UPMAIL Technical Report No. 39

Efficient Interpretation of Prolog Programs

Jonas Barklund

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Dept., Uppsala University
P. O. Box 520, S-751 20 UPPSALA, Sweden
+46–18–18 25 00

Abstract The paper focuses on three ideas for solving problems with writing interpreters for the logic programming language Prolog in Prolog and how to combine these ideas to an interpreter for Prolog which is both simple and efficient. The resulting interpreter system can be incorporated into a Prolog based on Warren's Abstract Machine and built mostly from existing parts of it. The interpreter has been implemented and is used in a Prolog system developed at Uppsala University.

An earlier version of this paper was presented at the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques.

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

1. INTRODUCTION

The subject of this paper is a description of the interpreter used in Tricia, the Prolog implementation for the DEC-2060 computer developed at UPMAIL [Barklund *et al.* 86; Barklund & Millroth A]. For an introduction to Prolog, see [Sterling & Shapiro 86; Clocksin & Mellish 81]. Tricia is based on a revised version of David H. D. Warren's abstract Prolog machine (WAM) [Warren 83] and is mainly compiler-based [Carlsson 86]. Still an interpreter is useful for debugging and fast prototyping.

The paper contains three main ideas:

1. To translate the program to be interpreted into a subset of the language for the abstract machine.
2. To implement Prolog's *cut* for interpreted programs in a simple and efficient way.
3. To implement an interface between compiled and interpreted parts of the program such that compiled and interpreted predicates can call each other freely.

We combine these three points into an efficient Prolog interpreter system for WAM which can be easily built from moderately revised existing parts of the system. This interpreter is probably the smallest existing interpreter for Prolog in Prolog which really implements all control structures including cut.

The paper assumes some basic knowledge about Prolog.

2. PROLOG INTERPRETERS

The language Prolog is based on SLD-resolution with unification of (Horn) clauses [Robinson 65; Kowalski 79; Lloyd 84] which gives it a more sound theoretical basis than most languages. However, theoretically appealing languages have gained a reputation of being difficult to implement efficiently. In Prolog this is manifested in the fact that resolution requires the ability to create an instance of a clause from the database. The variables in this formula must be different from the variables of any other instance of this (or another) clause created before. There are two main approaches to creating such instances.

The first approach is called *structure sharing* [Bruynooghe 82]. This means that the clause instance is represented by a *molecule* containing a *skeleton* which is a copy of the clause with "holes" for the variables and a sequence of variables. The skeleton can be shared between all generated instances of a clause. If the clause contains n variables we create a sequence of n uninstantiated variables and put this together with the skeleton of the clause into a molecule. A molecule has certain similarities to the concept of *closure* commonly used in functional programming to get lexical variable scoping.

The second approach is called *structure copying* [Mellish 82]. In this approach we actually create a copy of the clause in the database but replace each of the variables in the clause by new uninstantiated variables. Of course we can share ground parts of the clause. In compiler-based machines such as WAM a variant of this is used where we only copy the arguments of subgoals and sometimes head arguments.

Creating instances of whole clauses is usually cheaper when using the former approach, which is a good argument for applying it on Prolog interpreters. However, since WAM falls into the latter category it seems natural to base the interpreter also on the same concept (although this is not necessary, as was shown in LM-Prolog [Carlsson 84; Kahn 83]). For this reason we choose structure copying for the interpreter, and thus have to make the creation of clause instances as efficient as possible.

To make the interpreter easy to write, debug, understand and maintain we chose to write most of it in Prolog itself. This is not a new idea, it is easy to write a toy interpreter for a large subset of Prolog in Prolog (for a very simple example, see [Bowen 81]). In fact, the only part of the interpreter not written in Prolog is the part that creates new instances of clauses. The usual Prolog predicate interfacing to this is `clause(Head,Body)` which unifies `Head` with the head of an instance of a database clause and `Body` with its body. It is usually assumed that at least the name and arity of the head is known when calling `clause/2`.

3. CREATING INSTANCES OF CLAUSES

The simplest way to create instances of clauses would be to store one actual instance of a clause in the database. When a new instance is required this clause instance is traversed and a copy of it is built. When a variable is encountered in the database clause a new variable is put into the copy. This has the advantage that putting a new clause into the database becomes a very fast operation and if writing new information into the database is more frequent than reading information then certainly this would be efficient. However, in most applications reading information is more frequent than writing, so this idea must be discarded.

A more sophisticated idea is not to store an instance of the clause itself in the database, but rather a recipe saying *how* to create such an instance. This recipe would be a program in an intermediate language. We find that there is a language very well suited for the task: WAM instructions.

The WAM instruction set has five distinct parts: *control*, *put*, *get*, *unify* and *index* instructions. Observing that the predicate `clause/2` behaves as if it were defined by a number of compiled facts (clauses with trivial bodies)

```
clause(Head_1,Body_1).
        ⋮
clause(Head_n,Body_n).
```

we find that the intermediate code for each of these clauses could consist of a sequence of get and unify instructions, since this is what clause heads are compiled into.

When a clause is asserted we thus compile it as if it were a clause for `clause/2` above. It was easy to extract a part of Mats Carlsson's compiler and remove time consuming optimisations. It is probably easy to do this for any Prolog-to-WAM compiler. The generated WAM code is then represented as byte codes and `clause/2` calls a small byte code emulator to create instances of a clause. This emulator would be quite similar to a corresponding emulator for full compiled code but considerably simpler since it would only have to emulate get and unify instructions and only temporary variables appear.

Here we have ignored the question how `clause/2` finds a clause to match a goal. This can be solved in several ways. In Tricia it uses a hash table indexed on functors to find a list of clauses having a certain principal functor. It then tries these clauses one by one. This can be improved in several ways, e.g., by discriminating on the type of the first argument.

In Tricia this technique is also used for storing items in the "internal database", which is in Edinburgh Prolog compatible implementations manipulated by the predicates `record/3`, `recorded/3`, `erase/1` etc.

Below we will give a neat solution to the problem of implementing *cut* (or *slash*) in Prolog interpreters, examples of the intermediate code generated for some clauses, a representation of the intermediate code as byte codes and an emulator for the intermediate code.

4. HANDLING CUT

One of the few devices for controlling the execution of Prolog programs is cut. Its effect is to purge any alternative solutions to the current goal, and thus make the computation so far of the current goal become deterministic. If the computation so far actually is deterministic, the cut doesn't have any effect but serves as a declaration (e.g., to the compiler) that some optimisations can be done, based on the fact that the computation will be deterministic. For a more thorough examination of different usages of cut see [Sterling & Shapiro 86].

Remembering possible alternative ways to find a solution is usually implemented (e.g., in WAM) with a stack of *choice points*. Each choice point contains a copy of the state of the machine at some point. If the current attempt to find a solution fails the most recent choice point is reinstated (the machine *backtracks*), the next possible way to reach a solution for that choice point is tried and the choice point is modified so next time backtracking is done another way is tried. When the most recent choice point has no more possible ways to reach a solution it is popped from the stack and the next choice point becomes visible.

The most natural way to implement cut is therefore to save a pointer to the choice point which was the most recent when the attempt to prove the current goal began, and when a cut in the current goal is reached, to pop all choice point more recent than the saved one. This will have the desired effect. Also some of the other data areas should be trimmed.

In [Barklund & Millroth 86] the full algorithm for cut in compiled WAM code is given. The compiler will issue a `choice(Vn)` instruction at the beginning of the code for each predicate containing a cut. This will save a pointer to the most recent choice point in the variable `Vn`. Each occurrence of cut in the predicate is then compiled to either a `cute(Vn)` or `cutne(Vn)` instruction depending on whether there is an active environment or not.

Thus, an interpreter has to remember somehow the current choice point when the proof of a new goal is attempted and when it reaches cut execute a `cutne` instruction. This can be done by creating Prolog predicates `'$choice'/1` and `'$cut'/1` interfacing to the `choice` and `cutne` instructions respectively. The interpreter program would then look something like

```
interpret_goal(Head) :-
```

```
   '$choice'(C), clause(Head,Body), interpret_body(Body,C).

interpret_body(true,C).
interpret_body((Goal_1,Goal_2),C) :-
   interpret_body(Goal_1,C), interpret_body(Goal_2,C).
interpret_body((Goal_1;Goal_2),C) :-
   interpret_body(Goal_1,C); interpret_body(Goal_2,C).
interpret_body((Goal_1->Goal_2),C) :-
   interpret_body(Goal_1,C) -> interpret_body(Goal_2,C).
interpret_body(_^Goal,C) :-
   interpret_body(Goal,C).
interpret_body(!,C) :-
   '$cut'(C).
```

This works but having to propagate the extra choice point argument is not so elegant.

The only reason to have the predicate `interpret_body/2` at all is to propagate the choice point argument. If this could be eliminated the predicate would be superfluous (but see below on meta-calls).

We make the observation that the propagated choice point is particular to each instance of a clause. This means that if we have to create a new instance of a clause anyway, we may just as well save the choice point in the clause instance itself. Doing this means we should pay attention to where the saved choice point is actually needed, namely at each occurrence of cut.

Our solution can now be sketched as the following: when a clause is asserted into the database we inspect the clause to find all top-level occurrences of cut. Instead of compiling them into `get_constant` or `unify_constant` instructions we compile them into `get_cut` and `unify_cut` instructions. The predicate `clause/3` is the same as `clause/2` with an extra argument which is a choice point. When the emulator encounters a `get_cut` or `unify_cut` instruction it generates a constant structure `'$cut'(C)` where `C` is the choice point given to `clause/3`. We will illustrate this by an example and explain the solution more thoroughly below.

Suppose we have asserted a clause

```
foo(X) :- bar(X,Y), !, baz(Y).
```

into the database and try to prove a goal `foo(42)`. The interpreter calls `'$choice'(C)` where `C` gets unified with something like `$666$` if that is what a pointer to a choice point looks like. Then the interpreter calls `clause(foo(42),Body,$666$)` which returns with `Body = (bar(42,Y), '$cut'($666$), baz(Y))`. Interpreting this code will call `bar/2` and if that is successful remove any choice points created by it (or by the call to `foo/1` itself if there were other clauses for it) and finally call `baz/1`.

4.1. Cut and Meta-calls

When inspecting clauses for cuts we are careful to distinguish between occurrences of `!` used only as atoms in terms and occurrences of `!` used as cuts. Term occurrences of `!` which later turn out to be used as cuts by meta-predicates such as `call/1` or `bagof/3` are a problem and have to be handled differently, but they are relatively few.

We can handle cuts like the one in

```
..., bagof(Y,(bar(X,Y), !, baz(Y)),L),...
```

by making `call/1` and its friends call `'$choice'/1` themselves and then call `interpret_goal/2` above with their goal and the returned choice point as arguments.

An alternative solution is to let `call/1` inspect the goal to be executed and replace all top-level cuts by ground calls to `'$cut'(C)` where `C` is a pointer to the most recent choice point. We can then use the ordinary mechanism for interpreting such a goal.

5. INTERFACE TO PREDICATES

We have chosen a scheme for calling predicates that makes it completely invisible to the caller whether the called predicate is interpreted or compiled. The problem is that compiled predicates in WAM expect their arguments in a set of *argument registers* but interpreted predicates expect their arguments as the arguments of a structure whose functor is the principal functor of the called predicate.

5.1. Calling predicates from compiled code

From compiled code we want to be able to call a predicate just by jumping to some address, perhaps storing the return address somewhere (usually in the `L` register). It does not matter much if calls from compiled to interpreted code cost a little extra since they can be expected to be infrequent.

The key thing is to make every jump to a predicate from compiled code indirect through a word particular to the predicate. This word (which is a part of the principal functor of the predicate) will contain an address. If the predicate is compiled this is just the address of the compiled code of the predicate. If the predicate is interpreted the address will be to a short piece of code which loads the functor of the called predicate (which is otherwise implicit in the address jumped to) into a register (`T1`) and then jumps to a common subroutine `$CALLI`.

So `$CALLI` expects the functor of the called predicate in register `T1` and the arguments in a set of argument registers. From these it builds a structure whose functor is the one in `T1` and whose arguments are those in the argument registers. Finally it puts this structure in the first argument register and jumps to the (compiled) predicate `interpret_goal/1` above.

5.2. Calling predicates from interpreted code

From interpreted code we can expect calls to interpreted (user) predicates as well as to compiled (built-in and user) predicates to occur frequently.

We need only one predicate which makes such calls and we name it `xcall/1`. We can then define the predicates available to users for making meta-calls, such as `call/1` in terms of `xcall/1`. (This is so they can play the tricks with cut as described above without introducing the overhead to do this for subgoals.)

`xcall/1` expects a structure (or an atom if there are no arguments) in the first argument register. If the call is to an interpreted predicate it can immediately jump to the code for `interpret_goal/1`. Otherwise it will load the argument registers from the arguments of the structure and jump indirectly through the address mentioned above in the functor.

6. THE INTERPRETER PROGRAM

By removing the handling of cut from the meta-interpreter and using the schema outlined above for calling predicates, the whole interpreter gets reduced to the following very comprehensive program:

```
interpret_goal(Head) :-
   '$choice(C), clause(Head,Body,C), xcall(Body).

true.

X,Y :- xcall(X), xcall(Y).

X;Y :- xcall(X); xcall(Y).

X->Y :- xcall(X) -> xcall(Y).

_^X :- xcall(X).

call(X) :- gobble_cuts(X,Y), xcall(Y).
```

This is at the same time a very simple and efficient interpreter.

7. THE INTERMEDIATE LANGUAGE

The instructions of the intermediate language consists of the *get*-instructions, *unify*-instructions and `proceed` from WAM. For a description of their exact operational semantics see, e.g., [Carlsson 86] or [Warren 83].

Cuts are compiled to one of two new instructions, `get_cut Xi` or `unify_cut` depending on where it occurs (a cut is compiled to a `unify_cut` instruction only if it occurs as the last argument of a structure). It is desirable for backwards compatibility that `clause/2` (which is not used by the interpreter) represents cuts in clauses by the atom `!` although `clause/3` (which is not visible to users) represents them by structures `'$cut'(...)`. Therefore cuts must be compiled in much the same way as structures although they look like atoms in the source code.

The instructions for cut can be executed in two modes, depending on whether the clause is created by `clause/2` or `clause/3`.

`get_cut Xi` If the clause is created by `clause/2` this is equivalent to `get_constant !,Xi`. Otherwise it is equivalent to `get_structure '$cut'/1,Xi` followed by `unify_constant CUT` where `CUT` is the choice point given as argument to `clause/3`.

`unify_cut` If the clause is created by `clause/2` this is equivalent to `unify_constant !`. Otherwise it is equivalent to `unify_structure '$cut'/1` followed by `unify_constant Cut` where `Cut` is the (constant) choice point given as argument to `clause/3`. (`unify_structure` is not present in the original WAM but is described in [Carlsson 86]).

8. GENERATING INTERMEDIATE CODE

Clauses are translated to the intermediate code by a compiler. To avoid confusion with the compiler which compiles clauses to the full WAM instruction set we will refer below to this compiler as the κompiler. The κompiler is actually a simplified subset of the compiler.

The κompiler first traverses the control structure of the clause replacing variables `X` by `call(X)`, conjunctions nested the "wrong way" `(X,Y),Z` by `X,(Y,Z)` and cuts `!` by `'CUT'(_)`. Since goals are not traversed, occurrences of `!` in arguments to predicates will remain untouched when the clause is regenerated by `clause/3`. Since cuts are represented by structures the latter phases of the κompiler automagically knows that they may generate structures when a copy of the clause is created.

The next κompiler pass makes a pre-order traversal [Knuth 73] of the head and the body of the clause, generating a sequence of abstract instructions, temporary variables are allocated and finally the abstract instructions are translated to WAM instructions. It would be possible to perform various optimisations in the generated code but we think fast κompilation to intermediate code is probably more important than the optimizations; if users want very fast execution they should not use an interpreter.

Our κompiler consists of 55 Prolog clauses occupying 110 lines of code including documentation.

9. CREATING CLAUSE INSTANCES FROM THE INTERMEDIATE CODE

When the intermediate code for a clause has been generated it should be stored in the machine. We can either compile the intermediate code further to machine code (which is executed to create a copy of a clause) or we can store the intermediate language instructions byte-wise (and emulate them to copy clauses).

Tricia has a machine code generator for generating PDP-10 machine code from WAM instructions. It could easily be applied to the intermediate language but we have not done this. Generation of copies of clauses would become even faster but the representation of clauses would use more storage. We have instead designed a representation of intermediate language instructions as bytes and written an emulator for this representation.

Instructions are stored as nine-bit bytes (PDP-10 has a word length of 36) with argument(s) in the following byte(s). Since there are only 25 different instructions (including all variants) we have assigned special codes for many common combinations of instruction and argument. E.g., there are 64 special codes for the instruction `get_variable Xi,Xj` where `i` and `j` are between 0 and 7, 8 special codes for `get_list Xi` for `i` between 0 and 7, etc.

The emulator is driven by two large tables (one for read mode and one for write mode), dispatching on the byte code. In a system in which compiled code is emulated, the usual WAM emulator could probably be used with minor modifications, such as adding the `get_cut` and `unify_cut` instructions.

10. DEBUGGING

An interpreter for debugging should probably be more conventionally built to be able to step or trace executions of programs. Since such an interpreter will probably be slower than the one presented here it is desirable to have two modes of execution: *normal* mode (this interpreter) and a *debug* mode (a debug-interpreter). When a goal is to be interpreted a jump is made to one of these interpreters.

This can be accomplished by having an memory word `INTVEC` containing the address of the predicate `interpret_goal/1` when in normal mode and the address of the corresponding predicate in the debug interpreter when in debug mode. The procedure of calling predicates from compiled code, described in section 5.1, should be changed correspondingly; instead of jumping directly to the predicate `interpret_goal/1` an indirect jump should be made via `INTVEC`.

It is possible to have several modes of interpretation, e.g., normal mode, step mode and trace mode, if this makes interpretation more efficient.

One problem remains: if debug mode is on, `call/1`, `bagof/3` & Co. should give their argument to the debug interpreter, so it can be traced/stepped. One solution is to let these predicates check explicitly if debug mode is turned on and in this case jump to some suitable part of the debugger. Another solution is to let the debug interpreter know all predicates which do meta-calls and make a normal recursive call in the debug interpreter.

Debug interpreters will be more thoroughly examined in [Barklund & Åhs].

11. RELATED WORK

W. F. Clocksin uses an idea similar to ours for storing clauses, but for a somewhat different purpose: allowing users to perform database operations on compiled predicates [Clocksin 85].

The handling of cut is often a kludge in Prolog interpreters, so the algorithms for it are not always presented, but [Nilsson 83] does present an alternative solution.

12. RESULTS

We have designed an interpreter for Prolog which probably can be included in most WAM-based Prolog implementations with little difficulty. We have not tried to run extensive benchmarks to compare it with other interpreters but it runs the traditional 30-element naïve-reverse program at around 4K LIPS which is approximately twice as fast as the interpreter of DEC-10 Prolog. We experienced a speed-up of approximately 30% when we incorporated the new scheme for cut.

ACKNOWLEDGMENTS

The author would like to thank his colleagues at UPMAIL for the stimulating environment and comments on his work, but particularly Mats Carlsson and Håkan Millroth who have contributed much to Tricia, Torbjörn Åhs for valuable support on debug interpreters and Amelie Banks who has improved the language of this paper. He is also grateful to his family for their support.

REFERENCES

[Barklund *et al.* 86]
J. Barklund, L. Oestreicher, Å. Hugosson and M. Nylén, *Tricia User's Guide*, Computing Science Department, Uppsala University, Sept. 1986.

[Barklund & Millroth 86]
J. Barklund and H. Millroth, *Garbage Cut for Garbage Collection of Iterative Prolog Programs*, in *1986 Symposium on Logic Programming*, Salt Lake City, Sept. 1986.

[Barklund & Millroth A]
J. Barklund and H. Millroth, *Code Generation and Runtime System for Tricia*, UPMAIL Technical Report 36, Uppsala, in preparation.

[Barklund & Åhs]
J. Barklund and T. Åhs, *A Debug Interpreter for Tricia*, in preparation.

[Bowen 81] D. L. Bowen, *DECsystem-10 Prolog User's Manual*, University of Edinburgh, Dept. of Artificial Intelligence, Edinburgh, 1981.

[Bruynooghe 82]
M. Bruynooghe, *The Memory Management of Prolog Implementations*, in K. L. Clark and S.-Å. Tärnlund, *Logic Programming*, London, 1982.

[Carlsson 84] M. Carlsson, *LM-Prolog — the Language and Its Implementation*, UPMAIL Technical Report 30, Oct. 1984.

[Carlsson 86] M. Carlsson, *Compilation for Tricia and Its Abstract Machine*, UPMAIL Technical Report 35, Uppsala, Sept. 1986.

[Clocksin 85] W. F. Clocksin, *Design and Simulation of a Sequential Prolog Machine*, New Generation Computing, vol. 3, pp. 101–120, 1985.

[Clocksin & Mellish 81]
W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, New York, 1981.

[Kahn 83] K. Kahn, *Unique Features of LISP Machine Prolog*, UPMAIL Technical Report 15B, Uppsala, 1983 (revised by M. Carlsson Feb. 1985).

[Knuth 73] D. E. Knuth, *The Art of Computer Programming, Volume 1 / Fundamental Algorithms*, Reading, Mass., 1973.

[Kowalski 79] R. Kowalski, *Logic for Problem Solving*, New York, 1979.

[Lloyd 84] J. W. Lloyd, *Foundations of Logic Programming*, New York, 1984.

[Mellish 82] C. S. Mellish, *An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter*, in K. L. Clark and S.-Å. Tärnlund, *Logic Programming*, London, 1982.

[Nilsson 83] M. Nilsson, *FOOLOG—A Small and Efficient Prolog Interpreter*, UPMAIL Technical Report 20, Uppsala, June 1983.

[Robinson 65] J. A. Robinson, *A Machine-oriented Logic based on the Resolution Principle*, Journal of the ACM, vol. 12, no. 1, pp. 23–41, 1965.

[Sterling & Shapiro 86] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, Cambridge, Mass., 1986.

[Warren 83] D. H. D. Warren, *An Abstract Prolog Instruction Set*, SRI Technical Note 309, October 1983.