Actions when entering \( g \):
1. Evaluating the actual parameters \( \text{mark} \)
2. Saving \( FP, EP \) \( \text{call} \)
3. Determining the start address of \( g \) \( \text{enter} \)
4. Setting of the new \( FP \) \( \text{alloc} \)
5. Saving \( PC \) and
   Jump to the beginning of \( g \)
6. Setting of new \( EP \)
7. Allocating of local variables

Actions when terminating the call:

1. Storing of the return value \( \text{return} \)
2. Restoring of the registers \( FP, EP \) \( \text{slide} \)
3. Jumping back into the code of \( f \), i.e.,
   Restoration of the \( PC \)
4. Popping the stack

Accordingly, we obtain for a call to a function with at least one parameter and one return value:

\[
\text{code} g(e_1, \ldots, e_n) \rho = \text{code} e_n \rho \quad \text{mark}
\]
\[
\vdots
\]
\[
\text{code} e_1 \rho \quad \text{mark}
\]
\[
\text{code} g \rho \quad \text{call}
\]
\[
\text{slide} (m - 1)
\]

where \( m \) is the size of the actual parameters.
• Similar to declared arrays, function names are interpreted as constant pointers onto function code. Thus, the R-value of this pointer is the start address of the function.

  • Caveat! For a variable \( \text{int } (*)() \ g \): the two calls
    \[
    (g)() \quad \text{and} \quad g()
    \]
    are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant.

• During passing of parameters, these are copied.

Consequently,

\[
\begin{align*}
code_k f \rho & = \text{load}(\rho f) & \text{f name of a function} \\
code_k (\ast e) \rho & = \text{code}_k e \rho & \text{e function pointer} \\
code_k e \rho & = \text{code}_k e \rho \\
& \quad \text{move } k & \text{e a structure of size } k
\end{align*}
\]

where

Remark

• Of every expression which is passed as a parameter, we determine the R-value \( \rightarrow \) call-by-value passing of parameters.

• The function \( g \) may as well be denoted by an expression, whose R-value provides the start address of the called function ...
Remark

- Of every expression which is passed as a parameter, we determine the R-value.
- The function \( g \) may as well be denoted by an expression, whose R-value provides
  the start address of the called function ... 

The instruction **mark** saves the registers FP and EP onto the stack.

\[
S[SP+1] = EP; \\
S[SP+2] = FP; \\
SP = SP + 2; 
\]

The instruction **call** saves the return address and sets FP and PC onto the new values.

\[
tmp = S[SP]; \\
S[SP] = PC; \\
FP = SP; \\
PC = tmp; 
\]

The instruction **slide** copies the return values into the correct memory cell:

\[
tmp = S[SP]; \\
SP = SP-m; \\
S[SP] = tmp; 
\]
Accordingly, we translate a function definition:

```plaintext
code f (specs) \{ V_def \} \ p = _f
code \{ \}
if enter \(q\) // initialize EP
alloc \(c\) // allocate the local variables
code \(s\) \(\rho_f\)
return // return from call
```

where
- \(q = \text{max} \cdot k\) with
- \(\text{max}\) = maximal length of the local stack
- \(k\) = size of the local variables
- \(\rho_f\) = address environment for \(f\)

// takes specs, V_def \ and \(p\) into account

The instruction `enter q` sets the EP to the new value. If not enough space is available, program execution terminates.

The instruction `alloc k` allocates memory for locals on the stack.

The instruction `return` pops the current stack frame. This means it restores the registers PC, EP and FP and returns the return value on top of the stack.

\[ \text{PC} = S[FP], \text{EP} = S[FP - 2]; \]
\[ \text{if} (\text{EP} \geq \text{NP}) \, \text{Error} ("Stack Overflow"); \]
\[ \text{SP} = \text{FP} - 3, \text{FP} = S[\text{SP} + 2]; \]
9.4 Access to Variables, Formal Parameters and Returning of Values

Accesses to local variables or formal parameters are relative to the current FP. Accordingly, we modify $\text{code}_L$ for names of variables.

For $p x = \langle \text{tag}, j \rangle$, we define

$$
\text{code}_L x \rho = \begin{cases} 
\text{loadc}_j & \text{tag} = G \\
\text{load}_j & \text{tag} = L
\end{cases}
$$

As an optimization, we introduce analogously to $\text{load}_j$ and $\text{store}_j$ the new instructions $\text{loadr}_j$ and $\text{storer}_j$:

$$
\begin{align*}
\text{loadr}_j &= \text{loadc}_j \\
\text{storer}_j &= \text{loadc}_j
\end{align*}
$$

The instruction $\text{loadc}_j$ computes the sum of $FP$ and $j$.

$$
\begin{align*}
\text{FP} &\quad \text{loadc}_j \quad \text{FP} \\
\text{SP} &\quad \text{SP}+; \quad S[\text{SP}] = \text{FP}+j
\end{align*}
$$

The code for $\text{return } e; \rho$ corresponds to an assignment to a variable with relative address $-3$.

$$
\begin{align*}
\text{code } \text{return } e; \rho &= \text{code}_L e \rho \\
\text{storer } -3 &\\
\text{return }
\end{align*}
$$

Example: For function

```c
int fac (int x) {
if (x <= 0) return 1;
else return x + fac (x - 1);
}
```

we generate:
10 Translation of Whole Programs

Before program execution, we have:

\[ \text{SP} = -1 \quad \text{FP} = \text{EP} = -1 \quad \text{PC} = 0 \quad \text{NP} = \text{MAX} \]

Let \( p = V_{\text{def}}, F_{\text{def}_1}, \ldots, F_{\text{def}_n} \) denote a program where \( F_{\text{def}_i} \) is the definition of a function \( f_i \) of which one is called \( \text{main} \).

The code for the program \( p \) consists of:

- code for the function definitions \( F_{\text{def}_i} \);
- code for the allocation of global variables;
- code for the call of \( \text{int main}() \);
- the instruction \( \text{halt} \) which returns control to the operating system together with the value at address 0.

Then we define:

\[
\begin{align*}
\text{code } p &\equiv \text{ enter } (k + 4) \\
& \quad \text{ alloc } (k + 1) \\
& \quad \text{ mark} \\
& \quad \text{ load } \_\text{main} \\
& \quad \text{ call} \\
& \quad \text{ slide } k \\
& \quad \text{ halt} \\
& \quad \text{ } F_{\text{def}_1} \rho \\
& \quad \vdots \\
& \quad \text{ } F_{\text{def}_n} \rho
\end{align*}
\]

where \( \emptyset \cong \text{ empty address environment} \);

\( \rho \cong \text{ global address environment} \);

\( k \cong \text{ size of the global variables} \)
11 The language PuF

We only regard a mini-language PuF ("Pure Functions").

We do not treat, as yet:
- Side effects;
- Data structures;
- Exceptions.

Example

The following well-known function computes the factorial of a natural number:

\[
\text{let rec } \text{fac } \text{=} \text{fun } x \text{=} \begin{cases} 
    1 & \text{if } x \leq 1 \\
    x \cdot \text{fac } (x-1) & \text{else}
\end{cases}
\]

in fac 7

As usual, we only use the minimal amount of parentheses.

There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

A program is an expression \( e \) of the form:

\[
e \; ::= \; b \; | \; x \; | \; (\text{let } e) \; | \; (e_1 \; e_2)
\]

\[
| \; (\text{if } e_3 \; \text{then } e_1 \; \text{else } e_2)
\]

\[
| \; (e_1 \; e_0 \ldots e_{n-1})
\]

\[
| \; (\text{fun } x_0 \ldots x_{n-1} \rightarrow e)
\]

\[
| \; (\text{let } x_1 = e_1 \text{ in } e_0)
\]

\[
| \; (\text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_n = e_n \text{ in } e_0)
\]

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a let-expression, i.e. an expression with \textit{locally defined variables}, or
- a let-rec-expression, i.e. an expression with \textit{simultaneously defined} local variables.

For simplicity, we only allow \texttt{int} as basic type.

A program is an expression \( e \) of the form:

\[
e \; ::= \; b \; | \; x \; | \; (\text{let } e) \; | \; (e_1 \; e_2)
\]

\[
| \; (\text{if } e_3 \; \text{then } e_1 \; \text{else } e_2)
\]

\[
| \; (e_1 \; e_0 \ldots e_{n-1})
\]

\[
| \; (\text{fun } x_0 \ldots x_{n-1} \rightarrow e)
\]

\[
| \; (\text{let } x_1 = e_1 \text{ in } e_0)
\]

\[
| \; (\text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_n = e_n \text{ in } e_0)
\]

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a let-expression, i.e. an expression with \textit{locally defined variables}, or
- a let-rec-expression, i.e. an expression with \textit{simultaneously defined} local variables.

For simplicity, we only allow \texttt{int} as basic type.
Example
The following well-known function computes the factorial of a natural number:

\[
\text{let rec fac} = \begin{cases} 
\text{fun} \ x & \rightarrow \ \text{if} \ x \leq 1 \ \text{then} \ 1 \\
\text{else} \ x \cdot \text{fac} \ (x - 1) 
\end{cases} 
\text{in fac} 7
\]

As usual, we only use the minimal amount of parentheses.

There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

CBN: Arguments are passed un-evaluated; they are only evaluated when their value is needed (as in Haskell).

12 Architecture of the MaMa

We know already the following components:

\[
\begin{array}{c}
\text{C} \\
\text{0} \quad 1 \\
\text{PC}
\end{array}
\]

\[
\begin{array}{c}
\text{S} \\
\text{SP} \\
\text{FP}
\end{array}
\]

\[
\begin{array}{c}
\text{S} = \text{Runtime-Stack} - \text{each cell can hold a basic value or an address;}
\text{SP} = \text{Stack-Pointer} - \text{points to the topmost occupied cell;}
\text{as in the CMa implicitly represented;}
\text{FP} = \text{Frame-Pointer} - \text{points to the actual stack frame.}
\end{array}
\]
We also need a heap $H$:

\[ \text{Basic Value} \]

\[ \text{Closure} \]

\[ \text{Function} \]

\[ \text{Vector} \]

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

\[ v \]

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 $e$:

- `$\text{codev} e$` — (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);
- `$\text{codeg} e$` — computes the value of $e$, and returns it on the top of the stack (only for Basic types);
- `$\text{codec} e$` — 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:
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 \( e \) —** (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);
- **codep \( e \) —** computes the value of \( e \), and returns it on the top of the stack (only for Elasic types);
- **codec \( e \) —** 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: