Design and Simulation of a Sequential Prolog Machine

W F Clocksin

Computer Laboratory

University of Cambridge

Corn Exchange Street

Cambridge CB2 3QG, England

Keywords: Prolog, sequential inference machine, compiler, virtual machine

ABSTRACT

Prolog-X is an implemented portable interactive sequential Prolog system in which clauses are incrementally compiled for a virtual machine called the ZIP Machine. At present, the ZIP Machine is emulated by software, but it has been designed to permit easy implementation in microcode or hardware. Prolog-X running on the software-based emulator provides performance comparable with existing Prolog interpreters. To demonstrate its efficiency, compatibility, and comprehensiveness of implementation, Prolog-X has been used to compile and run several large applications programs. Several novel techniques are used in the implementation, particularly in the areas of the representation of the `recordz` database, the selection of clauses, and the compilation of arithmetic expressions.

1. Introduction

The motivation and key principles behind the design of sequential Prolog (Clocksin and Mellish, 1981) machines are described in (Yokota, *et al.*, 1983). In this paper we describe an abstract Prolog machine, called the ZIP machine, which is suitable for implementation in software, microcode, or hardware. The main requirements of the design are that

- It should be portable.

- The Prolog syntax should be compatible with DECsystem-10 Prolog (Warren, 1979), a *de facto* standard. Unlike the DECsystem-10 implementation however, there should be no perceived operational difference between compiled clauses and interpreted clauses.
- It should imply a high-performance implementation suitable for large-scale commercial and industrial applications.

These conflicting requirements have led to the design of a system which is a reasonable compromise. The ZIP machine forms the basis of a working Prolog system, called Prolog-X, which has been implemented twice in different computer languages, and which has been transported to several operating systems and three different versions of Unix.

Prolog-X contains a resident compiler which incrementally compiles Prolog clauses into compact bytecoded instructions which are then emulated by software. Using the software emulator, the ZIP machine runs Prolog programs at a speed similar to purpose-built Prolog interpreters, but slower than if programs were compiled into native machine instructions. The primary intention of the software emulator is that it should serve as a model for a microcode or hardware implementation from which one can expect higher performance.

Two fundamental principles have guided the design of the ZIP machine:

- Despite decreases in the price of memory, compact representations of code and data will continue to be important for languages such as Lisp and Prolog, which generally have poor locality of reference. Compactness should improve locality, which in turn should improve interaction with caches and paging.
- It is important to use low-overhead implementation techniques which "scale up" well. The performance of the system must not decrease dramatically and disproportionally as the size of the application program increases.

Consequences of these principles will be discussed below.

The ZIP machine uses a similar principle of operation as the DECsystem-10 Prolog system of Warren (1977), modified to incorporate tail recursion optimisation (Warren, 1980). The main differences are:

- The use of copying instead of structure-sharing as the means of constructing compound terms.
- Choice points are created only when required instead of at every procedure call.
- Potentially global variables in the final goal of a clause are made global only if required at run-time, instead of by default at compile-time.
- Extra checks are made at run-time to detect determinate computations as early as possible.

Many aspects of the ZIP machine are similar to a new Prolog machine design independently proposed by Warren (1983). The first implementation of ZIP predated Warren's new proposal, and specifies the design of some components, such as arithmetic, which are not covered in Warren's new proposal. In addition, the Prolog-X system built around the ZIP machine removes several restrictions imposed by the DECsystem-10 Prolog system:

- The user perceives no difference between compiled clauses and interpreted clauses (except for performance). For example, any clause in the database may be examined and modified.
- Database references from one clause to another are permitted.

The ZIP machine is defined by a word format, a set of registers, the format of storage areas, an instruction set, and assumptions about the layout of data structures in memory. Prolog data-structures are represented by a tagged 32-bit word, and Prolog clauses are represented as sequences of 8-bit bytes. In the Prolog-X system, a resident compiler (written in Prolog) incrementally compiles clauses into a compact bytecode sequence. Each instruction consists of a one-byte operation code followed by up to two arguments.

The bytecode emulator consists of a number of small routines that define the different operations. Some instructions can be executed in three different modes, so there is a separate routine for each mode.

The first version of the Prolog-X system was written in Pascal under VMS for the DEC VAX in 1982. It was then ported to the ICL 2980 under VME, and the

bytecode emulator was subsequently translated into the VME systems programming language S3. The first version was also used as the basis of a separate Prolog implementation design study (Bowen, Byrd, and Clocksin, 1983). The second (and current) version of Prolog-X is a translation of the first version into the C language. This version runs on the following machines: the ICL Perq under PNX (similar to System III Unix), the HLH Orion (made by High Level Hardware, Ltd.) under Berkeley Unix 4.1, the DEC VAX under Berkeley Unix 4.2, and the IBM 3081 (370 architecture) under MVS.

To demonstrate its efficiency, compatibility, and comprehensiveness of implementation, Prolog-X has been used to run Chat-80 (Warren and Pereira, 1982) and PRESS (Sterling, *et al.*, 1982), two large programs written originally for DECsystem-10 Prolog.

A study is currently in progress to estimate the feasibility of microcoding the HLH Orion to emulate the ZIP Machine. The HLH Orion is a 32-bit microprogrammed processor with performance comparable with a DEC VAX-11/750. Preliminary estimates suggest a speed ranging from 15K to 25K LIPS, depending on the extent of the emulation and on properties of the particular program executed.

2. Data Words

Every data structure is represented by a $(t + v)$-bit word which is divided into a *tag* field of t bits in length, and a *val* field of v bits in length. The *tag* field is ideally 8 bits long, but Prolog-X uses only 4, encoding unique tags as unique 4-bit integers. With more bits available, a redundant coding could be used to speed up certain tag tests. The *val* field is ideally 32 bits long so that single-precision floating point numbers can be represented directly. Prolog-X uses a 28-bit val and does not implement floating point numbers at all. Integers in the range $-2^{28} \ldots 2^{28} - 1$ are represented directly in the *val*, and have tag `INT`. The 16 primitive tagged objects are as follows:

Tag	Mnemonic	Val	Purpose
0	`INT`	integer value	integer
1	`FLOAT`	float value	single-precision float

2	BOX	pointer to a BLOCK cell	byte string
3	ATOM	pointer to an atom cell	atom
4	TERM	pointer to a compound term	compound term
5	CONS	pointer to a list cell	list cell (special case)
6	LINK	pointer to a variable binding	instantiation
7	UNDEF	pointer to self	unbound variable
8	FUNCTOR	pointer to a functor cell	functor information
9	BLOCK	byte count	header of null-padded byte string
10	EMPTY	unused	to catch bugs
11	TERMIN	unused	terminate internal chains
12	CLAUSE	pointer to clause cell	clause information
13	TABLE	component count + 1	header for word table
14	TABREF	pointer to TABLE cell	vector
15	PROC	pointer to a procedure cell	procedure

Except for one special case, the compound term of arity n is represented by a TERM pointer to a cell of $n+1$ words. The first word of the cell is a FUNCTOR-tagged pointer to the appropriate functor cell. The components of the term follow in the subsequent words. The compound term of arity 2 having functor '.' is commonly used as a list constructor, so a special case representation is provided. To represent a list cell, a CONS-tagged word points to a cell of two words, which correspond to the head and tail of a list. The alternative method used in some implementations is to represent a list cell as a compound term of arity 2, using a TERM-tagged pointer to a cell having at least three words (functor, head, and tail).

Some of the above tags are not strictly necessary. It is possible to represent procedures, clauses, tables, and functors as compound terms or tables. This would provide a more uniform and simple set of data structures, but would cost more in memory fetches. For example, to test whether something is a clause, we test the tag, which is very fast. Representing a clause as a compound term would mean testing for a clause by fetching and testing its header (a functor pointer).

3. Layout of Data Object Cells

In common with other cell representations used in LISP and POP-2, multiword storage cells are accessed by a tagged pointer to the first word in the cell. The most obvious difference is that LINK pointers are allowed to point directly to words within a cell. Although constants can be represented as functors of arity 0, ZIP does not use this convention; a functor of arity 0 is consistently represented as an atom, preventing unnecessary construction of functor cells. Any unused component of a cell is occupied by a TERMIN word.

We now show the layout of each data object understood by ZIP. We do not suggest that these layouts have been optimally designed. In particular, some of these objects contain extra fields for debugging and diagnostic purposes not connected with the execution of the ZIP machine. For example, given a compiled clause cell, it is possible to follow back a chain of pointers from its procedure to its functor to its atom, for the purposes of printing its name for diagnostic purposes.

ATOM. The ATOM word points to a three-word cell.

(1) A hash chain, continued by an ATOM pointer or terminated by a TERMIN word.

(2) A functor chain, continued by a FUNCTOR word or terminated by a TERMIN word. All functors having the same name but different arity are linked into this chain.

(3) A BOX-tagged word pointing to the byte string block containing the atom name.

FUNCTOR. The FUNCTOR word points to a cell of four words.

(1) An ATOM word to the atom naming this functor.

(2) An INT word containing the arity of the functor.

(3) A FUNCTOR word pointing to the next functor cell of any other functors of the same name having different arities.

(4) A PROC word pointing to the procedure cell, if any.

PROC. The PROC word points to a cell of six words:

(1) An INT word containing various flags.

(2) An ATOM word naming the module in which the procedure is defined.

(3) An ATOM word naming the module in which the procedure is visible.

(4) A FUNCTOR word pointing to the functor associated with this procedure.

(5) A PROC word pointing to the next procedure with the same functor, but with different module characteristics.

(6) The clauses for this procedure. For Primitive Procedures, this is an INT word identifying the particular Primitive. For procedures defined by Prolog clauses, this is a TABREF word pointing to a two-component table. The first component is a CLAUSE word pointing to the first clause; the second component is a CLAUSE word pointing to the last clause.

CLAUSE. The CLAUSE word points to a cell which represents a single Prolog clause. Clauses are bidirectionally chained so that inserting and deleting clauses from the chain can be performed in constant time. A clause is at least seven words in length. The eighth and subsequent words are any Prolog data words, which collectively are called the *external references table* for the clause, and are used as references to data structures used by the ZIP instructions of the clause. Entries in the external references table are accessed by an offset from the XC register, which always points to the clause cell of the currently executing clause. The first seven words of the clause cell are:

(1) An INT word contains various flags.

(2) A word used as a search key when searching clauses in the database. The key is related to the first argument of the clause. The key is the same as the first argument for constants; is an UNDEF word for variables or no first argument; is the principal functor for compound terms; it is a CONS-tagged word (having a don't care *val* field) for list constructors.

(3) A PROC word pointing to the procedure owning this clause.

(4) A BOX word pointing to the byte code block containing the ZIP instructions for this clause. Byte code blocks contain only instructions and arguments. Arguments are only literal values or offsets from a register, hence, no tagged words appear as arguments. The reason for this is so that instructions can be easily decoded on byte-boundaries, and so that is it never necessary to scan the code block to find references (as required for garbage collection). Instead, tagged words appear in the external references table, and instructions access

these using their argument as an offset into the table. An additional advantage is that external references are represented uniquely, thus improving compaction.

(5) An INT word containing the total size of the clause cell in words.

(6) Either a CLAUSE word pointing to the previous clause, or a TABREF word.

(7) Either a CLAUSE word pointing to the next clause, or a TERMIN word.

TERM. A compound term of arity n is represented by a TERM pointer to a cell of length $n+1$. The first word of the cell is a FUNCTOR pointer to the appropriate functor cell. Subsequent words are the components of the compound term.

BOX. The byte string of length n bytes is represented as a BOX word pointing to a block cell of length $\lfloor (n-1)/4 \rfloor + 2$ words. The first word of the block cell has *tag* BLOCK and *val* n. Subsequent words contain bytes packed four to the word, null-padded to the nearest word. The padding is required because boxes are compared with each other word-at-a-time.

TABREF. The vector of length n is represented by a TABREF word pointing to a table cell of $n+1$ words in length. The first word of the table cell has *tag* TABLE and *val* $n+1$. The subsequent n words contain the n components.

4. Registers

The current state of a computation is contained in a set of registers, most of which point into the storage areas discussed below. The registers having contents valid at all times are:

PM	processor mode
XC	current clause pointer
D	current data pointer
PC	current program counter
L	current (target) local frame
CL	current (source) local frame
CP	forward continuation program counter
CL0	forward continuation local frame
G	global stack allocated top

G0	global stack committed top
H	heap freelist
BL	backtrack continuation local frame
BG	backtrack global stack top
BP	backtrack continuation clause
TR	trail allocated top
TR0	trail committed top

Some of the above registers are redundant in that they are caches for slots in the current local frame. Other registers not mentioned here are scratchpad registers whose contents are valid only during the execution of a single ZIP machine instruction.

5. Storage Allocation

Storage is allocated in four main areas, although small scratchpad stacks are used for other housekeeping within primitive procedures not a part of the ZIP machine. The four areas are summarised here, and more detailed discussion is given below.

- Activation records are allocated on the Local stack, which is implemented as a true stack with contiguous storage but allowing indexing into it. Local stack frames do not require garbage collection, as this is done automatically as a result of certain ZIP instructions which control tail-recursion optimisation. Variable slots in the local frames are the source of all roots to data structures. Variable slots are not initialised upon activation, because the first use of a variable is identified by the compiler, and appropriate action is taken by the generated instruction.
- The Global stack, in which most temporary data structures are allocated, should also be a true stack, but it is also necessary to index from arbitrary pointers into the stack. Space is automatically recovered on backtracking, although garbage collection is nowadays considered necessary to recover space in situations where backtracking can never occur. These issues are discussed below.
- Persistent data structures are allocated in the Heap, which should be implemented as a heap. Allocations and deallocations are programmed explicitly

(by using `assert(X)`, `retract(X)`, and other more primitive predicates). A simple reference-bit garbage collection scheme discussed below is used. The ZIP machine sees the heap only in that registers PC, CP, XC, and BP point into it.

- The Trail is an historical record of variable instantiations. When certain variables are instantiated, a pointer to the variable is entered on the Trail so that the variable can be reset when backtracking. Only variables occurring previous to the current backtracking point are trailed. The trail also holds other information of a chronological nature required for garbage collection of clauses. This is also a true stack, with pushes and pops being done by the ZIP machine.

In addition, the ZIP machine uses a small scratchpad stack when executing arithmetic instructions, and when executing code between the `functor` and `pop` instructions (see below).

Local stack frames, called activation records, are offset from register CL. The order of the first eight entries is unimportant but must be consistent. A complete stack frame stores the following entries, although in some cases (determinacy), not all register save entries are used. An ordering could be imposed to increase speed, as registers CP, CL0, XC, and G0 are saved or restored at the same time by several of the machine instructions. An activation record, in order of increasing address, is laid out as:

(reserved)
continuation program counter
continuation local stack frame
backtrack clause pointer
global stack pointer
backtrack local stack pointer
trail pointer
current clause pointer
(arguments and local variables)
(temporary local variables)

The argument slots hold the actual parameters of the procedure call. If a variable appears at the top-level in the head of a clause, then its value is simply that of

the corresponding actual parameter, and there is no need to allocate a variable slot for it. Locals classified by the compiler as temporaries are allocated nearest the top of the stack, so that the stack space occupied by temporaries can be recovered automatically when the neck of a clause is executed.

6. ZIP Instructions

A complete list of the ZIP instructions is given in the Appendix. The ZIP Machine is always in one of three states called Processor Modes: `ARG`, `COPY`, or `MATCH`. Many of the instructions described here have alternative interpretations depending on the current Processor Mode. Some of the instructions switch the Processor Mode.

Each instruction is encoded by a byte containing a number 0...63, which is combined with the contents of the PM register (2-bit Processor Mode) to form an 8-bit operation code. Arguments, if any, follow in the subsequent zero, one, or two bytes. Arguments of an operation code are of three different types:

- An integer in the range 0...255. This is encoded directly as an argument byte.
- A procedure argument or variable within the current clause. All procedure arguments and variables are found in the current stack frame offset from register CL, so the argument is encoded as an offset from CL.
- A functor, procedure, or constant. These are known as external references, and data words referring to them are stored in an area of the clause cell known as the external references table. Register XC always points to the current clause cell, so such arguments are encoded as an offset from XC.

The instructions described here are generated by an optimising compiler written in Prolog, and residing in the Prolog-X system. The compiler is capable of identifying cases where full unification is not required, where data structures should migrate to other areas, where tail recursion is used, where unit and doublet clauses require less housekeeping, and where certain built-in predicates are translated directly as ZIP instructions (instead of generating calls). Special-purpose instructions are generated in these cases. The principles on which these optimisations are founded is discussed in Warren (1977). There are no instructions defined for

handling disjunctions. Disjunctions are interpreted using the `callx` instruction, which handles interpreted calls to the "cut" predicate correctly.

Before describing each instruction individually, we shall first show how instructions are generated from Prolog clauses. Suppose we are given terms $t_1, \ldots, t_l$, $u_1, \ldots, u_m$, $v_1, \ldots, v_n$, and predicate symbols p of arity l, q of arity m, r of arity n, not all necessarily distinct. Then the clause

$$\texttt{p}(t_1, \ldots, t_l) \texttt{ :- } \texttt{q}(u_1, \ldots, u_m), \texttt{ r}(v_1, \ldots, v_n)$$

compiles into the sequence (here shown unoptimised):

code for t_1
⋮
code for t_l
`enter`
code for u_1
⋮
code for u_m
`call` x
code for v_1
⋮
code for v_n
`call` y
`exit`

The `enter` instruction marks the division between the head and body of the clause, and creates a choice point if necessary. This instruction employs an argument, not shown, giving the size of the local stack frame. Each `call` instruction has byte argument which refers to an entry in the external references table which is a reference to the required procedure. Finally, `exit` marks the end of the clause, and passes control to the forward continuation.

Among the optimisations used here are translation of `call` `exit` sequences into a `depart` (or possibly `proceed`) instruction which handles "last call" optimisation (the general case of tail recursion optimisation). A discussion of this issue appears in Warren (1980).

The compilation of terms proceeds as follows:

- If the term is atomic, it is translated as `constant n`, where n is an offset from register XC used for addressing the external references table. The term is inserted into the nth entry of the external references table.
- If the term is a variable, it is translated as `var n`, where n is an offset from the current stack frame register CL.
- If the term is compound (consider the compound term $\mathtt{t}(a_1,\ldots,a_n)$, where each a_i is a term), then it is translated as:

```
functor f   n
code for a1
:
code for an
pop
```

The `functor f n` instruction refers to an offset f from XC to an entry in the the external references table pointing to the functor cell for the function symbol `t`. The arity n is included as an argument of the instruction for efficiency. Code for the n arguments and a housekeeping instruction follow the `functor` instruction.

The ZIP instruction set listed in the Appendix contains many optimised variants of the above instructions, which are all compiled by the Prolog-X resident compiler. For example, integer constants $0 \leq n \leq 255$ compile into `immed n` instead of taking up external reference space. The constant `[]`, used to represent the null list, compiles into `constnil` for the same reason. The functor '.' of arity 2, used as a list constructor, compiles into `conslist` instead of `functor n 2`, where n would be an XC offset. There are many variants of the `var` instruction, generated as a result of a flow analysis of the clause as it is compiled. Details of the general idea are given in Warren (1977), and are relevant despite the fact that Warren uses structure-sharing to represent data structures. A complete list of the instruction set is given in Appendix A.

We now give an example of the ZIP instructions into which the Prolog `concatenate` predicate compiles. Associated with each clause is a clause cell as previously described, which contains the external references table. The code sequences shown here constitute only the code body, and are each only a few bytes in length. The

`concatenate(X,Y,Z)` goal succeeds when the list `X` concatenated with the list `Y` gives the list `Z`:

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

The ZIP "assembly language" into which **concatenate** compiles is as follows. Two code sequences are shown, one for each clause of **concatenate**. Many of the instructions shown are optimised variants of those introduced above.

`constnil`	*unify with* `[]`
`skipvar`	*an optimisation of the first variable L*
`var 9`	*...because this one does the work*
`return`	*successfully pass control to continuation, no subgoals*
`conslist`	*unify the list constructor*
`firstvar 13`	*unify the head (first occurrence of variable X)*
`firstvar 11`	*unify the tail (first occurrence of variable L1)*
`pop`	*finished with that term*
`skipvar`	*optimisation for variable* L2
`lastconslist`	*unify list constructor, no* `pop` *needed*
`var 13`	*head* X
`firstvar 12`	*tail: first occurrence of* L3
`argmode`	*an optimised* `enter` *(only 1 subgoal)*
`var 11`	*push first argument L1*
`var 9`	*push second argument L2*
`var 12`	*push third argument L3*
`proceed 1 3`	*tail-recursive call of* **concatenate**

The final `proceed` instruction takes a pointer to the procedure cell for **concatenate** from the first entry in the external references table, and performs a tail-recursive call of three arguments, reusing the current stack frame.

The actual operation performed by each instruction depends on the context provided by the PM register. During the execution of the head of a clause, the processor is in MATCH mode, and each instruction attempts to unify its argument with

the arguments of the goal. Within the code for a term, data structure construction must be performed, so the processor switches to COPY mode so that each instruction can construct its argument by copying. During the execution of the body of a clause, the processor is in ARG mode, and each instruction passes its argument to the subgoal to be called. Maintaining the correct contents of PM is a simple task performed by some of the instructions. For example, `enter` and its optimised form `argmode` assign mode ARG to PM. The `functor` instruction pushes the contents of PM on a scratchpad stack and sets PM to COPY. The matching `pop` instruction restores PM from the scratchpad.

The current version of the ZIP machine calls for byte-coded instructions as shown in the Appendix. As an experiment I also implemented a word-coded machine to compare relative performance. A word-coded machine stores instruction and arguments in successive machine words, word-aligned. Because word-sized arguments can therefore fit in the instruction code body, the external references table is not required for execution; nevertheless, the external references table has been retained because the reference-counting garbage collector used to maintain the clause database also requires the information held in it. The supposed advantages of the word-coding scheme are to obtain the improved performance possible on many commercial processors by fetching only word-aligned data, and to remove the indirection overhead of accessing arguments through the external references table. Furthermore, a word-sized instruction could be represented as the starting address of the emulator definition of the instruction, further reducing the instruction dispatching overhead by means of the so-called "threaded code" technique.

Our results for the word-coded machine (which did not use threaded code) follow. First, program size increased by about fifty percent. For example, the size of the initial heap image (consisting of that part of the Prolog-X system written in Prolog) increased from 64352 bytes to 98616 bytes. Speed of execution (over a range of programs including CHAT-80) increased by less than five percent on an HLH Orion processor. This was less of an increase than expected, and can be explained by the following *post facto* reasons. First, byte-fetching is not significantly slower on the Orion processor, which fetches more than one word at a time and contains

a hardware byte shifter. Most of the use of the external references table is for referring to other procedures; fetching the external procedure reference is however an insignificant fraction of the total amount of time required to emulate procedure calls.

It remains to be seen the extent to which instruction dispatch can be reduced by adopting the threaded code technique. The present implementation (in C) uses the `switch` statement, which compiles into an indexed table; instruction dispatch time can be perhaps halved by using threaded code, but portability would be sacrificed.

7. Selection of Clauses

As mentioned above, clauses are compiled separately and are bidirectionally chained. This technique provides easy incremental compilation and modification of the database. However, there are some disadvantages of this technique compared with the alternative (Warren, 1983) of compiling entire procedures at a time. When an entire procedure is compiled as a unit, changes of context caused by unification failure can be performed more quickly. More information is available for the purposes of indexing, and it is possible in principle to share common subexpressions in clause heads so that unification need not be restarted every time an alternative clause is selected.

In order to compensate for the slower context changing in the ZIP model, we use two techniques to improve clause selection. The first technique, which is used in other implementations, is to associate a search key with the first argument of a clause. This acts as a "filter" to reduce the amount of futile matching. The second technique works as follows. If a clause has been selected, the system then "looks ahead" in the procedure for another clause that might match the goal according to the search key. If no further clause matches, then we have discovered that the selected clause can be executed deterministically (no choice point is necessary). If another clause matches, however, the backtracking pointer is then updated to point to the clause. It is worthwhile to spend the time looking ahead, as it is likely that the search will be required anyway.

The combination of the two techniques allows, for example, deterministic execution of the `concatenate` predicate regardless of the order of clauses, and renders superfluous the "cut" goal which is sometimes written for the empty-list case of `concatenate`. The techniques gain more purchase on larger programs: we estimate that the number of stacked choice points is more than halved when running programs such as Chat-80.

8. Arithmetic Instructions

The Prolog goal `is(X,Y)` is defined to consider the term bound to `Y` as an arithmetic expression, and to evaluate that term according to the rules of arithmetic, and to unify the result with `X`. This goal is normally written in infix form as "`X is Y`". Consider the following goal, which shifts `X` left by eight bits and adds `Y`, unifying the result with `Z`:

```
Z is X << 8 + Y.
```

The usual interpretation of arithmetic expressions in computer languages admits `X` and `Y` to be bound to numbers. However, evaluation of arithmetic expressions in Prolog is complicated by the fact that `X` and `Y` are allowed to be bound to arbitrary compound terms denoting arithmetic expressions. Whether a variable is bound to a number or a compound term can only be detected at run-time, so the usual methods for generating code for arithmetic expressions are not sufficient. Most (if not all) Prolog interpreters implement '`is`' correctly, but inefficiently, evaluating the entire arithmetic expression at run-time by recursive descent. This evaluation consumes global stack space which is recoverable only by garbage collection or backtracking. Furthermore, common expressions bound to the same variable are evaluated redundantly (for example, consider `X is Y << 8 + Y`, where `Y` is bound to an expression). Another strategy is adopted by the DECsystem-10 Prolog compiler, which does not completely implement '`is`', giving a run-time error when variables in arithmetic expressions are bound to compound terms. This is one way in which the DECsystem-10 Prolog compiler implements a different language than the DECsystem-10 interpreter does, and such a difference is considered awkward.

The solution taken by Prolog-X is to generate a sequence of instructions which tests the tag of the word bound to each unique variable in the expression. If the value is an integer, then execution proceeds normally, using the value in the subsequent calculation. If the value is a non-integer, the ZIP machine performs a call to a procedure, written in Prolog, which recursively evaluates the expression and passes the result back. The instruction for performing the run-time test is `eval` v_i v_j, which for variables (local stack offsets) v_i and v_j, evaluates the term bound to v_i, placing the resulting value in v_j. During the execution of the `eval` v_i v_j instruction, if the word at stack offset v_i is not an integer, then the term `is`(v_j, v_i) is constructed on the global stack, and control proceeds to the `callx` instruction.

Arithmetic instructions are compiled into stack (zero address) instructions. The ZIP Machine is equipped with a scratchpad stack used by the `functor` family of instructions during the unification of compound terms. This stack also serves as the operand stack during the evaluation of arithmetic expressions. Some care must be observed in dealing with this stack: the stack contents are not guaranteed over Prolog procedure calls. Thus, `eval` instructions, which could cause procedure calls, must be generated before the instructions in the goal which use the stack. This constraint does not unduly complicate the compiler. Instead of the usual compilation schema applied to each node of the expression parse tree

generate code to push arguments on stack
generate code for operator

the Prolog-X compiler first generates `eval` instructions for all previously unevaluated variables in the current goal being compiled, and then compiles the goal using the usual schema.

Here is an example of code generated from arithmetic expressions. We shall use mnemonic variable names to denote the byte quantities (offsets from the CL register) which are actually generated:

`eval X X'` *evaluate* X, *move result to* X'.
`eval Y Y'` *evaluate* Y, *move result to* Y'.
`pushv X'` *push what* X *evaluated to.*
`pushb 8` *push the byte 8.*

```
shl          pop arguments, perform shifting, and push result.
pushv Y'     push what Y evaluated to.
add          pop arguments, perform addition, and push result.
result Z     pop stack, unifying result with variable.
```

Although `eval` instructions are emitted at the beginning of a goal, only one `eval` instruction is generated per unique variable appearing in the entire clause. This prevents duplicate evaluation of common subexpressions bound to variables. More generally, `eval` instructions are not emitted for any variables in the entire clause known to be instantiated to integers. This of course applies to the variables found as the left-hand argument of '`is`' goals. Furthermore, if a variable known to be bound to an integer appears as the left-hand argument of an '`is`', then code for an numerical equality test is generated instead of the more general (but in this case unnecessary) `result` instruction. For example, the conjunction of goals

```
Y is 3 * X + Z, Z is X - W
```

compiles into:

```
eval X X'
eval Z Z'
pushb 3
pushv X'
mul
pushv Z'
add
result Y
eval W W'    in the second goal, only W requires possible evaluation.
pushv X'
pushv Y      Y is known to be an integer (from previous goal).
sub
pushv W'
add
pushv Z'     known to be an integer,
eq             so, simply test it for equality.
```

There is a considerable increase in execution speed for those arithmetic expressions from which ZIP instructions can be generated. Speed increases by a factor ranging from 10 to 50 compared with the interpretation of expressions by recursive descent with a Prolog program.

9. Discussion of Storage Allocation

9.1 Local Stack

When a goal succeeds determinately, its local frame is discarded. If the procedure is determinate at the point where the last goal in the body of the clause is about to be called, then the frame for that goal replaces the frame for the procedure. This is how tail recursion optimisation is implemented. One problem is as follows. Suppose a goal replaces a frame that has variables that refer to the goal. In this special case, which is detected during compilation, space for the affected variables is migrated to the global stack. The instructions generated to perform this task are `glofirvar` and `glovar` (see Appendix A). Migration may not in fact be necessary, and a run-time check determines this.

9.2 Global Stack

As mentioned above, garbage collection is required for the global stack only when inaccessible structures are created in the absence of backtracking. The current implementation of Prolog-X does not perform garbage collection in the absence of backtracking. If it is necessary to garbage collect the global stack, then a normal mark-sweep-reallocate algorithm can be used. References to data in the global stack are rooted in the local stack variables. A refinement of the usual algorithm recognises that it it not strictly necessary to mark accessible structures if it is known that the local variable will not be used subsequently in the current goal. This has the effect of reclaiming much space that normally would not become inaccessible until a determinism has been committed. Other relevant issues are discussed below.

9.3 Heap

Clauses and database entries are stored in the heap using the **assert** and **recordx** predicates, respectively. They are incrementally garbage collected after they have been **erased**, but only after they are not being used in a proof (a running program), and after no database references to them exist. Two bits for each clause are required for this purpose: a CLAIMED bit and a DOOMED bit. If a clause is being used in a proof, then the CLAIMED bit is set. When backtracking proceeds to the point at which the clause is no longer used, then the CLAIMED bit is cleared. It is at this point the clause can be deallocated if it has been **erased** during its use in the proof. The chronologically first claim of a clause is sufficient for it to remain claimed until the chronologically last use of a clause is discarded. This property results from the strict depth-first execution model used in Prolog-X procedure calls. In addition, database references to clauses are permitted, and a reference count field is used for this purpose. The reference counting system meshes conveniently with the claiming system, and full details of the incremental garbage collection method used are discussed by Clocksin (1984).

Compilation of clauses loses information about the source form of a clause. In DECsystem-10 Prolog, this has the effect of preventing the user from performing database operations (such as **retract** and **clause**) on compiled clauses, and this is another awkward difference between compilation and interpretation. In Prolog-X we get around this problem by compiling the input clause, and then compiling a unit clause of predicate **source**, in which the input clause appears as one of the arguments of the head of **source**. The advantage of this is that source terms are compactly represented as the ZIP machine codes necessary to construct a copy of the source term. Clause searching and matching is done by the usual mechanisms of the ZIP machine, and database references are used to link between a compiled term and its compiled source. The result is a very convenient and efficient implementation permitting full access to compiled clauses *via* looking up its source term.

Because clauses are compiled into the database, the time taken to execute a **record** is greater than if clauses were copied as terms into the database. For this reason, a small, fast compiler, which does no optimisations, is used for **recording**

instead of the usual compiler (used for **asserting**). Having unoptimised code, which may occupy twice the space of optimised code, is justified by the temporary nature of **recorded** terms. It must also be borne in mind that access cost (executing **recorded**) is greatly decreased by our method, because the benefits of compilation are exploited. This technique is fully described in Clocksin (1984).

10. Problems with Storage Management

The strict Prolog execution model permits the use of stack-like memory management, with which storage is reclaimed immediately upon backtracking. This property confers a number of advantages. Under usual circumstances, the need for costly allocation and garbage collection of cells from a heap is obviated. Also, most heap garbage collectors do not allow pointers to within cell bodies, thus preventing the most efficient way of implementing Prolog variable bindings. Furthermore, allocation from stacks improves locality of reference.

Despite the advantages of stack-like storage management used by Prolog-X, there are two problems with this method. First, certain Prolog programming techniques rely on never backtracking, using Prolog goals to simulate perpetual processes which communicate *via* shared variables. Such techniques (Shapiro, 1982; Warren, 1982) are more popular now than when Prolog-X was first designed, and are likely to form the foundation of future applications written in Prolog. Prolog-X however relies on backtracking to reclaim any redundant data structures constructed, and because no further garbage collection is performed, Prolog-X is unable to reclaim such structures in the absence of backtracking. Implementing a garbage collector for the global stack is only a short-term measure which does not counter the second problem.

The second problem arises from another recent trend, that of mixed-language programming with shared data structures. With systems such as Poplog (Mellish and Hardy, 1982) and LM-Prolog (Kahn, 1983), it is possible to share data structures and to call procedures between Lisp and Prolog programs, and this leads to a very attractive programming methodology. Such systems are most conveniently implemented by the use of a common virtual machine together with a common

garbage collected heap. The ZIP machine, on the other hand, is tailored so exclusively to the stack-like execution model of Prolog that it would be impracticable to compile Lisp programs, say, to run under the ZIP machine. Perhaps a future system could be designed around a heap-based common virtual machine, considering carefully the interaction with long-term and short-term data structures, and with incremental garbage collection.

Acknowledgements

The work reported here owes much to preliminary design effort in collaboration with Lawrence Byrd, a discussion with David Warren, and to much good advice contributed by Richard O'Keefe and Arthur Norman. Transporting the first version of Prolog-X to the ICL 2980 under VME was done by Paul Cager, Alan Reiblein, and Jim Doores of ICL. Transporting the second version of Prolog-X to the IBM 3081 was done by Alan Mycroft.

References

Bowen, D L, Byrd, L M, and Clocksin, W F, 1983. A portable Prolog compiler. *Proceedings of the Logic Programming Workshop*, Albufeira, Portugal.

Clocksin, W F, 1984. Implementation techniques for Prolog databases. Computer Laboratory, University of Cambridge. To appear in *Software—Practice and Experience.*

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

Kahn, K, 1983. Unique Features of LISP Machine Prolog, UPMAIL Report 14, University of Uppsala, Sweden.

Mellish, C S, and Hardy, S, 1982. Integrating Prolog in the Poplog Environment, Cognitive Science Research Paper 10, University of Sussex.

Shapiro, E Y, 1982. A subset of Concurrent Prolog and its interpreter, Technical Report TR-003. Institute for New Generation Computer Technology, Tokyo.

Sterling, L, Bundy, A, Byrd, L, O'Keefe, R, and Silver, B, 1982. Symbolic reasoning with PRESS, in *Computer Algebra* (ed J Calmet), *Lecture Notes in Computer Science 144*, Springer-Verlag.

Warren, D H D, 1977. Implementing Prolog – compiling logic programs. Research Reports 39, 40. Department of Artificial Intelligence, University of Edinburgh.

Warren, D H D, 1979. Prolog on the DECsystem-10, in *Expert Systems in the Micro-electronic Age* (D Michie, editor), Edinburgh University Press.

Warren, D H D, 1980. An improved Prolog implementation which optimises tail recursion, *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary.

Warren, D H D, 1982. Perpetual processes – an unexploited Prolog technique, *Logic Programming Newsletter*, **3**, 2.

Warren, D H D, 1983. An abstract Prolog instruction set. Technical Note 300. SRI International, Menlo Park, California.

Warren, D H D, and Pereira, F C N, 1982. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics* **8**, 110-122.

Yokota, M, Yamamoto, A, Taki, K, Nishikawa, H, and Uchida, S, 1983. The design and implementation of a personal sequential inference machine: PSI. *New Generation Computing* **1**, 125-144.

APPENDIX

A complete list of the ZIP instructions is given in the table below. Each instruction is followed by zero to two arguments. Arguments are byte-sized, and are interpreted in one of three possible addressing modes, denoted *n*, *a*, and *v* in the table.

n The argument is a ZIP word, located as the *n*th word of the external references table of the current clause (addressed by register XC).

a The argument is the byte *a*, not sign-extended. In the **immed** and **pushb** instructions, *a* forms the lowest significant eight bits of an INT-tagged ZIP word.

v The argument is a Prolog variable, located as the *v*th variable in the current local stack frame (addressed by register CL).

Mnemonic	Opcode	Arguments	Description
constant	11	*n*	unify constant
immed	27	*a*	unify constant small integer
constnil	35		unify the constant '[]'
functor	9	*n a*	unify functor of arity *a*
lastfunctor	10	*n a*	unify last functor in a nested term
conslist	33		unify the functor '.' of arity 2
lastconslist	34		unify the last functor '.' of arity 2 in a nested term
void	7		unify an anonymous variable
voidn	32	*a*	unify *a* consecutive anonymous variables
skipvar	8		unify first occurrence of an argument variable
skipvn	31	*a*	unify *a* consecutive first occurrences of argument variables
firstvar	5	*v*	unify first occurrence of local or temporary variable
glofirvar	29	*v*	unify first occurrence of variable in last goal
glovar	30	*v*	unify subsequent occurrence of variable in last goal
var	4	*v*	unify subsequent occurrence of variable
pop	1		complete a compound term
popmatch	2		complete a compound term in the head of a clause
poparg	3		complete a a compound term in the body of a clause
proceed	18	*n a*	call the only goal in a clause
depart	16	*n a*	call the last goal in a clause
call	17	*n*	call a goal
callx	26	*v*	convert a term to a goal and call it
return	13	*a*	the neck of a unit clause
argmode	14	*a*	the neck of a doublet clause
enter	12	*a*	the neck of a clause
exit	25		end a clause having an open-coded last goal
cut	15	*a*	compiled '!'
fail	19		compiled `fail`
provar	20	*v*	compiled `var(X)`
prononvar	21	*v*	compiled `nonvar(X)`
proatom	22	*v*	compiled `atom(X)`
proint	23	*v*	compiled `integer(X)`
proatomic	39	*v*	compiled `atomic(X)`
prosimple	40	*v*	compiled `simpleterm(X)`
succ	24		compiled `succ(X,Y)`
proarg	36		compiled `arg(X,Y,Z)`
profunctor	37		compiled `functor(X,Y,Z)`
proequal	38		compiled `X = Y`
eval	41	*v1 v2*	variable (binding unknown) in arithmetic expression
pushb	42	*a*	small integer in arithmetic expression
pushi	43	*n*	large integer in arithmetic expression
pushv	44	*v*	variable (known to be integer) in arithmetic expression
firstresult	6	*v*	unify result of arithmetic expression with a firstvar
result	45	*v*	unify result of arithmetic expression with a variable
add	46		integer addition
sub	47		integer subtraction
mul	48		integer multiplication
div	49		quotient of integer division
mod	50		remainder of integer division
shr	51		bit shift right
shl	52		bit shift left
and	53		bit and
or	54		bit or
not	55		bit complement
neg	56		integer negation
eq	57		integer equality
ne	58		integer inequality
lt	59		integer less than
le	60		integer less than or equal
gt	61		integer greater than
ge	62		integer greater than or equal to