記　号　処　理　21－4
（１９８２．１２．２１）

Associative Evaluation of PROLOG Programs *

中村克彦　（東京電機大学　理工学部）**

ABSTRACT　An evaluation method for Prolog programs is represented, which is employed in H-Prolog interpreter. In this method, hash memories are used to store several kinds of information for the purpose of high speed access and efficient comparison of data. The main working storage is a hash memory for variable-value pairs called bindings. The notion of binding is extended so that it is referred by its variable name and a label called a context which is generated at each application of a clause. Each binding contains a context to determine wehether it is "alive" or "dead". Another hash memory contains the "monocopy" lists which represent subterms in a program and the indices of clauses. The system written in the C language is simple because of employment of the data structures based on the hash techniques and of LISP functions.

1. Introduction

Recently, Prolog [2,3,10,11] is becoming to be recognized as a powerful means in various field of artificial intelligence. This language has been developed for predicate logic programming [7,8] which has its origin in mechanical theorem proving [10]. Because of this the implementation of Prolog, i.e. the construction of an interface between the language and current hardware and software, differs from that of most programming languages which reflect existing computer architectures. A unique feature of Prolog is that a program is evaluated by a sequence of pattern matching processes called unifications between two predicate terms. A unification generates a collection of variable-value(subterm) pairs called bindings, which are used in the later evaluation. Control of the evaluation is based on pattern-directed procedure call (or application of closes to goals) and nondeterminism which is implemented by means of depth-first tree search and backtracking as are other languages for artificial intelligence, e.g. PLANNER [5].

In this paper, we discuss some methods employed in our Prolog system called H-Prolog. In the system, several kinds of information are stored in hash memory. The main working storage is a hash memory and contains the bindings and some control information for evaluation of the program. Another hash memory contains subterms in source programs for the purpose of simple comparison of two subterms in unification. This memory also contains the indices of closes for efficient selection of the applicable clauses. Some comments are in order on the background of our methods in comparison with previous systems.

The data structure for instances, i.e. terms which are derived from source terms in source programs by substitution of variable(s), is based on "structure sharing" introduced by Boyer and Moore [1], as are many Prolog systems. In structure sharing, an instance of a source term is not constructed in working memory. The working memory contains a set of the bindings, called the environment, and whenever the system encounters a variable it refers the environment to get the corresponding value.

In our system, the bindings are stored in hash memory instead of stacks or list memory. Because some variables may have been defined in an earlier stage before being substituted in an evaluation of a goal, and since they must be restored to be uninstantiated in backtracking, the data structure in the working storage is essential to efficiency of the evaluation. In the stack-type Prolog systems, more than one stacks are used: binding information

---

* This work was done while the author was at Machine Intelligence Research Unit, University of Edinburgh.

** Katsuhiko Nakamura (Faculty of Science and Engineering, Tokyo Denki University, Hatoyama-machi, Saitama-ken 350-03 Japan).

is stored in two stacks [11], or the instances are generated (or copied) in a stack [9] in the unification. Another stack called the trail is used in these systems to store information for backtracking. In the system using a list memory as the working storage [1], the bindings are represented by means of "association lists," which may change to "garbage cells" in backtracking.

Our method utilizes the notion that we can determine whether a binding is "alive" or "dead" from a label (called a context) unique to an application of a clause added to the binding. A variable in a dead binding is considered to be unbound, and the location of the dead binding in the hash memory is treated as an unused place. Therefore, restoring the variable to be unbound in backtracking is done by simply "eliminating" the context. Furthermore, our method as well as the stack-type Prolog systems does not need garbage collection of the working storage found in list-processing systems.

In our system, certain subterms and indeces of the clauses are stored in a hash memory as "monocopy" lists. The concept of monocopy lists and their use for "associative computation" for LISP programs were introduced by Goto [4] and implemented in HLISP System. A monocopy list is a list such that each cell for the list, including a cell for an atom, is placed in the location determined from the two pointers in the cell by a hash function (in the case of an atom cell, the location is determined by its print name). One of the advantages of this method is economy of memory space, since identical subterms (or sublists) are stored only once. The other is that the equality of two list structures can be determined by simple comparison of the pointers. During an associative computation, the monocopy lists are used for indexing the partial results of computations in order not to repeat the same computation by checking the stored information before partial computations.

2. Preliminaries

A Prolog program is written as a sequence of clauses, where a clause is a sequence of one or more predicate term(s). A term is either a constant, a variable, or a function (or predicate) term composed of a function (predicate) symbol and its arguments, which are also terms. In H-Prolog, the first character of a variable is an upper-case letter, and that of a constant and a function symbol a lower-case letter as in [3]. Since we employ lists to store the source programs we represent a term by LISP S-expressions whose instances are of the form (f a1 ... an.am), and a clause of the form (P0 P1 ... Pn), n >= 0, where the a's and P's are function terms and predicate terms, respectively.* A set of clause is called a database. A clause (P0 P1 ... Pn) is equivalent to the logical expression P0 <- P1 & ... & Pn, where <- is the implication operaton, and all variables in the clause are universally quantified.

An environment is a set of bindings, each of which is a pair of a variable and a sublist of a term, i.e. either a term, a function symbol, or a list of a terms. A variable V is said to be bound to a term t in an environment E, if (V,t) is in E. The instance <P,E> of a list P in an environment E is the list which is constructed recursively by simultaneously replacing each of variables in P by the list to which it is bound. By an application of a clause C to an instance (called a goal) <G,E>, we mean an attempt to generate an environment E[n] in the following way:

(1) Renaming of the variables in C so that all the variable names are different from those in G and E. Let C'= (P0 P1 ... Pn) be the renamed clause.

(2) Unification for <G,E> and P0. If the unification succeeds, it generates the minimum set S of bindings such that the two instances <<G,E>,S> = <G, E∪S> and <P0,S> = <P0,E∪S> are identical. If the unification fails, the application fails.

(3) If the clause is a fact (n = 0), the application terminates. If the clause is a rule (n >= 1), predicate terms P1,...,Pn are evaluated from left

---

* In H-Prolog, the terms can be written as f(a1,...,an) or p(a1,...,an).

to right. This is to find a sequence of clauses C1,...,Cn in the database such that Ci is applied to (Pi,E[i-1]) and to generate E[i] for all i, 1 <= i <= n, where E[0] = E S. Each application of a clause may involve the subsequent applications of clauses until facts are applied to the subgoals.

Computation in Prolog system, or "top-level evaluation," is evaluation of a given sequence of goals as in Step (3). Finding the sequence of clauses is based on depth-first tree search and backtracking [3,11].

A clause may contain a special predicate term "!" as a goal which is called the cut. In backtracking the cut in a clause stops the re-evaluation of the goals before the cut in the clause and the goal to which the clause applied.

We use two kinds of lists: LISP-type lists (or L-lists) and the monocopy lists (H-lists). Although L-lists can contain H-lists as their sublists, all sublists of H-lists are H-lists. The basic functions in LISP such as atom(x), car(x), cdr(x), and caar(x) can be applied to lists of both types. H-type lists can be identified by the predicate hp(x), which is true if and only if x is a pointer to a cell of H-lists including to an atom cell.

We shall describe some basic parts of the system in the C language [6]. Explanations are included as comments in the programs for the readers who are not familiar with C, but know PASCAL.

3. Bindings and the Unification Algorithm

In this section, we introduce a particular type of binding, and present the unification algorithm we use. Each binding contains additional information to determine whether it is alive or not. A context is a value which is unique to each application of a clause, and is used to refer an environment. We define a binding to be a 5-tuple (v,c1,c2,c3,t), where v is the pointer to a variable, c1, c2, and c3 are contexts, and t is a pointer to a sublist of a source term. A binding (v,c1,c2,c3,t) is interpreted as follows:

(1) v has occurred in the clause which is applied in context c1. The pair (v,c1) is considered as a renamed variable.

(2) by structure sharing, c3 and t represent an instance of t to which (v,c1) is instantiated.

(3) c2 is the context of an application of a clause by which this binding is generated.

The bindings are stored in a hash memory. The key by which its location is determined and the binding is accessed is both v and c1. A context is generated before each application of a clause. A context is "eliminated", when the corresponding application fails. If c2 is an eliminated context, the binding is not in use and the place it has occupied can be used for storing a new binding.

The following are basic functions for generating and accessing the bindings:

place(v,c1,c2,c3,t): This function places the binding at the location determined by both v and c1 in the hash memory.

getc2(v,c1), getc3(v,c1), getterm(v,c1): if there is a binding with both v and c1 as its key, the value of getc2, getc3, and getterm is the third item (c2), fourth item (c3), or fifth item (t) of the binding, respectively. Otherwise, the value is FALSE.

Our unification algorithm is shown in Figure 1. Like most Prolog systems, our program does not include the "occur check," i.e. the test to confirm that a term which is substituted to a variable does not contain the variable. The function unify(u,v,cu,cv,c) is to unify two instances represented by the pairs (u,cu) and (v,cv), where u and v are (the pointers to) source terms, and cu, cv and c are contexts, and to generate bindings in the hash

memory as its side effect. The value returned is TRUE if the unification succeeds, and FALSE otherwise. The context of the application which initiates this unification is assigned to the fifth argument c, and each generated binding contains it as its third item. The truth-value functions used in the program to test the conditions are shown in Table I.

The basic idea of this algorithm is common to most Prolog systems which employ structure sharing, except that the binding information is stored in hash memory instead of stacks or a list memory and that all variable-free subterms are represented by H-lists. Therefore, if both subterms to be unified are represented by H-lists then the value of the unification is determined simply by the equality of their pointers.

Example. Consider the unification of the instance (p (f X) (f a)) and the term (p Y Y), where the instance is represented by the source term (p.X) and the binding (X,S,U,U,((f X) (f a))) in the environment. Suppose that the function unify is called by

```
unify((p X),(p Y Y),S,T,T).
```

The function unify is subsequently called as follows:

```
unify(p,p,S,T,T)
unify(X,(Y Y),S,T,T)
unify((f X),Y,U,T,T)
unify((f a),(f X),U,U,T)
unify(f,f,U,U,T)
unify(a,X,U,U,T)
```

As the side effect, two bindings (Y,T,T,U,(f X)) and (X,U,T,U,a) are generated and stored in the hash memory. The unification succeeds and the value of the function unify is TRUE.

4. The Interpreter

In this section we discuss the basic parts of the interpreter, which is represented by the function goal in Figure 2. The function goal(glls,c) evaluates sequentially the goals in the list glls with respect to the initial environment referred by c. It calls try(p,c), if the list glls is not empty (NIL) and if the leftmost predicate term in the glls is not the cut symbol. The value returned by try(p,c) is a list (clls) of clauses whose head terms may be unified with p. A practical approach for try(p,c) is to return the list of the clauses such that the predicate in the head of the clause is that in p. Later, we discuss a more efficient 'indexing' method to find a possible clauses. The function goal then tries to find a clause in clls which can be applied to the first goal in glls.

Because of possibility of backtracking the function goal(glls,c) returns the value TRUE only after the top-level evaluation of all the goals has succeeded, although it is called whenever a rule is applied to a goal. When a clause C = (P0 P1 ... Pn) is applied to the first goal of glls = (G1 ... Gm) such that the instance of G1 and P0 unifies, evaluation of the goals P1,...,Pn is followed by that of G2,...,Gm. This information is represented by the 5-tuple of the form (NIL,T,T,c,(G2 ... Gm)), where T is the context of the application of C. This 5-tuple is also stored in hash memory for the bindings. Since the first item of the 5-tuple is NIL, it can be distinguished from the bindings. The reason for using this method is that the lists of goals (P1,...,Pn) and (G1,...,Gm) is treated as a single list, but the contexts for the two lists are different.

The interpreter provides some built-in predicates similar to SUBRs and FSUBRs of LISP. When the system encounters a built-in predicate, it calls a subroutine associated with the predicate name. Although this facility must be included in the function goal, it is omitted in Figure 2. More detailed description of the function goal is given in Appendix.

5. Associative Indexing

When the database contains a large number of clauses, it is essential to select the applicable clauses efficiently. In addition to the indexing of clauses by the predicate names of the head clauses, our system employs another indexing method which utilizes the H-list. In this method, an index is a special term which contains no variable and is represented as an H-list. An index is constructed from an instance of a head clause by changing its 'non-indexing' sublists, which is specified as being no part of the index, into a special symbol '$'. This symbol is also used to specify the non-indexed sublists in the head clause to be indexed: if the car-part of a sublist is $, the cdr-part of the sublist is the specified sublist. For example, some possible indices for an original instance (p (f a X) b) are

(p $ b), (p (f.$) b), and (p (f a $) $).

These are generated from the instances,

(# p ($ f a X) b), (# p (f $ a X) b), and (# p (f a ($ X)) ($ b))

respectively, with specification of non-indexing parts. The symbol '#' means that these instances are to be used for the associative indexing.

Some functions are provided in H-Prolog system to generate the indices and store the clauses in database and to apply these clauses to goals. The built-in predicates asserta(C) and assertz(C) generate the index from the instances of the head term of the clause C and add the instance of the clause at the beginning and the end of the database, respectively, if the head term is the form (#.P).

The symbol # in a goal of the form (#.G) behaves like a special predicate to evaluate the goal G with respect to the clauses with associative indexing (it is handled by the function try(p,c)). It first finds a clause whose index is identical to that of the instance of G. This process is like the generation of an index by asserta or assertz except that it does not generate any H-list but 'traces' the generation of an index. Note that selecting a clause is done through several hashing operation in this process.

A special H-cell called an associator is used to relate an index to its clause. (The concept of associater is introduced by Goto [4].) An associater is a cell which contains two pointers and whose location is determined by one of the pointers called a key. Therefore, if the key points to an index and the other pointer refers to its clause, then the corresponding clause can be found from the index.

6. Implementation

The H-Prolog interpreter is written in the C language and was implemented in the DEC PDP-11 computer. A simple re-hashing method is employed to handle conflicts of hashing. In general, no 'garbage' list cell is produced in the evaluation. However, in some case that a clause is retracted from the database, or numerical computation is taken place, some 'garbage' H-cells may be remained. An unused L-cell is returned to the free list whenever it is produced by the storage allocation function of C.

The program is comparatively small: the size of the object code for the interpreter including those for built-in predicates is approximately 12 K bytes. The execution times for some simple Prolog programs are comparatible to those of the UNIX Prolog system [3,9] which uses stacks to store instances by 'non-structure-sharing' method and which is written in the assembly language. On the other hand, memory usage for the bindings is not efficient because a binding has five items (10 bytes in the case of PDP-11), and because evaluation speed decreases considerably when the major part of the hash hemory is occupied by the bindings.

7. Concluding Remarks

We have represented a new evaluation method for Prolog programs and its implementation. The system written in a higher level language is simple and portable. The simplicity is derived from the fact that our system need have only one stack used implicitly in the program as well as use of the Lisp functions and hash memories. This system is being transplanted to other machines.

Our bindings represented by the 5-tuples contain sufficient information even if the depth-first search is replaced to breadth-first or parallel search. Therefore, application of our method to "concurrent Prolog" is being planned.

ACKNOWLEDGEMENT The author would like to thank Professor Donald Michie for encouragement and interest in this work, and Tim Niblett, Alen Shapiro, and Dr. David Bowen for assistance in programming and preparing the manuscript.

References

(1) Boyer, R. S. and Moore, J. S., The sharing of structure in theorem proving programs, in Machine Intelligence 7, (ed. Melzer, B. and Michie, D.), Edinburgh University Press, (1972)

(2) Clark, K. L. and McCabe, F. G., The control facilities of IC-PROLOG, in Expert Systems in Micro Electronics Age (eds. Michie, D.), Edinburgh University Press, pp122-149 (1979).

(3) Clocksin, W. F. and Mellish, C. S., Programming in Prolog, Springer-Verlag (1981).

(4) Goto, E., Monocopy and associative algorithms in extended LISP, Technical Report of Information Science Labaratory, University of Tokyo (1974).

(5) Hewitt, C., Planner: a language for proving theorems in robots, Proc. First IJCAI, pp.295-302 (1969).

(6) Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall Inc., Englewood Cliffs (1978).

(7) Kowalski, R., Logic for Problem Solving, Elsevier North Holland, New York (1979).

(8) Kowalski, R., Algorithm = logic + control, Jour. ACM 22, (1979).

(9) Mellish, C. S., An alternative to structure sharing in the implementation of PROLOG programs, Dept. of Artificial Intelligence Research Paper NO.150, University of Edinburgh

(10) Robinson, J. A., A machine oriented logic based on resolution principle, Jour. ACM 12, pp.23-44 (1965).

(11) Warren, D. H. D, Implementing PROLOG - Compiling Predicate Logic Programs, Vol. 1, Dept. Artificial Intelligence Report No.39, University of Edinburgh (1977).

Appendix. Description of the Function goal

Consider the case that glls = (G1...Gm) and c = S, and that the value of try(G1,S) is (C1...Ck) and C1 = (P0 P1...Pn). Firstly, suppose that glls does not contain the cut. Then, evaluation of the goals by the call goal(glls,c) proceeds as follows.

(1) A new context is generated by the function newcontext() and assigned to t. Let T be the generated context.

(2) Unification is attempted for the instance represented by G1 and S and the term P0. It generates bindings in the hash memory. If the unification fails, the context T is marked as 'eliminated' by the function eliminate(t), and the next clause in clls is tried to apply the goal (G1,S) by the next iteration of the 'while' loop.

(3) In the case that the unification succeeds and the clause is a fact (n = 0), the list (G2 ... Gm) is evaluated by function goal. If the unification succeeds and n >= 1 then the 5-tuple (NIL,T,T,S,(G2...Gm)) is stored in the hash memory and the goals represented by (P1...Pn) are evaluated by goal. If these are evaluated successfully, then glls becomes to be equal to NIL, and G2,...,Gm are to be evaluated next.

(4) After the evaluation of the goals, the context T is eliminated. If the evaluation has succeeded (i.e. the value of goal is TRUE), the function goal returns TRUE. Otherwise, the system is in the backtracking state, and the next clause in clls is tried to apply the goal (G1,S).

It is supposed that every context is not equal to 0, except the context for 'top' evaluation. Note that the locations for 5-tuples stored in Step(3) as well as those for the bindings can be re-used after the context T is eliminated.

Next, consider the case that a predicate, say P[i], in the clause C1 is the cut and that goal ((P[i+1]...Pn),T) returns the value FALSE. Then the the context T is assigned to the global variable cutat, and goal returns the value CFAIL. The value CFAIL makes goal to return CFAIL immediately without retrying application of the other clause in clls, until the control backtracks to the stage where the value of cutat equals to T and the clause C1 was chosen. In this stage, the value FALSE is returned although remaining clauses (C2...Ck) are not also retried.

```
unify(u,v,cu,cv,c)        /* definition of the function */
  list u,v;               /* type declaration of the arguments */
  int cu,cv,c;            /* contexts are represented by integers */
{
  while (TRUE) {          /* this block is a loop */
    if (isvar(u)) {
          if (unbound(u,cu)) {
                place(u,cu,c,cv,v);
                return(TRUE);
                }
          u = getterm(u,cu);
          cu = getc3(u,cu);
          continue;       /* cause the next iteration of the loop */
          }
 L:  if (isvar(v)) {
          if (unbound(v,cv)) {
                place(v,cv,c,cu,u);
                return(TRUE);
                }
          v = getterm(v,cv);
          cv = getc3(v,cv);
          goto L;
          }
     if (hp(u) && hp(v))  /* && is the logical AND operator */
          return(u == v);   /*  == is the equality operator */
     if (atom(u) || atom(v))  return(FALSE);
                             /* || is the logical OR operator */
     if (unify(car(u),car(v),cu,cv,c) {
          u = cdr(u);
          v = cdr(v);
          continue;
         }
     return(FAIL);
    }
  }


 unbound(x,cx)
  list x;
  int cx;
    {
     return(getc2(x,cx) == FALSE || eliminated(getc2(x,cx));
    }
```

Figure 1. Definition unify and unbund of Function

Expressions connected by logical operators (&& and ||) are supposed to be evaluated from left to right, and evaluation stops as soon as TRUE or FALSE of the results are found.

function	condition such that the value is TRUE
isvar(x)	x is a (pointer to a) variable.
isconstant(x)	x is a (pointer to a) constant.
eliminated(c)	The context c is eliminated.
unbound(x,cx)	The variable x in the context cx is unbound.

Table I. Truth-Valued functions

```
int cutat;          /* definition of a global integer variable  */

goal(glls,c)        /* evaluate a list glls of goal terms.  The value is
                       an integer constant either TRUE, FALSE, or CFAIL */
 list glls;
 int c;
 {
   while (TRUE) {                /* this block is a loop  */
         if (glls == NIL)
              if (c != 0) {
                 return(goal(getterm(NIL,c),getc3(NIL,c)));
                 continue;
                               /* cause the next iteration of the loop */
                }
              return(TRUE);
              }
         if (car(glls) == CUT) {
             if (goal(cdr(glls),c)) return(TRUE);
             cutat = c;
             return(CFAIL);
            }
        break;                      /*  exit the block  */
       }
   {
    list clls;
    int t, p;
    clls = try(car(glls),c);
    while (clls != NIL) {   /* "!=" is "not equal to" */
         t = newcontext();
         if (unify(car(glls),caar(clls),c,t,t)) {
             if (cdar(clls) == NIL) p = goal(cdr(glls),c);
             else {
                place(NIL,t,t,c,cdr(glls));
                p = goal(cdar(clls),t);
                }
            eliminate(t);
            switch (p) {        /* equal to "case p of" in Pascal */
                 case FALSE:break;
                 case TRUE:return(TRUE);
                 case CFAIL:if (cutat = t) return(FALSE);
                            else return(CFAIL);
                }
            clls = cdr(clls);
            }
          else { eliminate(t);
                 clls = cdr(clls);
                 }
         }
     return(FALSE);
     }
}
```

Figure 2. The Function goal