction can not be used to produce values in theorem provers such as Prolog.

[restriction of syntax for "not" predicate]

(a) Each of variable arguments of the predicate in "not" appears in other (not in "not") predicates, or it is connected to arguments in other predicates by "equal" condition.

(b) The arguments of predicates in "not" that do not satisfy (a) are not arguments of predicate in head.

(c) Those arguments of predicate in "not" that do not satisfy (a) nor (b) must be instantiated to constant at run time.

The restriction (a) and (b) can be checked before program containing the query runs, but (c) is known only when the query is executed. The clause containing "not" is converted to semi-relational-algebra expressions as follows.

[Definition] Semi-difference: $-_{(f)}$

A semi-difference $R -_{(f)} S$ of relations R and S is defined with a condition f on attributes of R and S being given.

$$R -_{(f)} S = R - R \triangleright f \triangleleft S$$
$$= R - \pi_R (R \triangleright f \triangleleft S),$$

where "-" is difference operator, "▷f◁" is join, and "▷f<" is semi-join.

[conversion rule]

r :- p1,p2,...,pi,not(q1),not(q2),...,not(qj).

---->

$$r = \pi(p - \bigcup_{i=1}^{j} (p \triangleright f_i < q_i))$$

$$= \pi(\bigcap_{i=1}^{j} (p - {}_{(f_i)} q_i))$$

$$= \pi(\ldots((p - {}_{(f_1)} q1) - {}_{(f_2)} q2)\ldots - {}_{(f_j)} qj), \quad \ldots\ldots (II)$$

where p is an expression given by converting p1,p2,...,pi by rule (I) and eliminating projection. Many clauses can be converted to algebraic expressions involving difference rather than semi-difference in practical cases.

4.2 Evaluation of queries without negation as failure capability

If the original query is not recursive, the converted query in semi-relational-algebra is not recursive. Moreover, it does not include virtual relations in the right hand side of the expression, because corresponding clauses are eliminated by HCT. Therefore, it is just an expression in relational algebra and can be evaluated by a relational database management system. Even if expressions include virtual relations, they can be evaluated one by one, or resultant set of expressions can be reduced to a single expression by substituting expressions of virtual relations.

On the other hand, the result of conversion can not be reduced to ordinary relational algebra if the original query is recursive. The evaluation of a recursive query is discussed in this section. Suppose a query is expressed by a set of semi-relational-algebra expressions as follows.

$$r_i = f_i(q_{i1}, q_{i2}, \ldots, q_{ij}, r_{i1}, r_{i2}, \ldots, r_{im}) \quad (i=1,\ldots,n) \quad \text{---} \quad (III)$$

where ri's are virtual relations that correspond to the goal or recursive

predicates, and qi's are real relations corresponding to extensional predicates. We use an iterative algorithm to evaluate the query. The algorithm calculates simultaneous least fixed points (SLFP) of above virtual relations. Usual algorithm to compute least fixed point (LFP) is just a special case of our algorithm as shown in Section 5.

Let us consider following sequence of relations.

$$r_i^{k+1} = f_i(q_{i_1}, q_{i_2}, \ldots, q_{i_j}, r_{i_1}^k, r_{i_2}^k, \ldots, r_{i_m}^k) \qquad (i=1,\ldots,n)$$
$$r_i^0 = \{\},$$

Then, SLFP are given by,

$$r_i = \lim_{k\to\infty} r_i^k . \qquad (i=1,2,\ldots,n) \qquad \ldots(IV)$$

[Theorem 1]

The algorithm (IV) converges in finite times of iteration, and the result of (IV) satisfies (III).

This Theorem is an extention of LFP operation discussed in [Aho79]. The proof is given in Appendix B. The principal reason why SLFP exist is that operators of relational algebra is monotone except for difference. Thus, we can evaluate a recursive query by SLFP operation, and the problem is reduced to simple iterative program. Note that the algorithm gives least fixed points, but not every possible answers for the query. For instance, any relations satisfy [r=r], but the LFP of r is {}. This algorithm is slightly different from the one in [Yokota86a], which is given as (IV') below.

$$r_i^{k+1} = r_i^k \cup f_i(q_{i_1}, q_{i_2}, \ldots, q_{i_j}, r_{i_1}^k, r_{i_2}^k, \ldots, r_{i_m}^k)$$
$$r_i^0 = \{\}$$

$$r_i = \lim_{k \to \infty} r_i^k \quad . \qquad (i=1,2,\ldots,n) \qquad (IV')$$

It is easily shown that (IV') always converges and if (IV) converges both give the same result. The evaluation by (IV) is more efficient than by (IV').

4.3 Evaluation of queries with negation as failure capability ·

A query that includes "not" predicate can be converted to a set of semi-relational-algebra expressions as shown in previous sections. However, the algorithm (IV) may not converge in this case because difference (semi-difference) operation is not monotone. We discuss the condition for the convergence, and give a sufficient condition.

Consider an AND/OR graph that corresponds to a query in extensional normal form. "Not" predicate is expressed as a node whose content (predicate) is a goal of subgraph that corresponds to the AND/OR expansion of the predicate. The edges of the graph are directed from parents nodes to child nodes in AND/OR expansion. We may decompose the graph into a graph of subgraphs. A subgraph is said evaluable if it does not have outgoing edges nor nodes corresponding to "not" predicates. Any predicates in evaluable subgraph can be evaluated using the algorithm discussed in the previous sections.

We introduce a concept of decomposably evaluable as follows.

(1) An evaluable subgraph is decomposably evaluable.

(2) A subgraph is decomposably evaluable if all of its outgoing edges point to decomposably evaluable subgraphs.

Thus, there is a class of queries whose equivalent AND/OR graph is decomposed into a graph of decomposably evaluable subgraphs. These graphs are said decomposably evaluable. We can check the decomposability of an AND/OR graph corresponding to a query as follows.

(1) Select a head predicate.

(2) Expand it by BFPE in section 3.2. If it stops without finding "not" predicates, then add it to the set of decomposably evaluable predicates. If it detects "not" predicates check the content of it whether or not its content is in the set of decomposably evaluable predicates. If all the content of "not" are found in the set, add the expanded predicate to the set. Otherwise, go to (1)

(3) Stop if there are no head predicates not included in the set of decomposably evaluable predicates or there are no changes in the set while checking all the remaining head predicates.

The number of predicates in a set of clauses for a query is finite. Therefore, the checking of decomposability can be done in finite time. It is clear that we can evaluate decomposably evaluable query by evaluating subgraphs one by one starting from evaluable subgraphs. It is also clear that SLFP operation of the whole query converges if evaluation of subgraph by subgraph converges. Thus, we obtain the following theorem.

[Theorem 2]

The algorithm (IV) converges in finite number of iterations if the AND/OR graph corresponding to a given query in extensional normal form is decomposably evaluable. The result of the algorithm gives the answer of the query. The other way to handle queries with "not" is to use (IV') instead of (IV). Because (IV') always converges, we do not have to worry about infinite loops. But we have to check if the result satisfies (III) after computing it by (IV').

5. Efficiency Considerations

We have discussed the general algorithm for query evaluation in previous sections. However, this has two major drawbacks. One is the iteration involves certain redundancy. The other is the treatment of constants in the goal. Let us consider improvement of the algorithm in this section.

5.1 Redundancy elimination

A converted query expressed by (III) can be reformulated as shown below.

$$r_i = \lim_{k\to\infty} r_i^k$$

$$r_i^0 = \{\}$$
$$r_i^{k+1} = f_{i_1}(q_{i_1}, q_{i_2}, \ldots, q_{i_j}) \cup f_{i_2}(q_{i_1}, q_{i_2}, \ldots, q_{i_j}, r_{i_1}^k, r_{i_2}^k, \ldots, r_{i_m}^k)$$
$$(i=1,2,\ldots,n) \qquad \ldots(III')$$

where f_{i_1} corresponds those rules that do not contain recursive predicates and f_{i_2} corresponds other rules. Then,

$$r_i = \lim_{k\to\infty} r_i^k = r_i^N \quad (N \text{ is an integer: see Appendix B})$$

$$= f_{i_1}(q_{i_1}, q_{i_2}, \ldots, q_{i_j}) \cup f_{i_2}(q_{i_1}, q_{i_2}, \ldots, q_{i_j}, r_{i_1}^N, r_{i_2}^N, \ldots, r_{i_m}^N).$$

Therefore, we find followings.

(1) We do not have to evaluate f_{i_1} repeatedly. It can be evaluated at the beginning and is a part of the final result.

(2) We can start iteration from $r_i^0 = f_{i_1}$ instead of $r_i^0 = \{\}$, because $f_{i_2} = \{\}$ for $r_i^0 = \{\}$ $(i=1,2,\ldots,n)$.

(3) The goal can be evaluated after the iteration if it is not recursive itself.

Next, let us assume that there is only one recursive predicate in a query and the recursive predicate appear at most only once in the body of a clause. Then, $r = f1(\{q\}) \cup f2(\{q\},r)$, where $\{q\}$ is a short hand notation for $q1,q2,\ldots,qj$.

$$\begin{aligned} r^{k+1} &= f1(\{q\}) \cup f2(\{q\}, r^k) \\ &= f1(\{q\}) \cup f2(\{q\}, r^{k-1} \cup d(r^{k-1})) \\ &= f1(\{q\}) \cup f2(\{q\}, r^{k-1}) \cup f2(\{q\}, d(r^{k-1})) \\ &= r^k \cup f2(\{q\}, d(r^{k-1})) \end{aligned}$$

where we put $d(r^{k-1}) = r^{k} - r^{k-1}$.

Thus, we obtain $r^{k+1} = r^{k} \cup d(r^{k})$ if we put $d(r^{k}) = f2(q1,\ldots,qj,d(r^{k-1}))$. Therefore, we can get simple algorithm for this case as follows. Consider the sequence $d(r^{k})$ such that,

$$d(r^{0}) = f1(\{q\})$$
$$d(r^{k}) = f2(\{q\},d(r^{k-1})).$$

Then,

$$r = \bigcup_{k=0}^{\infty} d(r^{k}) = \bigcup_{k=0}^{N} d(r^{k}). \qquad \ldots\ldots\ldots (V)$$

Thus, we obtain usual transitive closure operation used to compute least fixed point [Aho79]. Algorithm (V) can be also used for a query that involves more than one recursive predicate and all of them are independent.

5.2 The treatment of constants

The constant or instantiated variables in the goal (or in rules) can be regarded as conditions of the query. The simplest way to handle constants is apply a selection operator after the evaluation without constants. However, this results in unnecessary computation because we have to compute whole recursive relations before applying the selection. This problem was discussed in [Aho79] [Henscen84] [Ullman85] [Vielle86]. [Aho79] analyses commutability of selection operator and LFP operation. [Vielle86] proposes a method that can be regarded as an extention of nested-loop join or tuple substitution method of relational database query processing. Others proposed methods that can be applied to special cases. We would like to preserve set oriented nature of relational operations because we are able to apply techniques developed for relational operation such as merge join and those used in database machines for set oriented operations. Although we have not obtained an efficient algorithm for general class of queries yet, we believe that preserving set oriented nature is essential in distributed environment to reduce interactions among related

sites.

Let us consider algorithm (V) again.

$$r = \bigcup_{k=0}^{\infty} d(r^k)$$
$$= \bigcup_{n=0}^{\infty} f2(\{q\}, f2(\{q\}, \ldots, f2(\{q\}, f1(\{q\}))))).$$

Let "σ" be the selection operator that corresponds constants in the goal. If $\sigma f2(\{q\},R) = f2(\{q\},\sigma R)$, then "$\sigma$" is commutative with LFP. That means, we can compute σR instead of R directly in the algorithm (V). This commutability can be checked during HCT. Applying the extended HCT discussed in 3.3, if the head predicate (recursive predicate) is as general as (or more general than) itself in body the selection is commutative. For instance, if the goal is ancestor(X,taro) instead of ancestor(X,Y) we can get a clause [ancestor(X,taro) :- edb(father(X,Z)), ancestor(Z,taro)] and another one in the same form. Because the head predicate is same as itself in the body, ?ancestor(X,taro) can be directly evaluated by algorithm (V). The evaluation is efficient because we can apply selection first strategy and the i+1th step of computation can be done using the result of the i-th step.

Next, we discuss the opposite case of the above, i.e., the case $\sigma f2(\{q\},r) = f2(\sigma\{q\},r)$ where $\sigma\{q\}$ means "σ" applies to some combination of q_j's in the expression f2. In this case, algorithm (V) results in $\sigma R = \bigcup f2(\sigma\{q\}, f2(\{q\},\ldots,f2(\{q\},f1(\{q\}))))$. It is not efficient to use this equation to compute the result. It is not efficient because we cannot use selection first strategy and reuse of the previous result at the same time. This redundancy can be sometimes avoided by storing the form of the subsequent steps and checks the redundancy as in [Yokota84], but this kind of checking is time consuming itself and sometimes fails to detect the redundancy. The other problem of (V) for this case is the difficulty to determine termination of the iteration.

This class of queries can be processed efficiently if they satisfy certain conditions discussed below. First, we assume that $f2(\{q\},R) = f2(Q,R)$ where Q is a relation expressed by $Q=g(\{q\})$. Queries that have only one clause corresponding to f2 always satisfies this condition. The "ancestor" example satisfies this although it has two clauses for f2. Second, we assume that there

exists f' such that f2(A,f2(B,C)) = f2(f'(A,B),C).

With these conditions, let

$s^{i+1} = f'(s^i, Q)$

$s^0 = Q$,

then, we can easily prove that

$d(r^{i+1}) = f2(Q, d(r^i))$

$= f2(s^j, d(r^{i-j}))$. .... (VIa)

Let j=i and we get

$d(r^{i+1}) = f2(s^i, d(r^0))$

$= f2(s^i, f1(\{q\}))$. ... (VIb)

Therefore, we can compute "r" by computing $d(r^i)$ by (VIa) and (VIb). Note that the termination condition is $s^{i+1} \subset \bigcup_{j=1}^{i} s^j$ but not $d(r^i) \subset r^i$.

Now, we return to the case $\sigma f2(Q,r) = f2(\sigma Q, r)$. Because $\sigma f2(A,f2(B,C)) = f2(\sigma A, f2(B,C)) = f2(f'(\sigma A, B), C)$, we can compute "$\sigma r$" by replacing s =Q with s =σQ in (VIa). This algorithm is efficient because it is selection first and uses previous result in subsequent processing. Similar algorithm for a special case of queries having only one recursive clause and its variations are discussed in [Han86] with comparison of performance.

We have discussed algorithm for two extreme cases. In more general case, i.e., $\sigma f2(\{q\},r) = f2(\sigma_1\{q\}, \sigma_2 r)$, we apply either algorithm (VI) using $f1(\{\sigma_2 q\})$ instead of f1({q}) in (VIb) or algorithm (V). In algorithm (VI), we have to find whether necessary two conditions hold for a given query. It is usually easy to find whether there exists Q = g({q}) for a given f2. For the second condition, f2(A,f2(B,C)) = f2(f'(A,B),C), we can usually find f' by comparing two alternative expressions (corresponding to r:-(<q>,<q>),r and r:-<q>,(<q>,r)) for a clause obtained by substituting r by its expression.

[Example5]

<Query for "ancestor's friend"> [Vielle86]

```
:- ancfr(X,Y).
ancfr(X,Y) :- edb(friend(X,Y)).
ancfr(X,Y) :- parent(X,Z), ancfr(Z,Y).
"Parent" is defined in terms of father and mother.

<The result of HCT>
:- ancfr(X,Y).
ancfr(X,Y) :- edb(friend(X,Y)).
ancfr(X,Y) :- edb(father(X,Z)), ancfr(Z,Y).
ancfr(X,Y) :- edb(mother(X,Z)), ancfr(Z,Y).

<Semi-Relational-Algebra>
```

$$ancfr = friend \cup \pi_{1,4}(father \underset{2=1}{\bowtie} ancfr) \cup \pi_{1,4}(mother \underset{2=1}{\bowtie} ancfr)$$

$$= friend \cup \pi_{1,4}((father \cup mother) \underset{2=1}{\bowtie} ancfr)$$

Thus,

$f1 = friend$

$f2 = \pi_{1,4}(Q \underset{2=1}{\bowtie} ancfr)$

$Q = father \cup mother$

$f' = \pi_{1,4}(A \underset{2=1}{\bowtie} B)$.

In the above example, possible query forms are ":-ancfr(X,Y).", ":-ancfr(c1,Y).", ":-ancfr(X,c2).", and ":-ancfr(c1,c2).". Corresponding "selection operators" are "all", "$\sigma_{1=c_1}$", "$\sigma_{2=c_2}$", and "$\sigma_{1=c_1 \wedge 2=c_2}$", respectively. Algorithm (V) can be used for condition "all". "$\sigma_{2=c_2}$" is commutative with LFP operation and algorithm (V) is applied. "$\sigma_{1=c_1}$" is not commutative with LFP, and algorithm (VI) is applied. Because "$\sigma_{1=c_1 \wedge 2=c_2}$" is partially commutative with LFP, we can apply either (V) or (VI) for this query.

6. Conclusions

We have presented a method to compile Horn clause queries in deductive databases. Most methods so far proposed in this field directly compile or interpret queries as they are given. In these methods, optimization is considered as selection of execution strategy. Complex queries would be

difficult to handle or result in redundancy in these methods. In our approach, queries are transformed to simple equivalent forms before actual compilation. This transformation, HCT, is a powerful tool to reduce the complexity of queries, and much of the work required to evaluate queries in subsequent steps is saved by it. HCT can be used as preprocessing of any algorithms which handle recursive queries, because it preserve syntax of queries.

The latter half of this paper discusses the generalization and improvement of the compilation method proposed in [Yokota86a]. We have shown that any queries can be converted to simple iterative programs of relational database operations, and have given a sufficient condition for the termination of the programs. Some ways to improve the efficiency of resulted programs are also discussed, although their applicability is limited.

We implemented a small prototype system that realizes HCT and simultaneous LFP. A depth-first partial evaluation instead of breadth-first is used in the prototype, because we implemented it in Dec-10 Prolog using meta-programming technique proposed by Bowen and Kowalski. The system uses a relational algebra simulator (also in Prolog) reported in [Yokota84]. Implementing optimization and handling "not" predicate are planned as the next step.

Acknowledgements

The authors appreciate valuable discussions with members of KBMS PMI project at ICOT and Oki Electric. The prototype was designed and implemented by Mr. H. Haniuda and Mr. Y. Abiru at Oki Electric.

[References]

[Aho79] Aho, A.V., Ullman, J.D., "Universality of Data Retrieval Languages", Proc. of 6th Symp. on Principles of Programming Languages, 1979.

[Chang81] Chang, C.L., "On Evaluation of Queries Containing Drived Relation in a Relational Data Base", in [Gallaire81].

[Chakravarthy82] Chakravarthy, U.S., Minker, J., Tran, D., "Interfacing Predicate Logic Languages and Relational Databases", Proc. of 1st Int. Conf. on LP, 1982.

[Chandra85] Chandra, A. K., Harel, D., "Horn Clause Queries and Generalizations" J. of Logic Programming, 1985, No. 2, pp. 1-15.

[Clark78] Clark, K.L., "Negation as Failure", in [Gallaire78].

[Codd72] Codd, E.F., "Relational Completeness of database Languages", in Data Base Systems (R. Rustin, ed.), Prentice-Hall, 1972.

[Gallaire78] Gallaire, H., Minker, J., (eds.) "Logic and Data Bases", Plenum Press, 1978.

[Gallaire81] Gallaire, H., Minker, J., Nicolas, J.-M., (eds.) "Advances in Data Base Theory, Vol. 1", Plenum Press, 1981.

[Gallaire84] Gallaire, H., Minker, J., Nicolas, J.-M., "Logic and Databases: A Deductive Approach", ACM Comp. Surveys, Vol. 16, No. 2, Jun. 1984.

[Han86] Han, J. and Lu, H., "Some Performance Result on Recursive Query Processing in Relational Databases", Proc. of Int. Conf. on Data Engineering, Feb. 1986.

[Henscen84] Henschen, L. J., Naqvi, S. A, "On Compiling Queries in Recursive First-Order Databases", J. of ACM Vol. 31, No. 1, Jan. 1984, PP. 47-85.

[Ioannidis86] Ioannidis, Y.E., Wong, E., "An Algebraic Approach to Recursive Inference", Proc. of 1st Int. Conf. on Expert Database Systems, Apr. 1986.

[Itoh86] Itoh, H., et.al., "KBMS PHI (1)" through "KBMS PHI (4)", Proc. of 32nd Conf. of IPSJ, Mar. 1986, (in Japanese).

[Kunufuji82] Kunifuji, S. and Yokota, H., "PROLOG and Relational Data Bases for Fifth Generation Computer Systems", Proc. of Workshop on Logical Bases for Data Bases, Toulouse, Dec. 1982.

[Lozinski85] Lozinski, E., "Evaluating Queries in Deductive Database by Generating", Proc. of Int. Joint Conf. on Artificial Intelligence, Aug., 1985.

[McKay81] McKay, D.P., Shapiro, S.C., "Using Active Connection Graphs for Reasoning with Recursive Rules", Proc. IJCAI, 1981.

[Miyazaki82] Miyazaki, N., "A Data Sublanguage Approach to Interfacing Predicate Logic Language and Relational Databases", ICOT Technical Memorandum, TM-001, Nov. 1982.

[Miyazaki86] Miyazaki, N., Yokota, H., Abiru, Y., Itoh, H., "Interfacing Prolog and Deductive Database in KBMS PHI", Proc. of National Conf. of IECEJ, Mar. 1986, (in Japanese).

[Morita86] Morita, Y., Yokota, H., Nishida, K., Itoh, H., " Retrieval By-Unification Operation on a Relational Knowledge Base", to appear in Proc. of 12th VLDB, Aug., 1986.

[MurakamiK84] Murakami, K., Kakuta, T., Miyazaki, N., Shibayama, S., Yokota, H., Delta Demonstration Team, "Delta Demonstration at ICOT Open House", ICOT Technical Memorandum TM-0085, Nov. 1984.

[MurakamiM85] Murakami, M., Yokota, H., Itoh, H., "Formal Semantics of a Relational Knowledge Base", ICOT Technical Report, TR-149, 1985.

[Reiter78a] Reiter, R., "On Closed World Data Bases", in [Gallaire78].

[Reiter78b] Reiter, R., "Deductive Question Answering on Relational Data Bases", in [Gallaire78]

[Ullman85] Ullman, J.D., "Implementation of Logical Query Languages for Databases", ACM TODS, Vol. 10, No. 3, Sep. 1985, pp. 289-321, also Stanford Univ. Report STAN-CS-84-1000, May 1984.

[Valduriez86] Valduriez, P., Boral, H., "Evaluation of Recursive Queries Using Join Indices", Proc. of 1st Int. Conf. on Expert Database Systems, Apr.

1986, also MCC Technical Rep. DB-069-85, Aug. 1985.

[Vielle86] Vielle, L., "Recursive Axioms in Deductive Databases: The Queries-subqueries Approach", Proc. of 1st Int. Conf. on Expert Database, California, Apr. 1986, also ECRC Internal Report KB-10 Dec. 1985.

[Yokota84] Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S., Murakami, K., "An Enhanced Inference Mechanism for Generating Relational Algebra Queries" Proc. ACM PODS, 1984.

[Yokota86a] Yokota, H., Sakai, K., Itoh, H., "Deductive Database System based on Unit Resolution", Proc. of Data Engineering, Feb. 1986, also ICOT Technical Report, TR-123, Jun. 1985.

[Yokota86b] Yokota, H., Itoh, H., "A model and an Architecture for a Relational Knowledge Base", Proc. of 13th Int. Symp. on Computer Architecture, Tokyo, 1986. also ICOT TR-141.

Appendix A: Breadth First Partial Expansion Algorithm in HCT

The following is the detail of the algorithm BFPE in Section 3.2.

```
Procedure BFPE(goal, IDB, Temp-rules, Subgoals);
    /* Input: goal, IDB(a set of rules in the query) */
    /* Output: Temp-rules   */
    /* Input & Output: Subgoals */
  begin;
    recursive-prd := Subgoals;
    /* Expansion of the root */
      Temp-rules := {};
      for every rule in IDB do;
        if goal <> head(rule) then do;
            /* "<>" means unifiable */
          construct a pseudo-clause T such that
            T := [goal :- body-with-sub<1>| {}| goal<0>|
```

```
            recursive-prd.};
          /* body-with-sub<n> corresponds the body of the
             clause with every predicate subscript
             by <n>. */
        Temp-rules := ++ T;
          /* ++ means add new elements */
    end if;
  end for;

/* expansion of nodes other than root */
  repeat until no change occurs in Temp-rules;
    /* change body of pseudo-clause by moving predicate
       not to be expanded. */
    repeat until no change occurs in Temp-rules;
      New-temp := {};
        for every T in Temp-rules do;
          if lmpb(T) is comparison or extensional or
            lmpb(T) <> an element of recursive-prd(T)
             then do;
            edb-body := ++ lmpb(T);
            body(T) := rb(T);
             /* lmpb(T) = left most predicate of body of T,
                rb(T) = body(T) - lmpb(T). */
          end if;
          if lmpb(T) <> an element of predecessor(T)
           then do;
            edb-body := ++ lmpb(T);
            body(T) := rb(T);
            recursive-predicate := ++ lmpb(T);
          end if;
          eliminate elements of predecessors(T) with
           subscript not less than the subscript of lmpb(T);
        end for;
      end inner repeat;
     /* expand nodes */
      New-temp := {};
      for every pair of T in Temp-rules and R in IDB do;
```

```
        if lmpb(T) <> head(R) then do;
          N := [head(T) :- body(R)<+1>,rb(T)| edb-body(T)|
                predecessors + lmpb(T)| recursive-prd(T).];
          New-temp := ++ N;
        end if;
      end for;
      Temp-rules := {pseudo-clauses with empty body in
                     Temp-rules} + New-temp;
    end outer repeat;
    Subgoals := ++ set of recursive-prd in Temp-rule;
  end BFPE;
```

Appendix B: Proof of Theorem 1

[Theorem 1]

The algorithm (IV) converges in finite times of iteration, and the result of (IV) satisfies (III).

[Lemma]

For any h,

$r_i^{h+1} \supset r_i^h \quad (i=1,2,\ldots,n)$

[Proof of Lemma]

$r_i^{h+1} = fi(\{q\},\{r^h\})$

$r_i^0 = \{\}$

where $\{q\}$ and $\{r^h\}$ are short hand notation for $q_{i_1},\ldots,q_{i_j}$ and $r_{i_1}^h,\ldots,r_{i_m}^h$ respectively.

We prove the Lemma by mathematical induction.

(1) For h=1:

Because $r_i^0 = \{\}$, $r_i^1 \supset \{\} = r_i^0$.

(2) For h>1:

Let us assume Lemma holds for h=j, i.e., $r_i^{j+1} \supset r_i^j$. Because relational operations other than difference are monotone,

$$\begin{aligned} r_i^{j+2} &= fi(\{q\}, \{r^{j+1}\}) \\ &\supset fi(\{q\}, \{r^j\}) \\ &= r_i^{j+1}. \end{aligned}$$

Q.E.D.

[Proof of Theorem 1]

Let D be the set of all constants in EDB. Then, each r_i (for i=1,2,..,n) is a subset of finite (Cartesian) products of D. By Lemma, $r_i^{j+1} \supset r_i^j$. Therefore, each r_i^h (h=1,2,...) is a bound and monotone sequence of relations. This means each r_i^h converges for any i and its limit r_i exists.

Let us assume that there exists the minimum N such that $r_i^{N+1} = r_i^N$ (for any i). Then, $r_i^{N+j} = r_i^N$ for any j>0 by the definition of r_i^h, and $\lim_{h \to \infty} r_i^h = r_i^N$. Because such limit is finite set, there always exists such N. This means,

$$\begin{cases} r_i^{N+1} = fi(\{q\}, \{r_i^N\}) \\ r_i^{N+1} = r_i^N = r_i. \end{cases}$$

Therefore, algorithm (IV) gives fixed points. Moreover, it is easy to show that (IV) gives the least fixed points, because the algorithm starts from {}.

Q.E.D.