Script generated by TTT

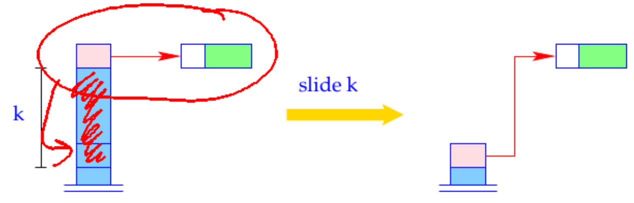
Title: Seidl: Virtual Machines (12.05.2014)

Date: Mon May 12 10:15:17 CEST 2014

Duration: 91:46 min

Pages: 31

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;

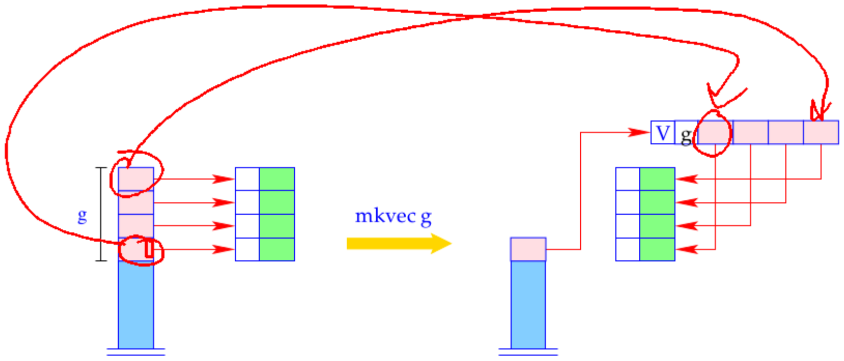
Separately, code for the body has to be generated.

Thus:

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

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

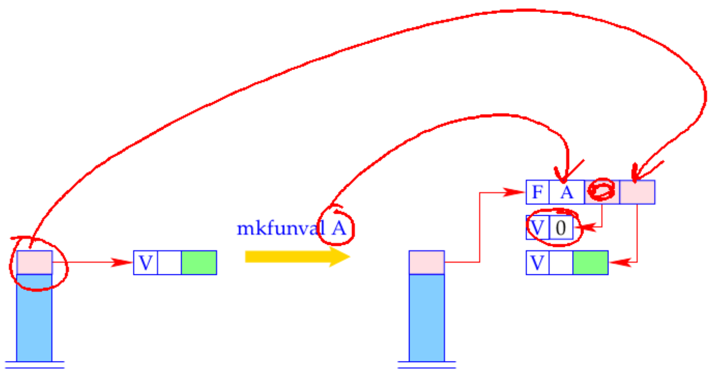
and $\rho' = \{x_i \mapsto (L, -i) \mid i = 0, \dots, k-1\} \cup \{z_j \mapsto (G, j) \mid j = 0, \dots, g-1\}$




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


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

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]);
```


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

Example:

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

$$\begin{array}{lll} \text{code}_V\,(e'\,e_0\,\ldots\,e_{m-1})\,\rho\,\text{sd} & = & \boxed{\begin{array}{l} \text{mark } A \\ \text{code}_C\; e_{m-1}\;\rho\;(\text{sd}+3) \\ \text{code}_C\; e_{m-2}\;\rho\;(\text{sd}+4) \\ \ldots \\ \text{code}_C\; e_0\;\rho\;(\text{sd}+m+2) \\ \text{code}_V\; e'\;\rho\;(\text{sd}+m+3) \\ \cancel{\text{apply}} \end{array}} \end{array}$$

// Allocation of the frame

// Evaluation of e'

call

// corresponds to call

A :

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:	...

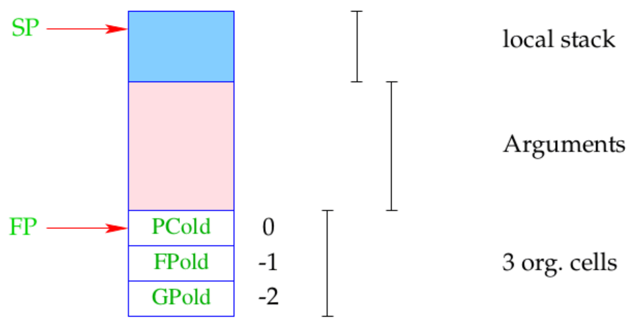
A Slightly Larger Example:

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

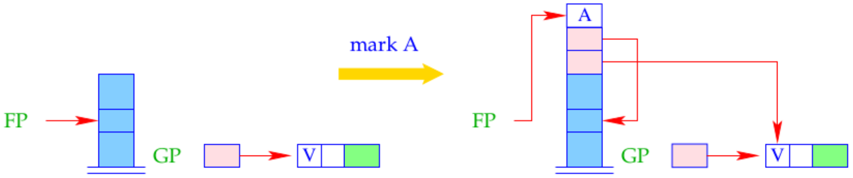
For CBV and sd $= 0$ we obtain:

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

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

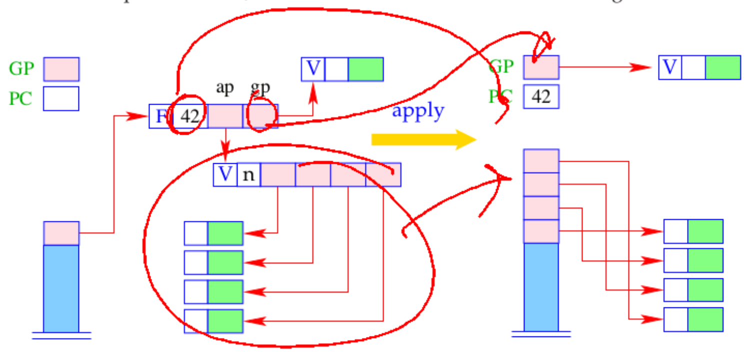


Different from the CMa, the instruction mark A already saves the return address:




```
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 !!!

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.

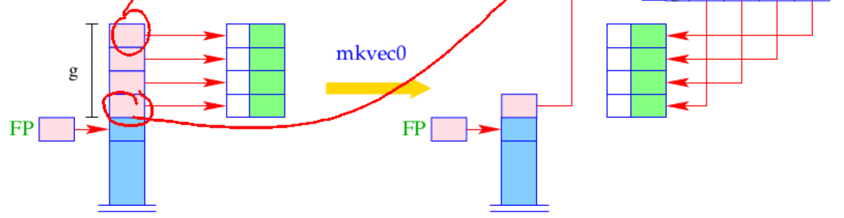
The test for number of arguments uses: SP – FP targ k is a complex instruction.

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;
```


targ k is a complex instruction.

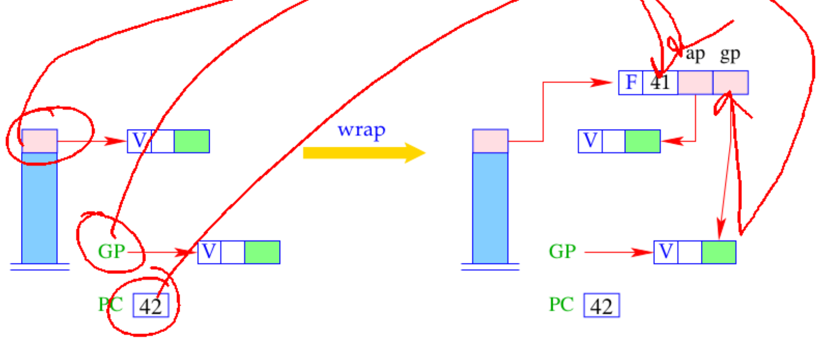
We decompose its execution in the case of under-supply into several steps:

```
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 :-)

147

The instruction wrap wraps the argument vector together with the global vector and PC-1 into an F-object:



S[SP] = new (F, PC-1, S[SP], GP);

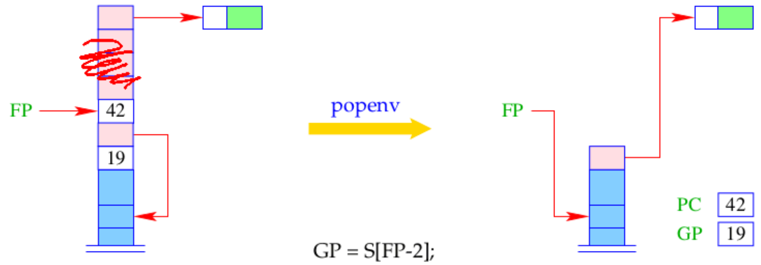
targ k is a complex instruction.

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
           }
```

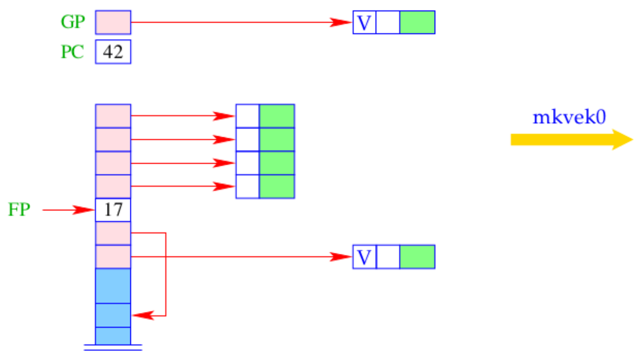
The combination of these steps into one instruction is a kind of optimization :-)

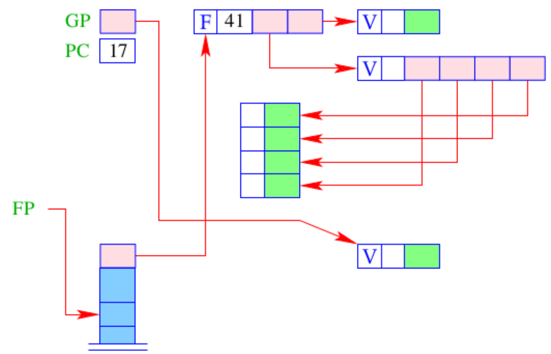
The instruction popenv finally releases the stack frame:




```
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:



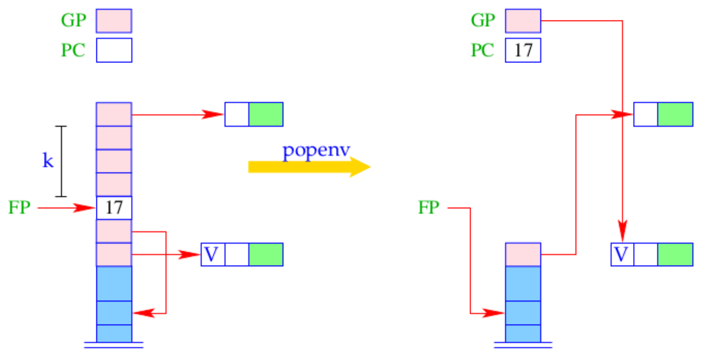


- The stack frame can be released after the execution of the body if exactly the right number of arguments was available.
- If there is an oversupply of arguments, the body must evaluate to a function, which consumes the rest of the arguments ...
- The check for this is done by return k:

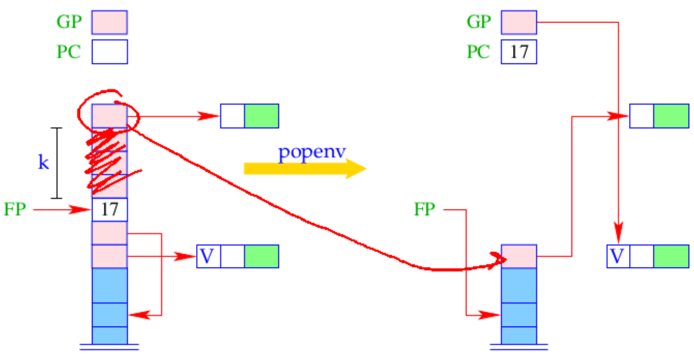
```
return k  =  if (SP − FP = k + 1)
                popenv;          // Done
             else {              // There are more arguments
                slide k;
                apply;           // another application
             }
```

The execution of return k results in:

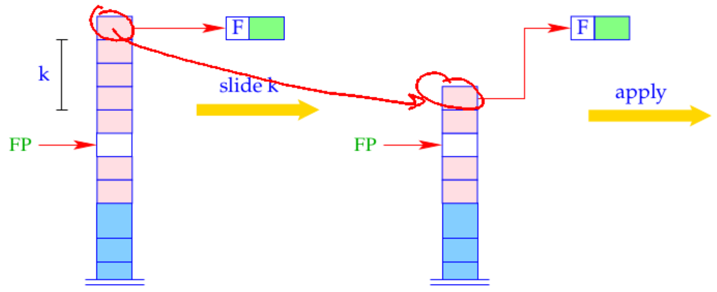
Case: Done



Case: Done



Case: Over-supply



let rec x = x+1

let rec f = fun x → f x

19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$.

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.

Warning:

In a **letrec**-expression, the definitions can use variables that will be allocated only later! $\Longrightarrow$ Dummy-values are put onto the stack before processing the definition.

For CBN, we obtain:

$$
\begin{array}{lcll}
\text{code}_V\ e\ \rho\ \text{sd} & = & \text{alloc n} & \text{// allocates local variables} \\
 & & \text{code}_C\ e_1\ \rho'\ (\text{sd}+n) & \\
 & & \text{rewrite n} & \\
 & & \ldots & \\
 & & \text{code}_C\ e_n\ \rho'\ (\text{sd}+n) & \\
 & & \text{rewrite 1} & \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{sd}+n) & \\
 & & \text{slide n} & \text{// deallocates local variables}
\end{array}
$$

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

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

Warning:

Recursive definitions of basic values are undefined with CBV!!!

19 let-rec-Expressions

Consider the expression $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$.

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.

Warning:

In a **letrec**-expression, the definitions can use variables that will be allocated only later! $\Longrightarrow$ Dummy-values are put onto the stack before processing the definition.

158

For CBN, we obtain:

$$
\begin{array}{lcll}
\text{code}_V\ e\ \rho\ \text{sd} & = & \text{alloc n} & \text{// allocates local variables} \\
 & & \text{code}_C\ e_1\ \rho'\ (\text{sd}+\boxed{n}) & \\
 & & \boxed{\text{rewrite n}} & \\
 & & \ldots & \\
 & & \text{code}_C\ e_n\ \rho'\ (\text{sd}+n) & \\
 & & \boxed{\text{rewrite 1}} & \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{sd}+n) & \\
 & & \text{slide n} & \text{// deallocates local variables}
\end{array}
$$

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

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

Warning:

Recursive definitions of basic values are undefined with CBV!!!

159