Script generated by TTT

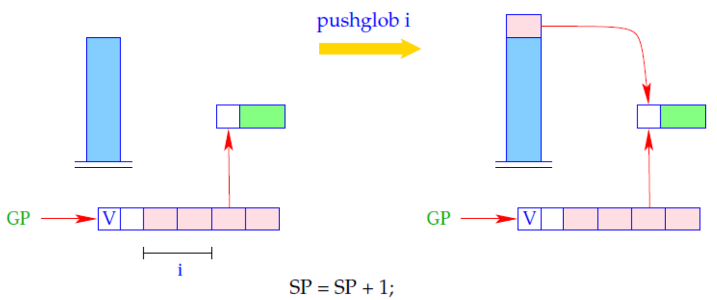
Title: Seidl: Virtual_Machines (12.06.2012)

Date: Tue Jun 12 14:02:49 CEST 2012

Duration: 85:11 min

Pages: 33

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

let x = 5 in
let x = x+1
in x

15 let-Expressions

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

Let $e \equiv \textbf{let}\ y_1 = e_1\ \textbf{in} \ldots \textbf{let}\ e_n\ \textbf{in}\ e_0$ 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_j only depends on $y_1, \ldots, y_{j-1}$. We obtain for CBN:

$$
\begin{aligned}
\text{code}_V\ e\ \rho\ \text{sd} \quad = \quad & \text{code}_C\ e_1\ \rho\ \text{sd} \\
& \text{code}_C\ e_2\ \rho_1\ (\text{sd}+1) \\
& \ldots \\
& \text{code}_C\ e_n\ \rho_{n-1}\ (\text{sd}+n-1) \\
& \text{code}_V\ e_0\ \rho_n\ (\text{sd}+n) \\
& \text{slide n} \qquad\qquad\qquad\qquad // \text{ deallocates local variables}
\end{aligned}
$$

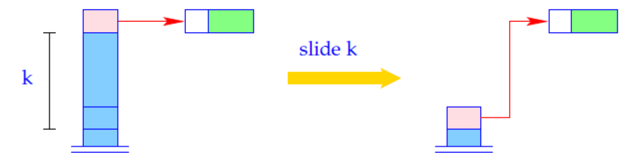
where $\quad \rho_j = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, j\}$.

In the case of CBV, we use code_V for the expressions $e_1, \ldots, e_n$.

Warning!

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

The instruction slide k deallocates again the space for the locals:



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

$$
\begin{aligned}
\text{code}_V\ e\ \rho\ \text{sd} \quad = \quad & \text{code}_C\ e_1\ \rho\ \text{sd} \\
& \text{code}_C\ e_2\ \rho_1\ (\text{sd}+1) \\
& \ldots \\
& \text{code}_C\ e_n\ \rho_{n-1}\ (\text{sd}+n-1) \\
& \text{code}_V\ e_0\ \rho_n\ (\text{sd}+n) \\
& \text{slide n} \qquad\qquad\qquad\qquad // \text{ deallocates local variables}
\end{aligned}
$$

where $\quad \rho_j = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, j\}$.

In the case of CBV, we use code_V for the expressions $e_1, \ldots, e_n$.

Warning!

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

$a \mapsto (L, 1)$
$b \mapsto (L, 2)$

Example:

Consider the expression

$$e \equiv \textbf{let } a = 19 \textbf{ in let } b = a * a \textbf{ in } \boxed{a + b}$$

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

0	loadc 19	3	getbasic	3	pushloc 1
1	mkbasic	3	mul	4	getbasic
1	pushloc 0	2	mkbasic	4	add
2	getbasic	2	pushloc 1	3	mkbasic
2	pushloc 1	3	getbasic	3	slide 2

F

F c_p a_p g_p

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;

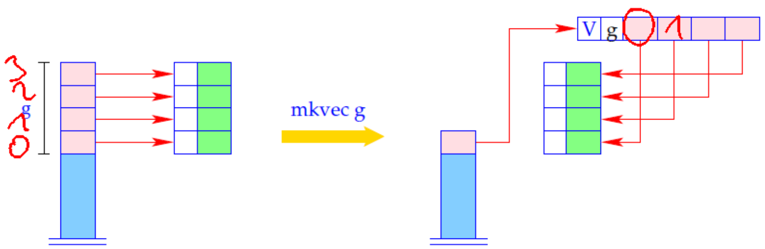
Separately, code for the body has to be generated.

Thus:

$$
\begin{array}{rrl}
\text{code}_V\ (\textbf{fun}\ x_0 \ldots x_{k-1} \to e)\ \rho\ \text{sd} & = & \boxed{\text{getvar}\ z_0\ \rho\ \text{sd}} \\
 & & \boxed{\text{getvar}\ z_1\ \rho\ (\text{sd}+1)} \\
 & & \ldots \\
 & & \boxed{\text{getvar}\ z_{g-1}\ \rho\ (\text{sd}+g-1)} \\
 & & \boxed{\text{mkvec}\ g} \\
 & & \text{mkfunval A} \\
 & & \text{jump B} \\
 & \text{A}: & \text{targ k} \\
 & & \text{code}_V\ e\ \rho'\ 0 \\
 & & \text{return k} \\
 & \text{B}: & \ldots
\end{array}
$$

where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(\textbf{fun}\ x_0 \ldots x_{k-1} \to e)$

and $\rho' = \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k-1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g-1\}$

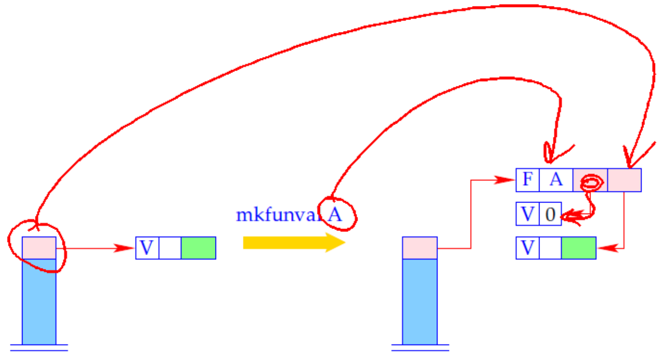



```
h = new (V, n);
SP = SP - g + 1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```


$$
\begin{array}{rrl}
\text{code}_V\ (\textbf{fun}\ x_0 \ldots x_{k-1} \to e)\ \rho\ \text{sd} & = & \text{getvar}\ z_0\ \rho\ \text{sd} \\
 & & \text{getvar}\ z_1\ \rho\ (\text{sd}+1) \\
 & & \ldots \\
 & & \text{getvar}\ z_{g-1}\ \rho\ (\text{sd}+g-1) \\
 & & \text{mkvec}\ g \\
 & & \text{mkfunval A} \\
 & & \text{jump B} \\
 & \text{A}: & \text{targ k} \\
 & & \text{code}_V\ e\ \rho'\ 0 \\
 & & \text{return k} \\
 & \text{B}: & \ldots
\end{array}
$$

where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(\textbf{fun}\ x_0 \ldots x_{k-1} \to e)$

and $\rho' = \{x_i \mapsto (L, -i) \mid i = 0, \ldots, k-1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \ldots, g-1\}$



a = new (V,0);
S[SP] = new (F, A, a, S[SP]);

$\rho' = \{b \mapsto (L,0), a \mapsto (G,0)\}$

Example:

Regard $f \equiv \textbf{fun}\ b \to \boxed{a} + b$ for $\boxed{\rho = \{a \mapsto (L,1)\}}$ and $\boxed{\text{sd} = 1}$.
$\boxed{\text{code}_V\ f\ \rho\ 1}$ 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 targ k and return k will be revealed later :-)

17 Function Application

Function applications correspond to function calls in C.
The necessary actions for the evaluation of $e'\ e_0\ \ldots\ e_{m-1}$ 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 e' to an F-object;
- Application of the function.

Thus for CBN:

$$
\begin{array}{llll}
\text{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \text{sd} & = & \text{mark}\ A & \text{// Allocation of the frame} \\
& & \text{code}_C\ e_{m-1}\ \rho\ (\text{sd}+3) & \\
& & \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+4) & \\
& & \ldots & \\
& & \text{code}_C\ e_0\ \rho\ (\text{sd}+m+2) & \\
& & \text{code}_V\ e'\ \rho\ (\text{sd}+m+3) & \text{// Evaluation of } e' \\
& & \text{apply} & \text{// corresponds to call} \\
& A: & \ldots &
\end{array}
$$

To implement CBV, we use code_V instead of code_C for the arguments e_i.

Example: For $(f\ 42)$, $\rho = \{f \mapsto (L,2)\}$ and sd $= 2$, we obtain with CBV:

2	mark A	6	mkbasic	7	apply
5	loadc 42	6	pushloc 4	3	A : ...

$\rho = \{ a \mapsto (L,1),\ f \mapsto (L,2) \}$

A Slightly Larger Example:

$$\textbf{let } a = 17 \textbf{ in let } f = \boxed{\textbf{fun } b \to a + b} \textbf{ in } \boxed{f\ 42}$$

For CBV and $\cancel{kp}$ sd $= 0$ we obtain:

0	loadc 17	2		jump B	2		getbasic	5		loadc 42
1	mkbasic	0	A:	targ 1	2		add	5		mkbasic
1	pushloc 0	0		pushglob 0	1		mkbasic	6		pushloc 4
2	mkvec 1	1		getbasic	1		return 1	7		apply
2	mkfunval A	1		pushloc 1	2	B:	mark C	3	C:	slide 2

A Slightly Larger Example:

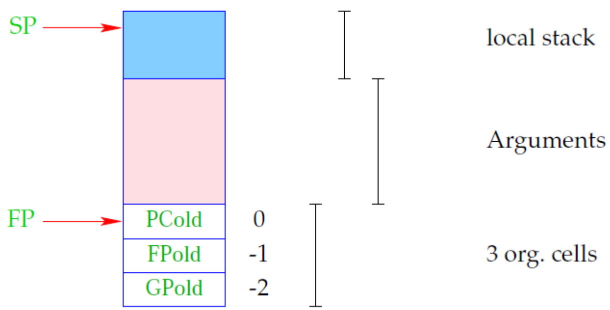
$$\textbf{let } a = 17 \textbf{ in let } f = \textbf{fun } b \to a + b \textbf{ in } f\ 42$$

For CBV and kp $= 0$ we obtain:

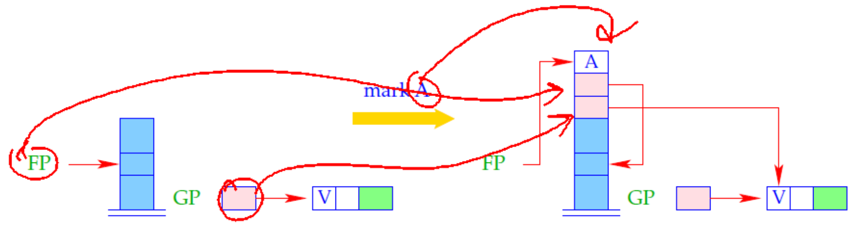
0	loadc 17	2		jump B	2		getbasic	5		loadc 42
1	mkbasic	0	A:	targ 1	2		add	5		mkbasic
1	pushloc 0	0		pushglob 0	1		mkbasic	6		pushloc 4
2	mkvec 1	1		getbasic	1		return 1	7		apply
2	mkfunval A	1		pushloc 1	2	B:	mark C	3	C:	slide 2

135

For the implementation of the new instruction, we must fix the organization of a stack frame:

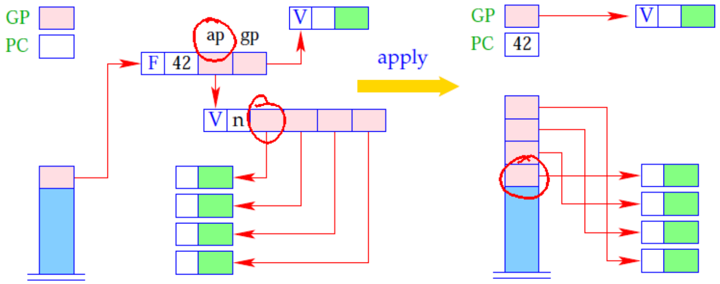


Different from the CMa, the instruction mark A already saves the return address:



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:

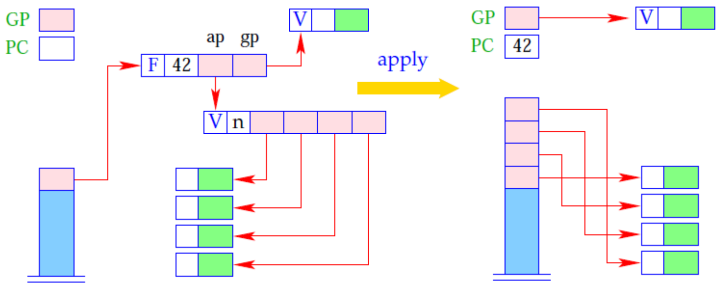



```
h = S[SP];
if (H[h] != (F,_,_))
    Error "no fun";
else {
    GP = h→gp; PC = h→cp;
    for (i=0; i< h→ap→n; i++)
        S[SP+i] = h→ap→v[i];
    SP = SP + h→ap→n – 1;
}
```


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 !!!

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];
if (H[h] != (F,_,_))
Error "no fun";
else {
GP = h→gp; PC = h→cp;
for (i=0; i< h→ap→n; i++)
S[SP+i] = h→ap→v[i];
SP = SP + h→ap→n – 1;
}

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.

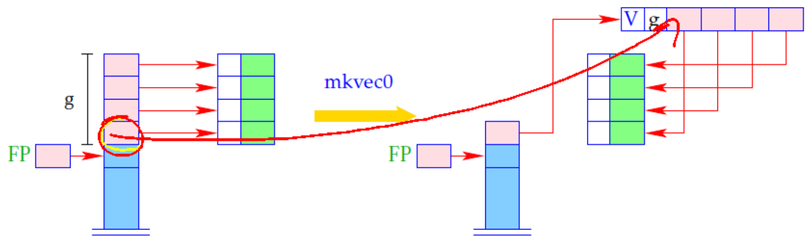
The test for number of arguments uses: SP – FP targ k is a complex instruction.

We decompose its execution in the case of under-supply into several steps:

```
targ k  =  if (SP – FP < k) {
               mkvec0;       // creating the argumentvector
               wrap;         // wrapping into an F – object
               popenv;       // popping the stack frame
           }
```

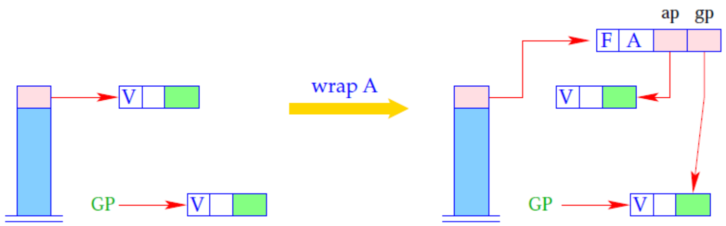
The combination of these steps into one instruction is a kind of optimization :-)

The instruction mkvec0 takes all references from the stack above FP and stores them into a vector:




```
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```

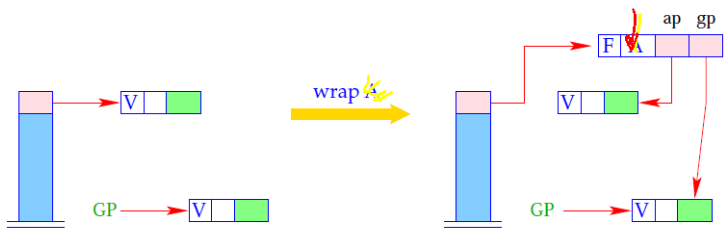

The instruction wrap A wraps the argument vector together with the global vector into an F-object:



S[SP] = new (F, A, S[SP], GP);

The instruction wrap A wraps the argument vector together with the global vector into an F-object:

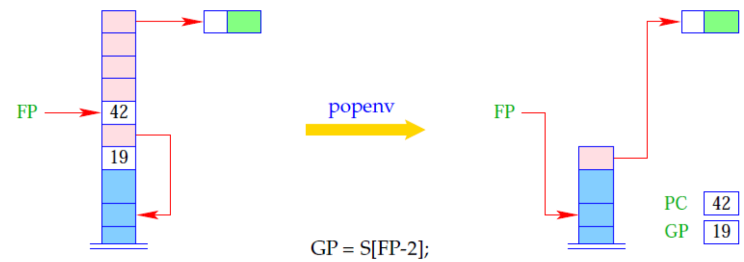
PC - 1



A = PC - 1;

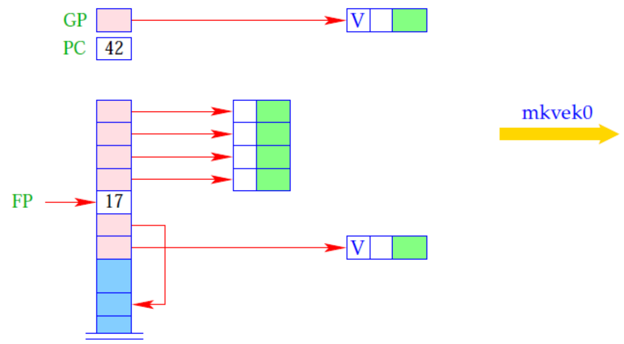
S[SP] = new (F, A, S[SP], GP);

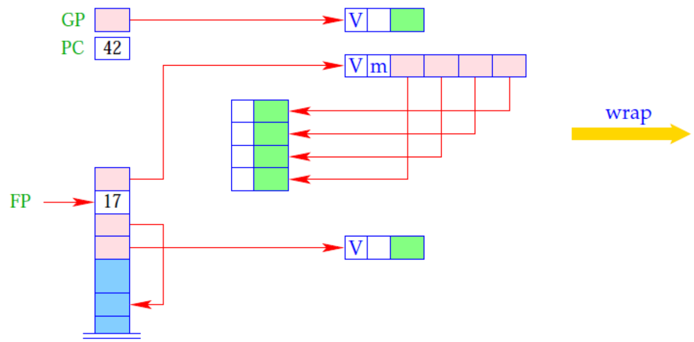
The instruction popenv finally releases the stack frame:

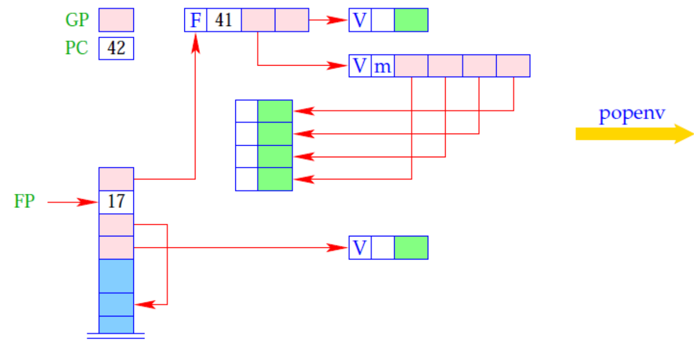


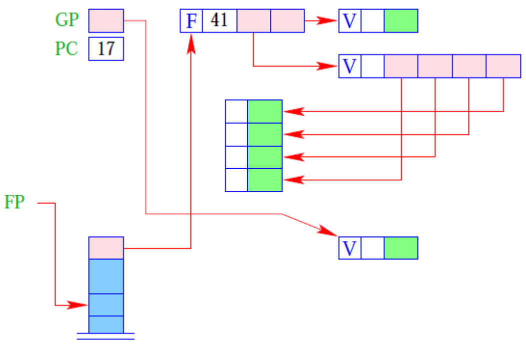

```
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];
```

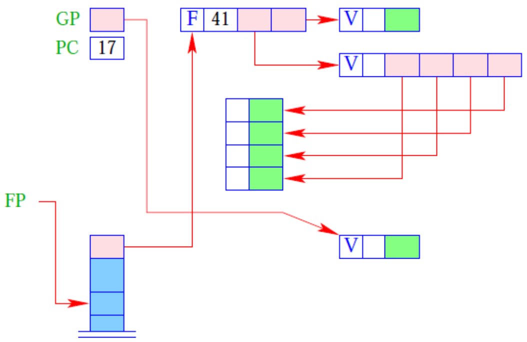

Thus, we obtain for targ k in the case of under supply:








GP
PC
17
F
41
V
V
FP
V


GP
PC
17
F
41
V
V
FP
V