9.3 Calling/Entering and Leaving Functions

Be \( f \) the actual function, the \texttt{Caller}, and let \( f \) call the function \( g \), the \texttt{Callee}.

The code for a function call has to be distributed among the \texttt{Caller} and the \texttt{Callee}.

The distribution depends on \texttt{who} has \texttt{which} information.

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

- the \texttt{FP}
- the \texttt{continuation address} after the call and
- the actual \texttt{EP}.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

- Generate code for the body!
- Generate code for calls!
Altogether we generate for a call:

\[
\begin{align*}
\text{code}_{\rho} g(e_1, \ldots, e_n) & = \text{mark} \\
\text{code}_{\rho} e_1 & \\
\vdots & \\
\text{code}_{\rho} e_n & \\
\text{call} & \\
\end{align*}
\]

where \( n \) = space for the actual parameters

**Note:**
- Expressions occurring as actual parameters will be evaluated to their R-value → Call-by-Value parameter passing.
- Function \( g \) can also be an expression, whose R-value is the start address of the function to be called ...

- Function names are regarded as constant pointers to functions, similarly to declared arrays. The R-value of such a pointer is the start address of the function.
- For a variable \( \text{int} (\ast) g_1 \), the two calls
  \[
  (*g_1)()\quad \text{und} \quad g()
  \]
  are equivalent :)  
  Normalization: Dereferencing of a function pointer is ignored.
- Structures are copied when they are passed as parameters.

In consequence:

\[
\begin{align*}
\text{code}_{\rho} f & = \text{loadc}(\rho f) & f \text{ a function name} \\
\text{code}_{\rho} (se) & = \text{codec} e & e \text{ a function pointer} \\
\text{code}_{\rho} e & = \text{codec} e \quad \text{move } k & e \text{ a structure of size } k
\end{align*}
\]

The instruction `mark` allocates space for the return value and for the organizational cells and saves the FP and EP.

\[
\begin{align*}
\text{for } (i = k-1; i \geq 0; i--) \\
S[SP + i] & = S[S[SP] + i]; \\
SP & = SP + k - 1;
\end{align*}
\]

\[
\begin{align*}
S[SP + 2] & = \text{EP}; \\
S[SP + 3] & = \text{FP}; \\
SP & = SP + 4;
\end{align*}
\]
The instruction call n saves the continuation address and assigns FP, SP, and PC their new values.

\[ \text{FP} = \text{SP} - n - 1; \]
\[ \text{S}[\text{FP}] = \text{PC}; \]
\[ \text{PC} = \text{S}[\text{SP}]; \]
\[ \text{SP}--; \]

Correspondingly, we translate a function definition:

\[ \text{code } t \cdot f (\text{spec}) \{ V, \text{defs} \} \cdot \rho = \]
\[ \text{\_enter } \text{\_} \quad \text{// Setting the EP} \]
\[ \text{\_alloc } k \quad \text{// Allocating the local variables} \]
\[ \text{Code } ss \text{ pe} \quad \text{// leaving the function} \]

where
\[ t = \text{return type of } f \text{ with } |t| \leq 1 \]
\[ q = \text{max} S + k \text{ where} \]
\[ \text{max} S = \text{maximal depth of the local stack} \]
\[ k = \text{space for the local variables} \]
\[ \text{pe} = \text{address environment for } f \]
\[ \text{// takes care of spec, } V, \text{defs} \text{ and } \rho \]

The instruction enter q sets EP to its new value. Program execution is terminated if not enough space is available.

\[ \text{EP} = \text{SP} + q; \]

if (EP ≥ NP)

Error ("Stack Overflow");

The instruction return pops the actual stack frame, i.e., it restores the registers PC, EP, SP, and FP and leaves the return value on top of the stack.
9.4 Access to Variables and Formal Parameters, and Return of Values

Local variables and formal parameters are addressed relative to the current FP. We therefore modify $\text{code}_\psi$ for the case of variable names.

For $\rho x = (\text{tag}, j)$ we define

$$\text{code}_\psi \rho = \begin{cases} \text{load} j & \text{tag} = G \\ \text{loadrc} j & \text{tag} = L. \end{cases}$$

As an optimization one introduces the instructions $\text{loadr} j$ and $\text{storer} j$.

This is analogous to $\text{load} j$ and $\text{store} j$.

$$\text{loadr} j = \text{loadrc} j$$

$$\text{storer} j = \text{loadrc} j$$

The code for $\text{return } e_j$ corresponds to an assignment to a variable with relative address $-3$.

$$\text{code return } e_j \rho = \begin{cases} \text{code} e \rho & \text{for } M = -3 \\ \text{store} -3 \\ \text{return} \end{cases}$$

The instruction $\text{loadrc} j$ computes the sum of FP and $j$.

$$\begin{array}{c}
\text{FP} \\
\text{SP} \\
\text{FP} + j
\end{array}$$

Example: For the function

$$\text{int fac(int } x) \{$$

if ($x \leq 0$) return 1;
else return ($x \times \text{fac } (x - 1)$);

we generate:

$$\begin{array}{c}
\text{fac: enter q} & \text{loadc } 1 & A: \text{loadr } 1 & \text{mul} \\
\text{alloc } 0 & \text{store } -3 & \text{mark} & \text{store } -3 \\
\text{loadr } 1 & \text{return} & \text{loadr } 1 & \text{return} \\
\text{loadc } 0 & \text{jump } B & \text{loadc } 1 & \text{B: return} \\
\text{cgp} & \text{sub} & \text{sub} & \\
\text{jumpz } A & \text{call } 1 & \text{call } 1
\end{array}$$

where $\rho_{\text{fac}} : x \mapsto (L, 1)$ and $q = 1 + 4 + 2 = 7$. 

$$\text{sum } e_1, e_2 = \max (\text{max } e_1, e_2, 1 + \text{max } e_1)$$
As an optimization one introduces the instructions \texttt{loadr} and \texttt{storer}.
This is analogous to \texttt{loada} and \texttt{storea}.

\[
\text{loadr}\ j = \text{loadrc}\ j \\
\text{store}\ j = \text{loadrc}\ j
\]

The code for \texttt{return e} corresponds to an assignment to a variable with relative address $-3$.

\[
\text{code return } e; \rho = \text{code}_0 e; \rho \\
\text{store } -3 \\
\text{return}
\]

\section{Translation of Whole Programs}

The state before program execution starts:

\[
\text{SP} = -1 \quad \text{FP} = \text{EP} = 0 \quad \text{PC} = 0 \quad \text{NP} = \text{MAX} + 1
\]

Be $p \equiv V_{\text{defs}} F_{\text{def}_1} \ldots F_{\text{def}_n}$ a program, where $F_{\text{def}_i}$ defines a function $f_i$, of which one is named \texttt{main}.

The code for the program $p$ consists of:

- Code for the function definitions $F_{\text{def}_i}$;
- Code for allocating the global variables;
- Code for the call of \texttt{main}();
- the instruction \texttt{halt}.

\section{Access to Variables and Formal Parameters, and Return of Values}

Local variables and formal parameters are addressed relative to the current FP.

We therefore modify \texttt{code}, for the case of variable names.

For $\rho x = (\text{tag}, j)$

\[
\text{code}_1 x; \rho = \begin{cases} 
\text{loadc}\ j; \rho & \text{tag} = G \\
\text{loadrc}\ j; \rho & \text{tag} = L 
\end{cases}
\]

We thus define:

\[
\text{code } p; \emptyset = \begin{array}{ll}
\text{enter } (k + 6) & \\
\text{alloc } (k + 1) & \\
\text{mark} & \\
\text{loadc}_{\text{main}} & \\
\text{call } 0 & \\
\text{pop} & \\
\text{halt} & \\
\end{array}
\]

\[
\text{code } F_{\text{def}_1}; \rho = \ldots \\
\text{code } F_{\text{def}_n}; \rho
\]

where $\emptyset$ $\equiv$ empty address environment;
$\rho$ $\equiv$ global address environment;
$k$ $\equiv$ space for global variables
\texttt{main} $\equiv \{ f_1, \ldots, f_n \}$
The language PuF

We only regard a mini-language PuF ("Pure Functions"). We do not treat, as yet:

- Side effects;
- Data structures.

11 The language PuF

A Program is an expression &e of the form:

\[
\begin{align*}
& e ::= b \mid x \mid \langle \oplus \rangle \mid (\epsilon_1 \| \epsilon_2) \\
& \quad | \text{(if } \epsilon_3 \text{ then } \epsilon_1 \text{ else } \epsilon_2) \rangle \\
& \quad | (\epsilon' \epsilon_3 \ldots \epsilon_{n-1}) \\
& \quad | (\text{let } x_0, \ldots, x_{n-1} \Rightarrow e) \\
& \quad | (\text{letrec } x_1 = \epsilon_1; \ldots; x_n = \epsilon_0 \text{ in } e) \\
& \quad | (\text{letrec } x_1 = \epsilon_1; \ldots; x_n = \epsilon_0 \text{ in } e) \\
\end{align*}
\]

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 basic type.

Example:

\[
\begin{align*}
& \text{let } \text{fac } = \text{fn } n \Rightarrow n = \text{if } n \geq 6 \text{ then } 6 \text{ else } n \times \text{fac } (n - 1) \\
& \text{fac } 6 = 720
\end{align*}
\]

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

\[
\begin{align*}
& \text{letrec } \text{fac } = \text{fn } x \Rightarrow \text{if } x \leq 1 \text{ then } 1 \\
& \quad \text{else } x \times \text{fac } (x - 1) \\
& \text{fac } 7 = 5040
\end{align*}
\]

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):
  \[
  (\text{fn } x y f = f \Rightarrow (x + f) \times (y + f)) (\text{fn } x y = x)(\text{fn } x y = y)
  \]

- **CBN**: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell):
  \[
  \begin{align*}
  & \text{let } \text{fac } = \text{fn } x y = x (\text{fn } f \Rightarrow \text{fn } x = x (\text{fn } x y = f)(\text{fn } x y = y)) \\
  & \text{fac } (\text{fn } f \Rightarrow f) (\text{fn } x y = x) (\text{fn } x y = y) = f
  \end{align*}
  \]
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;