Example:

Regard \( e \equiv (b + c) \) for \( \rho = \{ b \mapsto (L, 1), c \mapsto (G, 0) \} \) and \( sd = 1 \).

With CBN, we obtain:

\[
\begin{align*}
\text{code} & e \; \rho \; 1 & = & \text{getvar} \; b \; \rho \; 1 \quad & = & 1 \; \text{pushloc} \; 0 \\
\text{} & & & \text{eval} \quad & = & 2 \; \text{eval} \\
\text{} & & & \text{getbasic} \quad & = & 2 \; \text{getbasic} \\
\text{} & & & \text{getvar} \; c \; \rho \; 2 \quad & = & 2 \; \text{pushglob} \; 0 \\
\text{} & & & \text{eval} \quad & = & 3 \; \text{eval} \\
\text{} & & & \text{getbasic} \quad & = & 3 \; \text{getbasic} \\
\text{} & & & \text{add} \quad & = & 3 \; \text{add} \\
\text{} & & & \text{mkbasic} \quad & = & 2 \; \text{mkbasic}
\end{align*}
\]

15 let-Expressions

As a warm-up let us first consider the treatment of local variables.

Let \( e = \text{let } y_1 = e_1 \; \text{in} \ldots \text{let } y_n = e_n \) be a nested let-expression.

The translation of \( e \) must deliver an instruction sequence that

- allocates local variables \( y_1, \ldots, y_n \);
- in the case of
  - CBN: 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.

Here, we consider the non-recursive case only, i.e. where \( y_j \) only depends on \( y_1, \ldots, y_{j-1} \). We obtain for CBN:
code ν ρ sd = code ν 1 ρ sd
code ν 2 ρ₁ (sd + 1) 
... 
code ν n ρₙ₋₁ (sd + n - 1)
code ν n ρₙ (sd + n)
slide n // deallocate local variables

where ρᵢ = ρ ⊕ \{ yᵢ → (L, sd + i) | i = 1, ..., f\}.
In the case of CBV, we use code ν for the expressions e₁, ..., eₙ.

Warning!
All the eᵢ must be associated with the same binding for the global variables!

The instruction slide k deallocates again the space for the locals:

[k] → [k + 1]

Example:
Consider the expression
\[ \epsilon = \text{let } a = 19 \text{ in } b = a \times a \text{ if } \begin{array}{c}
\text{if } a + b \\
\end{array} \]
for \( \rho = \emptyset \) and \( sd = 0 \). We obtain (for CBV):

\[
\begin{array}{cccc}
0 & \text{load } 19 & 3 & \text{getbasic} & 3 & \text{pushloc } 1 \\
1 & \text{mkbasic} & 3 & \text{mul} & 4 & \text{getbasic} \\
2 & \text{pushloc } 0 & 4 & \text{mkbasic} & 4 & \text{add} \\
2 & \text{getbasic} & 2 & \text{pushloc } 1 & 3 & \text{mkbasic} \\
2 & \text{pushloc } 1 & 3 & \text{getbasic} & 3 & \text{slide } 2
\end{array}
\]
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:

\[
\text{codev (fun } x_0 \ldots x_{k-1} \rightarrow e \text{) } \rho \text{ } \sigma \text{ } d = \begin{align*}
\text{getvar } z_0 \rho \text{ } \sigma d \\
\text{getvar } z_1 \rho \text{ } (\sigma d + 1) \\
\vdots \\
\text{getvar } z_{k-1} \rho \text{ } (\sigma d + g - 1) \\
\text{mkvec } g \\
\text{mkfunval } A \\
\text{jump } B \\
\Lambda : \text{ } \text{tag } k \\
\text{codev } e \rho' \text{ } 0 \\
\text{return } k \\
B : \ldots
\end{align*}
\]

where \( \{z_0, \ldots, z_{k-1}\} = \text{free(fun } x_0 \ldots x_{k-1} \rightarrow e\) \)
and \( \rho' = \{x_i \mapsto (L_i - i) \mid i = 0, \ldots, k - 1\} \cup \{z_i \mapsto (G_i) \mid j = 0, \ldots, g - 1\} \)
17 Function Application

Function applications correspond to function calls in C.
The necessary actions for the evaluation of \( \varepsilon \epsilon_0 \ldots \epsilon_{n-1} \) 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 \( \varepsilon' \) to an F-object;
- Application of the function.

Thus for CBN:

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

\[
\begin{array}{cccc}
2 & \text{mark} & A & 6 \\
5 & \text{load} & 42 & 6 \\
\end{array}
\]

Example:
\[
\begin{array}{cccc}
1 & \text{pushloc} & 0 & 0 \\
2 & \text{pushloc} & 1 & 2 \\
2 & \text{mkvec} & 1 & 1 \\
2 & \text{mkfunc} & A & 1 \\
2 & \text{jump} & B & 1 \\
0 & \text{targ} & 1 & 2 \\
\end{array}
\]
A Slightly Larger Example:

\[ g = \{ a \mapsto (1, 1), f \mapsto (5, 2) \} \]

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadc</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkbasic</td>
<td></td>
<td>0</td>
<td>A:</td>
<td>tgr 1</td>
<td>2</td>
<td>add</td>
<td>5</td>
<td>mkbasic</td>
</tr>
<tr>
<td>pushloc</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getbasic</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkvec</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkfunval</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the implementation of the new instruction, we must fix the organization of a stack frame:

- SP
- FP
- PCold
- FPod
- GPold
- 3 org. cells
- local stack
- Arguments

Different from the CMA, the instruction \text{mark A} already saves the return address:

- SP
- FP
- GP
- V1
- \( A \)
- \( \text{mark A} \)
- \( \text{SP+1} = \text{GP} \)
- \( \text{SP+2} = \text{FP} \)
- \( \text{SP+3} = A \)
- \( \text{FP} = \text{SP}' = \text{SP} + 3 \)
The instruction \texttt{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{verbatim}

h = S(SP);
if (H[h] !≡ (F,∞))
  Error "no fun";
else {
  GP = h→gp;
  PC = h→cp;
  for (i=0; i< h→ap→n; i++)
    S[SP+i] = h→ap→v[i];
  SP = SP + h→ap→n - 1;
}

\end{verbatim}

\textbf{Warning:}

- The last element of the argument vector is the last to be put onto the stack. This must be the \textit{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

\section{Over- and Undersupply of Arguments}

The first instruction to be executed when entering a function body, i.e., after an apply \texttt{is} \texttt{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 \underline{under-supply}, a new F-object is returned.

The test for number of arguments uses: \texttt{SP - FP}
The instruction **mkvec0** takes all references from the stack above FP and stores them into a vector:

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

The instruction **wrap A** wraps the argument vector together with the global vector into an F-object:

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

\[ \text{GP} = s[\text{FP}-2]; \]
\[ s[\text{FP}-2] = s\{\text{SP}\}; \]
\[ \text{PC} = s[\text{FP}]; \]
\[ \text{SP} = \text{FP} - 2; \]
\[ \text{FP} = s[\text{FP}-1]; \]

Thus, we obtain for `targ k` in the case of under supply: