A portable Prolog compiler

Conference Paper · January 1983

CITATIONS
13

READS
380

3 authors, including:

William Clocksin
University of Hertfordshire
106 PUBLICATIONS **2,646** CITATIONS

SEE PROFILE

Proc. Logic Programming Workshop '83,
Albufeira, Portugal.

A PORTABLE PROLOG COMPILER

D.L. Bowen, L.M. Byrd

Dept of Artificial Intelligence
University of Edinburgh

and W.F. Clocksin

St Cross College, Oxford

ABSTRACT

This paper describes the basis of the design of a Prolog implementation which is currently being built. This new implementation is intended to combine a high degree of portability with speed and efficient utilisation of memory. Our approach is to compile Prolog clauses into instructions for a relatively high-level abstract machine. This abstract machine is implemented by an interpreter written in a high-level systems programming language (C), giving a portable Prolog system.

Some portability must be sacrificed, however, in order to achieve the high speed required. The design is well suited to tailoring for particular machines, because there is a small central core of the interpreter which does most of the work. This central core can be translated into assembly language or microcode when necessary.

An advantage of this approach is that it avoids the compiler/interpreter dichotomy found in DEC-10 Prolog and LISP systems with compilers. All clauses are compiled, but compilation is reversible so that it is not necessary to have a separate representation of the textual form of clauses.

1. Introduction

This paper describes some design principles behind current work at Oxford and Edinburgh Universities to build a new Prolog system. The desired qualities of the new system are that:

(1) It should be highly portable.

(2) It should be fast and use memory efficiently; this requirement directly conflicts with (1).

The approach we have chosen is to compile Prolog clauses into code for a relatively high-level (i.e. Prolog oriented) abstract machine. This abstract machine is implemented by an interpreter written in a high-level systems programming language (C). The compiler, and many of the evaluable predicates, are written in Prolog itself. This approach has allowed us to get a preliminary version of the system running fairly quickly.

However, this system as it stands will not meet our requirement for speed. A certain amount of non-portable work will be necessary in order to achieve high speed on particular computers. Our intended methodology is to translate the most heavily used parts of the C code into assembly code, or microcode where possible (e.g. on the ICL Perq). This non-portable work is minimised because the central core of the interpreter is simpler and smaller than that of a direct Prolog interpreter.

We have opted for the sructure-copying method of [Mellish 80] and [Bruynooghe 80], rather than structure-sharing [Warren 77]. An important reason for this is that structure-copying is expected to give better locality of reference and therefore better paging behaviour on virtual memory systems. Another advantage is that it allows us to dispense with holding the Prolog form of all the clauses in the heap: our abstract machine is so arranged that we can reconstruct these terms when they are needed (i.e. in the implementation of the evaluable predicates 'clause' and 'retract') by effectively decompiling the compiled form of the clauses.

Our storage management strategy is basically that of [Warren 77], i.e. there is a heap containing the program, a "local" stack for control information and variable bindings, a "global" stack for structures, and a "trail" stack which keeps track of when variables are bound so that they can be reset to "uninstantiated" at the appropriate time on backtracking. One change is that a reference count is maintained for each clause so that pointers to clauses (as returned by the predicate clause/3 in DEC-10 Prolog) can safely be included in asserted terms. A consequence of this slightly complex memory management is that it is never necessary for a garbage collector to do a full sweep of the heap; it only has to sweep the local and global stacks.

As our run-time system is based on previously published work [Warren 77] [Warren 80], we will concentrate in the rest of this paper on the new part of our design which is the intermediate language.

2. The Intermediate Language

In this section we introduce the kernel of the intermediate language into which Prolog clauses are translated. Although this language subset has only seven instructions, it is sufficient; the only reason for extending it is for efficiency as will be discussed later. We introduce it syntactically by discussion of the (reversible) compilation of a Prolog clause. The semantics of the language will be explicated in the following section by means of a simple interpreter for it written in Prolog.

A compiled clause has two main parts: an External Reference (XR) table, and a block of byte-codes. Let us consider the compilation of the clause:

```
p(tp1,tp2,...)  :- q(tq1,tq2,...), r(tr1,tr2,...).
```

where the tpi, tqi and tri are arbitrary terms. The general form of the byte-code block is then:

```
<code for tp1>
<code for tp2>
 ...
enter
<code for tq1>
<code for tq2>
 ...
call <XR offset for procedure q>
<code for tr1>
<code for tr2>
 ...
call <XR offset for procedure r>
exit
```

This introduces the three "control" instructions we need: 'enter', 'call' and 'exit'. The 'enter' instruction simply marks the division between the head and the body of the clause. Each 'call' has an argument (the next byte-code in the block) which refers to an entry in the XR table which is a reference to the required procedure. Finally, 'exit' marks the end of the clause.

The terms which are the arguments of the head of a clause, and those which are the arguments of goals, are all translated in the same way. Each term is compiled into "data" instructions as follows:

(1) If the term is atomic it is translated as

```
const <XR offset>
```

where the corresponding entry in the XR table is either an integer (if the term is an integer) or a pointer to an atom record.

(2) If the term is a variable it is translated as

```
var <number>
```

where the variables in the clause are numbered in order of appearance.

(3) If the term is compound it is translated as

```
functor <XR offset>
<code for 1st argument>
<code for 2nd argument>
 ...
pop
```

The 'functor' instruction refers to an XR table entry which points to the corresponding functor record. It is followed by the compiled form of each of its arguments, followed by a 'pop' instruction.

For the purposes of the interpreter to be presented in the next section, we need to represent compiled code as Prolog data structures. Compiled procedures will be represented as assertions of the form:

```
procedure( Name/Arity, List_of_Clauses ).
```

A clause will be represented by a term:

```
clause( XR_Table, Number_of_Variables, List_of_Bytecodes )
```

An XR table is also represented as a term:

```
xrtable(...)
```

where the table entries are either integers, atoms, functors (written in the form Name/Arity), or procedures (written as procedure(Name/Arity)).

For example, the compiled form of the procedure:

```
append(nil,L,L).
append(cons(X,L1),L2,cons(X,L3)) :- append(L1,L2,L3).
```

looks like this:

```
procedure( append/3, [
    clause( xrtable(nil), 1,
        [ const, 1,                                  % nil
          var, 1,                                    % L
          var, 1,                                    % L
          exit]),
    clause( xrtable(cons/2,procedure(append/3)), 4,
        [ functor, 1, var, 1, var, 2, pop,           % cons(X,L1)
          var, 3,                                    % L2
          functor, 1, var, 1, var, 4, pop,           % cons(X,L3)
          enter,
          var, 2, var, 3, var, 4, call, 2,           % append(L1,L2,L3)
          exit ])]).
```

3. An Interpreter for the Intermediate Language

We now present our mini-interpreter written in DEC-10 Prolog. For simplicity, we use the unification and backtracking capabilities of Prolog rather than doing everything explicitly as is necessary in a real implementation. A consequence of this is that cut cannot easily be implemented in the mini-interpreter.

The entry point to the interpreter is the procedure arrive/3. Its arguments are the procedure to be called, a list of its arguments, and a continuation list which represents goals still to be solved. E.g. to append one list to another we would call:

```
:- arrive(append/3,[cons(a,cons(b,nil)), cons(c,nil), L],[]).
```

This call should succeed, instantiating L to :

```
cons(a,cons(b,cons(c,nil))).
```

There are two clauses for arrive/3 (Figure 1). The first of these finds any compiled clauses for the procedure. It then non-determinately selects (using member/2) the first clause, i.e. future failure will cause us to backtrack here and select another clause if there is one. Next it creates a new set of (uninstantiated) variables by means of the built-in predicate functor/3 which sets Vars to be the functor with name 'vars' and having Nvars uninstantiated arguments. Finally control is passed to execute/6 to execute the byte-code list (which we have called PC because it corresponds to the Program Counter in a real implementation).

The second clause for arrive/3 allows the built-in predicates of Prolog to be used in the mini-interpreter.

```
arrive(Proc,Args,Cont) :-
        procedure(Proc,Clauses), !,             % Find clause list for Proc
        member(clause(XR,Nvars,PC),Clauses),    % Select one
        functor(Vars,vars,Nvars),               % Make new set of variables
        execute(PC,XR,Vars,Cont,Args,[]).       % Go to execute byte-codes
arrive(Name/Arity,Args,Cont) :-
        Proc =.. [Name|Args],                   % No compiled clauses: call
        call(Proc),                             %   normal Prolog procedure
        execute([exit],_,_,Cont,_,_).           % and continue

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

Figure 1: arrive/3

The clauses for execute/6 (Figure 2) are all determinate, so that it resembles a CASE statement in other languages. Let us consider the data instructions first, assuming for now that they are in the head of a clause (i.e. before the 'enter' instruction).

The 'const' instruction is fairly straightforward: it simply matches the first element of the argument list with the appropriate entry in the XR table. (arg(X,XR,Arg) unifies Arg with the Xth argument of the term XR.) If successful, it then tail-recursively calls execute/6 to execute the subsequent instructions with the rest of the argument list. Note that if Arg was initially uninstantiated it will have become instantiated to the given constant. Similarly, 'var' matches the given variable with the current argument.

For 'functor' we first check that the argument has the right principal functor (or instantiate it to the most general term with this principal functor if it is uninstantiated). If successful, we obtain the list Args of the arguments of Arg and go to execute subsequent instructions which are to be matched against them. There remains the list Arest of arguments to be matched after Arg. This list is stacked on Astack from where it is


```
execute([const,X|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(X,XR,Arg),                          % Match XR entry with Arg
        execute(PC,XR,Vars,Cont,Arest,Astack).
execute([var,V|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(V,Vars,Arg),                        % Match variable with Arg
        execute(PC,XR,Vars,Cont,Arest,Astack).
execute([functor,X|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(X,XR,Faton/Farity),                 % Get functor from XR table
        functor(Arg,Faton,Farity),              % Match principal functors
        Arg =.. [Faton|Args],                   % Get Args of Arg term
        execute(PC,XR,Vars,Cont,Args,[Arest|Astack]).
execute([pop|PC],XR,Vars,Cont,[],[Args|Astack]) :- !, % Pop Args off Astack
        execute(PC,XR,Vars,Cont,Args,Astack).
execute([enter|PC],XR,Vars,Cont,[],[]) :- !,
        execute(PC,XR,Vars,Cont,Args,Args).     % Initialise diff list:
execute([call,X|PC],XR,Vars,Cont,[],Args) :- !,
        arg(X,XR,procedure(Proc)),              % Extract proc name from XR
        arrive(Proc,Args,[frame(PC,XR,Vars)|Cont]).   % Save context & go
execute([exit],_,_,[frame(PC,XR,Vars)|Cont],[],[]) :- !,
        execute(PC,XR,Vars,Cont,Args,Args).     % Resume previous context
execute([exit],_,_,[],[],[]) :- !.              % No previous context: stop
```

Figure 2: execute/6

removed by the corresponding 'pop' instruction.

We have explained how the data instructions work in the head of a clause. It is the 'enter' instruction that ensures that they also work in the body, where what they are required to do is build up rather than take apart the argument list. What it does is initialise a difference list: a partially formed argument list is the difference between the 6th and 5th arguments of execute/6. For example, if two arguments have been processed we would get a goal of the form:

```
:- execute(_,_,_,_,X,[<arg 1>,<arg 2>|X]).
```

Thus each data instruction encountered in the body appends an argument onto this argument list by instantiating the variable at the end of it to [<argument>|<new variable>]. It is interesting to see how this works for 'functor': this is left as an exercise for the reader!

The 'call' instruction terminates the difference list by instantiating the variable at the end to []. It then goes off to arrive at the called procedure with the new argument list, first stacking all the information needed to resume this clause on the continuation list.

Of the two clauses for 'exit', the first is selected when the continuation list is non-empty. It causes resumption of a clause after the successful completion of a 'call'. Note that it is necessary to reinitialise the difference list here so that another argument list is constructed for the next 'call'. The second clause for 'exit' terminates the program.

4. Some Additions to the Intermediate Language

It may be noticed that there is no point in returning from the last 'call' in a clause and restoring its context only to immediately 'exit' and restore a previous context. This can be avoided by introducing a new 'depart' instruction which replaces the last 'call' and the subsequent 'exit' (cf. [Warren 80]). The interpreter is easily extended to handle this new instruction by the addition of one more clause for execute/6:

```
execute([depart,X],XR,Vars,Cont,[],Args) :- !,
        arg(X,XR,procedure(Proc)),
        arrive(Proc,Args,Cont).
```

This is just like 'call' except that no continuation frame is stacked.

Another inefficiency arises in the execution of 'functor' if it appears as the last argument in the clause head, or as the last argument of some other term. In either case there are no remaining arguments (Arest is [] or will be instantiated to [] later) but we are stacking Arest anyway and popping it back to no useful purpose when 'pop' is encountered. The cure is to introduce another new instruction, 'lastfunctor', which is like functor except that it has no corresponding 'pop'. It is interpreted thus:

```
execute([lastfunctor,X|PC],XR,Vars,Cont,[Arg],Astack) :- !,
        arg(X,XR,Fatom/Farity),
        functor(Arg,Fatom,Farity),
        Arg =.. [Fatom|Args],
        execute(PC,XR,Vars,Cont,Args,Astack).
```

Various other instructions can be introduced to save space in the clause representation or to gain speed. An example is <'immediate' N> which allows a small integer N to be represented directly in the byte-code block without the need for an XR table entry. It is also useful to provide instructions for the simpler built-in predicates such as integer/1, var/1 etc. A possibility is to combine some of the instructions with their most common arguments to make new single-byte versions of two-byte instructions, but the trade-off with increasing the size of the interpreter needs to be studied empirically.

5. Considerations for a Practical Implementation

The operation of our environment (or local) stack, which holds continuation and backtrack information as well as the arguments of procedures and variable bindings, is based closely on [Warren 80]. At the point where we are about to commence execution of a byte-code block, the top frame of this stack is like this:

CP (blank)	Continuation (byte-code) Pointer
CL (blank)	Continuation Local stack frame
XR (blank)	XR table for continuation
BP	Backtrack Point (clause pointer)
BL	Backtrack Local frame
TR	Trail marker
G (blank)	Global stack marker
Argument 1	
...	
Argument m	
Var 1 (blank)	
...	
Var n (blank)	

The first three words of the frame (marked blank because they have not yet been filled in) are for exactly the continuation information that was in the continuation stack of the mini-interpreter: the CL pointer allows access to the variables of the contination frame. The next four words are for control of backtracking. Then come the arguments to the procedure, which have already been filled in, followed by the variables which have not.

An argument register, A, is initially set to point to Argument 1. Each byte-coded instruction matches against the argument pointed to by A and then increments it. When a 'functor' instruction matches against an uninstantiated argument, it creates a new term with uninstantiated arguments on the global stack, and A is then set to point to the first of these new arguments. The previous value of A is saved on a special stack so that it can be retrieved by the corresponding 'pop'.

We do not actually have to initialise all the variables in the local stack frame to be "uninstantiated". The first occurrence of <'var' N> in a clause (for each N) is changed to be a new instruction <'firstvar' N> which simply assigns the value indicated by A to variable N. If a variable only appears once in a clause, there is no point in doing even this much work, so there is also a 'void' instruction which does nothing.

Another improvement we can make is to overlap the variable and argument blocks in the stack frame. That is, if a variable appears at the top level in the head of a clause, e.g. L2 in append([X|L1],L2,[X|L3]) :- ...), then we can use the appropriate argument slot for the variable value, thus saving space and avoiding superfluous assignments. All that has to be done is rearrange variable frame offsets appropriately (variables are not actually numbered 1,..,n, but by their offsets in the frame), and use 'void' instead of 'firstvar'.

Without special-purpose hardware, there is bound to be inefficiency in the way we have described building terms: first we build the term with all its arguments uninstantiated, and then subsequent instructions match against these uninstantiated arguments and fill them in. This involves (unnecessary) testing to see if each argument is uninstantiated; also it is in general necessary when instantiating a variable to test whether or not it

should be put on the trail. We avoid all this checking, and the need for initialising the arguments of the constructed term, by introducing a new mode of interpretation of our instruction set. This is called 'copy' mode, as opposed to 'match' mode which is what we have been discussing until now. In 'copy' mode data instructions simply copy the data they stand for over to A.

This concept of interpreter modes can also be useful for debugging. In normal operation, the abstract machine goes to great lengths to throw away any information which it will not need again. When debugging, this is undesirable, so we plan to include a 'debug' mode in which more information is kept.

One other complication should be mentioned. This is the problem described in [Warren 80] of dangling references arising from tail-recursion optimisation. We follow his approach of putting variables which may give rise to this problem onto the global stack. For this purpose we require two new instructions which are global stack versions of 'var' and 'firstvar'.

6. Related Work

A compiler for Prolog has been written in POP11 by C.S. Mellish at Sussex University. This actually compiles Prolog into the POP11 abstract machine language which is then in turn compiled into real machine language. Advantages of this approach are (1) relative ease of implementation, and (2) instant access to a good programming environment. The long-term drawback, however, is that there is no possibility of tailoring the memory management to the special needs of Prolog. The fully general POP11 garbage collector has to be used (even for backtracking).

Another approach has been taken by [McCabe 83]. His Abstract Prolog Machine is specified at a much lower level than ours, and depends on the availability of a LISP style garbage collector of some sort.

7. Conclusions

The design we have described is a compromise between pure interpretation and pure compilation. Preliminary tests have shown our initial system to be comparable in speed with Pereira's C-Prolog interpreter [Pereira 82]. It has the advantage over pure interpretation that it is easier to optimise for particular hardware: the kernel of the interpreter is relatively simple and compact and well suited to microcoding.

Our design requires much less space for program storage than pure compilation, due to the relatively high level of the byte-code instructions, and to the fact that we do not need to store a separate representation of the Prolog source code. Also there is the advantage that there is no dichotomy between interpreted code (that you can debug) and compiled code (which goes fast) as there is on the DEC-10 system. Finally, our design has the advantage of minimising the amount of machine-specific work which needs to be done in implementation.

It is our belief that people are going to want to run larger and larger ("knowledge-based") programs, and that therefore the efficiency of both program storage and garbage collection will become increasingly important. Prolog does not require the generality of a LISP or POP garbage collector, so it should have an advantage over these languages if more efficient, special-purpose memory management is used.

8. Acknowledgement

We are indebted to David Warren and Fernando Pereira for the inspiration behind this work.

References

[Bowen 82] D.L. Bowen (ed.), L.M. Byrd, F.C.N. Pereira, L.M. Pereira and D.H.D Warren,
"DECsystem-10 Prolog User's Manual",
Department of Artificial Intelligence,
University of Edinburgh, 1982.

[Bruynooghe 80] M. Bruynooghe,
"The Memory Management of Prolog Implementations",
In "Logic Programming", ed. K.L. Clark and S.-A. Tarnlund,
Academic Press, 1982.

[McCabe 83] F.G. McCabe,
"Abstract Prolog Machine - A Specification",
Technical Report, Department of Computing,
Imperial College, London, February 1983.

[Mellish 80] C.S. Mellish,
"An Alternative to Structure Sharing in the
Implementation of a Prolog Interpreter",
In "Logic Programming", ed. K.L. Clark and S.-A. Tarnlund,
Academic Press, 1982.

[Pereira 82] F.C.N. Pereira,
"C-Prolog User's Manual, Version 1.1",
Edinburgh Computer Aided Architectural Design,
University of Edinburgh, September 1982.

[Warren 77] D.H.D. Warren,
"Implementing Prolog - Compiling Logic Programs",
Research Reports 39 & 40, Department of Artificial
Intelligence, University of Edinburgh, 1977.

[Warren 80] D.H.D. Warren,
"An Improved Prolog Implementation which Optimises Tail
Recursion",
Proceedings of the Logic Programming Workshop, Debrecen,
Hungary, ed. S.-A Tarnlund, July, 1980,
(also available as Research Paper 141, Department of
Artificial Intelligence, University of Edinburgh).