18 Over– and Undersupply of Arguments

The first instruction to be executed when entering a function body, i.e., after an
apply is targ 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: \( SP - FP \)
targ $k$ is a complex instruction.

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

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

The combination of these steps into one instruction is a kind of optimization :-(
The instruction \texttt{mkvec0} takes all references from the stack above \texttt{FP} and stores them into a vector:

\begin{verbatim}
g = SP-FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
\end{verbatim}
The instruction \textbf{wrap A} wraps the argument vector together with the global vector into an F-object:

\begin{equation}
S[SP] = \text{new } (F, A, S[SP], GP);
\end{equation}
The instruction `popenv` finally releases the stack frame:

GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];
Thus, we obtain for $\text{targ } k$ in the case of under supply:
null
• 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`

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

The execution of `return k` results in:
Case: Done

\[ \text{popenv} \]
Case: Over-supply
19 letrec-Expressions

Consider the expression $e \equiv \text{letrec } y_1 = e_1; \ldots; y_n = e_n \text{ in } e_0$.

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

\[
\text{code}_V \ e \ \rho \ \text{sd} \ = \ \text{alloc} \ n \quad \text{// allocates local variables}
\]
\[
\text{code}_C \ e_1 \ \rho' \ (\text{sd} + n) \\
\text{rewrite} \ n \\
\ldots
\]
\[
\text{code}_C \ e_n \ \rho' \ (\text{sd} + n) \\
\text{rewrite} \ 1 \\
\text{code}_V \ e_0 \ \rho' \ (\text{sd} + n) \\
\text{slide} \ n \quad \text{// deallocates local variables}
\]

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{Warning:}

Recursive definitions of basic values are \textit{undefined} with CBV!!!
**Example:**

Consider the expression

\[ e \equiv \text{letrec } f = \text{fn} x, y \Rightarrow \text{if} y \leq 1 \text{ then } x \text{ else } f(x \ast y)(y - 1) \text{ in } f1 \]

for \( \rho = \emptyset \) and \( \text{sd} = 0 \). We obtain (for CBV):

<table>
<thead>
<tr>
<th></th>
<th>alloc 1</th>
<th></th>
<th>A: targ 2</th>
<th></th>
<th>loadc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pushloc 0</td>
<td></td>
<td>...</td>
<td></td>
<td>mkvec 1</td>
</tr>
<tr>
<td>1</td>
<td>mkvec 1</td>
<td>1</td>
<td>return 2</td>
<td>5</td>
<td>pushloc 4</td>
</tr>
<tr>
<td>2</td>
<td>mkfunval A</td>
<td>2</td>
<td>B: rewrite 1</td>
<td>6</td>
<td>apply</td>
</tr>
<tr>
<td>2</td>
<td>jump B</td>
<td>1</td>
<td>mark C</td>
<td>2</td>
<td>C: slide 1</td>
</tr>
</tbody>
</table>
The instruction \texttt{alloc n} reserves \( n \) cells on the stack and initialises them with \( n \) dummy nodes:

\begin{verbatim}
for (i=1; i<=n; i++)
  S[SP+i] = new (C,-1,-1);
SP = SP + n;
\end{verbatim}
The instruction \textit{rewrite n} overwrites the contents of the heap cell pointed to by the reference at \(S[SP-n]\):

\[H[S[SP-n]] = H[S[SP]];\]
\[SP = SP - 1;\]

- The \textit{reference} \(S[SP - n]\) remains unchanged!
- Only its \textit{contents} is changed!
20 Closures and their Evaluation

- Closures are needed for the implementation of CBN and for functional parameters.
- Before the value of a variable is accessed (with CBN), this value must be available.
- Otherwise, a stack frame must be created to determine this value.
- This task is performed by the instruction `eval`. 
**eval** can be decomposed into small actions:

\[
\text{eval} = \begin{cases} 
\text{mark0;} & \text{allocation of the stack frame} \\
\text{pushloc 3;} & \text{copying of the reference} \\
\text{apply0;} & \text{corresponds to apply}
\end{cases}
\]

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure thus means evaluation of an application of this function to 0 arguments.
- In contrast to **mark A**, **mark0** dumps the current **PC**.
- The difference between **apply** and **apply0** is that no argument vector is put on the stack.
FP = SP = SP + 3;
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;
We thus obtain for the instruction \texttt{eval}:

\[
\begin{align*}
\text{h} &= \text{S[SP]}; \text{SP}--; \\
\text{GP} &= \text{h} \rightarrow \text{gp}; \text{PC} = \text{h} \rightarrow \text{cp};
\end{align*}
\]
The construction of a closure for an expression $e$ consists of:

- Packing the bindings for the free variables into a vector;
- Creation of a C-object, which contains a reference to this vector and to the code for the evaluation of $e$:

$$
\text{code}_C \ e \ \rho \ \text{sd} = \text{getvar} \ z_0 \ \rho \ \text{sd} \\
\text{getvar} \ z_1 \ \rho \ (\text{sd} + 1) \\
\ldots \\
\text{getvar} \ z_{g-1} \ \rho \ (\text{sd} + g - 1) \\
\text{mkvec} \ g \\
\text{mkclos} \ A \\
\text{jump} \ B \\
A:\ \text{code}_V \ e \ \rho' \ 0 \\
\text{update} \\
B:\ \ldots
$$

where $\{z_0, \ldots, z_{g-1}\} = \text{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$. 
Example:

Consider $e \equiv a \ast a$ with $\rho = \{a \mapsto (L, 0)\}$ and $sd = 1$. We obtain:

<table>
<thead>
<tr>
<th></th>
<th>Operation</th>
<th>Line</th>
<th>Function</th>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pushloc 1</td>
<td>0</td>
<td>A:</td>
<td>2</td>
<td>getbasic</td>
</tr>
<tr>
<td>2</td>
<td>mkvec 1</td>
<td>1</td>
<td>eval</td>
<td>2</td>
<td>mul</td>
</tr>
<tr>
<td>2</td>
<td>mkclos A</td>
<td>1</td>
<td>getbasic</td>
<td>1</td>
<td>mkbasic</td>
</tr>
<tr>
<td>2</td>
<td>jump B</td>
<td>1</td>
<td>pushglob 0</td>
<td>1</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>eval</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

B: ...

166
• The instruction `mkclos A` is analogous to the instruction `mkfunval A`.
• It generates a C-object, where the included code pointer is `A`.

```
S[SP] = new (C, A, S[SP]);
```
In fact, the instruction \texttt{update} is the combination of the two actions:

\begin{itemize}
  \item \texttt{popenv}
  \item \texttt{rewrite 1}
\end{itemize}

It overwrites the closure with the computed value.