A program is an expression $e$ of the form:

$$
e ::= b | x | (e_1 e_2) | (\langle e_1 e_2 \rangle)
  | (\text{if } e_0 \text{ then } e_1 \text{ else } e_2)
  | (\text{if } e_0 \ldots e_{k-1})
  | (\text{fun } x_0 \ldots x_{n-1} \rightarrow e)
  | (\text{let } x_1 = e_1 \text{ in } e_2)
  | (\text{let rec } x_1 = e_1 \text{ and } \ldots \text{and } x_n = e_n \text{ in } e_3)
$$

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 locally defined variables, or
- a let-rec-expression, i.e. an expression with simultaneously defined local variables.

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

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:

```plaintext
let rec fac = fun x -> if x <= 1 then 1
  else x * fac (x - 1)
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).
S = Runtime-Stack – each cell can hold a basic value or an address;
SP = Stack-Pointer – points to the topmost occupied cell;
as in the CMA implicitly represented;
FP = Frame-Pointer – points to the actual stack frame.

We also need a heap H:

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

v

Basic Value

C

Closure

F

Function

v[0] ...... v[n-1]

Vector

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

v

Basic Value

C

Closure

F

Function

v[0] ...... v[n-1]

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

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

- \(\text{code}_v \ e\) — (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);
- \(\text{code}_g \ e\) — computes the value of \(e\), and returns it on the top of the stack (only for Basic types);
- \(\text{code}_c \ e\) — 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:

\[
\begin{align*}
\text{code}_g (\text{if } c_0 \text{ then } c_1 \text{ else } c_2) \rho sd & = \text{code}_g c_0 \rho sd \\
& \quad \text{jumpz } A \\
& \quad \text{code}_g c_1 \rho sd \\
& \quad \text{jump } B \\
& \quad \text{\textcolor{red}{A: code}_g c_2 \rho sd} \\
& \quad \text{\textcolor{red}{B: \ldots}}
\end{align*}
\]

\[
\begin{align*}
\text{code}_g b \rho sd & = \text{load } b \\
\text{code}_g (\text{op}_1 \ e) \rho sd & = \text{code}_g e \rho sd \\
& \quad \text{op}_1 \\
\text{code}_g (c_1 \text{ op } c_2) \rho sd & = \text{code}_g c_1 \rho sd \\
& \quad \text{\textcolor{red}{code}_g c_2 \rho (sd \op )} \\
& \quad \text{op}_2
\end{align*}
\]

### 13 Simple expressions

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

- \(\text{rho}\) denotes the actual address environment, in which the expression is translated.
- The extra argument \(sd\), 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 \(\oplus\) and \(\otimes\), in the same way as the the operators \(\neg\) and \(\text{add}\) implement negation resp. addition in the CMA.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

\[
\begin{align*}
\text{code}_g e \rho sd & = \text{code}_v e \rho sd \\
& \quad \text{getbasic}
\end{align*}
\]

Note:
For code$_V$ and simple expressions, we define analogously:

\[
\begin{align*}
code_{V} (b \rho sd) &= \text{load } b; \text{mkbasic} \\
code_{V} (\langle \cdot \rangle \ e) \rho sd &= \text{code}_{V} e \rho sd \\
code_{V} (\cdot \square \ e_1 \ e_2) \rho sd &= \text{code}_{V} e_1 \rho sd \\
&\quad \text{code}_{V} e_2 \rho (sd + 1) \\
code_{V} (\text{if } e_0 \ \text{then } e_1 \ \text{else } e_2) \rho sd &= \text{code}_{V} e_0 \rho sd \\
&\quad \text{jump } A \\
&\quad \text{code}_{V} e_1 \rho sd \\
&\quad \text{jump } B \\
&\quad A: \text{code}_{V} e_2 \rho sd \\
&\quad B: \ldots
\end{align*}
\]

For code$_V$ and simple expressions, we define analogously:

\[
\begin{align*}
code_{V} (b \rho sd) &= \text{load } b; \text{mkbasic} \\
code_{V} (\langle \cdot \rangle \ e) \rho sd &= \text{code}_{V} e \rho sd \\
code_{V} (\cdot \square \ e_1 \ e_2) \rho sd &= \text{code}_{V} e_1 \rho sd \\
&\quad \text{code}_{V} e_2 \rho (sd + 1) \\
code_{V} (\text{if } e_0 \ \text{then } e_1 \ \text{else } e_2) \rho sd &= \text{code}_{V} e_0 \rho sd \\
&\quad \text{jump } A \\
&\quad \text{code}_{V} e_1 \rho sd \\
&\quad \text{jump } B \\
&\quad A: \text{code}_{V} e_2 \rho sd \\
&\quad B: \ldots
\end{align*}
\]
14 Accessing Variables

We must distinguish between local and global variables.

Example

```
let c = 5
in let f = fun a → let b = a + a
        in b + c
```

This function 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 binding), 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...

---

General form of the address environment:

```
ρ : Vars → {L, G} × Z
```
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 ...

\[ \rho : \text{Vars} \to \{ \text{L, C} \} \to \mathbb{Z} \]

Accessing Local Variables

Local variables are administered on the stack, in stack frames.
Let $e = e_0 \ldots e_{m-1}$ be the application of a function $e'$ to arguments $e_0, \ldots, e_{m-1}$.

Caveat:
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 $t$ is a functional type (over supply).

\[ m \geq n \]

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.
Possible stack organisations:

+ Addressing of the arguments can be done relative to FP
- The local variables of \( \varepsilon' \) cannot be addressed relative to FP.
- If \( \varepsilon' \) 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 \( a_0, \ldots, a_{k-1} \) and the local variables can be allocated above the arguments.

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

- Addressing of arguments and local variables relative to FP is no more possible. (Remember: \( m \) is unknown when the function definition is translated.)
Way out:

- We address both, arguments and local variables, relative to the stack pointer \( SP \).

- However, the stack pointer changes during program execution...

\[ \text{sd} \]

\[ \text{SP} \]

\[ \text{SP}_0 \]

\[ \text{fp} \]

\[ \ell_0 \]

\[ \ell_{m-1} \]

- 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., \( p \ x_i = (L_i - i) \).

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

```
  codeν x ρ sd = getvar x ρ sd
           eval
```

The instruction `eval` checks whether the value has already been computed or whether its evaluation has to yet to be done (will be treated later).

With CBV, we can just delete `eval` from the above code schema.

The (compile-time) macro `getvar` is defined by:

```
getvar x ρ sd = let  \( i, j = \rho x \) in
  match \( t \) with
    L \( \to \) pushloc (sd - i)
  | G \( \to \) pushglob i
  end
```

The access to local variables:

```
S[SP + 1] \to S[SP - n]; SP++;
```
The access to local variables:

\[ S[SP+1] = S[SP - n]; SP++ \]

Correctness argument:

Let \( sp \) and \( sd \) 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 \( S[a] \) with

\[ a = sp - (sd - i) = (sp - sd) + i = sp_v + i \]

... exactly as it should be \( :) \)

The access to global variables is much simpler:

\[ SP = SP + 1; S[SP] = GP \rightarrow v[i]; \]

Example

Regard \( e \equiv (b + c) \) for \( \rho = \left[ b \rightarrow \mathbb{D} \right] \), \( c \rightarrow \mathbb{D} \) and \( sd \rightarrow 1 \).

With CBN, we obtain:

\begin{align*}
\text{code} & = \text{getvar} \ e \rho 1 = \text{getvar} b \rho 1 \\text{eval} \\text{getbasic} \text{eval} \\text{getvar} c \rho 2 \\text{eval} \\text{getbasic} \\text{pushglob} 0 \\text{eval} \\text{add} \text{mkbasic} \\text{add} \text{mkbasic} \\text{add} \text{mkbasic}
\end{align*}
15 let-Expressions

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

Let $e = \text{let } y_1 = e_1 \text{ in } \ldots \text{ let } y_n = e_n \text{ in } e_0$ 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:

The instruction \textbf{slide $k$} deallocates again the space for the locals:

\[
\text{slide } k \quad \text{deallocates local variables:}
\]

\[
S[SFP-k] = S[SFP];
SFP = SFP + k_c
\]

\[
\text{code}_v e \rho sd = \text{code}_v e_1 \rho sd
\]
\[
\text{code}_v e_2 \rho_1 (sd + 1)
\]
\[
\ldots
\]
\[
\text{code}_v e_n \rho_{n-1} (sd + n - 1)
\]
\[
\text{code}_v e_n \rho_n (sd + n)
\]
\[
\text{slide } n \quad \text{// deallocates local variables}
\]

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

In the case of CBV, we use $\text{code}_v$ for the expressions $e_1, \ldots, e_n$.

**Caveat!**

All the $e_i$ must be associated with the same binding for the global variables!
Example

Consider the expression:
\[ e = \text{let } a = 19 \text{ in let } b = a * 34 \text{ at } a + b \]

for \( r \) = 0 and \( scd \) = 0. We obtain (for CBV):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Line 1</th>
<th>Line 2</th>
<th>Line 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 19</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>mkbasic</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>push loc 0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>get basic</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>push loc 1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>get basic</td>
<td>3</td>
<td>3</td>
<td>slide 2</td>
</tr>
</tbody>
</table>

The instruction \textit{slide k} deallocates again the space for the locals:

\[ S[SP-k] = S[SP]; \]
\[ SP = SP - k; \]