CENTRO DE INFORMÁTICA

DA UNIVERSIDADE NOVA DE LISBOA

SELECTIVE BACKTRACKING AT WORK

Luis Moniz Pereira

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
[illegible]

SELECTIVE BACKTRACKING AT WORK

Luis Moniz Pereira

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
1899 Lisboa, Portugal

October 1980

Report 9/80 CIUNL

Abstract

In this report we describe a Prolog interpreter, written in Prolog, which performs selective backtracking on general Prolog programs, and discuss efficiency issues.

Contents

1 Introduction

In (4)(6) we presented a method for performing selective backtracking in Horn clause programs as applied to Prolog (2)(8)(9)(10)(11).

In this report we describe a Prolog interpreter, written in Prolog, which performs selective backtracking on general Prolog programs, and discuss efficiency issues.

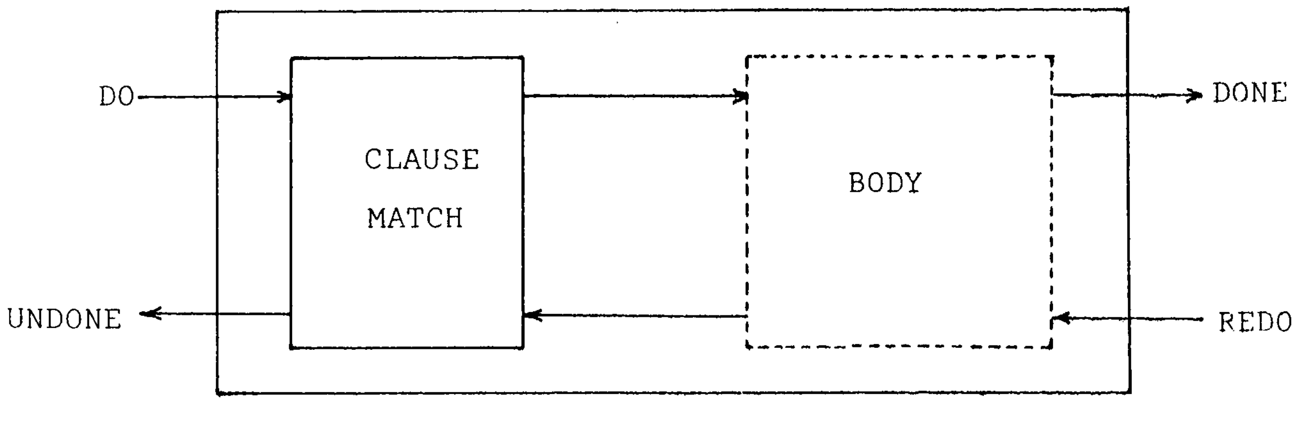
The interpreter implementations refered here supersede those of earlier reports (5)(7). Being written in Prolog itself, they are not aimed at efficiency but rather at clear, accurate and detailed description of our selective backtracking method. They should be viewed as working simulations at a high level of what requires a low level implementation to attain competitive efficiency.

We assume the reader has been exposed to our selective backtracking method (6) and is familiar with DECsystem-10 Prolog (8).

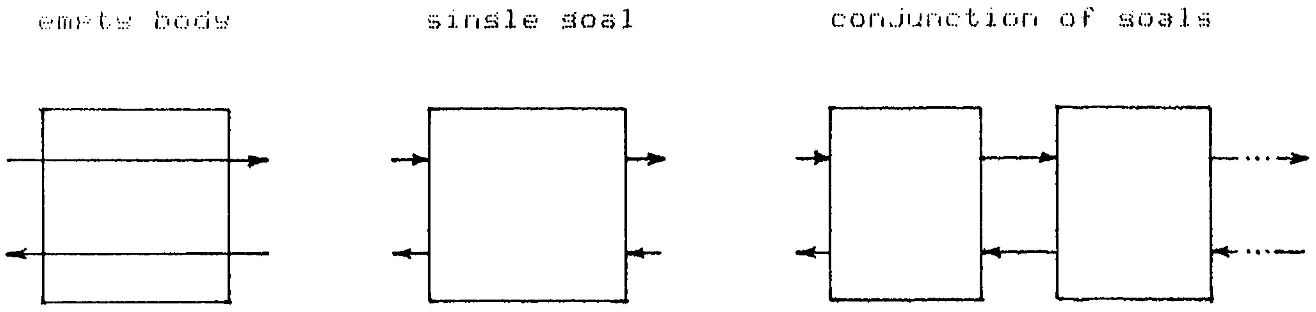
In Appendix 1 we present the core of the interpreter. Full listings will be provided on request.

2 A review of selective backtracking

The basic ideas of selective backtracking are illustrated and explained in the figures and text below. We depict each goal execution as a box with four ports, following Byrd (1). The DO port is entered when the goal is first activated, whereas the DONE port is exited on complete execution of the goal. Backtracking re-enters the goal execution via the REDO port, and exits the UNDONE port on unsuccessful execution. The goal execution box itself decomposes into two similar boxes: the clause matching box and the body execution box.

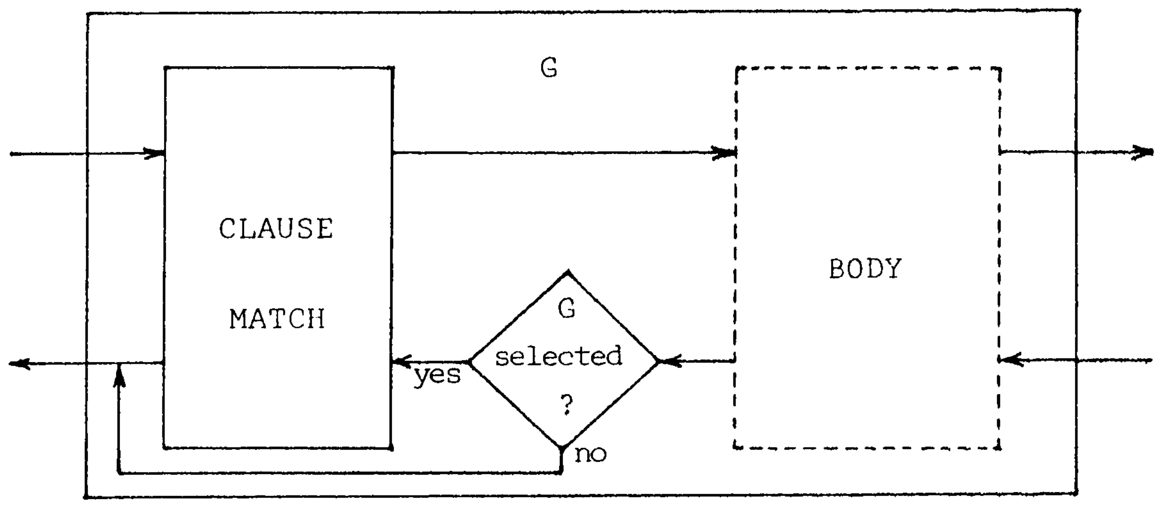


The body execution box is one of these:



Backtracking to a goal G actually consists in entering the REDO port of the clause matching box for G.

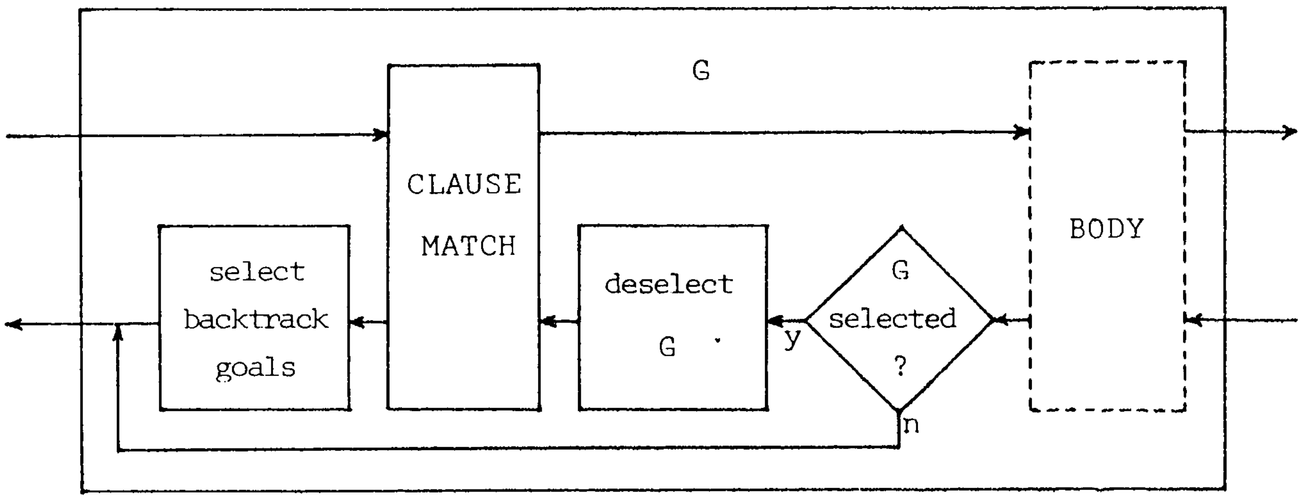
The main idea of selective backtracking is allowing entry into that port only when goal G has been selected as a backtrack goal; otherwise control must flow directly to the undone port of G.



Now, to implement this idea, there must be boxes for the selection and deselection of goals.

When should a goal S be selected for backtracking? When S can hopefully remedy an occurred failure of a goal G. That is to say, when backtracking to S and taking an alternative solution path for it can either avoid reactivation of G (S is an avoiding goal for G) or modify the bindings of the arguments of G in such a way that G may not fail again (S is a modifying goal for G). It is then obvious that selection of backtrack goals should be carried out after a goal has failed, ie. at the UNDONE port of its clause matching box.

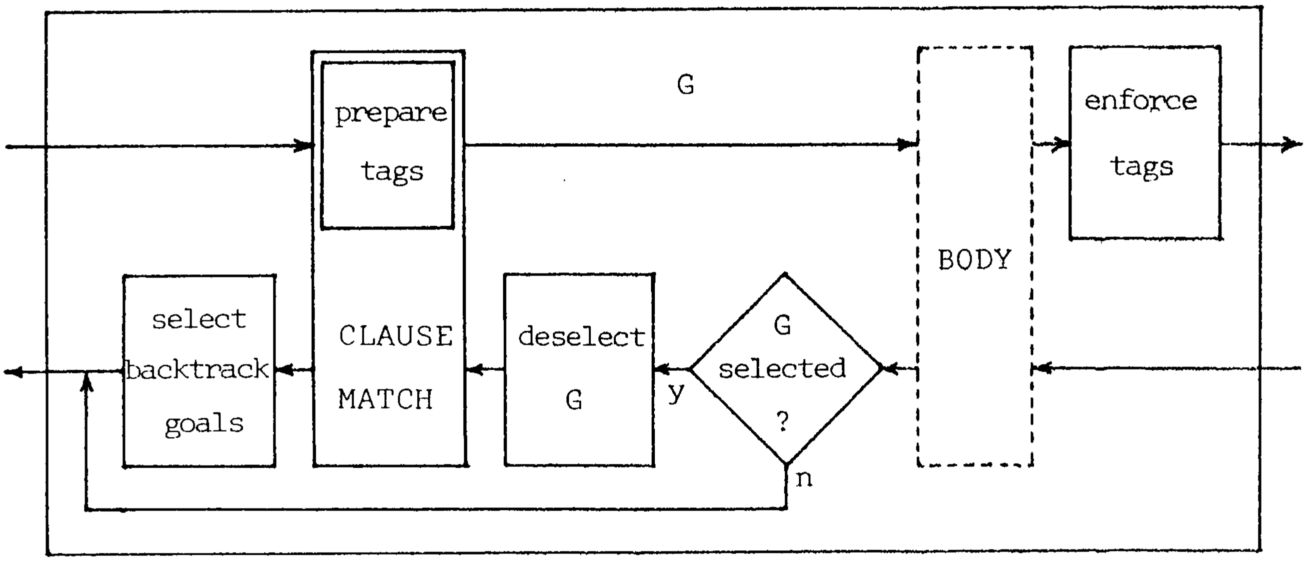
A soal should be deselected when backtrackins to it occurs.



The parent of a failed soal (its avoidins soal) is always selected.

Selection of modifyins soals is based on an analysis of the dependencies on other soals of the failure-orisinatins terms present in a failed soal. Such dependencies are noted by tassins each bindins with a reference to the soal whose match produced it.

G is an avoidins soal for any of its failed descendants, and not a modifyins soal. Therefore, tass of bindinss performed in a match of G are prepared durins the match but enforced only after the body of the matchins clause is done.



A more efficient execution can be achieved if information regarding goal determinism is used.

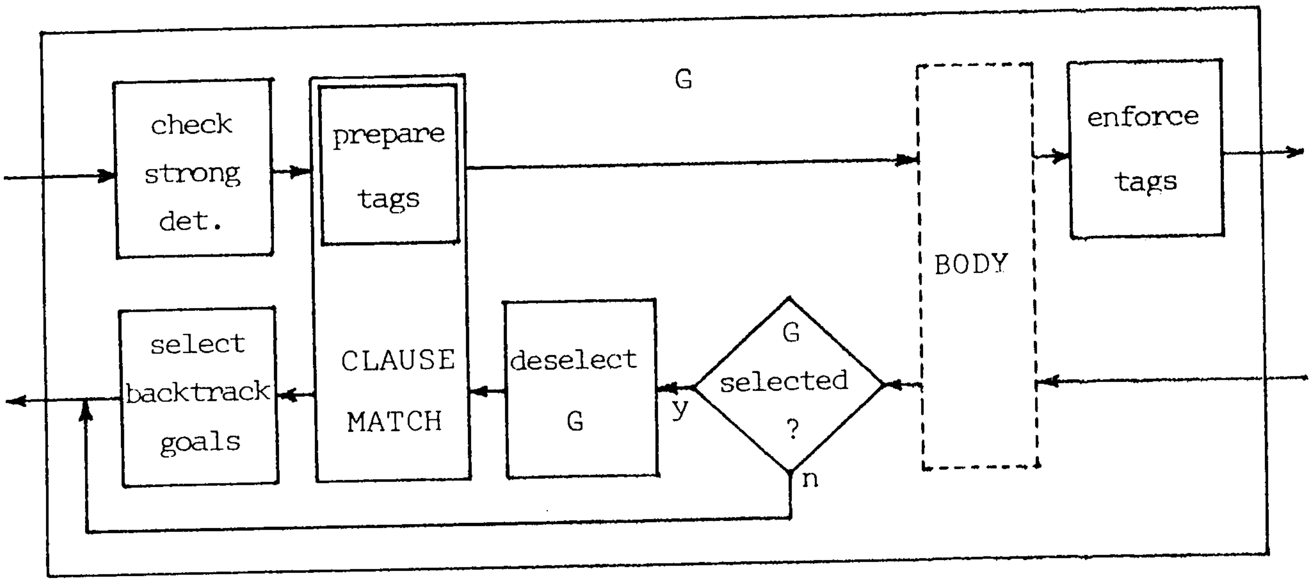
We say a goal is strongly-deterministic if some constant symbols in its arguments cannot be replaced, and those symbols only allow the goal to possibly match one clause (the unreplaceable symbols are those textual in the goal or acquired at strongly-deterministic matches).

It is irrelevant to backtrack to a strongly-deterministic goal since no alternative clauses exist for it, and analysis of its failed matches is irrelevant since the unreplaceable symbols will never allow it to match other clauses even after backtracking.

The bindings produced in a strongly-deterministic match rather than being tagged with a reference to the matching goal must be tagged with a reference to the goal's most recent non-strongly-deterministic ancestor - its avoiding goal.

Naturally, whenever a goal fails, the avoiding goal is now selected rather than the parent.

Also, tag preparation and enforcing are suitably modified to cater for the different tagging required if strong-determinism is detected.



3 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 (6), extended to include treatment of strongly-deterministic goals as explained in the previous section.

3.1 Representation

In this interpreter, terms within goals contain information on their own binding dependencies. In (6) we summed up such dependencies:

"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."

Now, only non-strongly-deterministic goals create dependencies. Accordingly, the avoiding goal that is always selected on failure is the most recent non-strongly-deterministic ancestor instead of the parent, and only non-strongly-deterministic goals are modifying goals.

In order to be referenced, non-strongly-deterministic goals are numbered from 1 onwards, in the order they are activated. During backtracking this numbering is undone. Also, each non-strongly--deterministic goal sets a reference variable, which will be bound to the goal's number only upon successful execution.

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 a goal G. N is the reference variable of G, or that of G's avoiding goal, in case G is strongly-deterministic.

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 Y is of the form

b(N,T)

where N is the reference variable of the goal G in whose match the binding was done, or that of G's avoiding goal if G is strongly--deterministic.

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 was 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 refernce 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 goal 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 goal 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 goal 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 tags the exact chain of goals through which each variable acquired its value - they are the modifying goals for the variable:

```
X -> 4

Y -> 1 - 4

Z -> 3 - 1 - 4

W -> 2 - 3 - 1 - 4
```

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

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.

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)))

is converted to

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

3.2 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, any backtracking information remaining from a previous execution is deleted, and interpretation starts with a call to 'execute', the main predicate of the selective backtracking interpreter. Upon successful execution, G' is converted back to its original form G.

Every call to 'execute' has the form

execute(G,N1,Nn,A,Ax,C)

where:

G is the goal expression to be interpreted.

N1 and Nn are respectively the entry and exit values for the number to be assigned to non-strongly-deterministic goals.

A and Ax are respectively the number and the reference variable of the avoiding goal for G.

C is to be set equal to 'cut' if a cut in G is activated.

3.2.1 Single goals

Execution of a single goal G is carried out along the following steps:

(1) Check if G is strongly-deterministic.

(2) Find a clause matching G.

(3) Digest info for 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 receive the reference variable of G, or that of G's avoiding goal if G is strongly-deterministic.

(4) Execute the body B of the clause. The number and reference variable of the avoiding goal for B are those of G, or those of G's own avoiding goal, if G is strongly-deterministic.

If B is successfully executed:

(5) Bind the reference variable of G to its number, if G is non--strongly-deterministic.

If execution of B does not succeed:

(6) Unless failure comes from a cut in B, check whether G 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 avoiding and the modifying goals for G are selected, and the call of 'execute' for G is failed.

3.2.2 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.

3.2.3 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 instanciation 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.

3.2.4 Obtaining backtracking information

Upon failure of a single goal G, the interpreter proceeds to select the avoiding 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 modifying goals are retained for the whole mismatch.

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 get all backtracking information, trying to get 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.

3.2.5 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 previous solution, ie. the arguments of the top goal. Now, the goals where those arguments may be modified are precisely all the arguments' modifying goals.

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

4 Comments on efficiency

The overhead associated with implementing the selective backtracking mechanism at a high level as we did advises against any practical use of the interpreter. Compared to compilation of source programs (or even interpretation) it clearly shows less efficiency.

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

First, the complexity introduced by the tags. In fact, 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 they seldom grow very much.

Second, 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.

To alleviate this second source of inefficiency, we have written a modified interpreter using a simplified form of selective backtracking. It is the same as the original interpreter, except that it 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.

Average execution times are exhibited below for test runs of the two selective backtracking interpreters, on three examples found in (5): the map colouring, the database query and the non-attacking queens examples. Times are also shown for an interpreter using standard backtracking, also written in Prolog and using the same system predicate for accessing user program's clauses as the two selective backtracking interpreters, for the sake of just comparison. All three interpreters were compiled by our DECsystem-10 (KI) Prolog compiler. Finally, times are given also for the Prolog system (compiled) interpreter, which makes use of a more efficient internal representation of clauses not available to the other interpreters.

	Full Selective	Simplified Selective	Standard	System
Map colouring	0.76	0.40	0.60	0.26
Database query	1.60	0.70	0.62	0.26
4 queens	3.18	3.22	1.22	0.50

Times are in seconds

These results persuade us that a low-level implementation of selective backtracking is worthwile. First, because at a low level most dependencies are implicit in pointers, and the extra tagging required will not slow down unification significantly. Second, because the results above show that conflict analysis probably does not payoff in general and a simplified form of backtrack goals selection is enough. The more so because a low-level implementation of conflict analysis would not specifically curtail its overhead.

It is significant that if information about strongly-deterministic goals is not used in the queens example (the only one that does use it) times more than double.

5 Conclusions

We have implemented our selective backtracking method (6) and showed it works as it should. Also we enhanced it by introducing the notion of strongly-deterministic goals. 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.

Finally, we believe that low-level implementations of selective backtracking can be obtained by a not excessive modification of standard backtracking implementations, if no conflict analysis is performed but all modifying goals in a goal are selected when it fails.

Acknowledgements

The support of the Instituto Nacional de Investigacao Cientifica, through the Centro de Informatica da Universidade Nova de Lisboa, and the computer facilities at Laboratorio Nacional de Engenharia Civil are gratefully acknowledged.

6 References

(1) Byrd, L.

Understanding the control flow of Prolog programs
Logic Programming Workshop, Debrecen, Hungary 1980.

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

How to solve it with Prolog (2nd edition)
Laboratorio Nacional de Engenharia Civil, Lisbon 1980.

(3) Kowalski,R.A.

Logic for problem solving
North-Holland Publ. Co. 1979.

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

Intelligent backtracking and sidetracking
in Horn clause programs - the theory
Departamento de Informatica
Universidade Nova de Lisboa, Lisbon 1979.

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

Intelligent backtracking and sidetracking
in Horn clause programs - the implementation
Departamento de Informatica
Universidade Nova de Lisboa, Lisbon 1979.

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

Selective backtracking for logic programs
5th Conference on Automated Deduction
Lecture Notes in Computer Science
Springer-Verlag 1980.

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

An interpreter of logic programs using selective backtracking
Logic Programming Workshop, Debrecen, Hungary 1980.

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

User's guide to DECsystem-10 Prolog
Laboratorio Nacional de Engenharia Civil, Lisbon 1978.

(9) Roussel,P.

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

(10) Warren,D.H.D.

Implementing Prolog -
Compiling predicate logic programs
Department of Artificial Intelligence
Edinburgh University, Edinburgh 1977.

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

Prolog, the language and its implementation compared with Lisp
ACM Symposium on Artificial Intelligence and
Programming Languages
Sigart Newsletter no.64, and Sigplan Notices vol.12,no.8, 1977.

Appendix 1   The core of the interpreter

```
/*        PROGRAM INPUT        */


load(Program) :- see(Program),
                 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),
                 recordz(New_head,clause(New_head,New_body,L),_),!.

convert(det(G,C)) :- new_head(G,New_G,_),
                     assert(det(New_G,C)),!.

convert(Unit_clause) :-
         new_head(Unit_clause,New_unit_clause,L),
         recordz(New_unit_clause,clause(New_unit_clause,true,L),_),!.



/*        TOP GOAL EXECUTION        */


goal(G) :- copy(G,CG),new_body(CG,NG),
           initialize,
           !,execute(NG,1,_,0,_,_),
             ( old_body(NG,G) ;
               deselect_goals,
               select_all_modifying_goals_for(NG),
               try_another_solution ).


initialize :- deselect_goals,( retract(cut_failure) ;
                                true ).


/*        EXECUTION        */


execute(true,N,N,_,_,_) :- !.
execute((G1,G2),N1,Nn,A,Ax,C) :- !,execute(G1,N1,Nk,A,Ax,C1),
                                 ( execute(G2,Nk,Nn,A,Ax,C),C=C1 ;
                                   ( cut_failure ;
                                     no_backtrack_goal_until(N1),
                                     ( no_previous_cut(C) ;
                                       assert(cut_failure) ) ),
                                   !,fail ).

execute((G1;G2),N1,Nn,A,Ax,C) :- !,( execute(G1,N1,Nn,A,Ax,C) ;
                                     execute(G2,N,M,A,Ax,C) ).

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

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

execute(G,N1,Nn,A,Ax,_) :-
                 check_det(G,N1,N2,A,Ax,N,Nx),
                 ( recorded(G,clause(G,Body,Info),_),
                     prepare_tags(Info,Nx),
                     ( execute(Body,N2,Nn,N,Nx,_),
                         enforce_tags(N1,N2,N,Nx) ;
                       ( retract(cut_failure),
                           !,N1>1,
                           deselect_goals_up_to(N1),
                           select(A),
                           ( N2=N1 ;
                             select_all_modifying_goals_for(G) ) ;
                         not_a_backtrack_goal(N1) ),
                         !,fail ) ;
                   clause_head(G,G_skeleton),
                     !,N1>1,
                     select(A),
                     ( N2=N1 ;
                       select_modifying_goals_for(G,G_skeleton) ),
                     !,fail ;
                   external(G,N1,A),Nn=N2 ).


external(G,_,A) :- atom(G),( G,! ;
                             select(A),!,fail ).

external(G,N,A) :- old_body(G,OG),copy(OG,External_G),
                   ( External_G,
                       ( tag_external_goal(G,External_G,N) ;
                         not_a_backtrack_goal(N),!,fail ) ;
                     select(A),
                     select_all_modifying_goals_for(G),!,fail ).


check_det(G,N1,N1,A,Ax,A,Ax) :- det(G,Conditions),check(Conditions),!.

check_det(_,N1,N2,_,_,N1,_) :- N2 is N1+1.
```