... it can be thought of as an abstract data type, being capable of holding data objects of the following form:

\[
\begin{align*}
  & v \\
  & B \ -173 \\
  & cP \quad gP \\
  & C \\
  & cP \quad aP \quad gP \\
  & F \\
  & v[0] \quad \ldots \quad v[n-1] \\
  & V \ n \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots
\end{align*}
\]

... Basic Value

The instruction `new (tag, args)` creates a corresponding object \( (B, C, F, V) \) in \( H \) and returns a reference to it.

We distinguish three different kinds of code for an expression \( e \):

- `code_0` \( e \) — (generates code that) computes the Value of \( e \), stores it in the heap and returns a reference to it on top of the stack (the normal case);
- `code_2` \( e \) — computes the value of \( e \), and returns it on the top of the stack (only for Basic types);
- `code_1` \( e \) — does not evaluate \( e \), but stores a Closure of \( e \) in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:
13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

\[
\begin{align*}
\text{code}_\rho (b) & \quad = \quad \text{load}_b \\
\text{code}_\rho (e_1 \oplus e_2) & \quad = \quad \text{code}_\rho e_1 \rho \text{sd} \\
& \quad \quad \quad \text{op}_1 \\
\text{code}_\rho (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) & \quad = \quad \text{code}_\rho e_0 \rho \text{sd} \\
& \quad \quad \quad \text{jump}_A \\
& \quad \quad \quad \text{code}_\rho e_1 \rho \text{sd} \\
& \quad \quad \quad \quad \text{jump}_B \\
& \quad \quad \quad \text{code}_\rho e_2 \rho \text{sd} \\
& \quad \quad \quad \quad \quad \ldots
\end{align*}
\]

Note:

- \( \rho \) denotes the actual address environment, in which the expression is translated.
- The extra argument \( \text{sd} \), the stack difference, simulates the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.
- The instructions \( \text{op}_1 \) and \( \text{op}_2 \) implement the operators \( \Box_1 \) and \( \Box_2 \); in the same way as the the operators \( \text{neg} \) and \( \text{add} \) implement negation resp. addition in the CMU.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

\[
\begin{align*}
\text{code}_\rho (e) & \quad = \quad \text{code}_\rho e \rho \text{sd} \\
& \quad \quad \quad \text{get}_\text{basic}
\end{align*}
\]
For \texttt{code_V} and simple expressions, we define analogously:

\[
\begin{align*}
\text{code}_V (b \; \rho \; s d) &= \text{load}_b \; \text{by}_\rho \; \text{mkbasic} \\
\text{code}_V (\mathsf{\Omega}_e \; \rho \; s d) &= \text{code}_B \; e \; \rho \; s d \\
&\quad \text{op}_\Omega \; \text{mkbasic} \\
\text{code}_V (e_1 \; \mathsf{?\_} \; e_2) \; \rho \; s d &= \text{code}_B \; e_1 \; \rho \; s d \\
&\quad \text{code}_B \; e_2 \; \rho \; (s + 1) \\
&\quad \text{op}_\mathsf{?\_} \; \text{mkbasic} \\
\text{code}_V (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \; \rho \; s d &= \text{code}_B \; e_0 \; \rho \; s d \\
&\quad \text{jump}_z \; A \\
&\quad \text{code}_B \; e_1 \; \rho \; s d \\
&\quad \text{jump}_B \\
A: \quad &\text{code}_B \; e_2 \; \rho \; s d \\
B: \quad &... \\
\end{align*}
\]
For `codeV` and simple expressions, we define analogously:

\[
\begin{align*}
\text{codeV } b \rho sd &= \text{ loader } \text{ by } \text{ mkbasic} \\
\text{codeV } (\land_1 \ e) \rho sd &= \text{ codeV } e \rho sd \\
\text{op}_\land \text{ mkbasic} \\
\text{codeV } (e_1 \land_2 e_2) \rho sd &= \text{ codeV } e_1 \rho sd \\
& \quad \text{codeV } e_2 \rho (sd + 1) \\
& \quad \text{op}_\land \text{ mkbasic} \\
\text{codeV } (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho sd &= \text{ codeV } e_0 \rho sd \\
& \quad \text{jumpZ } A \\
& \quad \text{codeV } e_1 \rho sd \\
& \quad \text{jump } B \\
& \quad \begin{array}{l}
\text{A: codeV } e_2 \rho sd \\
\text{B: ...}
\end{array}
\end{align*}
\]

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function \( f \) :

\[
\begin{align*}
\text{let } c &= 5 \\
\text{in } \text{let } f = \text{ fun } a \rightarrow \text{ let } b = a + a \\
& \quad \text{in } b + c \\
\text{in } f \ c
\end{align*}
\]

The function \( f \) uses the global variable \( c \) and the local variables \( a \) (as formal parameter) and \( b \) (introduced by the inner let).

The binding of a global variable is determined, when the function is constructed (static scoping!), and later only looked up.
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 ...

\[ \rho : \text{vars} \rightarrow \{L, G\} \times Z \]

Accessing Local Variables

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

Let \( e = e' \; e_0 \; \ldots \; e_{m-1} \) 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 \( f \) 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.

Alternative:

- The further arguments \( e_0, \ldots, e_{m-1} \) 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.)

- If \( e' \) evaluates to a function, which has already been partially applied to the parameters \( a_0, \ldots, a_{q-1} \), these have to be sneaked in underneath \( e_0 \):

- 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.
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., $sp_i = (L - i)$.
- 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 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 $sp_0 \downarrow i = (SP - sd) \downarrow i$

• The local let-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:

With CBN, we generate for the access to a variable:

```plaintext
code \ x \ rho \ sd = getvar \ x \ rho \ sd
   eval
```

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

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

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

```plaintext
getvar \ x \ rho \ sd = let (C) = \rho \ x in
   match i with
   | 0 \to \ pushloc (sd - i)
   | C \to \ pushglob i
   end
```

The access to local variables:

```plaintext
\texttt{S[SP+1] = S[SP - n]; SP++;}
```
With CBN, we generate for the access to a variable:

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

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

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

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

\[
\text{getvar}_{x\rho sd} = \text{let}\ (t, i) = \rho x\text{ in}
\]

\[
\text{match}\ i\text{ with}
\]

\[
L \rightarrow \text{pushloc}\ (sd - i)
\]

\| \ G \rightarrow \text{pushglob}\ i
\]

\end

\[\text{SP} = \text{SP} + 1;\]

\[\text{S}[\text{SP}] = \text{GP} \rightarrow v[i];\]

With CBN, we generate for the access to a variable:

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

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

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

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

\[
\text{getvar}_{x\rho sd} = \text{let}\ (t, i) = \rho x\text{ in}
\]

\[
\text{match}\ i\text{ with}
\]

\[
L \rightarrow \text{pushloc}\ (sd - i)
\]

\| \ G \rightarrow \text{pushglob}\ i
\]

\end

\[\text{SP} = \text{SP} + 1;\]

\[\text{S}[\text{SP}] = \text{GP} \rightarrow v[i];\]

Let \text{sp} and \text{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 \(S[i]\) with:

\[
a = \text{sp} - (\text{sd} - i) = (\text{sp} - \text{sd}) + i = \text{sp}_0 + i
\]

... exactly as it should be \(\Rightarrow\)
15 let-Expressions

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

Let $e \equiv \text{let } y_1 = e_1 \text{ in } \ldots \text{ let } y_n = e_n \text{ in } e_0$ 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
  - 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 $y_i$ only depends on $y_1, \ldots, y_{i-1}$. We obtain for CBN:

\[ e \equiv (b + c) \quad \text{for} \quad \rho = \{(b \mapsto \langle L, 1 \rangle), (c \mapsto \langle G, 0 \rangle)\} \quad \text{and} \quad \text{sd} = 1. \]

With CBN, we obtain:

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

Consider the expression

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

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

\[
\begin{array}{ccc}
0 & \text{loadc} & 19 \\
1 & \text{mkbasic} & 3 \\
2 & \text{pushloc} & 0 \\
3 & \text{getbasic} & 3 \\
4 & \text{mul} & 4 \\
5 & \text{getbasic} & 3 \\
6 & \text{add} & 4 \\
7 & \text{pushloc} & 1 \\
8 & \text{mkbasic} & 3 \\
9 & \text{pushloc} & 1 \\
10 & \text{getbasic} & 3 \\
11 & \text{slide} & 2 \\
\end{array}
\]

The instruction \( \text{slide } k \) deallocates again the space for the locals:

\[ S[\text{SP}+k] = S[\text{SP}]; \]
\[ \text{SP} = \text{SP} - k; \]