UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE INFORMÁTICA

SELECTIVE BACKTRACKING

Luis Moniz Pereira

Antônio Porto

UNL/FCT-11/81 [illegible]

SEMINÁRIO DOS OLIVAIS
QUINTA DO CABEÇO

SELECTIVE BACKTRACKING

Luis Moniz Pereira
António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa Codex, Portugal

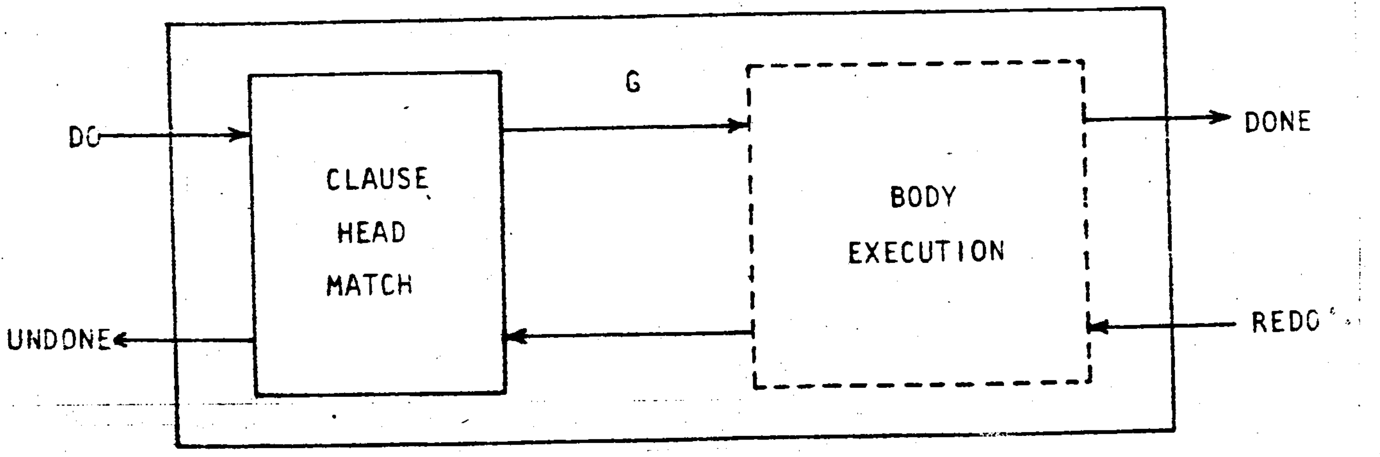
Introduction

In (4)(6) we presented a method for performing selective backtracking in Horn clause programs as applied to Prolog (2)(8)(9)(10). It is, in fact, a specialization to the depth-first strategy of a more general form of intelligent backtracking (3).

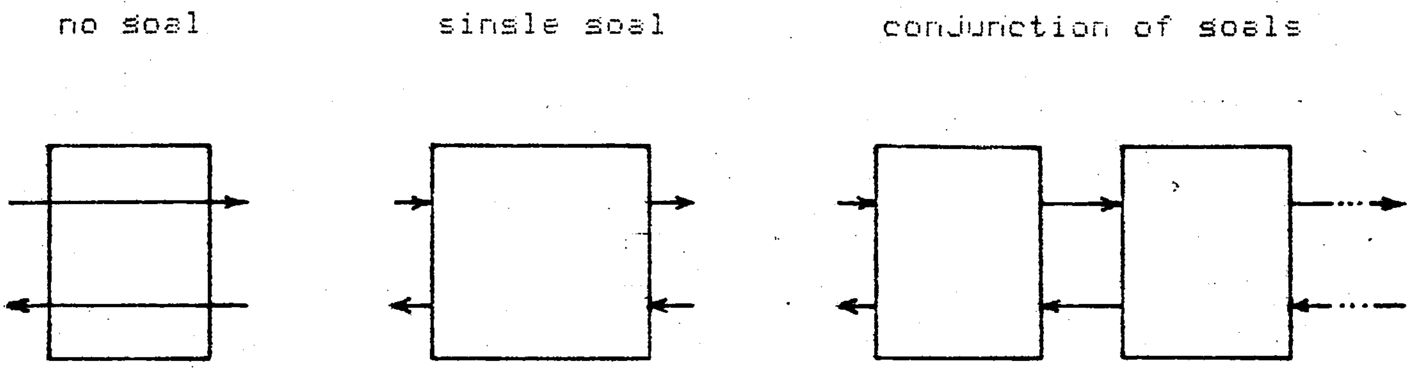
In this paper we review selective backtracking and address general implementation issues.

Basic ideas

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. On backtracking, control 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 head matching box and the clause body execution box.

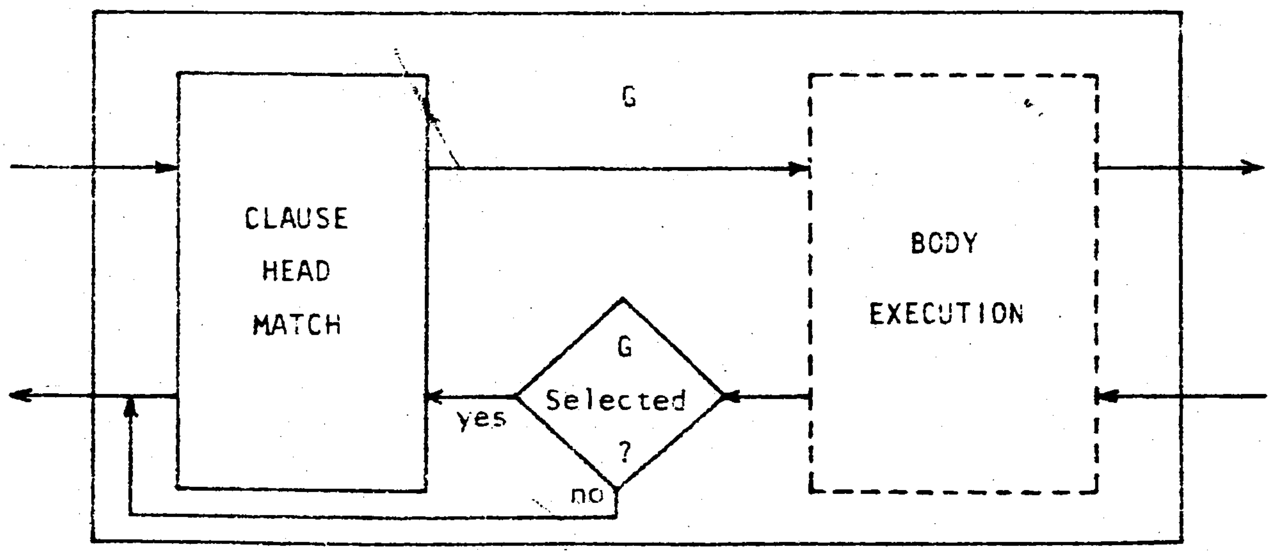


After matching goal G with a clause head, the clause body execution box for G becomes one of these:



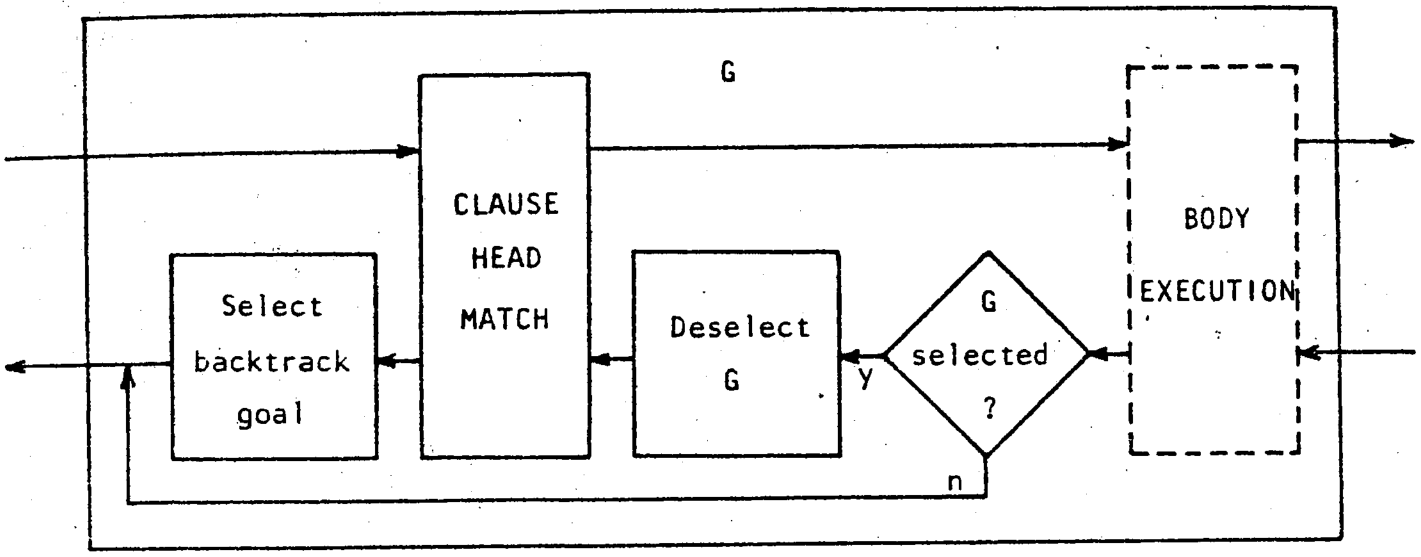
Backtrackins to a soal G actually consists in enterins the REDO port of the clause head matchins box for G.

The main idea of selective backtrackins is to select, at each failed soal, a sinsle soal to backtrack to, not necessarily the previous soal as in standard backtrackins. Consequently, entry into the REDO port of G is allowed only if G has been selected as the backtrack soal for the last failed soal; otherwise control flows directly to the UNDONE port of G.



At the UNDONE port of the clause head matchins box, ie. after failure, selection of the backtrack soal takes place.

Deselection of the backtrack soal is done when backtrackins to it occurs, ie. at the REDO port of its clause head matchins box.



How is the backtrack goal selected?

Upon failure of a goal G, the backtrack goal is chosen among the candidate goals for backtracking. These are:

1) The ancestors of G - These are the goals whose alternative clauses avoid reactivation of G.

2) The modifying goals for G - These are the goals whose match if undone will undo failure-originating bindings in the arguments of G. In the sequel we refer how they are obtained.

3) The legacy set of G - This is the set of candidate backtrack goals inherited from failed goals that selected G as the backtrack goal; they must be kept as candidates for backtracking to ensure completeness of the search.

Among all the candidates, the most recently activated one is selected as the backtrack goal. The remaining ones become part of its legacy set.

Refining the set of candidates

The set of candidates for backtracking can be refined as specified next.

The ancestors - Among the ancestors, it suffices to retain the parent as candidate, since if and when the parent fails, then its parent becomes a candidate. This way no ancestor will be left out.

The modifying goals - Rather than the whole set of modifying goals, some subsets may be used instead, without loosing completeness; they are the global modifying sets.

A global modifying set is the union of local modifying sets, where each of these is associated with the mismatch of the goal with a clause head.

A local modifying set is obtained from the unification conflicts in the mismatch. To each conflict is associated an elementary modifying set, and it is argued in (7) that taking as the local modifying set any one of the elementary modifying sets is sufficient to maintain completeness; it is also pointed out which is the best elementary modifying set choice.

An elementary modifying set is the set of modifying goals for variables whose links to non-variable terms were followed in accessing the two conflicting constant symbols.

The modifying goals for a variable linked to a non-variable term are goals whose matches produced bindings part of that link. (More on this later.)

A further refinement in this case is achieved by discarding from any modifying set, because redundant, all goals which are ancestors of another goal in the set. In fact, all ancestors are eventually backtracked to when their descendants fail.

Next we deal with one more important refinement.

Goal determinism - We say a goal is strongly-deterministic if some constant symbols in its arguments only allow the goal to possibly match one clause, and those symbols cannot be replaced. (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. Accordingly, the modifying goal for a variable bound in a strongly-deterministic match is not the matching goal but its most recent non-strongly-deterministic ancestor, called its avoiding goal.

Naturally, whenever a goal fails, the avoiding goal replaces the parent as a candidate for backtracking.

Representation of dependencies

In this section we present one solution to the problem of storing the information to be used by an interpreter for identifying modifying goals.

DEC-10 Prolog syntax will be used for terms, ie. names beginning with an upper-case letter will denote variables, otherwise constants, functors or predicates.

Terms are transformed to contain information on their own binding dependencies, and are unified through a special unification algorithm.

Let us sum up what dependencies must be remembered:

1) Because ancestors of failed goals are always backtracking candidates, the goal dependencies created by simple transmission of terms up and/or down the derivation tree (through chains of ancestors possibly linked by coomon variables at brother goals) need not be noted explicitly.

2) A goal in whose match a goal variable becomes instantiated must be kept as a modifying goal for that variable.

3) A goal in whose match two uninstantiated goal variables become bound together (through a double occurrence of a variable in the clause head) must be kept as a modifying goal for the two variables.

4) Strongly-deterministic goals do not create dependencies. Accordingly, should the goal in cases 2) and 3) be strongly-deterministic its avoiding goal, rather than itself, should be kept as a modifying goal.

Thus, only non-strongly-deterministic goals are modifying goals. In order to be referenced, they are numbered from 1 onwards, in the order they are activated. During backtracking this numbering is undone.

To convey dependencies, any binding of a variable X in the goal is represented in the form

TX : DX

DX is the dependency tag of X.

TX is the term bound to the variable, possibly comprising other bindings of variables in this form.

The dependency tag DX is a list

IX . BX

IX, the instantiation variable of X, shows the type of bindings performed on X:

IX = d(N) if X was directly instantiated (that is, without any intermediate binding to another goal variable) during the match of a goal G. N is the number of G, or that of G's avoiding goal in case G is strongly-deterministic.

IX = i if X was indirectly instantiated (that is, through an intermediate binding to another goal variable).

IX remains free while X is free or bound to variables alone.

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

b(N,D)

where N is the number of the goal G in whose match the binding is done, or that of G's avoiding goal in case G is strongly-deterministic.

If Y, bound to TY:DY, is instantiated, then D becomes DY. If Y is uninstantiated then D becomes DY1, and DY is updated to include the element b(N,DX1), expressing the dependency of Y on X. DY1 is the result of deleting b(N,DX1) from DY, and similarly DX1 is obtained from DX by excluding b(N,DY1) from it. Thus circularity of reference is prevented.

What happens is that when several goal variables get bound together through several explicit bindings among them, the graph of those bindings is a tree-like structure with unspecified root. The dependency tags we build are, for each variable, a representation of that tree having the variable as the root.

All this is best seen with an example. Suppose we have the goals and clauses:

```
             1          2         3                          p(V,V) <-

      <- p(X,Y) , p(Y,Z) , q(Y)                              q(a) <-


      After goal 1:

X = T : IX.b(1,IY.BY1).BX1                               X --1-- Y
Y = T : IY.b(1,IX.BX1).BY1

      After goal 2:

X = T : IX.b(1,IY.b(2,IZ.BZ1).BY2).BX1                   X --1-- Y --2-- Z
Y = T : IY.b(1,IX.BX1).b(2,IZ.BZ1).BY2
Z = T : IZ.b(2,IY.b(1,IX.BX1).BY2).BZ1

      After goal 3:

X = a : i.b(1,d(3).b(2,i.nil).nil).nil                   X --1-- Y --2-- Z
Y = a : i.b(1,i.nil).b(2,d(3).nil).nil                           :
Z = a : d(3).b(2,i.b(1,i.nil).nil).nil                           3
                                                                 :
                                                                 a
```

From these dependency tags the exact chain of goals through which each variable became instantiated can be obtained - they are its modifying goals.

Alternative solutions

If upon successful execution of a goal alternative solutions are sought, backtracking must take place. The selective backtracking mechanism views the process of backtracking as the need to explore the search space relevently, not as the need to explore it 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 modify the previous solution, ie. the arguments of the solved top goal.

Just picture replacing <- goal(X) by something like <- goal(X) , user_satisfied(X) .

Now, the goals where the arguments might be modified are precisely all the arguments' modifying goals.

Thus, after forgetting any subsisting legacy sets, all the modifying goals for the arguments of the top goal are taken as the backtracking candidates and backtracking is re-instated.

Simplifications

If a combination of some or all simplifications described next is used, selective backtracking can be incorporated as a standard facility without undue overhead.

1) Obtaining the modifying goals.

Rather than analysing the failed matches of G with clause heads, one may simply take all the modifying goals of all arguments of a failed G, irrespective of whether they entered in some conflict.

Alternatively, for every G one may take only the modifying goals of the first conflict encountered in each mismatch and remember them for eventual use.

Optional for the first alternative is letting the user state which arguments alone can give rise to failure. (This is a refinement of 'mode declarations'.)

2) Storing the modifying goals.

Only one slot is needed for each goal variable to store its modifying goals if the next simplification is made. When two free goal variables are bound, store nothing in their slots but mark the goal for backtracking, irrespective of whether it belongs to the modifying set of some future conflict. In case one of the variables is already instantiated, besides marking the goal for backtracking, also copy its slot into the slot of the free variable. In all other cases where a goal variable gets instantiated, store in its slot the goal number. Do nothing otherwise.

3) Determining strong-determinism.

It is irrealistic to expect the system to evaluate strong-determinism unaided and still be efficient. More expediently, the conditions under which a goal is strongly-deterministic are easily supplied by the user, eg.

```
strong_det( append(X,Y,Z) ) <- nonvar(X)
```

In our latest implementation, written in Prolog itself as the others had been (5)(7), we use 3) and the second alternative of 1), but not 2). For storing the modifying goals we use the representation shown before. A low level implementation is needed though to make the selective backtracking mechanisms presented truly competitive.

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) Bruynooghe,M. ; Pereira,L.M.
Revision of top-down logical reasoning
through intelligent backtracking
(This volume).

(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 n. 87
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. ; 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.