JUN 18 1985

UNIVERSIDAD DE LAS AMÉRICAS, A.C.

May 3, 1985.

45 ANIVERSARIO 1940 1985

DONNA McCRACKEN
TECHNICAL REPORTS
DEPT. OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
WATERLOO, ONTARIO, CANADA
N2L 3G1

Dear Miss McCracken:

Thank you very much for the copy of the thesis of Grant Roberts that you were kind enough to send us. Please find enclosed a check for $ 9.00 to cover your costs.

Having received as well the list of technical reports, I believe that your document

CS-81-28 An Algorithm for Interpreting Prolog Programs
by M. van Einden

would also be very useful. If you could send us a copy of this report, as well, it will be appreciated. As the list doesn't include a price I do not include payment, but we will pay whatever costs you might incur.

Thank you very much again for your help.

Yours truly,

[illegible]

RICARDO VILLAFAÑA F. M.S.
Chairman of Electronic &
Computer System Engineering
Department.

June 18th
cheque # 2382 for
$9.00 rec'd
Report mail'd

D

APARTADO POSTAL No. 100 SANTA CATARINA MARTIR 72820 PUEBLA MEXICO

University of Waterloo

Waterloo, Ontario, Canada
N2L 3G1

Faculty of Mathematics
Department of Computer Science
519/885-1211

Telex Number
069-55259

INVOICE

February 27, 1985

Ing. Ricardo Villafana
Chairman
Electronic & Computer System
Engineering Department
Universidad de las Americas, a.c.
Apartado Postal 100
Santa Catarina Martir
Mexico

Dear Sir:

Thank you for your letter dated February 4, 1985 requesting a copy of the thesis by G.M. Roberts.

A Copy of this thesis is enclosed. THe cost of copying the thesis is $9.00 Canadian. It would be appreciated if you would make your payment payable to the Department of Computer Science, University of Waterloo, And forward it to my attention.

I have also attached a listing of other Prolog reports and theses and a current technical report list. They may be of some interest to your department.

Thank you for your interest in our department. If I can be of further assistance, please do not hesitate to contact this office.

Yours truly

Donna McCracken
Technical Reports
Dept. of Computer Science

/dm

FEB 26 1985 FEB 25 1985

UNIVERSIDAD DE LAS AMÉRICAS, A. C.

February 4, 1985

Head of:

Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada.

Dear Sir or Madam:

We are seniors endering our final semester of studies of Computer Systems Engineering at the University of the Americas, Puebla, Mexico. Having heard of the work you are doing we would like to solicit your help.

As part of the requirements of our degree program, we must complete a research proyect and we have chosen to implement a "PROLOG" interpreter. Begining our investigation, we have come across bibliographic references indicating that in 1977, Mr. G M Roberts of your department wrote on this topic, under the title, "An Implementation of PROLOG". For us, any information that you could provide will be of grat help, and is possible, a copy of Mr. Roberts' paper would be invaluable. We gratefully remit any expenses you might incur.

Anticipating that you response will be favorable, we thank you sincerely for your help and considerations.

Respectfully

[illegible]
Fernando Castillo Reyes

[illegible]
José Cortez Gómez

[illegible]
Ing. Ricardo Villafaña
Chairman of Electronic and Computer System Engineering department.

An Algorithm for Interpreting PROLOG Programs

by

M.H. van Emden

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Research Report CS-81-28

September 1981

ABSTRACT

We describe in pseudocode the main routine of a Prolog interpreter. The algorithm is developed in several steps. Initially a search algorithm for trees is described. After a review of the Prolog theorem prover, the tree-search algorithm is applied to the search space of the theorem prover. Several inefficiencies of the result are then eliminated by the introduction of proof trees and structure sharing.

Introduction

There is no scarcity of publications dealing with Prolog implementation. Some of these contain expositions of structure sharing [3,12]. Other papers [11] treat memory management or compare the relative advantages of structure sharing versus copying [3,10]. In so far as these publications are concerned with interpretation rather than compilation, the algorithm used is taken for granted. Thus a prospective implementer has to unearth it from the assembler or FORTRAN code of an existing interpreter or has to reinvent an algorithm from scratch.

The present paper decribes the outline of a basic interpreter for Prolog programs. This should be useful in itself, as we expect that many more Prolog interpreters will be written. Moreover, we were able to derive the algorithm in a step-wise fashion from a basic tree-search algorithm which is easy to verify.

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 order among descendants. We do this in the style of "data abstraction": we posit the availability of a procedure son(x,y) which, for a given node x, exhibits (after being initialized) the following behaviour on successive calls. If x has n sons, then 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, 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-oriented [5] development of the algorithm. The minimum number 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 entire program is listed below:

```
          cn := root
     A:   if P(cn)
          then halt with success
          else initialize son() for cn; goto B
          fi

     B:   if son(cn,x)
          then {x is the 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
```

figure 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 to the current node. Then 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 figure 2.

```
        initialize the stack at empty
        cn := root
A:      if P(cn)
        then halt with success
        else initialize son() for cn; goto B
        fi

B:      if son(cn,x)
        then {x is the next son of cn}
             push cn; cn := x; goto A
        else {all sons of cn have been tried}
             goto C
        fi

C:      if stack nonempty
        then pop stack into cn; goto B
        else {cn is the root}
             halt with failure
        fi
```

figure 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), ... 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),...,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 [1,4] for correctness and completeness properties of the search tree.

The next step in the development of the interpreter is 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 figure 3.

```
      initialize the stack at empty
      cn := initial goal statement
A:    if cn is the empty goal statement
      then halt with success
      else initialize son() for cn; goto B
      fi

 B:   if son(cn,x)
      then push cn; cn := x; goto A
      else goto C
      fi

 C:   if stack nonempty
      then pop stack into cn; goto B
      else halt with failure
      fi

where son(cn,x) is defined as

      while nextclause(cn,y)
      do if head of y unifies with the selected
            goal of cn
         then x := goal statement obtained by
                   resolving y with cn
              return(true)
         fi
      od
      return(false)
```

figure 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 goal 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 small amount of data describing the resolution that generated the son concerned.

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 <- Q & R & S & T
Q <- U
U <- V
V
```

With <- P as initial goal statement, the search tree for this program is

```
<- P
    |
    |
<- Q & R & S & T
    |
    |
<- U & R & S & T
    |
    |
<- V & R & S & T
    |
    |
<- R & S & T
```

The proof tree for the path in this search tree is

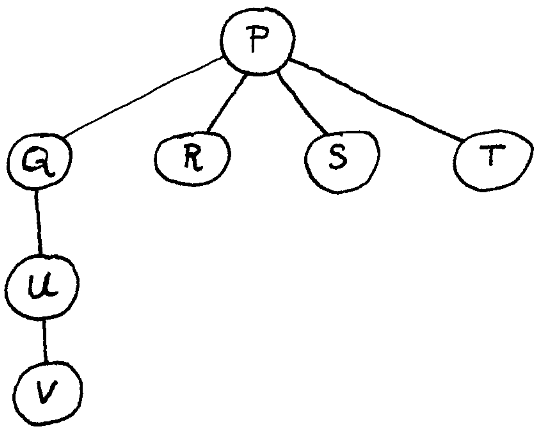

figure 4
A proof tree

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 has a single root. The proof tree for a path from the root of a search tree is defined as follows.

Let p = g(0),g(1),...,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),...,g(n-1). The proof tree for g(0),...,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 <u>structure sharing</u> [2] 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 <u>Ferguson diagram</u> for a proof tree (due to R.J. Ferguson [6]). Such a diagram shows explicitly the unifications and the structures which are borrowed from the program. Figure 5 shows the Ferguson diagram for the same example we used before.

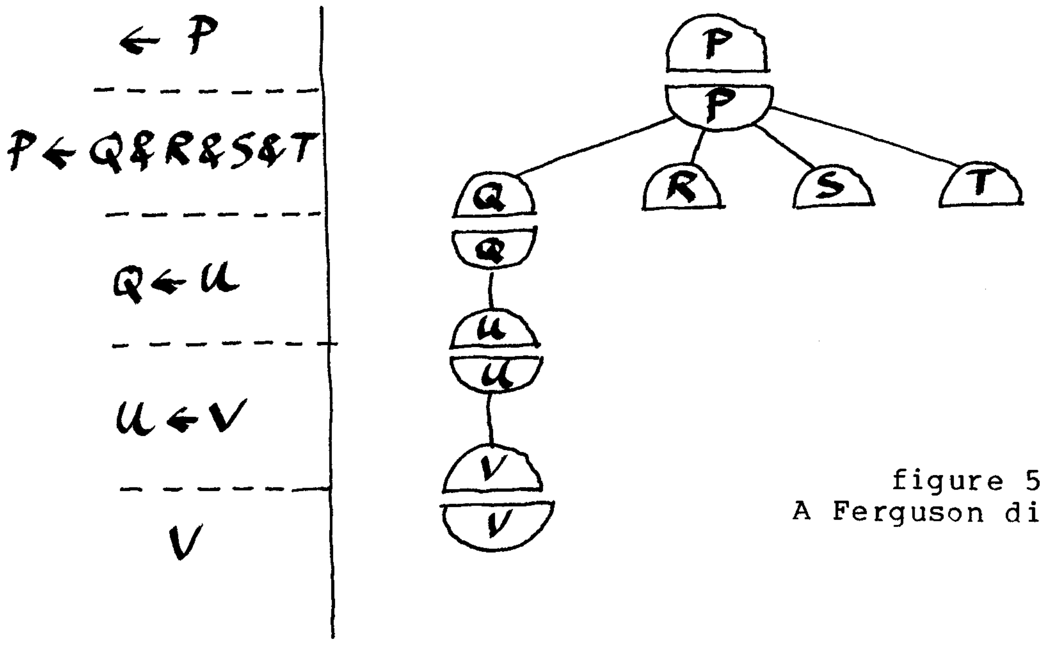

figure 5
A Ferguson diagram

Calls are upper half circles. Headings are lower half circles. A unification is represented by an upper half circle meeting a lower half circle. A procedure connects a heading with zero or more calls.

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" (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

```
subl(x,y) <- app(u,x,v) & app(v,w,y)
```

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 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.

The Ferguson diagram in figure 6 illustrates the four components of a frame recording just one unification U.

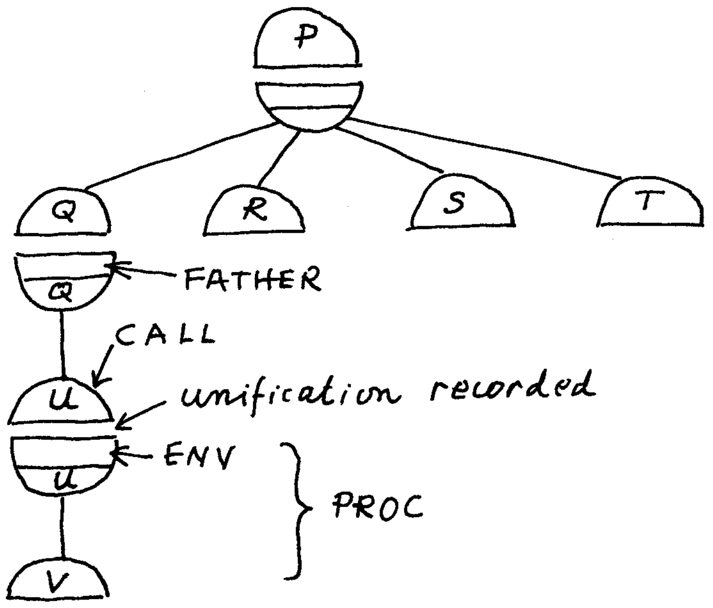

figure 6
Ferguson diagram illustrating the components of a frame

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 below.

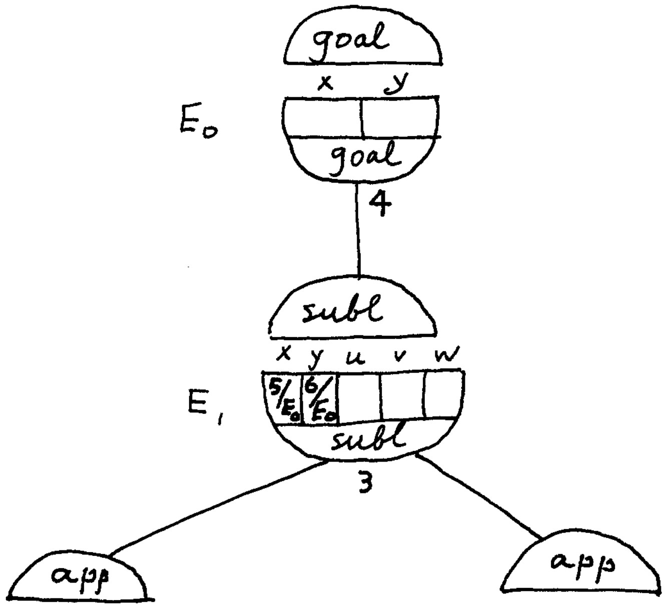

figure 7
The proof tree in an early stage

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.

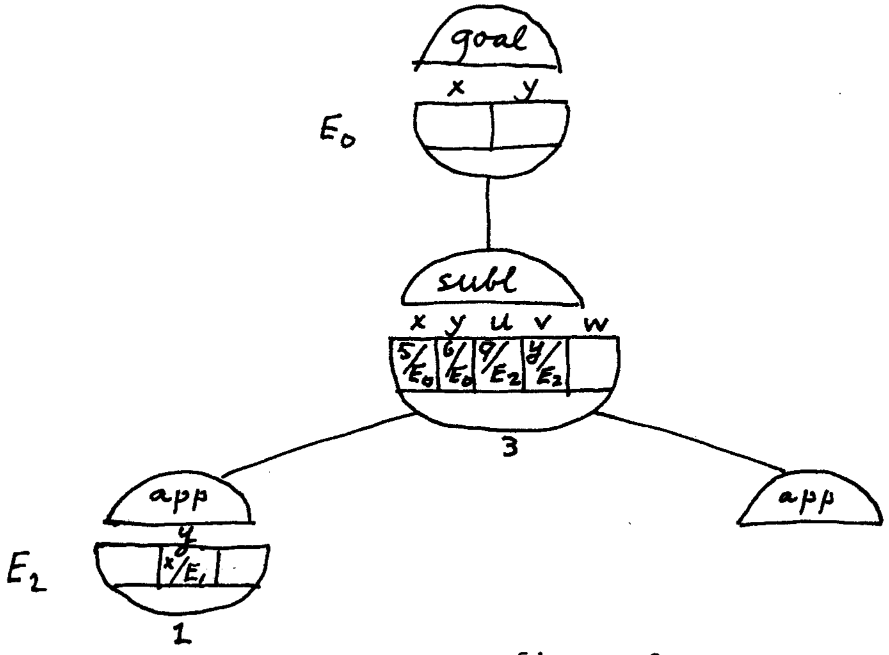

figure 8
After one more unification

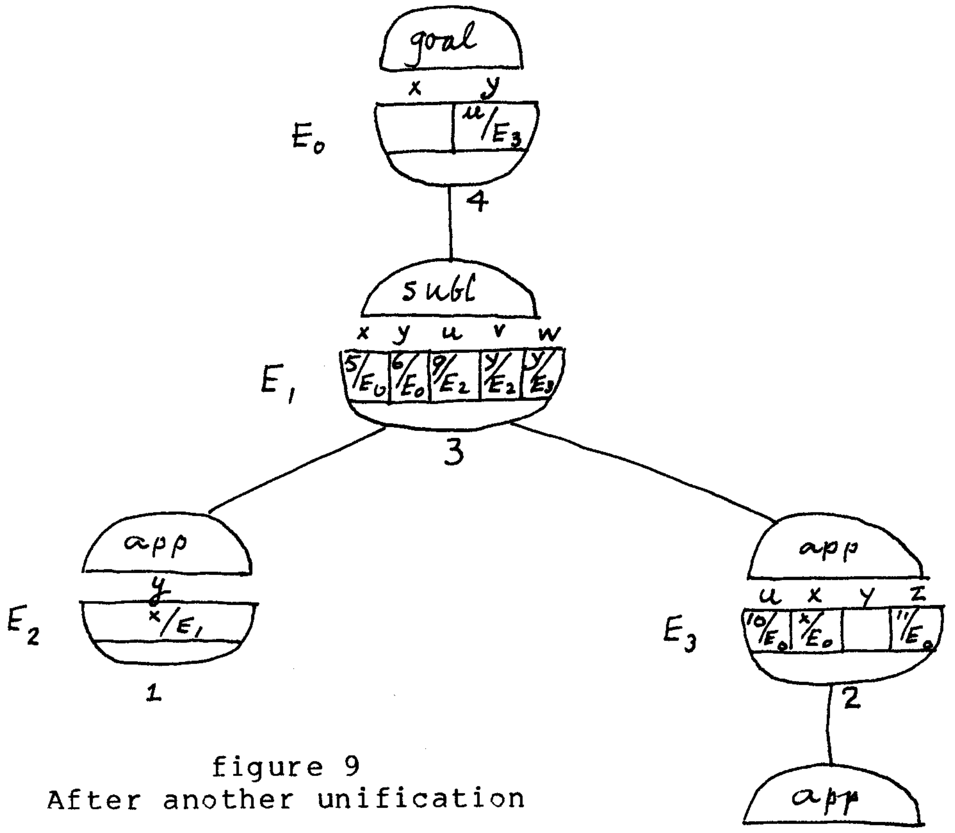

figure 9
After another unification

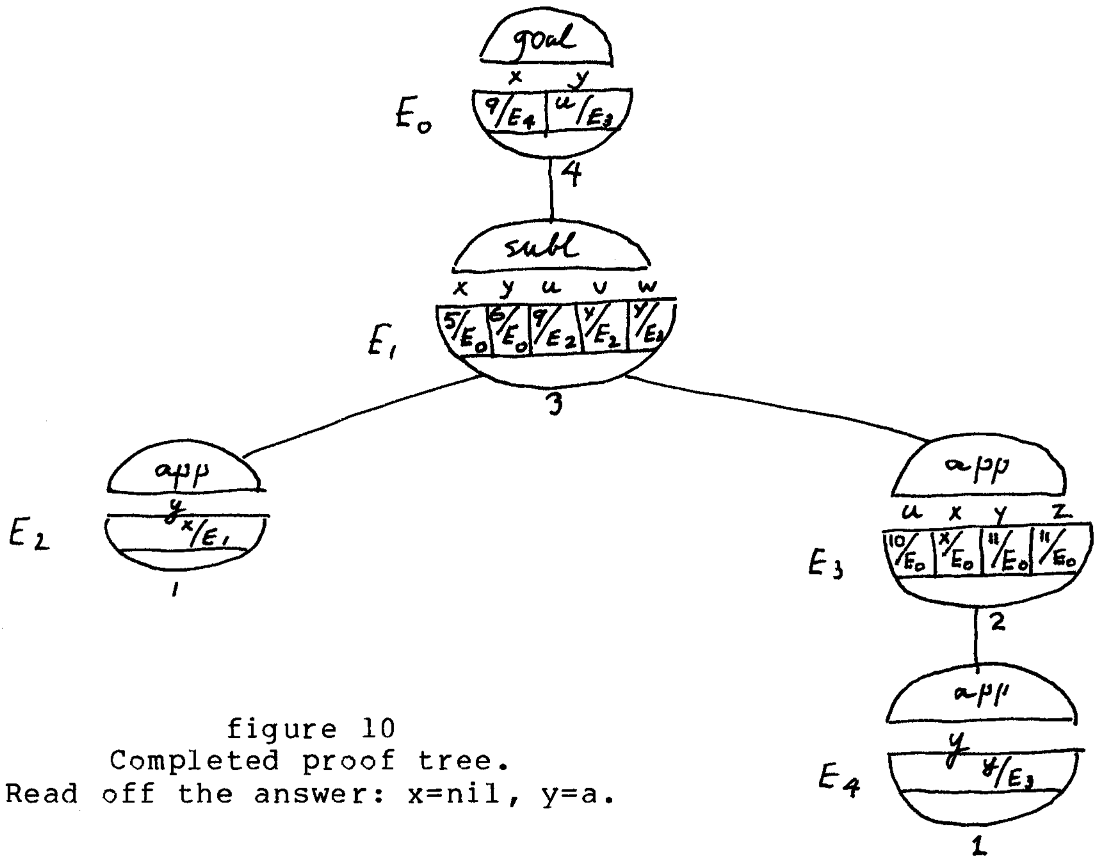

figure 10
Completed proof tree.
Read off the answer: x=nil, y=a.

A useable 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 realize 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 is 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 observed [6] that, with this selection, the order of the stack frames in the stack is the one obtained by preorder traversal [8] of the full circles of the proof tree.

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 [3]. 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 elaboration 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:	An 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. Only at one point in the ABC algorithm do all components of the state enter into play. This is where a new son 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 thus:

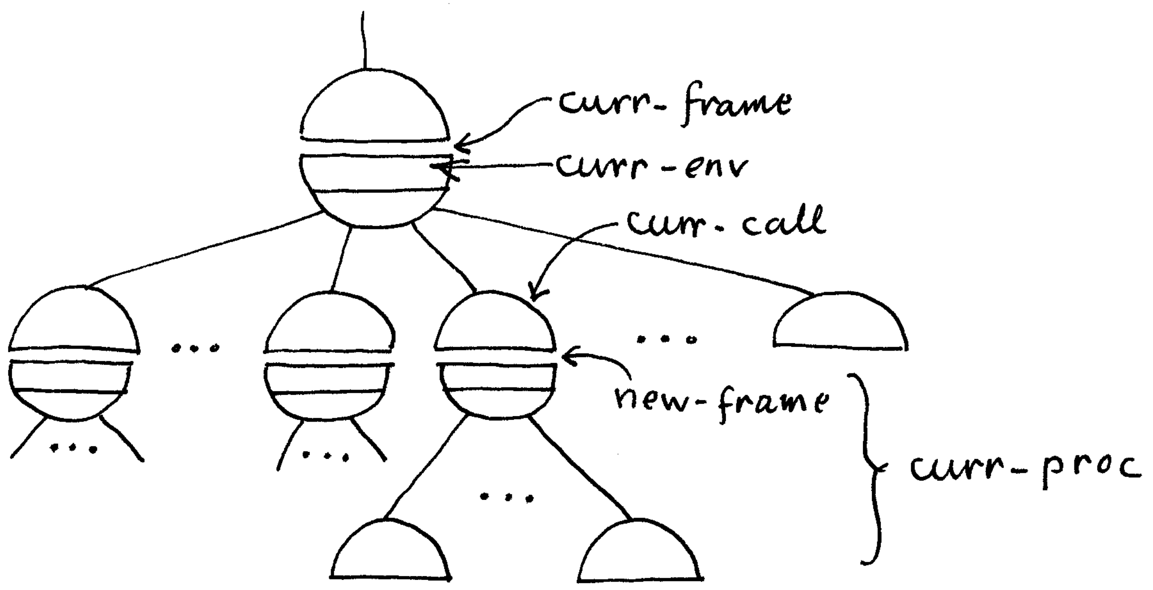

figure 11
All components of the state in use

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 as 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 ununified 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 below.

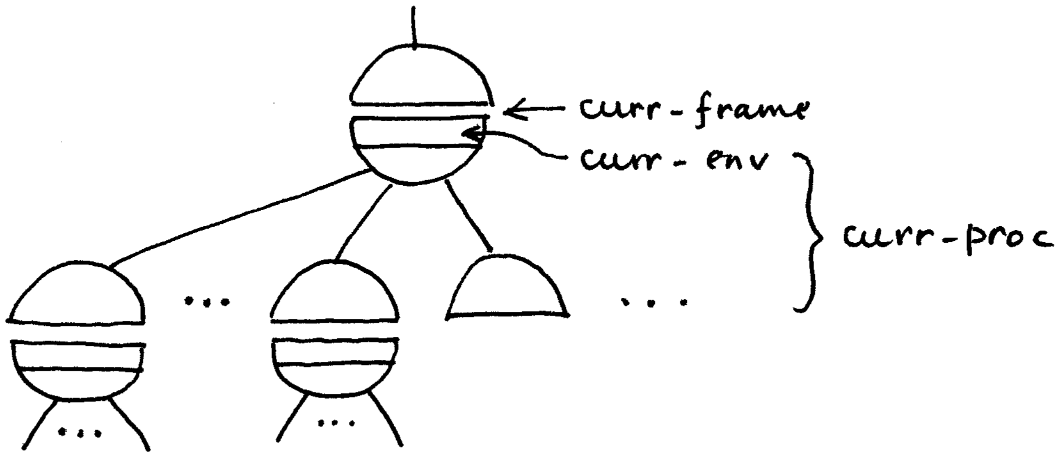

figure 12
Components of the state at label A

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, initializing 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 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 curr-frame and new-frame. Just after "son" has returned TRUE, the state is as indicated in figure 11. Just after "son" has returned

FALSE, the state is as indicated in figure 13.

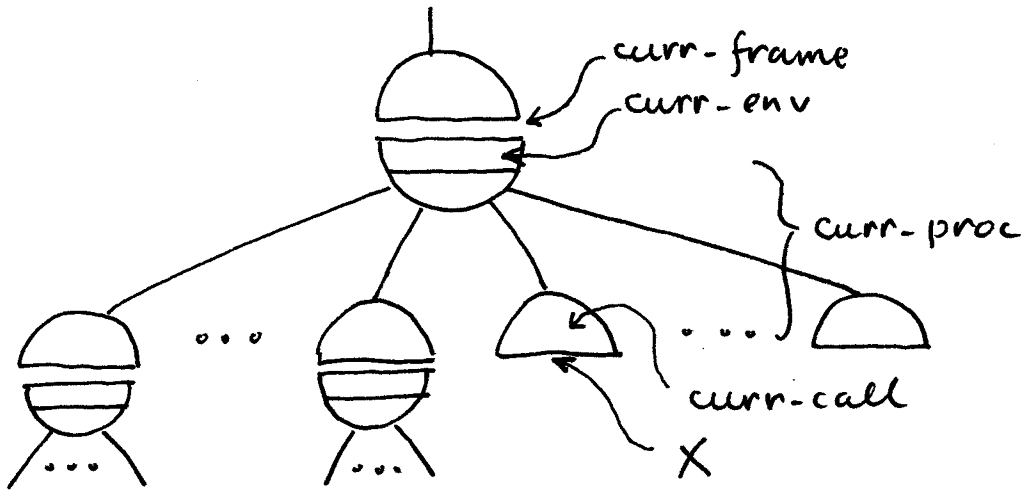

figure 13
The state just after failure to find a son

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.

```
        curr-proc := initial goal statement
                     {disguised as procedure
                      goal <- ...
                     }
        curr-env  := create-env(curr-proc)
        curr-frame:= create-frame(curr-env)
    A:  if select(curr-call)
        then {the current goal statement is nonempty;
              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 push curr-frame
             curr-frame := new-frame
             curr-env := ENV(new-frame)
             goto A
        else goto C
        fi
    C:  if stack nonempty
        then pop stack into temp-frame
             undo bindings recorded in RESET(temp-frame)
             next-clause := NEXT-CLAUSE(temp-frame)
             curr-frame := FATHER(temp-frame)
             curr-call := CALL(temp-frame)
             curr-env := ENV(curr-frame)
             curr-proc := PROC(curr-frame)
             goto B
        else halt with failure
        fi
```

figure 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)
```

figure 15
Auxiliary functions for the interpreter

Acknowledgments

Many thanks to Ron Ferguson for some very helpful discussions. The National Science and Engineering Research Council financed supporting facilities.

References

[1] K.R. Apt and M.H.van Emden: Contributions to the Theory of Logic Programming; Journal of the ACM, to appear.

[2] R.S. Boyer and J. Moore: The sharing of structure in theorem-proving programs; Machine Intelligence 7, B. Meltzer and D. Michie (eds.), Edinburgh University Press, 1972.

[3] M. Bruynooghe: Naar een betere beheersing van de uitvoering van programma's in de logika der Horn-uitdrukkingen. Thesis, Katholieke Universiteit Leuven, 1979.

[4] K.L. Clark: Predicate Logic as a Computational Formalism. Springer Verlag, to appear.

[5] M.H. van Emden: Programming with verification conditions. IEEE Transactions on Software Engineering, SE-5 (1979), 148-159.

[6] R.J. Ferguson: An Implementation of Prolog in C. Master's Thesis, Department of Computer Science, University of Waterloo.

[7] S.C. Johnson: Yacc, Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, New Jersey, 1978.

[8] D.E. Knuth: The Art of Programming, vol. I, Addison-Wesley, 1968.

[9] M.E. Lesk and E. Schmidt: Lex - A lexical analyzer generator. Bell Laboratories, Murray Hill, New Jersey, 1975.

[10] C.S. Mellish: An alternative to structure sharing in the implementation of a Prolog interpreter. Logic Programming, K.L. Clark and S.A. Tarnlund (eds.), Academic Press, to appear.

[11] G.M. Roberts: An Implementation of Prolog. Master's Thesis, Department of Computer Science, University of Waterloo, 1977.

[12] D.H.D. Warren: Implementing Prolog: Compiling Predicate Logic Programs. Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh, 1977.