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:

$$
ex := b | x | (\square_1 e) | (e_1 \square_2 e_2) | (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) | (e' e_0 \ldots e_{k-1}) | (\text{fn } x_0, \ldots, x_{k-1} \Rightarrow e) | (\text{let } x_1 = e_1; \ldots; x_n = e_n \text{ in } e_0) | (\text{letrec } x_1 = e_1; \ldots; x_n = e_n \text{ in } e_0)$$

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 \textit{locally defined variables}, or

- a letrec-expression, i.e. an expression with \textit{simultaneously defined} local variables.

For simplicity, we only allow \texttt{int} and \texttt{bool} as basic types.
Example:

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

\[
\text{letrec}\quad \text{fac} = \begin{array}{ll}
\text{fn} \ x & \Rightarrow \begin{array}{ll}
\text{if} \ x \leq 1 & \text{then} \ 1 \\
\text{else} & x \cdot \text{fac}(x - 1)
\end{array} \\
\end{array}
\text{in} \ \text{fac} \ 7
\]

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:

\[
\begin{align*}
\text{C} & = \text{Code-store} \quad \text{– contains the MaMa-program;}
\text{C} & \quad \text{each cell contains one instruction;}
\text{PC} & = \text{Program Counter} \quad \text{– points to the instruction to be executed next;}
\end{align*}
\]
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 implicitly represented;
FP = Frame-Pointer – points to the actual stack frame.
We also need a heap $H$: 

![Heap Diagram]

- **Tag**
- **Code Pointer**
- **Value**
- **Heap Pointer**
... it can be thought of as an abstract data type, being capable of holding data objects of the following form:

```
\begin{align*}
v & \begin{array}{|c|}
\multicolumn{1}{|c|}{B} \end{array} \text{ Basic Value} \\
\multicolumn{1}{|c|}{cp} \quad \multicolumn{1}{|c|}{gp} & \text{ Closure} \\
\multicolumn{1}{|c|}{cp} \quad \multicolumn{1}{|c|}{ap} \quad \multicolumn{1}{|c|}{gp} & \text{ Function} \\
\multicolumn{1}{|c|}{v[0]} \quad \ldots \quad \multicolumn{1}{|c|}{v[n-1]} & \text{ Vector} \\
\multicolumn{1}{|c|}{V} \quad \multicolumn{1}{|c|}{n} & 
\end{align*}
```
The instruction \texttt{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 $e$:

- \texttt{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);
- \texttt{code}_B \; e \; — \; computes the value of $e$, and returns it on the top of the stack (only for Basic types);
- \texttt{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:

\[
\begin{align*}
\text{code}_B b \rho \text{ sd} &= \text{loadc } b \\
\text{code}_B (\Diamond_1 e) \rho \text{ sd} &= \text{code}_B e \rho \text{ sd} \\
&\quad \text{op}_1 \\
\text{code}_B (e_1 \Diamond_2 e_2) \rho \text{ sd} &= \text{code}_B e_1 \rho \text{ sd} \\
&\quad \text{code}_B e_2 \rho (\text{sd} + 1) \\
&\quad \text{op}_2
\end{align*}
\]
\[ \text{code}_B (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \, \rho \, \text{sd} = \text{code}_B e_0 \, \rho \, \text{sd} \]
\[ \text{jumpz A} \]
\[ \text{code}_B e_1 \, \rho \, \text{sd} \]
\[ \text{jump B} \]
\[ \text{A: code}_B e_2 \, \rho \, \text{sd} \]
\[ \text{B: ...} \]
Note:

- \( \rho \) denotes the actual address environment, in which the expression is translated.

- 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 \( \text{neg} \) and \( \text{add} \) implement negation resp. addition in the \( \text{CMa} \).

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

\[
\text{code}_B e \rho sd = \text{code}_V e \rho sd \\
\text{getbasic}
\]
if (H[S[SP]] != (B, _))
    Error “not basic!”;
else
    S[SP] = H[S[SP]].v;
For $\text{code}_V$ and simple expressions, we define analogously:

\[
\begin{align*}
\text{code}_V b \rho \text{ sd} & = \text{loadc } b; \text{ mkbasic} \\
\text{code}_V (\Box_1 e) \rho \text{ sd} & = \text{code}_B e \rho \text{ sd} \text{ op}_1; \text{ mkbasic} \\
\text{code}_V (e_1 \Box_2 e_2) \rho \text{ sd} & = \text{code}_B e_1 \rho \text{ sd} \text{ code}_B e_2 \rho (\text{sd} + 1) \text{ op}_2; \text{ mkbasic} \\
\text{code}_V (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \text{ sd} & = \text{code}_B e_0 \rho \text{ sd} \text{ jumpz A} \\
& \quad \text{ code}_V e_1 \rho \text{ sd} \text{ jump B} \\
& \quad \text{ A: code}_V e_2 \rho \text{ sd} \\
& \quad \text{ B: ...}
\end{align*}
\]
S[SP] = new (B,S[SP]);
14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function $f$:

$$
\text{let } c = 5 \\
\quad f = \text{fn } a \Rightarrow \text{let } b = a \ast a \\
\quad \quad \text{in } b + c \\
\text{in } f c
$$

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 contrast, local variables should be administered on the stack ...

[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 $m$ :-)
- $f$ 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.