EFFICIENTLY IMPLEMENTING PURE PROLOG
or:
Not "YAWAM"

by

Peter Lüdemann

Technical Report 86-25

November 1986

Efficiently Implementing Pure Prolog

or:

Not "YAWAM"†

Peter Lüdemann‡

Department of Computer Science

University of British Columbia

Vancouver, B.C., Canada V6T 1W5

Abstract

High performance hardware and software implementations of Prolog are now being developed by many people, using the Warren Abstract Machine (or "WAM"). We have designed a somewhat different machine which supports a more powerful language than Prolog, featuring:

- efficiency similar to the WAM for sequential programs,
- tail recursion optimisation (TRO),
- sound negation,
- pseudo-parallelism (co-routining) with full backtracking,
- dynamic optimisation of clause order,
- efficient *if-then-else* ("shallow" backtracking),
- simple, regular instruction set designed for easily optimised compilation,
- efficient memory utilisation,
- integrated object-oriented virtual memory,
- predicates as first-class objects.

Our design gives the programmer more flexibility in designing programs than is provided by standard Prolog, yet it retains the efficiency of more limited designs.

---

† Yet Another Warren Abstract Machine.

‡ Present address: IBM Canada Ltd., 1 Park Centre, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3

Efficiently Implementing Pure Prolog

or:

Not "YAWAM"†

Peter Lüdemann‡

Department of Computer Science

University of British Columbia

Vancouver, B.C., Canada V6T 1W5

Abstract

High performance hardware and software implementations of Prolog are now being developed by many people, using the Warren Abstract Machine (or "WAM") [Warr83]. We have designed a somewhat different machine which supports a more powerful language than Prolog, featuring:

- efficiency similar to the WAM for sequential programs,
- tail recursion optimisation (TRO) [Warr86],
- sound negation,
- pseudo-parallelism (co-routining) with full backtracking,
- dynamic optimisation of clause order,
- efficient *if-then-else* ("shallow" backtracking),
- simple, regular instruction set designed for easily optimised compilation,
- efficient memory utilisation,
- integrated object-oriented virtual memory,
- predicates as first-class objects.

Our design gives the programmer more flexibility in designing programs than is provided by standard Prolog, yet it retains the efficiency of more limited designs.

Introduction

Warren's design for implementing logic programming [Warr77], [Warr83], [TiWa84] and [GLLO85] proves that logic languages like Prolog can be executed as efficiently as other symbol-oriented languages such as LISP [Tick86]. A number of people have implemented WAM in hardware ([DoPaDe84], [Mill86] and others). While we applaud these achievements (perhaps Fortran will be replaced by the 21st century!), we believe that a more flexible and powerful logic programming language than standard Prolog should be used. We will present an alternate design which allows implementing a more powerful language with similar efficiency.

Logic programming opens the exciting possibility of writing programs declaratively, considering only the logic of the programs and not the execution method. Prolog is a good step in this direction but programmers often feel they must use its "impure" non-logical constructs such as *cut* and *var*. By adding flexibility to Prolog's strict left-to-right depth-first computation rule, our design allows programs to be written in a more natural manner, using only "pure" logical constructs. These programs are often more efficient than equivalent "impure" programs and they are much easier to understand.

"Pure" Prolog programs are indifferent to the order in which clauses are tried [Lloy84]. If the implementation chooses to execute clauses in some parallel fashion, it should be free to do so. Programs written for such indeterminate implementations are often clearer than those which depend on execution order. A sequential solution of *an exercise attributed to R.W. Hamming* is given in [Dijk76]; a more elegant solution using co-routines (degenerate parallelism) is given in Chapter 8 of [Hend80]. [Kowa79] pp. 114-118 has other examples. [DDH72] has more general comments on why co-routines are desirable.

By freeing the implementation from left-to-right top-down order, we not only allow clearer – and often more efficient – programs but also provide sound implementation of negation, *if-then-else* and *setof* as described in [Nais85b]. The necessary control information can often be generated automatically as in [Nais85a].

Negation

"Standard" Prolog implements of negation unsoundly. Negation as failure [Clar78] requires that the terms be all ground. This is not enforced by the usual implementation:

```
not(Test) :- call(Test), !, fail.
not(Test).
```

Consider the list membership predicate:

```
member(X, X._).
member(X, _.Rst) :- member(X, Rst).
```

The query

```
?- X=4, not member(X, [1,2,3])
```

correctly succeeds but the logically equivalent

```
not member(X, [1,2,3]), X=4
```

† Yet Another Warren Abstract Machine.

‡ Present address: IBM Canada Ltd., 1 Park Centre, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3

fails. We can remedy this by having `not` delay until its arguments are sufficiently instantiated. That is, `not` provisionally succeeds and is retried later when the variable which caused the delay becomes instantiated.

A similar difficulty exists with `member1` which succeeds when the first element in a list is found:

```
member1(X, X._) :- !.
member1(X, Y.Rst) :- member1(X, Rst).
```

The query

```
X=2, member1(X, [1,2,3])
```

correctly succeeds but the logically equivalent

```
member1(X, [1,2,3]), X=2
```

incorrectly fails because the cut ("!") removes too many backtrack points. The solution herehere is to define `member1` using an *if-then-else* which delays until the test is sufficiently instantiated:

```
member1(X, Y.Rst) :-
    if X=Y then true
           else member1(X, Rst) endif
```

The call to `member1` provisionally succeeds with the test `X=1` delayed until `X` becomes instantiated: `X=2`. The resumed test fails and execution backtracks to the *else*. This tail recursively calls `member1` which again provisionally succeeds with the test `X=2` delayed until `X` becomes instantiated. The next goal, `X=2`, instantiates `X`, the backtrack point for the *else* is removed and the whole query succeeds.

Our original motivation for implementing delays was to provide sound implementation of negation and *if-then-else*. Delays also allow the extensions to Prolog, including first-order quantifiers, described in [LITo84] and they allow co-routining with backtracking. The result is a very flexible design which permits a variety of programming styles. To take advantage of our design, an extended Prolog must be used – but its description is beyond the scope of this paper.

In our implementation, deterministic predicates are not slowed by the more sophisticated delaying and backtracking features. Depending on the problem, the machine is well suited to purely deterministic predicates and to complex generate-and-test predicates using co-routines and backtracking.[3]

3 As a test of the usefulness, the program for solving a logic puzzle were changed so that the test predicates were initially delayed and woken up as the solution became instantiated. This removed so many permutations that execution time dropped from 65 minutes to 40 seconds.

The basic sequential inference engine

We describe first the basic sequential engine and then the features which allow backtracking and delaying.

The machine has 32 registers,[4] a single execution stack for saving status and registers across calls, a reset stack and a backtrack stack. There is no separate push down stack for unification and deallocation because we use the Deutsch-Schorr-Waite algorithm [Knut73] which traverses lists by reversing pointers using special tags which are not normally visible. All objects are first class citizens and are tagged:

- uninstantiated (or not ground) logical variable,
- reference (pointer) to another object which is automatically dereferenced whenever it is accessed,
- integer or floating point number,
- string,
- nil (denoted `[]`),
- list element (pointers to two objects, denoted `Hd.Tl` or `[Hd|Tl]`),
- paged out object with pointer into backing store[5] which is automatically paged in when accessed,
- code segment,
- "thunk" [Inge61] (code pointer with environment),
- cut point.

We use structure copying – implementation is simpler than structure sharing and efficiency is about the same [Mell82].

Each object is identified solely by its address which remains constant throughout the object's lifetime. Each object fits in a fixed size cell which is 10 to 16 bytes (depending on the base machine) containing:

- tag,
- flags,
- up to two pointers (or a double precision floating point number),
- reference count.

We use reference counting because its overhead seems to be about the same as for a marking garbage collect and it allows reclaiming memory sooner (Prolog programs seldom produce circular lists). We could easily use a marking garbage collector instead.

4 [AuHo82] notes that with 16 registers, about half the programs need register spill code; with 32, less than 5% need spill code.

5 Data in backing store never point to data in primary store. There is no space for a full description; [Gold83] has more details on a similar virtual memory system based on objects.

Numbers, nils and list elements fit entirely within these cells. Strings, code segment cells and thunks contain pointers into a separate area which is divided into segments and is compacted in a manner similar to Smalltalk-80's LOOM (large object oriented memory) [Gold83].

Although new strings can be created by concatenate or substring operations, most strings are constants which are known when the code is loaded. The loader ensures that only one copy is kept of each such constant string – two constant strings are equal if and only if they are at the same address. Dynamically created strings require full character by character comparison.

For simplicity, we treat functors as lists (as in micro-Prolog [ClMcC84]) so that, for example, `f(a,b)` ≈ `[f,a,b]`. Functors and lists are distinguished by a flag in the head element. (We could instead store the functor as a single element in the string area. Analysis of the trade-offs are beyond the scope of this paper.)

New cells are allocated from the free list. Because all the cells are the same size, this is as efficient as allocating and deallocating on a stack. We do not distinguish between "local" and "global" variables as in WAM. In WAM code about half to three-quarters of all memory references are to the global stack (extrapolated from [Tick86] and [Mats85]). Because we keep local variables entirely in registers, we can get similar performance without the complication of shadow registers (a hardware implementation would probably also cache the cells pointed by the registers). WAM's global stack is not a true stack – it requires garbage collection during deterministic computations because it is popped only on backtracking. Additional complications arise with delaying (described later) – in WAM, everything in the local environment must be globalised when a predicate delays. In summary, a machine free list allocation instead of stack allocation can be just as efficient – and much simpler to use.

Each of the registers contains either an object address or is flagged as being empty. When an *n*-ary predicate is called, the caller must put the arguments in registers 0 through *n*–1 and save its registers on the execution stack. The called predicate must ensure that on return all registers are empty – the caller can then pop the saved registers. "Freeing a register" simply means flagging the register as empty – if reference counting is used, the reference count is decremented.

The machine's status is kept in registers:

pc: program counter contains the code segment and offset of the next instruction.

cpc: continuation program counter contains the code segment and offset of the next instruction to be executed after a `return` instruction.

toes: top of execution stack.

tobs: top of backtrack stack.

tors: top of reset stack.

In instructions, each register is annotated:

v: contains a value,

n: empty, possibly requiring the allocation of a new object,

f: contains a value which is emptied after use.

x: is empty and the value is unneeded (*v*+*f*).

If reference counting is used, these annotations mark where reference counts are decremented. If reference counting is not used, registers must still be flagged as empty to minimise the amount of information stored when choice points or "thunks" are created.

Each code segment has a vector of up to 128 constant object addresses. Most instructions allow *c* annotations to indicate that the operand is the index of an entry in the constants' vector.

`Eq` r_1, r_2 unifies two registers. It can also be used to move or copy registers:

`eq n1,f2` moves register 2 into register 1;

`eq n3,v4` puts a copy of register 4 into register 3;

`eq n5,'[]'` loads register 5 with nil.

`EqSkip` is like `eq` except that it skips the following instruction if the equality test succeeds (`eq` fails to the most recent choice point – described later). If the equality test fails, any *f* annotations are ignored. Failure does not undo instantiations caused by unification so there will usually be some tests before `eqSkip` to ensure that the operands are not variables.

`EqLst` r_1, r_{Hd}, r_{Tl} does unification for a list element. `EqLst f1,n2,n3` tests for register 1 containing (the address of) a list element (or, if it is a logical variable, instantiates it to a list element) with the head being put in register 2 and the tail in register 3; register 1 is then emptied. `EqLst f0,n0,n1` is valid – it replaces register 0 by the list element's head (the tail goes into register 1).

`SwAVNL` r_1, r_{Hd}, r_{Tl} jumps to one of the following four instructions depending on whether the register contains an atom, a variable, nil or a list element. In the case of a list element, the head and tail are put into the indicated registers. If the register is a variable, any *f* annotation is ignored. As with `eqSkip`, instantiations caused by unification are not undone

on failure. Other switch instructions exist for multi-way branching on strings or numbers but we will not describe them because they can be emulated by sequences of `eqSkip`s.

`Push` *r* and `pop` *r* are used to push and pop registers on the execution stack before and after a `call` instruction.

The `call` instruction assumes that the argument registers are already loaded; *cpc* is pushed onto the execution stack, *pc* is copied into *cpc* and *pc* is set to the first instruction in the code segment. `Return` does the inverse by copying *cpc* into *pc* and popping the execution stack into *cpc* (all the registers should be empty when a `return` is executed). Using *cpc* this way allows the `lastCall` instruction for tail recursion optimisation (TRO) to function like a `goto`.

The `builtin` *number* instruction is used to extend the instruction set for arithmetic, string manipulation, i/o, etc. A `builtin` is like a `call` except that the register usage may be more idiosyncratic – a `builtin` may succeed, fail or delay.

As a (not very useful) example:

```
p([], []).
p(a.Rst, x.OutRst) :- p(Rst, OutRst).
p(b.Rst, y.OutRst) :- p(Rst, OutRst).
```

is compiled to:

```
  swAVNL   f0,n2,n0  % switch on parm0
  builtin  "error"   % invalid parm
  goto     var       % variable
  goto     nil       % parm0=[]
lst:                 % parm0=Hd.Rst
  eqSkip   f2,'a'    % test Hd = 'a'
  goto     else
  eqLst    f1,'x',n1 % 'x'.OutRst
  lastCallSelf       % p(Rst, OutRst)
else:
  eqSkip   f2,'b'    % test Hd = 'b'
  builtin  "error"   % else: invalid parm
  eqLst    f1,'y',n1 % 'y'.OutRst
  lastCallSelf       % p(Rst, OutRst)
nil:
  eq       f1,'[]'   % result := []
  return
var:
  builtin  "error"   % for now, an error
```

The `lastCallSelf` instruction has the same meaning as `goto 0` (the different opcode helps in debugging). This is a tail recursive call – recursion has been turned into iteration. The code for handling a variable for the first parameter has been left out. The above code implements a deterministic predicate – if backtracking code were added, it would not affect the efficiency of the deterministic code.

Backtracking

The backtrack (choice point) stack and reset stack ("trail") are used to implement backtracking. The backtrack stack could be included with the execution stack (as in WAM) – we have separated it for ease of explanation.

When unification instantiates a value cell which is older than the top choice frame, the cell's address is put on the reset stack. If the cell is newer, backtracking will simply free it, so there is no need to record it. The age number of an uninstantiated variable is the depth of the choice stack when the variable was created, so an entry is pushed onto the reset stack only if the cell's age number is less than the depth of the choice stack (a similar technique is used in WAM except that the relative positions on the stack are used rather than age numbers – global stack cells are considered to be older than local stack cells). Deterministic predicates will not create reset stack entries because such predicates do not create choice points.

Wherever backtracking is desired, choice points must be created on the backtrack stack using the `mkCh` ("make choice point") instruction. Each choice point frame contains sufficient information to reset the machine to the state it was in when the `mkCh` was executed:

contents of all non-empty registers,
value of *cpc*,
top of the execution and reset stacks,
failure instruction address.

When failure occurs, by a unification failing or by an explicit `fail` instruction, all registers are emptied and the top choice point frame is used to fill the registers. The reset and execution stacks are popped to what they were when the choice point was created. As the reset stack is popped, its entries are used to reset objects to uninstantiated. Execution then resumes at the failure instruction address.

For implementing *if-then-else*, the `mkChAt` instruction saves the *tobs* value in a register. A subsequent `cutAt` ("hard cut") pops the choice stack (and reset stack) back to the designated choice point; a `rmChAt` ("soft cut") changes the failure address to point to a `fail` instruction (if the choice point is at the top of the backtrack stack, the choice point is removed instead).

When backtracking occurs, the execution stack must be restored to what it was when the choice entry was made. This means that a `return` instruction may not pop the execution stack if a choice entry needs it – the choice entry "protects" the entry in the execution stack [GLLO85]. The top of execution

stack value in the top choice point frame is used to determine whether or not the execution stack can be popped. Each entry in the execution stack has a back pointer to the previous frame, skipping frames which are protected by the choice stack. If only deterministic predicates are executed, nothing is put onto the choice stack and the execution stack grows and shrinks just like the execution stack in a conventional machine (Algol, Pascal, etc.).

It turns out that simple instructions improve performance, even for software interpreters. Therefore, a call is a number of instructions:

```
link          % toes skips over protected frames
push  ...     % one push for each saved value
call
pop   ...     % one pop for each saved value
unlink        % reset toes below protected frames
```

Similarly, *make choice point* is coded:

```
pushB ...     % one pushB for each non-empty register
mkCh label    % push the failure address, toes, tors
   ...
label:
popB ...      % one popB for each saved register
```

where the `pushB` and `popB` instructions push and pop on the backtrack stack.

Deterministic predicates run slightly slower on the full backtracking machine than on a purely deterministic machine. There are four overheads:

- making choice points rather than just branching to a failure address for *if-then-else*.
- `link` and `unlink` instructions are not needed for deterministic execution.
- testing whether or not an instantiation should push an entry onto the reset stack (for deterministic execution, nothing will ever be pushed).
- recording delay information on the reset stack (described below).

The first two items can be avoided by a smart compiler, using the switch instructions. When backtracking is needed for *if-then-else*, some optimisations of `push` and `pushB` instructions are possible. We can avoid recording delay information on the reset stack by having a "set deterministic mode" instruction so that backtracking information is not recorded.

Delays

A delay is implemented by using a `swAVNL` or `varGoto` instruction to detect that a value is uninstantiated – a `delay` *`r,offset`* instruction then suspends the predicate by saving a thunk (with all the non-empty registers) on the delay list associated with the variable and executing a `return`. When the variable becomes instantiated, all associated delayed predicates are made eligible for resumption – the current predicate is suspended and all the delayed predicates are pushed onto the execution stack except for the oldest which is resumed at the where its *offset* indicated.

Because logic programs do not always distinguish between input and output parameters, the programmer may wish a delay until one of a number of variables becomes instantiated (`append` is such a predicate). `DelayOr` instructions may precede a `delay` instruction. These just add information to the delay list entry which is finished by the `delay` instruction. And-delays are done by delaying on the variables, one by one, in any sequence.[6] For sound negation and proper evaluation of *setof*, another instruction is provided which delays if it (recursively) finds an uninstantiated variable not in a specified list.

When a predicate delays it must also be recorded on the reset stack so that it can be removed from the delay list upon backtracking – when a delayed predicate is woken, it is recorded a second time on the reset stack so that backtracking can put it back on the delay list. An optimisation similar to TRO is performed: if no choice point has been created since the original delay entry was created, both entries are removed from the reset stack (shuffling the stack down if necessary).

Delayed predicates are useful for producer-consumer co-routining. Here is a simple example:

```
coroutine = consume(L), produce(L).
produce(Hd.Tl) = makeOne(Hd),
                 produce(Tl).
consume(Hd?.Tl) :- % "?" means delay on Hd
     useOne(Hd),
     consume(Tl).
consume([]?). % stop at end of list
```

---

6 Or-delays are not strictly necessary – Prolog-II [Colm82] does:

```
freeze(X, Control=c),
freeze(Y, Control=c),
freeze(Control, pred(...)).
```

Unfortunately, this leaves unexecuted predicates lying around.

Similarly, and-delays are done by:

```
freeze(X, freeze(Y, pred(...))).
```

`Consume` immediately delays on `L`. `Produce` then starts and continues with `makeOne` which instantiates `Hd`, wakens `consume` and suspends `makeOne`. `UseOne` then executes, followed by a tail recursive call to `consume` which delays on the uninstantiated tail of the list – this returns to the suspended `produce` and the cycle continues. This example is an infinite loop. Normally, `produce` would have some additional control parameters so that eventually `[]` would be generated to terminate the list `L`.

The list `L` needs to exist just one element at a time because it is used solely as a communication channel. If reference counting is used, the list cells are freed as soon as they are accessed and so the entire list never exists, just the current element.

Weak delays: dynamic reordering of clauses

The *ancestor* predicate is inefficient if the first argument is uninstantiated:

```
ancestor(Ancestor, Descendent) :-
     parent(Ancestor, Descendent).
ancestor(Ancestor, Descendent) :-
     parent(Ancestor, Z),
     ancestor(Z, Descendent).
?- ancestor(X, george).
```

In the second clause, `parent` with two uninstantiated variables will repeatedly generate all `parent` relations by backtracking. Changing `parent` to delay until both parameters are instantiated would prevent this and give efficient execution but would also cause `ancestor` to delay permanently if it is called with two uninstantiated variables or if it computes ancestors beyond grandparents. We have a associate "cost", proportionate to the size of the predicate's solution space, with each `delay` instruction. For example, if we have 100 parent-child relationships, an average of 2 parents per child and 3 children per parent, the code would be:

```
     varGoto     r0, §1
     varGoto     r1, §2
§0:  code for both r0 and r1 instantiated.
     return
§1:  varGoto     r1, §3
     delayCost   2          % delay parm0, cost=2
     delay       r0, §1a    % resume at next instr
§1a: notvarGoto  r0, §0     % parm0 possibly var
     code for r0 uninstantiated and r1 instantiated.
§2:  delayCost   3          % delay parm1, cost=3
     delay       r1, §2a
§2a: notvarGoto  r1, §0
     code for r0 instantiated and r1 uninstantiated.
§3:  delayCost   100        % delay parm1,
     delayOr     r1         % or parm2, cost=100
     delay       r0, §3a
§3a: varGoto     r0, §2
     notvarGoto  r1, §0
     code for r1 uninstantiated and r1 uninstantiated.
```

As before, the machine resumes clauses when their arguments are sufficiently instantiated. If all predicates are blocked, the least expensive one is resumed. The machine thereby dynamically decides the least expensive way to continue a non-deterministic computation.

Parallelism

Some parallel designs have retained standard Prolog with full backtracking [HeNa86]. The intent is to retain the semantics of "standard" Prolog, speeding execution when parallelism can be exploited. As our design provides full "pure" Prolog with co-routining, it can easily be used in such a parallel machine – the co-routining predicates could transparently be executed on a fully parallel machine.

In contrast, guarded Horn clause languages (such as GHC [Ueda85], Concurrent Prolog [Shap83] and Parlog [ClGr84]) have abolished backtracking – GHC has even discriminated against user predicates by not allowing them in guards. Standard Prolog can be implemented in such languages – but that can be said of even Fortran. Rather, we have the equation (attribution uncertain):

flat, safe, concurrent guarded Horn clauses = Occam + logical variable.

We believe that the indeterminicity of the guards – which has caused much semantic difficulty – is not a very valuable feature because the guards are usually mutually exclusive and can be transformed into an equally fast (or faster!) sequence of *if-then-elses*. The valuable feature of these languages is their ability to execute predicates in parallel, communicating via shared logical variables.

Although there are some problems which can benefit enormously from parallel execution, many parallel programs are really disguised co-routining programs because critical sections must execute sequentially. Typically, such programs look like:

server	**requester**
`initialise`	
`loop`	`send request`
`  wait for message`	
`  process message`	`wait for reply`
`  send reply`	
`endloop`	`process reply`

This is a producer-consumer co-routine, using message passing instead of co-routine calls. No matter how much parallelism is available, the speed of the two processes is limited by the slower of the two.

Environments on the execution stack

Warren observed that passing arguments in registers rather than in stack frames has two advantages: tail recursion optimisation can be easily performed and stack frames do not need to be created for unit clauses. However, his design keeps local variables in the stack. We have chosen to keep local variables in registers and to copy them to the execution stack only when they must be preserved across calls.

At first glance, our design appears less efficient. Although there are certainly cases where one or the other design is better, we believe that in for "typical" programs [Mats85], the two designs are similar in efficiency. In practice, only a few registers need to be saved around a call. In WAM code these registers would have to be loaded from the local or global stack anyway, so the number of executed instructions and amount of stack references are about the same (see sample code in the appendix).

The push/pop around a call in our design does not only save values over predicate calls; it also puts values into the correct registers. This simplifies compiler design because register allocation need only consider where the registers are needed between two adjacent calls – the compiler is fewer than 400 lines of Prolog (which took 4 days to write and debug).[7] The compiler seldom has to move the contents of one register to another; in WAM, instructions like `put_pval` are quite common.

Our design allows treating predicates as first class objects (discussed in a separate Technical Report). This is somewhat tricker in WAM because of the need to preserve and restore the state of the current stack frame. Additionally, we can delay and resume a predicate at any instruction whereas WAM is more difficult to delay after an environment (stack frame) has been allocated. Because we do not distinguish between local and global values, we do not need to globalise variables when a delay is made.

7 Exclusive of code for handling delays and for detecting deterministic predicates (we use a slightly different method than that in [DeWa86]; in essence, we detect *if-then-else* situations).

Conclusion

We have explored a variation on the popular WAM implementation of a logic engine. We retain WAM's efficiency, yet we can implement a more powerful language than standard Prolog, providing sound negation and co-routining. Our design is also suitable for functional programming [AbLu86].

We have implemented the logic engine (except for virtual memory), including an optimising compiler, and have attained performance comparable to other WAM implementations (we prefer not to give KLIPS figures because we feel that they are almost meaningless). Our compiler is a short, simple program because our logic engine's instructions lend themselves to easy compilation.

We do not think that our design is the last word in logic engines. We hope that it will inspire others to explore more deviations from current designs.

Acknowledgments

This research has been partially funded by an SUR grant from IBM Canada Ltd.

References

[AbLu86] Abramson, H. and Lüdemann, P.: *Compiling Functional Programming Constructs to a Logic Engine*, Technical Report 86-24, Dept. of Computer Science, University of British Columbia. Submitted to the Fourth International Logic Programming Conference.

[AuHo82] Auslander, M. and Hopkins, M.: *An Overview of the PL.8 Compiler* in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.*

[Clar78] Clark, K.L.: *Negation as failure.* In *Logic and Databases*, Gallaire, H. and Minker, J. (ed.), Plenum Press.

[ClGr84] Clark, K.L. and Gregory, S.: *PARLOG: Parallel Programming in Logic.* Research report DCO 84/4, Dept. of Computing, Imperial College, London. See also *ACM Transactions on Programming Languages and Systems* 8(1) pp. 1-49 (January 1986).

[ClMcC84] Clark, K.L. and McCabe, F.G.: *micro-PROLOG: Programming in Logic*, Prentice-Hall.

[Colm82] Colmerauer, *A.: PROLOG-II Manuel de Référence et Modèle Théorique*, Groupe Intelligence Artificielle, Univ. d'Aix-Marseille II, 1982.

[DDH72] Dahl, O-J., Dijkstra, E.W and Hoare, C.A.R.: *Structured Programming*, Academic Press.

[DeWa86] Debray, S.K. and Warren, D.S.: *Detection and Optimization of Functional Computations in Prolog*. Third International Conference on Logic Programming, Springer-Verlag.

[Dijk76] Dijkstra, E.W.: *A Discipline of Programming*, Prentice-Hall.

[DoPaDe84] Dobry, T.P., Patt, Y.N. and Despain, A.M.: *Design decisions influencing the microarchitecture for a Prolog machine*, Micro 17 Proceedings, October 1984.

[GLLO85] Gabriel, Lindholm, Lusk, Overbeek: *A Tutorial on the Warren Abstract Machine for Computational Logic*. Argonne National Laboratory Report ANL-84-84.

[Gold83] Goldberg, A.: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

[Hend80] Henderson, P.: *Functional Programming: Application and Implementation*. Prentice-Hall.

[HeNa86] Hermenegildo, M.V., Nasr, R.I.: *Efficient Management of Backtracking in AND-Parallelism*. Third International Conference on Logic Programming, Springer-Verlag.

[Inge61] Ingerman, P.Z.: *Thunks – A way of compiling procedure statements with some comments on procedure declarations*, Comm. A.C.M. 4, 1, pp. 55-58.

[Knut73] Knuth, D.E.: *The Art of Computer Programming, Volme 1: Fundamental Algorithms*. pp. 417–420. Addison-Wesley.

[Kras83] Krasner, G. (ed.): *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley.

[Kowa79] Kowalski, R.: *Logic for Problem Solving*. Elsevier North Holland.

[Lloy84] Lloyd, J.W.: *Foundations of Logic Programming* (pp. 45-47), Springer-Verlag.

[LlTo84] Lloyd, J.W. and Topor, R.W.: *Making Prolog more Expressive* in The Journal of Logic Programming, 4 (184).

[Mats85] Matsumoto, H.: *A Static Analysis of Prolog Programs*, SIGPLAN Notices V20 #10, October 1985.

[Mell82] Mellish, C.S.: *An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter*. In *Logic Programming*, Clark, K.L. and Tärnlund, S-A. (ed.), Academic Press.

[Mill86] Mills, J.W.: *A high performance LOW RISC machine for logic programming*, IEEE 1986 3rd International Symposium on Logic Programming.

[Nais85a] Naish, L: *Automating Control for Logic Programs*. The Journal of Logic Programming, Vol. 2, Num. 3, October 1985.

[Nais85b] Naish, L: *Negation and Control in Prolog*. Ph.D. Thesis, University of Melbourne.

[Tick85] Tick, E.: *Prolog Memory-Referencing Behavior*. Stanford University Technical Report No. 85-281 (September 1985).

[Tick86] Tick, E.: *Memory performance of Lisp and Prolog programs*. Third International Conference on Logic Programming, Springer-Verlag.

[TiWa84] Tick, E. and Warren, D.H.D.: *Towards a Pipelined Prolog Processor*. IEEE 1984 International Symposium on Logic Programming.

[Ueda85] Ueda, K. *Guarded Horn Clauses*, ICOT Technical Report TR-103 (June 1983).

[Warr77] Warren, D.H.D.: *Implementing Prolog – Compiling Predicate Logic Programs*. Technical Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

[Warr83] Warren, D.H.D.: *An Abstract Prolog Instruction Set*. SRI Technical Note 309.

[Warr86] Warren, D.H.D.: *Optimizing Tail Recursion* in *Logic Programming and its Applications*, van Caneghan, M. and Warren, D.H.D (ed.), Ablex Publishing.

Appendix: comparison with WAM

In Prolog, here is a typical predicate which applies the predicate q to each element of a list:

```
p([], []).
p(Hd.Tl, HdX.TlX) :-
    q(Hd, HdX), p(Tl, TlX).
```

Here is our code:

```
  swAVNL  f0,n0,n2
  fail              % none of the 3 below
  goto    var       % parm0 is var
  goto    nil       % parm0=[]
                    % continue to lst case
lst:                % parm0=Hd.Tl
  eqLst   f1,n1,n3  % parm1=HdX.TlX
  push    f2        % save Tl
  push    f3        % save TlX
  call    'q/2'     % q(Hd,HdX): regs 0 and
                    % 1 are already set
  pop     n1        % restore TlX
  pop     n0        % restore Tl
  lastCallSelf      % p(Tl,TlX)
nil:
  eq      f1,'[]'
  return
var:
  pushB   v0        % make the choice
  pushB   v1        % point by saving
  pushV   v2        % all active regs
  mkCh    else
  eq      f0,'[]'   % parm0=[]
  goto    nil
else:
  popB    n2        % restore the
  popB    n1        % active regs
  popB    n0        % on failure
  eqLst   f0,n0,n2  % parm0=Hd.Tl
  goto    lst
```

Here is the WAM code:

```
  switch_term  var, nil, lst
var:
  try_me_else else
nil:
  get_nil    1
  proceed
else:
  trust_me_else_fail
lst:
  allocate          % create an environment.
  get_list   1      % arg 1, in reg 1, is a list.
  unify_tvar 1      % Hd is a temporary, put it in
                    % register 1; it'll be needed
                    % there for the call to q/2.
  unify_pvar 2      % Tl is a permanent, save it at
                    % displacement 2 in
                    % environment.
  get_list   2      % arg 2, in reg 2, is a list.
  unify_tvar 2      % same comment as for Hd.
  unify_pvar 3      % TlX is a permanent, save it
                    % at displacement 3 in
                    % environment.
  call       q/2    % notice args are in proper
                    % positions
  put_pval   2, 1   % move Tl into register 1
  put_pval   3, 2   % move TlX into register 2
  deallocate        % get rid of environment
  execute    p/2    % last call
```

Even though the WAM instructions are more complex than ours, more of them are required (for deterministic *append*, the difference is even more dramatic: our design has just 3 instructions in the inner loop). Both have about the same number of memory and register references (in addition to call frame allocation and deallocation,ours has 2 references to the heap and 4 to the execution stack; WAM has 9 memory references [Tick86]). It is difficult to draw any general conclusions from this example; in general, WAM and our design appear to have similar efficiency.