Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the `gp`-component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In contrast, local variables should be administered on the stack ...

$\Rightarrow$ General form of the address environment:

$$
\rho : Vars \to \{L, G\} \times \mathbb{Z}
$$
Accessing Local Variables

Local variables are administered on the stack, in stack frames.

Let $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ be the application of a function $e'$ to arguments $e_0, \ldots, e_{m-1}$.

Warning:

The arity of $e'$ does not need to be $m$ :-)

- $f$ may therefore receive less than $n$ arguments (under supply);
- $f$ may also receive more than $n$ arguments, if $t$ is a functional type (over supply).
Possible stack organisations:

Addressing of the arguments can be done relative to FP.

- The local variables of $e'$ cannot be addressed relative to FP.

- If $e'$ is an $n$-ary function with $n < m$, i.e., we have an over-supplied function application, the remaining $m - n$ arguments will have to be shifted.
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 \( e_0 \):
Alternative:

+ 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 \texttt{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:

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

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

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

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

\[
\text{getvar} \ x \ \rho \ sd \ = \ \text{let} \ (t, i) = \rho \ x \ \text{in} \\
\text{match} \ t \ \text{with} \\
L \to \text{pushloc} \ (sd - i) \\
| \ G \to \text{pushglob} \ i \\
\text{end}
\]
The access to local variables:

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

Let \( sp \) and \( sd \) be the values of the stack pointer resp. stack distance \textbf{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:

\[ \text{pushglob } i \]

\[
\begin{align*}
\text{GP} & \quad V \\
\text{SP} & \quad = \text{SP} + 1; \\
\text{S[SP]} & \quad = \text{GP} \rightarrow v[i];
\end{align*}
\]
Example:

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

With CBN, we obtain:

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

As a warm-up let us first consider the treatment of local variables :-)  
Let \( e \equiv \textbf{let} \ y_1 = e_1 \ \textbf{in} \ldots \textbf{let} \ e_n \ \textbf{in} \ e_0 \) 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 \( \text{CBV} \) evaluates \( e_1, \ldots, e_n \) and binds the \( y_i \) to their values;  
  \( \text{CBN} \) 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 \( \text{CBN} \):
\[
\text{code}_V e \rho \text{ sd} = \text{code}_C e_1 \rho \text{ sd} \\
\text{code}_C e_2 \rho_1 (\text{sd} + 1) \\
\ldots \\
\text{code}_C e_n \rho_{n-1} (\text{sd} + n - 1) \\
\text{code}_V e_0 \rho_n (\text{sd} + n) \\
\text{slide n} \quad \text{// deallocates local variables}
\]

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

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

Warning!

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

Consider the expression

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

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

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

\begin{align*}
S[SP-k] &= S[SP]; \\
SP &= SP - k;
\end{align*}
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:
\[
\text{\texttt{code}}_V (\text{\texttt{fun}} \ x_0 \ldots x_{k-1} \to e) \ \rho \ \text{sd} = \begin{aligned}
&\text{getvar } z_0 \ \rho \ \text{sd} \\
&\text{getvar } z_1 \ \rho \ (\text{sd} + 1) \\
&\ldots \\
&\text{getvar } z_{g-1} \ \rho \ (\text{sd} + g - 1) \\
&\text{mkvec } g \\
&\text{mkfunval } A \\
&\text{jump } B \\
&A : \ \text{targ } k \\
&\text{\texttt{code}}_V e \ \rho' \ 0 \\
&\text{return } k \\
&B : \ \ldots
\end{aligned}
\]

where \[\{z_0, \ldots, z_{g-1}\} = \text{free}(\text{\texttt{fun}} \ x_0 \ldots x_{k-1} \to e)\]

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 = new (V, n);
SP = SP - g + 1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
a = new (V,0);
S[SP] = new (F, A, a, S[SP]);
Example:

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

code \( f \rho 1 \) produces:

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

The secrets around \( \text{targ} k \) and \( \text{return} k \) will be revealed later  :-(
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:
To implement CBV, we use code\textsubscript{V} instead of code\textsubscript{C} for the arguments \(e_i\).

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

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

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

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

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>loadc 17</td>
<td>2</td>
<td>jump B</td>
<td>2</td>
<td>getbasic</td>
<td>5</td>
<td>loadc 42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>mkbasic</td>
<td>0</td>
<td>A:</td>
<td>targ 1</td>
<td>2</td>
<td>add</td>
<td>5</td>
<td>mkbasic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>pushloc 0</td>
<td>0</td>
<td>pushglob 0</td>
<td>1</td>
<td>mkbasic</td>
<td>6</td>
<td>pushloc 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mkvec 1</td>
<td>1</td>
<td>getbasic</td>
<td>1</td>
<td>return 1</td>
<td>7</td>
<td>apply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mkfunval A</td>
<td>1</td>
<td>pushloc 1</td>
<td>2</td>
<td>B:</td>
<td>mark C</td>
<td>3</td>
<td>C:</td>
<td>slide 2</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:

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

\[
\begin{align*}
\text{h} &= \text{S[SP]}; \\
\text{if } (\text{H[h]} \neq (\text{F, }\_\_)) &\quad \text{GP} = \text{h} \to \text{gp}; \text{PC} = \text{h} \to \text{cp}; \\
\text{Error “no fun”;} &\quad \text{for } (\text{i=0}; \text{i}<\text{h} \to \text{ap} \to \text{n}; \text{i}++) \\
\text{else } \{ &\quad \text{S[SP+i]} = \text{h} \to \text{ap} \to \text{v[i]}; \\
\text{SP} &= \text{SP} + \text{h} \to \text{ap} \to \text{n} - 1; \\
\} \\
\end{align*}
\]
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 !!!