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:

$$
code_{e^y}(\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \rho s d =
\text{getvar } z_0 \rho s d
\text{getvar } z_1 \rho (s d + 1)
\ldots
\text{getvar } z_{n-1} \rho (s d + g - 1)
\text{mkvec } g
\text{mkunval } A
\text{jump } B
A:\text{target } k
\text{code}_e \rho' 0
\text{return } k
\text{B:} \ldots
$$

where $\{x_0, \ldots, x_{n-1}\} = \text{free}(\text{fun } x_0 \ldots x_{n-1} \rightarrow e)$
and $\rho' = \{x_i \mapsto (L, i) \mid i = 0, \ldots, k - 1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g - 1\}$
code\_\downarrow (\text{fun } x_0 \ldots x_{k-1} \to e) \rho \sd =
\begin{align*}
&\text{let } \tau_0 \rho \sd \ldots
&\text{getvar } z_1 \rho (\sd + 1)
&\text{getvar } z_{k-1} \rho (\sd + g - 1)
&\text{mkvec } g
&\text{mkfunval } A
&\text{jump } B
\end{align*}
A : \text{ targ } k
\begin{align*}
\text{code\_\downarrow e } \rho' \text{ 0}
&\text{return } k
\end{align*}
B : \ldots

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

---

Example

Regard \quad f \equiv \text{fun } b \to a + b \quad \text{for } \rho = \{ a \mapsto (L, 1) \} \quad \text{and } \sd = 1.

\text{code\_\downarrow f } \rho \text{ 1 produces:}

\begin{align*}
1 &\quad \text{pushloc } 0 & 0 &\quad \text{pushglob } 0 & 2 &\quad \text{getbasic} \\
2 &\quad \text{mkvec } 1 & 1 &\quad \text{eval} & 2 &\quad \text{add} \\
2 &\quad \text{mkfunval } A & 1 &\quad \text{getbasic} & 1 &\quad \text{mkbasic} \\
2 &\quad \text{jump } B & 1 &\quad \text{pushloc } 1 & 1 &\quad \text{return } 1 \\
0 &\quad A : \text{ targ } 1 & 2 &\quad \text{eval} & 2 &\quad B : \ldots
\end{align*}

The secrets around \text{ targ } k \quad \text{and } \text{ return } k \quad \text{will be revealed later} \; \therefore
17 Function Application

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

\[ S = \{ A \mapsto (\lambda_{1\mapsto 3} f \mapsto (\lambda_{2\mapsto 2} x \mapsto 17)) \} \]

A Slightly Larger Example

let \( a = 17 \) in let \( f = \text{fun} b \rightarrow a + b \) in \( f \; 42 \)

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>loadc 17</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>mkbasic</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>pushloc 0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>mkfuncall A</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>loadc 42</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>getbasic</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>pushloc 4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>apply</td>
<td></td>
</tr>
</tbody>
</table>

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.

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

---

**Caveat**

- 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.

---

18 Over- and Undersupply of Arguments

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.

\[
\begin{align*}
&h = S[SP]; \\
&\text{if } (H[h] = (F, \ldots)) \\
&\text{Error "no fun";} \\
&\text{else } \\
&\text{for } (i=0; i < h->ap->n; i++) \\
&\text{S[SP+i] = h->ap->v[i];} \\
&SP = SP + h->ap->n - 1;
\end{align*}
\]
\( \text{targ k} \) is a complex instruction. We decompose its execution in the case of \textit{under-supply} into several steps:

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

The combination of these steps into one instruction is a kind of optimization :-)

The instruction \text{mkvec0} takes all references from the stack above FP and stores them into a vector:

\[
g = \text{SP} - \text{FP}; \\
h = \text{new} \ (V, \ g); \\
\text{SP} = \text{FP} + 1; \\
\text{for} \ (i = 0; i < g; i++) \\
h \rightarrow v[i] = S[\text{SP} + i]; \\
S[\text{SP}] = h;
\]

The instruction \text{wrap} wraps the argument vector together with the global vector and PC-1 into an F-object:

\[
S[\text{SP}] = \text{new} \ (V, \text{PC-1}, S[\text{SP}], \text{GP});
\]

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

\[
\text{GP} = S[\text{FP} + 2]; \\
S[\text{FP} + 2] = S[\text{SP}]; \\
\text{PC} = S[\text{FP}]; \\
\text{SP} = \text{FP} - 2; \\
\text{FP} = S[\text{FP} + 1];
\]
The instruction \textit{wrap} wraps the argument vector together with the global vector and PC-1 into an F-object:

\begin{align*}
S[SP] &= \text{new } \langle F, PC-1, S[SP], GP \rangle;
\end{align*}

Thus, we obtain for \textit{targ k} in the case of under supply:
• 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`:

```plaintext
return k = if (SP - FP = k + 1)
    popenv; // Done
else {
    // There are more arguments
    slide k;
    apply; // another application
}
```

The execution of `return k` results in:

---

19 Let-rec-Expressions

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

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.

Caveat

In a letrec-expression, the definitions can use variables that will be allocated only later! `Dummy`-values are put onto the stack before processing the definition.
For CBN, we obtain:

\[
\text{code}_v e \; \rho \; \text{sd} = \text{alloc} \; n \quad \text{// allocates local variables}
\]

\[
\begin{align*}
\text{code}_c e_1 \; \rho' \; (\text{sd} + n) \\
\text{rewrite} \; n \\
\ldots \\
\text{code}_c e_n \; \rho' \; (\text{sd} + n) \\
\text{rewrite} \; 1 \\
\text{code}_v e_0 \; \rho' \; (\text{sd} + n) \\
\text{slide} \; n \quad \text{// deallocates local variables}
\end{align*}
\]

where \( \rho' = \rho \oplus \{ y_i \mapsto (L, \text{sd} + i) \mid i = 1, \ldots, n \} \).

In the case of CBV, we also use \text{code}_v for the expressions \( e_1, \ldots, e_n \).

\textbf{Caveat}

Recursive definitions of basic values are \textbf{undefined} with CBV!!!