17 Function Application

Function applications correspond to function calls in C.
The necessary actions for the evaluation of $e' \ r_0 \ldots r_{m-1}$ 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,

A Slightly Larger Example

For CBV and $sd = 0$ we obtain:

```
0  load 17  2  jump  B  2  getbasic  5  load 42
1  mkbasic  0  A:  target  1  2  add  6  mkbasic
1  pushloc  0  0  pushglob  0  1  mkbasic  6  pushloc  4
2  mkvec  1  1  getbasic  1  return  7  apply
2  mkfuncall  A  1  pushloc  1  2  B:  mark C  3  C:  slide 2
```
For the implementation of the new instruction, we must fix the organization of a stack frame:

\[
\begin{align*}
\text{SP} & \quad \text{local stack} \\
\text{FP} & \quad 0 \\
\text{PCold} & \quad \text{FPold} \\
\text{GPold} & \quad -1 \quad -2 \\
\end{align*}
\]

Arguments

3 org. cells

Different from the CMs, the instruction \textit{mark A} already saves the return address:

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

The instruction \textit{apply} unpacks the F-object, a reference to which (hopefully) resides on top of the stack, and continues execution at the address given there:

\[
\begin{align*}
\text{GP} & \quad \text{PC} \\
\text{ap} & \quad \text{gp} \\
\text{[F: 42]} & \quad \text{apply} \\
\text{V1n} & \quad \text{GP} \quad 42 \\
\end{align*}
\]

\[
\begin{align*}
h & = \text{SP}; \\
\text{if } \{h[h] \neq (F, \ldots)\} & \quad \text{Error "no fun"}; \\
\text{else } & \quad \text{GP} = h \rightarrow \text{gp}; \text{PC} = h \rightarrow \text{cp}; \\
& \quad \text{for } \{i = 0; i < \text{ap} \rightarrow n; i++\} \\
& \quad \text{SP} = \text{SP} + \text{ap} \rightarrow n - 1;
\end{align*}
\]

Caveat

- 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.
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: \[ SP - FP \]

\[\text{targ } k \text{ is a complex instruction. We decompose its execution in the case of under-supply into several steps:}\]

\[\text{targ } k = \begin{cases} \text{mkvec0;} & \text{// creating the argument vector} \\ \text{wrap;} & \text{// wrapping into an F-object} \\ \text{popenv;} & \text{// popping the stack frame} \end{cases}\]

The combination of these steps into one instruction is a kind of optimization.

The instruction `mkvec0` takes all references from the stack above `FP` and stores them into a vector:

\[ g = \text{SP} - \text{FP}; \]
\[ h = \text{new } (V, g); \]
\[ \text{sp} = \text{FP} + 1; \]
\[ \text{for }(i=0; i<g; i++) \]
\[ h->v[i] = S[\text{SP} + i]; \]
\[ S[\text{SP}] = h; \]

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

\[ S[\text{SP}] = \text{new } (F, \text{PC-1}, S[\text{SP}], \text{GP}); \]
The instruction popen finally releases the stack frame:

\[ \text{GP} = \text{S[FP-2]}; \]
\[ \text{S[FP-2]} = \text{S[SP]}; \]
\[ \text{PC} = \text{S[FP]}; \]
\[ \text{SP} = \text{FP} - 2; \]
\[ \text{FP} = \text{S[FP-1]}; \]
• 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:

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

The execution of return k results in:

155

Case: Over-supply

156

19 let-rec-Expressions

Consider the expression \( e \equiv \text{let rec } y_1 = e_1 \text{ and } \ldots \text{ and } y_n = e_n \text{ in } \tau_0 \).

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 \( \tau_0 \) and returns its value.

Caveat

In a \text{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:
\[
\begin{align*}
\text{let } \mathbf{f} & = \text{fun } x \to f x \\
\text{code}_{\rho} e @ p s d & = \text{alloc } n & // \text{allocates local variables} \\
\text{code}_{\rho} e_1 \rho' (s d + n) & \\
\text{rewrite } n & \\
\text{code}_{\rho} e_2 \rho' (s d + n) & \\
\text{slide } n & // \text{deallocates local variables}
\end{align*}
\]

where \( \rho' = \rho \oplus \{ y_i \mapsto (\text{Loc } s d + i) \mid i = 1, \ldots, n \} \).

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

Caveat
Recursive definitions of basic values are undefined with CBV!!!
20 Closures and their Evaluation

- Closures are needed in the implementation of CBN for let-, let-rec expressions as well as for actual parameters of functions.
- 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:

```plaintext
eval = if (H[S[SP]] ≡ (C, _, _)) {
  mark0();  // allocation of the stack frame
  copyrefc 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 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.

S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

We thus obtain for the instruction `eval`:
Different from the CMa, the instruction `mark A` already saves the return address:

\[
\text{S}[\text{SP}+1] = \text{GP}; \\
\text{S}[\text{SP}+2] = \text{FP}; \\
\text{S}[\text{SP}+3] = \text{PC}; \\
\text{FP} = \text{SP} = \text{SP} + 3;
\]

We thus obtain for the instruction `eval`:

\[
h = \text{S}[\text{SP}]; \text{SP}--; \\
\text{GP} = h \rightarrow \text{gp}; \text{PC} = h \rightarrow \text{cp};
\]
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 $e$:

$$
\text{code}_C \in \rho \cdot \text{sd} = \begin{cases} 
\text{getvar } z_0 \in \rho \cdot \text{sd} \\
\text{getvar } z_1 \in \rho \cdot (\text{sd} + 1) \\
\vdots \\
\text{getvar } z_{g-1} \in \rho \cdot (\text{sd} + g - 1) \\
\text{mkvec } g \\
\text{mkclos } A \\
\text{jump } B \\
A: \quad \text{code}_V \in \rho' \cdot 0 \\
\text{update} \\
B: \quad \ldots
\end{cases}
$$

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

Example

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

1. pushloc $1$  \hspace{1em} 0 \hspace{1em} A: \hspace{0.5em} pushglob $0$  \hspace{1em} 2 \hspace{1em} getbasic
2. mkvec $1$  \hspace{1em} 1 \hspace{1em} eval  \hspace{1em} 2 \hspace{1em} mul
2. mkclos $A$  \hspace{1em} 1 \hspace{1em} getbasic  \hspace{1em} 1 \hspace{1em} mkbasic
2. jump $B$  \hspace{1em} 1 \hspace{1em} pushglob $0$  \hspace{1em} 1 \hspace{1em} update
2. \hspace{0.5em} \text{eval}  \hspace{1em} 2 \hspace{1em} B: \hspace{0.5em} \ldots

- The instruction $\text{mkclos } A$ is analogous to the instruction $\text{mkfunval } A$.
- It generates a C-object, where the included code pointer is $A$.

\[ S[\text{SP}] = \text{new } (C, A, S[\text{SP}]); \]
In fact, the instruction \textit{update} is the combination of the two actions:

\begin{itemize}
  \item \texttt{popenv}
  \item \texttt{rewrite 1}
\end{itemize}

It overwrites the closure with the computed value.