TR-214

A Deductive Database System Written in Guarded Horn Clauses

by

N. Miyazaki and Y. Mitomo*

(Oki Electric Co.)

H. Itoh and T. Takewaki

November, 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

A Deductive Database System Written in Guarded Horn Clauses

Hidenori Itoh, Toshiaki Takewaki,
Nobuyoshi Miyazaki† and Yuji Mitomo‡

ICOT Research Center
Institute for New Generation Computer Technology
Mita-Kokusai Bldg. 21F, 1-4-28, Mita,
Minato-ku, Tokyo 108, Japan

(†Oki Electric Industry Co. Ltd., ‡ Japan Systems Co. Ltd.)

Abstract

In this paper, we propose a deductive database system written in the parallel logic programming language GHC (Guarded Horn Clauses).

1. Introduction

Knowledge information processing systems have been proposed from the viewpoint of logic programming, such as an inference machine for efficient inference processing and a knowledge base system for efficient retrieval processing. Moreover, advanced knowledge representation languages are being developed in logic programming to implement the knowledge information processing systems. The advantages of developing these systems in a logic programming environment are developing and processing efficiency, and expansion of facilities.

In this paper, as a first step to the knowledge base system, we propose a deductive database system (DDS) model which stores and manages a set of Horn clauses as knowledge, and we show that its basic functions can be written in the logic programming language GHC (Guarded Horn Clauses) [Ueda 85], which was implemented at ICOT for the parallel inference machine.

Our DDS is composed of two components and has a set of rules comprising the intentional database (IDB) and a set of facts as the extensional database (EDB) [Gallaire 84, Deyi 84]. IDB and EDB are managed separately by their respective components. The first component manages IDB and compiles the queries from user or application programs into relational commands using IDB. The second component manages and retrieves items from EDB using their relational commands. IDB and EDB are assumed to be accessible in common by user or

application programs, and the size of EDB is assumed to be very large. Now, for efficient searching in and handling of EDB the second component is equipped with multiple dedicated retrieval processors (RP) controlled in parallel.

2. Guarded Horn Clauses

GHC is a simple, powerful and efficient parallel logic programming language [Ueda 85].

A GHC program is a finite set of guarded Horn clauses of the following form:

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n. \quad (m>0, n>0).$$

where H, G_i's, and B_i's are atomic formulas defined in the usual way. H is called a clause head, the G_i's are called guard goals, and the B_i's body goals. The notation '|' is a commitment operator. The part of a clause preceding '|' is called a guard, and the part succeeding '|' is called a body. Declaratively, the above guarded Horn clause is read as "H is implied by $G_1, \ldots,$ and G_m and $B_1, \ldots,$ and B_n".

A goal clause has the following form:

$$:- B_1, \ldots, B_n. \quad (n>0).$$

This can be regarded as a guarded Horn clause with an empty guard. A goal clause is called an empty clause when n is equal to 0.

We use symbols beginning with uppercase letters for variables and ones beginning with lowercase letters for function and predicate symbols. The nullary predicate 'true' is used to denote an empty set of guarded body goals.

The semantics of GHC are quite simple. Informally, to execute a program is to reduce a given goal clause to the empty clause by means of input resolution using the clauses constituting the program. This can be done in a fully parallel manner under the following rules of suspension:

(a) The guard of a clause cannot export any bindings to the caller of that clause, and

(b) the body of a clause cannot export any bindings to the guard of that clause before commitment

In this paper, we use a subset of GHC, called Flat GHC (FGHC). FGHC is allowed to have only system predicates in guards of clauses. In the following section, we introduce a deductive database system model.

3. Deductive Database System Model

A deductive database consists of a set of Horn clauses. A set of ground unit clauses is called an extensional database (EDB) and is 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 [Gallaire 84]. Other clauses (rules) belong to the intentional database (IDB).

Our deductive database system model consists of a deductive processing component and a relational database processing component as shown in Figure 1. Here, both IDB and EDB are used in common by user and application programs.

These two components are sometimes called an intentional processor and an extensional processor. The deductive processing component accepts a query and compiles it into an equivalent program that includes a set of relational queries. Here, a query consists of a goal clause and a set of rules. It computes the result of the query by executing the compiled program using the relational database.

The query deductive processing algorithm is divided into two 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 bodies contain only recursive predicates and comparisons. Extensional predicates are special predicates and indicate that corresponding facts

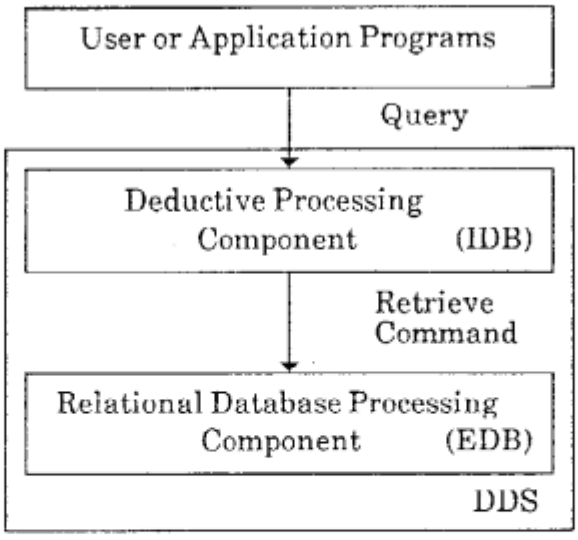

Figure 1 Deductive Database System Configuration

are stored in an extensional database. Extraction of necessary clauses is also done by HCT.

(2) Generation of a program containing relational queries:
The resultant program is iterative if the query includes recursive predicates, and it is non-iterative if the query does not involve recursion.

When applying the setting evaluation [Yokota 86], the query is always transformed into an iterative program even if it does not involves recursion, and the predicates eliminated by the partial evaluation technique in HCT have to be executed. Then, HCT is said to be an improvement method of the setting evaluation on these points.

The relational queries generated in the deductive processing component are sent to the relational processing component to retrieve from the EDB. The basic algorithm of HCT and the RP parallel control method are discussed in the following sections.

4. Horn Clause Transformation

The following is a basic algorithm for Horn clause transformation (HCT) written in GHC. In HCT, we introduce a basic partial evaluation algorithm for breadth first expansion.

```
Procedure HCT(goal, IDB, Transformed-rules);
   /* Input:    goal, IDB              */
   /* Output:   Transformed-rules      */

begin;
   /* search for the recursive predicates */
  recursive-predicates := [];
  call BFPE(goal, IDB, Temporary-rules, recursive-predicates);

   /* check if the result is already obtained */
   if recursive-predicates =[] or goal is the only element of it
    then do;
     extract "head" and "edb-body" of pseudo-clauses in Temporary-
     rules and Transformed-rules := [ head :- edb-body];
    end then;
    else do;
   /* reprocess if the query is complex */
     for goal and every element of recursive-predicates R do;
      call BFPE(R, IDB, Temporary-rules, recursive-predicates);
      extract "head" and "edb-body" of pseudo-clauses in Temporary-
      rules and  Transformed-rules := ++[head:-edb-body];
                    /* "++"means add new elements */
```

```
    end for;
   end else;

end HCT;
```

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

```
Procedure BFPE(goal, IDB, Temporary-rules, Subgoals);
   /* Input :      goal, IDB          */
   /* Output:      Temporary-rules    */
   /* Input & Output : Subgoals       */
begin;
  recursive-predicates := Subgoals;

/* expansion of the root */
Find rules in IDB that unify to goal and construct a set of pseudo-
clauses in the form
      {head :- body | edb-body | predecessor | recursive-preds};

/* expansion of nodes other than root */
  repeat until the body of all pseudo-clauses are empty;
    for all pseudo-clauses do;
      if the leftmost predicate of the body cannot be expanded,
        then  remove it from the body, and add to edb-body;
      if the leftmost predicate is found to be recursive,
        then add it to Subgoals;
    end for;

 /* expand nodes */
    for every pair of T in the set of pseudo-clauses and
                      R in IDB do;
       if the leftmost predicate of the body in T unifies head of R
       then construct new pseudo-clause and
            add it to Temporary-rules;
    end for;
  end repeat;
end BFPE;
```

An example program of this algorithm written in GHC is shown in the Appendix.

Because the basic algorithm for Horn clause transformation stops 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 new program transformed by HCT is equivalent to the original query.

Here, we will omit the detailed argument on the generalized theoretical analysis for example the treatment of mutual recursive predicates [Miyazaki 86].

Next, a simple example is shown.

Now, `:- ancestor(taro,Y)` is a query from user program, and the following rules are in IDB.

```
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 indicating that the corresponding facts are stored in EDB. The program transformed by HCT is as follows.

```
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).
```

And, the following relational commands can be generated from this result represented by only edb and recursive predicates.

$$
\begin{aligned}
\text{ancestor}_2 = \\
&\Pi_2((\ (\sigma_{1=\text{taro}}(\text{father})) \\
&\quad \cup (\sigma_{1=\text{taro}}(\text{mother})) \\
&\quad \cup \Pi_{1,4}(\sigma_{1=\text{taro}}(\text{father}_{2=1} \ |><| \ \text{ancestor})) \\
&\quad \cup \Pi_{1,4}(\sigma_{1=\text{taro}}(\text{mother}_{2=1} \ |><| \ \text{ancestor}))\ ))
\end{aligned}
$$

Here, 'σ' is selection, '|><|' is join, 'Π' is projection, and '$\cup$' is union, respectively. Execution of recursive commands should terminate when the least-fixed-point appears. In other words, for every command all solutions are always searched when execution terminates.

5. Relational Database Component

The relational database component has three sections as shown Figure 2.

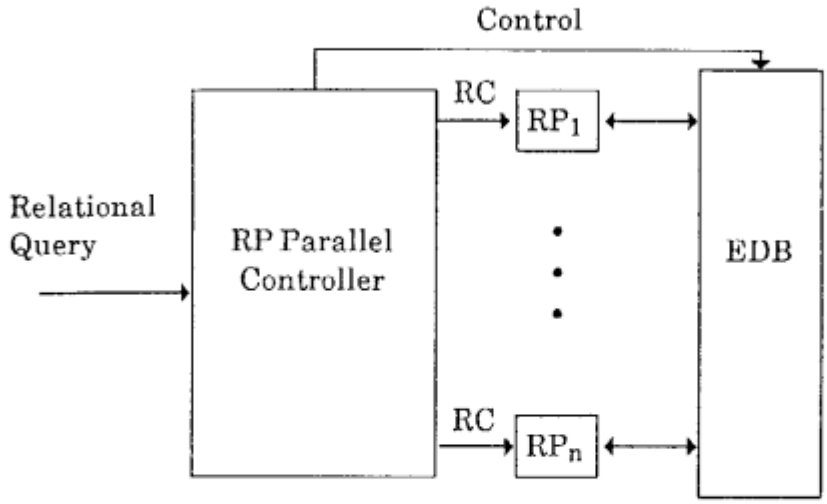

Figure 2 Relational Database Component Configuration

(a) a number of retrieval processors (RP); While the sets of data stream into the RPs, RPs perform the relational operations such as sort, join and selection on them.

(b) query analyzer and RP parallel controller,

(c) EDB storage.

RP parallel controller receives a relational retrieve command from the deductive processing section, analyzes it, and produces a strategy for efficient retrieval from EDB.

The parallel control strategy is developed from the following parameters [Itoh 87].

(a) Number of available RPs when the retrieve command is received from the deductive processing section,

(b) Type of commands (ex. external-sort, internal-sort, join and projection),

(c) Size of the data to be searched and handled, and

(d) Amount of current available working memory.

The following program shows how the RP parallel control method can be written in GHC. It is clear that the relation between this program efficiency and the number of RPs is liner. All annotations are in *italics*.

```
/* "schedule" process "InQuery" with N processors of retrieve. "RPscheduler" checks
status for all RPs, and assigns job to free RPs.  "in" convert the query to the retrieve
command.*/
schedule(N,InQuery) :-  true | in(InQuery,Command),
      'RPscheduler'(Command,StreamStream),
      generate_distribute(N,StreamStream).


'RPscheduler'([C|T],StreamSt) :- true |
      inspect(StreamSt,NewStreamSt,Ans),
      checking_state(Ans,Div,List),
      divide(Div,List,[C|T],NewStreamSt).
'RPscheduler'([]   ,StreamSt) :- true | closeStream(StreamSt).

/* "closeStream" informs every RPs that end of command. */
closeStream([stream(N,A)|Rest]) :- true | A=[term],
       closeStream(Rest).
closeStream([]                ) :- true | true.

/* Predicate "generate_distribute" creates streams for every RPs. */
generate_distribute(0,StSt) :- true | StSt=[].
generate_distribute(N,StSt) :- N=\=0| StSt=[stream(N,Stream)|Rest],
      'RP'(N,Stream), M := N-1, generate_distribute(M,Rest).

/* "inspect" inquire at every RPs for its status (busy or free). */
inspect([]                  ,New,Res) :- true | New=[], Res=[].
inspect([stream(N,St1)|Rest],New,Res) :- true |
      St1=[ins(State)|St2], Res=[(N,State)|ResR],
      New=[stream(N,St2)|NewR], inspect(Rest,NewR,ResR).

/* "checking_state" count number of free RPs, and search for number of RPs.   */
checking_state([]             , Div, List) :- true | Div=0,List=[].
checking_state([(N,busy)|Rest], Div, List) :- true |
      checking_state(Rest,Div,List).
 checking_state([(N,free)|Rest],NDiv,NList) :- true | NDiv := Div+1,
      NList=[N|List], checking_state(Rest,Div,List).

/* "divide" allocates job for free RPs.  "sendToFreeRP" divides job for each Free RP
through division of data and sends out commands to Free RPs.  The data is divided
into N parts by "splitNpart." */
divide(0,List,Command,StSt) :- true  | 'RPscheduler'(Command,StSt).
divide(N,List,[C|T]  ,StSt) :- N=\=0 |
    sendToFreeRP(N,List,C,StSt,NewStSt), 'RPscheduler'(T,NewStSt).

sendToFreeRP(N,List,Cmd,St,NewSt) :- true | splitNpart(N,Cmd,CL),
      sendingCommand(List,CL,St,NewSt).
```

```
sendingCommand([]   ,[]    ,St        ,NewSt) :- true | St=NewSt.
sendingCommand([N|T],[W|CL],[stream(N,St)|Rest],NewSt) :- true |
        St=[cmd(W)|St2], NewSt=[stream(N,St2)|NN],
        sendingCommand(T,CL,Rest,NN).
sendingCommand([N|T],CL    ,[stream(M,St)|Rest],NewSt) :- M =\= N |
   NewSt=[stream(M,St)|NewR], sendingCommand([N|T],CL,Rest,NewR).

/* predicate 'RP' simulates Retrieve Processor */
'RP'(N,[Exec|Command]) :- true | checkingCommand(N,Exec,Command).

/* 'sendToRP' sends the command to Nth RP and, 'Response' is instantiated when
execution is terminated. */
checkingCommand(N,term  ,Command) :- true | Command=[].
checkingCommand(N,ins(C),Command) :- true | C=free, 'RP'(N,Command).
checkingCommand(N,cmd(C),Command) :- true | sendToRP(N,C,Response),
      response(Response,Command,Next), 'RP'(N,Next).

response(R  ,[ins(C)|Cmd],N) :- true | C=busy, response(R,Cmd,N).
response(end,Command     ,N) :- true | Command=N.
```

6. Conclusion

In this paper we indicated the possibility of developing a deductive database system possessing a set of rules and facts using the parallel logic programming language GHC. Thus, the following advantages are expected from the deductive database system.

(a) Database (data, rules), system control program, database management program and user or application program can be handled with the logic programming in the same manner;

(b) Expansion of its own functions applying meta-programming techniques, for example database/knowledge base management function, knowledge acquisition and knowledge integrity constraint check.

We are now developing a deductive database system on the personal inference machine (PSI). This system can be connected by network physically and by Horn clauses logically with the personal inference machines (PSI) as host machines and the parallel inference machine which is also written in GHC. The parallel inference machine controls sending queries to the common database stored in the deductive database machine. In this system, we will investigate the relationship between degree of parallelism and granularity of data to be processed.

This means these whole systems will be written in GHC and we expect to realize thereby an efficient knowledge processing system.

Acknowledgment

The authors express their appreciation for various discussions with Dr. Furukawa, Dr. Ueda and members of KBMS project at ICOT.

References

[Ueda 85] Ueda,K., "Guarded Horn Clauses", in Proc. of Logic Programming'85, Wada,E. (ed.),Lecture Notes in Computer Science 221, Springer-Verlag, pp.168-179, 1986. Also ICOT Technical Report TR-103, June 1985.

[Miyazaki 86] Miyazaki, N., Yokota, H., and Itoh, H., "Compiling Horn clause queries in deductive databases: A Horn clause transformation approach", ICOT Technical Report TR-183, June 1986.

[Gallaire 84] Gallaire, H., Minker, J., and Nicolas, J.-M., "Logic and Databases: A deductive approach", ACM Computer Surveys, Vol. 16. No. 2, June 1984.

[Yokota 86] Yokota,H., Sakai,K., and Itoh,H., "Deductive Database System Based on Unit Resolution", Proc. of Data Engineering , pp. 228-235, Feb. 1986. Also ICOT Technical Report TR-123, 1986.

[Deyi 84] Deyi,L., "A PROLOG Database System",Research Studies Press Ltd., 1984.

[Itoh 87] Itoh, H., Abe, M., Sakama, C., and Mitomo, Y., "Parallel control techniques for dedicated relational database engines", Proc. of Data Engineering, Feb. 1987. Also ICOT Technical Report, TR-182, June 1986.

Appendix

This appendix gives a horn clause transformation program written in GHC. Annotations are preceded by the '%' symbol , or sandwiched in between '/*' and '*/'.

```
/* ************************** */
/* Horn Clause Transformation */
/* ************************** */

'HCT'(Goal, IDB, TransRule) :- true |
      'BFPE'(Goal, IDB, TempRule,[], Recursive),
      checkingAlreadyObtained(Recursive, Goal, Cont),
      nextHCT(Cont, Goal, IDB, TempRule, Recursive, TransRule).

/* check if the result is already obtained */
checkingAlreadyObtained([]  ,_  ,Result) :- true  | Result=true.
checkingAlreadyObtained([A],Term,Result) :- true  |
```

```
      unifiable(A,Term,Result).
checkingAlreadyObtained([A|B],_,Result) :- B \=[] | Result= false.

/* Reconstruct form "head :- edb-body" when the goal is only recursive predicate in
the query. */
nextHCT(true, Goal,IDB,TempRule,RecursivePrd,TransRule) :- true |
     reconstructRule(TempRule,TransRule-[]).
/* reprocess when the query is complex */
nextHCT(false,Goal,IDB,TempRule,RecursivePrd,TransRule) :- true |
      setUnion([Goal],RecursivePrd,List),
      reProcess(List, IDB, RecursivePrd, TransRule-[]).

reconstructRule([{(Head:-Body);EdbBody;_;_}|Rest],TP1-TPn) :- true |
     TP1=[(Head:- EdbBody)|TP2],
      reconstructRule(Rest, TP2-TPn).
reconstructRule([]                                  ,TP1-TP2) :- true |
      TP1=TP2.

reProcess([]      , _  , _         , TP1-TP2) :- true | TP1=TP2.
reProcess([T|Rest], IDB, Recursive, TP1-TPn) :- true |
     'BFPE'(T,IDB, TempRule, Recursive, IRecursive),
     reconstructRule(TempRule,TP1-TP2),
      reProcess(Rest, IDB, IRecursive,TP2-TPn).

/* ******************************** */
/* Breadth First Partial Expansion */
/* ******************************** */

'BFPE'(Goal, IDB, Temp, InRec, OutRec) :- true |
     expansionOfRoot(IDB,Goal,InRec, Temp1),
     outerLoopProcess(Temp1,IDB,InRec,Temp,OutRec).

/* ***************** */
/* expansion of root */
/* ***************** */

expansionOfRoot([]              ,Goal,Recursive,TRule) :- true |
      TRule=[].
expansionOfRoot([(H :-Body)|Rest],Goal,Recursive,TRule) :- true |
   unifiable(H,Goal,Res),
   expansionOfRoot1(Res,[(H :-Body)|Rest],Goal,Recursive,TRule).

expansionOfRoot1(true ,
                [(H :-Body)|Rest],Goal,Recursive,TRule) :- true |
      TRule=[{(H:- Body) ; []; [Goal]; Recursive}|Temp],
      expansionOfRoot(Rest,Goal,Recursive,Temp).
```

```
expansionOfRoot1(false,[_   |Rest],Goal,Recursive,Temp) :- true |
      expansionOfRoot(Rest,Goal,Recursive,Temp).

/* ********** */
/* Outer Loop */
/* ********** */

outerLoopProcess(Temp,IDB,Recursive,OutTemp,OutRec) :- true |
       innerLoopProcess(Temp,Recursive, TempList,NewRec),
       expandNodes(TempList,IDB, NewTemp-[]),
       extractEmptyBodyRule(TempList,Q1-NewTemp),
  /* repeat to process "outerLoopProcess" until the bodies of all rules are empty. */
       checkingAllEmptyRule(Q1,Result),
       checkingOuterFinal(Result,Q1,IDB,NewRec,OutTemp,OutRec).

checkingOuterFinal(true ,T ,IDB,REC    ,OT      ,OR     ) :- true |
        REC=OR, T=OT.
checkingOuterFinal(false,Q1,IDB,Newrec,OutTemp,OutRec) :- true |
       outerProcess(Q1,IDB,Newrec,OutTemp,OutRec).

%%%%%%%%%%%%%%
/* Inner Loop */
%%%%%%%%%%%%%%

/* repeat to process "innerLoopProcess" until no change occur in Temp. */
innerLoopProcess(Temp,Recursive,NewTemp,OutR) :- true |
      changeBodyAllTempRule(Temp,Recursive,IntRule,NR,Q1-[]),
      checkingNoChangeBody(Q1,Result),
      checkingInnerLoopFinal(Result,IntRule,NR,OutR,NewTemp).

checkingNoChangeBody([]          , Result) :- true | Result=true.
checkingNoChangeBody([false|Rest], Result) :- true |
            checkingNoChangeBody(Rest,Result).
checkingNoChangeBody([True |Rest], Result) :- True \= false |
            Result=false.

checkingInnerLoopFinal(true ,IntRule,Rec,NR,NewTemp) :- true |
      IntRule=NewTemp, Rec=NR.
checkingInnerLoopFinal(false,IntRule,Rec,NR,NewTemp) :- true |
      innerLoopProcess(IntRule,Rec,NewTemp,NR).

/* Expansion of node other than root */
changeBodyAllTempRule([T|Temp],RecurList,TT,NR,Q1-Qn) :-
       T={(TH:-[Lmpb|_]):_;Prede;_} |
       Q1=[Result|Q2], TT=[NewT|NewTemp],
       checkingEDBBody(Lmpb, Prede, RecurList, Result),
       execEdbBody(Result,T,NewT,AddRecur),
```

```
      append(AddRecur,RecurList,RecL),
      changeBodyAllTempRule(Temp,RecL,NewTemp,NR,Q2-Qn).
changeBodyAllTempRule([T|Temp],RecurList,TT,NR,Q1-Qn) :-
      T={(TH:-[]):_;Prede;_} | TT=[T|NewT],
      changeBodyAllTempRule(Temp,RecurList,NewT,NR,Q1-Qn).
changeBodyAllTempRule([]      , R    , T     ,NR,Q1-Qn) :- true |
      T=[], Q1=Qn, R=NR.
```

/ If the leftmost predicate of the body cannot be expanded, then remove it from the body, and add to edb-body. */*

```
execEdbBody(true, {(Head:-[Lmpb|Body]);Edb;Predecessor;Recursive},
            NewT,A) :- true |
     A=[], append(Edb,[Lmpb],NewEdb),
     NewT={(Head :- Body); NewEdb; Predecessor; Recursive}.
```

/ If the leftmost predicate is found to be recursive, then add it to the list of recursive predicates. */*

```
execEdbBody(rec ,{(Head:-[Lmpb|Body]);Edb;Predecessor;Recursive},
            NewT,A) :- true |
   A=[Lmpb], append(Edb,A,NewEdb),
   NewRecursive=[Lmpb|Recursive],
   NewT={(Head :- Body);NewEdb;Predecessor;NewRecursive}.
execEdbBody(false,T,NewT,A) :- true | T=NewT, A=[].
```

/ the leftmost predicate is comparison or extension, or the leftmost predicate can unify an element of the list of recursive predicates. */*

```
checkingEDBBody(Lmpb,Prede,RecursiveList,Result) :- true |
       isComparison(Lmpb,TF1,H),
       isExtension(Lmpb,TF2,H),
       isRecursivePredicate(Lmpb,RecursiveList,TF3,H),
       isPredecessor(Lmpb,Prede,TF4,H),
       edbBodyChecker(TF1,TF2,TF3,TF4,H,Result).

edbBodyChecker(true ,_  ,_  ,_  ,H,R) :- true | R=true, H=halt.
edbBodyChecker(_  ,true ,_  ,_  ,H,R) :- true | R=true, H=halt.
edbBodyChecker(_  ,_  ,true ,_  ,H,R) :- true | R=true, H=halt.
edbBodyChecker(_  ,_  ,_  ,true ,H,R) :- true | R=rec,  H=halt.
edbBodyChecker(false,false,false,false,H,R) :- true | R=false.

%%%%%%%%%%%%%%%%%%
/* Expand nodes */
%%%%%%%%%%%%%%%%%%

expandNodes([T|TempRules],IDB,Q1-Qn) :- true |
      oneExpandNodes(IDB,T,Q1-Q2),
      expandNodes(TempRules,IDB,Q2-Qn).
expandNodes([]           ,IDB,Q2-Qn) :- true | Q2=Qn.
```

```
oneExpandNodes([]      ,T,Q1-Q2) :- true | Q1=Q2.
oneExpandNodes([R|Rest],T,Q1-Qn) :- true |
        R=(RH :- _), T={(_ :- Body);_;_;_},
       extractLmpb(Body,Lmpb),
       unifiable(Lmpb,RH,Result),copyStructure(R,RR),
       oneExpandNode1(Result, T,RR,Q1-Q2),
       oneExpandNodes(Rest,T,Q2-Qn).

extractLmpb([A|_],L) :- true | A=L.
extractLmpb([]   ,L) :- true | L='$failfail$'.

oneExpandNode1(false,T,R,Q1-Q2) :- true | Q1=Q2.
oneExpandNode1(true ,{(Head :- [Lmpb|Body]); Edb; Prede; Recur},
               (HR :- BR),Q1-Q2) :- true |
       Q1=[N|Q2], Lmpb=HR,
       append(BR,Body,NB),
       append(Prede,[Lmpb],NewP),
       N = {(Head :- NB); Edb; NewP;Recur}.

/* 'extractEmptyBodyRule' extracts rules of empty bodies. */
extractEmptyBodyRule([]   ,Q1-Qn) :- true | Q1=Qn.
extractEmptyBodyRule([T|R],Q1-Qn) :- T={(H :- []);_;_;_} |
       Q1=[T|Q2], extractEmptyBodyRule(R,Q2-Qn).
extractEmptyBodyRule([T|R],Q1-Qn) :- T={(H :- B);_;_;_}, B\=[] |
       extractEmptyBodyRule(R,Q1-Qn).

checkingAllEmptyRule([]          ,Result) :- true | Result=true.
checkingAllEmptyRule([{(H:-[])  ;_;_;_}|Rest],Result) :- true |
     checkingAllEmptyRule(Rest,Result).
checkingAllEmptyRule([{(H:-Body);_;_;_}|Rest],Result) :- Body\=[] |
      Result=false.
```