13 Simple expressions

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

\[
\begin{align*}
\text{code}_S \ b \ p \ sd &= \ \text{loadc} \ b \\
\text{code}_S (\alpha_1 \ e) \ p \ sd &= \ \text{code}_S e \ p \ \text{sd} \\
\text{code}_S (e_1 \ \alpha_2 \ e_2) \ p \ sd &= \ \text{code}_S e_1 \ p \ \text{sd} \\
& \quad \text{code}_S e_2 \ p \ \text{sd} + 1 \\
& \quad \text{op}_2
\end{align*}
\]
Remark

- $\rho$ 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 $op_1$ and $op_2$ implement the operators $\sqcap_1$ and $\sqcap_2$, in the same way as the the operators $\text{neg}$ and $\text{add}$ implement negation resp. addition in the $CM_2$.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer.

$$\text{code}_{\text{v}} \ e \ p \ sd = \text{code}_{\text{v}} \ e \ p \ sd$$

getbasic

For $\text{code}_{\text{v}}$ and simple expressions, we define analogously:

$$\text{code}_{\text{v}} \ b \ p \ sd = \text{load} \ b; \text{mkbasic}$$

$$\text{code}_{\text{v}} \ (\sqcap_1 \ e) \ p \ sd = \text{code}_{\text{v}} \ e \ p \ sd$$

$op_1; \text{mkbasic}$

$$\text{code}_{\text{v}} \ (e_1 \ sqcap_2 \ e_2) \ p \ sd = \text{code}_{\text{v}} \ e_1 \ p \ sd$$

$\text{code}_{\text{v}} \ e_2 \ p \ (sd + 1)$

$op_2; \text{mkbasic}$

$$\text{code}_{\text{v}} \ (\text{if } c_0 \ \text{then } e_1 \ \text{else } e_2) \ p \ sd = \text{code}_{\text{v}} \ e_0 \ p \ sd$$

$\text{jumpz} \ A$

$\text{code}_{\text{v}} \ e_1 \ p \ sd$

$\text{jump} \ B$

A: $\text{code}_{\text{v}} \ e_2 \ p \ sd$

B: ...
For codeTy and simple expressions, we define analogously:

\[
\begin{align*}
\text{codeTy } b \; p \; s \; d & = \text{load } b; \text{ mkbasic} \\
\text{codeTy } (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \; p \; s \; d & = \text{codeTy } e_2 \; p \; s \; d \\
& \quad \text{codeTy } e_3 \; (p + 1) \; s \; d \\
& \quad \text{op } x; \text{ mkbasic} \\
\text{codeTy } (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \; p \; s \; d & = \text{codeTy } e_2 \; p \; s \; d \\
& \quad \text{jump } A \\
& \quad \text{codeTy } e_3 \; p \; s \; d \\
& \quad \text{jump } B \\
A & \quad \text{codeTy } e_2 \; p \; s \; d \\
B & \quad \ldots
\end{align*}
\]

### 14 Accessing Variables

We must distinguish between local and global variables.

**Example**

Regard the function \( f \):

\[
\begin{align*}
\text{let } c & = 5 \\
\text{in } f & = \text{fun } a \rightarrow \text{let } b = a \times a \\
& \quad \text{in } b + c
\end{align*}
\]

The function \( f \) 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 constrast, local variables should be administered on the stack ...

\[
\text{General form of the address environment:} \\
\rho : \text{Vars} \rightarrow \{L, G\} \times Z
\]

### Accessing Local Variables

Local variables are administered on the stack, in stack frames.

Let \( c = (e_0, e_1, \ldots, e_{m-1}) \) be the application of a function \( c' \) to arguments \( e_0, \ldots, e_{m-1} \).

**Caveat**

The arity of \( c' \) does not need to be \( m \).
- \( f \) may therefore receive less than \( m \) arguments (under supply);
- \( f \) may also receive more than \( m \) arguments, if \( f \) is a functional type (over supply).
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.

Alternative

- The further arguments $a_2, \ldots, a_{i-1}$ and the local variables can be allocated above the arguments.

- If $e'$ evaluates to a function, which has already been partially applied to the parameters $a_2, \ldots, a_{i-1}$, these have to be sneaked in underneath $e_0$.

- 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...

The difference between the current value of SP and its value sp<sub>0</sub> 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 <i>x_0, x_1, x_2, ...</i> successively receive the non-positive relative addresses 0, −1, −2, ..., i.e.,<br>
\[ \rho x_i = (L - i). \]

The absolute address of the <i>i</i>-th formal parameter consequently is<br>
\[ sp_0 - i = (SP - sd) - i \]

The local let-variables <i>y_1, y_2, y_3, ...</i> will be successively pushed onto the stack:

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

\[ \text{code}_{\gamma} x \rho sd = \text{getvar} x \rho sd \]

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

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

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

\[ \text{getvar} x \rho sd = \text{let } \{ t, i = \rho x \text{ in} \]

- match <i>t</i> with
  - \[ L \rightarrow \text{pushloc} \ (sd - i) \]
  - \[ G \rightarrow \text{pushglob} i \]
- end
The access to local variables:

\[
\begin{align*}
S[SP+1] &= (SP - n) \times SP++;
\end{align*}
\]

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

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

... exactly as it should be.

The access to local variables:

\[
S[SP+1] = (SP - n) \times SP++;
\]

The access to global variables is much simpler:

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

\[
SP = SP + 1;
\]
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:

\[
\text{code}_\text{C} e_{\rho} \rho_{sd} = \text{code}_\text{C} e_1 \rho_{sd} \\
\text{code}_\text{C} e_2 \rho_i (sd + 1) \\
\ldots \\
\text{code}_\text{C} e_n \rho_{n-1} (sd + n - 1) \\
\text{code}_\text{C} e_0 \rho_n (sd + n) \\
\text{slide } n \quad // \text{deallocates local variables}
\]

where \( \rho_j = \rho \uplus \{ y_i \mapsto (L_s, sd + i) \mid i = 1, \ldots, j \} \).

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

Caveat!

All the \( e_i \) must be associated with the same binding for the global variables!

\[
S_\rho = S \\
S_{\rho_{i+1}} = S_i \uplus \{ q_i \mapsto (L_i, sd_i) \}
\]
Example

Consider the expression

\[ e \equiv \text{let } a = 19 \text{ in let } b = a * a \text{ in } a + b \]

for \( \rho = 0 \) and \( sd = 0 \). We obtain (for CBV):

<table>
<thead>
<tr>
<th>Stack</th>
<th></th>
<th>Code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>pushc</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>mkbas</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>pushc</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>getbas</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>pushc</td>
<td>1</td>
</tr>
</tbody>
</table>

\( \{ a \mapsto (\lambda \eta . g (\eta, \phi)) \} 1 \mapsto (\lambda (\eta, \phi)) \)

16 Function Definitions

The definition of a function \( f \) requires code that allocates a functional value for \( f \) in the heap. This happens in the following steps:

- Creation of a Global Vector with the binding of the free variables;
- Creation of an (initially empty) argument vector;
- Creation of an F-Object, containing references to theses vectors and the start address of the code for the body;

Separately, code for the body has to be generated.

Thus,

codev \((\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \rho \) sd

\[ = \text{getvar } z_0 \rho \text{ sd} \]

\[ \text{getvar } z_1 \rho (sd + 1) \]

\[ \ldots \]

\[ \text{getvar } z_{n-1} \rho (sd + g - 1) \]

\[ \text{mkvec } g \]

\[ \text{mkfunval } A \]

\[ \text{jump } B \]

A: \( \text{targ } k \)

\[ \text{codev } e \rho' 0 \]

\[ \text{return } k \]

B: \( \ldots \)

where \( \{ z_0, \ldots, z_{g-1} \} = \text{free(} \text{fun } x_0 \ldots x_{n-1} \rightarrow e) \)

and \( \rho' = \{ x_i \mapsto (L_i - i) \mid i = 0, \ldots, k - 1 \} \cup \{ z_i \mapsto (G_i) \mid j = 0, \ldots, g - 1 \} \)
16 Function Definitions

The definition of a function \( f \) requires code that allocates a functional value for \( f \) in the heap. This happens in the following steps:

- Creation of a Global Vector with the binding of the free variables;
- Creation of an (initially empty) argument vector;
- Creation of an F-Object, containing references to these vectors and the start address of the code for the body;

Separately, code for the body has to be generated.

Thus,
Example

Regard \( f \equiv \textbf{fun} \ b \to a + b \) for \( \rho = \{ a \mapsto (L, 1) \} \) and \( sd = 1 \).

code: \( f \rho 1 \) produces:

```
1  pushloc 0    0  pushglob 0    2  getbasic
2  mkvec 1    1  eval    2  add
2  mkfunval A    1  getbasic    1  mkbasic
2  jump B    1  pushloc 1    1  return 1
0  A:  targ 1    2  eval    2  B:  ...
```

The secrets around \( \text{tag} \ k \) and \( \text{return} \ k \) will be revealed later.