The AM stack machine specification

Robin Cockett
Department of Computer Science
University of Calgary
Alberta

April 11, 2007

1 Introduction

The AM machine is not designed for efficiency: its aim is to illustrate the details of how stack-based runtime storage management is arranged. In particular it illustrates how activation records for functions and blocks can be arranged. The machine has a heap with automatic garbage collection.

While the details of how the machine is implemented are not important it is, nonetheless, interesting to know: it is actually implemented in SML. The stack, itself, is implemented as a doubly linked list so that one can travel up and down it with ease: the heap is implemented as lists of stack elements.

The fact that AM is implemented in SML allows one to do extensive type checking that otherwise might be difficult. This means that incorrect assembler will usually break the type constraints long before it starts exhibiting unexpected behaviour. This makes it a useful teaching tool which, after all, is its primary purpose.

2 Machine Layout

The machine has the following registers (which can be manipulated):

```
%sp   --- stack pointer
%fp   --- frame pointer
%cp   --- code pointer
```

One can alter `%fp` and `%cp` directly but as it is intended that `%sp` alway should point to the head of the stack one cannot directly change this without altering the stack size.

AM also has a "virtual register" which is the conceptual current top of the heap.

```
%hp  --- the heap pointer
```

which cannot be directly manipulated. Instead, this is affected by allocating heap elements.

On the stack are stored:

real/float values	VALUE_F
integer values	VALUE_I
boolean values	VALUE_B
character values	VALUE_C
void cells	VOID
stack pointers	STACK_PTR
heap pointers	HEAP_PTR
code pointers	CODE_PTR.

All computations proceed by either LOADing data onto the stack, APPlying operations on the top two elements, or STOREing the top (or top n) stack element either elsewhere onto the stack, onto the heap, or into a register.

There are no register to register addressing modes.

Conceptually the heap is storage which grows opposing the stack. However, the only access into the heap is through heap pointers and the only way to create a heap pointer is to either allocate heap storage or to copy the top n stack locations onto the heap.

2.1 The instructions

Here are the instructions:

1. Frame pointer instructions:

   `LOAD_R %fp`: Pushes `%fp` onto the top of the stack (literally load onto the stack register `%fp`)

   `LOAD_R %sp`: Pushes `%sp` onto the top of the stack (literally load onto the stack register `%sp`)

   `STORE_R %fp`: Removes the stack pointer on the top of the stack and sets register `%fp` to this value. (literally store to register `%fp` top of stack)

2. Code pointer instructions:

   `JUMP <code label>`: Sets the `%cp` to the value of the code label

   `JUMP_S`: Removes the code pointer on top of the stack and sets `%cp` to this value advanced by one step.

   `JUMP_C <code label>`: Removes the boolean value on top of the stack and if it is FALSE sets `%cp` to the value of the code label.

   `JUMP_O`: Removes the (positive) integer on top of the stack and moves the code pointer that number of steps ahead:

1	=	next instruction
2	=	skip instruction
3	=	skip 2 instructions
	....	

   `LOAD_R %cp`: Saves the program pointer advanced by one step on the stack. (literally load onto the stack register `%cp`)

`HALT`: End of execution

3. Data manipulation on stack:

   `LOAD_F r`: Load a constant float/real number onto the stack.

   `LOAD_I i`: Load a constant integer onto the stack.

   `LOAD_B b`: Load a constant boolean onto the stack.

   `LOAD_C c` Load a character onto the sack.

   `LOAD_O m`: Removes the stack pointer on top of the stack and replaces it by the value at offset m from the removed pointer. Note m can be negative in which case the offset is deeper into the stack.

   `LOAD_OS`: Removes the integer m and the stack pointer from the top of the stack and replaces it by the value at offset m from the removed pointer. Note m can be negative in which case the offset is deeper into the stack.

   `STORE_O m`: Removes the stack pointer on top of the stack and the cell below and replaces the cell at offset m from the removed stack pointer by the removed cell.

   `STORE_OS`: Removes the integer m, the stack pointer and the cell below from the top of the stack and replaces the cell at offset m from the removed stack pointer by the removed cell.

   `ALLOC n`: Creates n void cells on top of the stack. To deallocate n cells from the top of the stack use a negative number. Removes the top n cells from the stack.

   `ALLOC_S`: Removes the integer n from the top of the stack and creates n void cells on top of the stack. To deallocate n cells from the top of the stack use a negative number. Removes the top n cells (below the integer) from the stack.

4. Data manipulation on heap

   `LOAD_H` : Removes the heap pointer on top the stack and replaces it by the elements of the heap record loaded on top of the stack.

   `LOAD_HO m`: Removes the heap pointer on top of the stack and replaces it by the value at offset m from the removed pointer in the heap. Note m cannot be negative nor can it be outside the heap block which is being accessed.

   `STORE_H m`: Removes the top m elements of the stack and replaces them by a heap pointer to a heap record consisting of those top m elements.

   `STORE_HO m`: Removes the heap pointer on top of the stack and the cell below and replaces the cell at offset m from the removed heap pointer by the removed cell. Note m cannot be negative nor can it be outside the heap block which is being accessed.

   `ALLOC_H n`: Creates a block of n void cells on top of the heap and places the heap pointer to them on the top of the stack. Note one can only allocate to the heap there is no deallocation. Note that n must be greater than zero.

5. Arithmetic operations

APP <operation>: Removes the top one/two integer/boolean values and replaces them with the resulting of applying the operation.

Values of ¡operation¿ are:

(a) Float/real operations:

```
ADD_F SUB_F DIV_F MUL_F   arity 2
NEG_F FLOOR CIEL          arity 1
```

(b) Integer operations:

```
ADD SUB DIV MUL   arity 2
NEG FLOAT         arity 1
```

(c) comparison operations on floats/reals

```
LT_F,LE_F,GT_F,
GE_F,EQ_F         arity 2
```

(d) comparison operations on integers

```
LT,LE,GT,
GE,EQ             arity 2
```

(e) comparison operations on characters

```
LT_C,LE_C,GT_C,
GE_C,EQ_C         arity 2
```

(f) logical operations on booleans

```
AND,OR            arity 2
NOT               arity 1
```

READ_F: Read float/real value onto top of stack.

READ_I: Read integer value onto top of stack.

READ_B: Read boolean value onto top of stack.

READ_C: Reads next character to top of stack (the system buffers characters until the first newline).

PRINT_F: Removes top real value of stack and prints it.

PRINT_I: Removes top integer value of stack and prints it.

PRINT_B: Removes top boolean value of stack and prints it.

PRINT_C: Removes top character value from top of stack and prints it.

6. Labels:

Any instruction can be labelled. Labels must start with a lower case letter and can be followed by any sequence of lowercase letters, digits, or underscore.

```
<label> : <instruction>
        ( {label} = [a-z][_a-z0-9]* )
```

A given line may contain many instructions: a label refers to the instruction it precedes.

2.2 Synopsis of instructions

Notice that there are the following types of instruction:

`LOAD`: This is loading something onto the stack/heap

`STORE`: This is storing something on (or near) the top of the stack/heap somewhere.

- `_R` means registers (needs register argument)
- `_O` means stack offset (needs integer argument and stack pointer on top of the stack)
- `_H` means the top of the stack to depth of cell is either stored on heap leaving a heap pointer or retrieved
- `_HO` means heap offset (needs a positive integer argument and heap pointer on top of the stack)
- `_F` means an real/float constant (needs float argument)
- `_I` means an integer constant (needs integer argument)
- `_B` means a boolean constant (needs boolean argument)

`JUMP`: this alters the program counter (possibly)

- absolute jump
- `_C` conditional on the top of stack
- `_S` to code pointer on top of stack
- `_O` advances the code pointer the number of steps indicated by the integer on the top of the stack.

`APP`: This applies an operation to the top few elements of the stack (needs operation argument).

`ALLOC`: allocating and deallocating space on the stack

- needs integer argument
- `_S` allocating and deallocating space on the stack integer argument is taken from the top of the stack.
- `_H` allocates space on the heap needs positive integer argument.

`PRINT_F/I/B/C, READ_F/I/B/C`: interacts with outside world!

3 Activation record layout for `M+` functions

When a function is called in m+ here is the arrangement of the activation record:

BEFORE

argument 1	
argument n	
void	
static link	
dynamic link	
return ptr	
local var 1	← `%fp`
local var n	
dealloc counter	
local arrays	
local var n	

⇒

AFTER

return value
return pointer

3.1 Caller/callee responsibilities

In writing the code for creating an activation record the one tricky thing is to assign responsibilities correctly. The caller MUST BE responsible for

(i) setting the "return" code pointer

(ii) setting the access link for the called routine

as (i) only the caller knows where the code should be continued (ii) only the caller knows how many (static) levels the place of the call to the function is from the point of definition of the function.

In contrast either the caller or the callee can save the current frame pointer and reset it to the top of stack. Here we make the former the responsibility of the caller and the latter the responsibility of the callee.

Entering a function

Thus, entry into a `M+` function requires:

1. Load the arguments on the stack (caller code: usual postfix evaluation)
2. Caller starts the function call sequence:
   (a) Create space for a return value (this is vital if the function has no arguments: the caller is conservative and allocates the space anyway!).
   (caller code: `ALLOC 1` this creates a "void" stack element)

(b) Calculate the access pointer (caller code `LOAD_R %fp LOAD_O -2 LOAD_O -2`)

(c) save the current frame pointer - dynamic link
(caller code: `LOAD_R %fp`)

(d) Loads the (return) program counter on the stack
(caller code: `LOAD_R %cp` this advances the program counter one step and saves it on the stack recall that loading a code pointer will also advances it ...)

(e) Call the function
(caller code: `JUMP function-label`.)

3. Function finishes the call sequence:

   (a) Sets the frame pointer to top of stack
   (function code: `LOAD_R %sp STORE_R %fp`)

   (b) Allocate the static local storage
   (function code: `ALLOC m`)

   (c) Initializes the deallocation counter
   (function code: `LOAD_I m+2`)

   (d) Allocate the local arrays by calculating their bounds and allocating the required space and record this cumulative total in the deallocation counter.

4. continue with the function body.

Returning from a function

Return from a function involves:

1. Write the return value to the first argument slot. We assume the return value is at the top stack value: (function code: `LOAD_R %fp STORE_O -(n+3)`)

2. Write the return pointer to the second argument slot
(function code: `LOAD_R %fp LOAD_O 0 LOAD_R %fp STORE_O -(n+2)`)

3. Deallocate the local storage
(function code: `LOAD_R %fp LOAD_O m+1 APP NEG ALLOC_S`)

4. Restore the old frame pointers
(function code: `STORE_R %fp`)

5. Clean up the argument storage leaving the return pointer and return value on the stack (function code: `ALLOC -n`)

6. Do a jump based on the code address on the stack ... (function code: `JUMP_S`)

Collected code

Collecting this code we have:

(A) caller code:

```
        .....
        ALLOC 1                     % void on stack
        LOAD_R %fp                  %
        LOAD_O -2                   % calculate static/access link
        LOAD_O -2                   %
        LOAD_R %fp                  % set the dynamic link
        STORE_R %cp                 % save program counter
        JUMP <function label>       % jump to function code
        ......
```

(B) function/callee code:

```
<function label> : LOAD_R  %sp
                   STORE_R %fp                 % set new FP to top stack element
                   ALLOC m                     % allocate m void cells
                   LOAD_I m+2                  % initializes deallocation counter
                   ......


                   ......                      % value of result loaded
                   LOAD_R %fp                  % load frame pointer
                   STORE -(n+3)                % store <result> at begining
                   LOAD_R %fp                  % load frame pointer
                   LOAD_O 0                    % load <return>
                   LOAD_R %fp
                   STORE_O -(n+2)              % place <return> below <result>
                   LOAD_O m+1                  % retrieve the deallocation pointer
                   APP NEG                     % negate it
                   ALLOC_S                     % deallocate the local storage
                   STORE_R %fp                 % restore old frame pointer
                   ALLOC -n                    % remove top n stack elements
                   JUMP_S                      & jump to top of stack code address
```

3.2 Example of a recursive program call ...

Here is a complete example involving non-local references and a recursive function call. The m+ program is (roughly!):

```
var x:int;
var y:int;
fun exp(b:int):int
```

```
{ var z:int;
  begin if b=0 then z:= 1
        else z:= x * exp(b-1);
  return z;
  end
}
begin
  read x;
  read y;
  print exp(y);
end
```

This should translate into the following m+ assembler (which you should be able to run on the am+ machine simulator:

```
            LOAD_R %sp                  % access pointer for program
            LOAD_R %sp                  % set the frame pointer for prog
            STORE_R %fp
            ALLOC 2                     % allocate space for variables
            READ_I                      % read and store x
            LOAD_R %fp
            STORE_O 1
            READ_I                      % read and store y
            LOAD_R %fp
            STORE_O 2
            LOAD_R %fp                  %  load y
            LOAD_O 2
            ALLOC 1                     % void on stack for return value
            LOAD_R %fp                  % static link
            LOAD_R %fp                  % dynamic link
            LOAD_R %cp                  % save program counter
            JUMP fun_exp                % call function
            PRINT_I                     % print result
            ALLOC -3                    % deallocate stack
            HALT                        %  HALT!
% code for function exp ...
fun_exp :   LOAD_R  %sp
            STORE_R %fp                 % set new %fp to top stack element
            ALLOC 1                     % allocate 1 void cell for local var
            LOAD_I -3                   % set deallocation counter
            LOAD_R %fp
            LOAD_O -4                   % load argument of fuction
            LOAD_I 0
            APP EQ                      % test whether zero
            JUMP_C  label1              %  conditional
            LOAD_I 1
```

```
          LOAD_R %fp                  % set z to 1
          STORE_O 1
          JUMP label2
label1 :  LOAD_R %fp
          LOAD_O -2                   % access x by chasing static link once
          LOAD_O 1
          LOAD_R %fp
          LOAD_O -4                   % load argument b
          LOAD_I 1
          APP SUB                     % obtain b-1
          ALLOC 1
          LOAD_R %fp                  % access link
          LOAD_O -2
          LOAD_R %fp                  % dynamic link
          LOAD_R %cp                  % save program counter
          JUMP fun_exp                % call function recursively
          APP MUL
          LOAD_R %fp
          STORE_O 1                   % store result in z
label2:   LOAD_R %fp
          LOAD_O 1                    % load z again!
          LOAD_R %fp                  % load frame pointers
          STORE_O -4                  % store <result> at begining
          LOAD_R %fp                  % load frame pointer
          LOAD_O 0                    % load <return>
          LOAD_R %fp
          STORE_O -3                  % place <return> below <result>
          LOAD_R %fp
          LOAD_O 2                    % load deallocation counter
          ALLOC_S                     % remove top m+1 stack elements
          STORE_R %fp                 % restore old frame pointer
          ALLOC -1                    % remove top n stack elements
          JUMP_S                      % jump to top of stack code address
```

4 Activation record layout for M+ blocks

A block is treated like a function call in that one must set up space for local variables and arrays which must be deallocated on leaving the block.

Before entering a block the frame pointer is sitting below the local variables and arrays allocated for the calling routine. Entry into a block requires creating an activation record with its own frame pointer. This means that one must save the previous frame pointer. A block, however, does not need an explicit dynamic pointer as the dynamic and access pointers are always the same; nor does it need a return code pointer as the next instruction is the code to which one returns. One can therefore have a stripped-down activation record:

static/dynamic link	
void	
void	← %fp
local var 1	
local var n	
dealloc counter	
local arrays	

Notice that on entering a block a new activation record is created: this means that the compiler has to calculate the access required for variable addresses using the block as the current frame. To allow uniform access pointer calculations a void and a copy of the frame pointer has been inserted in the activation record so the access link (which is the same as the dynamic link, namely, the previous frame pointer) is still at position -2.

On exit from the block the only action is to decrement the stack pointer by $m + 2$, where m is the number of cells the block allocated for the local storage and to reinstate the old frame pointer.

Entering a block

Here is the entry sequence for a block invocation:

(a) Save the old frame pointer

(function code: `LOAD_R %fp`).

(b) Allocate two void stack cell

(function code: `ALLOC 2`).

(c) Alter the frame pointer to point at the last void cell cell

(function code: `LOAD_R %sp STORE_R %fp`).

(d) Allocate the required fixed local storage for the block

(function code: `ALLOC m`).

(e) Set the deallocation counter

(function code: `LOAD_I m+3`).

(f) Allocate the local arrays.

(g) Continue execution of the block code.

Leaving a block

Here is the exit sequence for leaving a block:

(a) Deallocate the local storage by decrementing the stack pointer (block)

(function code: `LOAD_R %fp LOAD_O m+1 APP NEG ALLOC_S`)

(b) Reset the frame pointer: after deallocation the old frame pointer should be sitting on top of the stack. This is how the number $m + 3$ was chosen!

(function code `STORE_R %fp`).

(c) Continue execution (callers code)

Collected code

Collecting this code we have:

(A) Entering a block:

```
LOAD_R %fp
ALLOC 2
LOAD_R %sp
STORE_R %fp
ALLOC m
LOAD_I m+3  % deallocation counter
.....
```

(B) Leaving a block:

```
LOAD_R %fp
LOAD_O m+1  % deallocation counter
APP NEG
ALLOC_S
STORE_R %fp
```

5 Using the heap in M+

Data decarlarations such as:

```
var x,y: intlist;
data intlist = #nil | #cons of int * intlist;
```

allow one to construct items which are stored on the heap. For example:

```
data intlist = #nil | #cons of int * intlist;
x := #nil;
y := #cons(6,#nil);
```

Both `x` and `y` should become heap pointers. `x` will point to a heap item with one entry, namely the constructor index. When a data declaration is made each constructor is assigned a constructor index (which is represented in the symbol table the "jump offset" associated with that constructor):

Constructor	Index/Jump offset
`#nil`	1
`#cons`	2

Then the variable `x` becomes an entry on the stack which is a heap pointer to a heap record with one entry namely the index of `#nil`:

$$\texttt{x} \longrightarrow \boxed{1} \qquad \text{(the constructor index)}$$

On the otherhand the variable `y` becomes a heap pointer to a heap record with three entries: the constructor index, the value 6 and the heap pointer `x`:

$$\texttt{y} \longrightarrow \boxed{\begin{array}{c} 2 \\ 6 \\ \mathrm{x} \end{array}}$$

The code to create a such heap items involves creating the record on top of he stack and then storing it on the heap. For the heap pointer `x` (which points to `#nil`) this is:

```
LOAD_I 1
STORE_H 1
```

which leaves a heap pointer on top of the stack. For the heap pointer for `y` the code may look like:

```
LOAD_I 1
STORE_H 1              % creating pointer to #nil
LOAD_I 6
LOAD_I 2               % constructor index
STORE_H 3              % creat heap pointer to #cons(6,#nil)
```

Associated with datatypes is the case statement: below are two suggestions of how to implement it. They differ by whether one sets up the required activation record on entering the case or on the branches. The latter method is slighly more efficient although it requires more code.

In both methods on entering a case one needs to calculate the argument to the case: this leaves a heap pointer sitting on the top of the stack.

In the first method one then sets up an activation record similar to a block with this heap pointer immediately below it: one then copies the heap pointer to the stack in the activation record itself. When one has the heap pointer to the heap record of the datatype on top of the stack in the activation record one reloads the heap record onto the stack (with LOAD_H).

	heap pointer
	static/dynamic link
	void
fp →	void
	copy of pointer

LOAD_H

	heap pointer
	static/dynamic link
	void
fp →	void
	cons argument 1
	...
	cons argument n
	jump offset

This leaves the constructor index with any arguments below it on the top of the stack. One then does a JUMP_O: the next instructions should be the jump instructions to the appropriate case branch code. This code is now treated like code inside a block although remember the arguments for the case are now on top of the activation record and so can be accessed by offsets from the frame pointer. The last instructions for the branch deallocates the local variables and the void items which start the block. When one returns to the common code one restores the frame pointer (which is on top of the stack) and one must remove the pointer to the argument of the case.

Here is a description of the code required for this method:

1. Calculate the value of the argument of the case statement leaving its heap pointer on top of the stack.

2. Set up an activation record, reset the frame pointer, copy the heap pointer into the activation record and perform a jump offset to the branch code:

```
LOAD_R %fp
ALLOC 2
LOAD_R %sp
STORE_R %fp
LOAD_R %fp
LOAD_O -3    % copying the heap pointer
LOAD_H
JUMP_O
JUMP case_1
 ...
JUMP case_n
```

3. On the branch of the case after completing the code deallocate the local storage (return sequence on branch):

```
case_i : ....                   % code for branch
         ALLOC -(n+2)           % remove void cells
         JUMP return_case
```

4. On return to the common code complete the return sequence: restore the frame pointer, remove the heap pointer and continue.

```
case_ret :
         STORE_R %fp            % restore fame pointer
         ALLOC -1               % remove heappointer
         ...
```

In the second method one sets up the activation record after the jump and accesses the arguments by *negative* offsets from the frame pointer. These offsets are calculated and stored when they are added to the symbol table so one must have set up the appropriate calculation at that stage. This technique has the advantage that one does not have to copy the heap pointer one simply loads the heap item and does a jump offset. However, this does mean that each branch must now individually set up its own activation record and remove it. This does increases the code size on each branch and so, the overall code size, as the entry and return activation code must be repeated.

In this case the stack after setting up the activation record on a branch looks like:

	cons argument 1
	...
	cons argument n
	static/dynamic link
	void
fp →	void

Note that the arguments are below the frame pointer this time so one must access them by negative offset (all other variables are treated as non-local). At the end of the code from the branch one must deallocate the local storage restoring the frame pointer.

Here is a description of the code required for this method:

1. Calculate the value of the argument of the case statement leaving its heap pointer on top of the stack.

2. Load the heap record and do a jump offset:

```
LOAD_H
JUMP_O
JUMP case_1
 ...
JUMP case_n
```

3. On the branch of the case set up a block activation record and after completing the code deallocate the local storage and restore the frame pointer:

```
case_i : LOAD_R %fp
         ALLOC 2
         LOAD_R %sp
         STORE_R %fp
         ....                    % code for branch
         ALLOC -2                % remove void cells
         STORE %fp               % restore the frame pointer
         ALLOC -n                % remove the arguments
         JUMP return_case
```

4. Return to the common code and continue.

6 Dynamic arrays in M+

Notice the fact that functions are declared does not affect the storage allocation: they are compiled separately and the only thing we need is their label. However, array declarations are a different matter!

Consider the problem of allocating an array declared as:

```
a[10][20]:real;
```

The sequence is

1. Create an entry for the first dimension

   (function code `LOAD_I 10`).

2. Create the array pointer by storing the stack pointer into the activation record entry corresponding to `a`.

   (function code `LOAD_R %sp LOAD_R %fp STORE_O m_a`).

3. Create the next dimension entry for the array

   (function code `LOAD_I 20`).

4. Calculate the size of the array. In general this will not be known at compile time so it must be calculated. Here we shall load the two dimensions and multiply them. We need this number then for two reasons we need to add it to the deallocation counter and we need to allocate the given amount of stack space. Accordingly we reload it and the deallocation counter (and 2) and form the sum putting this back into the deallocation counter.

   (function code:

   ```
   LOAD_R %fp
   LOAD_O m_a
   LOAD_O 1
   LOAD_R %fp
   LOAD_O m_a
   LOAD_O 2
   APP MUL     % the storage required by array
   LOAD_R %fp
   LOAD_O m+1  % deallocation counter
   LOAD_I 2    % storage space for array bounds
   LOAD_R %fp
   LOAD_O a_I
   LOAD_O 3    % array size reloaded
   APP ADD
   APP ADD
   LOAD_R %fp
   STORE_O m+1 % store modified deallocation counter
   ```

).

5. Finally we allocate the required storage for the array.

   (function code `ALLOC_S`).

This has now to be repeated for each declared array. In general, there may be a computation which will replace the simple loading of the dimension here (step 1 and 3 here).