If $e'$ evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath $e_0$:
The further arguments $a_0, \ldots, a_{k-1}$ and the local variables can be allocated above the arguments.
Addressing of arguments and local variables relative to FP is no more possible. (Remember: $m$ is unknown when the function definition is translated.)
Way out:

- We address both, arguments and local variables, relative to the stack pointer $SP$.

- However, the stack pointer changes during program execution...
• The difference between the current value of SP and its value \( sp_0 \) at the entry of the function body is called the stack distance, \( sd \).

• Fortunately, this stack distance can be determined at compile time for each program point, by simulating the movement of the SP.

• The formal parameters \( x_0, x_1, x_2, \ldots \) successively receive the non-positive relative addresses \( 0, -1, -2, \ldots \), i.e., \( \rho x_i = (L, -i) \).

• The absolute address of the \( i \)-th formal parameter consequently is

\[
sp_0 - i = (SP - sd) - i
\]

• The local let-variables \( y_1, y_2, y_3, \ldots \) will be successively pushed onto the stack:
The $y_i$ have positive relative addresses 1, 2, 3, . . ., that is: $\rho y_i = (L, i)$.

The absolute address of $y_i$ is then $sp_0 + i = (SP - sd) + i$
With CBN, we generate for the access to a variable:

\[
\text{code}_V \ x \ \rho \ \text{sd} = \ \text{getvar} \ x \ \rho \ \text{sd} \ \\\text{eval}
\]

The instruction \text{eval} checks, whether the value has already been computed or whether its evaluation has to yet to be done \(\Rightarrow\) will be treated later \(\text{:-)}\)

With CBV, we can just delete \text{eval} from the above code schema.

The (compile-time) macro \text{getvar} is defined by:

\[
\text{getvar} \ x \ \rho \ \text{sd} = \ \text{let} \ (t, i) = \rho \ x \ \text{in}
\text{case} \ t \ \text{of}
\begin{align*}
L & \Rightarrow \text{pushloc} \ (\text{sd} - i) \\
G & \Rightarrow \text{pushglob} \ i
\end{align*}
\text{end}
\]
The access to local variables:

\[ \text{pushloc } n \]

\[ S[SP+1] = S[SP - n]; SP++; \]
Correctness argument:

Let \( sp \) and \( sd \) be the values of the stack pointer resp. stack distance before the execution of the instruction. The value of the local variable with address \( i \) is loaded from \( S[a] \) with

\[
a = sp - (sd - i) = (sp - sd) + i = sp_0 + i
\]

... exactly as it should be  :-)

The access to global variables is much simpler:

```
pushglob i

SP = SP + 1;
S[SP] = GP→v[i];
```
Example:

Regard \( e \equiv (b + c) \) for \( \rho = \{ b \mapsto (L, 1), c \mapsto (G, 0) \} \) and \( \text{sd} = 1 \). With CBN, we obtain:

\[
\begin{align*}
\text{code}_V e \rho 1 & = \text{getvar} b \rho 1 = 1 \quad \text{pushloc} 0 \\
& \quad \text{eval} \\
& \quad \text{getbasic} \\
& \quad \text{getvar} c \rho 2 \\
& \quad \text{eval} \\
& \quad \text{getbasic} \\
& \quad \text{add} \\
& \quad \text{mkbasic}
\end{align*}
\]
15 let-Expressions

As a warm-up let us first consider the treatment of local variables :-) Let \( e \equiv \textbf{let} \ y_1 = e_1; \ldots ; y_n = e_n \ \textbf{in} \ e_0 \) be a let-expression.

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.

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:
\text{code}_V e \rho \text{sd} = \text{code}_C e_1 \rho \text{sd} \\
\text{code}_C e_2 \rho_1 (\text{sd} + 1) \\
\ldots \\
\text{code}_C e_n \rho_{n-1} (\text{sd} + n - 1) \\
\text{code}_V e_0 \rho_n (\text{sd} + n) \\
\text{slide } n \quad \text{// deallocates local variables}

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

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

\textbf{Warning!}

All the \( e_i \) must be associated with the same binding for the global variables!
Example:

Consider the expression

\[ e \equiv \textbf{let} \ a = 19; b = a \times a \ \textbf{in} \ a + b \]

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

<table>
<thead>
<tr>
<th></th>
<th>loadc 19</th>
<th></th>
<th>getbasic</th>
<th></th>
<th>pushloc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>mkbasic</td>
<td></td>
<td>mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>pushloc 0</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>getbasic</td>
<td></td>
<td></td>
<td>4</td>
<td>add</td>
</tr>
<tr>
<td>2</td>
<td>pushloc 1</td>
<td></td>
<td>pushloc 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>slide 2</td>
</tr>
</tbody>
</table>
The instruction **slide k** deallocates again the space for the locals:

\[
S[SP-k] = S[SP]; \\
SP = SP - k;
\]
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 these vectors and the start address of the code for the body;

Separately, code for the body has to be generated.

Thus:
\[
\text{code}_V (\text{fn } x_0, \ldots, x_{k-1} \Rightarrow e) \rho \text{ sd} = \text{getvar } z_0 \rho \text{ sd} \\
\text{getvar } z_1 \rho (\text{sd} + 1) \\
\ldots \\
\text{getvar } z_{g-1} \rho (\text{sd} + g - 1) \\
\text{mkvec } g \\
\text{mkfunval } A \\
\text{jump } B \\
A : \text{targ } k \\
\text{code}_V e \rho' 0 \\
\text{return } k \\
B : \ldots
\]

where \( \{z_0, \ldots, z_{g-1}\} = \text{free}(\text{fn } x_0, \ldots, x_{k-1} \Rightarrow e) \)

and \( \rho' = \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k - 1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g - 1\} \)
\[
h = \text{new} \ (V, n);
SP = SP - g + 1;
\text{for} \ (i=0; \ i<g; \ i++)
\quad h \rightarrow v[i] = S[SP + i];
S[SP] = h;
\]
a = new (V,0);
S[SP] = new (F, A, a, S[SP]);
Example:

Regard $f \equiv \textbf{fn} \ b \Rightarrow a + b$ for $\rho = \{a \mapsto (L, 1)\}$ and $sd = 1$.

code $f \rho 1$ produces:

```
1  pushloc 0
2   mkvec 1
2      mkfunval A
2        jump B
0    A :  targ 1
0  pushglob 0
1       eval
1              getbasic
1         mkbasic
1          return 1
```

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:
  - \textbf{CBV}: Evaluation of the actual parameters;
  - \textbf{CBN}: Allocation of closures for the actual parameters;
- Evaluation of the expression \( e' \) to an F-object;
- Application of the function.

Thus for \textbf{CBN}:
To implement CBV, we use code\_V instead of code\_C for the arguments e\_i.

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

\begin{verbatim}
2  mark A
5  loadc 42
6  pushloc 4
7  apply
\end{verbatim}
A Slightly Larger Example:

\[
\text{let } a = 17; \ f = \text{fn } b \Rightarrow a + b \ \text{in } f \ 42
\]

For \text{CBV and } k p = 0 \ we \ obtain:

\[
\begin{array}{cccccc}
0 & \text{loadc } 17 & 2 & \text{jump } B & 2 & \text{getbasic} & 5 & \text{loadc } 42 \\
1 & \text{mkbasic} & 0 & \text{A:} & \text{targ } 1 & 2 & \text{add} & 5 & \text{mkbasic} \\
1 & \text{pushloc } 0 & 0 & \text{pushglob } 0 & 1 & \text{mkbasic} & 6 & \text{pushloc } 4 \\
2 & \text{mkvec } 1 & 1 & \text{getbasic} & 1 & \text{return } 1 & 7 & \text{apply} \\
2 & \text{mkfunval } A & 1 & \text{pushloc } 1 & 2 & \text{B:} & \text{mark } C & 3 & \text{C:} & \text{slide } 2
\end{array}
\]
For the implementation of the new instruction, we must fix the organization of a stack frame:

```
SP →
  local stack
  Arguments
  3 org. cells

FP →
  PCold     0
  FPold     -1
  GPold     -2
```

FPold
PCold
GPold
Remember: Addressing of arguments and local variables

\[ \text{sp}_0 \rightarrow e_0 \rightarrow e_{m-1} \]
Different from the CMa, the instruction \textbf{mark A} already saves the return address:

\[
S[SP+1] = GP; \\
S[SP+2] = FP; \\
S[SP+3] = A; \\
FP = SP = SP + 3;
\]
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 {
    for (i=0; i<h→ap→n; i++)
        S[SP+i] = h→ap→v[i];
    SP = SP + h→ap→n - 1;
}
\end{verbatim}