Script **generated by TTT**

Title: Seidl: GAD (04.05.2015)

Date: Mon May 04 10:31:59 CEST 2015

Duration: 75:30 min

Pages: 16

```
_fac:  enter q     loadc 1     A:  loadr -3          mul
       alloc 0     storer -3       loadr -3          storer -3
       loadr -3    return          loadc 1           return
       loadc 0     jump B          sub           B:  return
       leq                         mark
       jumpz A                     loadc _fac
                                   call
                                   slide 0
```

where $\rho_{\text{fac}} : x \mapsto (L, -3)$ and $q = 5$.

---

10 Translation of Whole Programs

Before program execution, we have:

$SP = -1$ $FP = EP = -1$ $PC = 0$ $NP = MAX$

Let $p \equiv V\_defs\ \ F\_def_1 \ldots F\_def_n$, denote a program where F_def_i is the definition of a function f_i of which one is called main .

The code for the program p consists of:

- code for the function definitions F_def_i;
- code for the allocation of global variables;
- code for the call of int main();
- the instruction halt which returns control to the operating system together with the value at address 0.

Then we define:

```
code p ∅  =  0: enter (k + 4)
                alloc (k + 1)
                mark
                loadc _main
                call
                slide k
                halt
           _f1: code F_def1 ρ
                  ⋮
           _fn: code F_defn ρ
```

SP == -1

EP = k+4 -1 = k+3

SP = 2

where $\emptyset \mathrel{\widehat{=}}$ empty address environment;
$\rho \mathrel{\widehat{=}}$ global address environment;
$k \mathrel{\widehat{=}}$ size of the global variables

11 The language PuF

We only regard a mini-language PuF (“Pure Functions”).

We do not treat, as yet:

- Side effects;
- Data structures;
- Exceptions.

The Translation of Functional Programming Languages

11 The language PuF

We only regard a mini-language PuF (“Pure Functions”).

We do not treat, as yet:

- Side effects;
- Data structures;
- Exceptions.

A program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \\
 & \mid & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
 & \mid & (e'\ e_0 \ldots e_{k-1}) \\
 & \mid & (\textbf{fun}\ x_0 \ldots\ x_{k-1} \to e) \\
 & \mid & (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0) \\
 & \mid & (\textbf{let rec}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_n = e_n\ \textbf{in}\ e_0)
\end{array}
$$

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a **let**-expression, i.e. an expression with locally defined variables, or
- a **let-rec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` as basic type.

Example

The following well-known function computes the factorial of a natural number:

$$\begin{array}{lcl} \textbf{let rec } \text{fac} & = & \textbf{fun } x \to \textbf{if } x \leq 1 \textbf{ then } 1 \\ & & \textbf{else } x \cdot \text{fac } (x-1) \\ \textbf{in } \text{fac } 7 & & \end{array}$$

As usual, we only use the minimal amount of parentheses.

There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

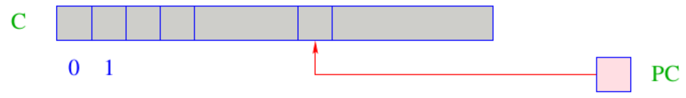
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

"need"

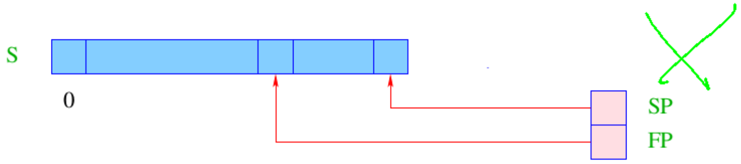
---

12 Architecture of the MaMa:

We know already the following components:

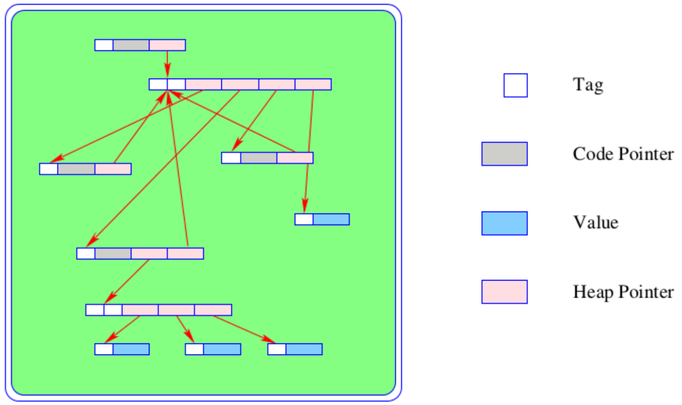


C = Code-store – contains the MaMa-program;
each cell contains one instruction;
PC = Program Counter – points to the instruction to be executed next;

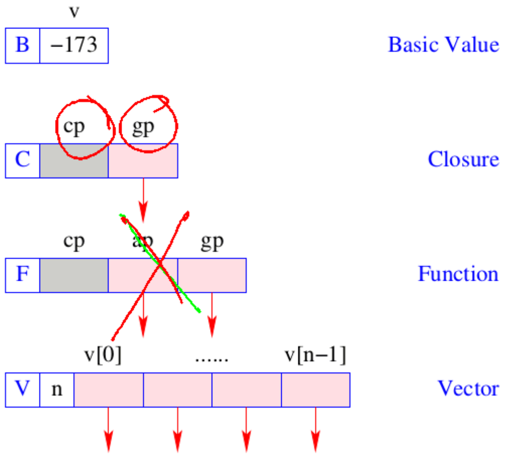


S = Runtime-Stack – each cell can hold a basic value or an address;
SP = Stack-Pointer – points to the topmost occupied cell;
as in the CMa implicitely represented;
FP = Frame-Pointer – points to the actual stack frame.

We also need a heap H:

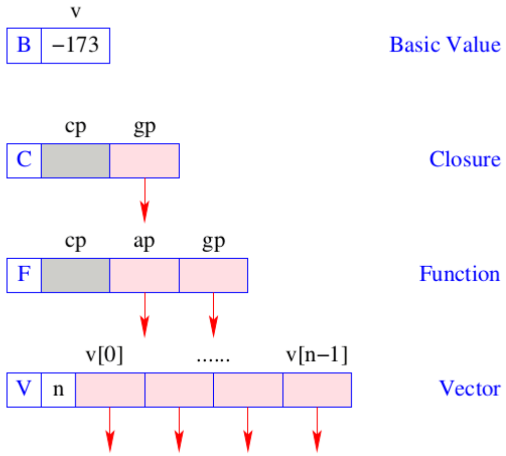


... it can be thought of as an abstract data type, being capable of holding data objects of the following form:



f( )   f : int → int

... it can be thought of as an abstract data type, being capable of holding data objects of the following form:



1+2

The instruction new $(tag, args)$ creates a corresponding object (B, C, F, V) in H and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- $\text{code}_V\ e$ — (generates code that) computes the Value of e, stores it in the heap and returns a reference to it on top of the stack (the normal case);
- $\text{code}_B\ e$ — computes the value of e, and returns it on the top of the stack (only for Basic types);
- $\text{code}_C\ e$ — does not evaluate e, but stores a Closure of e in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:

The instruction new $(tag, args)$ creates a corresponding object (B, C, F, V) in H and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- $\text{code}_V\ e$ — (generates code that) computes the Value of e, stores it in the heap and returns a reference to it on top of the stack (the normal case);
- $\text{code}_B\ e$ — computes the value of e, and returns it on the top of the stack (only for Basic types);
- $\text{code}_C\ e$ — does not evaluate e, but stores a Closure of e in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:

13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

$$\begin{array}{lcl}
\text{code}_B\ b\ \rho\ \text{sd} & = & \text{loadc b} \\
\text{code}_B\ (\Box_1\ e)\ \rho\ \text{sd} & = & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & \text{op}_1 \\
\text{code}_B\ (e_1\ \Box_2\ e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & \text{op}_2
\end{array}$$