Script generated by TTT

Title: Seidl: Virtual_Machines (05.06.2012)

Date: Tue Jun 05 14:03:13 CEST 2012

Duration: 88:20 min

Pages: 41

Example:

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

$$\begin{array}{lll} \textbf{letrec}\ \text{fac} & = & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\ \textbf{in}\ \text{fac}\ 7 & & \end{array}$$

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).

Feed

Example:

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

$$\begin{array}{lll} \textbf{letrec}\ \text{fac} & = & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\ \textbf{in}\ \text{fac}\ 7 & & \end{array}$$

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

There are two Semantics:

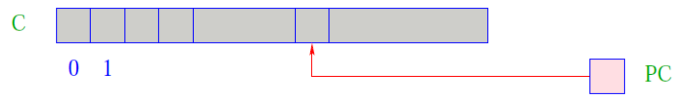
CBV: Arguments are evaluated before they are passed to the function (as in SML);

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

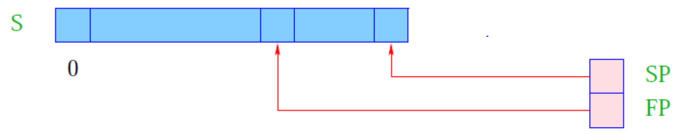
---

12 Architecture of the MaMa:

We know already the following components:

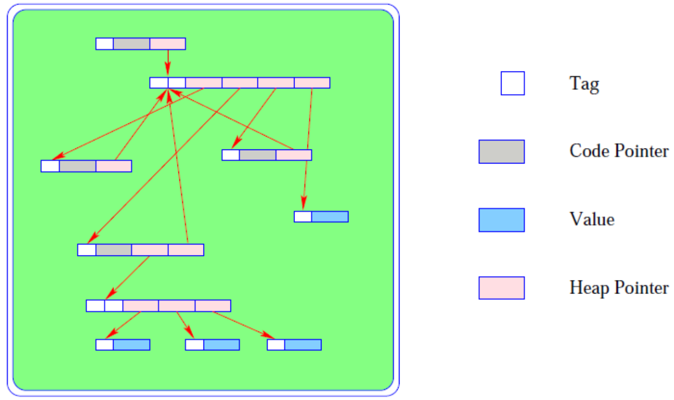


C	=	Code-store – contains the MaMa-program; each cell contains one instruction;
PC	=	Program Counter – points to the instruction to be executed next;

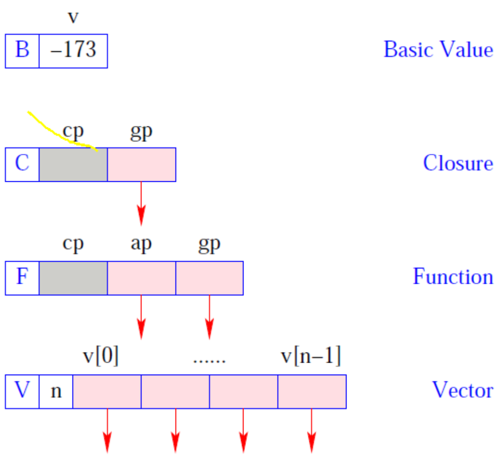


S	=	Runtime-Stack – each cell can hold a basic value or an address;
SP	=	Stack-Pointer – points to the topmost occupied cell; as in the CMa implicitely represented;
FP	=	Frame-Pointer – points to the actual stack frame.

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:



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:

- $\text{code}_V\ e$ — (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);
- $\text{code}_B\ e$ — 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:

13 Simple expressions

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

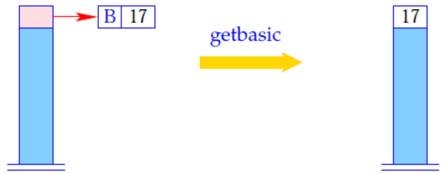
$$
\begin{array}{lcl}
\mathrm{code}_B\; b\; \rho\; \mathrm{sd} & = & \mathrm{loadc}\; b \\
\mathrm{code}_B\; (\square_1\; e)\; \rho\; \mathrm{sd} & = & \mathrm{code}_B\; e\; \rho\; \mathrm{sd} \\
 & & \mathrm{op}_1 \\
\mathrm{code}_B\; (e_1\; \square_2\; e_2)\; \rho\; \mathrm{sd} & = & \mathrm{code}_B\; e_1\; \rho\; \mathrm{sd} \\
 & & \mathrm{code}_B\; e_2\; \rho\; (\mathrm{sd}+1) \\
 & & \mathrm{op}_2
\end{array}
$$

$$
\begin{array}{lcrl}
\mathrm{code}_B\; (\mathbf{if}\; e_0\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e_0\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jumpz}\; A \\
 & & & \mathrm{code}_B\; e_1\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jump}\; B \\
 & & A: & \mathrm{code}_B\; e_2\; \rho\; \mathrm{sd} \\
 & & B: & \ldots
\end{array}
$$

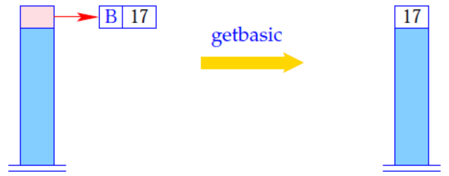
Note:

- ρ 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 $\square_1$ and $\square_2$, in the same way as the the operators neg and 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:

$$
\begin{array}{lcl}
\mathrm{code}_B\; e\; \rho\; \mathrm{sd} & = & \mathrm{code}_V\; e\; \rho\; \mathrm{sd} \\
 & & \mathrm{getbasic}
\end{array}
$$



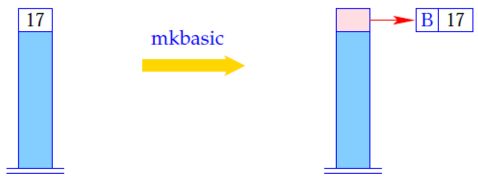

```
if (H[S[SP]] != (B,_))
     Error "not basic!";
else
     S[SP] = H[S[SP]].v;
```




```
if (H[S[SP]] != (B,_))
     Error "not basic!";
else
     S[SP] = H[S[SP]].v;
```


For code_V and simple expressions, we define analogously:

$$\begin{array}{lcll}
\mathrm{code}_V\; b\; \rho\; \mathrm{sd} & = & & \mathrm{loadc}\; b;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (\Box_1\; e)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e\; \rho\; \mathrm{sd} \\
 & & & \mathrm{op}_1;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (e_1\; \Box_2\; e_2)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e_1\; \rho\; \mathrm{sd} \\
 & & & \mathrm{code}_B\; e_2\; \rho\; (\mathrm{sd}+1) \\
 & & & \mathrm{op}_2;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (\mathbf{if}\; e_0\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e_0\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jumpz}\; A \\
 & & & \mathrm{code}_V\; e_1\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jump}\; B \\
 & & A: & \mathrm{code}_V\; e_2\; \rho\; \mathrm{sd} \\
 & & B: & \ldots
\end{array}$$

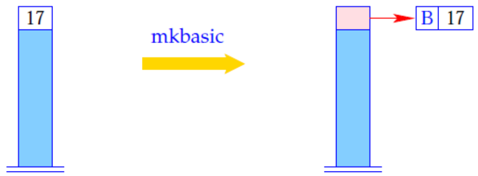



```
S[SP] = new (B,S[SP]);
```


For code_V and simple expressions, we define analogously:

$$\begin{array}{lcll}
\mathrm{code}_V\; b\; \rho\; \mathrm{sd} & = & & \mathrm{loadc}\; b;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (\Box_1\; e)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e\; \rho\; \mathrm{sd} \\
 & & & \mathrm{op}_1;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (e_1\; \Box_2\; e_2)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e_1\; \rho\; \mathrm{sd} \\
 & & & \mathrm{code}_B\; e_2\; \rho\; (\mathrm{sd}+1) \\
 & & & \mathrm{op}_2;\ \mathrm{mkbasic} \\
\mathrm{code}_V\; (\mathbf{if}\; e_0\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2)\; \rho\; \mathrm{sd} & = & & \mathrm{code}_B\; e_0\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jumpz}\; A \\
 & & & \mathrm{code}_V\; e_1\; \rho\; \mathrm{sd} \\
 & & & \mathrm{jump}\; B \\
 & & A: & \mathrm{code}_V\; e_2\; \rho\; \mathrm{sd} \\
 & & B: & \ldots
\end{array}$$

107



S[SP] = new (B,S[SP]);

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

$$\begin{array}{lll} \textbf{let} & c = 5 & \\ & f = \textbf{fn}\ a & \Rightarrow \ \textbf{let}\ b = a * a \\ & & \quad\ \textbf{in}\ b + c \\ \textbf{in} & f\ c & \end{array}$$

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 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 -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 ...

$\Longrightarrow$ General form of the address environment:

$$\rho : Vars \to \{L, G\} \times \mathbb{Z}$$

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

c

$$\begin{array}{lll} \textbf{let} & c = 5 & \\ & f = \textbf{fn}\ a & \Rightarrow \ \textbf{let}\ b = a * a \\ & & \quad\ \textbf{in}\ b + c \\ \textbf{in} & f\ c & \end{array}$$

15

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 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 -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 ...

$\Longrightarrow$ General form of the address environment:

$$\rho : Vars \to \{L, G\} \times \mathbb{Z}$$

Accessing Local Variables

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

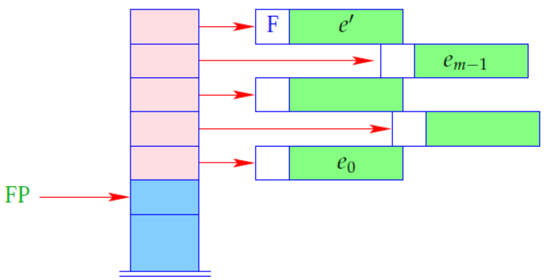
Let $e \equiv e' \boxed{e_0 \; \ldots \; e_{m-1}}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

The arity of e' does not need to be m :-)

- f may therefore receive less than n arguments $\boxed{\text{(under supply);}}$
- f may also receive more than n arguments, if t 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.

Accessing Local Variables

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

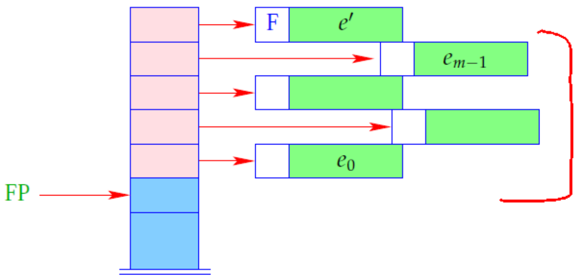
Let $e \equiv e' \; e_0 \; \ldots \; e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

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).

Possible stack organisations:

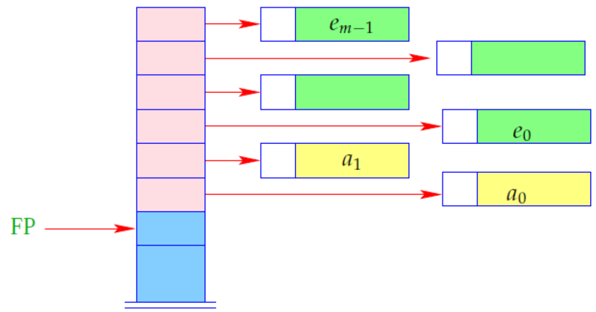


+ Addressing of the arguments can be done relative to FP

– 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.

– If e' 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 e_0:



Accessing Local Variables

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

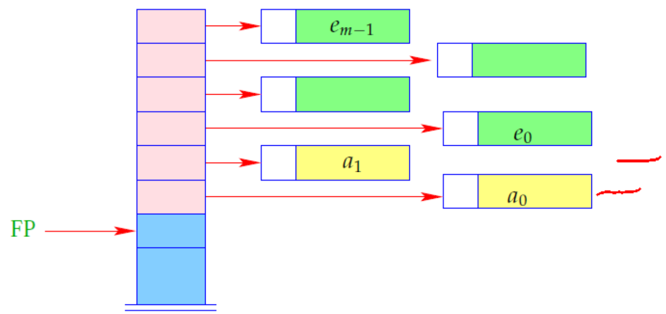
Let $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

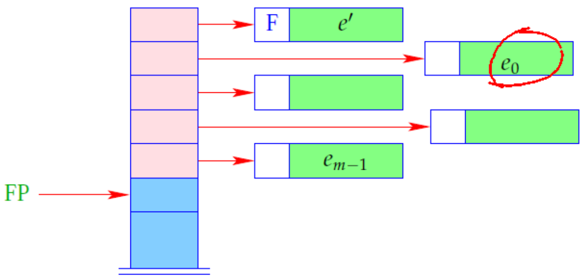
The arity of e' does not need to be m :-)

- 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).

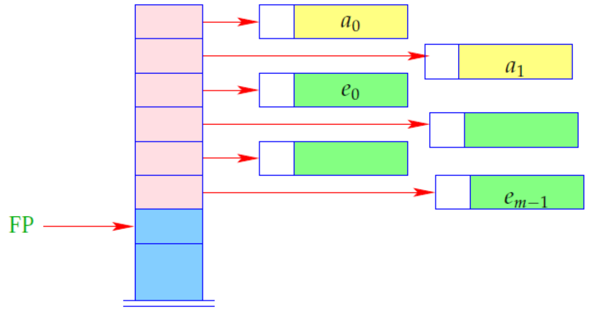
– If e' evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath e_0:



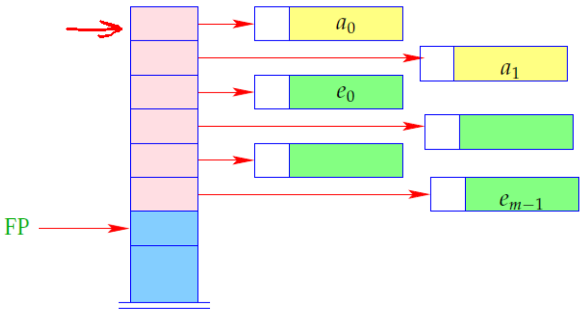
Alternative:



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



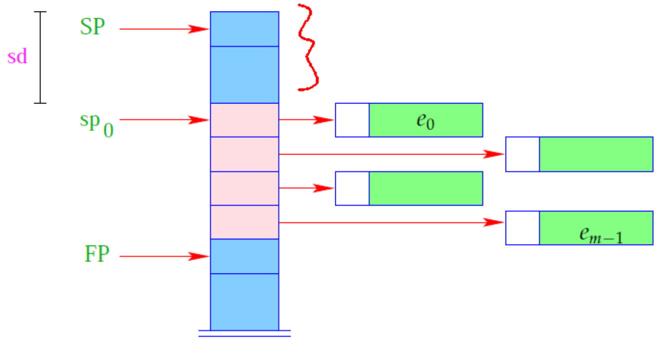
– Addressing of arguments and local variables relative to FP is no more possible. (Remember: m is unknown when the function definition is translated.)



– 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 differerence 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

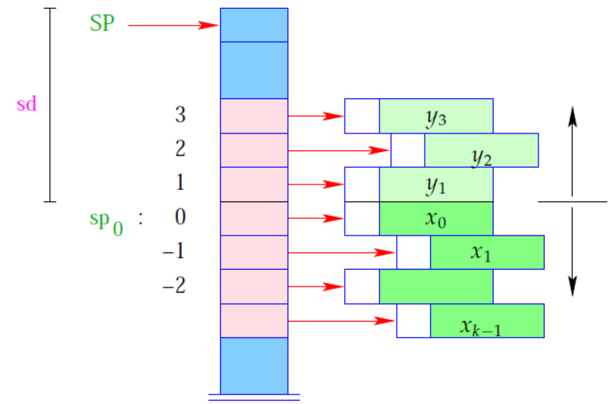
$$\mathrm{sp}_0 - i = (\mathrm{SP} - \mathrm{sd}) - i$$

- The local **let**-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:

- The differerence 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

$$\mathrm{sp}_0 - i = (\mathrm{SP} - \mathrm{sd}) - i$$

- The local **let**-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:



- The y_i have positive relative addresses $1, 2, 3, \ldots$, that is: $\rho\, y_i = (L, i)$.
- The absolute address of y_i is then $\mathrm{sp}_0 + i = (\mathrm{SP} - \mathrm{sd}) + i$

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

$$\begin{aligned} \mathrm{code}_V\; x\; \rho\; \mathrm{sd} \quad = \quad & \mathrm{getvar}\; x\; \rho\; \mathrm{sd} \\ & \mathrm{eval} \end{aligned}$$

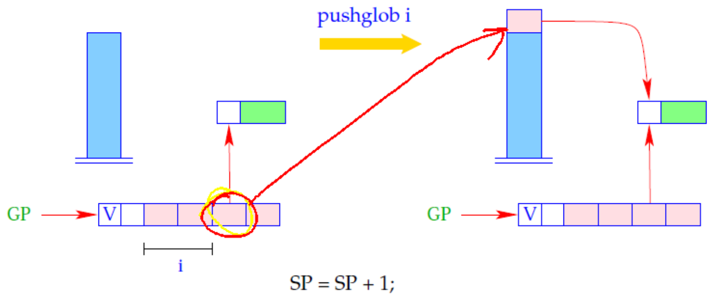
The instruction eval checks, whether the value has already been computed or whether its evaluation has to yet to be done ($\Longrightarrow$ will be treated later :-)

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

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

$$\begin{aligned} \mathrm{getvar}\; x\; \rho\; \mathrm{sd} \quad = \quad & \textbf{let } (t, i) = \rho\; x \textbf{ in} \\ & \textbf{case } t \textbf{ of} \\ & \quad L \Rightarrow \mathrm{pushloc}\; (\mathrm{sd} - i) \\ & \quad G \Rightarrow \mathrm{pushglob}\; \mathrm{i} \\ & \textbf{end} \end{aligned}$$

The access to global variables is much simpler:



SP = SP + 1;
S[SP] = GP→v[i];

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

$$\begin{array}{lll} \mathrm{code}_V\ x\ \rho\ \mathrm{sd} & = & \mathrm{getvar}\ x\ \rho\ \mathrm{sd} \\ & & \mathrm{eval} \end{array}$$

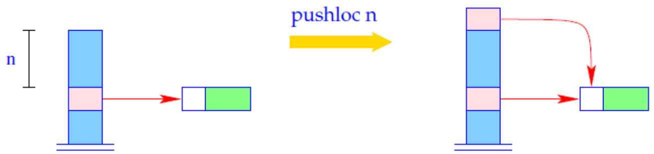
The instruction eval checks, whether the value has already been computed or whether its evaluation has to yet to be done ($\Longrightarrow$ will be treated later :-)

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

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

$$\begin{array}{lll} \mathrm{getvar}\ x\ \rho\ \mathrm{sd} & = & \textbf{let}\ (t, i) = \rho\ x\ \textbf{in} \\ & & \textbf{case}\ t\ \textbf{of} \\ & & \quad L \Rightarrow \mathrm{pushloc}\ (\mathrm{sd} - i) \\ & & \quad G \Rightarrow \mathrm{pushglob}\ \mathrm{i} \\ & & \textbf{end} \end{array}$$

The access to local variables:



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

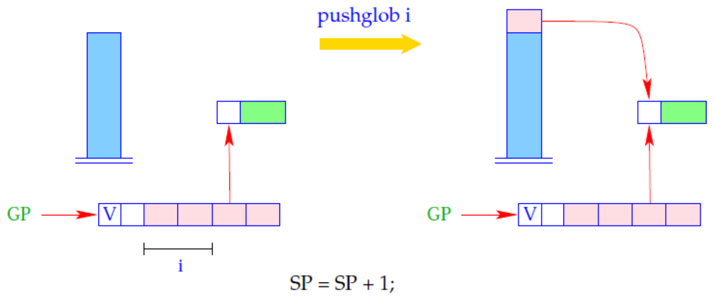
Correctness argument:

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

$$a = \mathrm{sp} - (\mathrm{sd} - i) = (\mathrm{sp} - \mathrm{sd}) + i = \mathrm{sp}_0 + i$$

... exactly as it should be :-)

The access to global variables is much simpler:



SP = SP + 1;
S[SP] = GP→v[i];

Example:

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

With CBN, we obtain:

$\text{code}_V\ e\ \rho\ 1$	=	getvar $b\ \rho$ 1	=	1	pushloc 0
		eval		2	eval
		getbasic		2	getbasic
		getvar $c\ \rho$ 2		2	pushglob 0
		eval		3	eval
		getbasic		3	getbasic
		add		3	add
		mkbasic		2	mkbasic