14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function $f$:

$$
\begin{align*}
&\text{let } c = 7 \\
&\text{in let } f = \text{fun } a \rightarrow \text{let } b = a \cdot a \\
&\text{in } \ f c
\end{align*}
$$

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 scoping), 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 E-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 ...

\[ \forall i \mapsto \{ \_ \} \]

\[ \{ f \} \]

General form of the address environment:

$$
\rho : \text{Vars} \rightarrow \{ L, G \} \times I
$$

Accessing Local Variables

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

Let $\epsilon = \epsilon' \cdot \epsilon_0 \ldots \epsilon_{m-1}$ be the application of a function $\epsilon'$ to arguments $\epsilon_0, \ldots, \epsilon_{m-1}$.

Warning:

- The arity of $\epsilon'$ 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 $f$ 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 $a_0, \ldots, a_{n-1}$ and the local variables can be allocated above the arguments.
Alternative:

+ The further arguments \( a_0, \ldots, a_{n-1} \) and the local variables can be allocated above the arguments.

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

\[ f = \sum_{i=5} \left( \hat{f} 5 \right) + \left( \hat{f} 5 \right) \]

Addressing of arguments and local variables relative to FP is no more possible. (Remember: \( n \) 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 \( sp_0 \) 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., \( \rho x_i = (L, -i) \).
- 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.
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 \( s_{p_0} \) at the entry of the function body is called the stack distance, \( s_d \).

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., \( \rho i = (L, -i) \).

The absolute address of the \( i \)-th formal parameter consequently is

\[
sp_0 - i = (SP - s_d) - i
\]

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:

\[
\text{code} \quad x \cdot \rho \cdot s_d = \text{getvar} x \cdot \rho \cdot s_d
\]

The instruction \text{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 \text{eval} from the above code schema.

The (compile-time) macro \text{getvar} is defined by:

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

\[
\text{match } t \text{ with }
\]

\[
L \rightarrow \text{pushloc } (s_d - i)
\]

\[
G \rightarrow \text{pushglob } i
\]

end
The access to local variables:

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

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

\[ a = sp - (sd - i) = (sp - sd) + i = sp_0 + i \]

... exactly as it should be  :)

The access to global variables is much simpler:

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

Example:

Regard \( \epsilon \equiv (b + c) \) for \( \rho = \{ \beta \mapsto (L, 1), c \mapsto (G, 0) \} \) and \( sd = 1 \).

With CBN, we obtain:

\[
\begin{align*}
\text{code}_{\epsilon} \rho 1 &= \text{getvar}_{\beta} \rho 1 = 1 \quad \text{pushloc} 0 \\
&\quad \text{eval} \\
&\quad \text{gobasic} \\
&\quad \text{getvar}_{\epsilon} \rho 2 = 2 \quad \text{pushglob} 0 \\
&\quad \text{eval} \\
&\quad \text{gobasic} \\
&\quad \text{add} \\
&\quad \text{mkbasic} 2 \\
\end{align*}
\]
15 let-Expressions

As a warm-up let us first consider the treatment of local variables. Let \( e = \text{let } y_1 = e_1 \text{ in...let } y_n = e_0 \) be a nested let-expression. The translation of \( e \) must deliver an instruction sequence that

- allocates local variables \( y_1, \ldots, y_n \);
- \( \text{in the case of } \)
  - 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 \( y_j \) only depends on \( y_1, \ldots, y_{j-1} \). We obtain for CBN:

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

\[
\text{codey} = \text{getvar} \quad \text{eval}
\]

The instruction \( \text{eval} \) checks, whether the value has already been computed or whether its evaluation has to yet to be done \( \quad (\quad \text{will be treated later}) \). With CBV, we can just delete \( \text{eval} \) from the above code schema. The (compile-time) macro \( \text{getvar} \) is defined by:

\[
\text{getvar} = \text{let } (i, i) = \rho x \text{ in}
\]

match \( r \) with

\[
\text{L} \rightarrow \text{pushloc} (\text{sd} - i)
\]

| \( G \rightarrow \text{pushglob} i \)

end

15 let-Expressions

As a warm-up let us first consider the treatment of local variables. Let \( e = \text{let } y_1 = e_1 \text{ in...let } y_n = e_0 \) be a nested let-expression. The translation of \( e \) must deliver an instruction sequence that

- allocates local variables \( y_1, \ldots, y_n \);
- \( \text{in the case of } \)
  - 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 \( y_j \) only depends on \( y_1, \ldots, y_{j-1} \). We obtain for CBN:
$s_1 = \{ a \rightarrow (L, 2) \}$

Example:

Consider the expression

$e \equiv \text{let } a = 19 \text{ in let } b = a \times a \text{ in } a + b$

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

- $\text{loadc } 19$
- $\text{getbasic}$
- $\text{pushloc } 1$
- $\text{mkbasic}$
- $\text{mul}$
- $\text{getbasic}$
- $\text{pushloc } 0$
- $\text{mkbasic}$
- $\text{add}$
- $\text{getbasic}$
- $\text{pushloc } 1$
- $\text{mkbasic}$
- $\text{slide } 2$

$\rho_1 = \{ a \rightarrow (L, 1) \}$

Example:

Consider the expression

$e \equiv \text{let } a = 19 \text{ in let } b = a \times a \text{ in } a + b$

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

- $\text{loadc } 19$
- $\text{getbasic}$
- $\text{pushloc } 1$
- $\text{mkbasic}$
- $\text{mul}$
- $\text{getbasic}$
- $\text{pushloc } 0$
- $\text{mkbasic}$
- $\text{add}$
- $\text{getbasic}$
- $\text{pushloc } 1$
- $\text{mkbasic}$
- $\text{slide } 2$

$\rho_2 = \{ a \rightarrow (L, 1), b \rightarrow (L, 2) \}$

Warning!

All the $e_i$ must be associated with the same binding for the global variables!
Example:
Consider the expression
\[ c = \text{let } a = 19 \text{ in let } b = a \times a \text{ in } a + b \]
for \(\rho = 0\) and \(sd = 0\). We obtain (for CBV):

\[
\begin{array}{cccc}
0 & \text{loadc } 19 & 3 & \text{getbasic} & 3 & \text{pushloc } 1 \\
1 & \text{mkbasic} & 3 & \text{mul} & 4 & \text{getbasic} \\
1 & \text{pushloc } 0 & 2 & \text{mkbasic} & 4 & \text{add} \\
2 & \text{getbasic} & 2 & \text{pushloc } 1 & 3 & \text{mkbasic} \\
2 & \text{pushloc } 1 & 3 & \text{getbasic} & 3 & \text{slide } 2 \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{slide } k \\
\end{array}
\]

Example:
Consider the expression
\[ c = \text{let } a = 19 \text{ in let } b = a \times a \text{ in } a + b \]
for \(\rho = 0\) and \(sd = 0\). We obtain (for CBV):

\[
\begin{array}{cccc}
0 & \text{loadc } 19 & 3 & \text{getbasic} & 3 & \text{pushloc } 1 \\
1 & \text{mkbasic} & 3 & \text{mul} & 4 & \text{getbasic} \\
1 & \text{pushloc } 0 & 2 & \text{mkbasic} & 4 & \text{add} \\
2 & \text{getbasic} & 2 & \text{pushloc } 1 & 3 & \text{mkbasic} \\
2 & \text{pushloc } 1 & 3 & \text{getbasic} & 3 & \text{slide } 2 \\
\end{array}
\]
The instruction `slide k` deallocates again the space for the locals:

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

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 \left( \text{fun} \quad \overline{x_0, \ldots, x_{k-1} \rightarrow e} \right) \rho sd =
\]

where \( \{x_0, \ldots, x_{k-1}\} = \text{free}(\text{fun} \quad \overline{x_0, \ldots, x_{k-1} \rightarrow e}) \)

and \( \rho' = \{x_0 \mapsto (E_0), \ldots, x_{k-1} \mapsto (E_{k-1}) \} \cup \{E_i \mapsto (G,E_i) \} \quad j = 0, \ldots, g - 1 \)
\begin{align*}
\text{codev (fun } x_0 \ldots x_{k-1} \rightarrow e) \rho \text{ sd} &= \text{getvar } z_0 \rho \text{ sd} \tag*{136}
\text{getvar } z_1 \rho \text{ (sd + 1)} \\
\ldots \\
\text{getvar } z_{k-1} \rho \text{ (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
\end{align*}

where \( \{z_0, \ldots, z_{k-1}\} = \text{free(fun } x_0 \ldots x_{k-1} \rightarrow e) \)
and \( \rho' = \{ x_j \mapsto (L, i) \mid i = 0, \ldots, k - 1 \} \cup \{ z_j \mapsto (G, j) \mid j = 0, \ldots, g - 1 \} \)
Example:

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

code: \( f \rho \) produces:

1. pushloc 0
2. mkvec 1
3. mkfunval A
4. jump B
5. A : targ 1

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

17 Function Application

Function applications correspond to function calls in C.

The necessary actions for the evaluation of \( \phi \in C_1 \ldots C_{n-1} \) 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 \( \phi' \) to an F-object;
- Application of the function.

Thus for CBV:
\[
\text{code} (\rho^e e_0 \ldots e_{n-1}, 1) \rho \text{sd} = \begin{cases} \\
A \\
\text{mark } A \\
\text{code } e_{n-1} \rho (s_{d+3}) \\
\text{code } e_{n-2} \rho (s_{d+4}) \\
\vdots \\
\text{code } e_0 \rho (s_{d+m+2}) \\
\text{code } e' \rho (s_{d+m+3}) \end{cases} // \text{Allocation of the frame}
\]

\[
\text{apply } A : \ldots \]

// \text{Evaluation of } e'

// corresponds to call

To implement CBV, we use code instead of code for the arguments \(e_i\).

**Example:** \(f(42), \rho = \{(f, L, 2)\}\) and \(s_d = 2\); we obtain with CBV:

\[
\begin{array}{cccccc}
2 & \text{mark } A & 6 & \text{mkbasic} & 7 & \text{apply} \\
5 & \text{load } 42 & 6 & \text{push } 4 & 3 & A : \ldots \\
& & 6 - 2 & & &
\end{array}
\]

A Slightly Larger Example:

let \(a = 17\) in let \(f = \text{fun } b \rightarrow a + b\) in \(f\) \(42\)

For CBV and \(s_d = 0\) we obtain:

\[
\begin{array}{cccccc}
0 & \text{load } 17 & 2 & \text{jump } B & 2 & \text{getbasic} & 5 & \text{load } 42 \\
1 & \text{mkbasic} & 0 & A : \text{tag } 1 & 2 & \text{add} & 5 & \text{mkbasic} \\
1 & \text{pushloc } 0 & 0 & \text{pushglob } 0 & 1 & \text{mkbasic} & 6 & \text{pushloc } 4 \\
2 & \text{mkvec } 1 & 1 & \text{getbasic} & 1 & \text{return } 1 & 7 & \text{apply} \\
2 & \text{mkfunval } A & 1 & \text{pushloc } 1 & 2 & B & \text{mark } C & 3 & C : \text{slide } 2
\end{array}
\]

For the implementation of the new instruction, we must fix the organization of a stack frame.
Different from the CMz, the instruction **mark A** already saves the return address:

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

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

\[
h = S[SP]; \\
\text{if \( (H[h] \rightarrow (f_{\infty})) \)} \\
\text{Error "no fun";} \\
\text{else} \\
\]

\[
S[SP+1] = h \rightarrow ap \rightarrow v[i]; \\
SP = SP + h \rightarrow ap \rightarrow n - 1;
\]

---

Warning:

- 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`