AN ABSTRACT PROLOG INSTRUCTION SET

Technical Note 309

October 1983

By: David H.D. Warren, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

SRI Project 4776

Client: Digital Equipment Corporation

Open Publication. Release of Information.

SRI International

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

AN ABSTRACT PROLOG INSTRUCTION SET

David H D Warren
Artificial Intelligence Center
SRI International
31 August 1983

1. Introduction

This report describes an abstract Prolog instruction set suitable for software, firmware, or hardware implementation. The instruction set is abstract in that certain details of its encoding and implementation are left open, so that it may be realized in a number of different forms. The forms that are contemplated are:

- Translation into a compact bytecode, with emulators written in C (for maximum portability), Progol (a macrolanguage generating machine code, for efficient software implementations as an alternative to direct compilation on machines such as the VAX), and VAX-730 microcode.

- Compilation into the standard instructions of machines such as the VAX or DECsystem-10/20.

- Hardware (or firmware) emulation of the instruction set on a specially designed Prolog processor [3].

The abstract machine described herein ("new Prolog Engine") is a major revision of the "old Prolog Engine" described in a previous document. The new model overcomes certain difficulties in the old model, which are discussed in a later section. The new model can be considered to be a modification of the old model, where the stack contains compiler-defined goals called environments instead of user-defined goals. The environments correspond to some number of goals forming the tail of a clause. The old model was developed having primarily in mind a VAX-730 microcode implementation. The new model has, in addition, been influenced by hardware implementation considerations [3], but should remain equally amenable to software or firmware implementation on machines such as the VAX.

The new model is very similar to the abstract machine based on DEC-10 Prolog described by Warren [4], modified to incorporate tail recursion optimization [5]. The main differences are:

- Copying replaces structure-sharing as the means for constructing complex terms; however structure-sharing is still used to represent the goals constituting a resolvent.

- Choice points are separated from environments (local stack frames), and are created only when needed rather than at every procedure call.

- Environments are "trimmed" during execution (if the computation is determinate), by discarding variables no longer needed. This can be viewed as a generalization of tail recursion optimization.

- Potentially "unsafe" variables in the final goal of a clause are made global only if needed at runtime, rather than by default at compile time.

The architecture also has much in common with the abstract machine design of Bowen, Byrd, and Clocksin [1].

One of the main ways to realize the architecture in software or firmware is via a bytecode emulator, and this approach is stressed in this report. The design of the bytecode emulator calls for a large virtual memory, byte-addressable machine, and is particularly oriented towards the VAX architecture. Prolog run-time data structures are encoded as sequences of 32-bit words. Prolog programs are represented as sequences of instructions, encoded as sequences of 8-bit bytes. Each instruction consists of a one-byte operation code (opcode), followed by a number of arguments (typically one, two, or zero). An argument may be 1, 2, or 4 bytes long.

The bytecode emulator comprises a large number of small routines defining the different operations. Execution proceeds from one routine to the next by dispatching on the opcode of the next instruction. Some instructions can be executed in two different modes ("read" mode or "write" mode), so there is a separate routine for each mode.

An earlier version of the emulator (for the old Engine design) has been implemented in a Prolog-based macro language called Progol, which was used to generate a VAX machine code version. The Progol implementation should be fairly easy to transport to a variety of machines to give efficient software implementations. A transliteration of this Progol code into C has been performed. The primary intention, however, behind the Progol form of the emulator was that it should serve as a model for a microcode implementation on a VAX-730 or other suitable machine.

2. Data Objects

A Prolog term is represented by a word containing a value (which is generally an address) and a tag. (Possible formats for these words are given in Appendix V.) A large address space is assumed, with values occupying around 32 bits. The tag distinguishes the type of term, and must be at least 2 bits and preferably up to 8 bits. The main types are references (corresponding to bound or unbound variables), structures, lists, and constants (including atoms and integers). An unbound variable is represented by a reference to itself. It could be distinguished by a separate tag in a hardware implementation.

Structures and lists are represented in a non-structure-sharing manner, i.e., they are created by explicitly copying the functor and arguments into consecutive words of memory. For efficiency, lists have a separate tag from structures, and so no functor needs to be stored.

3. Data Areas

The main data areas are the code area, containing instructions and other data representing the program itself, and three areas operated as stacks, the (local) **stack**, the **heap** (or global stack), and the **trail**. (There is also a small push-down list (PDL) used for unification). The stacks generally expand with each procedure invocation, and they contract on backtracking. In addition, **tail recursion optimization** removes information from the local stack when executing the last procedure call in a determinate procedure, and the cut operator excises backtracking information from both the local stack and the trail.

The different areas are laid out in memory as follows:

```
low                                                                 high
 ________________________________________________________________________
| code area      |->    | heap        |->   | stack    |->   | trail |->    <-| PDL |
|________________|______|_____________|_____|__________|_____|_______|________|_____|
        :                           :    :           :   :          :
        P                          HB    H           B   E          TR
```

It turns out to be important that the stack and heap are arranged as shown (since then the simple strategy of always binding the variable with the lowest address when making a variable-variable binding is sufficient to prevent dangling references).

The heap contains all the structures and lists created by unification and procedure invocation. The trail contains references to variables that have been bound during unification and that must be unbound on backtracking. The stack contains two kinds of objects: **environments** and **choice points** (whose formats are given in Appendix IV). An environment consists of a vector of value cells for variables occurring in the body of some clause, together with a **continuation** comprising a pointer into the body of another clause and its associated environment. In effect, a continuation represents a list of (instantiated) goals still to be executed. A choice point contains all the information necessary to restore an earlier state of computation in the event of backtracking. It is created when entering a procedure if (and only if) the procedure has more than one clause which can potentially match the call. The information that is stored is a pointer to the alternative clauses, plus the values of the following registers (see below) at the time the procedure is entered: H, TR, B, CP, E, and A1 to Am where m is the number of arguments of the procedure.

4. Registers and Treatment of Variables

The current state of a Prolog computation is defined by certain registers containing pointers into the main data areas (cf. Appendices II and III). The main registers are as follows:

```
P          program pointer (to the code area)
CP         continuation program pointer (to the code area)

E          last environment (on the local stack)
B          last choice point (backtrack point) (on the local stack)
A          top of stack (not strictly essential)

TR         top of trail
H          top of heap
HB         heap backtrack point (i.e., the H value corresponding to B)

S          structure pointer (to the heap)

A1, A2, ...       argument registers
X1, X2, ...       temporary variables
```

The A registers and X registers are, in fact, identical; the different names merely reflect different usages. The A registers are used to pass the arguments to a procedure. The X registers are used to hold the values of a clause's temporary variables.

A **temporary** variable is a variable that has its first occurrence in the head or in a structure or in the last goal, and that does not occur in more than one goal in the body, where the head of the clause is counted as part of the first goal. Temporary variables do not need to be stored in the clause's environment.

A **permanent** variable is any variable not classified as a temporary variable. Permanent variables are stored in an environment and are addressed by offsets from the environment pointer. They are referred to as **Y1**, **Y2**, etc. Note that there can be no permanent variables in clauses with less than two goals in the body, and, therefore, such clauses do not need environments. Permanent variables are arranged in their environment in such a way that they can be discarded as soon as they are no longer needed. This "trimming" of the environment only has real effect when the environment is more recent than the last choice point.

5. The Instruction Set

Prolog programs are encoded as sequences of Prolog instructions. In general, there is one instruction for each Prolog symbol. An instruction consists of an operation code (**opcode**) with some operands (typically just one). The opcode generally encodes the type of Prolog symbol together with the context in which it occurs. It need occupy no more than one byte (eight bits). The operands include small integers, offsets, and addresses, which identify the different kinds of Prolog symbol. Depending on the details of the encoding, operands might occupy one, two, or four bytes, or in some cases less than one byte.

The Prolog instruction set can be classified into *get* instructions, *put* instructions, *unify* instructions, *procedural* instructions, and *indexing* instructions. (The instruction set is summarized in Appendix I.)

The *get* instructions correspond to the arguments of the head of a clause and are responsible for matching against the procedure's arguments given in the **A** registers. The main instructions are:

```
get_variable Yn,Ai        get_variable Xn,Ai
get_value Yn,Ai           get_value Xn,Ai
get_constant C,Ai         get_nil Ai
get_structure F,Ai        get_list Ai
```

Here (and in the description of other classes of instructions, below) **Ai** represents the argument register concerned, and **Xn**, **Yn**, **C**, and **F** represent, respectively, a temporary variable, a permanent variable, a constant, and a functor. The **get_variable** instruction is used if the variable is currently uninstantiated (i.e., if this is the first occurrence of the variable in the clause). Otherwise the **get_value** instruction is used.

The *put* instructions correspond to the arguments of a goal in the body of a clause and are responsible for loading the arguments into the **A** registers. The main instructions are:

```
put_variable Yn,Ai        put_variable Xn,Ai
put_value Yn,Ai           put_value Xn,Ai
put_unsafe_value Yn,Ai
put_constant C,Ai         put_nil Ai
put_structure F,Ai        put_list Ai
```

The **put_unsafe_value** instruction is used in place of the **put_value** instruction in the last goal in which an unsafe variable appears. An **unsafe** variable is a permanent variable that did not first occur in the head or in a structure, i.e., the variable was initialized by a **put_variable** instruction. The **put_unsafe_value** instruction ensures that the unsafe variable is dereferenced to something other than a reference to the current environment, binding the variable to a new value cell on the heap, if necessary, thus "globalizing" the variable. This measure is necessary to prevent possible dangling references to a part of the environment about to be discarded by the **execute** or **call** instruction which follows.

The *unify* instructions correspond to the arguments of a structure (or list) and are responsible both for unifying with existing structures and for constructing new structures. The main instructions are:

```
unify_void N
unify_variable Yn         unify_variable Xn
unify_value Yn            unify_value Xn
unify_local_value Yn      unify_local_value Xn
unify_constant C          unify_nil
```

The **unify_void N** instruction represents a sequence of **N** single-occurrence variables; no temporary or permanent variable cell is needed for such "void" variables. The **unify_local_value** instruction is used in place of the **unify_value** instruction if the variable has *not* been initialized to a global value (by, for example, a **unify_variable** instruction).

A sequence of *unify* instructions is preceded by an instruction to *get* or *put* a structure or list. This preceding instruction determines one of two modes, **read mode** or **write mode**, that the following unify instructions will be executed in. In read mode, ***unify*** instructions perform unification with successive arguments of an existing structure, addressed via the `S` register. In write mode, ***unify*** instructions construct the successive arguments of a new structure, addressed via the `H` register.

Nested substructures or sublists are translated as follows. If the substructure or sublist occurs in the head, it is translated by a **`unify_variable Xn`** instruction followed, after the end of the current *unify* sequence, by a corresponding **`get_structure F,Xn`** or **`get_list Xn`** instruction. If the substructure or sublist occurs in the body, it is translated by a **`unify_value Xn`** instruction preceded, before the start of the current *unify* sequence, by a corresponding **`put_structure F,Xn`** or **`put_list Xn`** instruction.

The *procedural* instructions correspond to the predicates that form the head and goals of the clause and are responsible for the control transfer and environment allocation associated with procedure calling. The main instructions are:

```
proceed                 allocate
execute P               deallocate
call P,N
```

where `P` represents a predicate and `N` is the number of variables (still in use) in the environment. The procedural instructions are used in the translation of clauses with zero, one, or two or more goals in the body as follows:

`P.`	`P :- Q.`	`P :- Q, R, S.`
get args of `P`	*get args of* `P`	**`allocate`**
`proceed`	*put args of* `Q`	*get args of* `P`
	`execute Q`	*put args of* `Q`
		`call Q,N`
		put args of `R`
		`call R,N1`
		put args of `S`
		`deallocate`
		`execute S`

Note that the size of an environment is specified dynamically by the **`call`** instruction. The size always decreases, so `N1` is less than or equal to `N`.

The *indexing* instructions link together the different clauses that make up a procedure and are responsible for filtering out a subset of those clauses that could potentially match a given procedure call. This filtering, or indexing, function is based on a **key** which is the principal functor of the first argument of the procedure (given in register A1). The main instructions are:

```
try_me_else L          try L          switch_on_term Lv,Lc,Ll,Ls
retry_me_else L        retry L        switch_on_constant N,Table
trust_me_else fail     trust L        switch_on_structure N,Table
```

Here L, Lv, Lc, Ll, Ls are addresses of clauses (or sets of clauses), and **Table** is a hash table of size **N**.

Each clause is preceded by a **try_me_else**, **retry_me_else**, or **trust_me_else** instruction, depending on whether it is the first, an intermediate, or the last clause in the procedure. These instructions are executed only in the case that **A1** dereferences to a variable and all clauses have to be tried for a match. The operand L is the address of the following clause.

The **switch_on_term** instruction dispatches to one of four addresses, Lv, Lc, Ll, Ls, depending on whether **A1** dereferences to a variable, a constant, a list, or a structure. Lv will be the address of the **try_me_else** (or **trust_me_else**) instruction, which precedes the first clause in the procedure. Ll will be either the address of the single clause whose key is a list, or the address of a sequence of such clauses, identified by a sequence of **try**, **retry**, and **trust** instructions. Lc and Ls may be the addresses of a single clause or sequence of clauses (as in the case of Ll), or more generally may be, respectively, the address of a **switch_on_constant** or **switch_on_structure** instruction, which provides hash table access to the clause or clauses that match the given key.

6. Optimizations

Since the argument registers and the temporary registers are identical, certain instructions are null operations and can be omitted:

```
get_variable Xi,Ai
put_value Xi,Ai
```

The compiler takes pains to allocate temporary variables to **X** registers in such a way as to maximize the scope for this optimization.

Note also that the following instructions denote the same operation of simply transferring the contents of register **Xi** to register **Xj**:

```
get_variable Xj,Ai
put_variable Xi,Aj
```

7. Examples of Clause Encoding

As examples of clause encoding, here is the code for the *concatenate* and *quick sort* procedures.

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

concatenate/3: switch_on_term C1a,C1,C2,fail

C1a:    try_me_else C2a              % concatenate(
C1:     get_nil A1                   %    [],
        get_value A2,A3              %    L,L
        proceed                      % ).

C2a:    trust_me_else fail           % concatenate(
C2:     get_list A1                  %    [
        unify_variable X4            %       X|
        unify_variable A1            %       L1], L2,
        get_list A3                  %    [
        unify_value X4               %       X|
        unify_variable A3            %       L3]) :-
        execute concatenate/3        % concatenate(L1,L2,L3).


qsort([],R,R).
qsort([X|L],R0,R) :-
   split(L,X,L1,L2), qsort(L1,R0,[X|R1]), qsort(L2,R1,R).

qsort/3: switch_on_term C1a,C1,C2,fail

C1a:    try_me_else C2a              % qsort(
C1:     get_nil A1                   %    [],
        get_value A2,A3              %    R,R
        proceed                      % ).

C2a:    trust_me_else fail           % qsort(
C2:     allocate
        get_list A1                  %    [
        unify_variable Y6            %       X|
        unify_variable A1            %       L],
        get_variable Y5,A2           %    R0,
        get_variable Y3,A3           %    R) :-
        put_value Y6,A2              % split(L,X,
        put_variable Y4,A3           %    L1,
        put_variable Y1,A4           %    L2
        call split/4,6               % ),
        put_unsafe_value Y4,A1       % qsort(L1,
        put_value Y5,A2              %    R0,
        put_list A3                  %    [
```

```
unify_value Y6                  %       X|
unify_variable Y2               %       R1]
call qsort/3,3                  % ),
put_unsafe_value Y1,A1          % qsort(L2,
put_value Y2,A2                 %    R1,
put_value Y3,A3                 %    R
deallocate
execute qsort/3                 % ).
```

The following example further illustrates the handling of permanent variables:

```
compile(Clause,Instructions) :-
   preprocess(Clause,C1),
   translate(C1,Symbols),
   number_variables(Symbols,0,N,Saga),
   complete_saga(0,N,Saga),
   allocate_registers(Saga),
   generate(Symbols,Instructions).
```

```
try_me_else fail                % compile( Clause,
allocate
get_variable Y2,A2              %    Instructions) :-
put_variable Y5,A2              % preprocess(Clause,C1
call preprocess/2,5             % ),
put_unsafe_value Y5,A1          % translate(C1,
put_variable Y1,A2              %    Symbols
call translate/2,4              % ),
put_value Y1,A1                 % number_variables(Symbols,
put_constant 0,A2               %    0,
put_variable Y4,A3              %    N,
put_variable Y3,A4              %    Saga
call number_variables/4,4       % ),
put_constant 0,A1               % complete_saga(0,
put_unsafe_value Y4,A2          %    N,
put_variable Y3,A3              %    Saga
call complete_saga/3,3          % ),
put_unsafe_value Y3,A1          % allocate_registers(Saga
call allocate_registers/1,2     % ),
put_unsafe_value Y1,A1          % generate(Symbols,
put_value Y2,A2                 %    Instructions
deallocate
execute generate/2              % ).
```

The following two examples illustrate the encoding of nested substructures:

```
d(U*V,X,(DU*V)+(U*DV)) :- d(U,X,DU), d(V,X,DV).
```

```
try_me_else ...                 % d(
get_structure '*'/2,A1          %    *(
```

```
        unify_variable A1               %        U,
        unify_variable Y1               %        V),
        get_variable Y2,A2              %     X,
        get_structure '+'/2,A3          %     +(
        unify_variable X4               %        SS1,
        unify_variable X5               %        SS2),
        get_structure '*'/2,X4          % SS1 = *(
        unify_variable A3               %            DU,
        unify_value Y1                  %            V),
        get_structure '*'/2,X5          % SS2 = *(
        unify_value A1                  %            U,
        unify_variable Y3               %            DV)) :-
        call d/3,3                      % d(U,X,DU),
        put_value Y1,A1                 % d(V,
        put_value Y2,A2                 %     X,
        put_value Y3,A3                 %     DV
        execute d/3                     % ).
```

```
test :- do(parse(s(np,vp),[birds,fly],[])).

        trust_me_else fail              % test :-
        put_structure s/2,X2            % do( SS1 = s(
        unify_constant np               %        np,
        unify_constant vp               %        vp),
        put_list X4                     % SS2 = [
        unify_constant fly              %        fly|
        unify_nil                       %        []],
        put_list X3                     % SS3 = [
        unify_constant birds            %        birds|
        unify_value X4                  %        SS2],
        put_structure parse/3,A1        %     parse(
        unify_value X2                  %        SS1,
        unify_value X3                  %        SS2,
        unify_nil                       %        [])
        execute do/1                    % ).
```

The following example illustrates the use of the indexing instructions:

```
call(X or Y) :- call(X).
call(X or Y) :- call(Y).
call(trace) :- trace.
call(notrace) :- notrace.
call(nl) :- nl.
call(X) :- builtin(X).
call(X) :- ext(X).
call(call(X)) :- call(X).
call(repeat).
call(repeat) :- call(repeat).
```

```
call(true).

call/1: try_me_else C6a
        switch_on_type C1a,L1,fail,L2

L1:     switch_on_constant 4, $(trace: C3,
                                notrace: C4,
                                fail,
                                nl: C5)
L2:     switch_on_structure 1, $(or/2: L3)

L3:     try C1
        trust C2

C1a:    try_me_else C2a                 % call(
C1:     get_structure or/2,A1           %    or(
        unify_variable A1               %       X,Y)) :-
        execute call/1.                 % call(X).

C2a:    retry_me else C3a               % call(
C2:     get_structure or/2,A1           %    or(
        unify_void 1                    %       X,
        unify_variable A1               %       Y)) :-
        execute call/1                  % call(Y).

C3a:    retry_me_else C4a               % call(
C3:     get_constant trace,A1           %    trace) :-
        execute trace/0                 % trace.

C4a:    retry_me_else C5a               % call(
C4:     get_constant notrace,A1         %    notrace) :-
        execute notrace/0               % notrace.

C5a:    trust_me_else fail              % call(
C5:     get_constant nl ,A1             %    nl) :-
        execute nl/0                    % nl.

C6a:    retry_me_else C7a               % call(X) :-
        execute builtin/1               % builtin(X).

C7a:    retry_me_else L4                % call(X) :-
        execute ext/1                   % ext(X).

L4:     trust_me_else fail
        switch_on_type C8a,L5,fail,L7

L5:     switch_on_constant 2, $(repeat: L6, true: C11)

L6:     try C9
```

```
        trust C10

L7:     switch_on_structure 1, $(call/1: C8)

C8a:    try_me_else C9a              % call(
C8:     get_structure call/1,A1      %    call(
        unify_variable A1            %       X)) :-
        execute call/1               % call(X).

C9a:    retry_me_else C10a           % call(
C9:     get_constant repeat,A1       %    repeat
        proceed                      % ).

C10a:   retry_me_else C11a           % call(
C10:    get_constant repeat,A1       %    repeat) :-
        put_constant repeat,A1       % call(repeat
        execute call/1               % ).

C11a:   trust_me_else fail           % call(
C11:    get_constant true,A1         %    true
        proceed                      % ).
```

8. Description of Instructions and Basic Operations

Note: In the descriptions that follow, **Vn** is used generically to denote either a permanent variable **Yn** or a temporary variable **Xn**. Some of the descriptions are followed by algorithmic code for the operation performed, for the simpler cases.

8.1. Control Instructions

allocate This instruction appears at the beginning of a clause with more than one goal in the body. (It can, in fact, be placed anywhere before the first occurrence of a permanent variable). Space for the new environment is allocated on the stack after the last choice point or environment, the continuation is saved, and **E** is set to point to the new environment.

```
CE := E
E := (CE < B -> B | CE + env_size(CP))
CP(E) := CP
CE(E) := CE
```

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.

```
CP := CP(E)
E  := CE(E)
```

`call Proc,N` This instruction terminates a body goal and is responsible for setting CP to the following code, and the program pointer P to the procedure. N is the number of variables in the environment at this point. It is accessed as an offset from CP by certain instructions in the called procedure.

```
CP := following code
P  := Proc
```

`execute Proc` This instruction terminates the final goal in the body of a clause. The program pointer P is set to point to the procedure.

```
P := Proc
```

`proceed` This instruction terminates a unit clause. The program pointer P is reset to the continuation pointer CP.

```
P := CP
```

8.2. Put Instructions

`put_variable Yn,Ai`

This instruction represents a goal argument that is an unbound (permanent) variable. The instruction puts a reference to permanent variable Yn into the register Ai, and also initializes Yn with the same reference.

```
Ai := Yn := ref_to(Yn)
```

`put_variable Xn,Ai`

This instruction represents an argument of the final goal that is an unbound variable. The instruction creates an unbound variable on the heap, and puts a reference to it into registers Ai and Xn.

```
Ai := Xn := next_term(H) := tag_ref(H)
```

`put_value Vn,Ai` This instruction represents a goal argument thata is a bound variable. The instruction simply puts the value of variable Vn into the register Ai.

```
Ai := Vn
```

put_unsafe_value Yn,Ai

This instruction represents the last occurrence of an unsafe variable. The instruction dereferences **Yn** and puts the result in register **Ai**. If **Yn** dereferences to a variable in the current environment, that variable is bound to a new global variable created on the heap, the binding is trailed if necessary, and register **Ai** is set to a reference to the new global variable.

put_const C,Ai This instruction represents a goal argument that is a constant. The instruction simply puts the constant **C** into register **Ai**.

```
Ai := C
```

put_nil Ai This instruction represents a goal argument that is the constant **[]**. The instruction simply puts the constant **[]** into register **Ai**.

```
Ai := nil
```

put_structure F,Ai

This instruction marks the beginning of a structure (without embedded substructures) occurring as a goal argument. The instruction pushes the functor **F** for the structure onto the heap, and puts a corresponding structure pointer into register **Ai**. Execution then proceeds in "write" mode.

```
Ai := tag_struct(H)
next_term(H) := F
```

put_list Ai This instruction marks the beginning of a list occurring as a goal argument. The instruction places a list pointer corresponding to the top of the heap into register **Ai**. Execution then proceeds in "write" mode.

```
Ai := tag_list(H)
```

8.3. Get Instructions

get_variable Vn,Ai

This instruction represents a head argument that is an unbound variable. The instruction simply gets the value of register **Ai** and stores it in variable **Vn**.

Vn := Ai

get_value Vn,Ai This instruction represents a head argument that is a bound variable. The instruction gets the value of register **Ai** and unifies it with the contents of variable **Vn**. The fully dereferenced result of the unification is left in variable **Vn** if **Vn** is a temporary.

get_constant C,Ai

This instruction represents a head argument that is a constant. The instruction gets the value of register **Ai** 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. Otherwise, the result is compared with the constant **C**, and if the two values are not identical, backtracking occurs.

get_nil Ai This instruction represents a head argument that is the constant **[]**. The instruction gets the value of register **Ai** and dereferences it. If the result is a reference to a variable, that variable is bound to the constant **[]**, and the binding is trailed if necessary. Otherwise, the result is compared with the constant **[]**, and if the two values are not identical, backtracking occurs.

get_structure F,Ai

This instruction marks the beginning of a structure (without embedded substructures) occurring as a head argument. The instruction gets the value of register **Ai** and dereferences it. If the result is a reference to a variable, that variable is bound to a new structure pointer pointing at the top of the heap, and the binding is trailed if necessary, functor **F** is pushed onto the heap, and execution proceeds in "write" mode. Otherwise, if the result is a structure and its functor is identical to functor **F**, the pointer **S** is set to point to the arguments of the structure, and execution proceeds in "read" mode.

Otherwise, backtracking occurs.

`get_list Ai` This instruction marks the beginning of a list occurring as a head argument. The instruction gets the value of register **Ai** and dereferences it. If the result is a reference to a variable, that variable is bound to a new list pointer pointing at the top of the heap, the binding is trailed if necessary, and execution proceeds in "write" mode. Otherwise, if the result is a list, the pointer **S** is set to point to the arguments of the list, and execution proceeds in "read" mode. Otherwise, backtracking occurs.

8.4. Unify Instructions

`unify_void N` This instruction represents a sequence of **N** head structure arguments that are single occurrence variables. If the instruction is executed in "read" mode, it simple skips the next **N** arguments from **S**. If the instruction is executed in "write" mode, it pushes **N** new unbound variables onto the heap.

```
In read mode:
        S := S + N*word_width
In write mode:
        next_term(H) := tag_ref(H)
        ... {repeated N times}
```

`unify_variable Vn`

This instruction represents a head structure argument that is an unbound variable. If the instruction is executed in "read" mode, it simply gets the next argument from **S** and stores it in variable **Vn**. If the instruction is executed in "write" mode, it pushes a new unbound variable onto the heap, and stores a reference to it in variable **Vn**.

```
In read mode:
        Vn := next_term(S)
In write mode:
        Vn := next_term(H) := tag_ref(H)
```

`unify_value Vn` This instruction represents a head structure argument that is a variable bound to some global value. If the instruction is executed in "read" mode, it gets the next argument from **S**, and unifies it with the value in variable **Vn**, leaving the dereferenced result in **Vn** if **Vn** is a temporary. If the instruction is executed in "write" mode, it pushes the value of variable **Vn** onto the heap.

```
In write mode:
        next_term(H) := Vn
```

`unify_local_value Vn`

This instruction represents a head structure argument that is a variable bound to a value that is not necessarily global. The effect is the same as **unify_value**, except that, in "write" mode, it dereferences the value of variable **Vn** and only pushes the result onto the heap if the result is not a reference to a variable on the stack. If the result is a reference to a variable on the stack, a new unbound variable is pushed onto the heap, the variable on the stack is bound to a reference to the new variable, the binding is trailed if necessary, and variable **Vn** is set to point to the new variable if **Vn** is a temporary.

`unify_constant C`

This instruction represents a head structure argument that is a constant. If the instruction is executed in "read" mode, it 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 nonreference value, that value is compared with the constant **C** and backtracking occurs if the two values are not identical. If the instruction is executed in "write" mode, the constant **C** is pushed onto the heap.

```
In write mode:
        next_term(H) := C
```

8.5. Indexing Instructions

`try_me_else L` This instruction precedes the code for the first clause in a procedure with more than one clause. A choice point is created by saving the following **n+6** values on the stack: registers **An** through **A1**, the current environment pointer **E**, the current continuation **CP**, a pointer to the previous choice point **B**, the address **L** of the next clause, the current trail pointer **TR**, and the current heap pointer **H**. **HB** is set to the current heap pointer, and **B** is set to point to the current top of stack.

`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 is updated with the address L of the next clause.

```
BP(B) := L
```

`trust_me_else fail`

This instruction precedes the code for the last clause in a procedure. (The argument of the instruction is arbitrary, but exists simply to reserve space in the instruction in order to facilitate the asserting and retracting of clauses). The current choice point is discarded, and registers B and HB are reset to correspond to the previous choice point.

```
B := B(B)
HB := H(B)
```

`try L` This instruction is the first of a sequence of instructions identifying clauses with the same key. A choice point is created by saving the following **n+6** values on the stack: registers **An** through **A1**, the current environment pointer **E**, the current continuation **CP**, a pointer to the previous choice point **B**, the address of the following instruction (alternative clauses), the current trail pointer **TR**, and the current heap pointer **H**. **HB** is set to the current heap pointer, and **B** is set to point to the current top of stack. Finally, the program pointer **P** is set to the clause address L.

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

```
BP(B) := following code
P := L
```

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

```
B := B(B)
HB := H(B)
P := L
```

switch_on_term Lv,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 **A1** is dereferenced and, depending on whether the result is a variable, constant, (non-empty) list, or structure, the program pointer **P** is set to **Lv**, **Lc**, **Ll**, or **Ls**, respectively.

switch_on_constant N,Table

This instruction provides hash table access to a group of clauses having constants in the first head argument position. Register **A1** holds a constant, whose value is hashed to compute an index in the range 0 to N-1 into the hash table **Table**. The size of the hash table is N, which is a power of 2. The hash table entry gives access to the clause or clauses whose keys hash to that index. The constant in **A1** is compared with the different keys until one is found that 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.

switch_on_structure N,Table

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_on_constant**, except that the key used is the principal functor of the structure in **A1**.

8.6. Other Basic Operations

fail — This operation is performed when a failure occurs during unification. It causes backtracking to the most recent choice point. The trail is "unwound" as far as the choice point trail pointer, by popping references off the trail and resetting the variables they address to unbound. Registers H, **A**, and C are restored to the values saved in the choice point. The program pointer P is set to the next alternative clause as recorded in the choice point.

trail(R) — This operation is performed when a variable, whose reference is R, is bound during unification. If the variable is in the heap and is before the heap backtrack point **HB**, or the variable is in the stack and is before the stack backtrack point **B**, the reference **R** is pushed onto the trail. Otherwise, no action is taken.

9. Encoding of Instructions

The instructions could be encoded in various ways. A possible encoding, suitable for software emulation, is shown in Appendix VI.

Each opcode occupies a single byte. This is followed, in the case of *get* and *put* instructions, by another byte giving the number of the **A** register concerned. Other arguments are encoded as follows.

Temporary or permanent variable numbers are encoded as a single byte. Constants are encoded by giving their full-word (32-bit) value (including tag). Special opcodes may be provided to support a half-word (16-bit) representation, in cases where the constant value can be obtained by sign-extending a 16-bit value. Functors are encoded as a 16-bit functor number, which is used to index into a functor table to obtain the full-word representation of the functor. Predicates and clause addresses are represented as 16-bit offsets into the current **segment** of the address space, i.e., the full address of the corresponding procedure or clause is obtained by appending the 16-bit offset to the top 16 bits of the address of the current instruction. Some escape mechanism must be provided in order to cross segment boundaries.

It is assumed that 16-bit and 32-bit arguments do not have to be specially aligned, as is allowed on the VAX. On machines that require alignment, dummy one-byte skip instructions can be inserted by the compiler to provide the correct alignment.

An important optimization of the instruction set would be to provide opcodes that build in the values of certain small numeric arguments, making the instructions shorter (and probably faster). The main candidates for this optimization are the one-byte arguments giving the number **n** of a register **An**, **Xn**, or **Yn**, where **n** is small. For example, **get_list A3** might be repaced by a new instruction **get_list_3**.

10. Environment Stacking versus Goal Stacking

The present design is an **environment-stacking** model. Although it is a non-structure-sharing implementation as far as terms are concerned, structure-sharing is still used to represent the goals on the stack.

An earlier version of the design used a **goal-stacking** model. The goal-stacking model differs from all existing Prolog implementations, that I know of, in that there is *no* structure-sharing whatsoever. Not only are constructed terms (structures) represented explicitly, but goals are too. The goal stack contains an explicit representation of the list of goals remaining to be executed. This list is just the "resolvent" of traditional resolution theory. There is no need to store vectors of variable cells representing binding environments.

The advantages of the goal stacking model are:

- Implementation simplicity. The implementation (i.e., kernel code, microcode, or specialized hardware) should be smaller.

- Garbage collection is more straightforward (and Bruynooghe's 1982 optimization [2] follows by default).

- Tail recursion optimization is much simpler and is applicable at *every* procedure call--one simply discards the calling goal if it is later than the last choice point.

- All variables in a clause are "temporaries" and can correspond directly to hardware registers.

- Once resolution with a clause is complete, there is no further reference to the code for that clause. This will tend to reduce paging in a virtual memory system. In contrast, structure-sharing (full or partial) tends to cause random accesses to the code area.

- Related to the previous item, there are no jumps within a clause. Fewer jumps mean better performance on pipelined hardware.

However, goal stacking also has significant disadvantages relative to environment stacking:

- Time can be wasted in unnecessary copying, particularly when a clause is entered and then fails early in the body. This disadvantage is not too severe, however, since copying can be relatively fast, compared with other overheads.

- The stack size is less stable, so there is less scope for optimizations that buffer the top of stack in registers or fast memory.

- As each goal is popped off the stack, one has to check for unsafe variables to avoid dangling references. There does not seem to be an elegant solution to this problem.

- It is difficult to optimize the body code. Once the goal has been copied onto the stack, it is hard to take special actions. This makes it awkward to handle arithmetic expressions and frustrates the possibility of checking for unsafe variables in the body code.

- Disjunction is awkward to handle, for similar reasons.

- The representation of goals on the stack is less compact than the environment model, at least in the present refinement of the environment model where environments are trimmed during execution.

The problem of dealing with unsafe variables is particularly severe, since it involves much checking at runtime, which can be largely avoided in the environment-stacking model by compile-time analysis generating special instructions only where needed. For this reason, the goal-stacking model was dropped in favor of the environment stacking model.

However the environment-stacking model has been strongly influenced by the earlier design and can be viewed as a source-language-level variation of goal stacking. From this point of view, an environment is a compiler-generated goal corresponding to the tail of a clause. A clause:

```
P :- Q, R, S.
```

is viewed as being transformed into:

```
P :- Q, Z1.
Z1 :- R, Z2.
Z2 :- S.
```

where Z1, Z2 correspond to successive states of the environment.

11. Pros and Cons of Copying Nondeterminate Environments

With the present model, the current environment is not necessarily at the top of the stack. It may have become "buried" by subsequent choice points. Leaving it in its original position conserves space and avoids copying overheads. However, there would be a number of advantages in copying "buried" environments to the top of the stack:

- Permanent and temporary variables can be accessed uniformly as offsets from the top of stack.
- Environments can be modified as well as trimmed, allowing them to be smaller.
- When dereferencing a variable, it is permissible to modify it; i.e., permanent variables can be treated just like temporary variables.
- If copying of the environment includes relocating any self references, then a lot of trailing will be avoided.
- Memory accesses are less random, improving performance of paging and stack buffering.

For a software implementation, these advantages do not appear to outweigh the copying overhead. However, the tradeoffs may well be different for a firmware or hardware implementation.

12. Acknowledgements

This work was supported by a Digital Equipment Corporation external research grant. I would like to particularly thank the following for making possible and encouraging this research: Peter Jessel, Nils Nilsson, Michael Poe, and Daniel Sagalowicz.

Appendix

I. Summary of Instructions

	HEAD	BODY
PROCEDURAL	proceed	execute P call P,N
	allocate	deallocate
GET/PUT	get_variable Xn,Ai	put_variable Xn,Ai
	get_variable Yn,Ai	put_variable Yn,Ai
	get_value Xn,Ai	put_value Xn,Ai
	get_value Yn,Ai	put_value Yn,Ai
		put_unsafe_value Yn,Ai
	get_constant C,Ai	put_constant C,Ai
	get_nil Ai	put_nil Ai
	get_structure F,Ai	put_structure F,Ai
	get_list Ai	put_list Ai
UNIFY	unify_void N	
	unify_variable Xn	
	unify_variable Yn	
	unify_local_value Xn	
	unify_local_value Yn	
	unify_value Xn	
	unify_value Yn	
	unify_constant C	
	unify_nil	
INDEXING	try_me_else L	try L
	retry_me_else L	retry L
	trust_me_else fail	trust L
	switch_on_term Lv,Lc,Ll,Ls	
	switch_on_constant N,Table	
	switch_on_structure N,Table	

II. Prolog Machine State (during unification)

```
                 stack                          heap                      trail
             :                :         :                 :        :                :
             :                :         :                 :        :                :
             |________________| HB->H'  |_________________|   TR' |________________|
  p   c      |             Am |         |                 |        |                |
  o   h      | goal         : |         |                 |        | bound          |
  s   o      | arguments   A2 |         |                 |        | variable       |
  s   i      |_ _ _ _ _ _ _A1_|         |                 |        | addresses      |
  i   c      | contin.    BCE |         | structures      |        |                |
  b   e      |_ _ _ _ _ _ BCP_|         |                 |  TR -> |________________|
  l          | backtrack   B' |         |                 |
  e   p      | state       BP |    H -> |_________________|
      t.     |            TR' |
 E,B ->      |_____________H'_|
      e      | contin.     CE |
      n      |_ _ _ _ _ _ _CP_|
      v      |             Y1 |
      i      | permanent   Y2 |
      r      | variables    : |
      o      |              : |
      n      |_____________Yn_|
      m
      e       ________________
      n      |             X1 |
      t      | temporary    : |
             | variables    : |
             |____________Xm'_|
             |             Am |
             | goal         : |
             | arguments   A2 |
  A   ->     |_____________A1_|
 (PDL)       | unification    |
             | arguments      |
             :                :
```

III. Stack State (during procedure call)

```
        determinate call                 nondeterminate call
        |              |                 |              |
        |______________|                 |              |
        |              |            E -> |______________|
        | choice point |                 |              |
   B -> |______________|                 | environment  |
        |              |                 |              |
        |              |                 |______________|
   E -> |______________|                 |              |
        |              |                 |______________|
        | environment  |                 |              |
        |              |                 | choice point |
        |______________|            B -> |______________|
```

IV. Run-Time Structure Formats

```
ENVIRONMENT                                 STRUCTURE (COMPLEX TERM)

 _______________                              _______________
| cont. env.    |   (CE)                     | functor       |
|_______________|                            |_______________|
| cont. code    |   (CP)                     | argument 1    |
|_______________|                            |     :         |
| variable 1    |   (Y1)                     |     :         |
|    :          |                            | argument N    |
|    :          |                            |_______________|
| variable N    |   (Yn)
|_______________|


CHOICE POINT

 _______________
| goal arg. M   |   (Am)
|    :          |
|    :          |
| goal arg. 1   |   (A1)
|_______________|
| cont. env.    |   (BCE)
|_______________|
| cont. code    |   (BCP)
|_______________|
| prev. choice  |   (B')
|_______________|
| next clause   |   (BP)
|_______________|
| trail point   |   (TR')
|_______________|
| heap point    |   (H')
|_______________|
```

V. Data Formats (provisional)

```
           Value / Address                                    Tag

bit:    32                                                  2      0
         ________________________________________________________
        |               reference address                  | 0 0 |
        |__________________________________________________|_____|

         ________________________________________________________
        |               structure (or box) address         | 0 1 |
        |__________________________________________________|_____|

         ________________________________________________________
        |               list address                       | 1 0 |
        |__________________________________________________|_____|
        32                                                  2     0

         ________________________________________________________
        | + |           integer value                    | 0 1 1 |
        |___|____________________________________________|_______|
        32  31                                           3       0

         ________________________________________________________
        |               atom or functor number           | 1 1 1 |
        |________________________________________________|_______|
        32                                               3       0


       N.B. Key = Term<32:3>

         ________________________________________________________
        |               box "FRACTION"                   | 1 1 1 |
        |________________________________________________|_______|
        |                                                        |
        |_______________floating point number____________________|
        |                                                        |
        |________________________________________________________|
```

VI. Instruction Formats (provisional)

In the formats marked with a +, the opcode may be immediately followed by a one byte argument number in the case of *get* and *put* instructions. The formats marked with an asterisk are nonessential optimisations.

```
            Op-Code   Argument

byte:     0          1          2          3          4          5

*          ____________
          | var        |
          |____________|

           ______________________
     +    | var        | number  |
          |____________|_________|

*          ________________________________
     +    | const      |  short value      |     (sign extended)
          |____________|___________________|

           _____________________________________________________
     +    | const      |  long value                             |
          |____________|_________________________________________|

           ________________________________
     +    | struct     |  functor no.      |
          |____________|___________________|

           ________________________________
          | pred       | procedure addr.   |     (an offset within the segment)
          |____________|___________________|

           ________________________________
          | try        | clause address    |     (an offset within the segment)
          |____________|___________________|

           ______________________________________________
          | switch     |           |    table size      |
          |____________|___________|____________________|
          | key                    | clause address     |     (an offset within the segment)
          |________________________|____________________|
          |                        |                    |
```

References

1. D. L. Bowen, L. M. Byrd and W. F. Clocksin. A portable Prolog compiler. Logic Programming Workshop '83, Universidade Nova de Lisboa, June, 1983, pp. 74-83.

2. M. Bruynooghe. A note on garbage collection in Prolog interpreters. First International Logic Programming Conference, University of Marseille, September, 1982, pp. 52-55.

3. E. Tick. An overlapped Prolog processor. Artificial Intelligence Center, SRI International, Menlo Park, California 94025, 1983.

4. D. H. D. Warren. *Applied Logic -- its use and implementation as programming tool*. Ph.D. Th., University of Edinburgh, Scotland, 1977. Available as Technical Note 290, Artificial Intelligence Center, SRI International.

5. D. H. D. Warren. An improved Prolog implementation which optimises tail recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1980. Presented at the 1980 Logic Programming Workshop, Debrecen, Hungary.