SRN SICS-T--89/16-SE

# An Extension of WAM to Execute Functional Programs

by

Per Kreuger

SICS Technical Report
T89:16
ISSN 1100-3154

SICS/T89/89016

# EWAM:
## An Extension of WAM to Execute Functional Programs

by

Per Kreuger

Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

Friday, November 17, 1989

Abstract:

An extension of WAM to execute functional programs is described. The Warren abstract machine can serve as a basis to implement an SK- graph reduction machine. The effect of such an extension on the Prolog language is described, and its implementation is described in some detail. In particular, the system supports the use of lazy evaluation of functional expressions in Prolog.

# 1. Overview

## 1.1. A WAM Emulator Implemented in Prolog

This paper describes an implementation of an emulator for the Warren Abstract Machine (WAM). The emulator is very simple and its only purpose is to facilitate experimenting with extensions of WAM. The extension described here is an SK-graph reduction machine working in the context of WAM and integrated with it.

## 1.2. Functional Programs in Prolog

One can imagine several ways to extend Prolog with functions. Most of these includes extending Prolog´s concept of equality quite radically. The work described here does not pretend to do that in any novel manner. The problem addressed here is rather that of implementing functional evaluation in a reasonably efficient manner in the context of the WAM. Most system trying to integrate functions into Prolog would face problems similar to the ones described in this paper.

## 1.3. The SK-machine

The SK-machine is a relatively simple graph reduction machine based on the theory of combinators [Tu 79], [HS 86]. There exist several other abstract machines for evaluating functional programs (notably the G-machine) but the simplicity of the SK-machine, and the fact the the data structure representing the program is successively changed into the value (normal form) of the function application made it a clear choice, at least for a first implementation.

## 1.4. Lazy Evaluation

The SK-machine implements the concept of lazy evaluation or call by value. This means that parts of the program that are not needed for the computation of the whole are never computed. In addition parts of programs occurring in several places in a larger program are never computed more than once (provided we have a smart enough compiler). These are very desirable features in a functional language, and we would like to bring as many as possible of these with us when integrating a functional language with Prolog.

# 2. Prolog implementation of a WAM emulator

## 2.1. WAM Overview

The WAM emulator is implemented in Prolog. There is no special reason for this except that it is a language I like to work in. There are very few shortcuts however. The implementation follows Warren´s original description [Wa 83] very closely. All manipulation of the data areas of WAM are done on a representation of a continual address space. The emulator actually does pointer manipulation in the manner of, e.g., a C implementation. The implementation makes very little (if any) use of backtracking, and most recursion is by tail recursion. For these reasons the implementation should be relatively easily port into an iterative language.

## 2.2. The Toplevel

The toplevel is the least clean part of the emulator. Both the reader, the writer and call are actually implemented outside the emulator itself. There is no internal call or read in the emulator. The toplevel simply reads the goal from input, transforms its input into a structure on the WAM heap, loads the A-registers with the arguments of the structure, and starts the emulator with the address of the code for the main functor of the goal. This is not very elegant and should be improved in future versions.

There is also a loader, that loads WAM code into the code area of the representation of memory.

## 2.3. The WAM

The WAM is implemented as a large predicate ('WAM') indexed on the WAM instructions. Two more arguments represent the state description before and after the instruction execution.

The emulator is implemented by a small tail recursive program that calls 'WAM' for each instruction it reads from the state description.

## 2.4. The State Representation

WAM´s state is represented as the following 12-tuple

```
(Mode P CP E B A TR H HB S Regs Memory),
```
where:

`Mode` is either write mode or read mode
`P` is the Program Pointer (to the code area)
`CP` is the Continuation Pointer (to the code area)

`E` is a pointer to the last Environment (on the local stack)
`B` is a pointer to the last Choice point (on the local stack)
`A` is the current top of the local stack

`TR` is the top of the trail
`H` is the top of the heap
`HB` is the heap backtrack point

`S` is the structure pointer (to the heap)

`Regs` is a representation of Temporary/Argument registers
`Memory` is a representation of the memory containing all the stacks etc.

Choice points and environments are represented as objects on WAM´s stack in the manner of vanilla WAM.

## 2.5. The Memory Representation

itself is: why not use Prolog´s list structures on WAM´s heap as a representation of the graph!

## 4. Integrating the SK-machine into the WAM Framework

### 4.1. Prolog Lists as Functional Expressions

In other words we could use WAM´s heap as storage for our graph, and let the user construct functional expressions using ordinary Prolog lists. The lists would of course have to be compiled into a combinator expression.

The SK-machine could then operate directly on the data structures already present in WAM.

All this suggests that we represent the graph that the SK-machine operates on as a list. This is actually more general than necessary. As function application is always curried, each function has exactly one argument, and the application could be represented as a pair, rather than as a list.

There is a tradeoff between transforming the list representation of an expression supplied by the WAM into pairs, and actually operating on lists in the SK-machine. The current implementation has chosen to retain the lists in the graph representation with a resulting overhead in the SK-machine. It is quite possible that it would be more efficient to do a runtime transformation of these lists into pairs before calling the SK-machine, and back again after it terminates.

### 4.2. The Instruction Set

The extension to the WAM instruction set is simply the addition of the `eval` instruction. `eval` takes two arguments: the form to be evaluated, and the term to be unified with the result[1]. The implementation simply dereferences its first argument, runs a transformation on the resulting expression and calls the SK-machine to reduce the resulting graph.

### 4.3. Compilation and Runtime Transformations

When we call the WAM instruction `eval` we expect `eval` to act as an interpreter for our functional language. As the SK-machine depends on a compilation of the functional program into a combinator expression we would first have to compile the expression supplied to `eval`, second let the SK-machine do its job with the result, and then finally perhaps see to it that the representation of the resulting expression is compatible with what WAM expects. All of this would have to be done in runtime.

An alternative is to compile the functions that you know that we will use. We would still have to do some runtime processing of the functional expression, but much less than if we were to do complete compilation at every call to `eval`. Of course this does not exclude the support of full compilation in runtime .

The implementation does not currently support full compilation in runtime, but it lets you define functions and use their definitions in calls to `eval`. The only runtime processing that is done is currying the expressions supplied to `eval`, and translating quoted lists into a functional representation in order to differentiate them from Prolog lists (which are used to represent functional applications).

---

1. A quite possible further extension would be to evaluate both arguments to eval. This presupposes a way to distinguish between Prolog´s data structures and the functional expression expected by eval.

A similar transformation of lists, but in the reverse order, is done with the result when the SK-machine terminates.

## 4.4. The Stack, the Heap and Garbage

One of the main ideas of this work was to use the data areas of WAM as an environment for the SK-machine. The SK-machine uses two types of data structures for its execution: a stack, and a representation of a graph corresponding to a program. The memory used by the stack is needed only during the execution of the functional program and can be discarded when it terminates.

When incorporating the SK-machine with WAM, the SK-stack could have been put almost anywhere. I choose to put it on top of the WAM stack, as this will not change during the execution of the functional program. The case of the graph representation is somewhat more complicated. As the functional expressions supplied to `eval` consist of (possibly nested) Prolog lists, it is natural to let these lists (after certain runtime transformations) represent the graph or functional program.

When executing, the SK-machine successively changes its program or combinator expression into a representation of its value. It does this by destructively changing pointers in the data structure representing the expression, and creating new structures on some suitably structured memory. In practice this means that the SK-machine will construct Prolog lists on the WAM's heap, and put pointers to these structures in old areas of the heap. Not all of the memory used by the SK-machine will be referenced by the old heap after the SK-machine terminates (in fact, for most programs only a very small part of it will).

All of this leaves us with a considerable amount of garbage on top of the Prolog heap after the execution of `eval`. An ordinary WAM garbage collector should take care of this, but as the amount of garbage can be quite large, its locality is good and the fact that we should be able to infer which parts of it are actually used, we may consider implementing a specialized garbage collector for the SK-machine. This would have the additional benefit that it would be easy to call from inside the SK-machine should heap space run low.

The current implementation does not feature such a garbage collector, but its implementation is a possible extension of this work.

## 4.5. Lazy Evaluation

Lazy evaluation is an evaluation order for functional programs that guarantees the evaluation procedure will terminate with a normal form of the expression supplied to it, if there is one. That is, it is a complete procedure for normalization. When incorporating the SK-machine into WAM we would like to preserve as much as possible of the SK-machine's inherent laziness.

The effect of lazy evaluation can be observed in several ways.

### 4.5.1. Call by Name

Lazy evaluation means that when evaluating a function, we do not evaluate its arguments unless we really need their value. In the usual context of functional programming this means that we never evaluate anything unless we need to

print its value. In the case of a functional language incorporated into Prolog, there are really two ways to go.

The simplest, and the one implemented by the system described here is to fully evaluate the expression supplied to `eval`. This has the advantage that once the call to evaluate terminates, the resulting structure is a representation of the value of the function application. The disadvantage, of course, is that unless the value is really used the computation may be unnecessary.

To implement full lazy evaluation in the Prolog context would involve freezing the evaluation to each call to `eval` (and unifications of the resulting terms) until the value is actually needed for the computed answer.

There are however a number of features that even the simpler approach achieves.

### 4.5.2. Infinite Objects

We can represent infinite objects by means of recursive functions. The standard example is a list of the natural numbers. This list can be represented as the function:

```
(defun from (n) (cons n (from (+ 1 n)))).
```

If we try to evaluate the call `(from 0)` the computation will not terminate but continue to construct a list of all natural numbers. Lazy evaluation means that in spite of this fact we could compute any object in this list. The call `(hd (from 0))` will return `0`, `(hd (tl (from 0)), 1` etc.

### 4.5.3. Sharing & Destructive Change

A property which is not strictly part of lazy evaluation, but is usually associated with it, is that subexpressions that occur at more than one place in an expression never have to be evaluated more than once. The reason this feature is mentioned in this context is probably that call by name would be hopelessly inefficient without it. For example in the application:

```
((lambda (x) (f x (g x))) lic),
```

where `lic` is a long involved calculation, we would not like to perform lic more than once.

In any case this feature is also inherent in the SK-machine. It is also inherited into Prolog; extended in the manner we have suggested.

All (bound) Prolog variables containing references to the list representation of a functional expression will actually refer to its value after the call to `eval`. This is true of all subexpressions of the complete expression as well.

This is not merely a matter of efficiency, it also means that the arguments of `eval` will be unifiable after the call to `eval` (if the call terminates) as well as all other references to the original expression. Once we use a list as a functional expression, its interpretation is clear. It is no longer regarded as a list, but as a representation of its value as a functional expression.

### 4.5.4. Trailing and Backtracking

The fact that we actually change the bindings for variables when evaluating functional expressions means that we have to reconsider the concept of backtracking. The fact that we may have to undo the effect of evaluating a functional expression means that we somehow have to record the old values of all pointers changed during the evaluation process.

Well not all values... it turns out that exactly the same criteria can be used in this context as in the case of trailing unbound variables. If the pointer got its value after the creation of the last choice point we do not need to record its old value. If we do have to record its old value we do this by making a special kind of trail entry, recording not only the address of the variable (as in the case of ordinary variable binding) but also its old value (if it has one). There are conceivably other ways to handle this, but the cost should prove to be approximately the same. We record exactly the information we have to, nothing else.

## 5. Examples

The following is a trace of some demos and examples contained in the file `ewam-tests` (see Appendix A). A more complete trace appears in the file `ewam-tests.trace`. The file `demos` contains in addition to the examples in `ewam-tests` some demonstrations of the capabilities of the WAM emulator (nothing special, except for the fact that it works).

```
Starting ZYX Macintosh Prolog version (2 7).
yes
```

;;; define a trivial predicate to test the evaluator!

```
; (defpred (eval * +)
;    ((my-eval ?result ?expr)
;     (eval ?result ?expr)))

(load-predicate (my-eval 2)
  (?     (eval (reg 1) (reg 2)))
  (?     (proceed)))
yes


;---- Prove: ----
(top)                                          ; Start top-loop
```

; Some arithmetic:

; Numerical constants

```
;---- Enter: ----
{my-eval ?result 1}
?result = 1
```

```
YES

;---- Enter: ----
{my-eval ?result (+ (* 2 3) (/ 6 (- 5 2)))}
?result = 8
YES
```

; Symbolic constants

```
;---- Enter: ----
{my-eval ?result a}                    ; no need to quote constants!
?result = a
YES

;---- Enter: ----
{my-eval ?result (quote (a b c))}      ; lists must be quoted however!
?result = (a b c)
YES
```

; List operations (note that result is fully evaluated)

```
;---- Enter: ----
{my-eval ?result (cons a (quote (b c)))}
?result = (a b c)
YES

;---- Enter: ----
{my-eval ?result (hd (quote (a b c)))}
?result = a
YES

;---- Enter: ----
{my-eval ?result (tl (cons a (quote (b c))))}
?result = (b c)
YES
```

;;; define the append function

```
; (defun append (l1 l2) (if (eq  l1 ()) l2 (cons (hd  l1) (append (tl
l1) l2))))

;---- Enter: ----
{my-eval ?result (append (quote (a b)) (quote (c d)))}
?result = (a b c d)
YES
```

;;; faculty demonstrates recursion

```
; (defun fac (n) (if (eq 0 n) 1 (* n (fac (- n 1)))))

;---- Enter: ----
{my-eval ?result (fac 5)}
?result = 120
YES
```

;;; fibbonacci uses lazy evaluation (I hope)

```
; (defun fib (x) (if (eq x 0) 1 (if (eq x 1) 1 (+ (fib (- x 1)) (fib (-
x 2))))))

;---- Enter: ----
{my-eval ?result (fib 5)}
?result = 8
YES
```

;;; f throws away its second argument, but does not terminate!
;;; Demonstrates lazy evaluation!

```
; (defun bot (x) (bot x))
; (defun f (x y) (* 2 x))

;---- Enter: ----
{my-eval ?result (f 5 (bot x))}
?result = 10
YES

;---- Enter: ----
{my-eval ?result (f (bot x) 5)}
[Ok]
```

;;; Manually aborted. does not terminate (naturally)

```
;---- Prove: ----
(top)                                                    ; Restart top-loop

;---- Enter: ----
{my-eval ?result (f 5 ?Variable)}     ; It works for Prolog variables as well
?result = 10
?variable = (ref 2010)
YES
```

;;; from is a function representing the infinite list of natural numbers

```
; (defun from (n) (cons n (from (+ 1 n))))
```

;;; Its first element:

```
;---- Enter: ----
{my-eval ?result (hd (from 0))}
?result = 0
YES
```

;;; Its second:

```
;---- Enter: ----
{my-eval ?result (hd (tl (from 0)))}
?result = 1
YES
```

;;; And the whole list:

```
;---- Enter: ----
{my-eval ?result (from 0)}
[Ok]
```

; Manually aborted. Does not terminate!

```
;---- Prove: ----
(top)                                          ; Restart top-loop
```

;;; Backtracking works!

```
; (defun back
;    ((back ?x ?y ?z)
;     (eval ?x (+ ?y ?z)))
;    ((back ?x ?y ?z)
;     (eval ?x (* ?x ?y))))


;---- Enter: ----
{back ?value 2 3}
Heap grown by 4 cells
?value = 5
YES

;---- Enter: ----
;                                              ; More solutions!
Heap grown by 4 cells
?value = 6
YES
;---- Enter: ----
;                                              ; More solutions?
NO
;---- Enter: ----
```

;;; Sharing of structures works

```
; (defun share
;    ((share ?sum ?prod ?x ?y)
;     (eval ?sum (+ ?x ?y))
;     (eval ?prod (* ?x ?y))))

;---- Enter: ----
{share ?s ?p (+ 1 1) (- 4 1)}
Heap grown by 12 cells                         ; in first call to eval
Heap grown by 4 cells                          ; in second call to eval
?s = 5
?p = 6
YES
```

;;; Note that the second call to eval produced less garbage on heap. This
;;; is because pointers to the lists (+ 1 1) and (- 4 1) have been
;;; replaced by pointers to the values of these lists when regarded as
;;; functional expressions. On backtracking this is undone of course.

# References


| | | |
|---|---|---|
| HS 86 | J . Roger Hindley<br>Jonathan P. Seldin | *Introduction to Combinators and λ-calculus*<br>Cambridge U.P., Cambridge 86 |
| Tu 79 | D. A. Turner | *A New Implementation Technique for Applicative Languages*<br>in Software-Practice & Experience,<br>Vol 9, 31-49 (1979). |
| Wa 83 | David H. D. Warren | *An abstract Prolog Instruction Set*<br>SRI Technical Note 309 (1983) |

# Appendix A

The complete code is available separately from:
uucp: piak@sics.se; or
the SICS prototype archive,
reference : SICS/PA89/89001

Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden