COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192

AMES GRANT
7N-61-CR
146979
p. 55

A Prolog Emulator

Evan Tick

Technical Note No. CSL-TN-87-324

(NASA-CR-192403) A PROLOG EMULATOR (Stanford Univ.) 55 p

N93-71600

Unclas

Z9/61 0146979

May 1987

The work described herein was supported by NASA under consortium agreement NCA2-109, using facilities provided under contract NAGW 419.

A Prolog Emulator

by

Evan Tick

Technical Note No. CSL-TN-87-324

May 1987

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

This note describes an efficient software emulator for the Warren Abstract Machine (WAM) Prolog architecture. The version of the WAM implemented is called Lcode. The Lcode emulator, written in C, executes the "naive reverse" benchmark at 3900 LIPS. The emulator is one of a set of tools used to measure the memory-referencing characteristics and performance of Prolog programs. These tools include a compiler, assembler, and memory simulators. An overview of the Lcode architecture is given here, followed by a description and listing of the emulator code implementing each Lcode instruction. This note will be of special interest to those studying the WAM and its performance characteristics. In general, this note will be of interest to those creating efficient software emulators for abstract machine architectures.

Key Words and Phrases: Prolog, Warren Abstract Machine, instruction set architecture, emulation

Copyright © 1987
by
Evan Tick

Table of Contents

List of Figures

List of Tables

1. Introduction

The Warren Abstract Machine (WAM) Prolog architecture was designed during the summer of 1983 at SRI by D. H. D. Warren [20]. It represents a rethinking of the DEC-10 Prolog architecture described in his dissertation [18] and [19]. The WAM is currently implemented on general purpose hosts via native-code (e.g., Tricia [3]), interpretation (e.g., Quintus Prolog [13]), and microcoded interpretation (e.g., on the VAX 8600 [7]), and on dedicated hosts (e.g., the UC Berkeley Programmed Logic Machine (PLM) [4] and the ICOT PSI-II [12]). In addition, extensions of the WAM architecture for parallel execution have been developed [8, 11, 1].

The WAM architecture is attractive because its storage model is very efficient. The storage areas are split into an instruction (code) space and data space. The data space is split into a heap, stack, trail and push-down-list (pdl). These areas are managed in a stack-like manner, offering a limited form of automatic garbage collection. Structures are stored in the heap, and are manipulated using a *structure copying* policy. Choice points and environments are stored in the stack. Choice points freeze all stack objects below them on the stack, creating the need for referencing *deep* environments. The stack-like management of these areas clean up garbage created for failed branches during non-determinate program execution. A traditional garbage collector is still required however for cleaning up garbage created as byproducts of deterministic program execution. The WAM architecture does *not* include a specification for this type of garbage collection. The trail is used to hold the addresses of logical variables in the stack and heap which have been bound, but may need to be *unbound* should failure cause backtracking to an execution point *before* the binding was created. The pdl is used by general unification as an argument stack for recursive unifications.

This report describes in great detail a modified version of the WAM instruction set, called Lcode. Lcode simplifies many aspects of the WAM, and fills in other regions conspicuously absent in the original specification. A description of a system of tools used to measure the memory performance of Lcode benchmarks is given. These tools include a Prolog to Lcode compiler, assembler and Lcode emulator. The compiler is a modification of the UC Berkeley PLM compiler, which is well documented in [17]. In this report only the Lcode emulator is described in detail, including abstractions of the actual C-code used to implement the emulator.

Knowledge of the WAM instruction set and general architecture are necessary to understand this report. For an overview of the WAM architecture, the reader is referred to [14]. For

detailed descriptions, see [20, 6, 5]. Lcode simplifies the WAM by removing *environment trimming*. Lcode simplifies the PLM architecture by removing *cdr-coding*. Lcode extends the WAM by including arithmetic instructions, cut instructions, and conditional branch instructions. The semantics of all instructions are described and compared to the original WAM semantics when appropriate.

2. The Lcode System

The Lcode system is a set of tools developed to empirically measure the memory characteristics of Prolog benchmarks. Memory reference behavior is measured using address trace-driven memory simulators. Traces are produced using an Lcode emulator that executes object files produced by an Lcode assembler. The assembler translates assembler files produced by a Prolog compiler. These tools are summarized in Table 2-1 and illustrated in Figure 2-1. The tools run on the Stanford Emulation Laboratory VAX-750, under Unix[1] 4.3 BSD.

tool	input	output	how implemented
compiler	Prolog source	Lcode assembler	Prolog
assembler	Lcode assembler	binary object	LEX/YACC
emulator	binary object	trace file	C
simulator	trace file	statistics	C

Table 2-1: Stanford Emulation Laboratory Prolog Tools

2.1 Compiler

The compiler is a modified version of the UC Berkeley PLM compiler [17]. The compiler, written in Prolog, is about 2900 source lines. The modifications are listed below. Refer to [15] for a complete description of the optimizations.

- **removal of cdr-coding**
  cdr-coding was removed to simplify the architecture.
- **static-sized environments**
  environment trimming was removed to simplify the architecture.
- **increased number of registers**
  16 registers were implemented as opposed to eight in the PLM.
- **arithmetic instructions**
  arithmetic and other primitive operations, e.g., `var/1`, have been lifted into the instruction set.
- **conditional branches**
  a peephole optimization was introduced wherein under certain circumstances, simple builtin conditionals, e.g., `>/2`, can be moved up into the head. If a conditional can be moved up in front of choice point creation, it is replaced with a conditional branch. Subsequently, if the choice point creation meets a cut, both are removed.

[1]Unix is a trademark of Bell Laboratory

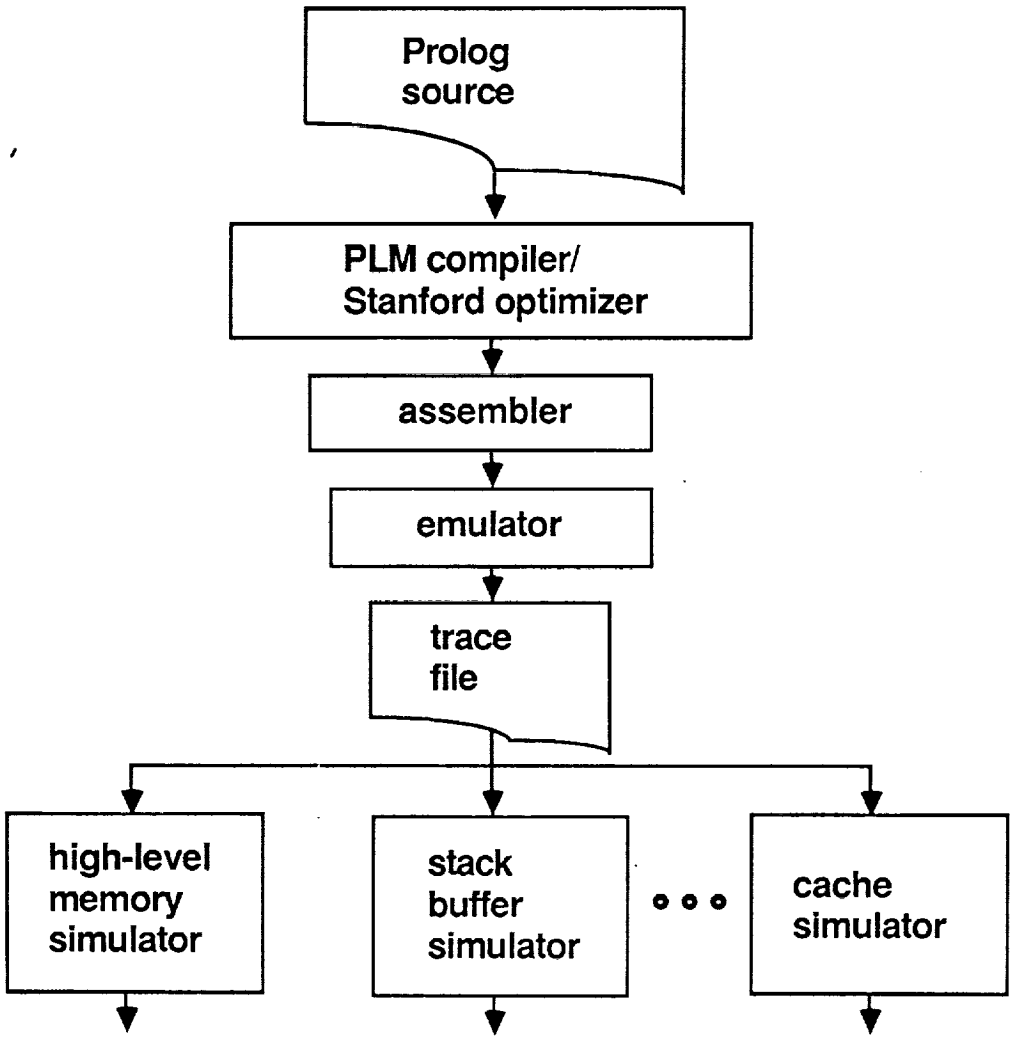

Figure 2-1: Memory Performance Methodology

- **incremental indexing**
  this type of indexing is a slight modification of the method outlined in [20], whereby the number of branches is reduced.

2.2 Assembler

The assembler is written in C around a LEX/YACC parser [10, 9], about 1000 source lines. The function of the assembler is to transform the symbolic intermediate code generated by the compiler into an object image readable by the emulator. The advantage of having the emulator read an object image is the much reduced time in loading executable programs.

Syntactic details follow. These rules are not important to the user because the compiler has taken over the burden of code generation. In rare instances, however, the user may wish to determine the performance of new compilation strategies without modifying the compiler. In these cases, direct modification and creation of assembler code is advantageous.

Comments can appear anywhere between a "**%**" and newline. White space can be used liberally; however, symbols cannot have white space between characters, labels must start in leftmost column and opcodes must not start in the leftmost column.

Labels, opcodes and functors are symbols of not more than nine characters. A label may have an optional colon as its last character, which is ignored. Labels and opcodes must *not* begin with a digit, but otherwise can consist of a wide variety of symbols. Functors must be specified as name/arity where name is a symbol and arity is an integer. Integer constants must be preceded by a "**&**".

Each assembler file must contain at least one **end** directive, which causes immediate termination of assembly at that location. There are two assembler flags. **-s** signifies that the symbol table should be dumped to standard output. **-w** signifies that assembler warnings should be suppressed.

Figure 2-2 shows the Lcode compiler output for the **append/3** program. Superfluous labels are generated and should be ignored. The **neck** instruction is used in various experiments but is not included in the basic Lcode architecture (it is ignored by the standard Lcode emulator).

2.3 Emulator

The Prolog emulator used to measure the memory performance of benchmark programs, is implemented in C. Arbitrarily large programs (to the limit of the VAX address space) can be emulated. The emulator kernel is about 2000 source lines with another 3000 source lines of support code. The emulator kernel is consists of a single large function wherein each intermediate level instruction is implemented. Primitive procedures not transformed by the compiler are dynamically interpreted in C. Notably, I/O primitives are implemented in LEX/YACC. A side effect of executing the program is the production of a memory reference trace file. Both data and instruction references can be traced. Another emulator option is procedure profiling, useful in determining Prolog program hot spots. Memory references made

```
        procedure           append/3

        switch_on_term      _3016,_3017,fail
        try                 3,_3022
        trust               _3026
_3016:
_3022:
        get_constant        X0,nil
        get_value           X1,X2
        neck
        proceed
_3061:
_3017:
_3026:
        get_list            X0
        unify_variable      X3
        unify_variable      X0
        get_list            X2
        unify_local_value   X3
        unify_variable      X2
        neck
        execute             append/3
```

Figure 2-2: Lcode Example: append/3

by primitive procedures are counted as other references; however, these primitives are not restricted to using the state registers of the WAM model. The assumption is that these primitives would be microcoded and required temporary registers would be available. The emulator has primitive debugging capabilities. The code space can be displayed through a disassembler and a single break point can be set. Memory areas and terms can be displayed symbolically. The emulator (with tracing off) runs at 3900 LIPS for naive reverse.

The emulator emulates Lcode, described in the next section in detail. WAM instructions are emulated in close correspondence to the detailed semantics given in [20]. Common Lcode operations which lend themselves to alternative semantics include general unification, cut, indexing instructions and builtins. The emulator implementation of these operations are described in the following sections. In addition, the emulator can emulate the Prolog CIF, split stack, and shadow register architectures [16].

2.4 Storage Management

Throughout the Lcode system, design decisions were made with speed and simplicity the most important considerations. The emulator is only used to analyze program execution and therefore user interface, error recovery and ease of program development were minor or nonexistent considerations. Note that the specifics of Lcode data types, tags, storage areas and storage management, as defined below, do not accurately resemble a realistic Prolog implementation. Many details, necessary for such an implementation (e.g., garbage collection), are purposely missing to facilitate analysis of the features which *are* included. The Lcode system is used to emulate a number of alternative architecture attributes and therefore is representative of a range of Prolog architectures, e.g., PLM and WAM.

The Lcode emulator manages six memory areas: code space, symbol table, heap, trail, stack and pdl. The code space contains the Lcode image. Assert and retract are not implemented, so this area is fixed. The symbol table holds the print-names of atoms, functors, procedures and top-level variables. The heap holds structures and unsafe values and is dynamically managed as a stack. The stack holds environments and choice points. The pdl is a push down stack used by general unification and univ (`==/2`), both of which are implemented as recursive functions. The emulator does not check for memory area overflows. No facilities for data area shifting, trimming or garbage collection are implemented. In addition, `cut` does *not* garbage-collect the trail. Maximum data area sizes may be specified as emulator input, and stay fixed during execution.

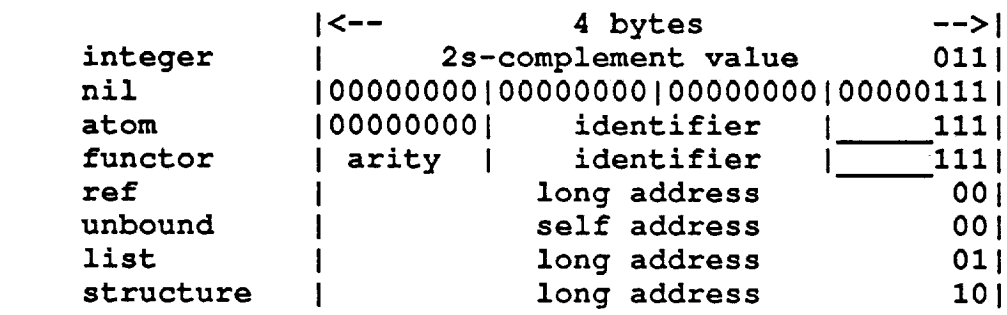

Table 2-2: Lcode Data Object Formats

A data object is a word (32 bits) composed of a variable length *tag* and a *value*. Lcode data objects are defined in Table 2-2. An *identifier* is an offset into the emulator's symbol table.

Unification of atoms, for instance, is done by comparing identifiers. An Lcode linker has not been implemented, so that entire Lcode programs must be assembled together to allow proper identifier assignment. A *long address* is a full 30 bit address pointing to another data object. An unbound variable points to itself (a *self address*) to differentiate it from an indirect reference.

Note the Lcode (and WAM) architecture is *structure-copying*, i.e., unifying an unbound variable with a structure involves copying the entire structure in the heap. In addition, the Lcode emulator uses standard list coding, requiring two heap words per list cell.

Lcode instructions are one or two words long. Minimal encoding is de-emphasized to allow fast emulation. The first halfword of each instruction is an opcode. An opcode is the address of the C code emulating that instruction. This allows fast instruction dispatch and requires that the emulator kernel fit in the first 64 Kbytes of virtual memory.

Arbitrarily large programs can be compiled and executed. This is implemented with both absolute and instruction relative addressing. To avoid a linkage phase, absolute addressing is actually implemented as base relative, where the base is the first location of the program. Base relative addresses are a full 32 bits long and are used only by inter-procedural branches, i.e., **`call`** and **`execute`**. Instruction relative addresses are 16 bits and are used by all other branches, i.e., all intra-procedural branches. It is for this distinction that the **`jump`** instruction was introduced to implement disjunction, rather than with the **`execute`** instruction, as is done in the PLM compiler. Note that intra-procedural branch offsets for the PLM are only 8 bits.

Lcode choice points are composed of a fixed size bookkeeping area (7 words) and a variable size argument area (c.f., PLM choice points which are fixed size of 15 words). Lcode environments are composed of a fixed size bookkeeping area (4 words — c.f., WAM with 2 words) and a variable size permanent variable area. Both choice points and environments remain statically fixed in size once they are created (c.f., WAM which trims environments).

An environment is created by saving the following four "bookkeeping" values on the stack: **`E`**, **`B`**, **`CP`**, and **`n`**, where **`n`** is the number of permanent variables saved in the environment. **`n+4`** words are allocated for the environment. The environment is summarized in Figure 2-3. Note that the **`n`** entry is not strictly necessary and can be removed if the **`put_unsafe_value_y`** implementation is modified. The Lcode environment is summarized in Figure 2-3. In the standard emulator, **`E`** points to the top (the highest memory address) of the environment (to **`n`**).

n	number of permanent variables
E	(tag signifies determinancy)
B	points to current choice point
CP	continuation program pointer
Y0	permanent variables
.	
.	
.	
Yn-1	

Figure 2-3: Lcode Environment Contents

Xn-1	valid argument registers
.	
.	
.	
X0	
H	current heap pointer
TR	current trail pointer
B	current backtrack pointer (to previous choice point)
P	address of clause to try next
CP	continuation pointer
E	current environment pointer
n	number of arguments

Figure 2-4: Lcode Choice Point Contents

An Lcode choice point is created by saving the following **n+7** values on the stack: the value **n**, temporary registers **Xn-1** through **X0**, the current environment pointer **E**, the current continuation **CP**, the address **P** of the next clause to try, a pointer to the previous choice point **B**, the current trail pointer **TR**, and the current heap pointer **H**. **HB** is then set to the current heap pointer and **B** is set to point to the current top of stack. The choice point is summarized in Figure 2-4. In the standard, single-stack emulator, **B** points to the bottom (the lowest memory address) of the choice point (to **n**).

3. The Lcode Emulator

Table 3-1 summarizes the Lcode instruction set. The operands are denoted as `C` — atom, integer or functor, `Yi` — permanent variable (offset in current environment), `Vi` — argument register or permanent variable, `L` — instruction address and `n` — integer. In the following sections, for each operation, an abstract listing of the Lcode emulator C-code is given. Execution of each instruction results either in failure or success. All failures are processed by the `fail` routine given in Section 3.5.9. Success implies the dispatch of the next instruction (pointed to by `P`). Macros used in the code segments are listed in Section 3.1. The components of the environments and choice points defined in Figures 2-3 and 2-4 are accessed in the emulator with macros. For example, `B_E` represents the environment pointer in the current choice point and `E_B` represents the choice point pointer in the current environment. These are noted in the text as `E(B)` and `B(E)` respectively.

```
top: {
#ifdef DEBUG                    /* compile time option  */
    if (P==break_address)       /* single break address */
        single_step=1;          /* single step flag     */
    if (single_step) {
        save_state;             /* save Prolog state    */
        debug(&state);          /* enter debugger       */
        restore_state;          /* restore Prolog state */
        }
#endif DEBUG
    label = VV;                 /* get opcode (address) */
    asm("  jmp  *_label");      /* jump to address      */
    }
```

Figure 3-1: Emulator Top Level

The Lcode instruction set formats are summarized in Appendix A. The emulator uses the loosely word encoded formats because on the VAX host, this facilitates decoding. The formats are wasteful of space, for instance allocating a byte for a temporary register specifier. The macros used for instruction object accesses are V, VV and VVVV. These access one, two and four bytes respectively. On the VAX, it is advantageous to access objects aligned on byte boundaries. Therefore all instructions are composed of pieces occupying integral numbers of bytes. To ensure this fit, different instructions may use both V and VV, for instance, to access register specifiers. Opcodes, however, always occupy two bytes. The value of an instruction's opcode is

goal matching
```
put_variable Vi,Ai
put_constant Ai,C
put_nil Ai
put_list Ai
put_structure Ai,C
put_value Vi,Ai
put_unsafe_value Yi,Ai
```

head matching
```
get_variable Vi,Ai
get_constant Ai,C
get_nil Ai
get_list Ai
get_structure Ai,C
get_value Vi,Ai
```

structure matching
```
unify_variable Vi
unify_constant C
unify_nil

unify_value Vi
unify_local_value Vi
unify_void n
```

clause control
```
allocate n
deallocate
call L
execute L
proceed
escape n
```

indexing
```
branch n,Ai,L
comp n,Vi,Vj
cond n,Vi
hash C,L
jump L
switch_type Lc,Ll,Ls
switch_constant n
switch_structure n
```

procedure control
```
try n,L
retry L
trust L
try_me_else n,L
retry_me_else L
trust_me_else_fail
cut
cut_strong
cutd L
fail
```

arithmetic
```
add Ai,Aj,Ak
add_constant Ai,Aj,C
decrement Ai,Aj
divide Ai,Aj,Ak
divide_constant Ai,Aj,C
increment Ai,Aj
mod Ai,Aj,Ak
mod_constant Ai,Aj,C
multiply Ai,Aj,Ak
multiply_constant Ai,Aj,C
subtract Ai,Aj,Ak
subtract_constant Ai,Aj,C
```

Table 3-1: Lcode Instruction Set

the address, within the emulator, of the code for executing that instruction. Therefore to dispatch an instruction requires VAX assembly code to *jump* to address specified by the instruction's opcode. Although this method is not especially reliable or portable, it is fast. The top-level of the emulator is shown in Figure 3-1.

The support functions of the Lcode system, such as the loader, disassembler, I/O package, debugger, symbol-table manager, etc., are *not* described in this note. These support functions are highly system dependent and unrelated to Prolog architecture issues. As was previously mentioned, and is typical for systems such as this, the support code size exceeds the emulator kernel code size.

3.1 Emulator Macros

```
typedef union {int w; char b; short h;} blob;
struct symtab_rec{      /* symbol table entry        */
        char type;      /* type of entry             */
        char length;    /* length of identifier name */
        char key[40];   /* identifier name           */
        int  value;};   /* value of entry            */

/* instruction object access functions */
#define AsBlobPtr(x)    ((blob *)(x))
#define V               ((AsBlobPtr(P++)->b) & 0x000000ff)
#define VV              (P+=2,((AsBlobPtr(P-2)->h) & 0x0000ffff))
#define VVVV            (P+=4, AsBlobPtr(P-4)->w)

/* data object access functions */
#define tagof(x)        ( (x) & 0x00000007)
#define arity(x)        (((x) & 0xff000000) >> 24)
#define ident(x)        (((x) & 0x00ffff00) >>  8)
#define intval(x)       ((x)>>3)
#define MaskArity(f,a)  ((a<<24) | f)

/* data object type check functions */
#define IsRef(x)        (((x) & 3)==0)
#define IsList(x)       (((x) & 3)==1)
#define IsStrct(x)      (((x) & 3)==2)
#define IsInteger(x)    (((x) & 7)==3)
#define IsFunctor(x)    (((x) & 7)==7)
#define IsAtom(x)       (IsFunctor(x) && (arity(x)==0))
#define IsNil(x)        (IsFunctor(x) && (ident(x)==0))

/* data object type conversion functions */
#define AsRef(x)        ((x) & 0xfffffffc)
#define AsStrct(x)      ((x) | 0x00000005)
#define AsList(x)       ((x) | 0x00000001)
#define car(x)          *(AsRef(x))
#define cdr(x)          *(AsRef(x)+4)

/* primitive Prolog operations */
#define deref(x)        {int t; while(IsRef(x) && (t = x,x = *t,t != x));}
#define trail(x)        if ((((x)>STACKBOT)&&((x)<B))||((x)<HB)) *TR-- = (x);
#define bindT(x)        {*T = (x); trail(T);}
```

```
#define bindS(x)        {*S = (x); trail(S);}
#define popS            *S; S+=4;
#define pushH(x)        {*H = (x); trail(H); H++;}

#define ref             0: case 4:
#define list            1: case 5:
#define struct          2: case 6:
#define integer         3
#define atom            7

#define E_S             *(E-0)      /* env access functions  */
#define E_E             *(E-1)
#define E_CP            *(E-2)
#define E_B             *(E-3)
#define Y(x)            *(E-(x)-4)  /* permanent registers   */

#define B_B             *(B-0)      /* cp access functions   */
#define B_S             *(B-1)
#define B_H             *(B-2)
#define B_E             *(B-3)
#define B_CP            *(B-4)
#define B_TR            *(B-5)
#define B_P             *(B-6)
#define B_X(x)          *(B-(x)-7)

#define NIL             7               /* nil symtab key        */
#define LIST_FUNCTOR    ...             /* ./2 symtab key        */
#define STACKBOT        ...             /* bottom addr of stack  */
#define CODEBOT         ...             /* bottom addr of code   */
#define PDLBOT          ...             /* bottom addr of pdl    */

/* abstract machine state */
int     X[16];                          /* temporary registers   */
int     B,CP,E,H,HB,P,Q,S;              /* state registers       */
int     R,T,U,W,Z;                      /* temporary registers   */
char    rmode, wmode, dmode;            /* modes                 */
struct symtab_rec symtab[...];          /* symbol table          */
```

3.2 Get Instructions

3.2.1 get_constant l,c

This instruction represents a head argument which is a constant. `i` is a temporary register and `c` is a constant. `get_nil` can be implemented with `c==nil`. The instruction gets the value of register `Xi` and dereferences it. If the result is a reference to an unbound variable, that variable is bound to `c`, and the binding is trailed if necessary. Otherwise, the result is compared with `c`, and if the two values are not identical, backtracking occurs.

```
get_constant: {
    S = X[VV]; deref(S);
    switch (tagof(S)) {
        case ref:     bindS(VVVV); break;
        case atom:
        case integer: if (VVVV == S) break;
        case list:
        case strct:   goto fail;
    }   }
```

3.2.2 get_list i

This instruction marks the beginning of a list occurring as a head argument. The instruction gets the value of register `Xi` and dereferences it.

If the result is a reference to an unbound variable then the variable is bound to a new list pointer pointing at the top of heap. The binding is trailed if necessary and execution proceeds in "write" mode. `H` will be used by two subsequent `unify` instructions to access the head and the tail of the list. Note that Lcode does not implement cdr-coding.

Otherwise, if the result is a list then the `S` pointer is set to point to the arguments of the list and execution proceeds in *read* mode. Otherwise, backtracking occurs.

```
get_list: {
    S = X[VV]; deref(S);
    switch (tagof(S)) {
        case ref:     bindS(AsList(H));
                      wmode = 1;
                      break;
        case list:    S = ToRef(S);
                      rmode = 1;
                      break;
        case atom:
        case strct:
        case integer: goto fail;
    }   }
```

3.2.3 get_structure i,f

This instruction marks the beginning of a structure (without embedded substructures) occurring as a head argument. `f` is the functor of the desired structure (name and arity encoded in one word). The instruction gets the value of register `Xi` and dereferences it.

If the result is a reference to a variable, that variable is bound to a new structure pointer pointing to the top of heap. The binding is trailed if necessary, the functor `f` is pushed onto the

heap, and execution proceeds in "write" mode. Subsequent **unify** instructions access the components of the structure with the H pointer.

Otherwise, if the result is a structure and its functor is identical to **f**, the **S** pointer is set to point to the arguments of the structure and execution proceeds in *read* mode. Otherwise, backtracking occurs.

```
get_structure:
    S = X[VV]; deref(S);
    switch (tagof(S)) {
        case ref:
            bindS(AsStruct(H));
            pushH(VVVV);
            wmode = 1;
            break;
        case strct:
            S = ToRef(S); /* strip tag */
            R = popS;
            if (VVVV == R) {
                rmode = 1;
                break;
                }
        case atom:
        case list:
        case integer: goto fail;
    }   }
```

3.2.4 get_value_v i,j

This instruction represents a head argument which is a bound variable. The instruction unifies the contents of register **Xj** with the contents of register **Vi**. The semantics in [20] indicate that for **get_value_x**, the final result is left in register **Xj** to speed up subsequent dereferences. This optimization is removed to simplify the implementation.

```
get_value_x: {U = X[V]; T = X[V]; goto unify;}
get_value_y: {U = Y(V); T = X[V]; goto unify;}
```

3.2.5 get_variable_v i,j

This instruction represents a head argument which is an unbound variable. The instruction loads the contents of register **Xj** into register **Vi**.

```
get_variable_x: {T = V; X[T] = X[V];}
get_variable_y: {T = V; Y(T) = X[V];}
```

3.3 Put Instructions

3.3.1 put_constant i,c

This instruction represents a goal argument which is a constant, `c`. The instruction loads `c` into register `Xi`. `put_nil i` can be implemented with `put_constant i,nil`.

```
put_constant: {T = VV; X[T] = VVVV;}
```

3.3.2 put_list I

This instruction marks the beginning of a list occurring as a goal argument and is similar to `get_list` encountering an unbound variable. The instruction places a list pointer corresponding to the top of heap into register `Xi`. Execution then proceeds in *write* mode.

```
put_list: {X[VV] = AsList(H); wmode = 1;}
```

3.3.3 put_structure I,f

This instruction marks the beginning of a structure occurring as a goal argument and is similar to `get_structure` encountering an unbound variable. The instruction pushes the functor `f` onto the top of heap via the `H` pointer and puts a corresponding structure pointer into register `Xi`. Execution then proceeds in *write* mode.

```
put_structure: {
    X[VV] = AsStruct(H);
    pushH(VVVV);
    wmode = 1;
    }
```

3.3.4 put_unsafe_value_y i,J

This instruction represents the last occurrence of an unsafe variable. The instruction dereferences `Yi`. If `Yi` dereferences to a variable in the current environment, that variable is bound to a new global variable created on the top of heap, the binding is trailed if necessary, and register `Xj` is set to a reference to the new global variable. Otherwise, the dereferenced value of `Yi` is loaded directly into register `Xj`.

```
put_unsafe_value_y: {
    S = Y(V); deref(S);
    if (((E-4-E_S) < S) && ((E-4) >= S)) {
        bindS(H);
        X[V] = H;
        pushH(H);
        } else
        X[V] = S;
    }
```

3.3.5 put_unsafe_integer_v i,j

This instruction has been introduced to facilitate compilation of efficient code for arithmetic expressions. It dereferences register `Vi` and checks if it is an integer. If it passes the type check, the dereferenced value is loaded into register `Xj`. Otherwise the failure occurs.

```
put_unsafe_integer_x: {
    S = X[V]; deref(S);
    if (!IsInteger(S)) goto fail;
    X[V] = S;
    }

put_unsafe_integer_y: {
    S = Y(V); deref(S);
    if (!IsInteger(S)) goto fail;
    X[V] = S;
    }
```

3.3.6 put_value_v i,j

This instruction represents a goal argument which is a bound variable. The instruction loads the value of register `Vi` into register `Xj`. Note that `put_value_x` is identical to `get_variable_x`.

```
put_value_x: {T = V; X[V] = X[T];}
put_value_y: {T = V; X[V] = Y(T);}
```

3.3.7 put_variable_x i,j

This instruction represents an goal argument which is an unbound variable. The instruction creates an unbound variable on the heap, and puts a reference to it into registers `Xj` and `Xi`.

```
put_variable_x: {X[V] = X[V] = H; pushH(H);}
```

3.3.8 put_variable_y i,j

This instruction represents a goal argument which is an unbound permanent variable. The instruction puts a *reference* to permanent variable `Yi` into register `Xj` and makes `Yi` an unbound *variable*.

```
put_variable_y: {
    S = E-V-4;          /* address of Yi */
    X[V] = *S = S;
    }
```

3.4 Unify Instructions

3.4.1 unify_constant c

This instruction represents a structure argument which is a constant, `c`. In read mode, it is similar to `get_constant`. The instruction gets the next argument from `S`, and dereferences it. If the result is a reference to a variable, that variable is bound to the constant `c`, and the binding is trailed if necessary. If the result is a non-reference value, that value is compared with the constant `c` and backtracking occurs if the two values are not identical. In write mode, the constant `c` is pushed onto the heap via the `H` pointer.

```
unify_constant: {
    if (rmode) {
        T = popS; deref(T);
        switch (tagof(T)) {
            case ref:     bindT(VVVV);
                          break;
            case integer: if (VVVV==T) break;
            case list:
            case atom:
            case strct:   goto fail;
            }
        } else /* copy_integer */
        pushH(VVVV);
    }
```

3.4.2 unify_local_value_v i

This instruction represents a structure argument which is a variable bound to a value that is not necessarily global. In read mode, the actions are identical to those of the `unify_value_v` instruction. In write mode, the value of register `Vi` is dereferenced. If the result is *not* a reference to a variable on the stack then the dereferenced result is pushed onto the heap via the `H` pointer. If the result *is* a reference to a variable on the stack, a new unbound variable is pushed

onto the heap via the H pointer, the variable on the stack is bound to a reference to the new unbound variable, and the stack binding is trailed if necessary. Note that to test if an address is in the stack, we only check if it is above the bottom of the stack because the heap is allocated *below* the stack.

```
unify_local_value_x: {
    if (rmode)
        goto unify_value_x;
        else { /* copy_local_value_x */
        R = VV;
        T = X[R]; deref(T);
        if (STACKBOT < T) {
            X[R] = H;
            bindT(H);
            pushH(H)
            } else
            pushH(T);
    }   }
unify_local_value_y: {
    if (rmode)
        goto unify_value_y;
        else { /* copy_local_value_y */
        T = Y(VV); deref(T);
        if (STACKBOT < T)
            bindT(H);
            pushH(H);
            } else
            pushH(T);
    }   }
```

3.4.3 unify_value_x i

This instruction represents a structure argument which is a variable bound to a global value. In read mode, it gets the next argument from S, and *unifies* it with the value in register Xi. The WAM specification indicates that the dereferenced result of the unification should be loaded into register Xi. This optimization has been measured and does not significantly reduce the number of memory references made by typical programs. It has been removed to simplify the implementation. In write mode, the value of variable Xi is pushed onto the heap via the H pointer.

```
unify_value_x: {
    U = X[VV];
    if (rmode) {
        T = popS;
        goto unify;
        } else /* copy_value_x */
        pushH(U);
    }
```

3.4.4 unify_value_y i

This instruction represents a structure argument which is a variable bound to some global value. In read mode, it gets the next argument from S, and *unifies* it with the value in register Yi. In write mode, the value of variable Yi is pushed onto the top of heap via the H pointer.

```
unify_value_y: {
    U = Y(VV);
    if (rmode) {
        T = popS;
        goto unify;
        } else /* copy_value_y */
        pushH(U);
    }
```

3.4.5 unify_variable_v i

This instruction represents a structure argument which is an unbound variable. In read mode, it gets the next argument from S and stores it in register Vi. In write mode, it pushes a new unbound variable onto the heap via the H pointer, and stores a reference to it in register Vi.

```
unify_variable_x : {
    if (rmode)
        X[VV] = popS;
        else { /* copy_variable_x */
        X[VV] = H;
        pushH(H);
    }   }

unify_variable_y: {
    if (rmode)
        Y(VV) = popS;
        else { /* copy_variable_y */
        Y(VV) = H;
        pushH(H);
    }   }
```

3.4.6 unify_void n

This instruction represents a sequence of n structure arguments which are single occurrence variables. In read mode, the next n arguments are skipped by incrementing H by n. In write mode, n new unbound variables are pushed onto the heap via the H pointer.

```
unify_void: {
    if (rmode)
        H += VV*4;
        else /* copy_void */
        for (T=VV; T>0; T--)
            pushH(H);
    }
```

3.5 Control Instructions

3.5.1 allocate n

This instruction appears in a clause with more than one goal in the body. It can be placed anywhere before the first occurrence of a permanent variable. **n** is the number of permanent variables in the clause. The **allocate** instruction allocates space for the new environment on the top of the environment stack (or local stack). **E** is then set pointing to the topmost word of the new environment (i.e., the topmost valid word of the environment stack).

```
allocate: {
    U = (dmode) ? AsRef(E):AsList(E); /* deter tag = 00 (ref)      */
    T = VV;                           /* nondeter tag = 01 (list) */
    E = TOS+T+4;                      /* TOS=(B>E)?B:E) for WAM   */
    E_S = T;                          /* TOS=(C>E)?C:E) for split */
    E_E = U;
    E_B = B;                          /* for fast cut             */
    E_CP = CP;
    }
```

3.5.2 branch L,n,i

This instruction performs a conditional local branch, calculating the branch target as a two byte offset, **L** from the end of the branch instruction. The condition is specified by an integer **n**. Temporary register **Xi** is checked for the condition, and if the check is successful, the branch is taken. Otherwise the next instruction is executed.

```
branch : {
    R = VV;
    P += 2;      /* realign things */
    T = V;
    S = X[V]; deref(S);
    switch (T) {
        case 0: if   IsNil(S)                          P+=R; break;
        case 1: if (!IsNil(S))                         P+=R; break;
        case 2: if   (IsInteger(S) && (!intval(S)))    P+=R; break;
        case 3: if (!(IsInteger(S) && (!intval(S)))) P+=R; break;
        case 4: if (IsInteger(S) && (intval(S)>0))     P+=R; break;
        case 5: if (IsInteger(S) && (intval(S)<=0))    P+=R; break;
        case 6: if (IsInteger(S) && (intval(S)>=0))    P+=R; break;
        case 7: if (IsInteger(S) && (intval(S)<0)      P+=R; break;
    }   }
```

3.5.3 call L

This instruction terminates a body goal. `CP` is set to the address of the instruction following the `call`. `P` is set to `L`, the callee address.

```
call: {
    dmode = 1;
    S = CODEBOT + VVVV;     /* segment register addressing */
    CP = P + 2;             /* P+2 because strange format  */
    P = S;
    }
```

3.5.4 comp_v n,i,j

This instruction compares register `Vi` with `Vj` for condition `n`. If the comparison succeeds, execution proceeds with the next instruction. Otherwise failure occurs.

```
comp_x: {
    R = V;
    S = X[V];
    T = X[V];
    goto comparison;
comp_y: {
    R = V;
    S = V; S = Y(S);
    T = V; T = Y(T);
comparison:
    deref(S); deref(T);
    if (!(IsInteger(S) && IsInteger(T))) goto fail;
    S = intval(S); T = intval(T);
    P += 3;      /* skip over rest of second word */
    switch (R) {
        case 0: if (S==T) break; else goto fail;
        case 1: if (S!=T) break; else goto fail;
        case 2: if (S<T)  break; else goto fail;
        case 3: if (S>=T) break; else goto fail;
        case 4: if (S>T)  break; else goto fail;
        case 5: if (S<=T) break; else goto fail;
    }   }
```

3.5.5 cond_v n,I

This instruction tests the tag of register `Vi`, specified by condition `n`. If the test succeeds, execution proceeds with the next instruction. Otherwise failure occurs.

```
cond_x : {
    T = V;
    S = X[V];
    goto condition;
cond_y : {
    T = V;
    S = V; S = Y(S);
condition:
    deref(S);
    switch (T) {
    case 0: if (TagIsRef(S) )                    break; goto fail;
    case 1: if (!TagIsRef(S))                    break; goto fail;
    case 2: if (IsFunctor(S)||IsInteger(S))      break; goto fail;
    case 3: if (!(IsFunctor(S)||IsInteger(S))) break; goto fail;
    case 4: if (IsList(S))                       break; goto fail;
    case 5: if (!IsList(S))                      break; goto fail;
    case 6: if (TagIsStruct(S))                  break; goto fail;
    case 7: if (!TagIsStruct(S))                 break; goto fail;
    case 8: if (IsAtom(S))                       break; goto fail;
    case 9: if (!IsAtom(S))                      break; goto fail;
    case 10:if (IsInteger(S))                    break; goto fail;
    case 11:if (!IsInteger(S))                   break; goto fail;
    case 12:if (TagIsStruct(S)||IsList(S))       break; goto fail;
    case 13:if (!(TagIsStruct(S)||IsList(S)))    break; goto fail;
    }}
```

3.5.6 cut, cut_strong, and cutd L

There are three types of Lcode cuts: standard cut, strong cut (operates without an enclosing environment) and disjunctive cut (introduced in [17]). Standard cut requires an enclosing environment, i.e., a previous `allocate` instruction within the same clause. As in [4], a state bit, **dmode** is dynamically updated indicating if the current environment belongs to a clause with an associated choice point or to a clause with no choice point. This condition is referred to as the *determinacy* of the clause. Cut is implemented by saving, in each environment, the determinacy bit and a pointer, **B(E)**, pointing to the choice point current when the environment was allocated. If the current environment is determinate, all choice points more recent than the environment's choice point are removed, i.e., **B** is reset to **B(E)**. If the current environment is nondeterminate, all choice points more recent than *and including* the environment's choice point are removed, i.e., **B** is reset to the choice point below **B(E)**. If any choice points remain, the heap backtrack point, **HB** is cut back to the new current choice point's heap pointer.

Strong cut, `cut_strong`, is used to cut a predicate without an environment. In this case, the determinacy bit, **dmode**, is checked directly. If the predicate is determinate, nothing is done. If the predicate is nondeterminate, the current choice point, **B**, is reset to the choice point immediately preceding it. If any choice points remain, the heap backtrack point, **HB** is cut back to the new current choice point's heap pointer. Note that the multiple cut problem occurs for

clauses of the form

```
p :- !,q,!.
```

for nondeterminate **p**. Here the second strong cut will attempt to remove **p**'s choice point, already removed by the first strong cut. Two solutions exist: either generate standard cuts here (requiring allocation of an environment), or transform the clause into

```
p :- !,q'.
q' :- q,!.
```

Disjunctive cut, **cutd**, is generated by the compiler only between the "if" and "then" parts of a conditional. Cuts in a disjunction are translated into **cut**, thus cutting out of the entire predicate. Thus **cutd** is implemented slightly differently than in the PLM. First, the choice point chain is searched for a choice point matching the **cutd** operand. The choice point just before (earlier than) that one is selected. This correctly implements conditionals by cutting out the disjunction but not the whole predicate when the "then" part fails.

```
cut: {
    B = E_B;
    if nondeterminate(E)
        B = B_B;
    if (STACKBOT < B)
        HB = B_H;
    dmode = 1;
    }

cut_strong: {
    if (!dmode) {
        B = B_B;
        if (STACKBOT < B)
            HB = B_H;
        }
    dmode = 1;
    }

cutd: {
    S = P + VV;
    while (B_P != S)
        B = B_B;
    if (STACKBOT < B) {
        B = B_B;
        if (STACKBOT < B)
            HB = B_H;
    }   }
```

3.5.7 deallocate

This instruction appears before the final **execute** instruction in a clause with more than one goal in the body. The previous continuation is restored and the current environment is discarded. In the case of a single local stack, this instruction resets the environment to either the top of stack or somewhere deep in stack. If **E>B**, then we use same management scheme as for **fail** because the object becomes the new top of stack.

```
deallocate: {
    CP = E_CP;
    E  = ToRef(E_E);
    }
```

3.5.8 execute L

This instruction terminates the final goal in the body of a clause. **P** is set to the callee address **L**.

```
execute: {
    dmode = 1;
    P = CODEBOT + VVVV;
    }
```

3.5.9 fail

This operation is used by both the user and system. The **X** registers, **E**, **P**, and **CP** pointers are restored from the current choice point. The trail is "unwound" as far as the choice point trail pointer, **TR(B)**, by popping references off the trail and resetting the variables they address to unbound.

Note the choice point is *not* removed and the **B(B)** value is *not* used during failure. This is because the choice point is kept until a **trust_me_else fail** instruction removes it. A current choice point can be *modified* by **retry_me_else** instructions, thus saving work. Note also that **H** is restored not from **H(B)**, although this would be correct, but rather from **HB**, the state register shadowing **H(B)**.

A note about the trail: the trail grows downwards in memory as a stack. The **TR** pointer points to the last valid entry on the trail. When a choice point is created, the saved **TR(B)** points to the last trailed address *before* the choice point jurisdiction. Thus during detrailing upon failure, the trail is popped until **TR==TR(B)**. This also obviates any need for checking if the trail has

underflowed.

```
fail: {
    if (!(STACKBOT < B)) {      /* if no more choice pts*/
        printf("no\n\n");       /* then program fails   */
        goto top;
        } else {                /* restore choice point */
        H = HB;                  ,
        E = B_E;
        CP = B_CP;
        P = B_P;
        S = B_S;                /* if split: S=B(B)-B-7 */
        for (T=0; T<S; T++)     /* restore args         */
            X[T] = B_X(T);      /* from choice point    */
        S = B_TR;               /* detrail              */
        while (TR < S) {
            TR++;
            T = *TR;
            *T = T;             /* unbind trail address */
    }       }
```

3.5.10 jump L

This instruction is an unconditional branch. The target address is calculated as a two byte offset, **L**, from end of the **jump** instruction. **L** is interpreted as a two byte twos-complement integer. **jump** is used in disjunctions instead of **execute** to distinguish between local and global transfer of control.

```
jump: {T = VV; P += T;}
```

3.5.11 proceed

This instruction terminates a unit clause. **P** is reset to **CP**.

```
proceed: {dmode = 1; P = CP;}
```

3.6 Indexing Instructions

3.6.1 hash L,f

This instruction defines a single hash table entry and is placed after a **switch_constant** or **switch_structure** instruction, forming the actual hash table as in-line data words. A hash table entry is two words - the first, **L** holds the value (a pointer to a clause) and the second, **f**, holds the key (a constant). This instruction is not executed, but rather defines data needed by the previous **switch** instruction. Note the single argument of the **switch** instructions must be

equal to the number of following **hash** instructions.

3.6.2 retry L

This instruction is one in the middle of a sequence of instructions identifying clauses with the same key. The current choice point **P(B)** entry is assigned the address of the instruction following the **retry** instruction and the program pointer **P** is set to the clause address **L**.

```
retry: {
    dmode = 0;
    B_P = P+2;
    R = VV;
    P += R;
    }
```

3.6.3 retry_me_else L

This instruction precedes the code for a clause in the middle of a procedure (i.e. it is not the first or last clause). The current choice point entry **P(B)** is assigned the address **L**.

```
retry_me_else: {
    dmode = 0;
    B_P = P+VV;
    }
```

3.6.4 switch_constant n

This instruction defines a hash table for a group of clauses having constants in the first head argument position. The instruction dereferences **X0** and fails if the dereferenced result is not a constant. Otherwise the constant value is hashed to compute an index in the range 0 to n-1 into the hash table defined by the words following the **switch_constant** instruction. The size of the hash table is **n**.

Each hash table entry gives access to the clause or clauses whose keys hash to that index. The constant in **X0** is compared with the different keys until one is found which is identical, at which point the program pointer **P** is set to point to the corresponding clause or clauses. If the key is not found, backtracking occurs. See the **hash** instruction for a description of a hash table entry.

Note that in the Lcode emulator, a hash function was not implemented — instead a linear search is used. Implementing an efficient hash function is an important method for speeding-up

the emulator.

```
switch_constant: {
    T = X[0]; deref(T);
    if ((tagof(T)==integer) || (tagof(T)==atom)) {
        S = VV;                     /* grab size of table       */
        for (W=0;W<S;W++) {         /* iterate for now          */
            P += 2;                 /* skip hash opcode         */
            U = P;                  /* save P for later calc    */
            R = VV;                 /* grab address offset      */
            Z = VVVV;               /* grab key                 */
            if (T==Z) {             /* if match we're done      */
                if (!R) goto fail; /* recall: fail==0          */
                P = R+U;            /* calc instr relative addr */
                goto top;
        }   }   }
    goto fail;
    }
```

3.6.5 switch_structure n

This instruction provides hash table access to a group of clauses having structures in the first head argument position. The effect is identical to that of **switch_constant**, except that the key used is the principal functor of the structure in **X0**. The instruction fails if **X0** does not hold a structure. Again, linear search is implemented instead of a hash function.

```
switch_structure: {
    T = X[0]; deref(T);
    if TagIsStruct(T) {
        T = *ToRef(T);
        S = VV;
        for (W=0;W<S;W++) {
            P += 2;
            U = P;
            R = VV;
            Z = VVVV;
            if (T==Z) {
                if (!R) goto fail;
                P = R+U;
                goto top;
        }   }   }
    goto fail;
    }
```

3.6.6 switch_type Lc,Ll,Ls

This instruction provides access to a group of clauses with a non-variable in the first head argument. It causes a dispatch on the type of the first argument of the call. The argument **X0** is dereferenced and, depending on whether the result is a constant, (non-empty) list, or structure, the program pointer **P** is set to **Lc**, **Ll**, or **Ls**, respectively. If **X0** is unbound, program

execution proceeds with the next instruction.

```
switch_term: {
    S = X[0]; deref(S);
    switch (tagof(S)) {
        case ref:   P += 6; break;
        case strct: P += 2;
        case list:  P += 2;
        case atom:
        case integer:
                    U = P;
                    R = VV;
                    if (!R) goto fail;
                    P = R+U;
    }   }
```

3.6.7 trust L

This instruction is the last of a sequence of instructions identifying clauses with the same key. The current choice point is discarded, registers `B` and `HB` are reset to correspond to the previous choice point and the program pointer `P` is set to the clause address `L`.

```
trust: {
    R = VV;
    P += R;
    goto trust_me_else;
    }
```

3.6.8 trust_me_else fail

This instruction precedes the code for the last clause in a procedure. The current choice point is discarded, and registers `B` and `HB` are reset to correspond to the previous choice point.

```
trust_me_else: {
    B = B_B;
    HB = B_H;
    dmode = 1;
    }
```

3.6.9 try n,L

This instruction is the first of a sequence of instructions identifying clauses with the same key. A choice point is created on the top of the choice point stack (which may be the same as the environment stack, or distinct). `L` is the address of the next clause. `n` is the arity of the clause. `HB` is then set to the current heap pointer and `B` is set to point to the top of the new current choice point. Finally, the program pointer `P` is set to the clause address `L`.

```
try: {
    dmode = 0;
    S = VV;
    T = B;
    B = ((B>E)?B:E)+S+7;
    B_S = S;              /* single stack model */
    B_E = E;
    B_B = T;
    B_H = H;
    B_CP = CP;
    B_TR = TR;
    for (T=0;T<S;T++)
        B_X(T) = X[T];
    HB = H;
    B_P = P+4;
    R = VV;
    P += R;
    }
```

3.7 try_me_else n,L

This instruction precedes the code for the first clause in a procedure with more than one clause. A choice point is created on the top of the choice point stack (which may be the same as the environment stack, or distinct). **L** is the address of the next clause to try. **n** is the arity of the clause.

```
try_me_else: {
    dmode = 0;
    S = VV;
    T = B;
    B = ((B>E)?B:E)+S+7;
    B_S = S;              /* single stack model */
    B_E = E;
    B_B = T;
    B_H = H;
    B_CP = CP;
    B_TR = TR;
    for (T=0;T<S;T++)
        B_X(T) = X[T];
    HB = H;
    R = VV;
    B_P = P+R;
    P += 2;               /* skip last halfword */
    }
```

3.8 Arithmetic Instructions

Arithmetic has been included in the Lcode instruction set (c.f., the WAM). Each arithmetic operator includes two instructions, e.g., `add` and `add_constant`. Both of these instructions modify their destination operand. The compiler must realize this and generate code

appropriately. Shown below are the **add** instructions. Others are similar: **subtract**, **mod**, **multiply**, **divide**, **increment**, and **decrement**.

```
add i,j,k: {
    S = V;
    T = intval(X[V]);
    R = intval(X[V]);
    X[S] = AsInteger(T+R);
    }

add_constant i,j,c: {
    S = V;
    T = intval(X[V]);
    R = intval(VVVV);
    X[S] = AsInteger(T+R);
    }
```

3.9 General Unifier

This operation is used by several instructions to perform a general unification of two terms. All calls to the general unifier are immediately followed by an instruction dispatch. Thus the unifier can do the dispatch itself and need not return to the top-level caller. In addition, the unifier is written with only one recursive call. These two properties allow the unify code to be accessed with a simple jump. This design owes much to fruitful discussions with R. O'Keefe of Quintus Computer Inc.

The general unification algorithm uses a push down list, PDL. The top of PDL pointer is **Q**, and the base of the PDL is **PDLBOT**. Frames on the PDL are three words in length: term #1, term #2 and arity. Notice the lack of return address. The unifier decides if it should dispatch the next instruction or return to a recursive call by checking the arity. Initially, the caller loads term #1 into **T**, term #2 into **U** and jumps to the unifier.

The unifier initially loads a zero into **R** and then recursively unifies the two terms. The unification algorithm calculates, in **R**, a running total of the arities of complex terms encountered. Thus **R** represents the number of recursive iterations necessary to complete the unification and when **R==0**, the operation is complete.

```
unify: {
    R = 0;
Unify_top:
    deref(U); deref(T);
    if (U!=T) {
        switch (tagof(U)) {
            case ref:
                if (IsRef(T) && (U<=T)) bindT(U)
                    else bindS(T)
                break;
            case atom: case integer:
                switch (tagof(T)) {
                    case ref:
                        bindT(U); break;
                    case atom: case strct:
                    case integer: case list:
                        Q = PDLBOT; goto fail;
                break;
            case list:
                switch (tagof(T)) {
                    case ref:
                        bindT(U); break;
                    case list:
                        U = AsRef(U); T = AsRef(T);
                        R += 2; W = 2;
                        goto Unify_recurse;
                    case strct: case atom: case integer:
                        Q = PDLBOT; goto fail;
                break;
            case strct:
                switch (tagof(T)) {
                    case ref:
                        bindT(U); break;
                    case strct:
                        U = AsRef(U); Z = *U;
                        T = AsRef(T);
                        if ((Z != *T) || (!IsFunctor(Z)))
                            {Q = PDLBOT; goto fail;}
                        W = arity(Z); R += W;
                        U += 4; T += 4;
Unify_recurse:          while (W-->0) {
                            R--;
                            if (W>0) {
                                *Q++ = U;
                                *Q++ = T;
                                *Q++ = i;}
                            U = *U;
                            T = *T;
                            goto Unify_top;
Unify_return:               W = *--Q;
                            T = *--Q + 4;
                            U = *--Q + 4;
                            }
                        break;
                    case list: case atom: case integer:
                        Q = PDLBOT; goto fail;
                break;
        }   }
    if (R == 0) goto top;
    goto Unify_return;
    }
```

3.10 Built-in Predicates

Built-in predicates are predefined procedures in Prolog. Only a subset of the standard built-ins [2, 13] are supported by the Lcode system. These include arithmetic comparison, type checking, I/O, and control facilities. Built-in predicates are categorized as either *simple* or *complex*, depending on how they are implemented. Simple built-ins are implemented with a single Lcode instruction, and take their arguments from any of the **X** or **Y** registers. Complex built-ins are implemented with the **escape** instruction, and take their arguments from **X0**,**X1**,... using standard calling conventions. All of the built-in predicates, except for **call/1**, are safe, i.e., they do not modify **X** registers other then their own arguments. Therefore the **X** registers can be allocated across built-ins within a clause. A small set of built-ins (**\=/2**, **not/1**, **true**, and **\+/1**) are transformed into other predicates in the pretranslation phase of the compiler.

instruction	built-ins	
cut	!/0	
fail	fail/0	
get_value_v...	=/2	
comp_v	</2, >/2, <=/2,	>=/2, =:=/2, =\=/2
cond_v	atom/1,	nonatom/1,
	atomic/1,	nonatomic/1,
	composite/1,	noncomposite/1,
	integer/1,	noninteger/1,
	list/1,	nonlist/1,
	simple/1,	nonsimple/1,
	structure/1,	nonstructure/1,
	var/1,	nonvar/1

Table 3-2: Simple Lcode Built-in Predicates

The simple built-in predicates of the Lcode system are listed in Table 3-2, categorized by the Lcode instruction used to implement them. The complex built-ins are listed in Table 3-3. The emulator C-code for the first six complex built-ins is listed below. Other complex built-ins are not listed because they are highly system dependent. The following descriptions assume, as do the previous instruction descriptions, that the next instruction is dispatched after successful execution of the built-in.

==/2	=../2	arg/3
call/1	functor/3	length/2
nl/0	read/1	readcell/1
see/1	seen/0	tab/0
time/1	write/1	writecell/1

Table 3-3: Complex Lcode Built-in Predicates

3.10.1 arg/3

This predicate unifies its third argument with a subcomponent of the second argument. The index of the subcomponent is specified by the first argument. If the second argument is not a list or a structure or the first argument is not an integer index in the proper range, the predicate fails.

```
arg: {
    R = X[0]; deref(R); /* Index */
    T = X[1]; deref(T); /* Term  */
    S = X[2]; deref(S); /* Item  */
    switch (tagof(T)) {
        case list:
            if IsInteger(R) {
                R = intval(R)-1;
                if ((R==0) || (R==1))
                    break;
                }
            goto fail;
        case strct:
            if IsInteger(R) {
                R = intval(R);
                if ((R > 0) && (R <= arity(*(ToRef(T))))
                    break;
                }
            goto fail;
        case ref:
        case atom:
        case integer: goto fail;
        }
    T = ToRef(T) + 4*R;
    goto unify;
    }
```

3.10.2 call/1

This predicate executes the procedure specified by its argument. For example, `call(concat([1,2,3],[4],X))` will cause the execution of `concat([1,2,3],[4],X))`. The procedure must be specified as either an atom (if it requires no arguments) or a structure. Otherwise `call/1` fails. If the procedure specified does is not defined, `call/1` fails. The description below uses the support C-function `lookup`, which queries the symbol-table. Several other symbol-table support functions, not shown, are included in the Lcode system.

```
call: {
    char tempstring[40];
    T = X[0]; deref(T);
    switch (tagof(T)) {
        case atom:
            S = T;
            break;
        case strct:
            T = ToRef(T);
            S = *T;
            for (R=0,T+=4;R<arity(S);R++,T+=4)
                X[R] = *T;
            break;
        case ref:
        case list:
        case integer: goto fail;
        }
    CP = P;
    /* construct procedure name from structure name and arity */
    strcpy(tempstring,symtab[identifier(S)].key);
    strcat(tempstring,itoa(arity(S)));
    P = lookup(tempstring);
    }

int lookup(yytext)
        byteptr yytext;
{       int i,yyleng;
yyleng = strlen(yytext);
for (i=0;i<tabsize;++i)
    if (symtab[i].type == PROCEDURE)
        if (symtab[i].length == yyleng)
            if (!strcmp(symtab[i].key,yytext))
                return(symtab[i].value+CODEBOT);
return(CODEBOT);
}
```

3.10.3 functor/3

This predicate can be used either to create a structure or to determine the name and arity of an existing structure. `LIST_FUNCTOR` is the 32-bit identifier representing the "./2" functor.

```
functor/3: {
    T = X[0]; deref(T);
    U = X[1]; deref(U);
    W = X[2]; deref(W);
    switch (tagof(T)) {
    case ref:
        if (IsAtom(U) && IsInteger(W) && (intval(W)>=0)) {
            *T = AsStrct(H);
            *H = MaskArity(U,intval(W));
            for (Z=(++H), H+=intval(W); Z<H; Z++) *Z = Z;
            } else
            if (IsInteger(U) && IsInteger(W) && (intval(W)==0))
                *T = U;
                else
                goto fail;
        goto top;
    case atom:
    case integer:
        R = AsInteger(0);
        break;
    case list:
        R = AsInteger(2);
        T = LIST_FUNCTOR;
        break;
    case strct:
        T = *(AsRef(T));
        R = AsInteger(arity(T));
        T = AsFunctor(ident(T),0);
        break;
    }
    if (IsRef(W))
        *W = R;
        else
        if (W!=R) goto fail;
    goto unify;
    }
```

3.10.4 length/2

If the first argument is a list, this predicate returns the length of the list as the second argument. If the first argument is unbound or something other than a list, the predicate fails. A list must have a nil cdr for its last element. Thus, for example, `length([a|b],X)` fails. `length/2` is implemented iteratively, successively cdring down the first argument while counting.

```
length: {
    U = 0;
    T = X[0];
leng: deref(T);
    if IsNil(T) {
        S = X[1]; deref(S);
        if TagIsRef(S)
            *S = AsInteger(U);
            else
            if (!(IsInteger(S)) || (intval(S)!=U))
                goto fail;
        } else
        if IsList(T) {
            U++;
            T = ToRef(T)+4;      /* get cdr */
            goto leng;
            } else
            goto fail;
    }
```

3.10.5 ==/2

This operation tests whether two terms are exactly equivalent. This code is much simpler than the unifier, but has the same recursion mechanism.

```
==/2: {
    U = X[0]; T = X[1]; R = 0;
Univ_top:
    deref(U); deref(T);
    switch (tagof(U)) {
        case ref:
        case atom:
        case integer:
            if (U != T) {Q = PDLBOT; goto fail;}
            break;
        case list:
            if (!IsList(T)) {Q = PDLBOT; goto fail;}
            U = AsRef(U); T = AsRef(T);
            R += 2; W = 2;
            goto Univ_recurse;
        case strct:
            if (!IsStrct(T)) {Q = PDLBOT; goto fail;}
            U = AsRef(U); Z = *U;
            T = AsRef(T);
            if ((Z != *T) || (!IsFunctor(Z)))
                {Q = PDLBOT; goto fail;}
            W = arity(Z); R += W;
            U += 4; T += 4;
Univ_recurse:
            while (W-->0) {
                R--;
                if (W>0) {
                    *Q++ = U;
                    *Q++ = T;
                    *Q++ = W;}
                U = *U;
                T = *T;
                goto Univ_top;
Univ_return:    W = *--Q;
                T = *--Q + 4;
                U = *--Q + 4;}
             break;
        }
    if (R == 0) goto top;
    goto Univ_return;
    }
```

3.10.6 =../2

This operation either creates a structure from an existing list or decomposes a structure into a list. **NIL** is the 32-bit identifier for the atomic constant representing an empty list.

```
=../2: {
    S = X[0]; deref(S);
    T = X[1]; deref(T);
    switch tagof(S) {
        case ref:
            if (!IsList(T)) goto fail;
            W = car(T); Z = cdr(T);
            if (IsInteger(W) && IsNil(Z)) {
                *S = W;
                goto top;}
            if IsAtom(W) {
                T = Z;
                if IsNil(T) {
                    *S = W;
                    goto top;}
                *S = AsStrct(H);
                U = H++;
                R = 0;
                while (!IsNil(T)) {
                    R++;
                    *H++ = car(T);
                    T = cdr(T);}
                *U = AsFunctor(ident(W),R);
                goto top;
                }
            goto fail;
        case atom:
        case integer:
            U = AsList(H);
            *H++ = S;
            *H++ = NIL;
            break;
        case list:
            U = AsList(H);
            *H++ = LIST_FUNCTOR;
            *H   = AsList(H+1);   H++;
            *H++ = car(S);
            *H   = AsList(H+1);   H++;
            *H++ = cdr(S);
            *H++ = NIL;
            break;
        case strct:
            U = AsList(H);
            R = arity(Z = *AsRef(S));
            *H++ = AsAtom(ident(Z));
            for (W=1; W<=R; W++) {
                *H   = AsList(H+1); H++;
                *H++ = *(AsRef(S)+W*4);
                }
            *H++ = NIL;
            break;
        }
    goto unify;
    }
```

Appendix A. Lcode Instruction Set Summary

Table A-1 lists each Lcode instruction with its sizes for both word and byte encoding schemes. Each instruction is listed alphabetically by opcode, with an instance of the assembly code. The word encoding size is given in units of words. The byte encoding size is given in units of bytes. Notes concerning Table A-1 follow.

1. Local branch instructions (i.e., branches within a procedure) are given two sizes for each encoding scheme. The first size corresponds to a short offset of one byte. The second size corresponds to a long offset of two bytes. For example, with a byte encoding, **branch** requires 3 bytes for short offsets and 4 bytes for long offsets.

2. Non-local branch targets (**call** and **execute** instructions) are encoded as a two byte offset from a segment register.

3. The index instructions **switch_constant** and **switch_structure**, have sizes of 1 word or 2 bytes. This does not include the size of the hash table following the instruction. During emulation, only one hash entry reference (two reads — one for the key, one for the value) is counted in addition to the instruction fetch.

4. In general, the **trust_me_else** operand can be a local clause label. This facilitates code assertion and retraction. Since assertion/retraction of any kind is not implemented in the Lcode system, the **trust_me_else** instruction is always given a **fail** operand.

Table A-2 lists each Lcode instruction with associated dynamic statistics measured by averaging the statistics from the individual benchmark programs (CHAT, PLM, QC1 and ILI). Instructions not executed in any of the programs are not included in the table. The mean instruction frequency, data and instruction references per instruction (in *bytes*) and percent weight are shown. Instruction weight is calculated as the product of instruction frequency and references per instruction. All instructions have a fixed number of instruction references (except for the indexing instructions for which instruction references were not accurately measured). Notes concerning Table A-2 follow.

1. The **escape** statistics are averaged over those built-ins present in the benchmarks.

2. The *failure* statistics are averaged over *all* failures. No instruction bytes are referenced because failure is similar to a software trap.

3. The **get_constant**, **put_constant** and **unify_constant** instructions are further categorized as **atom** or **integer**. All the statistics presented as additive, so that for instance, **get_constant** accounts for 2.046% of all instructions executed, with 1.67% of the total weight. Note the benchmarks show a strong bias towards symbolic rather than arithmetic computation.

4. The Lcode compiler did not have the ability to generate **unify_value** instructions. Only the unoptimized form of **unify_local_value** instructions

were generated. For read mode, these instructions are equivalent, and are listed as **unify_value**.

5. Copy instructions correspond to unify instructions executed in write mode.
6. In write mode, a **unify_local_value** instruction dereferences its operand and globalizes it onto the heap if necessary. The **copy_local_value** category corresponds to write mode execution of **unify_local_value** instructions that *do* require globalization.
7. The **copy_value** category corresponds not to **unify_value** instructions executed in write mode, but rather to **unify_local_value** instructions that did *not* require globalization (in this case, execution of the two forms are identical, except for the extra dereference). Note that globalization was required only about 1 in 9 times.

Table A-3 summarizes these statistics by *instruction type*, as defined in Table 3-1. The instruction types are listed in order of greatest percent weight. These statistics consider *failure*, *general unification*, and **escape** as separate instruction types. Therefore the cost of general unification is *not* counted in the head or structure matching groups. Note that the indexing weight is highly optimistic, calculated assuming perfect hashing.

opcode	assembly instance	words	bytes
add	add X1,X2,X3	1	3
add_constant	add_constant X1,X2,15	2	6
allocate	allocate 8	1	2
branch (1)	branch nil,X1,_1234	1	3/4
call (2)	call _1234	1	3
comp_x	comp <,X1,X2	1	3
comp_y	comp <,Y1,Y2	1	4
cond_x	cond var,X1	1	2
cond_y	cond var,Y1	1	3
cut	cut	1	1
cutd	cutd _1234	1	2/3
cut_strong	cut_strong	1	1
deallocate	deallocate	1	1
decrement	decrement X1,X2	1	2
divide	divide X1,X2,X3	1	3
divide_constant	divide_constant X1,X2,15	2	6
escape	escape 3	1	2
execute	execute _1234	1	3
fail	fail	1	1
get_constant	get_constant X1,-44	2	6
get_list	get_list X1	1	2
get_nil	get_nil X1	1	2
get_structure	get_structure X1,f/4	2	6
get_value_x	get_value X1,X2	1	2
get_value_y	get_value Y1,X2	1	3
get_variable_x	get_variable X1,X2	1	2
get_variable_y	get_variable Y1,X2	1	3
increment	increment X1,X2	1	2
jump	jump _1234	1	2/3
mod	mod X1,X2,X3	1	3
mod_constant	mod_constant X1,X2,15	2	6
multiply	multiply X1,X2,X3	1	3
multiply_constant	multiply_constant X1,X2,15	2	6
proceed	proceed	1	1

Table A-1: Lcode Instruction Set Formats

opcode	assembly instance	words	bytes
put_constant	put_constant X1,-44	2	6
put_list	put_list X1	1	2
put_nil	put_nil X1	1	2
put_structure	put_structure X1,f/4	2	6
put_unsafe_integer_x	put_unsafe_integer X1	1	2
put_unsafe_integer_y	put_unsafe_integer Y1	1	2
put_unsafe_value_y	put_unsafe_value Y1,X2	1	3
put_value_x	put_value X1,X2	1	2
put_value_y	put_value Y1,X2	1	3
put_variable_x	put_variable X1,X2	1	2
put_variable_y	put_variable Y1,X2	1	3
retry	retry _1234	1	2/3
retry_me_else	retry_me_else _1234	1	2/3
stop	stop	1	1
subtract	subtract X1,X2,X3	1	3
subtract_constant	subtract_constant X1,X2,15	2	6
switch_constant (3)	switch_constant 8	1+2	2+8
switch_structure	switch_structure 8	1+2	2+8
switch_term	switch_term _123,fail,_123	1/2	4/7
trust	trust _1234	1	2/3
trust_me_else (4)	trust_me_else fail	1	1
try	try 8,_1234	1	3/4
try_me_else	try_me_else 8,_1234	1	3/4
unify_constant	unify_constant -44	2	5
unify_local_value_x	unify_local_value_x X1	1	2
unify_local_value_y	unify_local_value_y Y1	1	2
unify_nil	unify_nil	1	1
unify_value_x	unify_value_x X1	1	2
unify_value_y	unify_value_y Y1	1	2
unify_variable_x	unify_variable_x X1	1	2
unify_variable_y	unify_variable_y Y1	1	2
unify_void	unify_void 8	1	2

Table A-1: Lcode Instruction Set Formats - *continued*

opcode	% instr	data bytes	instr bytes	% weight
add	0.026	0.00	3	0.01
add_constant	0.014	0.00	6	0.01
allocate	3.491	16.00	2	5.27
call	3.347	0.00	3	0.84
comp_x	0.151	1.35	3	0.05
comp_y	0.114	6.04	4	0.12
cond_x	1.104	1.10	2	0.23
cond_y	0.416	7.20	3	0.29
cut	0.859	14.88	1	1.18
cutd	0.247	12.53	2	0.30
cut_strong	0.628	6.84	1	0.43
deallocate	1.670	8.00	1	1.26
decrement	0.047	0.00	2	0.01
divide_constant	0.026	0.00	6	0.01
escape (1)	1.119	23.62	2	2.60
execute	3.037	0.00	3	0.76
failure (2)	6.009	44.59	0	22.49
get_atom (3)	1.823	4.40	6	1.49
get_integer (3)	0.223	4.52	6	0.18
get_list	5.117	2.64	2	1.88
get_nil	0.500	3.20	2	0.20
get_structure	6.437	5.83	6	6.52
get_value_x	1.953	11.17	2	2.13
get_value_y	0.187	13.21	3	0.25
get_variable_x	0.560	0.00	2	0.09
get_variable_y	6.051	4.00	3	3.56
increment	0.234	0.00	2	0.04
jump	0.359	0.00	2	0.06
proceed	2.447	0.00	1	0.21
put_atom	0.254	0.00	6	0.13
put_integer	0.107	0.00	6	0.05
put_list	0.531	0.00	2	0.09
put_nil	0.049	0.00	2	0.01

Table A-2: Lcode Instruction Reference Characteristics

type	% instr	data bytes	instr bytes	% weight
put_value_x	2.647	0.00	2	0.44
put_value_y	6.878	4.00	3	4.04
put_structure	0.383	4.00	6	0.32
put_unsafe_integer_x	0.277	0.40	2	0.06
put_unsafe_integer_y	0.096	3.04	2	0.05
put_unsafe_value_y	1.617	8.61	3	1.57
put_variable_x	0.372	4.00	2	0.19
put_variable_y	2.475	4.00	3	1.45
retry	0.768	4.00	2	0.39
retry_me_else	2.133	4.00	2	1.07
switch_constant	0.867	0.61	10	0.75
switch_structure	0.914	4.72	10	1.12
switch_term	3.657	0.51	4	1.36
trust	0.267	7.93	2	0.22
trust_me_else	2.842	8.00	1	2.15
try	0.330	44.17	3	1.34
try_me_else	4.414	42.64	3	16.69
unify_atom	0.890	5.12	5	0.71
unify_integer	0.092	4.20	5	0.07
unify_nil	0.051	3.37	1	0.03
unify_value_x (4)	0.905	26.86	2	2.11
unify_value_y	0.042	6.74	2	0.05
unify_variable_x	6.257	4.00	2	3.15
unify_variable_y	2.627	8.00	2	2.20
unify_void	3.099	0.00	2	0.52
copy_atom (5)	0.396	4.00	5	0.30
copy_integer	0.270	4.00	5	0.20
copy_local_value_x (6)	0.230	6.33	2	0.18
copy_local_value_y	0.103	11.89	2	0.11
copy_nil	0.398	4.00	1	0.17
copy_value_x (7)	1.928	5.90	2	1.26
copy_value_y	0.912	10.65	2	0.94
copy_variable_x	1.794	4.00	2	0.90
copy_variable_y	1.110	8.00	2	0.93
copy_void	0.302	5.24	2	0.19

Table A-2: Lcode Instruction Reference Characteristics - *continued*

type	% instr	data bytes	instr bytes	% weight
procedure control	12.59	14.18	1.80	24.31
failure	6.36	38.24		21.32
head matching	20.94	6.75	3.44	13.91
structure matching	19.97	6.01	2.44	12.83
clause control	14.11	4.80	2.20	9.35
goal matching	14.15	2.45	3.25	8.77
unification	3.11	14.36		3.54
escape	1.49	16.66	2.00	3.00
indexing	7.55	3.78	2.75	2.89
arithmetic	0.39	0.00	3.80	0.09

Table A-3: Lcode Characteristics by Type

References

[1] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek.
ANLWAM: A Parallel Implmentation of the Warren Abstract Machine.
Internal Report, Argonne National Laboratory, Argonne, IL 60439, 1986.

[2] L. Byrd, F. C. N. Pereira, and D. H. D. Warren.
A Guide to Version 3 of DEC-10 PROLOG.
Technical Report 19, Dept. of Artificial Intelligence, University of Edinburgh, July, 1980.

[3] M. Carlsson.
Compilation for Tricia and its Abstract Machine.
Technical Report 35, UPMAIL, Uppsala University, September, 1986.

[4] T. P. Dobry, A. M. Despain, and Y. N. Patt.
Performance Studies of a Prolog Machine Architecture.
In *12th Annual International Symposium on Computer Architecture*, pages 180-190. IEEE Computer Society, December, 1985.

[5] B. Fagin and T. P. Dobry.
The Berkeley PLM Instruction Set: An Instruction Set for Prolog.
Research Report UCB/CSD 86/257, Computer Science Division, University of California at Berkeley, September, 1985.

[6] J. Gabriel, T. G. Lindholm, E. L. Lusk, and R. A. Overbeek.
A Tutorial on the Warren Abstract Machine for Computational Logic.
Research Paper ANL-84-84, Argonne National Laboratory, Argonne, IL 60439, June, 1985.

[7] J. Gee, S. W. Melvin, Y. N. Patt.
Advantages of Implementing Prolog by Microprogramming a Host General Purpose Computer.
In *Fourth International Conference on Logic Programming*. University of Melborne, MIT Press, May, 1987.

[8] M. V. Hermenegildo.
Restricted AND-Parallel Prolog and its Architecture.
Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[9] S. C. Johnson.
YACC - Yet Another Compiler Compiler.
Unix Programmer's Manual.

[10] M. E. Lesk and E. Schmidt.
LEX - Lexical Analyzer Generator.
Unix Programmer's Manual.

[11] J. Levy.
A GHC Abstract Machine and Instruction Set.
In *Third International Conference on Logic Programming*, pages 157-171. Imperial College, Springer-Verlag, July, 1986.

[12] H. Nakashima and K. Nakajima.
Hardware Architecture of the Sequential Inference Machine: PSI-II.
In *1987 International Symposium on Logic Programming*. IEEE Computer Society, August, 1987.

[13]
Quintus Prolog User's Guide and Reference Manual - Version 6.
Quintus Computer Systems Inc., Mountain View CA 94041.
April, 1986

[14] E. Tick and D. H. D. Warren.
Towards a Pipelined Prolog Processor.
In *1984 International Symposium on Logic Programming*. IEEE Computer Society, February, 1984.
also in New Generation Computing, 2(4):323-345.

[15] E. Tick.
Lisp and Prolog Memory Performance.
Technical Report CSL-TR-86-291, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, January, 1986.

[16] E. Tick.
Studies In Prolog Architectures.
PhD thesis, Stanford University, June, 1987.

[17] P. Van Roy.
A Prolog Compiler for the PLM.
Master's thesis, University of California at Berkeley, August, 1984.
also available as Technical Report UCB/CSD 84/203.

[18] D. H. D. Warren.
Applied Logic — Its Use and Implementation as Programming Tool.
PhD thesis, University of Edinburgh, 1977.
also available as SRI Technical Note 290.

[19] D. H. D. Warren.
An Improved Prolog Implementation which Optimises Tail Recursion.
Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[20] D. H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Report 309, Artificial Intelligence Center, SRI International, 1983.