The ML Abstract Machine

The state of the Abstract Machine (AM) is determined by four quantities (together with their denotations):

**AP - Argument Pointer:** Pointer to the top of the Argument Stack (AS), where arguments are loaded to be passed to functions, and results of functions are delivered. This stack is also used to store local and temporary values.

**FP - Frame Pointer:** Pointer to the current closure consisting of the text of the currently executed program, and of an environment for the free variables of the program.

**SP - Stack Pointer:** Pointer to the top of the Return Stack (RS), where program counter and frame pointer are saved during function calls.

**PC - Program Counter:** Pointer to the next instruction to be executed inside the current program.
Here is a snapshot of the AM:

The AM assumes the existence of an infinite memory of cells of different sizes.
Typical cells are null, bool, num, toy, ref, pair, list, injection, closure and text cells. The exact format of these cells is unessential, as long as the primitive operations on these cells respect the expected properties.

The AM does not assume these cells to contain any information about their type.

Stacks, closures, texts and data contain pointers to cells, and this convention will be strictly observed in these notes. However, implementations may directly clone certain cells, instead of their pointers, on stacks etc; this usually happens for null, empty, bool and num cells, but should never happen for ref cells.
Data Operations

These are operations which transfer data back and forth between the Argument Stack and data cells. With the exception of destructives (see below) they take n arguments \((n \geq 0)\) from the top of AS (the first argument is the deepest one) popping AS \(n\) times. Then they push their result back on the top of AS.

Note that the type checking of the ML source program will guarantee against any misuse of AT operations.

The notation \(n\) means that "\(n\)" is a displacement from the top of AS where the top has displacement \(0\).
Null Operations

- **Null**
  
  ![Diagram of Null Operation]

- **NullEq**
  
  ![Diagram of NullEq Operation]

- **NullPoint**
**Boolean Operations**

- **True**
  - AP \(\rightarrow\) True \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) true

- **False**
  - AP \(\rightarrow\) False \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) false

- **Not**
  - AP \(\rightarrow\) bool \(\rightarrow\) Not \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) not

- **And**
  - AP \(\rightarrow\) bool \(\rightarrow\) bool \(\rightarrow\) And \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) anb

- **Or**
  - AP \(\rightarrow\) bool \(\rightarrow\) bool \(\rightarrow\) Or \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) avb

- **BoolEq**
  - AP \(\rightarrow\) bool \(\rightarrow\) bool \(\rightarrow\) BoolEq \(\rightarrow\) AP \(\rightarrow\) bool \(\rightarrow\) a=b

- **Bool Print**
Numeric Operations

All numbers are real numbers.

- Num n
  (push a new cell containing n on AS)

- Minus
  (Minus a = -a)

- Plus
  (Plus(a, b) = a+b)

- Diff
  (Diff(a, b) = a-b)

- Times
  (Times(a, b) = a*b)

- Divide
  (Divide(a, b) = a/b)

- IntDiv
  (IntDiv(a, b) = ⌊a/b⌋)

- Module
  (Module(a, b) = (a/b) - ⌊a/b⌋)

- Greater
  (Greater(a, b) = a>b)

- Less
  (Less(a, b) = a<b)

- GreatEq
  (GreatEq(a, b) = a>=b)

- LessEq
  (LessEq(a, b) = a<=b)

- NumEq
  (NumEq(a, b) = a=b)

- NumPoint
String Operations

- **Ton `...`** (push a `Ton` cell containing `...` on AS)
- **Conc**
  \[(Conc(`...`; `---`) = `...---`)\]
- **SubTon**
  \[(SubTon(K; E; `c_1...c_n`) = `c_k...c_{k+E-1}`)\]
- **TonLength**
  \[(TonLength(`c_1...c_n`) = n)\]
- **Explode**
  \[(Explode(`c_1...c_n`) = `[c_1; ...; c_n]`)\]
- **Impplode**
  \[(Impplode([`---`; ...; `---`]) = `---...---`)\]
- **Search**
  \[(Search(`p_0...p_m`, `...q_0...q_n`) = n)\]
  where \(p_0 = q_{n+1}\) (first occurrence from left)
  fails if no occurrence
- **Ton Eq**
  \[(TonEq(`p_0...p_m`, `q_0...q_n`) = (n=m) \land (p_i = q_i))\]
- **Ton Point**
- **Ton UnQPoint** (print Unquoted)
Reference Operations

- Fetch
  \[
  \text{Fetch}(z) = a \quad \text{where} \quad z = \text{ref}(a)
  \]

- Store
  \[
  \text{Store}(a, z) = (z := a ; a)
  \]

- RefEq
  \[
  \text{RefEq}(z_1, z_2) = (a_1 = a_2) \quad \text{where} \quad z_1 = \text{ref}(a_1) ; z_2 = \text{ref}(a_2)
  \]

- RefPrint
Pair Operations

- **Pair**
  \[ \text{Pair}(t, s) \text{ pushes a pair all } (t, s) \text{ on } AS \]

- **Fst**
  \[ \text{Fst}(p) = t \text{ when } p = t, s \]

- **Snd**
  \[ \text{Snd}(p) = s \text{ when } p = t, s \]

- **DestPair**

  No constraints on the order or coincidence of \( p, t, s \).
List Operations

- **Empty**: \( \text{push a list empty cell on } \text{AS} \)
- **Cons**: \( \text{Cons}(h, t) \) pushes a list cons cell \((h-t)\) on \( \text{AS} \)
- **Hd**: \( \text{Hd}(p) = h \) where \( p = h-t \)
- **Tl**: \( \text{Tl}(p) = t \) where \( p = h-t \)
- **Null**: \( \text{Null}(p) = p=[\ ] \)

- **Dest Empty**
  
  Fails if the list cell is not an empty cell.

- **Dest Cons**
  
  Fails if the list cell is not a cons cell.
  
  No constraints on the order or coincidence of \( p, h, t \).

- **Cons Point**
Disjunction Operations

- \text{Inl} (a) \text{ pushes a left injection cell } (a) \text{ on AS }
- \text{Inr} (a) \text{ pushes a right injection cell } (a) \text{ on AS }
- \text{Outl} (j) = a \text{ where } j = \text{inl}(a)
- \text{Outr} (j) = a \text{ where } j = \text{inr}(a)
- \text{Isl} (i) = (j = \text{inl}(a)) \text{ for some } a
- \text{Isr} (i) = (j = \text{inr}(a)) \text{ for some } a

- \text{DisjPoint}
ML Run-Time Data Structures

**Pair**

```
Fst  Snd
```

**Cons**

```
Hd   Tl
```

**Injection**

```
Out  Tag
```

**Reference**

```
At
```

**Token**

```
Size
```

```c
ch(1)  ch(n)
```
Empty

True

False

Nil

Integer

-32768 .. 32767

Number

(Contains a Floating Point)

Note: Empty, True, False, Nil and Integers are unboxed values. They are contained directly on the stack or inside data structures. When an unboxed value (occupying a word) is contained in the least significative part of a long word (e.g., in a pair) the other word must be zero. Pointers can be distinguished from unboxed values because the most significative word of a pointer cannot be zero.
The pointer "\( A \)" is used when passing Texts around by themselves (it is convenient for garbage collection).

The pointer "\( A \)" is used when referring to a Text from a closure (it is convenient for function application).

"Size" fields are placed before the arrow "\( A \)" because they are only used during garbage collection.
Array

\[ u_b + 1 \geq 16 \]

Table

\[ \text{Size} \]

\[ n \]

\[ \text{Ass}(a) \]

\[ n \geq 0 \]

\[ \text{Ass}(a) \]
Abstract Syntax Operations

- Parse (read a string and push a parse tree on AS)

- make-decl-is primitives for each syntactic clause
Text Operations

- Eval

- Text
Stack Operations

- GetLocal

- GetGlobal

- Pop

- Squeeze

- Rise
Control Operations

These are operations affecting the Program Counter and the Stack Pointer.

Jump Operations

Jump displacements are expressed in number of skipped instructions. Positive displacements are jumps forward.

- Jump n (jump leaving AS unchanged)
- True Jump n (pop AS and jump if the top was "true")
- False Jump n (pop AS and jump if the top was "false")
Call Operations

- Save Frame

- Rest Frame

- Appl Frame

Note: The complete sequence is "SaveFrame; ApplFrame; RestFrame".
A is the callee and B is the called closure.
• **Return**

![Diagram of Return](image)

- Return(n)  n ≥ 0

• **TailApply**

![Diagram of TailApply](image)

- TailApply

**Notes:** "TailApply" is equivalent to "SaveFrame; ApplFrame; PostFrame; Return"
Fail Operations

A Trap Frame consists of five fields: the first one is a datum of type "+ trap list" (the trap list) and the other four ones contain PC, SP, FP and AP. Trap jump displacements are expressed in number of skipped instructions. A typical compilation is:


Trap

Also:

"A ?? "..." B"  \rightarrow  ""..."" [TrapList L1] "A" [UnTrap L2] [L1: Pop 1] "B" [L2:

"A ? \times, B"  \rightarrow  [Trap L1] "A" [UnTrap L2] [L1:] "B" [L2: Squeeze 1]
- Trap List

```
AP  \rightarrow  tor list  SP  \rightarrow  FP  \rightarrow  closure  Text  \rightarrow  PC  \\
\downarrow

TrapList n

\rightarrow

FP  \rightarrow  closure  Text  \rightarrow  PC  \\
\rightarrow  \rightarrow  \rightarrow  \rightarrow  \rightarrow

\rightarrow

tor list
```

- Trap List
Notice: If execution reaches an UnTrap, then there have been no (untrapped) failures since the corresponding Trap; hence FP and SP are the same, AP has grown by 1 and PC₀ has almost reached the PC stored in the Trap frame (the difference being the UnTrap instruction itself). It is then enough to remove the Trap frame and skip the failure treatment, jumping to PC₀+1+n.
• FailWith

(m) If FailList = int(1) or FailList = int(TouList) where Tou occurs in TouList.
If no such TrapFrame is found scanning the Trap Stack from top to bottom, a top-level failure is generated.

Note: PCₐ points to the failure recovery text.