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.

A Program is an expression e of the form:

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

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 **letrec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` and `bool` as basic types.

Example:

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

$$\begin{array}{lcl} \textbf{letrec}\ \text{fac} & = & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\ \textbf{in}\ \text{fac}\ 7 & & \end{array}$$

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

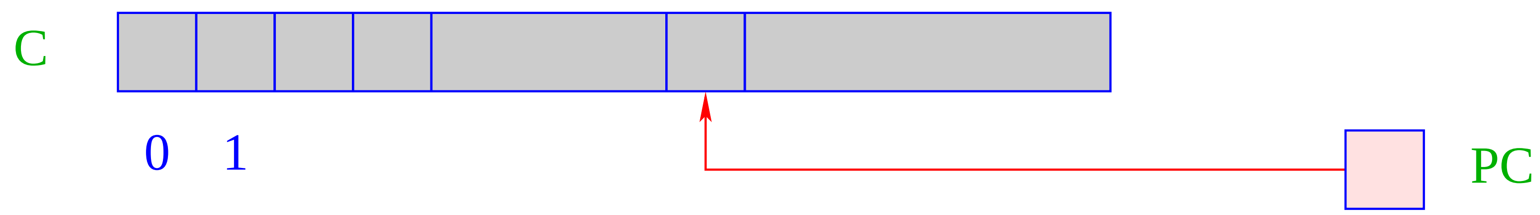
There are two Semantics:

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

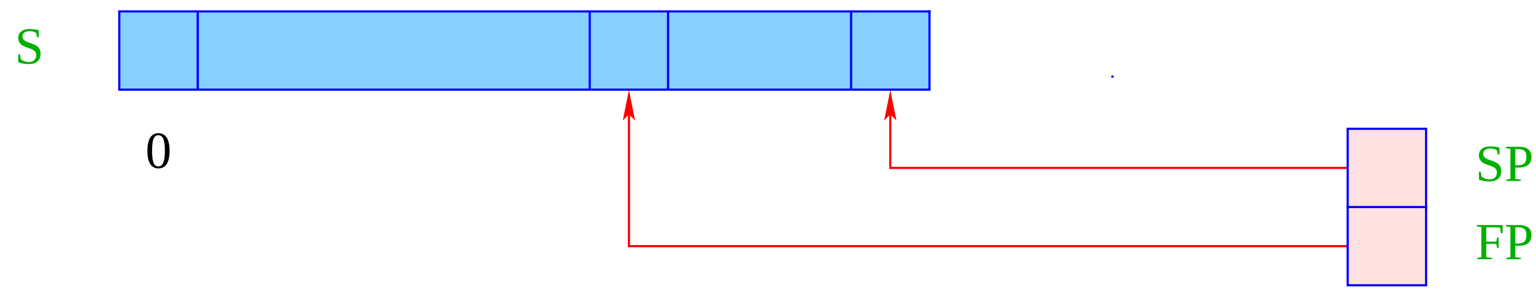
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

12 Architecture of the MaMa:

We know already the following components:

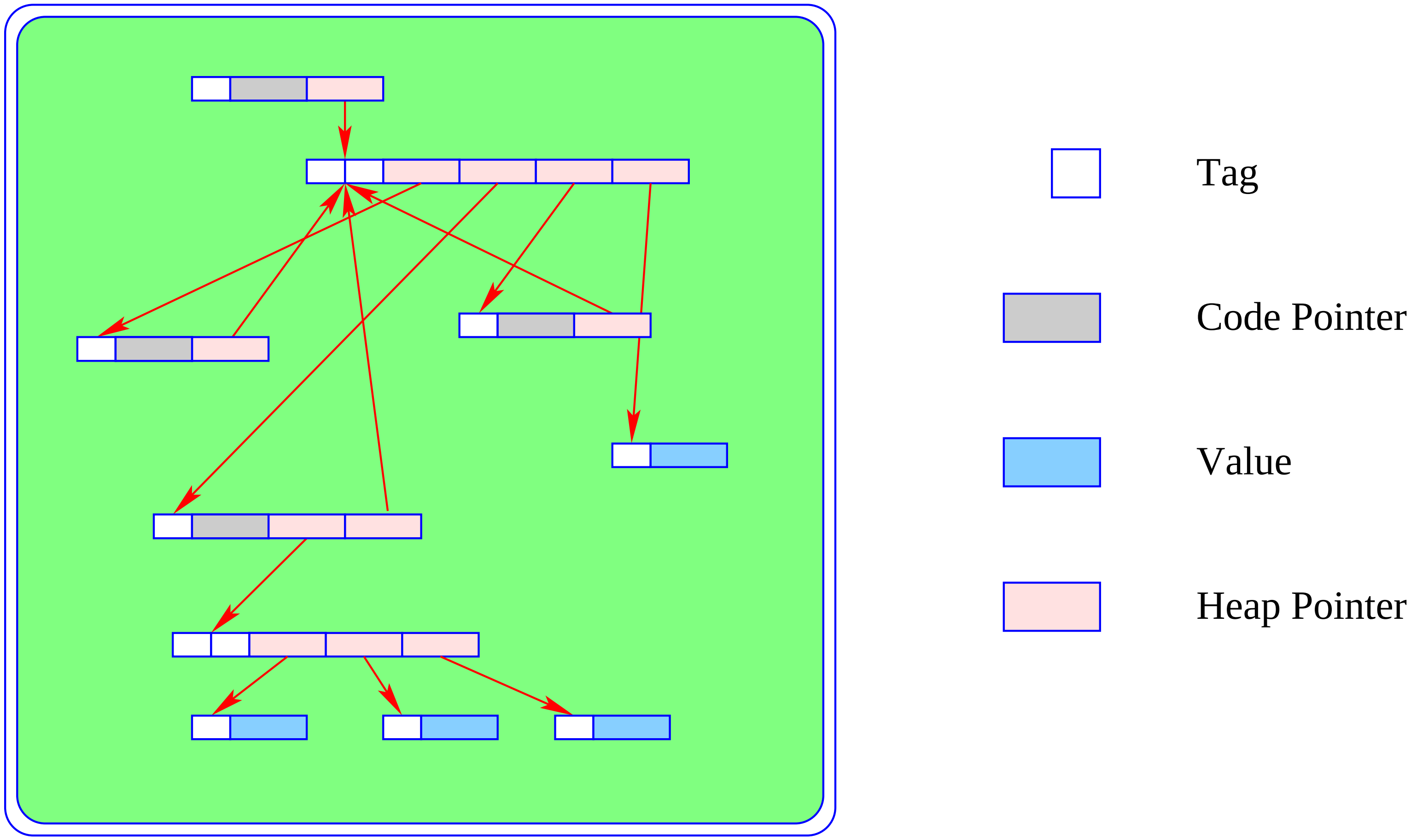


C	=	Code-store – contains the MaMa-program;
		each cell contains one instruction;
PC	=	Program Counter – points to the instruction to be executed next;

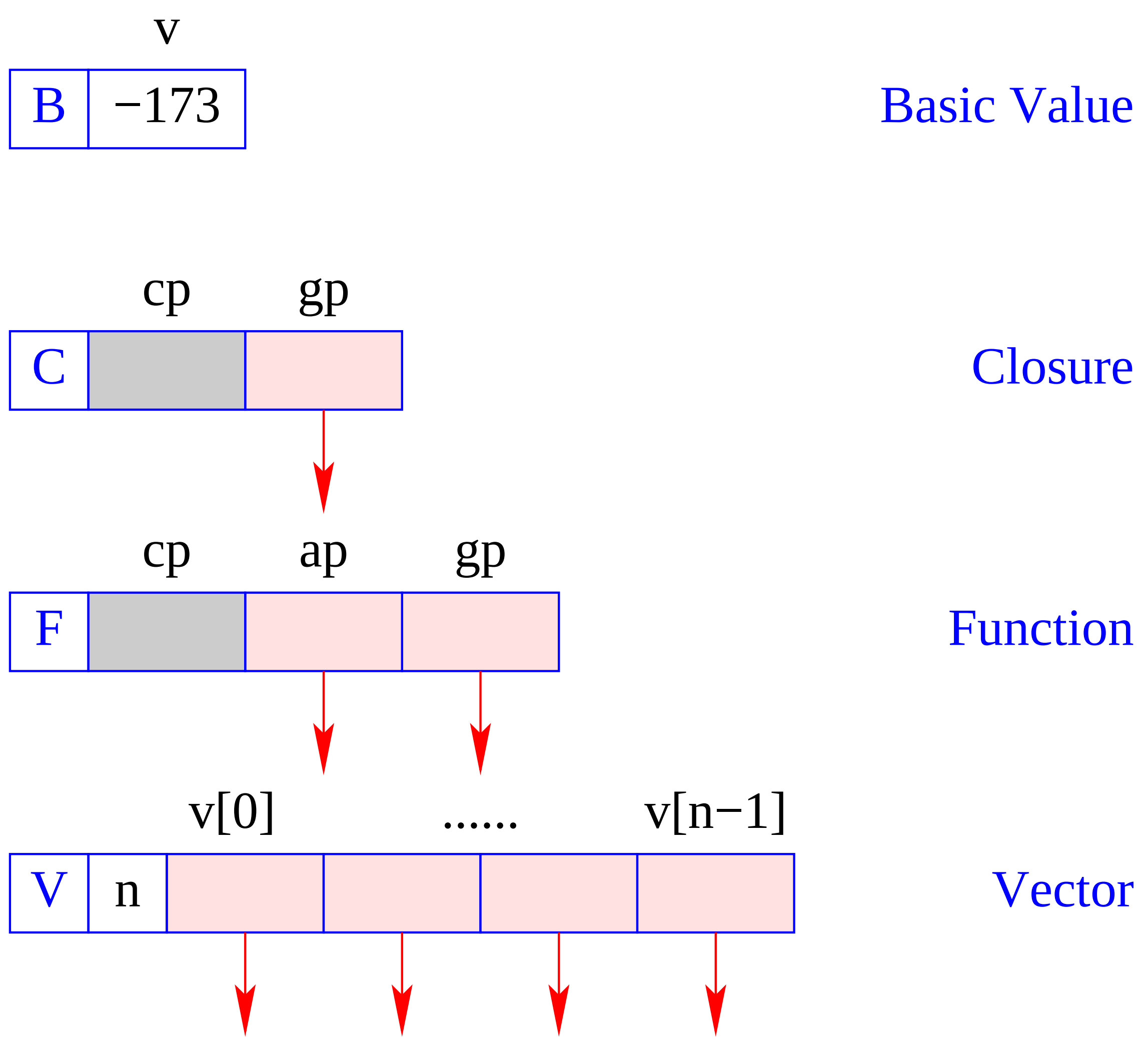


S	=	Runtime-Stack – each cell can hold a basic value or an address;
SP	=	Stack-Pointer – points to the topmost occupied cell;
		as in the CMa implicitely represented;
FP	=	Frame-Pointer – points to the actual stack frame.

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:



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:

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

$$
\begin{array}{lcll}
\mathrm{code}_B\,(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \mathrm{sd} & = & & \mathrm{code}_B\ e_0\ \rho\ \mathrm{sd} \\
 & & & \mathrm{jumpz\ A} \\
 & & & \mathrm{code}_B\ e_1\ \rho\ \mathrm{sd} \\
 & & & \mathrm{jump\ B} \\
 & & \mathrm{A:} & \mathrm{code}_B\ e_2\ \rho\ \mathrm{sd} \\
 & & \mathrm{B:} & \ldots
\end{array}
$$

Note:

- ρ denotes the actual address environment, in which the expression is translated. Address environments have the form:

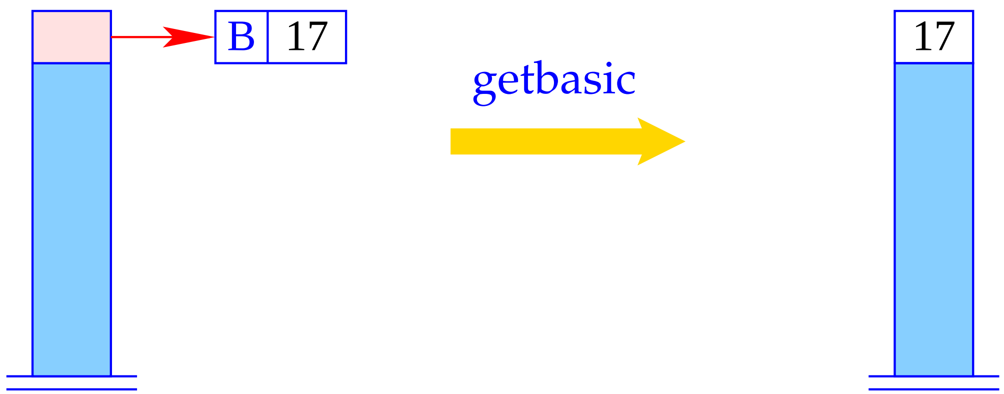
$$\rho : \mathit{Vars} \rightarrow \{L, G\} \times \mathbb{Z}$$

- The extra argument sd, the stack difference, *simulates* the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.

- The instructions op_1 and op_2 implement the operators $\Box_1$ and $\Box_2$, in the same way as the the operators neg and add implement negation resp. addition in the CMa.

- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

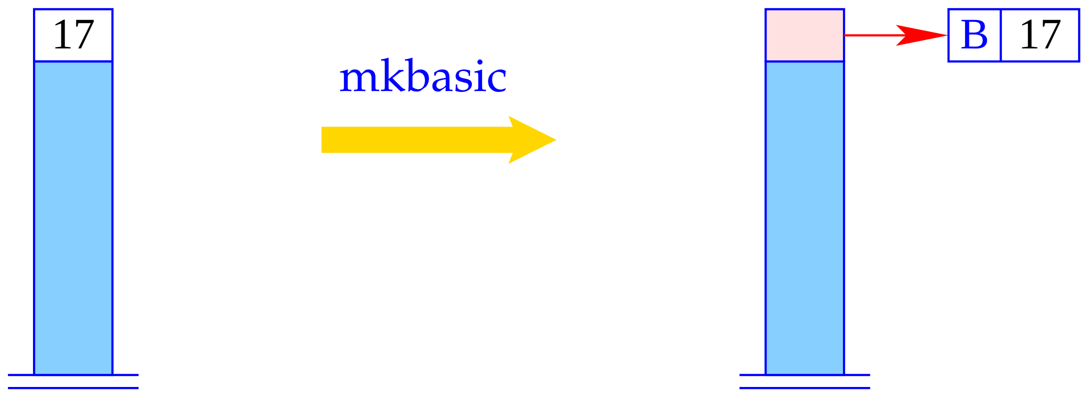
$$\begin{array}{rcl} \text{code}_B\, e\, \rho\, \text{sd} & = & \text{code}_V\, e\, \rho\, \text{sd} \\ & & \text{getbasic} \end{array}$$




```
if (H[S[SP]] != (B,_))
      Error "not basic!";
else
      S[SP] = H[S[SP]].v;
```

For code_V and simple expressions, we define analogously:

$$
\begin{array}{lcll}
\text{code}_V\; b\; \rho\; \text{sd} & = & & \text{loadc b; mkbasic} \\[1ex]
\text{code}_V\; (\Box_1\; e)\; \rho\; \text{sd} & = & & \text{code}_B\; e\; \rho\; \text{sd} \\
 & & & \text{op}_1\text{; mkbasic} \\[1ex]
\text{code}_V\; (e_1\; \Box_2\; e_2)\; \rho\; \text{sd} & = & & \text{code}_B\; e_1\; \rho\; \text{sd} \\
 & & & \text{code}_B\; e_2\; \rho\; (\text{sd} + 1) \\
 & & & \text{op}_2\text{; mkbasic} \\[1ex]
\text{code}_V\; (\textbf{if}\; e_0\; \textbf{then}\; e_1\; \textbf{else}\; e_2)\; \rho\; \text{sd} & = & & \text{code}_B\; e_0\; \rho\; \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_V\; e_1\; \rho\; \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_V\; e_2\; \rho\; \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$



S[SP] = new (B,S[SP]);

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

$$\begin{array}{lll} \textbf{let} & c = 5 & \\ & f = \textbf{fn}\ a & \Rightarrow \quad \textbf{let}\ b = a * a \\ & & \quad \textbf{in}\ b + c \\ \textbf{in} & f\ c & \end{array}$$

The function f uses the global variable c and the local variables a (as formal parameter) and b (introduced by the inner **let**).

The binding of a global variable is determined, when the function is constructed (static scoping!), and later only looked up.

Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the gp-component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In constrast, local variables should be administered on the stack ...

$\Longrightarrow$ General form of the address environment:

$$\rho : Vars \to \{L, G\} \times \mathbb{Z}$$

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

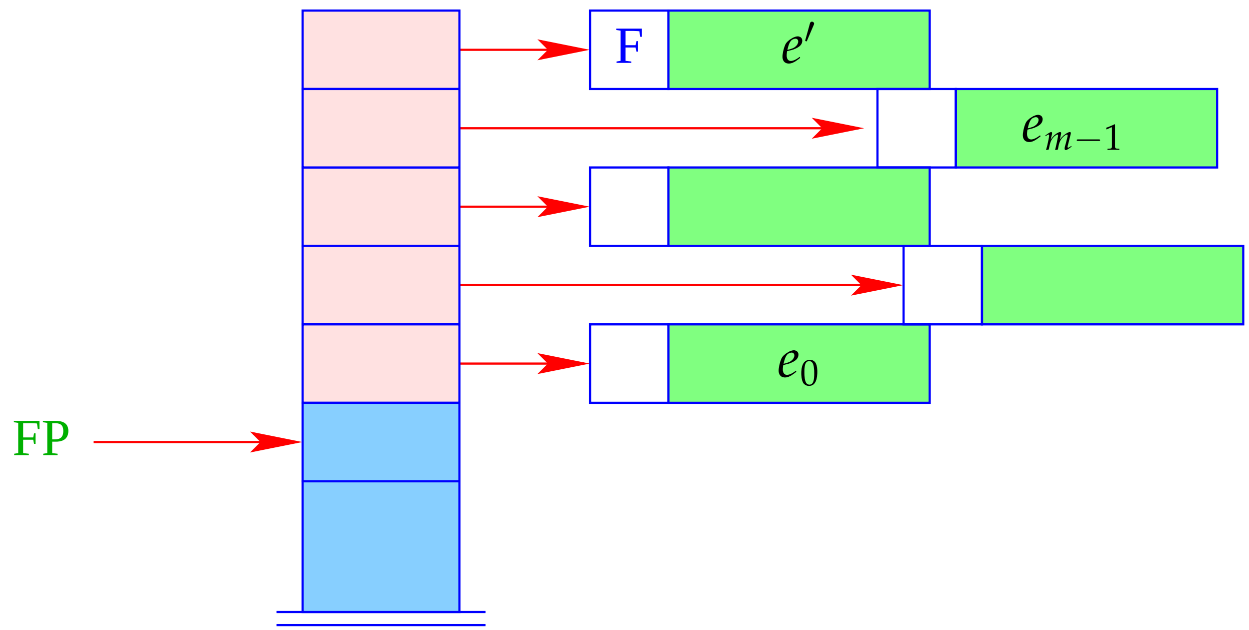
Let $e \equiv e' \; e_0 \; \ldots \; e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

The arity of e' does not need to be m :-)

- PuF functions have curried types, $f : t_1 \to t_2 \to \ldots \to t_n \to t$
- f may therefore receive less than n arguments (under supply);
- f may also receive more than n arguments, if t is a functional type (over supply).

Possible stack organisations:



$+$ Addressing of the arguments can be done relative to FP

$-$ The local variables of e' cannot be addressed relative to FP.

$-$ If e' is an n-ary function with $n < m$, i.e., we have an over-supplied function application, the remaining $m - n$ arguments will have to be shifted.

– If e' evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath e_0: