CENTRO DE INFORMÁTICA

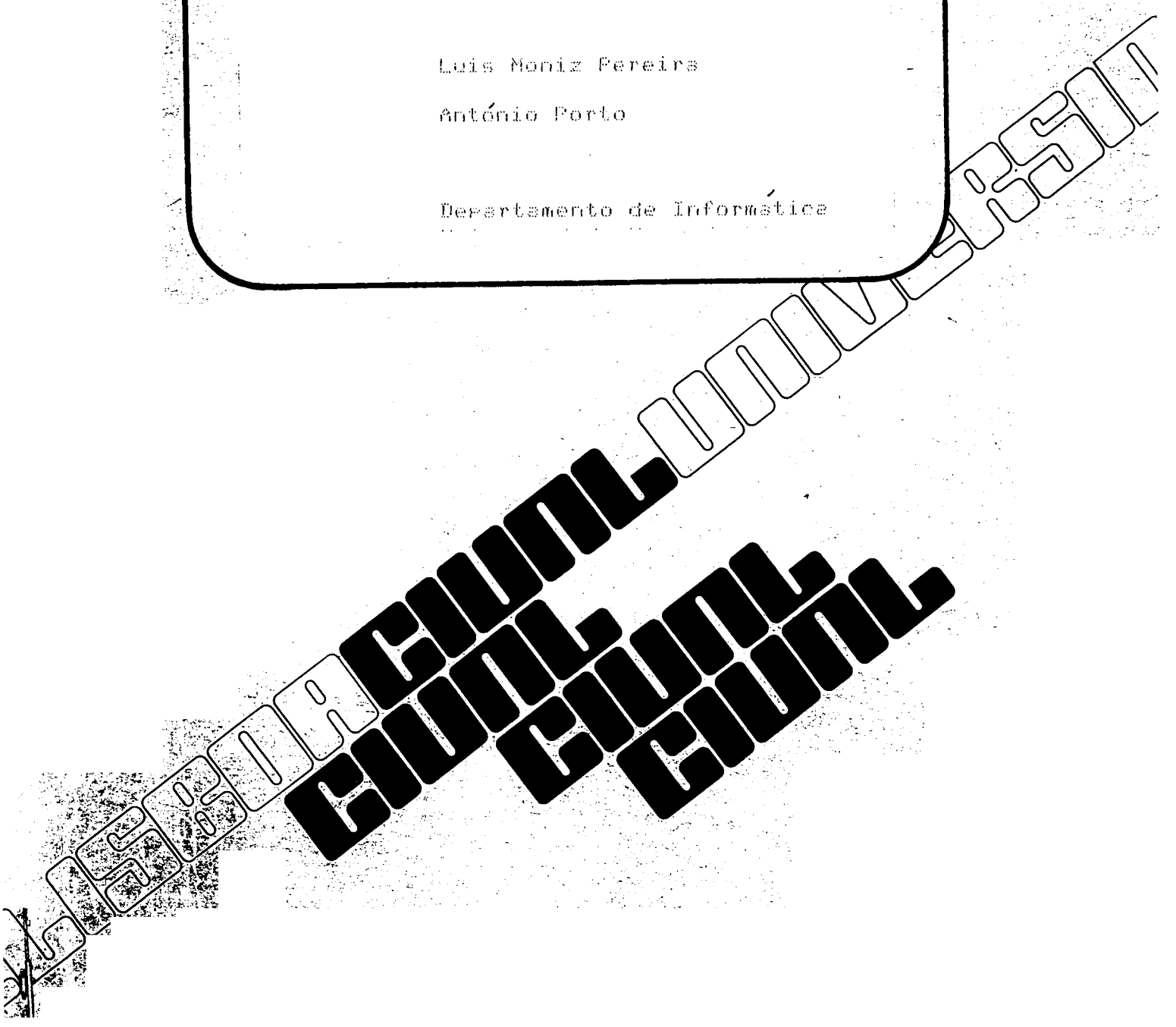
DA UNIVERSIDADE NOVA DE LISBOA

AN INTERPRETER OF LOGIC PROGRAMS USING SELECTIVE BACKTRACKING

Luis Moniz Pereira

António Porto

Departamento de Informática



AN INTERPRETER OF LOGIC PROGRAMS USING SELECTIVE BACKTRACKING

Luis Moniz Pereira

António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa, Portugal

July 1980

Report 3/80 CIUNL

Paper presented at
Workshop on Logic Programming
John von Neumann Computer Society
Debrecen, Hungary, July 1980

1. Introduction

In (3)(5) we presented a method for performing selective backtracking in Horn clause programs as applied to Prolog.

In this report, we describe a Prolog interpreter, written in Prolog, which performs selective backtracking on general Prolog programs. We discuss efficiency issues, and a simplified, and less selective though general version of the interpreter, which accomplishes a good trade-off compromise between selective power and its associated overhead. We also present a selective backtracker specialized to querying relational databases satisfying a reasonable set of assumptions.

The interpreter implementations presented here supersede those of an earlier report (4). They are written in Prolog itself, and are not aimed at efficiency but rather at clear, accurate and detailed description of our selective backtracking method. They should be viewed as working simulations at a high level of what requires a low level implementation to attain competitive efficiency. Our specialized relational database interpreter, nevertheless, is already competitive as it stands, even for smallish databases. We believe it can be of immediate use for Prolog programs consulting practical databases.

In this report we assume the reader has been exposed to our selective backtracking method (5) and is familiar with DECsystem-10 Prolog (7).

Because the selective backtracking interpreter is quite a complex program, it may be helpful to complement our description with running the interpreter on examples with tracing.

2. The selective backtracking interpreter

In this section we describe an interpreter of general Prolog programs, that works in full accordance with the theory of selective backtracking as expressed in (5). A listing of the interpreter is presented in Appendix 1. The interpreter may be called from the standard one at any number of places.

2.1 Representation

In this interpreter, terms within goals contain information on their own binding dependencies. In (5) we summed up such dependencies as follows:

"Because parent goals of failed goals are always selected as backtrack goals, the goal dependencies created by simple transmission of bindings up and/or down the tree (through chains of ancestors possibly linked by common variables at brother goals) should not be noted explicitly.

Any solved goal in whose match a goal variable was bound to a textual non-variable term must be retained as a modifying goal for any constant name which then became part of the binding of that variable.

The only other case in which a solved goal must also be retained as a modifying goal is when two (or more) still free variables in the goal have been unified to one another in the matching of the goal."

In order to be referenced, goals are numbered from 1 onwards, in the order they are activated. During backtracking this numbering is undone. To be able to convey dependencies, every term X is represented in the form

VX - TX

VX is the value of the term.

TX is the tag of X. It is associated just with the principal functor of X, for any subterm of X will have its own tag.

If X is a textual non-variable term, TX=t. If, otherwise, X is a textual variable term, then TX is a list

[BX|DX]

BX, the binding variable of X, shows the type of binding performed on X:

BX=d(N) if X was directly bound to a textual non-variable term in a clause head, during the match of goal number N.

BX=i if X was indirectly bound to a non-variable term, through some other textual variable bound to X.

DX is a (possibly empty) list of the dependencies of X created by unification with other textual variables. Each such dependency created by binding X to term Y is of the form

b(N,T)

where N is the number of the goal in whose match the binding was done.

If Y, itself of the form VY-TY, was a non-variable term when the binding was done, T=TY. If Y was a variable term, T=TY1, and TY is updated then to include an element b(N,TX1), expressing the dependency of Y on X. TY1 is the result of deleting b(N,TX1) from TY, and similarly TX1 is obtained from TX by excluding b(N,TY1) from it. Thus circularity of reference is prevented. This last 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 - [BX| DX]

Y = VY - [BY| DY]

Z = VZ - [BZ| DZ]

W = VW - [BW| DW]

After goal 1

X = V1 - [BX, b(1, [BY| DY1])| DX1]

Y = V1 - [BY, b(1, [BX| DX1])| DY1]

Z = VZ - [BZ| DZ]

W = VW - [BW| DW]

After soal 2

X = V1 - [BX, b(1, [BY| DY1])| DX1]

Y = V1 - [BY, b(1, [BX| DX1])| DY1]

Z = V2 - [BZ, b(2, [BW| DW1])| DZ1]

W = V2 - [BW, b(2, [BZ| DZ1])| DW1]

After soal 3

X = V - [BX, b(1, [BY, b(3, [BZ, b(2, [BW| DW1])| DZ2])| DY2])| DX1]

Y = V - [BY, b(1, [BX| DX1]), b(3, [BZ, b(2, [BW| DW1])| DZ2])| DY2]

Z = V - [BZ, b(2, [BW| DW1]), b(3, [BY, b(1, [BX| DX1])| DY2])| DZ2]

W = V - [BW, b(2, [BZ, b(3, [BY, b(1, [BX| DX1])| DY2])| DZ2])| DW1]

After soal 4

X = a - [d(4), b(1, [i, b(3, [i, b(2, [i])])])]

Y = a - [i, b(1, [d(4)]), b(3, [i, b(2, [i])])]

Z = a - [i, b(2, [i]), b(3, [i, b(1, [d(4)])])]

W = a - [i, b(2, [i, b(3, [i, b(1, [d(4)])])])]

It is easy to obtain from the tass the exact chain of soals throush which each variable acquired its value:

X -> 4

Y -> 1 - 4

Z -> 3 - 1 - 4

W -> 2 - 3 - 1 - 4

These are the modifying goals for each variable.

This choice of representation requires that the original clauses of a program be changed accordingly. So, before execution, the program is read from a file and modified.

Each clause is converted to another clause of the form

find_a_clause(H',B',I)

The clause head H is converted to the new head H', thereby producing the list I with information to be used by the interpreter. If the clause has a body B it is converted to the new body B', otherwise B'=true.

Every occurrence of a non-variable term X in H has the form X-T in H', a different variable T being associated with each occurrence of the same X. The first element of I is a list containing all such variables T.

Various occurrences of the same variable within H will have different variables standing for them in H', because matching terms do not necessarily have matching tags. For each multiple occurring variable X in H, the different variables standing for X in H' are put in a list L and X=L is made an element of I. As an example, the clause head

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

is converted to

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

and I is

[[G,F2,B,F1,A2,A1], X=[X1,X2,X3], Y=[Y1,Y2]]

To convert the body of a clause we simply replace every non-variable term X by X-t.

As an example, the goal

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

will be converted to

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

2.2 Operation

After reading and converting the program with 'load' any goal G can be executed with the directive 'goal(G)' . G is first converted to G' like the body of a clause, and interpretation begins with a call to

'execute', the main predicate of the selective backtracking interpreter. Upon successful interpretation, G' is converted back to its standard original form G.

Every call to 'execute' has the form

execute(G,N,M,Parent,C)

where G is the goal expression to be interpreted, N is the number of the first single goal to be solved within G, M is the number of the next single goal to be solved after solving G, Parent is the number of the parent goal of G, and C is set equal to 'cut' if a cut in G is activated.

2.2.1 Single goals

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

(1) Increment the number of the current goal.

(2) Find a clause for G.

(3) Digest info in this clause: set equal the terms so required (a stepwise unification procedure with updating of tags where necessary) and update the tags of variables directly matching head constants. All tags thus updated do not actually receive the number N of G until it is solved, and then only if needed. This will be done on the taginfo produced at this step.

(4) Execute the body B of the clause.

(5) If B was successfully executed, G is solved, and so, by using tag info, tags conveying dependency of terms on G acquire the constant N ; otherwise:

(6) If failure did not come from a cut in B, check whether N has been selected as a backtrack goal. If so, fail the call of 'execute' for G (this is where selective backtracking informs that no alternatives should be sought for G) ; however, if G is a backtrack goal, alternatives must be tried, so go back to (2).

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 'load'). If G has an external predicate, G is converted back to its original form, is executed by the standard interpreter, is reconverted again, and updating of its tags is performed. If not, the parent and the modifying goals for G are selected, and the call of 'execute' for G is failed.

2.2.2 Conjunctions and disjunctions

Having failed the execution of G2 in a conjunction (G1,G2), in case there are no backtrack goals within the execution of G1, backtracking over the whole execution of G1 takes place.

Disjunctions are handled in a straightforward way.

2.2.3 The cut

Special treatment is reserved for the cut symbol, two clauses being provided to handle the call 'execute(!,_,_,C)'.

The first one succeeds with 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 activated.

If backtracking reactivates a cut, the second clause will be activated, which asserts the fact that a cut was passed on backtracking.

In the execution of a conjunction (G1,G2), when G2 fails it is checked whether the current backtracking comes from a cut, in which case G1 is readily skipped. On the other hand, if G1 is to be skipped because there are no backtrack goals within its execution, it is checked whether a cut in G1 was activated, in which case the fact that backtracking past a cut occured is asserted.

When the parent P of the cut is reached, the fact asserting a failed cut is retracted and P is failed thus avoiding alternative clauses, as prescribed by the reactivated cut.

2.2.4 Obtaining backtracking information

Upon failure of a single goal G, the interpreter proceeds to select the parent and the modifying goals for G.

These are obtained according to the OR and AND rules of selective backtracking, by performing a conflict analysis on every clause head that failed to match G.

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 may change - the one in the goal, if not also textual.

The modifying goals for this conflict are obtained by searching the tag of the goal term for the dependencies that led to its conflicting binding.

e_conflicts :-

These occur when two different constant symbols are bound to two variables in the goal, that the clause head requires to hold the same value.

Since this conflict can be solved if either of the variables changes its value to match the other, the OR rule applies. After the modifying goals for each variable are obtained, they are merged into a single set, which is the set of modifying goals for this conflict.

If at any stage in the analysis an irrevocable conflict arises, since there are no modifying goals for it, no further analysis takes place, and no backtrack goals are selected for this clause.

Failure of a goal G caused by backtracking over a cut within the body of the activated clause for G is a very special case, since no amount of analysis is guaranteed to deliver all backtracking information. In fact, a possible solution to G 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 perhaps permitting activation of another one; but the variables are not carrying information on all the goals where such instantiation might be achieved. Besides not being able to set all backtracking information, trying to set the possible information proves to be too complicated. On account of this, the interpreter simply selects as backtrack goals all the modifying goals of non-variable terms in the goal, just to make sure.

2.2.5 Alternative solutions

If, upon successful execution of a goal, alternative solutions are wanted, backtracking into the interpreter must take place. The selective backtracking mechanism views the process of backtracking as the need to explore the search space relevantly, and not as the need to explore the search space thoroughly. One can view the search for alternative solutions as a user-generated failure of the previous solution. 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' modifying goals.

Thus, after forgetting any unused selected backtrack goals, all the arguments' modifying goals are selected and backtracking is re-instated.

3. Comments on efficiency

The overhead associated with implementing the general selective backtracking mechanism at a high level as we did in our interpreter is considerable, preventing any practical use of the interpreter.

It is not difficult to point out the two main sources of inefficiency:

First,the conflict analysis performed after each failure of a goal. Indeed, as this analysis is carried out for every clause for the predicate of the failed goal, when there are many such clauses, as in databases, efficiency will be severely affected.

Second, the complexity introduced by the tags. In fact, because of unification between goal variables (so required by multiple occurring variables in clause heads), unification has to be performed incrementally to allow updating of tags, which requires construction of new terms. Furthermore, tags can grow arbitrarily large, although, in practice, this seldom happens.

In view of this, we have written two other interpreters using simplified forms of selective backtracking.

One was devised to cope with the first mentioned source of inefficiency. It is basically the same as the general interpreter, but simply does not perform conflict analysis. The modifying goals for a failed goal are the modifying goals for all the arguments of the failed goal, irrespective of whether they have conflicted. A less selective but still general interpreter obtains.

The other interpreter again does not perform conflict analysis, and moreover limits each tag to a single goal number. This can only be achieved if restrictions are set up on the type of clauses to be interpreted, so this is not a general Prolog interpreter. It is intended for database query only, where the imposed restrictions are frequently met.

Average execution times are next shown for test runs of the three selective backtracking interpreters, along with a standard backtracking interpreter, on three examples found in (5): the map colouring, the database query and the non-attacking queens examples. The standard backtracking interpreter was written in Prolog in a similar way to the other three, for the sake of comparison. All interpreters were themselves interpreted by our current DECsystem-10 (KI) compiled Prolog interpreter. Times are given in seconds.

	Full Selective	Simplified Selective	Database Selective	Standard
Map colouring	3.40	1.35	0.84	1.00
Database query	7.40	2.64	1.38	1.54
4 queens	48.00	60.00	--	4.00

We see that the special database interpreter manages to be faster than the standard version on the first two examples.

The simplified general interpreter, although slower, is not very far from the efficiency of the standard one on the first two examples. Should the database be larger it would surely behave better.

Regarding the 4 queens example, the 48.00 seconds figure results from the nested list terms, whose list constructors' tags are unuseful for selective backtracking and burden unnecessarily the execution. All the more so because most calls are deterministic. The 60.00 seconds figure results from the ultimate dependency of every list structure on all previous calls. Since no conflict analysis takes place, all goals are consequently selected.

Anyway, this kind of high level implementation is not at all adequate, and we feel that many of the inefficiencies could be overcome by a low level implementation.

4. Description of an interpreter specialized to databases

The general interpreter previously described becomes a lot simpler 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 as before. The database may be compiled or interpreted, and does not need to be converted as before.

The interpreter is presented in Appendix 2.

4.1 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 must 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) Backtracking into the database query may only used for finding alternative solutions, not with the intention of exhaustively exploring a subspace (e.g. for certain types of side-effects).

4.2 Database access preparation

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

Because, in general, goal 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,...,An), p(A1-N1,...,An-Nn), [N1,...,Nn]).

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 bindings of the database variables (though they may test conditions and produce side-effects).

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 so that each of its ground terms is tagged initially with t. 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. This reverse form is unified with the original goal to produce the answer to the query.

In Appendix 3 we present the additional module containing the extra 'unit' and 'non_unit' clauses needed by the interpreter for the database example found in (5).

4.3 Tagging

Tagging of variable bindings is accomplished with a call to 'number'. 'number' takes the list of tags in the head of a (unit) clause and numbers those that are still free with the current goal number. As before, goals are numbered from 1 onwards. Upon backtracking the numbering is, of course, undone.

4.4 Obtaining backtrack goals on failure

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 modifying goal for some unit clause. Thus, each goal tagging a bound argument would be a modifying goal.

Consequently, for economy, we refrain from conflict analysis, and always take as modifying goals those in the tags of all 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.

4.5 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 obtaining 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).

4.6 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

database_query(G)

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

5. Conclusions

We have implemented our selective backtracking method (5) and showed it works as it should. Furthermore, our test runs indicate that selective backtracking requires a low level implementation of its own to become competitive with the compiled or other low level implementations of standard backtracking. However, the specialized relational database selective backtracker can be competitive, though implemented at a high level, even for smallish databases.

Finally, we believe that low level implementations of selective backtracking can be obtained by slight modification of standard backtracking implementations, if no conflict analysis is performed but all modifying goals in a goal are selected when it fails. Notice that a single element tag is just a pointer to the binding environment, which is easily accomodated by structure-sharing implementations. The most difficult problem is the less frequent case where tags can grow arbitrarily through bindings of free variables. The obvious candidate for storing these dependencies is the trail or reset list.

Acknowledgements

The support of the Instituto Nacional de Investigação Científica, through the Centro de Informática da Universidade Nova de Lisboa, and the computer facilities at Laboratório Nacional de Engenharia Civil are gratefully acknowledged.

6.References

(1) Coelho,H. ; Cotta,J.C. ; Pereira,L.M.

How to solve it with Prolos (2nd edition)
Laboratório Nacional de Ensenharia Civil, Lisbon 1980.

(2) Kowalski,R.A.

Losic for problem solvins
North-Holland Publ. Co. 1979.

(3) Pereira,L.M. ; Porto,A.

Intellisent backtrackins and sidetrackins
in Horn clause prosrams - the theory
Departamento de Informática
Universidade Nova de Lisboa, Lisbon 1979.

(4) Pereira,L.M. ; Porto,A.

Intellisent backtrackins and sidetrackins
in Horn clause prosrams - the implementation
Departamento de Informática
Universidade Nova de Lisboa, Lisbon 1979.

(5) Pereira,L.M. ; Porto,A.

Selective backtrackins for losic prosrams
5th Conference on Automated Deduction
Lecture Notes in Computer Science
Springer-Verlas 1980.

(6) Pereira,L.M.

Backtrackins intellisently in AND/OR trees
Departamento de Informática
Universidade Nova de Lisboa, Lisbon 1979.

(7) Pereira, L.M.; Pereira, FCN; Warren, D HD
User's guide to DECsystem-10 Prolog
Laboratório Nacional de Engenharia
Civil, Lisbon 1978

(8) Roussel,P.

Prolos: manuel de reference et d'utilisation
Groupe d'Intellisence Artificielle
Universite' d'Aix-Marseille II, Marseille 1975.

(9) Warren,D.H.D.

Implementins Prolos, Parts I and II
Department of Artificial Intellisence
Edinbursh University, Edinbursh 1977.

(10) Warren,D.H.D. ; Pereira,L.M. ; Pereira,F.C.N.

Prolos, the lansuase and its implementation compared with Lisp
ACM Symposium on Artificial Intellisence and
Prosramming Lansuases
Sisart Newsletter no.64, and Sisplan Notices vol.12,no.8, 1977.

Appendix 1. The selective backtracking interpreter

```
/*************************************************************************/
/*                                                                       */
/*              SELECTIVE BACKTRACKING INTERPRETER                       */
/*                                                                       */
/*************************************************************************/


/*      PROGRAM INPUT       */

load(Prosram) :- see(Prosram),
                 repeat,read(Statement),
                        ( Statement=end_of_file -> seen ;
                          convert(Statement),load_more ),


convert((:-Directive)) :- !,Directive,!.

convert((Head:-Body)) :-
                    !,new_head(Head,New_head,L),
                    new_body(Body,New_body),
                    assert(find_a_clause(New_head,New_body,L)),!.

convert(Unit_clause) :-
                    new_head(Unit_clause,New_unit_clause,L),
                    assert(find_a_clause(New_unit_clause,true,L)),!.


/*      TOP GOAL EXECUTION      */


soal(G) :- copy(G,CG),new_body(CG,NG),
           initialize,
           !,execute(NG,1,_,0,_),
             ( old_body(G,NG) ;
               deselect_soals,
               select_all_modifying_soals_for(NG),
               try_another_solution ),


initialize :- deselect_soals,( retract(cut_failure) ;
                               true ),
```

```
/*      EXECUTION       */


execute(true,N,N,_,_) :- !.

execute((G1,G2),N1,Nn,Parent,C) :- !,execute(G1,N1,Nk,Parent,C1),
                                   ( execute(G2,Nk,Nn,Parent,C),C=C1 ;
                                     ( cut_failure ;
                                       no_backtrack_goal_until(N1),
                                       ( no_previous_cut(C1) ;
                                         assert(cut_failure) ) ),
                                     !,fail ).

execute((G1;G2),N,M,Parent,C) :- !,( execute(G1,N,M,Parent,C) ;
                                     execute(G2,N,M,Parent,C) ).

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

execute(!,_,_,_,_) :- assert(cut_failure),!,fail.

execute(G,N,M,Parent,_) :- N1 is N+1,
                           ( find_a_clause(G,Body,Info),
                             dieset(Info,Tas_info),
                             ( execute(Body,N1,M,N,_),
                               number(Tas_info,N) ;
                               ( retract(cut_failure),
                                 !,N>1,
                                 deselect_goals_up_to(N),
                                 select(Parent),
                                 select_all_modifying_goals_for(G) ;
                                 not_a_backtrack_goal(N) ),
                                 !,fail ) ;
                             clause_head(G,G_skeleton),
                               !,N1>1,
                               select(Parent),
                               select_modifying_goals_for(G,G_skeleton),
                               !,fail ;
                             external(G,N,Parent),M=N1 ).



external(G,N,Parent) :- atom(G),( G,( true ;
                                      not_a_backtrack_goal(N),!,fail ) ;
                                  select(Parent),!,fail ).

external(G,N,Parent) :- old_body(OG,G),copy(OG,External_G),
                        ( External_G,
                          ( tas_external_goal(G,External_G,N) ;
                            not_a_backtrack_goal(N),!,fail ) ;
                          select(Parent),
                          select_all_modifying_goals_for(G),!,fail ).
```

```
no_previous_cut(no_cut).


/*        BACKTRACKING CONTROL        */


no_backtrack_soal_until(N) :- backtrack_soal(X),X>=N,!,fail ;
                              true.


not_a_backtrack_soal(N) :- retract(backtrack_soal(N)),!,fail ;
                           true.


/*        TAG MANIPULATION        */


disest([Constants|Equal_variables],[N|P]) :-
                                  set_equal(Equal_variables,N,P),
                                  check_new_constants(Constants,N),!.


set_equal([],_,[]).

set_equal([X=EX|E],N,P) :- separate(EX,Non_vars,Vars),
                           equal(Non_vars,Vars,X,N,P,Px),
                           set_equal(E,N,Px).


separate([],[],[]).

separate([X1|Xn],[X1|Non_vars],Vars) :- X1=X-_,nonvar(X),
                                        separate(Xn,Non_vars,Vars).

separate([X1|Xn],Non_vars,[X1|Vars]) :- separate(Xn,Non_vars,Vars).


equal([],Vars,X-T,_,[X-N|P],P) :- equal_vars(Vars,X-T,N).

equal([X-Tas],Vars,X-Tas,N,P,P) :- equal_vars_term(Vars,X,Tas,N).

equal(Non_vars,Vars,X-Tas,N,P,Px) :- split(Non_vars,Functor,Arss,Tas),
                                     equal_arss(X_arss,Arss,N,P,Px),
                                     X=..[Functor|X_arss],
                                     equal_vars_term(Vars,X,Tas,N).
```

```
equal_vars([X1,X2],X1,N) :- equal_var_pair(X1,X2,N).

equal_vars([X1,X2|Xn],X,N) :- equal_var_pair(X1,X2,N),
                              equal_vars([X2|Xn],X,N).


equal_var_pair(X-T1,X-T2,N) :- cross_tass(T1,T2,N).


cross_tass(T1,T2,N) :- T1=[_|_],T2=[_|_],
                       make(NT1,NTx1,T1,Tx1),make(NT2,NTx2,T2,Tx2),
                       Tx1=[b(N,NT2)|NTx1],Tx2=[b(N,NT1)|NTx2].


make(NT,NT,T,T) :- var(T).

make([X|NT],NTx,[X|T],Tx) :- make(NT,NTx,T,Tx).


equal_vars_term([],_,_,_).

equal_vars_term([X-[i|D]|Xn],X,T,N) :- close_tas(D,[b(N,T)]),
                                       equal_vars_term(Xn,X,T,N).


close_tas(T,X) :- var(T),T=X.

close_tas([i|D],X) :- close_tas(D,X).

close_tas([b(_,T)|D],X) :- close_tas(T,[]),close_tas(D,X).


split([X-T],F,[A],T) :- X=..[F|A].

split([X1-T1|Xn],F,[A1|An],T) :- X1=..[F|A1],
                                 split(Xn,F,An,Tk),
                                 choose(T,T1,Tk).


choose(T,T1,T2) :- search(T1,MG1),search(T2,MG2),
                   sort(MG1,SMG1),sort(MG2,SMG2),
                   least(SMG1,SMG2,SMG1) -> T=T1 ;
                                            T=T2.

choose(t,_,_).
```

```
search([d(N)|_],[N]) :- var(N) -> !,fail ;
                                  true.

search(t,[]).

search([i|D],MG) :- search(D,MG).

search([b(N,T)|D],MG) :- search(T,MGi),
                         ( var(N), -> MG=MGi ;
                                      MG=[N|MGi] ) ;
                         search(D,MG).


sort([],[]).

sort([X],[X]).

sort(L,[S1|S]) :- greatest(L,S1,NL),sort(NL,S).


greatest([X1,X2|Xn],X,[Xi|Xk]) :- X1<X2 -> Xi=X1,
                                     greatest([X2|Xn],X,Xk) ;
                                   Xi=X2,
                                   greatest([X1|Xn],X,Xk).

greatest([X],X,[]).


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

least([X|A],[Y|B],[X|A]) :- X<Y.

least([X|A],[Y|B],[Y|B]).

least(A,[],[]).

least([],B,[]).


check_new_constants([T1|Tn],N) :- ( check(T1) ;
                                    T1=[d(N)|D],close_tas(D,[]) ),
                                  check_new_constants(Tn,N).

check_new_constants([],_).


check([d(N)|_]) :- var(N),!,fail.

check(_).
```

```
number([N|P],N) :- check_free_vars(P,N),!.


check_free_vars([],_).

check_free_vars([X-Nk|P],N) :- ( nonvar(X) ;
                                 Nk=*=N ),
                               check_free_vars(P,N).


tas_external_soal(G,EG,N) :- new_body(EG,NG),
                             G=..[_|A],NG=..[_|NA],
                             equal_external_arss(A,NA,N),!.


equal_external_arss([A1|An],[NA1|NAn],N) :-
                                        equal_external(A1,NA1,N),
                                        equal_external_arss(An,NAn,N).

equal_external_arss([],[],_).


equal_external(X-T,NX,_) :- var(NX),NX=X-T.

equal_external(X-T1,NX-T2,N) :- ( var(NX),
                                    cross_tass(T1,T2,N) ;
                                  var(X),
                                    T1=[d(N)|D],close_tas(D,[]) ;
                                  atomic(X) ),
                                  X=NX ;
                                X=..[F|A],NX=..[F|NA],
                                equal_external_arss(A,NA,N).


/*      SELECTION       */


select_all_modifyins_soals_for((G1,G2)) :-
                                select_all_modifyins_soals_for(G1),
                                select_all_modifyins_soals_for(G2),!.

select_all_modifyins_soals_for((G1;G2)) :-
                                select_all_modifyins_soals_for(G1),
                                select_all_modifyins_soals_for(G2),!.

select_all_modifyins_soals_for(G) :-
                                G=..[_|A],
                                select_modifyins_soals_for_all(A),!.
```

```
select_modifying_goals_for_all([A1|An]) :-
                                    select_the_modifying_goals_for(A1),
                                    select_modifying_goals_for_all(An).

select_modifying_goals_for_all([]).


select_the_modifying_goals_for(X-_) :- var(X).

select_the_modifying_goals_for(X-T) :-
                                    ( search(T,MG) -> select_goals(MG) ;
                                                      true ),
                                    ( atomic(X) ;
                                      X=..[_|A],
                                      select_modifying_goals_for_all(A) ).


select_goals([N1|Nn]) :- select(N1),select_goals(Nn).

select_goals([]).


select(0).

select(N) :- backtrack_goal(N) ;
             asserta(backtrack_goal(N)).


select_modifying_goals_for(G,H) :- find_a_clause(H,_,[_|E]),
                                     conflicts(G,H,E) ;
                                   true.


deselect_goals :- deselect_goals_up_to(1).


deselect_goals_up_to(N) :- backtrack_goal(X),X>=N,
                             retract(backtrack_goal(X)),
                             try_to_find_more ;
                           true.


/*       CONFLICT ANALYSIS        */
```

```
conflicts(G,H,E) :- G=..[_|G_args],H=..[_|H_args],
                    c_conflicts(G_args,H_args,[],MGc_set),
                    !,e_conflicts(E,MGc_set,MG_set),
                    choose_the_least(MG,MG_set),
                    select_goals(MG),
                    !,fail.


c_conflicts([G_term|G_terms],[H_term|H_terms],Old_MG,New_MG) :-
                  c_conflict(G_term,H_term,Old_MG,MG),
                  !,c_conflicts(G_terms,H_terms,MG,New_MG).

c_conflicts([],[],MG,MG).


c_conflict(GX-_,HX-_,Old_MG,New_MG) :-
        simple(GX,HX) -> GX=HX,New_MG=Old_MG ;
                         GX=..[F|GX_args],HX=..[F|HX_args],
                         c_conflicts(GX_args,HX_args,Old_MG,New_MG).

c_conflict(_-Tag,_,Old_MG,[MG|Old_MG]) :- search(Tag,MG).


e_conflicts([],MG,MG).

e_conflicts([_-E1|E],Old_MG,New_MG) :- e_conflict(E1,Old_MG,MG),
                                       !,e_conflicts(E,MG,New_MG).


e_conflict([X1,X2],Old_MG,New_MG) :- pair_conflict(X1,X2,Old_MG,New_MG).

e_conflict([X1,X2|Xn],Old_MG,New_MG) :-
                                  pair_conflict(X1,X2,Old_MG,MG1),
                                  !,e_conflict([X1|Xn],MG1,MG2),
                                  !,e_conflict([X2|Xn],MG2,New_MG).


pair_conflict(X-_,Y-_,Old_MG,New_MG) :-
          simple(X,Y) -> X=Y,New_MG=Old_MG ;
                         X=..[F|X_args],Y=..[F|Y_args],
                         a_conflicts(X_args,Y_args,Old_MG,New_MG).

pair_conflict(_-T1,_-T2,Old_MG,[MG|Old_MG]) :- search(T1,MG1),
                                                ( search(T2,MG2),
                                                  merge(MG1,MG2,MG) ;
                                                  MG=MG1 ) ;
                                               search(T2,MG).
```

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

a_conflicts([X1|Xn],[Y1|Yn],Old_MG,New_MG) :-
                                    pair_conflict(X1,Y1,Old_MG,MG),
                                    a_conflicts(Xn,Yn,MG,New_MG).


simple(GX,HX) :- var(GX) ;
                 var(HX) ;
                 atomic(GX),atomic(HX).


choose_the_least(MG,[MG]).

choose_the_least(MG,[MG1|MGn]) :- sort(MG1,SMG1),
                                  set_the_least(MG,SMG1,MGn).


set_the_least(MG,SMG1,[MG2|MGn]) :- sort(MG2,SMG2),
                                    least(SMG1,SMG2,SMG),
                                    set_the_least(MG,SMG,MGn).

set_the_least(MG,MG,[]).


/*      CLAUSE CONVERSION       */


new_head(Head,New_head,[Constants|Equal_vars]) :-
                 Head=..[Predicate_name|Arss],
                 var_list(New_arss,Arss),
                 New_head=..[Predicate_name|New_arss],
                 ( call(clause_head(New_head,_)) ;
                   copy(New_head,Free_head),
                   assert(clause_head(New_head,Free_head)) ),
                 new_head_arss(Arss,New_arss,[],Constants,[],All_vars),
                 remove_single_vars(All_vars,Equal_vars),!.


var_list([],[]).

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


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

```
new_head_args([A1|An],[New_A1|New_An],C,New_C,V,New_V) :-
                    new_head_term(A1,New_A1,C,C1,V,V1),
                    new_head_args(An,New_An,C1,New_C,V1,New_V).

new_head_args([],[],C,C,V,V).


new_head_term(X,New_X,C,C,V,New_V) :- var(X),
                                      ( equal_var(X,V,New_X,New_V) ;
                                        New_V=[X=[New_X]|V] ).

new_head_term(X,New_X-Tas,C,[Tas|New_C],V,New_V) :-
                          X=..[Functor|Args],
                          new_head_args(Args,New_args,C,New_C,V,New_V),
                          New_X=..[Functor|New_args].


equal_var(X,[X1=EX1|E],New_X,[X1=[New_X|EX1]|E]) :- X==X1.

equal_var(X,[E1|E],New_X,[E1|New_E]) :- equal_var(X,E,New_X,New_E).


remove_single_vars([],[]).

remove_single_vars([X=[X]|E],New_E) :- remove_single_vars(E,New_E).

remove_single_vars([E1|E],[E1|New_E]) :- remove_single_vars(E,New_E).


new_body((G1,G2),(New_G1,New_G2)) :- new_body(G1,New_G1),
                                     new_body(G2,New_G2).

new_body((G1;G2),(New_G1;New_G2)) :- new_body(G1,New_G1),
                                     new_body(G2,New_G2).

new_body(G,New_G) :- G=..[Predicate_name|Args],
                     new_bodys_args(Args,New_args),
                     New_G=..[Predicate_name|New_args].


new_bodys_args([],[]).

new_bodys_args([A1|An],[New_A1|New_An]) :- new_body_term(A1,New_A1),
                                           new_bodys_args(An,New_An).


new_body_term(X,X) :- var(X).
```

```
new_body_term(X,X-t) :- atomic(X).

new_body_term(X,New_X-t) :- X=..[Functor|Args],
                            new_body_args(Args,New_args),
                            New_X=..[Functor|New_args].


old_body((Old_G1;Old_G2),(G1,G2)) :- old_body(Old_G1,G1),
                                     old_body(Old_G2,G2).

old_body((Old_G1;Old_G2),(G1;G2)) :- old_body(Old_G1,G1),
                                     old_body(Old_G2,G2).

old_body(Old_G,G) :- G=..[Predicate_name|Args],
                     old_body_args(Old_args,Args),
                     Old_G=..[Predicate_name|Old_args],!.


old_body_args([],[]).

old_body_args([Old_A1|Old_An],[A1|An]) :- old_body_term(Old_A1,A1),
                                          old_body_args(Old_An,An).


old_body_term(X,X-_) :- var(X) ;
                        atomic(X).

old_body_term(Old_X,X-_) :- X=..[Functor|Args],
                            old_body_args(Old_args,Args),
                            Old_X=..[Functor|Old_args].
```

Appendix 2. The specialized database interpreter

```
/*****************************************************************/
/*                                                               */
/*                    DATABASE INTERPRETER                       */
/*                                                               */
/*****************************************************************/


/*        QUERY EXECUTION        */


database_query(Q) :- copy(Q,CQ),new_query(CQ,NQ),
                     deselect_goals,
                     !,execute(NQ,1,_,0),
                     ( old_query(Q,NQ) ;
                       deselect_goals,
                       select_all_modifying_goals_for(NQ),
                       try_another_solution ).


/*      EXECUTION       */


execute((G1,G2),N1,Nn,Parent) :- !,execute(G1,N1,Nk,Parent),
                                 ( execute(G2,Nk,Nn,Parent) ;
                                   no_backtrack_goal_until(N1),
                                   !,fail ).

execute((G1;G2),N,M,Parent) :- !,( execute(G1,N,M,Parent) ;
                                   execute(G2,N,M,Parent) ).

execute(G,N,M,Parent) :- unit(Old_G,G,List_of_tags),
                         ( Old_G,
                           ( number(List_of_tags,N),
                             M is N+1 ;
                             not_a_backtrack_goal(N),
                             !,fail ) ;
                         select(Parent),
                         select_goals(List_of_tags),
                         !,fail ).

execute(G,N,M,Parent) :- non_unit(G,Body),
                         N1 is N+1,
                         ( execute(Body,N1,M,N) ;
                           not_a_backtrack_goal(N),
                           !,fail ) ;
                        select(Parent),
                        !,fail.
```

```
/*      BACKTRACKING CONTROL      */

no_backtrack_soal_until(N) :- backtrack_soal(X),X>=N,!,fail ;
                              true.

not_a_backtrack_soal(N) :- retract(backtrack_soal(N)),!,fail ;
                           true.

/*      TAG MANIPULATION        */

number([T1|Tn],N) :- ( var(T1) -> T1=N ;
                                  true ),
                     number(Tn,N),!.

number([],_).

/*      SELECTION       */

select_all_modifyins_goals_for((G1,G2)) :-
                                  select_all_modifyins_goals_for(G1),
                                  select_all_modifyins_goals_for(G2),!.

select_all_modifyins_goals_for((G1;G2)) :-
                                  select_all_modifyins_goals_for(G1),
                                  select_all_modifyins_goals_for(G2),!.

select_all_modifyins_goals_for(G) :-
                                  G=..[_|A],
                                  select_modifyins_goals_for_all(A),!.


select_modifyins_goals_for_all([A1-N|An]) :-
                                  select(N),
                                  select_modifyins_goals_for_all(An).

select_modifyins_goals_for_all([]).
```

```
select_goals([N1|Nn]) :- select(N1),select_goals(Nn).

select_goals([]).


select(t).

select(0).

select(N) :- backtrack_goal(N) ;
             asserta(backtrack_goal(N)).


deselect_goals :- retract(backtrack_goal(_)),retract_more ;
                  true.


/*        QUERY CONVERSION        */


copy(Q,CQ) :- assert(copying(Q)),retract(copying(CQ)).


new_query((Q1,Q2),(New_Q1,New_Q2)) :- new_query(Q1,New_Q1),
                                      new_query(Q2,New_Q2).

new_query((Q1;Q2),(New_Q1;New_Q2)) :- new_query(Q1,New_Q1),
                                      new_query(Q2,New_Q2).

new_query(Q,New_Q) :- Q=..[Predicate_name|Args],
                      new_query_args(Args,New_args),
                      New_Q=..[Predicate_name|New_args].


new_query_args([],[]).

new_query_args([A1|An],[New_A1|New_An]) :- new_query_term(A1,New_A1),
                                           new_query_args(An,New_An).


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

new_query_term(X,X-t).


old_query((Old_Q1;Old_Q2),(Q1,Q2)) :- old_query(Old_Q1,Q1),
                                      old_query(Old_Q2,Q2).
```

```
old_query((Old_Q1;Old_Q2),(Q1;Q2)) :- old_query(Old_Q1,Q1),
                                      old_query(Old_Q2,Q2).

old_query(Old_Q,Q) :- Q=..[Predicate_name|Args],
                      old_query_args(Old_args,Args),
                      Old_Q=..[Predicate_name|Old_args],!.


old_query_args([],[]).

old_query_args([Old_A1|Old_An],[A1|An]) :- old_query_term(Old_A1,A1),
                                           old_query_args(Old_An,An).


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

Appendix 3. Additional module for the database example

```
unit( (C1 \== C2) , (C1-T1 == C2-T2) , [T1,T2] ).

unit( student(S,C) , student(S-TS,C-TC) , [TS,TC] ).

unit( professor(P,C) , professor(P-TP,C-TC) , [TP,TC] ).

unit( course(C,D,R) , course(C-TC,D-TD,R-TR) , [TC,TD,TR] ).


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