

An interpreting algorithm for Prolog programs[†]

M. H. van Emden, University of Waterloo

The basic algorithm for interpreting Prolog programs was invented by A. Colmerauer and Ph. Roussel in 1972 when the first implementation was built in Marseille. We are not aware of a published description of the complete basic algorithm, although there is no scarcity of publications dealing with Prolog implementation. Some of these contain expositions of structure sharing (Bruynooghe, 1982; Warren, 1977). Other papers (Roberts, 1977) treat memory management or compare the relative advantages of structure sharing versus copying (Bruynooghe, 1982; Mellish, 1982). Only in Bruynooghe (1982) do we find a fragment of an algorithm. In 1981, Colmerauer lectured on an interpreting algorithm in 'clock' form. We lack details on this and look forward for an opportunity to compare it with the algorithm presented here.

The purpose of this paper is to present and to justify a simple algorithm for interpreting Prolog programs. We believe that programming environments such as Unix will make it relatively easy to implement reasonably efficient Prolog systems. We expect many individuals and groups to take advantage of this opportunity because the wide variety of potential Prolog applications (expert systems, knowledge bases, natural-language processing, symbolic computation) will encourage building specialised implementations.

We do not expect the algorithm to have any novel features. It is hard to be sure, because of the machine-oriented form in which currently-used algorithms exist. The components of the 'state' and the 'stack frame' are from Bruynooghe (1982). The only novelty we believe this paper to have is the human-oriented form of the algorithm and its derivation from a mathematical description of the SLD theorem-prover.

A SEARCH ALGORITHM FOR TREES WITH UNSPECIFIED NODES

Our starting point is a depth-first, left-to-right algorithm for traversing a tree in search of a terminal node having some given property P. At this point we do not make any assumptions about the nature of the nodes of the tree.

We do have to make an assumption about the way the relation is specified between a node and its descendants and also about the specification of the

[†] Reprinted from the proceedings of the First International Logic Programming Conference at the University of Marseille, 1982.

order among descendants. We do this in the style of 'data abstraction': we posit the availability of a procedure $\text{son}(x,y)$ which, for a given node x , exhibits (after being initialised) the following behaviour on successive calls. If x has n sons, then $\text{son}(x,y)$ will return TRUE the first n times it is called and FALSE forever after. The first n times y is assigned successively the n sons. This procedure may therefore be called a *generator* of sons. Similarly, $\text{father}(x,y)$ returns, for given x , FALSE if node x is the root and TRUE otherwise. In the latter case y is assigned the father node of x . Note that we have avoided any assumptions about data structures representing trees; hence the earlier reference to data abstraction.

We pursue an assertion-based (van Emden, 1974) development of the algorithm. The minimum set of variables required seems to include one indicating how far the tree has been searched. We call it 'cn', for 'current node'. Here 'assertion' is used in Floyd's sense: it is an assertion about the values that variables have at a particular point in the execution of an imperative program.

The following assertion is useful because it applies to the initial situation as a special case, where no part of the tree has been searched.

A: no terminal node to the left of the current node has property P

The assertion has been labelled 'A'. 'To the left of x ' means: occurring in a subtree rooted in an older sibling of x or in an older sibling of an ancestor of x .

B: all terminal nodes to the left of all nodes still in the son generator of the current node do not have property P

C: B holds; furthermore, the son generator of the current node is empty

The following program may be verified with respect to the assertions described. This verification is in the sense of Floyd: it means that, whenever execution reaches a label, the corresponding assertion, as described above, holds.

For example, if, in the initial situation where no part of the tree has been searched, we set the current node to the root, then assertion A holds. This implication is expressed in the program as:

```
cn := root
A:
```

Another example: if C holds and if the current node has x as father, then, after making x the current node, B holds. Again, this implication is expressed by the program; this time by

```
C: if father(cn, x)
    then cn := x; goto B
```

The complete program is listed in Fig. 1.

```

        cn := root
A:  if P (cn)
    then halt with success
    else initialise son ( ) for cn;
    goto B
    fi
B:  if son (cn, x)
    then {x is next son of cn}
        cn := x; goto A
    else {all sons of cn have
        been tried
        }
        goto C
    fi
C:  if father (cn, x)
    then {x is the father of cn}
        cn := x; goto B
    else {cn is the root}
        halt with failure
    fi

```

Fig. 1 – The basic tree-search algorithm.

Note that we retain some instructions ‘goto B’ and ‘goto C’ which are not needed for a computer executing the algorithm. But these are necessary if we are to be able to read the program as a system of logical implications true about the set of computations to be performed by the algorithm.

An efficient way to implement the ‘father’ operation is to keep on a stack the sequence of nodes from the root up to and including the current node. The variable *cn* is an alias for the top of the stack and it holds the current node. The father of the current node is obtained by popping the stack. If we want to use this method, then the current node has to be pushed on the stack when its son becomes the current node (see Fig. 2).

```

initialise the stack at empty
cn := root
push cn
A: if P (cn)
  then halt with success
  else initialise son( ) for cn;
      goto B
fi
B: if son (cn, x)
  then {x is the next son of cn}
      cn := x; push cn; goto A
  else {all sons of cn have
        been tried
      }
      pop stack; goto C
fi
C: if stack empty
  then halt with failure
  else cn becomes top of stack
      goto B
fi

```

Fig. 2 – The stack version of the ABC algorithm.

THE THEOREM PROVER ON WHICH PROLOG IS BASED

A Prolog program is a set of definite clauses, i.e. Horn clauses which are not goal statements. For a given program P and goal statement G , a *derivation* is a sequence $G(0), G(1), \dots$ of goal statements such that $G(0) = G$ and $G(i+1)$ is obtained from $G(i)$ by resolution between a clause of P and $G(i)$.

Notice that often a given derivation $G(0), \dots, G(n)$ can be extended in several different ways. This is usually true even if we restrict (as we will do) resolution to unification between the conclusion of the definite clause and one particular ('selected') goal of $G(n)$. The set of all derivations thus restricted and starting at G can be arranged in the form of a tree of goal statements having the property that the set of all paths from the root is exactly the set of all possible derivations using most general unifications. This tree is called the *search tree*. See Apt and van Emden (1984) and Clark (1984) for correctness and completeness properties of the search tree.

The next step in the development of the interpreter results from the following observation;

An interpreter for a Prolog program P and initial goal statement G is obtained by applying the ABC algorithm to the search tree for P having G as root.

The resulting interpreter is shown in Fig. 3.

```

        initialise the stack at empty
        cn := initial goal statement
        push cn
A:   if cn is empty goal statement
      then halt with success
      else initialise son ( ) for cn
           goto B
      fi
B:   if son (cn, x)
      then cn := x; push cn; goto A
      else pop stack; goto C
      fi
C:   if stack empty
      then halt with failure
      else cn becomes top of stack
           goto B
      fi
where son (cn, x) is defined as
      while nextclause (cn, y)
      do if head of y unifies with
           selected goal of cn
          then
              x := goal statement obtained
                  by resolving y with cn
              return (true)
          fi
      od
      return (false)

```

Fig. 3 – The stack-based ABC algorithm applied to a search tree.

STRUCTURE SHARING AND PROOF TREES

The ABC algorithm for searching a tree has to store the sequence of nodes between the root and the current node. When the tree is the search tree described in the previous section, this sequence is typically enormously redundant. The reason is that in the search tree every node contains a complete description of a global statement, without reference to any other data. Yet, given any node in a search tree (i.e. any goal statement), each of its sons can be specified by means of a description of the resolution that generated the son concerned. This description is typically more compact than the full goal statement.

A *proof tree* is a data structure which stores in a non-redundant way the path in a search tree between the root and a current node. Consider for example the program

$$P \leftarrow Q \& R \& S \& T$$

$$Q \leftarrow U$$

$$U \leftarrow V$$

$$V$$

With $\leftarrow P$ as initial goal statement, the search tree for this program is

$$\begin{array}{c} \leftarrow P \\ | \\ \leftarrow Q \& R \& S \& T \\ | \\ \leftarrow U \& R \& S \& T \\ | \\ \leftarrow V \& R \& S \& T \\ | \\ \leftarrow R \& S \& T \end{array}$$

The proof tree for the path in this search tree is as shown in Fig. 4.

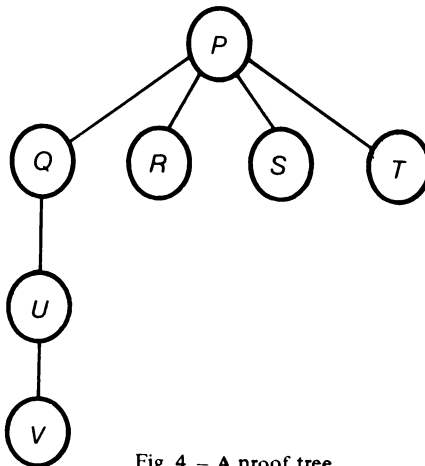


Fig. 4 – A proof tree.

We may assume without loss of generality that the initial goal statement (the root of the search tree) has a single goal with the distinguished predicate 'goal'. This ensures that the proof tree is indeed a tree rather than a forest. The proof tree for a path from the root of a search tree is defined as follows.

Let $p = g(0), g(1), \dots, g(n-1), g(n)$ be a path in a search tree, where $g(0)$ is the root. If $n = 0$, then the proof tree for p contains only the root and it is $g(0)$. Let $n > 0$, let G be the selected goal of $g(n-1)$, and let C be the clause that was resolved with $g(n-1)$ (unifying the head of C with G) to give $g(n)$. Let us call T the proof tree of $g(0), \dots, g(n-1)$. The proof tree of $g(0), \dots, g(n)$ is obtained from T by attaching as sons to G (which must be a terminal node of T) the goals of the right-hand side of C and by applying throughout T and these goals the most general unifier of G and the head of C .

The proof tree avoids redundantly repeating goals from one goal statement to the next. Ultimately, storage of goals in proof trees will be avoided altogether. To every nonterminal node of a proof tree there corresponds a unification. The basic idea of *structure sharing* (Boyer and Moore, 1972) is to represent a proof tree by storing only records of these unifications and to refer wherever possible to the program for the structure of clauses, goals, and terms. To help explain this we show an example of what we call a *Ferguson diagram* for a proof tree (after its originator R. J. Ferguson (1977)). Such a diagram shows explicitly the unifications and the structures which are borrowed from the program. Fig. 5 shows the Ferguson diagram for the same example we used before.

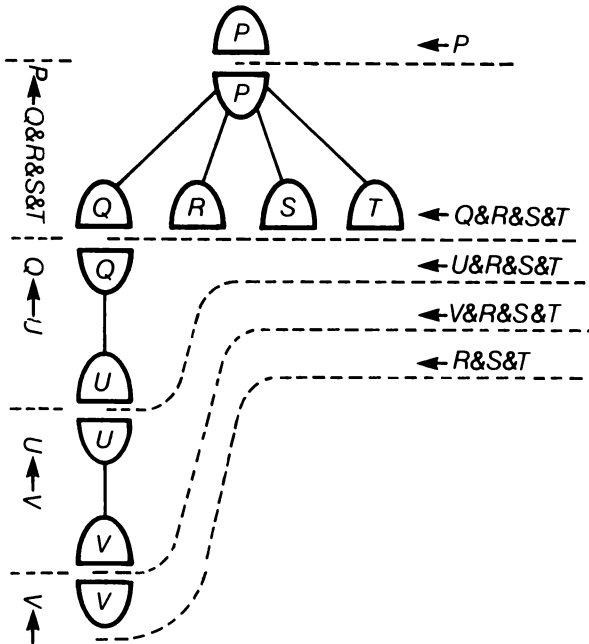


Fig. 5 — A Ferguson diagram.

Calls are upper half circles. Headings are lower half circles. A unification is represented by an upper half circle meeting a lower half circle. The graphic equivalent of a procedure connects its heading with the zero or more calls which constitute its body.

In general, a Ferguson diagram consists of an instance of the initial goal statement together with instances of program procedures. Structure sharing avoids redundancy in the representation of two different instances of the same procedure by representing each procedure by a pair consisting of a pointer to the procedure in the program and an *environment*, that is, a vector of substituting terms, one for each variable in the procedure. Each substituting term, if composite, is itself a pair: the first component points to the occurrence of the term in the program of which the substituting term is an instance; the second points to an environment where substitutions for possibly occurring variables in the program term can be found.

Thus, in structure sharing there is a strict segregation in the information specifying an instance of procedure, call, or term. On the one hand there is the 'structure', obtained from the program; this is also called '(pure) code' or 'skeleton' (Warren, 1977) (lacking the 'flesh' of the substitution). On the other hand there are the substitutions, one value for each variable. For the representation of each of these one also uses structure sharing.

Consider for example the procedure

$$\text{sub1}(x, y) \leftarrow \text{app}(u, x, v) \ \& \ \text{app}(v, w, y)$$

See below for an explanation of the meaning of the predicates 'sub1' and 'app'. The environment of this procedure is a vector of 5 pairs, one each for the variables x, y, u, v, w .

We can now be more specific about the method for storing a proof tree. We store each unification in a 'frame'. Each frame records a unification (hence corresponds to a full circle of the proof tree). Each frame has the following components:

- CALL:** A pointer to the occurrence of the call in the code of which the call in the proof tree is an instance.
- FATHER:** A pointer to the environment where the substitutions for the variables in CALL may be found.
- PROC:** A pointer to the occurrence in the code of a procedure. The heading which participated in the unification is an instance of the heading of this procedure.
- ENV:** An environment for PROC.

The reasoning behind this is simple: a unification happens between two participants. One is determined by CALL and FATHER; the other by PROC and ENV. Notice that, by having PROC point to an entire procedure rather than just the heading involved in the unification, we have included in the representation not just the full circles of the proof tree, but also the upper half circles.

Fig. 5 is untypical, in that there are no variables in the clauses, so that no environments are needed to store substitutions. The Ferguson diagram in Fig. 6

illustrates the more general situation where there are four components of a frame recording a unification.

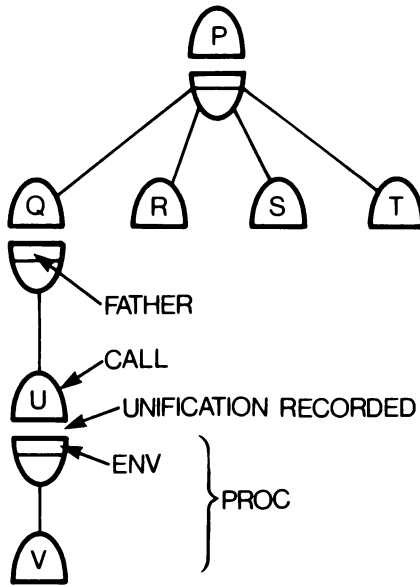


Fig. 6 – Ferguson diagram illustrating the components of a frame.

As an illustration we show the growth of the proof tree for the following program and initial goal statement.

```

1{app(9{nil},y,y)}
2{app(7{u.x},y,8{u.z}) ← app(x,y,z)}
3{subl(x,y) ← app(u,x,v) & app(v,w,y)}
4{goal(x,y)
  ← subl(5{10{a}.x},6{y.11{nil}})}
}
← goal(x,y)
    
```

Let 'app' mean 'append', let 'subl' mean 'sublist'. To obtain the Prolog program, remove all numerals and braces. A number refers to the expression enclosed by the matching pair of braces of which the opening brace immediately follows the numeral. For example, 10 refers to 'a', 5 refers to 'a.x'. These references are needed in the proof trees shown in Figs. 7–10.

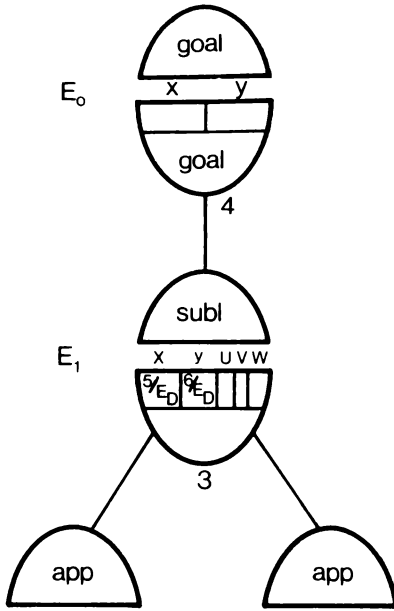


Fig. 7 – The proof tree in an early stage.

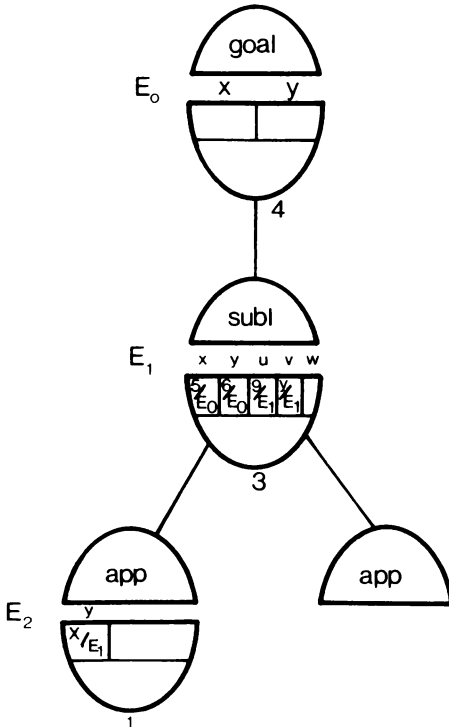


Fig. 8 – After one more unification.

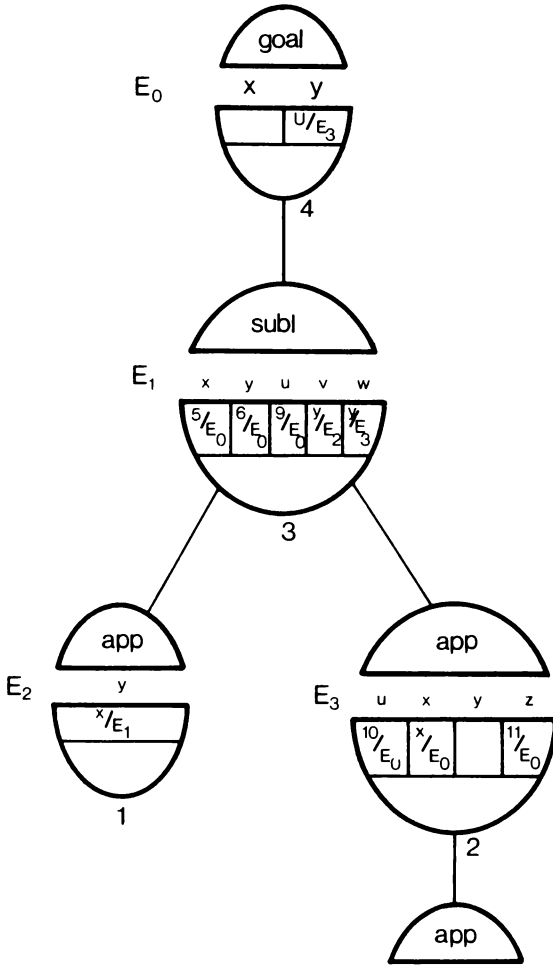


Fig. 9 -- After another unification.

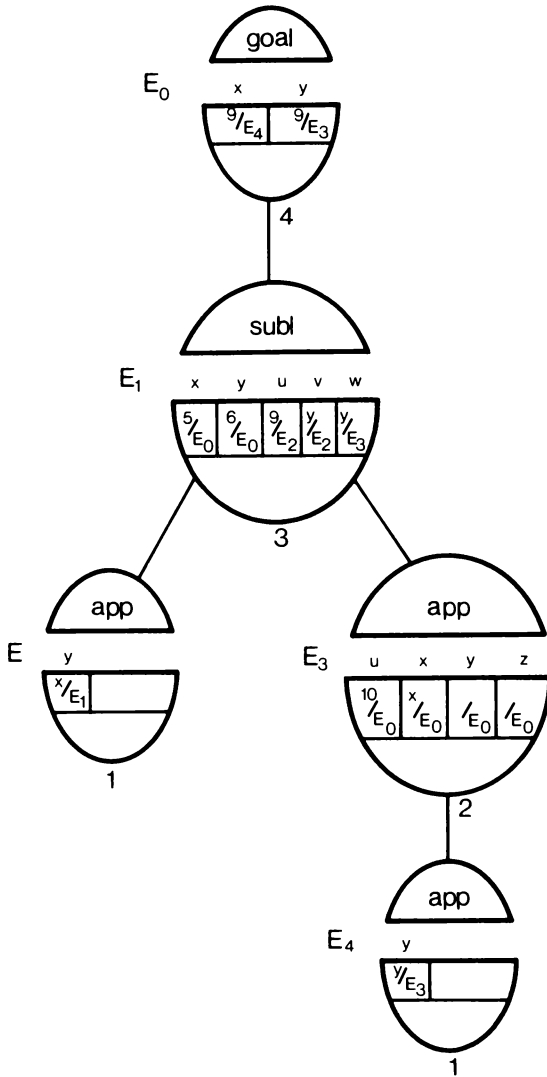


Fig. 10 -- Completed proof tree. Read off the answer: $x = \text{nil}$, $y = a$.

The numeral shown in the 'crotch' of a procedure refers to its code in the program. Entries in the environment are shown as p/e where p refers to an expression in the program and e is an environment elsewhere in the proof tree.

A USABLE INTERPRETING ALGORITHM

As we observed before, the proof tree is a representation of a path from the root in the search tree. Extending the path in the search tree by one node (i.e. one goal statement) extends the corresponding proof tree by one full circle and zero or more upper half circles. To avoid becoming confused between nodes in the search tree and nodes in the proof tree, one should realise that to one node in the search tree (i.e. one goal statement) corresponds the frontier of upper half circles in the proof tree. We claimed that the proof tree not only represents that single goal statement but also the entire sequence of predecessors. With the information we included in the frames so far, this is only true if we disregard the effects of substitutions. In this section, we will include additional information in each frame of a proof tree for a path so that from it one can reconstruct proof trees corresponding to initial segments of that path.

Let us consider again the ABC algorithm using a stack to represent the sequence S of nodes of the search tree (i.e. goal statements) between the root and the current node. The operations the algorithm performs on this stack are to push (resulting, say, in S1) and to pop (resulting, say, in S2). Given that S is a proof tree we ask for a convenient storage representation of it which allows us to obtain efficiently the proof trees representing S1 and S2. The answer is that S should be a stack of frames and that S1 is obtained from S by a push operation and S2 by a pop operation. It is important not to confuse the stack of goal statements in the previous version of the interpreting algorithm with the stack of frames in the following version. Now that we are committed to the stack representation of proof trees, we will refer to its frames as *stack frames*.

Steps in the execution of a Prolog program are most naturally measured by extensions to the proof tree where one of the calls (upper half circles of the proof tree) is selected and made into a full circle by attaching a procedure to it. This adds one internal node, hence one stack frame to the proof tree. Most Prologs always select the leftmost call. Ferguson (1977) observed that, with this selection, the order of the stack frames in the stack is the one obtained by preorder traversal of the full circles of the proof tree (Knuth, 1968).

The interpreter acts on two categories of data. One is the code, the internal representation of the program. This does not change during execution, at least not in pure Prolog (i.e. in the absence of extra-logical facilities for adding or deleting clauses). The other category of data does change during execution. It comprises the stack (representing the proof tree) which changes both in size and content. It also comprises what is called the state (Bruynooghe, 1982). This is constant in size and variable in content. It plays the role of the 'current node' in the earlier versions of the ABC algorithm.

Let us now determine the components of the stack frame, which has to store the information concerning a unification. This information was already identified in the 'frame' of the previous section. The stack frame also has to

contain data which allow the proof tree to be restored to the state in which it was before the unification. Hence the stack frame is an extension of the frame, with the following components:

- CALL:** A pointer to the occurrence of the call in the code of which the call in the proof tree is an instance.
- FATHER:** A pointer to the stack frame of which the procedure contains the call of CALL.
- PROC:** A pointer to the occurrence of a procedure in the code. The heading which participated in the unification is an instance of the heading of this procedure.
- ENV:** The environment for PROC.
- RESET:** The reset list, i.e. a list of variables in the proof tree that obtained substitutions as a result of the unification.
- NEXT-CLAUSE:** A generator for clauses which are candidates for attempts at unification with the call of CALL.

The RESET component of the stack frame is obviously necessary to restore the previous state. FATHER is changed from a pointer to an environment to a pointer to the entire stack frame containing that environment. In this way, one can still get the environment, but one also has the remaining information which is required for restoring the previous state. Finally, NEXT-CLAUSE is included to facilitate the implementation of the son generator.

Let us now discuss the components of the state. A point in the ABC algorithm where all components of the state enter into play is where a new son (in the search-tree sense) has just been found. When the tree is known to be a resolution search tree, the corresponding point in the algorithm is where a unification has just been successfully completed. The relevant part of the search tree is shown in Fig. 11.

Another change is in the time when the stack is popped. In the previous version it seemed most natural to do it right after failure of unification. Now

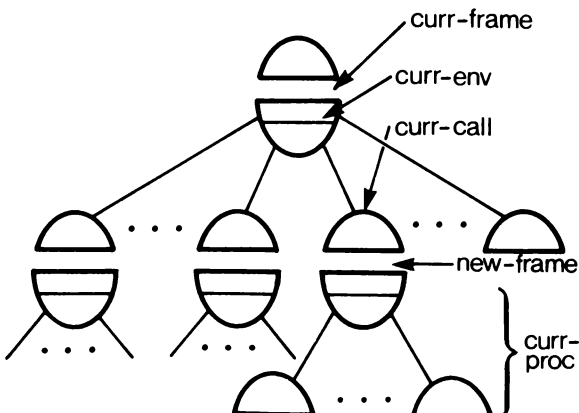


Fig. 11 -- All components of the state in use. Not shown is the next-clause component which is the generator of clauses which are candidates for matching with curr-call.

we have to worry about the details of installing the appropriate state. The required information comes from the stack frame which must be popped. In the next version, popping the stack is delayed so as to have time to copy information from the top frame before it disappears.

The next version of the algorithm is obtained from the previous one by taking into account the consequences of our chosen representation of the path in the search tree by a stack of stack frames and of the current node by the state. The following notes should clarify the transition to the next version of the algorithm.

Instead of testing whether the current node *cn* is the empty goal statement, we call a boolean procedure 'select' which returns FALSE if the proof tree contains no unified call (i.e. the goal statement is empty) and TRUE otherwise. In the latter case, a pointer to such a call is returned in the argument. At label A of the program not every component of the state necessarily has a meaningful value. Those that do are indicated in Fig. 12.

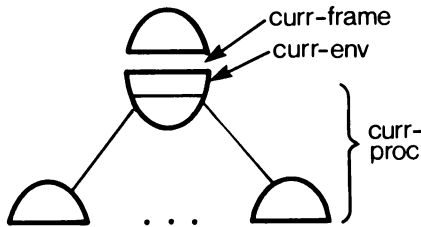


Fig. 12 – Components of the state at label A.

As before, 'son' is the generator of sons in the search tree. In the current context, finding a son translates to finding a procedure whose head matches the selected call ('curr-call'), performing the unification, initialising a clause generator, and returning a newly created frame ('new-frame') recording the unification. 'Son' also returns in *curr-proc* the procedure it found matching *curr-call*. Conceptually, the son (in the search-tree sense) is not just the new frame, but the entire goal statement implicit in the proof tree which is now represented by the stack together with *new-frame*. Just after 'son' has returned TRUE, the state is as indicated in Fig. 11. Just after 'son' has returned FALSE, the state is as indicated in Fig. 13.

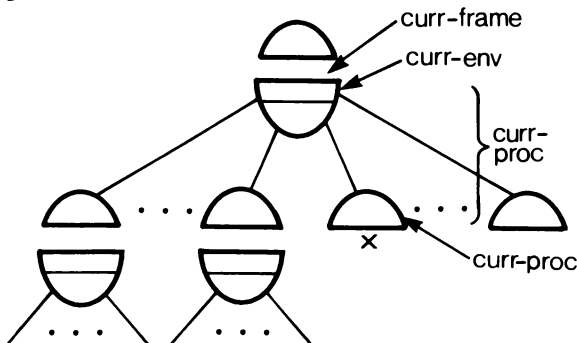


Fig. 13 The state just after failure to find a son (in the search-tree sense).

```

initialise stack at empty
curr-proc
:= initial goal statement
   {disguised as procedure
   goal ← ...
   }
curr-env := create-env (curr-proc)
curr-frame
:= create-frame (curr-env)
push curr-frame
A: if select (curr-call)
   then
       {the current goal statement is non-
       empty; curr-call is the selected
       goal
       }
       next-clause := create-cg (curr-call)
       goto B
   else halt with success
   fi
B: If son (next-clause, curr-call, curr-
   env, new-frame, curr-proc
   )
   then curr-frame := new-frame
       push curr-frame
       curr-env := ENV (new-frame)
       goto A
   else goto C
   fi
C: if stack has only one frame
   then halt with failure
   else top-frame := top of stack
       curr-frame := FATHER (top-frame)
       curr-env := ENV (curr-frame)
       curr-call := CALL (top-frame)
       curr-proc := PROC (curr-frame)
       next-clause
       := NEXT-CLAUSE (top-frame)
       undo bindings of RESET (top-frame)
       pop stack; goto B
   fi

```

Fig. 14 ... The interpreter using proof trees implemented as a stack.


```

function
  son (next-clause, curr-call, curr-env
      , new-frame, curr-proc
      ): boolean
  while next-clause (curr-proc)
  do if unifies (curr-call, curr-env
                , curr-proc
                , new-env, res-list
                )
      then create new-frame with
          CALL = curr-call
          PROC = curr-proc
          FATHER = curr-frame
          ENV = new-env
          RESET = res-list
          NEXT-CLAUSE = next-clause
          return (TRUE)
      fi
  od return (FALSE)

function
  select (curr-call) : boolean
  curr-call := first-call (curr-proc)
  while curr-call = nil
  do curr-frame
      := FATHER (curr-frame)
      if curr-frame = nil
      then return (FALSE)
      else curr-call
          := next-call (CALL (curr-frame))
      fi
  od
  curr-proc := PROC (curr-frame)
  curr-env := ENV (curr-frame)
  return (TRUE)

```

Fig. 15 – Auxiliary functions for the interpreter.

At X no matching procedure head was found. This means that in the search tree the father of the current node becomes the current node. To effect this, the proof tree has to be restored to the state in which it was before the unification recorded in curr-frame was performed.

The function 'unifies' unifies curr-call (with curr-env as environment) with the head of curr-proc. It creates the environment new-env for curr-proc and may place bindings in it. 'Unifies' also creates the reset list and makes it accessible in res-list (Fig. 14).

We need a place to store the environment for the initial goal statement. Such a place is obtained as the environment for a fictitious procedure

goal ← (initial goal statement)

The very first frame ever to be pushed on the stack contains this environment and records the unification between the fictitious call 'goal' and the above fictitious procedure. A consequence for the ABC algorithm is now that at label C the appropriate test is not for the empty stack but for one containing only a single frame (Fig. 15).

ACKNOWLEDGEMENTS

Many thanks to Ron Ferguson for some very helpful discussions and to Paul Ng for his willingness to use the algorithm in this form as basis for his implementation work. Maurice Bruynooghe kindly pointed out various obscurities in an earlier version. The National Science and Engineering Research Council financed supporting facilities.

REFERENCES

- Apt, K. R. and van Emden, M. H., (1984), Contributions to the theory of logic programming, *Journal of the ACM*.
- Boyer, R. S. and Moore, J., (1972), The sharing of structure in theorem-proving programs, *Machine Intelligence 7*, B. Meltzer, and D. Michie, (eds.), Edinburgh University Press.
- Bruynooghe, M., (1982), The memory management of Prolog implementations, *Logic Programming*, K. L. Clark and S.-Å. Tärnlund, (eds.), Academic Press.
- Clark, K. L., (1984), *Predicate Logic as a Computational Formalism*, Springer-Verlag.
- van Emden, M. H., (1979), Programming with verification conditions, *IEEE Transactions on Software Engineering*, SE-5, 148-159.
- Ferguson, R. J., (1977), An Implementation of Prolog in C. Master's Thesis, Department of Computer Science, University of Waterloo.
- Knuth, D. E., (1978), *The Art of Programming*, I, Addison-Wesley.
- Mellish, C. S., (1982), An alternative to structure sharing in the implementation of a Prolog interpreter, *Logic Programming*, K. L. Clark and S.-Å. Tärnlund, (eds.), Academic Press.
- Roberts, G. M., (1977), An Implementation of Prolog, Master's Thesis, Department of Computer Science, University of Waterloo.
- Warren, D. H. D., (1977), *Implementing Prolog: Compiling Predicate Logic Programs*, Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh.