16 Function Definitions

The definition of a function $f$ requires code that allocates a functional value for $f$ in the heap. This happens in the following steps:

- Creation of a Global Vector with the binding of the free variables;
- Creation of an (initially empty) argument vector;
- Creation of an F-Object, containing references to theses vectors and the start address of the code for the body;

Separately, code for the body has to be generated.

Thus:

\[
\text{codey } \{\text{fun } x_0 \ldots x_{k-1} \rightarrow e\} \rho, sd = \begin{align*}
&\text{getvar } z_0, sd \\
&\text{getvar } z_1, \rho, sd + 1 \\
&\ldots \\
&\text{getvar } z_{k-1}, \rho, sd + g - 1 \\
&\text{mkvec } g \\
&\text{mkfunval } A \\
&\text{jump } B \\
A : \text{targ } k \\
&\text{codey } e \rho', 0 \\
&\text{return } k \\
B : \ldots
\end{align*}
\]

where $\{z_0, \ldots, z_{k-1}\} = \text{free}(\text{fun } x_0 \ldots x_{k-1} \rightarrow e)$

and $\rho' = [x_i \mapsto (L, i) \mid i = 0, \ldots, k - 1] \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g - 1\}$
codev \( (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \rho \sigma d \) =
getvar \( z_0 \rho \sigma d \)
getvar \( z_1 \rho (\sigma d + 1) \)
\[ \ldots \]
getvar \( z_{k-1} \rho (\sigma d + g - 1) \)
mkvec \( g \)
mktunval \( A \)
jump \( B \)
\[ A : \text{targ } k \]
codev \( e \rho (\sigma d) \)
\[ B : \ldots \]

where \( \{z_0, \ldots, z_{k-1}\} = \text{free}(\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \)
and \( \rho' = [x_i \mapsto (L, 1) | i = 0, \ldots, k - 1] \cup \{z_j \mapsto (G_i, 0) | j = 0, \ldots, g - 1\} \)

Example:

\( \lambda = \lambda a \rightarrow (G_0, 0) \mapsto (L, 1) \)

Regard \( f = \text{fun } b \rightarrow a + b \) for \( \rho = \{a \mapsto (L, 1)\} \) and \( \sigma d = 1 \).

codev \( f \rho \sigma d \) produces:

\[
\begin{array}{cccc}
0 & \text{pushloc } 0 & 0 & \text{pushglob } 0 & 2 & \text{getbasic} \\
1 & \text{mkvec } 1 & 1 & \text{eval} & 2 & \text{add} \\
1 & \text{mktunval } A & 1 & \text{getbasic} & 1 & \text{mkbasic} \\
0 & \text{jump } B & 1 & \text{pushloc } 1 & 1 & \text{return } 1 \\
0 & A : \text{targ } 1 & 2 & \text{eval} & 2 & B : \ldots
\end{array}
\]

The secrets around \text{targ } k \) and \text{return } k \) will be revealed later :-)
17 Function Application

Function applications correspond to function calls in C.
The necessary actions for the evaluation of \( e' (e_0 \ldots e_{m-1}) \) are:

- Allocation of a stack frame;
- Transfer of the actual parameters, i.e. with:
  - CBV: Evaluation of the actual parameters;
  - CBN: Allocation of closures for the actual parameters;
- Evaluation of the expression \( e' \) to an F-object;
- Application of the function.

Thus for CBN:

\[
\text{code}_{e'}(e_0 \ldots e_{m-1})\rho \quad \text{sd} = \begin{cases} \text{mark} & \text{// Allocation of the frame} \\ \text{code}_C \ e_{n-1} \rho \ (\text{sd} + 3) \\ \text{code}_C \ e_{n-2} \rho \ (\text{sd} + 4) \\ \vdots \\ \text{code}_C \ e_0 \rho \ (\text{sd} + m + 2) \\ \text{code}_{e'} \ e' \rho \ (\text{sd} + m + 3) & \text{// Evaluation of } e' \\ \text{apply} & \text{// corresponds to call} \end{cases}
\]

To implement CBV, we use \text{code}_{e'} \text{ instead of code}_C for the arguments \( e_i \).

Example: For \( (f \ 42), \rho = (f \rightarrow (L, 2)) \) and \( \text{sd} = 2 \), we obtain with CBV:

\[
\begin{align*}
2 & \text{mark A} & 6 & \text{mkbasic} & 7 & \text{apply} \\
5 & \text{loadc} 42 & 6 & \text{pushloc 4} & 3 & \text{A: ...}
\end{align*}
\]

A Slightly Larger Example:

\[
\text{let } a = 17 \text{ in let } f = \text{fun } b \rightarrow a + b \text{ in } f 42
\]

For CBV and \( \text{sd} = 0 \) we obtain:

\[
\begin{align*}
0 & \text{loadc 17} & 2 & \text{jump B} & 2 & \text{getbasic} & 5 & \text{loadc 42} \\
1 & \text{mkbasic} & 0 & \text{A: tag 1} & 2 & \text{add} & 5 & \text{mkbasic} \\
1 & \text{pushloc 0} & 0 & \text{pushglob 0} & 1 & \text{mkbasic} & 6 & \text{pushloc 4} \\
2 & \text{mkvec 1} & 1 & \text{getbasic} & 1 & \text{return 1} & 7 & \text{apply} \\
2 & \text{mkfunval} A & 1 & \text{pushloc 1} & 2 & B: \text{mark C} & 3 & \text{C: slide 2}
\end{align*}
\]

For the implementation of the new instruction, we must fix the organization of a stack frame:
Different from the CMA, the instruction **mark A** already saves the return address:

\[
\begin{align*}
S[SP+1] &= GP; \\
S[SP+2] &= FP; \\
S[SP+3] &= A; \\
FP &= SP = SP + 3;
\end{align*}
\]

Warning:

- The last element of the argument vector is the last to be put onto the stack. This must be the **first** argument reference.
- This should be kept in mind, when we treat the packing of arguments of an under-supplied function application into an F-object !!!

The instruction **apply** unpacks the F-object, a reference to which (hopefully) resides on top of the stack, and continues execution at the address given there:

\[h = S[SP];\]
\[\text{if } (H[h] \neq (\text{tag } k)) \]
\[\text{Error } "\text{no fun}";\]
\[\text{else } \{ \]
\[\text{GP} = h \rightarrow gp; \text{PC} = h \rightarrow cp; \]
\[\text{for } i = 0; \ i < h \rightarrow ap \rightarrow n; \ i++; \]
\[S[SP+i] = h \rightarrow ap \rightarrow v[i]; \]
\[SP = SP + h \rightarrow ap \rightarrow n - 1; \]
\[\} \]

18 Over- and Undersupply of Arguments

The first instruction to be executed when entering a function body, i.e., after an **apply** is **tag k**.

This instruction checks whether there are enough arguments to evaluate the body.

Only if this is the case, the execution of the code for the body is started.

Otherwise, i.e. in the case of **under-supply**, a new F-object is returned.

The test for number of arguments uses: \[SP - FP\]
\texttt{targ} \, k \, \text{is a complex instruction.}

We decompose its execution in the case of \textit{under-supply} into several steps:

\begin{verbatim}
targ \, k \, = \, \text{if} \, (\text{SP} - \text{FP} < k) \, \{\text{mkvec0}; \text{if creating the argument vector}
\text{wrap}; \text{if wrapping into an F-object}
\text{popenv}; \text{if popping the stack frame}\}\}
\end{verbatim}

The combination of these steps into one instruction is a kind of optimization \textit{\textbf{\textit{\textbf{D}}}.)}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{The instruction \texttt{mkvec0} takes all references from the stack above \texttt{FP} and stores them into a vector.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{The instruction \texttt{wrap} wraps the argument vector together with the global vector and \texttt{PC-1} into an \texttt{F-object}.}
\end{figure}
targ \( k \) is a complex instruction.

We decompose its execution in the case of under-supply into several steps:

\[
\text{targ } k = \begin{cases} 
\text{mkvec0}; & \text{// creating the argument vector} \\
\text{wrap}; & \text{// wrapping into an F-object} \\
\text{popenv; } & \text{// popping the stack frame} 
\end{cases}
\]

The combination of these steps into one instruction is a kind of optimization \( \implies \)

Thus, we obtain for \( \text{targ } k \) in the case of under-supply:

The instruction \( \text{popenv} \) finally releases the stack frame:

\[
\begin{align*}
\text{FP} & \rightarrow 42 \\
\text{PC} & \rightarrow 42 \\
\text{SP} & \rightarrow \text{FP} - 2 \\
\text{FP} & \rightarrow \text{SP} - 1 \\
\end{align*}
\]
- The stack frame can be released after the execution of the body if exactly the right number of arguments was available.
- If there is an oversupply of arguments, the body must evaluate to a function, which consumes the rest of the arguments ...
- The check for this is done by return k:

\[
\text{return } k = \begin{array}{ll}
& \text{if } (SP - FP = k + 1) \\
& \quad \text{popenv;} \\
& \phantom{=} \text{// Done} \\
& \text{else} \{ \\
& \quad \text{// There are more arguments} \\
& \quad \text{slide } k; \\
& \quad \text{apply;} \\
& \phantom{=} \text{// another application} \\
& \}\end{array}
\]

The execution of return k results in:

Case: Done

Case: Over-supply
19 let-rec-Expressions

Consider the expression $e \equiv \text{let rec } y_1 = e_1 \text{ and } \ldots \text{ and } y_n = e_n \text{ in } e_0$.

The translation of $e$ must deliver an instruction sequence that

- allocates local variables $y_1, \ldots, y_n$;
- in the case of
  - CBV: evaluates $e_1, \ldots, e_n$ and binds the $y_i$ to their values;
  - CBN: constructs closures for the $e_1, \ldots, e_n$ and binds the $y_i$ to them;
- evaluates the expression $e_0$ and returns its value.

Warning:

In a let-rec-expression, the definitions can use variables that will be allocated only later! $\Rightarrow$ Dummy-values are put onto the stack before processing the definition.

---

For CBN, we obtain:

$$\begin{align*}
\text{code}_v e \rho \text{ sd} &= \text{alloc } n \quad \text{ // allocates local variables} \\
\quad &\quad \text{code}_v e_1 \rho' (sd + n) \\
\quad &\quad \text{rewite n} \\
\quad &\quad \ldots \\
\quad &\quad \text{code}_v e_n \rho' (sd + n) \\
\quad &\quad \text{rewite 1} \\
\quad &\quad \text{code}_v e_0 \rho' (sd + n) \\
\quad &\quad \text{slide n} \quad \text{ // deallocates local variables}
\end{align*}$$

where $\rho' = \rho \uplus \{y_i \rightarrow (L, sd + i) \mid i = 1, \ldots, n\}$.

In the case of CBV, we also use code$_v$ for the expressions $e_1, \ldots, e_n$.

Warning:

Recursive definitions of basic values are undefined with CBV!!!