What is Prolog-X?

W F Clocksin

Computer Laboratory

Prolog-X is a portable *design* for a high-performance Prolog system intended for large-scale applications. The terms *portable* and *high performance* may seem contradictory, but the intent is to try hard to maximise both. At the time Prolog-X was designed, the only high-performance system in existence was the (non portable) DECsystem-10 compiler by David Warren. With the impending demise of the DEC-10, together with the wish to run big programs on machines such as the VAX and 68000, I set out to design a system based on Lawrence Byrd's ZIP abstract instruction set, and having the following objectives:

- High performance. We would use the compiler technology introduced by Warren's implementation, together with some improvements devised by us. Improvements include: compiling byte codes for an abstract machine (the ZIP machine), which is potenially easily implemented by a choice of emulation, microcode, hardware, etc.; early checking for determinacy; 'lazy' stack allocation; tail recursion optimisation; no need for mode declarations; automatic indexing on the first argument of a clause.
- Portability. The DEC-10 system is impractical to port. We wanted a portable system, but were willing to sacrifice some performance. A system kernal would be written in a high-level language, and the rest of the system would be written in Prolog.
- Large applications. Needing to run large applications means that the design should use methods that scale up cheaply. Some existing Prolog systems use implementation techniques that lead to disproportional losses in performance as the size of an application grows.
- Improved user interface. Compiled procedures should 'look' the same as interpreted ones. In the DEC-10 implementation, clauses cannot be compiled incrementally and cannot be retracted or listed, for example. We would improve on this by having incrementally compiled clauses and sacrificing performance.
- Improved features. We would offer proper strings, garbage collection, floating point numbers, better error handling and hooks for future programming aids.

Starting around March 1982 I implemented as much as I could, time permitting, in Pascal on a VAX under VMS. Although I would have preferred to use the C language, Pascal was used for two reasons: (a) it was the only suitable language available to me at the time, and (b) the strong type-checking would help to catch blunders. The consequent effectiveness of reason (b) was astonishing. The implementation was intended to be a *runnable specification* rather than a finely-tuned *product*. The reason was that I needed an instrumented prototype to test the new ideas, and I wanted to make explicit the design's modularity so that future implementation decisions could be more rationally made. In any case, it was unlikely that a final implementation would be written in Pascal. The result, a nearly complete implementation[1], consisted of about 5000 lines of Pascal plus about 1000 lines of Prolog.

As expected, the execution speed is slow, approximately comparable with a popular Prolog interpreter written in C. The ZIP emulator (written in Pascal) executes about 6000 bytecodes per second on a VAX 780, and about double that on an ICL 2980. The slowness is for two reasons: (a) the Pascal compilers used emit poor code, and more importantly, (b) Pascal prevents the expression of certain low-level concepts in a way that can be effeciently executed by typical machine instructions. Translating the design into a more suitable language such as BCPL or C should realise a speedup of at least 5-fold.

An example of why Pascal is unsuitable is as follows. Consider the following declarations:

```
bytevect = packed array[0..3] of 0..255;     { overlay bytes per word }
overlay = 0..2;                          { number of variants }
word = packed record                         { machine word overlays }
  case overlay of
    0 : (raw : integer);
    1 : (byte : bytevect);
    2 : (val : 0..16777215; tag : 0..255);
  end;
var m : array[1..100000] of word;
```

Array m is considered to be a memory array of 32-bit words which can be accessed either as a tagged value, an integer, or a byte vector. To fetch the next bytecode instruction[2], the ZIP emulator must read a byte from the byte address PC: b := m[PC div 4].byte[PC mod 4]. Most Pascal compilers emit some 20-odd instructions for this statement, including multiplies and divides. In 'C' this would simply be b = *PC, and the compiler would emit the one or two instructions actually needed.

The last release was Release 1.6 (August, 1983). Since that time I have reconsidered many of my original design decisions in the light of experience. The relative importance of some features has been reevaluated. In particular, memory management must be reconsidered, and this is the subject of another

[1] Not implemented: full garbage collection, floating point arithmetic, and the newest DEC-10 Prolog built-in predicates.

[2] Fetching the next instruction is the most often used operation, an observation which is consistent with previously known von Neumann style architectures.

report. The remainder of this report describes what needs to be redesigned and reimplemented.

(1) *Cut must work within disjunctions.* At the moment, the 'cut' operation does not work properly within disjunctions. The compiler must be changed to compile cuts correctly inside disjunctions, and the ZIP emulator must be changed to implement the new instructions.

(2) *Redesign of clause heap.* Release 1.6 implements *clause* by decompilation. This is undesirable. An experimental alternative performs better under most conditions but cannot be prevented from occasionally producing dangling clause references. The real answer is to redesign the heap at the same time that GC is added to the global stack, perhaps combining the two. Another report discusses the problems raised here.

(3) *Garbage Collection of global stack in the absence of backtracking.* At one time considered unnecessary in principle, this is now considered essential to support the use of programming techniques (suspended processes, iteration by tail recursion, token scanning) that are becoming widely used. In fact, a general-purpose heap is considered desirable now, and performance must again be sacrificed.

(4) *Rewrite parser in Prolog.* The *read* predicate is implemented in Pascal. It is fast, but, like many Prolog parsers, cannot parse the full Prolog grammar. It is also bulky, and accounts for about 25 percent of the bulk of the Pascal code. The official Prolog parser is written in Prolog, and is freely available. It should replace the existing one, which will mean writing a tokeniser. The implementation of operators will be much improved as a side effect, but there will be a greater need for a garbage collector.

(5) *Compiled arithmetic operations.* Although compilation will be more expensive, execution speed will be faster and global stack space will not be required (for copying the expression).

(6) *Floating-point arithmetic.* The DEC-10 does not have this, but almost every other competitive system (C-Prolog, POPLOG) does. The ZipMachine design does this easily by having a 40-bit word, so the floating value can be held in the 32-bit val. For ZIP, we must use one of the several possible hacks: use global stack (then need GC), or use a separate area (not good).

(7) *Compilation into native CPU code.* Although emulation overhead is not particularly high, native code compilation can be used to "buy back" some of the performance lost when the new memory redesigns are implemented. The almost undeserved speed of the POPLOG system shows that this trick is effective. The new Prolog systems under development by Quintus apparently rely on generation of good native code. The ZIP codes are more suitable if the underlying hardware executes the ZIP instruction set.

(8) *Add debugging features.* There are several ways to offer this:

(a) An embedded interpretdr in Prolog. Given proper disjunction and GC, this should be adequate. *Clause* would need to be very fast.

(b) Automatic advising of goals as in InterLisp. Requires compiler change, seems awkward, but easy to interface to user trace code in Prolog.

(c) Hooks in the ZIP virtual machine. Probably impossible to interface to user trace code in Prolog.

(9) *Non-incremental compilation.* Currently clauses are compiled separately and linked together. increased performance can result by compiling a group of clauses as one indivisible unit, so there is less overhead for selecting the next clause in a procedure. This pays off substantially when compiling into a native code. This is what the DEC-10 system does. The penalty is that it is then impossible to retract clauses or to otherwise inspect clauses separately. This penalty is worth accepting for built-in predicates written in Prolog, because they cannot be inspected or retracted anyway, and would benefit by increased speed.