Intelligent backtracking and sidetracking in Horn clause programs - implementation

Luis Moniz Pereira

António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisbon, Portugal

December 1979

report no. 13/79 CIUNL

Abstract

This is the second report of our work on intelligent backtracking and sidetracking strategies in Horn clause programs.

This second part is a description of three interpreters (written in Prolog) for Prolog programs, working on the basis of the theory described in the first part [Pereira et al. 1979]. They offer practical information about the features of the new strategies.

The first is a general-purpose interpreter which uses intelligent backtracking instead of the standard blind one.

The second is a specialization of the first, for database query only, which uses intelligent backtracking in a much more restricted way, although sufficient for Prolog relational databases satisfying a reasonable set of assumptions.

The third is an interpreter working through sidetracking, which by its very nature is not extensive to the full language without some redefinitions. The use of cut, notably, does not make sense without a fixed order of execution of goals. In a future paper, however, some control constructs will be presented to achieve an effect similar to the cut.

Intellisent backtracking and sidetracking in Horn clause programs - implementation

Luis Moniz Pereira

António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisbon, Portugal

December 1979

report no. 13/79 CIUNL

1. Introduction

The three interpreters presented in this report are written in Prolog itself and are not aimed at efficiency. Two of them, for example, achieve intelligent backtracking by using the standard backtracking provided by the compiler. So they must be viewed as working simulations at a high level of what must have an implementation of its own to achieve competitive efficiency. We hope this will be done in the near future. The database interpreter, nevertheless, is already competitive as it stands, even for smallish databases. We believe it can be of immediate use for Prolog programs which consult a practical relational database.

Since the intelligent backtracking interpreter is quite a complex program, even for Prolog knowledgeable persons, it may be useful to start by studying its simpler specialization to databases. The sidetracking interpreter can be understood on its own.

The best way to understand these interpreters is to complement the theory by running them on examples with tracing. But the paper on the theory should be read first.

2. The intelligent backtracking interpreter

We assume the reader familiar both with the theory of intelligent backtracking, which is exposed in detail in the first part of this report [Pereira et al. 1979], and with the programming language Prolog. An abridged version of the first report is available [Pereira et al. 1980].

2.1 Representation

The representation of variables within the interpreter for the purpose of carrying dependency information is as follows:

Every variable X is represented in the form

VX - TX

VX is the value of the variable.

TX is the tag-list of X, and is of the form

[IX | UX]

IX, the instantiation variable of X, acquires the number of the goal in which X is instantiated if such instantiation is direct, i.e. if X is unified with a textual non-variable term in a clause head. If X is indirectly instantiated by unification with another variable, IX acquires an asterisk.

UX is a (possibly empty) list of any dependencies of X created by unification with other variables . Every such dependency within UX is of the form

[N ! TY1]

where N is the number of the goal in which X was bound to a certain variable Y = VY - TY , and TY1 is obtained from TY by excluding its element [N ! TX1] , which expresses the dependency of Y on X, thus preventing circularity of reference. TX1 is in turn obtained from TX by excluding [N ! TY1] from it.

This process of dependency creation is best seen with an example:

Suppose we have the goals

1 2 3 4

p(X , Y) , p(W , Z) , p(Z , Y) , q(X)

and the clauses

p(V , V)

q(a)

Initial state of the variables

X = VX - [IX ! UX]

Y = VY - [IY ! UY]

Z = VZ - [IZ ! UZ]

W = VW - [IW ! UW]

After goal 1

X = V1 - [IX, [1, IY ! UY1] ! UX1]

Y = V1 - [IY, [1, IX ! UX1] ! UY1]

```
  = VZ - [ IZ ! UZ ]
  = VW - [ IW ! UW ]
```

After goal 2

```
  = V1 - [ IX, [ 1, IY ! UY1 ] ! UX1 ]
  = V1 - [ IY, [ 1, IX ! UX1 ] ! UY1 ]
  = V2 - [ IZ, [ 2, IW ! UW1 ] ! UZ1 ]
  = V2 - [ IW, [ 2, IZ ! UZ1 ] ! UW1 ]
```

After goal 3

```
  = V - [ IX, [ 1, IY, [ 3, IZ, [ 2, IW ! UW1 ] ! UZ2 ] ! UY2 ] ! UX1 ]
  = V - [ IY, [ 1, IX ! UX1 ], [ 3, IZ, [ 2, IW ! UW1 ] ! UZ2 ] ! UY2 ]
  = V - [ IZ, [ 2, IW ! UW1 ], [ 3, IY, [ 1, IX ! UX1 ] ! UY2 ] ! UZ2 ]
  = V - [ IW, [ 2, IZ, [ 3, IY, [ 1, IX ! UX1 ] ! UY2 ] ! UZ2 ] ! UW1 ]
```

After goal 4

```
  = a - [ 4, [ 1, *, [ 3, *, [ 2, * ] ] ] ]
  = a - [ *, [ 1, 4 ], [ 3, *, [ 2, * ] ] ]
  = a - [ *, [ 2, * ], [ 3, *, [ 1, 4 ] ] ]
  = a - [ *, [ 2, *, [ 3, *, [ 1, 4 ] ] ] ]
```

It is easy to obtain from the tag-lists the chains of goals through which each variable acquired its value:

```
  -> 4
  -> 1 - 4
```

Z -> 3 - 1 - 4

W -> 2 - 3 - 1 - 4

For each variable, these are the backtrack goals for a failure to unify caused by its value.

To be able to cope with this representation the original clauses of a program must be changed accordingly, and so before execution the program is read from a File and modified. This is accomplished with the directive 'program_in(File)'. The clauses are converted to an internal representation as follows :

Every clause head H is converted to

H' - [C , E]

Every constant symbol (atom, integer or functor) k appearing in H will have the form k - T in H', so that a different variable T is associated with every constant symbol, even if the symbols are equal.

C is a list containing every such variable T.

Various ocurrences of the same variable in H will have different variables standing for them in H' , because variables which match equal values do not necessarily have matching tag-lists. Every group of different variables in H' standing for the same variable in H is put on a list, so that their values may be compared irrespective of their tag-lists. E is the list of all such lists.

As an example, the clause head

p(a,X,f(a,X),f(b,Y),g(X,Y),Z)

will be converted to

p(a-A1,X1,f(a-A2,X2)-F1,f(b-B,Y1)-F2,g(X3,Y2)-G,Z)-
-[[A1,F1,A2,F2,B,G],[[X1,X2,X3],[Y1,Y2]]]

To convert the body of a clause we simply replace, in each single goal, every constant symbol k by k - [0] . This provides a way of distinguishing textual constant symbols of a goal from constant symbols acquired by variables at clause heads, while maintaining the same type of representation, because the latter acquire the number of the goal that produced the instantiation, and goals are always numbered from 1 onwards.

As an example, the goal

p(X,f(a,g(b,X)))

will be converted to

p(X,f(a-[0],g(b-[0],X)-[0])-[0])

2.2 Operation

After reading and pre-processing the program with 'program_in' (an ':-end.' is required after the last clause), any goal G can be executed with the directive 'goal(G)' . G is then converted like the body of a clause, and interpretation begins with a call to 'ib' , the main predicate of the intelligent backtracking interpreter. Upon successful

interpretation G is converted back to its standard original form.

Every call to 'ib' has the form

ib(G,N1,N2,PN,C)

where G is the goal expression to be interpreted, N1 is the number of the first single goal to be solved within G, N2 is the number of the next single goal to be solved after solving G, PN is the number of the parent goal of G, and C is set equal to 'cut' if G contains a cut along its successful execution path.

2.2.1 Single goals

Execution of a single goal G is carried out along the following steps (leaving some details aside) :

1) Find a clause G - [C , E] :- B .

2) For each free variable that matched a textual constant in the head of the clause assign the current goal number to its instantiation variable, and close its tag-list, assigning asterisks to all free instantiation variables appearing inside it. This is the effect of 'number(C,N1)' . If the variable is not free, but matches the constant, 'number' ignores it.

3) For each list within E make all their elements have the same value, and update their tag-lists accordingly ('set_equal(E,N1)'). If this proves to be impossible return to 1) and try to get another clause for G.

4) Increment the number of the current goal.

5) Call 'ib' to execute the body B.

If B was successfully executed, G is solved ; otherwise:

6) Call 'no_bk_here(N1)' (N1 is the number of goal G) to see if G was not recorded as a backtrack goal, and if so fail the call of 'ib' for G (this is where intelligent backtracking is informing that it is no use looking for alternative solutions of G) ; if G is a backtrack goal, alternatives must be tried, so go back to 1).

When no more clauses for G are available (it can happen at the first try) it is checked whether the predicate of G is an external one (any predicate for which no clauses were read by 'program_in'). If so, the standard interpreter is called to execute G ; if not, a conflict analysis takes place with a call to 'set_bkgoals' , which makes a call to 'conflicts' for every clause for G, to produce the backtrack goals for this failure; in the end, the parent goal PN is also registered as a backtrack goal.

2.2.2 Conjunctions and disjunctions

Having successfully executed goal G1 in a conjunction (G1,G), and having failed the execution of G, a call is made to 'no_bk_until(N1)', where N1 is the number of the first single goal executed within G1, to check if there are no backtrack goals back until N1 (including it); if so, and if no cut was passed along the successful execution path of G1, backtracking over the whole subtree of G1 is readily accomplished.

If there are no backtrack goals back until N1 but a cut exists in the solution path of G1, backtracking goes to its parent reporting that it in turn must fail because of the cut. This process is elucidated in the next

section.

No special treatment is required for disjunctions.

2.2.3 The cut

Special treatment is reserved for the cut symbol, two clauses being provided to handle the call 'ib(!,N1,N2,PN,C)'.

The first one succeeds with N1=N2 and C=cut. This instantiation of C will let the interpreter know, at any subsequent point in the conjunction containing the cut, that it has been passed along the conjunction, thus modifying the outcome upon a successful 'no_bk_until'.

If backtracking arrives at a cut the second clause will be activated, and a simulation of a cut failure will ensue by succeeding this clause, but with N2=cut. This value of N2 is trapped by the interpreter, that repeatedly reports success, always with N2=cut, until the parent of the cut is reached, and then failed, thus avoiding possible alternative clauses for it, as prescribed by the failed cut.

2.2.4 The obtention of backtracking information

Upon failure of a single goal a conflict analysis takes place to obtain backtracking information ('set_bkgoals').

'set_bkgoals' obtains, for each clause that failed to match the current goal, a list of the backtrack goals susceptible of producing changes in the goal variables leading to a match with the clause, using predicate 'conflicts'. Since the clauses form an OR node, the individual

lists are merged into a single one, that therefore contains all the backtrack goals available to hopefully solve the goal. In the end 'set_bkgoals' adds to the list the parent of the failed goal, which is always a backtrack goal.

For each clause all conflicts must be solved for the matching to be possible, so, according to the AND rule, the resulting list of backtrack goals is the least among the lists for individual conflicts (corresponding to least recent goals).

Two main types of conflicts are analysed:

c_conflicts :-

These are originated when a currently non-variable term in the goal tries to match a textual non-variable term in the clause head having a different principal functor.

It is a clash between two constant symbols, only one of which can change - the one in the goal, if not also textual.

The list of backtrack goals BG is obtained by calling 'depends(T,BG)', which searches the tag-list T of the goal term for the dependencies that led to its instantiation.

Each list thus obtained is compared by predicate 'least' with the previous one (initially empty), and only the least is retained.

e_conflicts :-

These occur when two different constant symbols appear in positions in the goal corresponding to positions in the clause head that are required to hold the same value.

Since this conflict can be solved if either of the constant symbols changes its value to match the other, after the two dependency lists are obtained (inside 'f_conflict') they are merged into a single one, which is compared with the existing one for this clause, to keep just the least (corresponding to the AND of all conflicts analysed so far).

If at any stage an irrevocable conflict arises (BG=[]) no further analysis takes place, and there are no backtrack goals for this clause.

Failure of a goal caused by backtracking on a cut within the body of the activated clause for the goal is a very special case, since no amount of analysis is guaranteed to deliver all backtracking information. In fact, a possible solution to the problem might exist if some goal variable, that matched a constant in the head of the clause containing the cut, should be instantiated to a non-matching constant, therefore avoiding activation of that clause and maybe permitting activation of another one; but the variables are not carrying information of all the goals where such instantiation might be achieved. Besides not being able to get all backtracking information, trying to get the possible information proves to be too complicated. In face of this, the interpreter just registers as backtrack goals all the dependencies of non-variable terms in the goal ('register_all_bkgoals').

2.2.5 Alternative solutions

If, upon successful interpretation of a goal, we want to produce alternative solutions, we have to backtrack into the interpreter again. However, the intelligent backtracking mechanism is viewing the process of backtracking as being caused by a failure of a goal, and not as a general tool for exploring the search space of the problem.

Actually, we can view the search for alternative solutions as a user-generated failure of any previous solutions. What the user wants is, in fact, to try to modify the arguments of the previous solution. Now, the goals where the arguments may be modified are precisely all the arguments' correcting goals.

Thus, after forgetting any remaining backtrack goals in the global list ('reset_bkgoals'), all the arguments' correcting goals are put in the list ('register_all_bkgoals'), and backtracking is re-instated.

3. Description of an interpreter specialized to databases

The general interpreter previously described becomes a lot simpler, and even competitive as it stands, when specialized to relational databases with unit and possibly non-unit clauses. This specialized interpreter may be called from the standard one at any number of places in the program. The database may be compiled or interpreted.

The interpreter is presented in Appendix 2.

3.1 Specialization assumptions

Some (frequently met) assumptions regarding the database are necessary to keep the interpreter simple.

1) There are no cuts in the database or query.

2) Non-unit clauses contain only variables as arguments (this could be relaxed to allow ground terms, with little extra complication).

3) Unit clauses and the goal contain only variables or ground terms as arguments.

4) There are no multiple occurrences of variables in the head of any clause, unless all identical occurrences will match a ground term.

5) Unit clauses are assumed to come before non-unit clauses for the same predicate; in case they do not, an extra clause must be introduced, e.g. :-

```
h(X):- b(X).                    h(X):- b(X).
h(a).            becomes        h(X):- h'(X).
                                h'(a).
```

6) Predicate 'fail' may only be used to backtrack into the database with the intention of getting alternative solutions, not with the intention of exploring the whole space (e.g. for certain types of side-effects).

3.2 Database access preparation

Our assumptions (2, 3, and 4, specifically) guarantee that each variable will, possibly, be tagged only with the number of the node where it becomes unified with a ground term as a whole.

Because, in general, arguments will be tagged, a way must be provided for goals to access the database with non-tagged arguments :-

1) For each database predicate p with n arguments for which there are unit-clauses, the following clause must be added

```
unit(p(A1-N1,...,An-Nn)-[N1,...,Nn], p(A1,...,An)).
```

which allows expeditious translation of one form into the other.

2) A similar clause must also be added for each predicate external to the database but called from within it (e.g. system predicates). These external predicates must not perform any binding of the database variables (though they may test conditions and produce side-effects), and must not set up backtrack points.

3) To allow both for interpretation and compilation of non-unit clauses, these must be written as single arguments of binary predicate 'non_unit'. A non-unit database clause of the form

H :- G1 , G2 .

must be programmed as

non_unit(H , (G1 , G2)) .

4) Each top goal (query) fed to the interpreter is automatically converted (by 'conv_in') so that each of its ground terms is tagged initially with a variable. Original variables in the goal are left as they stand since we want all the occurrences of a variable to acquire the same value and tag. Actually, it is not the original goal that is converted but a new copy of it, since we do not want the goal variables to be tagged. Thus, upon execution of the converted copy of the goal, there occurs a reverse conversion of the copy into non-tagged form (by 'conv_out'). This reverse form is unified with the original goal to produce the answer to the query.

3.3 Tagging

Tagging of variable bindings is accomplished in a way similar to the general interpreter's, with a call to 'number'. 'number' takes the list of tag variables in the head of a (unit) clause and numbers with the current goal number those not already numbered. Upon backtracking the numbering is, of course, undone.

3.4 Obtention of backtrack goals on failure

This differs from the general interpreter's method. When a goal fails, there is no analysis to determine which specific arguments have caused failure. Such an analysis would be too costly if carried out for each unit clause of a database predicate. Furthermore, as a result of such an analysis, it is likely, although not certain, that each bound argument in the goal would be responsible for the most ancient backtrack goal for some unit clause. Thus, each goal tagging a bound argument would be a backtrack goal.

Consequently, for economy, we refrain from conflict analysis, and always take as backtrack goals all goals in the tagging of the arguments of the failed goal. The price to pay for this simplification may be extra unnecessary backtracking on ocasion. However, this is highly compensated by the economy of foregoing conflict analysis over all the unit clauses for each failed goal.

3.5 Generation of alternative solutions

This database interpreter may be called from a Prolog program with the purpose of answering a database query expressed as a conjunction and/or disjunction of goals.

After a solution is produced for a query by the database interpreter, there may occur subsequent backtracking from the calling program into the interpreter for the obtention of alternative solutions.

This is carried out in a similar way to the one used by the general intelligent backtracking interpreter (refer to section 2.2.5).

3.6 Using the interpreter

The interpreter, because it is written in Prolog, is called like any other procedure after it has been loaded with the user's host program. The call

```
call_dbib( G )
```

where G is a conjunction and/or disjunction of goals may be inserted anywhere in a Prolog host program.

On first entry in the interpreter, goals are numbered from 1 onwards. If another call is made further on in the host program, goal numbering starts with the number after that of the most recent goal in the list of backtrack goals.

In Appendix 3 we present a database example, with the additional module containing the extra 'unit' and 'non_unit' clauses needed by the interpreter.

4. The sidetracking interpreter

This simple interpreter internally represents the conjunction of goals to be solved as G - X , G being a list of single goals whose tail is the free variable X, so that appending a similar list G1 - X1 can be done just by unifying it with X, the new list being now G - X1. Thus a practical implementation of a circular list is achieved.

Each program P that we want to be used by the interpreter has to be loaded first with the directive 'load(P)', its clauses being then converted so as to change their bodies to the list form above described. Clause bodies become thus ready to be appended to the main list of goals. An ':-end.' is required after the last clause.

A goal G is interpreted by calling 'si(G)'. 'si' builds the goal list from G, and calls the main predicate of the interpreter, 'sidetrack'.

'sidetrack' delivers the goal list to 'solve_det', to get all possible deterministic steps carried out first, then, if goals remain, performs a non-deterministic step with a call to 'step', and calls itself again with the remaining list, if nonempty. The execution ends when the list of goals to be solved becomes empty.

During backtracking, 'solve_det' is automatically skipped.

'solve_det' continually scans the list of goals, with 'scan', replacing every successful deterministic one by the body of the only clause it activates, until either a goal fails (failure is spotted because a failing goal is deterministic), all goals are solved (the list is empty), or the whole list has been searched and no deterministic goal

was found. In the call 'scan(G,OG,SG-XS,D)' G is the list of goals to be scanned, that gets splitted in OG, the old goals not yet scanned, and SG-XS, the scanned goals (including in the end the new goals from the replacement of a deterministic goal, if one was found; D gets instantiated if that is the case).

'scan' checks the condition of determinism of a goal G by calling 'det(G,NG)'. 'det' succeeds if G is deterministic, and then NG is either the body of the clause activated by G, if there is one, or still a free variable, if G fails.

The first clause for 'det' calls 'ext_det', to check if there are external definitions for the determinism of G; this provides the user a way to define determinism conditions of certain goals that may speed up considerably the execution of the interpreter.

The second clause is provided for goals of the type C : G , which are not standard in Prolog, but described in our report on the theory of sidetracking: G is the goal, but it may only be activated if condition C is true. Since sidetracking destroys the standard order of execution of goals, this type of control becomes necessary, for example with goals involving arithmetic expressions, that must be checked to see if all variables within them have already acquired integer values.

The third clause simply considers any disjunction as a non-deterministic goal.

The fourth clause is the general one, that tries to get two different clauses matching G, so deciding if it is deterministic or not. If no clause is found execution of G is tried by calling the standard

interpreter, for G may be a system predicate.

'step' calls 'replace' to replace the current goal. It may be replaced by the body of a clause, if it is a single goal, or, if it is a disjunction, by one of the disjuncts. If the goal is of the type C : G it cannot be replaced, so 'step' calls itself with the remaining list of goals. If all goals in the list happen to be of type C : G the interpreter is facing a deadlock situation, so failure occurs.

5. References

[Pereira et al. 1979] Pereira,L.M. ; Porto,A.

Intelligent backtracking and sidetracking
in Horn clause programs - the theory
Departamento de Informatica
Universidade Nova de Lisboa, Lisbon.

[Pereira et al. 1980] Pereira,L.M. ; Porto, A.

Intelligent backtracking and sidetracking for logic programs
Submitted to the 5th Conference on Automated Deduction,
July 8-11, 1980, Les Arcs, Savoie, France, organized by
Institut de Recherche d'Informatique et d'Automatique,
Rocquencourt, France.

Appendix 1. The intelligent backtracking interpreter

```
:- 'LC'.

:- program([],[program_in(1),goal(1)]).

:- mode ib(+,-,-,+,-).
:- mode no_bk_until(+).
:- mode no_bk_here(+).
:- mode reset_bkgoals(+).
:- mode number(+,+).
:- mode close(+,+).
:- mode set_equal(+,+).
:- mode equal(?,?,+).
:- mode var_var(-,?,-,?,+).
:- mode var_term(-,-,+,+,+).
:- mode eq_args(+,+,+).
:- mode make(-,-,?,-).
:- mode external(+,+,+).
:- mode ext_eq_args(+,+,+).
:- mode ext_eq(?,?,+).
:- mode register_all_bkgoals(+).
:- mode all_bkgoals(+).
:- mode set_bkgoals(+,+,+).
:- mode conflicts(+,+,+).
:- mode c_conflicts(?,?,+,-).
:- mode depends(+,-).
:- mode search(+,-).
```

```
:- mode inst(+,-).

:- mode sort(+,-).

:- mode bubble(+,+).

:- mode least(+,+,-).

:- mode e_conflicts(+,+,-).

:- mode e_conflict(+,+,-).

:- mode decompose(?,-).

:- mode first(+,-,-).

:- mode f_conflicts(+,+,+,-).

:- mode f_conflict(?,?,+,-).

:- mode a_conflicts(+,+,-).

:- mode merge(+,+,-).

:- mode register_bkgoal(+).

:- mode register_bkgoals(+).

:- mode program_in(+).

:- mode convert(+).

:- mode goal(+).

:- mode new_head(+,-).

:- mode new_head_args(+,-,+,+).

:- mode new_head_term(?,-,+,+).

:- mode equal_var(-,+,-,-).

:- mode remove_singles(+,-).

:- mode new_body(+,-,+).

:- mode new_body_term(?,-,+).

:- mode new_body_args(+,-,+).

:- mode old_body(-,+).

:- mode old_body_term(-,+).

:- mode old_body_args(-,+).
```

```
:- mode var_list(+,-).

:- mode copy(+,-).


ib((G1,G),N1,N3,PN,C):- !,ib(G1,N1,N2,PN,C),
                         ( N2=cut,N3=cut ;
                           ( ib(G,N2,N3,PN,C) ;
                             no_bk_until(N1),( C=cut,N3=cut ;
                                               !,fail ) ) ).

ib((G1;G),N1,N2,PN,C):- !,( ib(G1,N1,N2,PN,C) ; ib(G,N1,N2,PN,C) ) .

ib(!,N,N,_,cut).

ib(!,_,cut,_,_).

ib(G,N1,N3,PN,_):- clause(G-[C,E],B),
                   number(C,N1),set_equal(E,N1),
                   N2 is N1+1,
                   ( ib(B,N2,N3,N1,_) ;
                     no_bk_here(N1),!,fail ),
                   ( N3\==cut ;
                     register_all_bkgoals(G),
                     register_bkgoal(PN),!,fail ).

ib(G,N1,N2,PN,C):- call(pred(G,H)),set_bkgoals(G,H,PN),!,fail ;
                   external(G,N1,PN),N2 is N1+1.


no_bk_until(N):- call(bkgoal(X)),X>=N,!,fail.

no_bk_until(_).


no_bk_here(N):- call(bkgoal(N)),
                reset_bkgoals(N),!,fail.

no_bk_here(_).


reset_bkgoals(N):- call(bkgoal(X)),X>=N,retract(bkgoal(X)),fail.

reset_bkgoals(_).


number([],_):-!.

number([[I|U]|T],N):- ( I=N,close(U,[]) ; true ),number(X,N),!.
```

```
close(U,X):- var(U),U=X.

close([[_,*|U1]|U],X):- close(U1,[]),close(U,X).


set_equal([],_):-!.

set_equal([E1|E],N):- all_equal(E1,N),set_equal(E,N),!.


all_equal([X1,X2],N):- equal(X1,X2,N).

all_equal([X1,X2|X],N):- equal(X1,X2,N),all_equal([X2|X],N).


equal(V1-T1,V2-T2,N):- V1==V2 ;
                       var(V1),( var(V2),var_var(V1,T1,V2,T2,N) ;
                                 var_term(V1,T1,V2,T2,N) ) ;
                       var(V2),var_term(V2,T2,V1,T1,N) ;
                       atomic(V1),!,V1=V2 ;
                       atomic(V2),!,fail ;
                       V1=..[F|A1],V2=..[F|A2],eq_args(A1,A2,N).


var_var(V,[I1|U1],V,[I2|U2],N):- make(NU1,NX1,U1,X1),
                                 make(NU2,NX2,U2,X2),
                                 X1=[[N,I2|NU2]|NX1],
                                 X2=[[N,I1|NU1]|NX2].


var_term(V1,[*|U1],V2,T2,N):- close(U1,[[N|T2]]),
                              ( atomic(V2),V1=V2 ;
                                V2=..[F|A2],
                                eq_args(A1,A2,N),
                                V1=..[F|A1] ).


eq_args([A1|A],[B1|B],N):- equal(A1,B1,N),eq_args(A,B,N).

eq_args([],[],_).


make(V1,V1,V,V):- var(V).

make([X|U1],V1,[X|U],V):-make(U1,V1,U,V).
```

```
external(G,N,PN):- atom(G),( call(G),( true ;
                                       no_bk_here(N),!,fail ) ;
                             register_bkgoal(PN),!,fail ).

external(G,N,PN):- old_body(OG,G),copy(OG,CG),
                   ( call(CG),new_body(CG,NG,N),
                     G=..[_|A],NG=..[_|NA],ext_eq_args(A,NA,N),
                     ( true ;
                       no_bk_here(N),!,fail ) ;
                     register_all_bkgoals(G),
                     register_bkgoal(PN),!,fail ).


ext_eq_args([A1|A],[NA1|NA],N):- ext_eq(A1,NA1,N),ext_eq_args(A,NA,N),!.

ext_eq_args([],[],_).


ext_eq(V1-T1,V,N):- var(V),V=V1-T1 ;
                    V=V2-T2,
                    ( var(V2),var_var(V1,T1,V2,T2,N) ;
                      var(V1),V1=V2,T1=T2 ;
                      atomic(V1),V1=V2 ;
                      V1=..[_|A1],V2=..[_|A2],ext_eq_args(A1,A2,N) ).


register_all_bkgoals((G1,G)):- register_all_bkgoals(G1),
                               register_all_bkgoals(G),!.

register_all_bkgoals((G1;G)):- register_all_bkgoals(G1),
                               register_all_bkgoals(G),!.

register_all_bkgoals(G):- G=..[!,_|A],all_bkgoals(A).


all_bkgoals([V1-T1|V]):- ( var(T1) ;
                           depends(T1,BG),register_bkgoals(BG),
                           ( atomic(V1) ; V1=..[_|A],all_bkgoals(A) ) ),
                         all_bkgoals(V).

all_bkgoals([]).


set_bkgoals(G,H,PN):- clause(H-[_,E],_),conflicts(G,H,E) ;
                      register_bkgoal(PN).
```

```
conflicts(G,H,E):- c_conflicts(G-_,H-_,[],CC),!,
                   e_conflicts(E,CC,C),register_bk_goals(C),!,fail.


c_conflicts(V-T,V-_,C,C):- number([T],0).

c_conflicts(GT-T,HT-_,C,NC):- GT=..[F|GA],HT=..[F|HA],
                              ( GA=[],NC=C ;
                                !,c_conflicts(GA,HA,C,NC) ) ;
                              depends(T,BG),!,BG\==[],least(C,BG,NC).

c_conflicts([GT1|GT],[HT1|HT],C,NC):- c_conflicts(GT1,HT1,C,IC),!,
                                      c_conflicts(GT,HT,IC,NC).

c_conflicts([],[],C,C).


depends([*|U],BG):- search(U,L),sort(L,BG).

depends([0],[]).

depends([N|_],[N]).


search([[N|T]|U],L):- inst(T,IL),L=[N|IL] ;
                      search(U,L).


inst([*|U],L):- !,search(U,L).

inst([0],[]).

inst([N|_],[N]).


sort([],[]).

sort([X],[X]).

sort(L,[S1|S]):- bubble(L,[S1|X]),sort(X,S).


bubble([X1,X2|X],[Y1,Y2|Y]):- ( X1>=X2,X0=X1,Y2=X2 ; X0=X2,Y2=X1 ),
                               bubble([X0|X],[Y1|Y]).

bubble([X],[X]).
```

```
least([X|A],[X|B],[X|C]):- least(A,B,C).

least([A1|A],[B1|B],C):- A1<B1,C=[A1|A] ; C=[B1|B].

least(X,[],X).

least([],X,X).


e_conflicts([],C,C).

e_conflicts([E1|E],C,NC):- e_conflict(E1,C,X),!,e_conflicts(E,X,NC).


e_conflict(E,C,NC):- decompose(E,DE),first(DE,FE,AE),
                     !,f_conflicts(FE,AE,C,NC).


decompose(V-_,V):- var(V).

decompose(T-K,[F-K|A]):- T=..[F|A].

decompose([T1|T],[D1|D]):- decompose(T1,D1),decompose(T,D).

decompose([],[]).


first([[X1|X]],[X1],[X]).

first([[X1|X]|L],[X1|Y1],[X|Y]):- first(L,Y1,Y).


f_conflicts([F1,F2],A,C,NC):- f_conflict(F1,F2,A,C,NC).

f_conflicts([F1,F2|F],[A1,A2|A],C,NC):- f_conflict(F1,F2,[A1,A2],C,X),!,
                                        f_conflicts([F1|F],[A1|A],X,Y),
                                      !,f_conflicts([F2|F],[A2|A],Y,NC).


f_conflict(X,Y,_,C,C):- var(X) ; var(Y).

f_conflict(F-_,F-_,A,C,NC):- a_conflicts(A,C,NC).

f_conflict(_-T1,_-T2,_,C,NC):- depends(T1,BG1),depends(T2,BG2),
                               merge(BG1,BG2,BG),!,BG\==[],
                               least(C,BG,NC).
```

```
a_conflicts([[],[]],C,C).

a_conflicts([[],[]|A],C,C):- a_conflicts([[]|A],C,C).

a_conflicts(A,C,NC):- first(A,A1,AN),e_conflict(A1,C,IC),
                      a_conflicts(AN,IC,NC).


merge([X|A],[X|B],[X|C]):- merge(A,B,C).

merge([A1|A],[B1|B],[X|C]):- A1>B1,X=A1,merge(A,[B1|B],C) ;
                                X=B1,merge(B,[A1|A],C).

merge([],X,X).

merge(X,[],X).


register_bksoal(0).

register_bksoal(N):- call(bksoal(N)) ; asserta(bksoal(N)) .


register_bksoals([N1|N]):- register_bksoal(N1),register_bksoals(N).

register_bksoals([]).


program_in(File):- see(File),repeat,read(C),
                   ( C==(:-end),seen ; convert(C),fail ),


convert((:-B)):- !,call(B),!.

convert((H:-B)):- !,new_head(H,NH),new_body(B,NB,0),
                  assert((NH:-NB)),!.

convert(UC):- !,new_head(UC,NUC),assert(NUC),!.


soal(G):- copy(G,CG),new_body(CG,NG,0),
          ( call(bksoal(N1)),N is N1+1 ; N=1 ),
          !,ib(NG,N,NN,0,_),
            ( NN=cut,!,fail ;
              ( old_body(G,NG) ;
                reset_bksoals(N),register_all_bksoals(NG),fail ) ).
```

```
new_head(H,NH-[C,E]):- H=..[F|A],var_list(A,NA),NH=..[F|NA],
                       var_list(NA,NA1),NH1=..[F|NA1],
                       ( call(pred(NH,_)) ; assert(pred(NH,NH1)) ),
                       new_head_args(A,NA,C-[],E1-[]),
                       remove_singles(E1,E).


new_head_args([A1|A],[NA1|NA],NC-C,NE-E):-
                       new_head_term(A1,NA1,IC-C,IE-E),
                       new_head_args(A,NA,NC-IC,NE-IE).

new_head_args([],[],C-C,E-E).


new_head_term(V,NV,C-C,NE-E):- var(V),( equal_var(V,E,NV,NE) ;
                                        NV=V,NE=[[V]|E] ).

new_head_term(V,NV-T,[T|NC]-C,NE-E):- V=..[F|A],
                                      new_head_args(A,NA,NC-C,NE-E),
                                      NV=..[F|NA].


equal_var(V,[E1|E],NV,[NE1|NE]):- E1=[X1|X],V==X1,NE1=[X1,NV|X],NE=E ;
                                  equal_var(V,E,NV,NE),NE1=E1.


remove_singles([],[]).

remove_singles([[_]|E],NE):- remove_singles(E,NE).

remove_singles([E1|E],[E1|NE]):- remove_singles(E,NE).


new_body((G1,G),(NG1,NG),N):- new_body(G1,NG1,N),new_body(G,NG,N),!.

new_body((G1;G),(NG1;NG),N):- new_body(G1,NG1,N),new_body(G,NG,N),!.

new_body(G,NG,N):- new_body_term(G,NG-_,N),!.


new_body_term(V,V,_):- var(V).

new_body_term(T,NT-[N],N):- T=..[F|A],new_body_args(A,NA,N),NT=..[F|NA].


new_body_args([A1|A],[NA1|NA],N):- new_body_term(A1,NA1,N),
                                   new_body_args(A,NA,N).
```

```
new_body_args([],[],_).


old_body((OG1,OG),(G1,G)):- old_body(OG1,G1),old_body(OG,G),!.

old_body((OG1;OG),(G1;G)):- old_body(OG1,G1),old_body(OG,G),!.

old_body(OG,G):- old_body_term(OG,G-_),!.


old_body_term(V,V-_):- var(V).

old_body_term(OT,T-_):- T=..[F|A],old_body_args(OA,A),OT=..[F|OA].


old_body_args([OA1|OA],[A1|A]):- old_body_term(OA1,A1),
                                 old_body_args(OA,A).

old_body_args([],[]).


var_list([],[]).

var_list([_|L],[_|NL]):- var_list(L,NL).


copy(X,CX):- assert(copying(X)),retract(copying(CX)).


:-end.
```

Appendix 2. The specialized database interpreter

```
:- 'LC'.

:- ext(unit,2,unit).

:- ext(non_unit,2,non_unit).


:- program([dbpred],[call_dbib(1)]).


:- mode dbib(+,+,-,+).

:- mode number(+,+).

:- mode nobkhere(+).

:- mode nobkuntil(+).

:- mode registerbkgoals(+).

:- mode args(+,?).

:- mode call_dbib(+).

:- mode register_all_bkgoals(+).

:- mode all_bkgoals(+).

:- mode conv_in(-,+).

:- mode conv_out(+,+).


dbib((P,Q),N1,N3,PN):- !,dbib(P,N1,N2,PN),
                        ( dbib(Q,N2,N3,PN) ;
                          nobkuntil(N1),!,fail ).

dbib((P;Q),N1,N2,PN):- !,( dbib(P,N1,N2,PN) ;
                           dbib(Q,N1,N2,PN) ).

dbib(P,N1,N2,PN):- unit(P-LP,NP),
                   ( N2 is N1+1, number(LP,N1),
                     NP,( true ;
                          nobkhere(N1),!,fail) ;
```

```
                    registerbkgoals([PN|LP]),!,fail ).

dbib(P,N1,N3,_):- N2 is N1+1,
                  ( non_unit(P,Q),( dbib(Q,N2,N3,N1) ;
                                    nobkhere(N1),!,fail ) ;
                    registerbkgoals([PN]),fail ).


number([N|L],N):-number(L,N),!.

number([_|L],N):-number(L,N),!.

number([],_).


nobkhere(N):- retract(bkgoal(N)),!,fail.

nobkhere(_).


nobkuntil(N):- call(bkgoal(N1)), N1>=N,!,fail.

nobkuntil(_).


registerbkgoals([N|L]):- (var(N) ; call(bkgoal(N)) ; asserta(bkgoal(N))),
                         !,registerbkgoals(L).

registerbkgoals([]).


call_dbib(I):- assert(input(I)),retract(input(G)),
               ( call(bkgoal(LN)),IN is LN+1 ; IN=1 ), PN is IN-1,
               conv_in(NG,G),!,dbib(NG,IN,N,PN),
               ( conv_out(NG,I) ;
                 reset_bkgoals(IN),register_all_bkgoals(NG),fail ).


register_all_bkgoals((G1,G)):- register_all_bkgoals(G1),
                               register_all_bkgoals(G),!.

register_all_bkgoals((G1;G)):- register_all_bkgoals(G1),
                               register_all_bkgoals(G),!.

register_all_bkgoals(G):- ( unit(G-L,_),registerbkgoals(L) ;
                            G=..[_|A],all_bkgoals(A) ),!.
```

```
all_bkgoals([]).

all_bkgoals([A1-N1|A]):- ( var(N1) ;
                           registerbkgoals([N1]),all_bkgoals(A) ).


conv_out((NP,NQ),(P,Q)):- conv_out(NP,P),conv_out(NQ,Q),!.

conv_out((NP;NP),(P,Q)):- conv_out(NP,P),conv_out(NQ,Q),!.

conv_out(NP,P):- P=..[N|A], NP=..[N|NA], args(A,NA),!.


conv_in((NP,NQ),(P,Q)):- conv_in(NP,P),conv_in(NQ,Q).

conv_in((NP;NQ),(P;Q)):- conv_in(NP,P),conv_in(NQ,Q).

conv_in(NP,P):- P=..[N|A],args(A,NA),NP=..[N|NA].


args([A|AS],[NA|NAS]):- arg(NA,A),args(AS,NAS).

args([],[]).


arg(X,X):- var(X).

arg(X-_,X).
```

Appendix 3. Database example

```
student(robert,prolog).

student(john,music).
student(john,prolog).
student(john,surf).

student(mary,science).
student(mary,art).
student(mary,physics).

professor(luis,prolog).
professor(luis,surf).

professor(eureka,music).
professor(eureka,art).
professor(eureka,science).
professor(eureka,physics).

course(prolog,mon,room1).
course(prolog,fri,room1).

course(surf,sun,beach).

course(maths,tue,room1).
course(maths,fri,room2).

course(science,thu,room1).
course(science,fri,room2).

course(art,tue,room1).

course(physics,thu,room3).
course(physics,sat,room2).
```

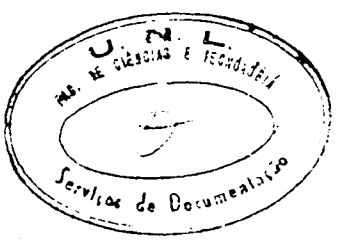
Additional module

```
:- 'LC'.

:- ext(unit,2,unit).

:- ext(non_unit,2,non_unit).

:- entry(unit,2).

:- entry(non_unit,2).


:- module(dbpred,[]).


:- mode unit(+,-).

:- mode non_unit(+,-).


unit((C1-NC1 \== C2-NC2)-[NC1,NC2] , (C1 \== C2)).

unit(student(S-NS,C-NC)-[NS,NC] , student(S,C)).

unit(prof(P-NP,C-NC)-[NP,NC] , prof(P,C)).

unit(course(C-NC,T-NT,R-NR)-[NC,NT,NR] , course(C,T,R)).


non_unit(query(S,P) , ( student(S,C1),
                        course(C1,T1,R),
                        prof(P,C1),
                        student(S,C2),
                        course(C2,T2,R),
                        prof(P,C2),
                        C1 \== C2  ) ).
```

Appendix 4. The sidetracking interpreter

```
:- program([],[load(1),si(1),ext_det(2)]).

:- mode load(+).
:- mode change(+).
:- mode build(-,+,+).
:- mode si(+).
:- mode sidetrack(+).
:- mode solve_det(+,-).
:- mode scan(+,-,+,-).
:- mode det(+,-).
:- mode step(+,+,-).
:- mode replace(+,+,-).

:- op(500,xfx,:).

load(P):- see(P),repeat,read(C),( C=(:-end),seen ;
                                  change(C),fail ).

change((H:-B)):- !,build(NB,B,X),assert(clause_(H,NB-X)).
change(H):- assert(clause_(H,X-X)).

build(NG,(G1,G),X):- !,build(NG,G1,X1),build(X1,G,X).
build(((NG1-X1;NG2-X2),X),(G1;G2),X):- !,build(NG1,G1,X1),
                                        build(NG2,G2,X2).
build((G,X),G,X).
```

```
si(G):- init,build(NG,G,X),sidetrack(NG-X).


init:- retract(twice) ; true.


sidetrack(G):- solve_det(G,G1-X1),!,
               ( var(G1),! ;
                 step(G1-X1,G2-X,X-X2),( var(G2),! ;
                                         sidetrack(G2-X2) ) ),


solve_det(G-X,NG):- scan(G,OG,SG-XS,D),
                    ( var(D),NG=SG-XS ;
                      X=SG,( var(OG) ;
                             !,solve_det(OG-XS,NG) ) ),


scan((G1,G),OG,SG,D):- det(G1,NG),( var(NG),!,fail ;
                                    OG=G,SG=NG,D=d ) ;
                       var(G),SG=(G1,X)-X ;
                       scan(G,OG,X-Y,D),SG=(G1,X)-Y .


det(G,NG):- call(ext_det(G,NG)).

det((C;G),NG):- !,call(C),det(G,NG).

det((_;_),_):- !,fail.

det(G,NG):- call(clause_(G,_)),( call(twice),retract(twice),!,fail ;
                                 assert(twice),fail ) ;
            retract(twice),call(clause_(G,NG)) ;
            call(G),NG=X-X ;
            true.


step((G1,G)-X,LG-L,RG):- replace(G1,LG-L1,NG),
                         ( var(NG),!,
                           nonvar(G),step(G-X,LG-L2,RG),L2=(G1,L) ;
                           L1=G,X=L,RG=NG ).


replace((G1;G2),G,X-X):- !,( G=G1 ;
                             G=G2 ).

replace((_;_),_,_).
```



```
replace(G,X-X,NG):- call(clause_(G,NG)).
```