18 Over- and Undersupply of Arguments

The first instruction to be executed when entering a function body, i.e., after an apply is $\text{targ } k$.

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 undersupply, a new F-object is returned.

The test for number of arguments uses: $\text{SP} - \text{FP}$

$\text{targ } k$ is a complex instruction.
We decompose its execution in the case of undersupply into several steps:

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

The combination of these steps into one instruction is a kind of optimization $\Rightarrow$
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 return k:

```c
return k = if (SP - FP = k + 1)
    popenv; // Done
else {
    // There are more arguments
    slide k;
    apply; // another application
}
```

The execution of return k results in:

19 let-rec-Expressions

Consider the expression \( e = \text{let rec } y_1 = e_1 \text{ and ... and } y_n = e_n \text{ in } e_0 \). The translation of \( e \) must deliver an instruction sequence that

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

Warning:

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

code\( e \rho \sigma d \sigma \) = alloc n \hspace{1em} // allocates local variables
code\( e_1 \rho' (sd + n) \)
rewrite n
...
code\( e_n \rho' (sd + n) \)
rewrite 1
code\( e_0 \rho' (sd + n) \)
slide n \hspace{1em} // deallocates local variables

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

In the case of CBV, we also use code\( e_0 \) for the expressions \( e_1, \ldots, e_n \).

Warning:
Recursive definitions of basic values are undefined with CBV!!!

19 \hspace{1em} let-rec-Expressions

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

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

- allocates local variables \( y_1, \ldots, y_n \);
- in the case of CBV: evaluates \( e_1, \ldots, e_n \) and binds the \( y_i \) to their values;
- in the case of 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.

Warning:

In a let-rec-expression, the definitions can use variables that will be allocated only later! \( \implies \) Dummy-values are put onto the stack before processing the definition.
The instruction `alloc n` reserves `n` cells on the stack and initialises them with `n` dummy nodes:

```
alloc n
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

for (i=1; i<=n; i++)
S[SP+i] = new (C, 1, 1);
SP = SP + n;

137

For CBN, we obtain:

```
code\_E \rho \_sd = alloc n  
  code\_E \, e_i \, \rho' \, (sd + n)  
  rewrite n  
  ...  
  code\_E \, e_n \, \rho' \, (sd + n)  
  rewrite 1  
  code\_E \, \epsilon_0 \, \rho' \, (sd + n)  
  slide n  
```

// allocates local variables

// deallocates local variables

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

In the case of CBV, we also use `code_E` for the expressions \( e_1, \ldots, e_n \).

Warning:
Recursive definitions of basic values are **undefined** with CBV!!!
For CBN, we obtain:

\[
\text{let } y_1 = x_1 \text{ in } \text{let } y_2 = x_2 \text{ end}
\]

\[\text{code} \ e \ \rho \ \sigma_1 \ \sigma_2 = \text{alloc} \ n \quad // \text{allocates local variables}
\]

\[\text{code} \ e \ \rho \ \sigma_1 \ \rho' \ (\sigma_2 + n) \text{ rewrite } n
\]

\[\ldots
\]

\[\text{code} \ e_3 \ \rho \ \sigma_1 \ \rho' \ (\sigma_2 + n) \text{ rewrite } 1
\]

\[\text{code} \ e_4 \ \rho' \ (\sigma_2 + n) \text{ slide } n \quad // \text{deallocates local variables}
\]

where \( \rho' = \rho \oplus \{ y_i \mapsto (L, \sigma_2 + i) \mid i = 1, \ldots, n \} \).

In the case of CBV, we also use code\(_v\) for the expressions \(e_1, \ldots, e_n\).

Warning:

Recursive definitions of basic values are undefined with CBV!!!
**eval** can be decomposed into small actions:

```c
    if \{ H[S[SP]] !\equiv \langle C, \_ \_ \_ \_ \_ \rangle \} {
        mark0;     // allocation of the stack frame
        pushloc 3; // copying of the reference
        apply0;    // 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 **mark A**, **mark0** dumps the current PC.
- The difference between **apply** and **apply0** is that no argument vector is put on the stack.

---

We thus obtain the instruction **eval**:

```c
    h = S[SP]; SP--;  
    GP = h->gp; PC = h->cp;
```

---

**eval** can be decomposed into small actions:

```c
    if \{ H[S[SP]] !\equiv \langle C, \_ \_ \_ \_ \_ \rangle \} {
        mark0;     // allocation of the stack frame
        pushloc 3; // copying of the reference
        apply0;    // 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 **mark A**, **mark0** dumps the current PC.
- The difference between **apply** and **apply0** is that no argument vector is put on the stack.
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 $e$:

$$
\text{codec} \ e \ \rho \ sd = \begin{cases} 
    \text{getvar} \ z_0 \ \rho \ sd \\
    \text{getvar} \ z_1 \ \rho \ (sd + 1) \\
    \ldots \\
    \text{getvar} \ z_{g-1} \ \rho \ (sd + g - 1) \\
    \text{mkvec} \ g \\
    \text{mkclso} \ A \\
    \text{jump} \ B \\
    A : \ \text{codec} \ e \ \rho' \ 0 \\
    \text{update} \\
    B : \ \ldots 
\end{cases}
$$

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

In fact, the instruction update is the combination of the two actions:

\begin{align*}
\text{popenv} \\
\text{rewrite} 1
\end{align*}

It overwrites the closure with the computed value.
The instruction `mkclos A` is analogous to the instruction `mkfunval A`.

It generates a C-object, where the included code pointer is `A`.

\[ S[SP] = \text{new} \ (C, A, S[SP]); \]

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
codec e ρ sd =
  getvar z_0 ρ sd
  getvar z_1 ρ (sd + 1)
  ...
  getvar z_{g-1} ρ (sd + \(g - 1\))
  mkvec g
  mkclos A
  jump B
A : codec e ρ' 0
  update
B : ...
```

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

Example:

Consider \( e \equiv a \ast a \) with \( ρ = \{a \mapsto (L, 0)\} \) and \( sd = 1 \). We obtain:

1. pushloc 1
2. mkvec 1
2. mkclos A
2. jump B

\[ \begin{array}{cccc}
1 & \begin{array}{c}
\begin{array}{c}
\text{pushloc} 0
\end{array}
\end{array}
2 & \begin{array}{c}
\begin{array}{c}
\text{getbasic}
\end{array}
\end{array}
2 & \begin{array}{c}
\begin{array}{c}
\text{mul}
\end{array}
\end{array}
\end{array} \]

In fact, the instruction `update` is the combination of the two actions:

- `popenv`
- `rewrite 1`

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

```
popenv
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 $e$...

```
ld y_1 = a + b in

\[\text{code}_{\rho, \varphi, \text{sd}} = \begin{cases} 
\text{getvar} \ z_1 \ (\varphi + 1) \\
\ldots \\
\text{getvar} \ z_{g-1} \ (\varphi + g - 1) \\
\text{mkvec} \ g \\
\text{mkclose} \ A \\
\text{jump} \ B \\
A : \text{codey} \ e \ \varphi \ 0 \\
\text{update} \\
B : \ldots \\
\end{cases} \]
```

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

### 22 Optimizations II: Closures

In some cases, the construction of closures can be avoided, namely for

- Basic values,
- Variables,
- Functions.

**Basic Values:**

The construction of a closure for the value is at least as expensive as the construction of the B-object itself!

Therefore:

\[
\text{codec} \ b \ \rho \ s \ d = \ \text{codec} \ y \ b \ \rho \ s \ d = \ \text{loadc} \ b \\
\text{mkbasic}
\]

This replaces:

\[
\text{mkvec} \ 0 \ \text{jump} \ B \ \text{mkbasic} \\
\text{mkcloes} \ A \ A: \ \text{loadc} \ b \ \text{update}
\]

**Variables:**

Variables are either bound to values or to C-objects. Constructing another closure is therefore superfluous. Therefore:

\[
\text{codec} \ x \ \rho \ s \ d = \ \text{getvar} \ x \ \rho \ s \ d
\]

This replaces:

\[
\text{getvar} \ x \ \rho \ s \ d \ \text{mkcloes} \ A \ A: \ \text{pushgrob} \ 0 \ \text{update} \\
\text{mkvec} \ 1 \ \text{jump} \ B \ \text{eval} \ B: \ ... \\
\]

**Example:**

\[
\text{let rec } a = b \text{ and } b = 7 \text{ in } a.
\]

\[
\text{codec} \ e 0 0
\]

produces:

\[
0 \ \text{alloc} \ 2 \ \\
2 \ \text{pushloc} \ 0 \ \\
3 \ \text{loadc} \ 7 \ \\
3 \ \text{mkbasic} \ 2 \ \text{pushloc} \ 1 \ \\
2 \ \text{loadc} \ 2 \ \\
3 \ \text{rewrite} \ 2 \ \\
3 \ \text{rewrite} \ 1 \ \\
3 \ \text{pushloc} \ 0 \ \\
3 \ \text{eval} \ \\
3 \ \text{slide} \ 2
\]
Variables:

Variables are either bound to values or to C-objects. Constructing another closure is therefore superfluous. Therefore:

\[ \text{codec } x \rho \text{ sd } = \text{getvar } x \rho \text{ sd} \]

This replaces:

\[ \text{getvar } x \rho \text{ sd } \quad \text{mkclos A} \quad A: \quad \text{pushglob 0} \quad \text{update} \]
\[ \text{mkvec 1} \quad \text{jump B} \quad \text{eval} \quad B: \quad \ldots \]

Example:

\[ e = \text{let rec } a = b \quad \text{and } b = 7 \text{ in } a. \]

produces:

\[ \text{codev } e \quad 0 \]

\[ 0 \quad \text{alloc 2} \quad 3 \quad \text{rewrite 2} \quad 3 \quad \text{mkbasic} \quad 2 \quad \text{pushloc 1} \]
\[ 2 \quad \text{pushloc 0} \quad 2 \quad \text{loadc 7} \quad 3 \quad \text{rewrite 1} \quad 3 \quad \text{eval} \]
\[ 3 \quad \text{slide 2} \]

\[ \text{alloc 2} \]

\[ \text{rewiite 2} \]

\[ \text{rewiite 2} \]

\[ \text{rewiite 2} \]
Apparently, this optimization was not quite correct :-(

The Problem:

Binding of variable \( y \) to variable \( x \) before \( x \)’s dummy node is replaced!!

The Solution:

cyclic definitions: reject sequences of definitions like
let \( a = b; . . . b = a \) in . . .

acyclic definitions: order the definitions \( y = x \) such that the dummy node for the right side of \( x \) is already overwritten.
Functions:
Functions are values, which are not evaluated further. Instead of generating code that constructs a closure for an F-object, we generate code that constructs the F-object directly.

Therefore:

\[
\text{code} \mathcal{C} (\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \rho \sigma d = \text{code}_\mathcal{Y} (\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \rho \sigma d
\]

23 The Translation of a Program Expression

Execution of a program \( \varepsilon \) starts with

\[
\text{PC} = 0 \quad \text{SP} = \text{FP} = \text{GP} = -1
\]

The expression \( \varepsilon \) must not contain free variables.
The value of \( \varepsilon \) should be determined and then a \textbf{halt} instruction should be executed.

\[
\text{code } \varepsilon = \text{code}_\mathcal{Y} \varepsilon 0 0
\]
\[
\text{halt}
\]
Remarks:

- The code schemata as defined so far produce Spaghetti code.
- Reason: Code for function bodies and closures placed directly behind the instructions mkfunval resp. mkclos with a jump over this code.
- Alternative: Place this code somewhere else, e.g. following the halt-instructions.
  
  **Advantage:** Elimination of the direct jumps following mkfunval and mkclos.
  
  **Disadvantage:** The code schemata are more complex as they would have to accumulate the code pieces in a Code-Dump.

Solution:

Disentangle the Spaghetti code in a subsequent optimization phase :-)