ABSTRACT

A VIRTUAL MACHINE TO IMPLEMENT PROLOG.

Gerard BALLIEU
Department of Computer Sciences
K.U.Leuven
Celestijnenlaan 200 A
B-3030 Heverlee (Belgium)

We describe the design and the definition of a vitual Prolog machine. Like other computers, this virtual machine has an instruction set and a working storage (sstatements and data).

The design of the instruction set is mainly based on the implementation by D. Warren on the DEC10 where he used an abtract machine to explain the principles involved in his compiler. The organization of the working storage corresponds to the "non-structure sharing" technique of C.S. Mellish or the "copying" approach of M. Bruynooghe.

One of our main purposes is of course to realise the idea of the virtual machine. The execution of a Prolog program on the vitual machine consists of two steps:

- Compilation of Prolog programs to virtual machine instructions. The compiler is written in Prolog and the compilation process should be completely reversible.

- Interpretation of the virtual machine instructions. An interpreter is being developed in a high level language (Pascal and C) and it should be mainly portable.

It is our goal to combine the advantages of both compiled Prolog (efficiency) and interpreted Prolog (adaptibility). We argue that this implementation is easily portable to different computer systems be rewriting only that part of the interpreter which implements the built-in procedures.

A VIRTUAL MACHINE TO IMPLEMENT PROLOG.

Gerard BALLIEU
Department of Computer Sciences
K.U.Leuven
Celestijnenlaan 200 A
B-3030 Heverlee (Belgium)

1. Introduction.

Prolog is a simple but powerful programming language based on symbolic logic. A lot of specific features such as declarative reading, incomplete data structures, unification and non determinism make Prolog programs very attractive and well suited for solving a great variety of problems. There is a growing interest to use Prolog as a software tool to design and develop new projects. In order to support Prolog as a real programming language, we design a Prolog system having the following charateristics:

- the Prolog system has to be efficient: compared with other languages the execution time must be reasonable (maximum 3 or 4 times slower) and the storage use may not overload the computer system.

- the Prolog system should be portable to a variety of machines and it should be easily adaptable to the specific capabilities of a particular computer.

- Prolog programs have to be compiled to virtual machine instructions which are completely machine independant.

- the data representation in the Prolog system should cover both Prolog implementations on conventional machines and on dedicated hardware.

In the next section we describe the main features (storage areas and instructions) of the virtual machine. Some design decisions are discussed and compared with the Prolog implementation of D. Warren [5]. Finally we discuss the current implementation and give some future developments.

2. Description of the virtual Prolog machine.

2.1. General processes.

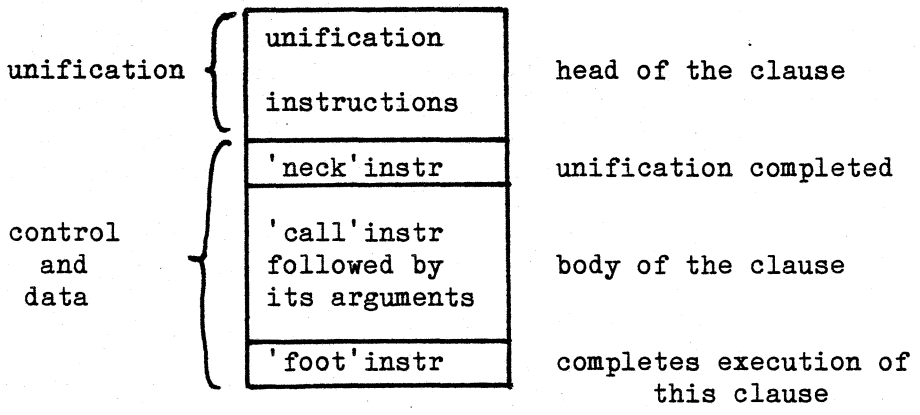
We design a virtual machine with an architecture which should support the efficient execution of Prolog programs. The execution mechanism of logic programs consists in constructing a sequence of

proof-trees according to the depth-first left-to-right search strategy [1] and to store in each node the appropriate variables and data. The fundamental questions we have to answer are of the form: "what does the machine do?" and "where and how does it represent its data?".

A Prolog machine has to perform two kind of processes: a control process and a unification process. On a sequential machine architecture these processes are alternated. The control process selects the next goal and the procedure definition, adjusts the proof-tree or restores the proof-tree to a previous state. The unification process is in fact a computation process which tests and assigns data or creates complex data structures.

To represent the control information and the data structures involved in the execution of a Prolog program, the virtual machine will provide a complex run-time structure consisting of an environmentstack, a copystack and a resetstack (or trail). For complex terms we use the "structure copying" approach.

The design of our virtual machine has strongly been influenced by the work of D. Warren [5] where he used an abstract machine to explain the Prolog compilation process. We also compile each Prolog clause into a sequence of virtual machine instructions according to the following scheme:



2.2. The main working storage.

The major data area of the virtual machine is the environmentstack. Like in block structured languages this stack is used to build a run-time environment for each goal (procedure call). When a new goal or subgoal is takled, a new stackframe is created and space is reserved for the variables and for linking (management) information.

The stack frames are linked in two kinds of lists: a father-list and an alternative list. Each stackframe belongs at least to one of the lists. The father-list corresponds to a path in the proof-tree from the root to the current node. The alternative list is a list of backtrackpoints or nodes with alternative choices to solve the goal corresponding to the node. In figure 1 we show for a given proof tree the corresponding environmentstack: P is the initial goal or problem, Di is a deterministic node and Bi is a backtrackpoint.

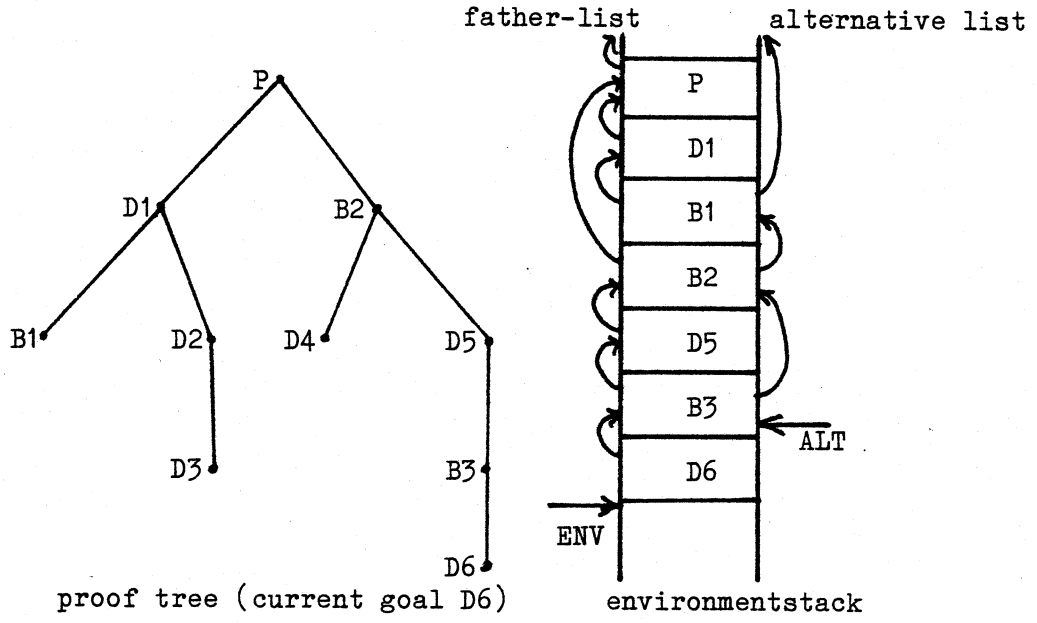


figure 1

The top element of the father-list and the alternative list is pointed by ENV respectively ALT. When a goal is successfully completed and no alternative choices remain (no backtrackpoint), the top frame of the stack (father-list) is removed. When a goal fails, the last backtrackpoint becomes the current frame and an alternative clause is chosen to solve the current goal.

Each stack frame also has space for the variables in the corresponding clause. Due to the general tree structures and the incomplete data structures in Prolog (dynamic data structures) it is not always possible to put the variable binding in the reserved space. When a variable's value is a constant (atom or integer) the value is put in the stack frame. When a variable is bound to a compound term (functor and arguments) a copy of this term is made and put on a second stack, the copystack, and a reference to this copy is put in the stack frame. Another reason for having two stacks is that on successful completion of a deterministic goal we will deallocate a stack frame and that for further computation we

still need the variable bindings. The value of a variable in the environmentstack can either be a constant, undefined (free variables), a reference to a compound term on the copystack or a reference to another variable earlier in the environmentstack.

The third working area of the virtual machine is the resetstack which is a trail or a push-down list. This area is used to store the addresses of variables which need to be reset to undefined (free) on backtracking.

The copystack and the resetstack generally increase in size with each new goal and are reduced by backtracking. The top elements are pointed by COPY respectively RESET and the old values of these pointers are kept in the mangement information part of the last backtrack frame. The management info contains also the links of the father-list and the alternative list, and pointers to the current goal and the alternative clauses if any.

Next to the working storage areas which are writable, we have the code area for storing the code of the compiled program. Information in the code area is generally accessed in a "read-only" manner.

2.3. The instruction set.

According to the control process and the unification process we can classify the virtual machine instructions in two classes: the unification instructions and the control instructions.

2.3.1. The unification instructions.

The main computation in Prolog consists of a sequence of unifications or pattern matching operations. Each unification involves matching two terms. One term is a "goal" (or procedure call) followed by its parameters and is instantiated. The other is the uninstantiated "head" of a clause. The control instructions verify that unification only takes place between a goal and a clause with the same name and arity. The unification process tries to match each of the arguments of the head of the clause against the corresponding arguments of the goal.

Instead of using a general matching procedure, the head of a clause is translated into unification instructions, most of which are simple tests and assignments. The arguments of the goal are translated into a sequence of literals (or "argument instructions").

The variables of a clause are categorised in three classes as follows:

- local variables: multiple occurences, with at least one in the body, numbered from 1 to n

- temporary variables: multiple occurences, all in the head of the clause, numbered from n+1 onwards

- void variables: single occurences.

The unification instructions are:

```
uvar(i)     : matching of the free variable i against ...
uref(i)     : matching of the bound variable i against ...
uint(j)     : matching of the integer value j against ...
uatom(a)    : matching of the atom a against ...
uvoid       : matching always succeeds
uterm(fn,n): matching of the functor fn with arity n against ...
```

(the number of a variable refers to a variable in the current frame.)

The literals (argument instructions) are:

```
var(i)       : the free variable i
ref(i)       : the bound variable i
atom(a)      : the atom a
void         : a void variable
funct(fn,n) : the functor fn with arity n
```

(the number of a variable refers to a variable in the goal frame.)

The next table gives an overview of the unification process:

goal / head	var	ref	atom	int	void	funct
uvar	assign	assign	assign	assign	assign	copy assign
uref	assign	general	case of	case of	success	case of general
uint	assign	case of	fail	test	success	fail
uatom	assign	case of	test	fail	success	fail
uvoid	assign	success	success	success	success	skip
uterm	copy assign	case of general	fail	fail	skip assign	test

```
assign : simple assignment
copy   : copy a compound term
test   : simple test (and assignment)
case of: multiple test
general: general unification algorithm
```

Most of the unification instructions are simple test and assignment instructions. If one of the terms is a reference we have to dereference that term until we get its value (undef, atom, int or funct). We can avoid long reference chains if we use only references to compound terms or to free variables. (Otherwise the value is copied.) The length of the reference chain would be mostly one.

There are two cases where we have to copy a compound term, depending on its source:

- the compound term appears in the haed of the clause

- the compound term appears in the argument list of the goal. Since the argument list is accessed in a read-only manner, only the parts containing variables must be copied. Therefore the compound terms are marked with "labelvar" or "labelcons".

There are three cases where the general unification algorithm can be invocated. This happens when two compound terms are to be unified and neither of them is known at compile time.

Remark that the virtual machine has no special instructions for initialising variables since the types "ref" and "var" indicate if a variable is free or bound.

2.3.2. Control instructions.

Each clause of a Prolog program is translated into a sequence of virtual machine instructions consisting of unification instructions for the haed of the clause, literals for the argument lists and control instructions (neck, call and foot).

- neck(n) : unification is completed; n is the number of local variables to be kept on the current environment.

- call(p) : this is a procedure call; a new frame is created, the call or return address is saved and a jump to address p is performed.

- foot : completes the execution of a goal, possibly removes the current frame and transfers control to the next instruction of the parent goal.

A Prolog procedure is composed of one or more clauses and is

translated into a list of control instructions of the form:

```
p : enter
    try(C1)
    try(C2)
      .
      .
    trylast(Cn)
```

enter : new procedure starts
try(Ci) : execute the instructions of clause Ci and note that there are alternative choices (backtrackpoint).
trylast(Cn) : execute the instructions of the clause Cn.

Note that these instructions manage the different clauses of a procedure and that they are generated at the end of the compilation process. If we extend our Prolog system with built-in predicates for adding or deleting clauses, this part of the code must be changeable.

Finally we have two control instructions which are strongly related to the Prolog source program: "cut" and "fail".

- cut(i) : i is the number of local variables; the alternative list must be adjusted and space can be recovered from the environmentstack.

- fail : forces backtracking.

2.4. Example.

As an example we show the quicksort program: source and virtual machine instructions.

```
qsort(.(X,L),R,R0):-partition(L,X,L1,L2),
                     qsort(L2,R1,R0),
                     qsort(L1,R,.(X,R1)).
qsort(nil,R,R).

partition(.(X,L),Y,.(X,L1),L2):-
lt(X,Y),!,partition(L,Y,L1,L2).
partition(.(X,L),Y,L1,.(X,L2)):-
partition(L,Y,L1,L2).
partition(nil,_,nil,nil).
```

```
3qsort1 : uterm(.,2)
          uvar(0)
          uvar(1)
          uvar(2)
          uvar(3)

3qsort2 : uatom(nil)
          uvar(0)
          uref(0)
          neck(0)
          foot
```

```
                neck(7)
                call(partition,4)
                ref(1)
                ref(0)
                var(4)
                var(5)
                call(qsort,3)
                ref(5)
                var(6)
                ref(3)
                call(qsort,3)
                ref(4)
                ref(2)
                labelvar(1)
                fn(.,2)                    3qsort : enter
                ref(0)                              try(3qsort1)
                ref(6)                              trylast(3qsort2)
                foot

4partition1 : uterm(.,2)        4partition2 : uterm(.,2)
              uvar(0)                         uvar(4)
              uvar(1)                         uvar(0)
              uvar(2)                         uvar(1)
              uterm(.,2)                      uvar(2)
              uref(0)                         uterm(.,2)
              uvar(3)                         uref(4)
              uvar(4)                         uvar(3)
              neck(5)                         neck(4)
              call(lt,2)                      call(partition,4)
              ref(0)                          ref(0)
              ref(2)                          ref(1)
              cut(5)                          ref(2)
              call(partition,4)               ref(3)
              ref(1)                          foot
              ref(2)
              ref(3)
              ref(4)
              foot

4partition3 : uatom(nil)          4partition : enter
              uvoid                            try(4partition1)
              uatom(nil)                       try(4partition2)
              uatom(nil)                       trylast(4partition3)
              neck(0)
              foot
```

3. Implementation.

As a first step in our Prolog system the Prolog source programs are compiled into a sequence of virtual machine instructions. A first version of the compiler has been written in Prolog itself. [6] The output consists of symbolic Prolog machine code as illustrated in the previous example and of two tables: a functor table (names of the predicates and arity) and an atom table.

The next step in our Prolog system is the interpretation of the virtual machine instructions. The interpreter should be query-oriented and has the following structure:

```
init read-only part (code area)
WHILE not end
  DO  read query
      compile query (set Program Counter to first instr.)
      init working storage
      execute (Program Counter)
      remove query
```

The initialisation part reads the symbolic code and transforms it into a sequence of word-codes which are loaded in the code area. The call instructions are divided in two classes: calls of evaluable predicates (built-in procedures) and calls of user-defined procedures. In the WHILE-loop a query is read and compiled into a sequence of word-codes which are added to the code area. This compilation can result in extending the atom table and the functor table. After execution of the query, the code area and the tables are restored.

In our prototype version we have split up the code in two parts:

- the executable part (unification and control instructions) is put in the code area

- the literal part (argument lists) is put on the copystack as a read-only segment. The literal part has the same structure as the compound terms except that a literal can be "var(j)" while a compound term on the copystack has the value "undef" instead of "var".

Figure 3 gives an overview of the Prolog system.

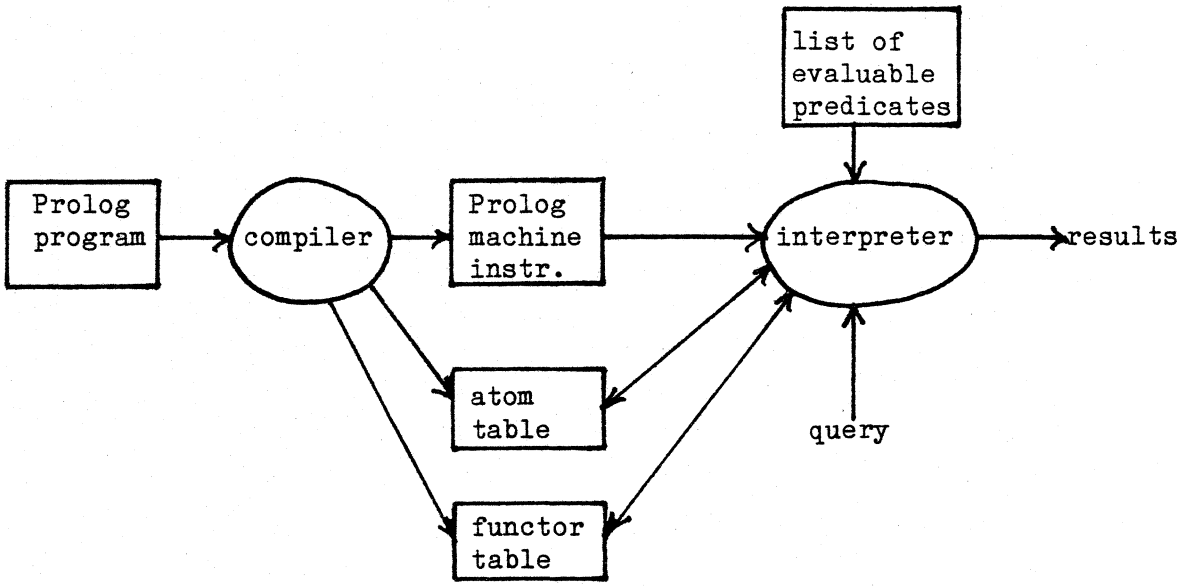


Figure 3

4. Design concepts.

Having described the main features of our Prolog system, we now comment some design concepts and their consequences.

- The "structure copying" approach is used as data representation technique for compound terms. Compared with the implementation of D. Warren, in our system there is no need to split up the variables in globals and locals: they are all local. A compound term is copied on the copystack only if it has variables. In addition the copying approach will behave better when a garbage collector is needed. [2]

- For the head of a clause we generate executable code for all terms nested to any level. We also detect the first occurrences of the variables in the body of the clause and the arguments of the goals are marked with "var" or "ref". Due to this decision we have eliminated the need to initialise the variables and the specific initialization instructions.

- The variables of the parent frame which are bound during the execution of a goal are never to be put on the resetstack because the arguments of the goal define which variables are free.

- The Prolog system has a modular strucure. Optimizations and extensions of the system require only small adjustments. The

implementation of the "neck"-instruction is responsible for tail recursion optimization. If we will add the "occurcheck" to the unification process, we only have to extend the implementation of "uref".

- The Prolog system is easily portable to other machines. If we will take full use of the capabilities offered by the underlying machine, it is sufficient to adapt the implementation of the evaluable predicates or to add new built-in procedures.

5. Future developments.

The virtual machine described in this paper is being implemented. A prototype of this machine has been written in the language C (under the UNIX operating system) and some simple Prolog programs have been tested. In comparison with the existing interpreter (written in C by M. Bruynooghe) our system behaves favourably in speed and space. For more complex programs we expect better results. Another implementation will be written in Pascal for machines with a Pascal-oriented architecture such as the PERQ.

We further plan to set up a complete Prolog program environment for this Prolog system:

- the current implementation will be optimized: tail recursion, clause selection based on the arguments, intelligent backtracking...
- development of a Prolog debugging tool
- different modules of a Prolog program may be compiled and linked into one executable program.
- the list of built-procedures and utility programs has to be extended
- the Prolog system has to be coupled with a relational database or with a database machine.

Acknowledgements

I am grateful to Maurice Bruynooghe for the many helpful discussions and to Gerda Janssens who has implemented and tested as a student project, this virtual machine in Prolog and C.

6. References.

[1] Bruynooghe, M. The memory management of PROLOG implementations, in Logic Programming (Clark & Tarnlund eds.), Academic Press, 1982.

[2] Bruynooghe, M. A note on garbage collection in Prolog interpreters, Proc of the first Int Logic Conf., Marseille, September 1982.

[3] Cloksin, W.F. and Mellish, C. Programming in Prolog, Springer Verlag, 1981.

[4] Mellish, C.S. An alternative to structure sharind in the implementation of a PROLOG-interpreter, in Logic Programming (Clark & Tarnlund eds.), Academic Press, 1982.

[5] Warren, D.H. Implementing PROLOG- Compiling Logic Programs, 1 and 2, D.A.I. Research Report No 39, 40, University of Edinburgh, 1977.

[6] Warren, D.H. Logic programming and Compiler Writing, Software Practice and Experience, Vol 10, nr 2, pp97-126.