PROLOG ENGINE

- a byte-coded non-structure-sharing abstract machine definition

**** DRAFT ****

David H D Warren
Artificial Intelligence Center
SRI International
20 April 1983

Abstract

N.B. This note is an interim document; the material it contains is not complete and some details may be inconsistent.

This note describes an abstract Prolog machine, called "Prolog Engine", suitable for software, firmware, or hardware implementation. The design calls for a large virtual memory, byte-addressable machine, and is particularly oriented towards 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 (op-code), followed by a number of arguments (usually zero or one). An argument may be 1, 2, or 4 bytes long.

The Engine implementation comprises a large number of small routines defining the different operations. Execution proceeds from one routine to the next by despatching on the op-code 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.

The Engine is currently implemented in a Prolog-based macro language called Progol, which is expanded into VAX machine code. Approximations to the VAX machine code expansions of the main routine are included in this document. The Progol implementation should be fairly easy to transport to a variety of machines to give an efficient software implementation of the Engine. The primary intention, however, is that the Engine should be implemented directly in microcode on a suitable machine.

1. Full Non-Structure-Sharing

The present design 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". This is in contrast to other "non-structure-sharing" implementations, such as those of Mellish and Bruynooghe, which still use structure-sharing for representing goals.

Some advantages of full non-structure-sharing are:

- Implementation simplicity. The implementation (ie. kernel code, microcode, or specialised hardware) should be smaller.

- Garbage collection is more straightforward (and Bruynooghe's 1982 optimisation follows by default).

- Tail recursion optimisation 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. (Nice!)

- 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.

- The code for the first goal in the body of a clause can be reduced to almost nothing provided variables are allocated to registers in the right way. For example the main 'concatenate' clause can be represented by just 8 instructions, although it comprises some 12 source symbols. Fewer instructions to decode means faster execution.

The main disadvantage of full non-structure-sharing (and of partial non-structure-sharing too) is time wasted in unnecessary copying, particularly when a clause is entered and then fails early in the body. This disadvantage doesn't seem too severe, since:

- Copying can be relatively fast, compared with other overheads.

- Clauses can be preprocessed by the compiler (or by the user!) to minimise unnecessary copying by extracting parts of the code into auxiliary clauses accessed through extra predicates. In this way, one arrives at something very close to traditional structure-sharing.

2. Steps Involved in One Resolution

The main execution step of Prolog is called a "resolution", and amounts to a single "logical inference". A list of goals is transformed into a new list of goals by matching a selected goal against the head of a clause selected from the program. Putting in a bit more implementation detail, one resolution consists of the following steps:

(0) Take the first goal in the list of outstanding goals as the current goal, and find the first clause which could potentially match. (Indexing of clauses generally ensures that only a few clauses have to be considered).

(1) If there are other clauses which could potentially match, create (or retain) a choice point which preserves the current execution state. Backtracking may later return us to this point.

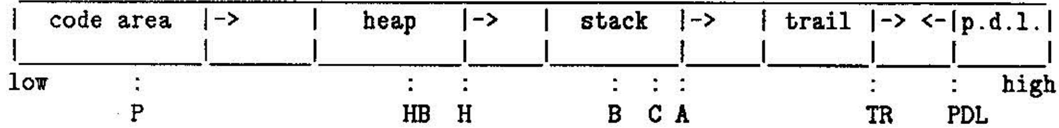
(2) Unify the head of the clause against the goal, remembering all variable bindings which will need to be undone if we subsequently backtrack.

(3) If there are no choice points later than the goal we have just matched, then discard that goal. N.B. We must make sure we don't leave "dangling references" into the discarded goal.

(4) Copy any goals in the body of the clause onto the front of the list of outstanding goals, and then proceed with step (0).

3. Data Areas

The main data areas are the code area, containing instructions (etc.) representing the program itself, and three areas operated as stacks, the (local) stack, the heap (or global stack), and the trail. These are laid out in memory as follows:



It turns out to be important that the stack and heap grow in the same direction, and that the stack grows away from the heap. (This simplifies the policing of certain restrictions on variable-variable bindings needed to prevent dangling references).

Each resolution leads to the creation of three stack frames, one on each stack. Backtracking will eventually discard these stack frames. The local frame contains information that is needed only as long as the corresponding procedure is active, namely bookkeeping information (required mainly for backtracking) plus a representation of the body of the matching clause. The heap frame contains a representation of new structures (complex terms) created by the procedure invocation. The trail frame contains addresses of variable cells which have been bound during unification and which must be unbound on backtracking.

The trail is accessed only by pushing or popping the top item. The stack is also mainly accessed by pushes and pops, but there are a few random accesses too. The heap is in general randomly accessed, although it grows (by resolution) and contracts (by backtracking) as a stack.

4. Registers

The current state of a Prolog computation is defined by certain registers containing pointers into the main data areas. These registers (with their VAX realisations) are as follows:

Register	Description	VAX
PDL	top of push-down list	(SP)
P	program pointer (to the code area)	(R4)
A	argument pointer (to local stack)	(R12)
B	backtrack pointer (to the local stack)	(R11)
C	continuation pointer (to the local stack)	(R0)
TR	top of trail	(R10)
H	top of heap	(R9)
HB	heap backtrack pointer	(R8)
S	structure pointer (to the heap)	(R7)
T	term/temporary register	(R6)
T1	term/temporary register	(R5)
X1, X2, X3	variables 1-3	(R1-R3)
X4, X5, ...	other variables	main memory locations

(Why oh why does the VAX tie up so many of its registers!)

5. Prolog Machine State (between resolutions)

```
              stack                       heap                        trail
       :                 :         :                 :          :                 :
       :                 :         :                 :          :                 :
       |_________________| HB->H'|_________________|      TR'|_________________|
       |              B' |         |                 |          |                 |
       |  choice      P' |         |                 |          |  bound          |
       |  point       A' |         |                 |          |  variable       |
       | (if any)    TR' |         |   structures    |          |  addresses      |
  B -> |_____________H'__|         |                 |    TR -> |_________________|
       |                 |         |                 |
       |                 |         |                 |
       |                 |    H -> |_________________|
       |  other goals    |
       |                 |
  C -> |_________________|
       |                 |
       |  goal 1 args    |
       |                 |
  A -> |_________________|
       .................
    C:  continuation   :
       :...............:
    P:  procedure      :
       :...............:
```

OR (after resolution with a unit clause)

```
              stack
       :                 :
       :                 :
  C -> |_________________|
       |                 |
       |  goal 1 args    |
       |                 |
  A -> |_________________|    .................
       |       =>        |    C:  continuation   :
       |.................|    :...............:
       |       =>        |    P:  procedure      :
       |.................|    :...............:
       :                 :

       :                 :
       |_________________|
       |                 |
       |   choice        |
       |   point         |
  B -> |_________________|
```

6. Run-Time Structure Formats

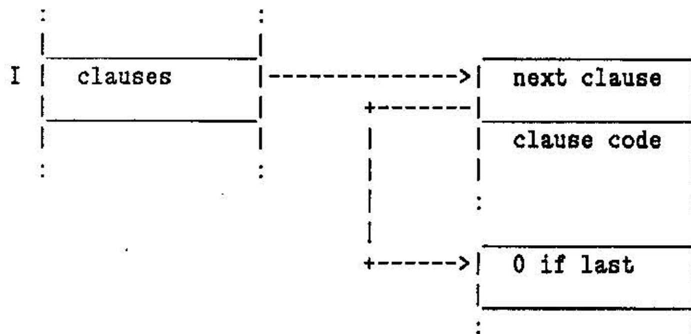
```
GOAL                                STRUCTURE (COMPLEX TERM)
 ______________                      ______________
| argument N   |                    | functor      |
|    :         |                    |______________|
|    :         |                    | argument 1   |
| argument 1   |                    |    :         |
|______________|                    |    :         |
| next goal    |                    | argument N   |
|______________|                    |______________|
| procedure    |
|______________|
```

```
CHOICE POINT
 ______________
| prev. choice |  (B')
|______________|
| next clause  |  (P')
|______________|
| arg. pointer |  (A')  {caller's continuation and arguments}
|______________|
| trail point  |  (TR')
|______________|
| heap point   |  (H')
|______________|
```

7. Procedure Formats

This is the interim version which (a) is rather extravagant of storage and (b) lacks indexing (cf. later sections for description of the proposed indexing scheme).

procedure



8. 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____________________|
        |                                                     |
        |_____________________________________________________|
```

9. Instruction Formats

```
         Op-Code  Argument

byte:   0        1        2        3        4        5

         ________
        |  var   |
        |________|

         ________ ______________________________________
        | const  |                 value                |
        |________|______________________________________|

         ________ __________________
        | struct |  functor no.     |
        |________|__________________|

         ________ __________________
        | pred   |  predicate no.   |
        |________|__________________|
```

10. Analysis of a Prolog Program into Basic Operations

Ignoring certain optimisations for the time being, typical clauses such as:

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

translate into the following operations:

```
pop_goal(P)            pop_goal(P)
succeed                proceed
push_goal(S)
push_goal(R)
execute_goal(Q)
```

where the the operations:

```
pop_goal(P(T1,T2,T3))  push_goal(P(T1,T2,T3))  execute_goal(P(T1,T2))
```

are defined as follows:

```
pop_arg(T1)            push_arg(T3)
pop_arg(T2)            push_arg(T2)            push_arg(T2)
pop_arg(T3)            push_arg(T1)            push_arg(T1)
                       push_pred P             execute P
```

The generic operations 'pop_arg(T)' and 'push_arg(T)' translate into different basic operations, depending on the kind of term T and its state of instantiation (if it is a variable). Operations on structures, such as:

```
pop_arg(F(T1,T2))      push_arg(F(T1,T2))      unify_arg(F(T1,T2))
```

are translated as follows:

```
pop_struct F           push_struct F           unify_struct F
unify_arg(T1)          unify_arg(T1)           unify_arg(T1)
unify_arg(T2)          unify_arg(T2)           unify_arg(T2)
resume_head            resume_body             resume
```

with the following optimisations if none of the arguments T1, T2, etc is a structure:

```
pop_easy_struct F      push_easy_struct F
unify_arg(T1)          unify_arg(T1)
unify_arg(T2)          unify_arg(T2)
                       continue_body
```

Operations on a variable, such as:

```
pop_arg(X)             push_arg(X)             unify_arg(X)
```

are translated as follows. The variable X is allocated to a register, number N. If it is the first occurrence of the variable in the clause code, the translations are:

```
pop_varN               push_varN               unify_varN
```

otherwise:

```
pop_valN               push_valN               unify_valN
```

If a variable occurs both in the head and in the body, and it does not occur in a "unify" context in the head, then the last occurrence in the head must be translated by the appropriate one of:

```
pop_perishable_varN
pop_perishable_valN
```

(This is to ensure that the perishable variables are properly preserved on the global stack).

If a variable has previously occurred in a "unify" context, then 'unify_valN' is optimised to:

```
unify_global_valN
```

As an example of clause encoding, here are the (not fully optimised) translations of the two 'concatenate' clauses:

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

pop_easy_list
unify_var4
unify_var1
pop_perishable_var2
pop_easy_list
unify_val4
unify_var3
succeed
push_val3
push_val2
push_val1
execute concatenate
```

```
concatenate([],L,L).

pop_nil
pop_var1
pop_val1
proceed
```

11. Optimisations

Cetain sequences of basic operations are common, and can be replaced by "short-cut" operations. For example:

```
invoke P = push_valN + ... + push_val1 + execute P

instate P = succeed + invoke P
```

12. Examples of Clause Encoding

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

X1              [-------------                [--------
X2                  [---------                   [-----
X3            [---------] [---                      [--

        pop_easy_list
        unify_var3
        unify_var1
        pop_perishable_var2
        pop_easy_list
        unify_global_val3
        unify_var3
        instate concatenate

concatenate([],A,A).

X1              [-]

        pop_nil
        pop_var1
        pop_val1
        proceed

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

X1     [-------         [----------------------------------------
X2   [---------           [-------------------*-------------------
X3        [----             [----------]  [-----------------------
X4                             [-------------------------]
X5           [-                                [-----------]  [--

        pop_easy_list
        unify_var2
        unify_var1
        pop_perishable_var3
        pop_perishable_var5
        succeed
        push_val5
        push_var5
        push_var4
        push_pred qsort
        push_easy_list
        unify_global_val2
        unify_val5
        continue_body
        push_val3
        push_var3
        push_pred qsort
        invoke split

qs([],R,R).

X1     [-]
```

```
        pop_nil
        pop_var1
        pop_val1
        proceed

split([X|L],Y,[X|L1],L2) :- X < Y, !, split(L,X,L1,L2).

X1        [---------------------------          [---------
X2      [-------*------------*--------            [-------
X3                [-------------------                [----
X4                    [---------------                  [--
X5           [-------------------]

        pop_easy_list
        unify_var2
        unify_var1
        pop_var5
        pop_easy_list
        unify_global_val2
        unify_var3
        pop_perishable_var4
        succeed
        push_val2
        push_val5
        do <
        cutsucceed
        invoke split

split([],Y,L,L).

X1          [-]

        pop_nil
        pop_void
        pop_var1
        pop_val1
        proceed
```

13. Classification of Operations

(Only the starred operations are strictly essential).

```
    HEAD                              BODY

  * proceed                         * execute P
  * succeed                         * push_pred P
    instate P                         invoke P

    pop_void                          push_void
    pop_varN                        * push_varN
  * pop_valN                        * push_valN
  * pop_perishable_varN
    pop_perishable_valN

  * pop_const C                     * push_const C
    pop_list                          push_list
    pop_struct F                      push_struct F
    pop_easy_list                     push_easy_list
  * pop_easy_struct F               * push_easy_struct F


    UNIFY                             COPY

    unify_void                        copy_void
  * unify_varN                      * copy_varN
  * unify_valN                      * copy_valN
    unify_global_valN                 copy_global_valN

  * unify_const C                   * copy_const C
    unify_list                        copy_list
    unify_struct F                    copy_struct F
    unify_end_list                    copy_end_list
    unify_end_struct F                copy_end_struct F


    resume_unify                      resume
    continue_head                     resume_head
                                      resume_copy
                                      resume_body
                                    * continue_body
```

NB. Corresponding unify and copy operations are represented by the same opcodes, since it is only at runtime that one can distinguish which operation is required ("read" mode versus "write" mode). To get "double mileage" out of the 256 available opcodes, body operation opcodes are allowed to overlap with the opcodes for other operations.

14. Description of Instructions and Basic Operations

14.1. Control Instructions

succeed This instruction precedes the body of a non-unit clause. It sets the argument pointer A to point to the last thing still needed on the stack, which will be either the continuation C or the backtrack point B. This has the effect of dicarding the goal that has just been matched if there are no backtrack points after it. Subsequent instructions will use A as the top of stack pointer for the purpose of pushing the body goals.

push_pred N This instruction terminates a body goal, and is responsible for pushing the continuation C and a pointer to the procedure for predicate number N onto the stack. The continuation C is updated to point to the goal just formed.

execute N This instruction terminates the final goal in the body of a clause. The program pointer P is set to point to the procedure for predicate number N, and the alternatives pointer for the first clause is fetched. If the alternatives are nonempty, a backtrack point is created on the stack.

proceed This instruction terminates a unit clause. The argument pointer A is set to the continuation C, and top goal procedure pointer is popped from A, becoming the new program pointer P. The alternatives pointer for the first clause is fetched, and if nonempty a backtrack point is created. Finally, the new continuation is popped from A.

14.2. Push Instructions

push_var_N This instruction represents a goal argument which is an unbound variable. The instruction pushes a new unbound variable onto the stack, and stores a reference to it in register N.

push_val_N This instruction represents a goal argument which is a bound variable. The instruction simply pushes the value in register N onto the stack.

push_const X This instruction represents a goal argument which is a constant. The instruction simply pushes the constant X onto the stack.

push_easy_struct N

This instruction marks the beginning of a structure without substructures occurring as a goal argument. The instruction pushes the functor N for the structure onto the heap, and pushes a corresponding structure pointer onto the stack.

continue_body This instruction marks the end of a structure without substructures occurring as a goal argument. The only effect is to signal that the following instructions are to be executed as body instructions. It could be dispensed with if body instruction opcodes were

distinct from head instruction opcodes.

14.3. Pop Instructions

pop_var_N This instruction represents a head argument which is an unbound variable not needing special protection. The instruction simply pops a value off the stack into register N.

pop_perishable_var_N
This instruction represents a head argument which is an unbound variable needing special protection. The instruction pops a value off the stack into register N, and dereferences it. If the result is a reference to an unbound variable in the current stack frame, the variable is "globalised" onto the heap. The final result is left in register N.

pop_val_N This instruction represents a head argument which is a bound variable not needing special protection. The instruction pops a value off the stack and unifies it with the contents of register N. The final result is left in register N.

pop_perishable_val_N
Not implemented yet, search me!

pop_const X This instruction represents a head argument which is a constant. The instruction pops a value off the stack 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.

pop_easy_struct N
This instruction marks the beginning of a structure without substructures occurring as a head argument. The instruction pops a value off the stack 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, functor N 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 N, the pointer S is set to point to the arguments of the structure and execution proceeds in "read" mode. Otherwise, backtracking occurs.

14.4. Unify Instructions

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

unify_global_val_N
This instruction represents a head structure argument which 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 register N, leaving the result in register N. If the instruction is executed in "write" mode, it pushes the value of register N onto the heap.

unify_val_N This instruction represents a head structure argument which is a variable bound to a value that is not necessarily global. The effect is the same as 'unify_global_val', except that in "write" mode it dereferences the value of register N 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 register N is set to point to the new variable.

unify_const X This instruction represents a head structure argument which 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 X, 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. If the instruction is executed in "write" mode, the constant X is pushed onto the heap.

14.5. Other Basic Operations

create_choice_point

This operation is performed when entering a Prolog procedure for which there is more than one potentially matching clause. The following values are pushed onto the stack: a pointer to the previous choice point, a pointer to the alternative clauses, a pointer to the caller's continuation and arguments, the current trail pointer, and the current heap pointer. HB is set to the current heap pointer, and B is set to point to the current top of stack.

fail This operation is performed when a failure occurs during unification. It causes backtracking to the most recent choice point. 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. If there are other alternatives, a pointer to them is recorded in the choice point, and the choice point is retained; otherwise the choice point is discarded by restoring B to the previous value saved in the choice point. Finally, 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.

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.

14.6. Indexing Instructions

try_generic [C']

This is the first instruction in a **generic** clause, i.e. a clause whose first argument is a variable. The next clause field in the current choice point is set to C', unless C' is zero, in which case the current choice point is discarded.

try_big_index [C', C", Mask, Table]

This instruction precedes a group of **specific** clauses, i.e. clauses whose first arguments are not variables. The first argument on the stack is dereferenced. If the result is a variable, execution proceeds with the next clause, C'. Otherwise the next clause field in the current choice point is set to C" (the next generic clause), unless C" is zero, in which case the current choice point is discarded. A key is determined from the dereferenced first argument: if it is a constant, the key is simply that constant; if it is a structure, the key is the principal functor of that structure, and the argument pointer S is set to point to its arguments. The value of the low-order bits of the key, as determined by the mask Mask, is used as an index into the table Table, yielding a chain of specific clauses. The key is compared with the key field of each of these clauses until one is found which has the identical key, in which case execution proceeds with that clause, skipping the first instruction. If the identical key is not found, execution proceeds to clause C", or, if C" is zero, backtracking occurs.

try_const [C', C", K]

This is the first instruction of a clause whose first argument is a constant K, where that clause is the only one in its group having the key K. The next clause field of the current choice point is set to C', unless C' is zero, in which case the current choice point is discarded. The first argument on the stack has already been dereferenced to a variable; that variable is now bound to the constant K and the binding is trailed if necessary.

try_struct [C', C", K]

This is the first instruction of a clause whose first argument is a structure with principal functor K, where that clause is the only one in its group having the key K. The next clause field of the current choice point is set to C', unless C' is zero, in which case the current choice point is discarded. The first argument on the stack has already been dereferenced to a variable; that variable is now bound to a new structure pointer pointing at the top of the heap (with the binding being trailed if necessary), functor K is pushed on the heap, and execution proceeds in "write" mode.

share_key [C', C", K]

This is the second instruction of a dummy clause, which precedes the first of several

clauses in the same group having the same key; (the first instruction of the dummy clause is never executed). A **create_choice_point** operation is performed, and execution then proceeds with the following clause C'.

retry_const [C', C'', K]

This is the first instruction of a clause whose first argument is a constant K, where that clause is NOT the only one in its group having the key K. If the first argument on the stack has been dereferenced to a variable, the effect is the same as **try_const**; otherwise the next clause field in the current choice point is set to C'' (the next clause for key K), unless C'' is zero, in which case the current choice point is discarded.

retry_struct [C', C'', K]

This is the first instruction of a clause whose first argument is a structure with principal functor K, where that clause is NOT the only one in its group having the key K. If the first argument on the stack has been dereferenced to a variable, the effect is the same as **try_struct**; otherwise the next clause field in the current choice point is set to C'' (the next clause for key K), unless C'' is zero, in which case the current choice point is discarded.

15. Clause Indexing

Clause indexing reduces the set of candidate clauses that must be considered when executing a goal. It is important for minimising the amount of computation needed to find matching clauses, and for helping the system to recognise determinate choices in the absence of cuts. The version described here follows DEC-10 Prolog in providing for indexing only on the predicate and a single **key** which is the principal functor of the first argument. It should be fairly easy to extend things to permit the key to be any user defined function of the goal, which the user might optionally specify for each predicate.

The main idea is that the clauses for a predicate will be linked on two distinct (but partly coinciding) chains providing for access with a key, and access without a key (for use in the case that the key is not fully specified in the goal). The unkeyed access chain will be essentially just a list of the clauses in the order they were entered. The keyed access chain will link clauses with other **nodes** in a branching structure, so that only certain branches have to be considered for a given key. It will consist essentially of a list of **generic** clauses (first head argument a variable), interspersed with **index** nodes, which give access to groups of **specific** clauses (first head argument a nonvariable).

[As a later optimisation, we will probably make a special case of a **plain** procedure, one having no generic clauses and no more than say 8 distinct keys].

Each node commences with an **entry** instruction, which indicates the type of node. Preceding the entry instruction, there is usually a next node pointer, which indicates the next node for unkeyed access. The different nodes, with their names, formats, and (to be added shortly) roles, are as follows.

generic clause

```
                                          |
                                __________|_______________
                               | next node | try  | clause
                               |__________|_instr_|_code____
```

specific clause with a unique key

```
                                          |
 ________________________________________|_______________
| key         | next key    | next node  | try  | clause
|_____________|_node________|____________|_instr_|_code____
```

shared key node

```
                                          |
 ________________________________________|______________
| key         | next key    | first clause |       | share |
|_____________|_node________|_for_this_key_|_______|_key___|
```

specific clause with a shared key

```
                                          |
 ________________________________________|_______________
| key         | next clause  | next node  | retry | clause
|_____________|_for_this_key_|____________|_instr_|_code____
```

big-index node

(small-index node will be a special case)

```
                            |
 __________________________|____________________________________
| next generic | next node  | index | keys  | mask  | despatch table ...
|_clause_______|____________|_instr_|_count_|_______|________________
```

procedure header node

first node	last generic clause	4-way despatch table ?

<<What follows has been partially superseded; see the description of the indexing instructions in the preceding section for the latest word>>. Entry instructions can be executed in one of three different modes: **enter** mode, **traverse** mode, and **reenter** mode. The different entry instructions, with their possible modes of execution, are as follows.

	enter	traverse	reenter
generic	*		*
index	*		*
constant		*	*
structure		*	*
list		*	*

The effect of the different actions is roughly as follows:

```
enter(generic(C')) = create_choice(C').

enter(index(C',C'',C)) =
   ( keyed_access -> create_choice(C''), seek(C)
   | otherwise -> create_choice, traverse(C') ).

seek(specific(C',C'',K)) =
   ( key = K -> continue
   | otherwise -> seek(C'') ).

do(share_key(C',C'')) = create_choice(C''), continue(C').

traverse(specific(C',C'',K)) = retain_choice(C'), emit(K), continue.

reenter(specific(C',C'',K)) =
   ( keyed_access -> retain_choice(C''), skip_over(K)
   | otherwise -> retain_choice(C'), continue(K) ).

reenter(generic(C')) = retain_choice(C').

reenter(index(C',C'',C)) =
   ( keyed_access -> retain_choice(C''), seek(C)
   | otherwise -> traverse(C') ).

retain_choice(C) =
   ( null(C) -> remove_choice
   | otherwise -> BC := C ).

create_choice(C) =
   ( null(C) -> continue
   | otherwise -> create_choice, BC := C ).
```

15.1. Example of Clause Indexing

To illustrate the way indexing information is encoded, here is an example of a Prolog procedure followed by the corresponding code.

```
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) :- repeat.
call(true)
```

```
call:   I1, C7, ...(?).
I1:     C6, C1, try_big_index, 4, 2'11000, D1, C3, 0, C4.
D1:     'or'/2, 0, C1, no_op, share_key.
C1:     'or'/2, C2, C2, retry_struct, ...
C2:     'or'/2, 0, C3, retry_struct, ...
C3:     'trace', 0, C4, try_const, ...
C4:     'notrace', C5, C5, try_const, ...
C5:     'nl', 0, C6, try_const, ...
C6:     C7, try_generic, ...
C7:     I2, try_generic, ...
I2:     0, C8, try_big_index, 3, 2'1000, C8, D9.
C8:     'call'/1, 0, C9, try_struct, ...
D9:     'repeat', C11, C9, no_op, share_key.
C9:     'repeat', C10, C10, retry_const, ...
C10:    'repeat', 0, C11, retry_const, ...
C11:    'true', 0, 0, try_const, ...
```

16. Machine Definition (to a first approximation)

```
Routine
  VAX implementation       Definition               Description

continue_head:             ; {Decode the next instruction in "head" mode.}
  MOVZBL (P)+,I            ; I <- next_byte(P)
  MOVL head_op[I],Q        ; Q <- head_op(I)
  JMP (Q)                  ; do Q

continue_copy:             ; {Decode the next instruction in "write" mode.}
  MOVZBL (P)+,I            ; I <- next_byte(P)
  MOVL copy_op[I],Q        ; Q <- copy_op(I)
  JMP (Q)                  ; do Q

continue_body:             ; {Decode the next instruction in "body" mode.}
  MOVZBL (P)+,I            ; I <- next_byte(P)
  MOVL body_op[I],Q        ; Q <- body_op(I)
  JMP (Q)                  ; do Q

head_op:                   ; {Table of "head" routines.}
head_op(0):       ...
                  ...
                  ...

copy_op:                   ; {Table of "write" routines.}
copy_op(0):       ...
                  ...
                  ...

body_op:                   ; {Table of "body" routines.}
body_op(0):       ...
                  ...
                  ...

succeed:                   ; {Tidy the local stack: A <- maximumof(B,C).}
  CMPL B,C                 ; B is_before C else succeed_provisionally
  BGEQ succeed_provisionally
  MOVL C,A                 ; A <- C
  BR continue_head         ; continue_head

succeed_provisionally:
  MOVL B,A                 ; A <- B
  BR continue_head         ; continue_head

proceed:                   ; {Proceed to the next goal.}
  MOVL C,A                 ; A <- C
  MOVL -(A),P              ; P <- pop_item(A)
  MOVL -(A),C              ; C <- pop_item(A)
  BR choose                ; choose

execute:                   ; {Execute the current goal.}
  MOVZWL (P),I             ; I <- short(P)
  MOVL predicate[I],P      ; P <- predicate(I)
  BR choose                ; choose
```

```
push_pred:                  ; {Push a predicate.}
  MOVL C,(A)+               ; C -> push_item(A)
  MOVZWL (P)+,I             ; I <- get_short(P)
  MOVL predicate[I],(A)+ ; predicate(I) -> push_item(A)
  MOVL A,C                  ; C <- A
  BR continue_head          ; continue_head

instate:                    ; {Tidy stack, save goal, and execute it.}
  CMPL B,C                  ; B is_before C else ...
  BGEQ ...
  MOVL C,A                  ; A <- C
invoke:                     ; {Save goal and execute it.}
  MOVZWL (P)+,I             ; I <- get_short(P)
  MOVL procedure[I],P       ; P <- procedure(I)
  MOVZBL arity[I],I         ; I <- arity(I)
  MOVL @call_op[I],Q       ; Q <- call_op(I)
  JMP (Q)                   ; do Q

  :
  :
call2:                      ; {Save argument 2.}
  MOVL X2,(A)+              ; X2 -> push_item(A)
call1:                      ; {Save argument 1.}
  MOVL X1,(A)+              ; X1 -> push_item(A)
choose:                     ; {Choose clause.}
  MOVZWL (P)+,P1            ; P1 <- get_short(P)
  BEQL continue_head        ; P1 \= 0 else continue_head
  MOVL B,B1                 ; B1 <- B
  CMPL B,A                  ; B is_before A else ...
  BGEQ ...
  MOVL A,B                  ; B < - A
  MOVL B1,(B)+              ; B1 -> push_item(B)
  MOVL P1,(B)+              ; P1 -> push_item(B)
  MOVL A,(B)+               ; A -> push_item(B)
  MOVL TR,(B)+              ; TR -> push_item(B)
  MOVL H,(B)+               ; H -> push_item(B)
  BR continue_head          ; continue_head

fail:                       ; {Restore the state of latest choice point.}
  MOVL -(B),H               ; H <- pop_item(B)
  MOVL -(B),TR0             ; TR0 <- pop_item(B)
  MOVL -(B),A               ; A <- pop_item(B)
  MOVL -(B),P               ; P <- pop_item(B)
  MOVL -(A),C               ; C <- pop_item(A)
undo:
  CMPL TR0,TR               ; TR0 is_before TR else undone
  BGEQ undone
  MOVL -(TR),T              ; T <- pop_item(TR)
  MOVL T,(T)                ; T -> deref(T)
  BR undo                   ; undo

undone:
  MOVL (P)+,P1              ; P1 <- get_long(P)
  BNEQ retain_choice        ; P1 = 0 else retain_choice
  MOVL -(B),B               ; B <- pop_item(B)
```

```
  MOVL -1(B),HB          ; HB <- top_item(B)
  BR continue_head       ; continue_head

retain_choice:
  MOVL P1,(B)+           ; P1 -> push_item(B)
  ADDL 3,B               ; B <- B+3
  BR continue_head       ; continue_head

unify_var3:              ; {Unify with variable 3, which is unbound.}
  MOVL (S)+,X3           ; X3 <- next_term(S)
  BR continue_head       ; continue_head

unify_const:             ; {Unify with a constant.}
  MOVL (S)+,T            ; T <- next_term(S)
  BITB 2'11,T            ; non-ref(T)
     BEQL ...            ;    else ...
  CMPL T,(P)+            ; T = next_term(P)
     BNEQ fail           ;    else fail
  BR continue_head           ; continue_head

pop_var3:                ; {Match with variable 3, which is unbound.}
  MOVL -(A),X3           ; X3 <- pop_term(A)
  BITB 2'11,X3           ; non-ref(X3)
     BEQL ...            ;    else ...
  BR continue_head       ; continue_head

pop_easy_struct:         ; {Match with structure.}
  MOVL -(A),S            ; S <- pop-term(A)
  BITB 2'11,S            ; non-ref(S)
     BEQL ...            ;    else ...
  BITB 2'10,S            ; is_struct(S)
     BNEQ fail           ;    else fail
  BICB 2'11,S            ; S <- untag_struct(S)
  MOVZWL (P)+,I          ; I <- get_short(P)
  CMPL (S)+,(P)+         ; get_fn(S) = functor(I)
     BNEQ fail           ;    else fail
  BR continue_head       ; continue_head

pop_easy_list:           ; {Match with a list.}
  MOVL -(A),S            ; S <- pop-term(A)
  BITB 2'11,S            ; non-ref(S) else pop_list_ref
  BEQL pop_list_ref
  BITB 2'01,S            ; is_list(S) else fail
  BNEQ fail
  BICB 2'11,S            ; S <- untag_struct(S)
  BR continue_head       ; continue_head

pop_list_ref:
  CMPL (S),S             ; S = deref(S) else ...
  BNEQ ...
  MOVAL ^B10(H),(S)      ; tag_list(H) -> deref(S)
  CMPL S,H               ; is_global(S)
     BLSS ...            ;    else ...
  CMPL S,HB              ; S is_newer_than HB
     BGEQ ...            ;    else ...
```

```
  BR continue_copy       ; continue_copy

copy_var3:               ; {Copy variable 3, which is unbound.}
  MOVAL (S)+,X3          ; X3 <- ref_to_next_term(S)
  MOVL X3,(X3)           ; X3 -> deref(X3)
  BR continue_copy       ; continue_copy

copy_val3:               ; {Copy variable 3, which is bound.}
  MOVL X3,(S)+           ; X3 -> next_term(S)
  BR continue_copy       ; continue_copy

copy_const:              ; {Copy constant.}
  MOVL (P)+,(S)+         ; next_term(P) -> next_term(S)
  BR continue_copy       ; continue_copy

push_var3:               ; {Push variable 3, which is unbound.}
  MOVL A,X3              ; X3 <- ref_to(A)
  MOVL X3,(A)+           ; X3 -> push_item(A)
```

17. Performance Benchmark

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

pop_easy_list   MOVL -(A),S
                BITB ^B11,S
                BEQL ...
                BITB ^B01,S
                BNEQ ...
                BICB ^B11,S
  (6)           continue_head              =        MOVZBL (P)+,I
                                                    MOVL head_op[I],Q
unify_var4      MOVL (S)+,X4                        JMP (Q)
  (1)           continue_head

unify_var1      MOVL (S)+,X1
  (1)           continue_head

pop_perishable_var2     MOVL -(A),X2
                BITB ^B11,X2
                BEQL ...
  (3)           continue_head

pop_easy_list   MOVL -(A),S
                BITB ^B11,S
                BEQL ---
                CMPL (S),S
                BNEQ ...
                MOVAL ^B10(H),(S)
                CMPL S,H
                BGTR ...
                CMPL HB,S
                BGEQ ...
 (10)           continue_copy              =        MOVZBL (P)+,I
                                                    MOVL copy_op[I],Q
copy_global_val4        MOVL X4,(H)+                     JMP (Q)
  (1)           continue_copy

copy_var3       MOVL H,X3
                MOVL X3,(H)+
  (2)           continue_copy

succeed         CMPL B,C
                BGEQ ...
                MOVL C,A
  (3)           continue_body              =        MOVZBL (P)+,I
                                                    MOVL body_op[I],Q
push_val3       MOVL X3,(A)+                        JMP (Q)
  (1)           continue_body

push_val2       MOVL X2,(A)+
  (1)           continue_body

push_val1       MOVL X1,(A)+
  (1)           continue_body
```

```
execute         MOVZWL (P),I
                MOVL predicate[I],P
  (2)           continue_head

Total VAX instructions =   32 (actual execution)
                         + 36 (decoding 12 opcodes)
                         ----
                           68   [cf. 50 instructions on DEC-10 Prolog]
```

18. Trace of Concatenate (as currently implemented)

```
pop_easy_list
        MOVL    -(AP),R6
        BITL    #03,R6
        BEQL    001B385A
        BLBS    R6,001B3857
        SUBL3   #02,R6,R7
 (5)    ---

unify_var3
        MOVL    (R7)+,R3
 (1)    ---

unify_var1
        MOVL    (R7)+,R1
 (1)    ---

pop_perishable_var2
        MOVL    -(AP),R2
        BITL    #03,R2
        BEQL    001B35EA
 (3)    ---

pop_easy_list
        MOVL    -(AP),R6
        BITL    #03,R6
        BEQL    001B385A
        CMPL    (R6),R6
        BNEQ    001B387E
        ADDL3   #02,R9,(R6)
        CMPL R6,R9
        BLEQU   001B3873
        CMPL    R6,R8
        BGEQU   001B387B
(10)    ---

copy_global_val3
        MOVL    R3,(R9)+
 (1)    ---

copy_var3
        MOVL    R9,R3
        MOVL    R3,(R9)+
 (2)    ---
```

```
succeed
        CMPL     R0,R11
        BLEQU    ENGINE+0A3
        MOVL     R11,AP
 (3)    ---

push_val3
        MOVL     R3,(AP)+
 (1)    ---

push_val2
        MOVL     R2,(AP)+
 (1)    ---

push_val1
        MOVL     R1,(AP)+
 (1)    ---

execute
        MOVL     R11,L^RMSGBL\. BLANK .+04
        MOVZWL   (R4)+,R6
        MOVL     L^001B30DC[R6],R4
        MOVL     (R4)+,R5
        BEQL     ENGINE+0F3
        MOVL     R0,(AP)+
        MOVL     AP,R11
        MOVL     L^RMSGBL\. BLANK .+04,(R11)+
        MOVL     R5,(R11)+
        MOVL     AP,(R11)+
        MOVL     R10,(R11)+
        MOVL     R9,(R11)+
        MOVL     R9,R8
        SUBL2    #04,AP
(14)    ---

pop_nil
        MOVL     -(AP),R6
        BITL     #03,R6
        BEQL     001B376B
        CMPL     R6,#0F
        BNEQ     001B3768
        BRW      001B3553
 (6)
```

```
fail
        MOVL    -(R11),R9
        MOVL    -(R11),R7
        MOVL    -(R11),AP
        MOVL    -(R11),R4
        MOVL    -(AP),R0
        MOVL    (R4)+,R5
        BNEQ    001B3571
        MOVL    -(R11),R11
        MOVL    B^0FC(R11),R8
        BRW     001B3577
        CMPL    R10,R7
        BLEQU   001B3585
(12)    ---
```

```
Total =   61 (actual execution
        + 52 (decoding 13 opcodes)
        ----
         113
```

This current version has been clocked at 7,900 lips on the VAX-780. Some straightforward optimisations (primitive clause indexing, doing the instruction despatches in-line, coalescing the pushes with the execute instruction) will reduce the number of instructions executed from 113 to about 60. This should improve the speed by a factor of about 113/60, achieving of the order of 15,000 lips.

19. Engine Assembly Code Format

Each line of an engine code file is either empty, or has one of the following three formats:

```
<size><type><space><number>
<space><byte-codes>
F <number> <value>
<other><comment>
```

The first format consists of the character B, H, or W, indicating an item of size "Byte", "Halfword", or "Word", followed by the character space, C, P, F, or A, indicating an item of type "numeric" "Clause", "Predicate", "Functor", or "Atom", followed finally by a number (expressed in decimal) preceded by a space.

The second format consists of a space character followed by a sequence of simple byte codes, expressed in decimal, and separated by spaces.

The third format consists of the letter F followed by a space followed by a functor number followed by a space followed by the functor value.

The fourth format, a line starting with any character other than F, space, B, H, or W, is simply a comment (to be ignored by the loader).

Thus possible forms for a line include:

```
F <functor-number> <functor-value>
WC <predicate-number>
HP <predicate-number>
HF <functor-number>
WA <atom-value>
W  <code>
 <code> <code> <code>
%<comment>
```

For example, here are some clauses, and the corresponding code:

```
flatten(void,S,S).
flatten(pair(L,R),S0,S) :- flatten(L,S0,S1), flatten(R,S1,S).

WC 3 flatten
 16
WA void
 4 12 0

WC 3 flatten
 18
HF 2 pair
 22 25 9 10 2 8 4 9 1
HP 3 flatten
 8 7 6 0
HP 3 flatten
```

Table of Contents

1. Full Non-Structure-Sharing 0
2. Steps Involved in One Resolution 1
3. Data Areas 2
4. Registers 2
5. Prolog Machine State (between resolutions) 3
6. Run-Time Structure Formats 4
7. Procedure Formats 4
8. Data Formats (provisional) 5
9. Instruction Formats 6
10. Analysis of a Prolog Program into Basic Operations 7
11. Optimisations 8
12. Examples of Clause Encoding 9
13. Classification of Operations 11
14. Description of Instructions and Basic Operations 12
    - 14.1. Control Instructions 12
    - 14.2. Push Instructions 12
    - 14.3. Pop Instructions 13
    - 14.4. Unify Instructions 13
    - 14.5. Other Basic Operations 14
    - 14.6. Indexing Instructions 15
15. Clause Indexing 17
    - 15.1. Example of Clause Indexing 19
16. Machine Definition (to a first approximation) 20
17. Performance Benchmark 24
18. Trace of Concatenate (as currently implemented) 25
19. Engine Assembly Code Format 28