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: \[
\text{SP} - \text{FP}
\]

\[
\text{Function}\ x, y \rightarrow \text{id}
\]

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

The combination of these steps into one instruction is a kind of optimization :)
The instruction `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->v[i] = S[SP + i];
S[SP] = h;
```

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->v[i] = S[SP + i];
S[SP] = h;
```

The instruction `wrap` wraps the argument vector together with the global vector and PC-1 into an F-object:

```
S[SP] = new (f, PC-1, S[SP], GP);
```

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:
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 \( \text{return k:} \)

\[
\text{return } k = \begin{cases} 
\text{if } (SP - FP = k + 1) \\
\text{popenv;} \\
\text{else} \{ \\
\text{slide } k; \\
\text{apply;} \\
\}
\end{cases}
\]

The execution of \( \text{return k} \) results in:

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 \( \text{return k:} \)

\[
\text{return } k = \begin{cases} 
\text{if } (SP - FP = k + 1) \\
\text{popenv;} \\
\text{else} \{ \\
\text{slide } k; \\
\text{apply;} \\
\}
\end{cases}
\]

The execution of \( \text{return k} \) results in:

Case: Done

Case: Over-supply
Case: Over-supply

- 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 \texttt{return k}:

\[
\text{return } k = \begin{cases} 
\text{if (SP - FP = k + 1)} & \text{popenv;} \\
\text{else} & \\
\text{slide } k; & \text{// Done} \\
\text{apply;} & \text{// There are more arguments} \\
\end{cases}
\]

The execution of \texttt{return k} results in:

19 \textbf{let-rec-Expressions}

Consider the expression \( e = \text{let rec } y_1 = e_1 \text{ and } \ldots \text{ and } y_n = e_n \text{ in } e_0 \) .

The translation of \( e \) must deliver an instruction sequence that
- allocates local variables \( y_1, \ldots, y_n \);
- in the case of\textbf{CBV:} evaluates \( e_1, \ldots, e_n \) and binds the \( y_i \) to their values;
\textbf{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.

\textbf{Warning:}

In a letrec-expression, the definitions can use variables that will be allocated only \textbf{later!} \( \Rightarrow \text{Dummy-values are put onto the stack before processing the definition.} \)
For CBN, we obtain:

\[
\begin{align*}
\text{codey } & \ e \ p \ sd & = & \text{alloc } n \quad & \text{// allocates local variables} \\
& & & \text{codey } & \ e_1 \ p' \ (sd + n) \\
& & & \text{rewrite } & n \\
& & & \vdots \\
& & & \text{codey } & \ e_\alpha \ p' \ (sd + n) \\
& & & \text{rewrite } & 1 \\
& & & \text{codey } & \ e_\alpha \ p' \ (sd + n) \\
& & & \text{slide } & n \quad & \text{// deallocates local variables} \\
\end{align*}
\]

where \( p' = p \oplus \{ y_i \mapsto (L, sd + i) \mid i = 1, \ldots, n \} \).

In the case of CBV, we also use codey for the expressions \( e_1, \ldots, e_\alpha \).

**Warning:**

Recursive definitions of basic values are **undefined** with CBV!!!
The instruction `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 reference $S[SP-n]$ remains unchanged!
- Only its 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:

```
    mark0
    pushloc 3;
    apply0;
```

```
    if (H[S[SP]] == (C, ...)) {
        // allocation of the stack frame
        // copying of the reference
        // corresponds to apply
    }
```

- 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 markA, mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put on the stack.

```
h = S[SP]; SP~;
GP = h→gp; PC = h→cp;
```

We thus obtain for the instruction eval:
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 $e$.

$$\text{codec}_c \; e \; \rho \; \text{sd} = \begin{cases} \text{getvar} \; z_0 \; \rho \; \text{sd} \\ \text{getvar} \; z_1 \; \rho \; (\text{sd} + 1) \\ \vdots \\ \text{getvar} \; z_{g-1} \; \rho \; (\text{sd} + g - 1) \\ \text{mkvec} \; g \\ \text{mkcl} \; A \\ \text{jump} \; B \end{cases}$$

$A : \; \text{codec}_c \; e \; \rho' \; 0$

$B : \; \cdots$

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

Example:

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

1. pushloc 1 0 A: pushglob 0 2 getbasic
2. mkvec 1 1 eval
2. mkcl A 1 getbasic 1 mkclmk
2. jump B 1 pushglob 0 1 eval
2. B: ...
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 $e$:

```plaintext
code_c e ρ sd  =
getvar z_0 ρ sd
getvar z_1 ρ (sd + 1)
...
getvar z_{i-1} ρ (sd + g - 1)
mkvec g
mkclo s A
jump B
A : code_c e ρ' 0
update
B : ...
```

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

**Example:**

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

```
1 pushloc 1 0 A: pushglob 0 2 gotbasic
2 mkvec 1 1 eval 2 mul
2 mkclo s A 1 gotbasic 1 mkclo s update
2 jump B 1 pushglob 0 1 update
```

The instruction `mkclo s 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 `update` is the combination of the two actions:

```
popen
rewrite 1
```

It overwrites the closure with the computed value.
21 Optimizations I: Global Variables

Observation:

- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables.
  Recall, e.g., the construction of a closure for an expression $e$...

In fact, the instruction `update` is the combination of the two actions:
```
popenv
rewrite 1
```
It overwrites the closure with the computed value.

\[
\text{code: } e \rho \sigma d = \begin{align*}
&\text{getvar } z_0 \rho \sigma d \\
&\text{getvar } z_1 \rho (\sigma d + 1) \\
&\ldots \\
&\text{getvar } z_{g-1} \rho (\sigma d + g - 1) \\
&\text{mkvec } g \\
&\text{mkclos } A \\
&\text{jump } B \\
A : &\text{ code: } e \rho' 0 \\
B : &\text{ update} \\
\end{align*}
\]
where $\{z_0, \ldots, z_{g-1}\} = \text{free}(e)$ and $\rho' = \{z_i \mapsto (C, i) \mid i = 0, \ldots, g - 1\}$. 

Idea:

- **Reuse** Global Vectors, i.e. share Global Vectors!
- Profitable in the translation of let-expressions or function applications: Build one Global Vector for the union of the free-variable sets of all let-definitions resp. all arguments.
- Allocate (references to) global vectors with multiple uses in the stack frame like local variables!
- Support the access to the current GP by an instruction `copyglob`:
• The optimization will cause Global Vectors to contain more components than just references to the free the variables that occur in one expression...

**Disadvantage:** Superfluous components in Global Vectors prevent the deallocation of already useless heap objects  
  
**Potential Remedy:** Deletion of references at the end of their life time.