Script generated by TTT

Title: Seidl: Virtual_Machines (03.05.2016)

Date: Tue May 03 10:28:20 CEST 2016

Duration: 85:17 min

Pages: 37

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,

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

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 :	...

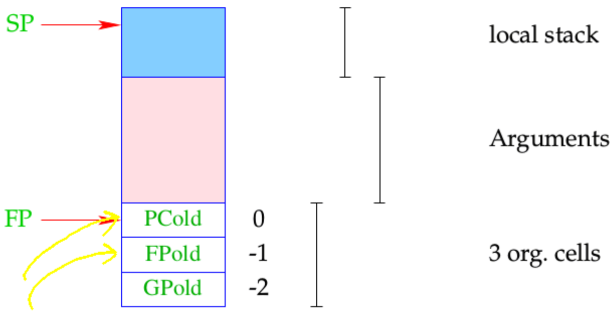
A Slightly Larger Example

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

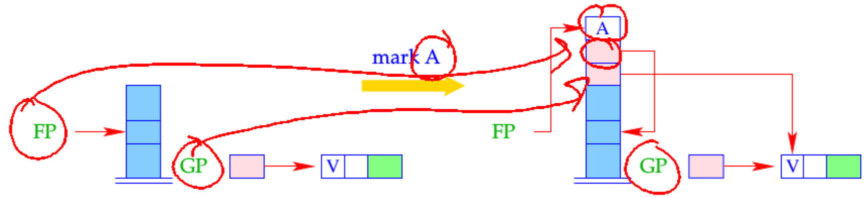
For CBV and sd $= 0$ we obtain:

0	loadc 17	2		jump B	2		getbasic	5		loadc 42
1	mkbasic	0	A:	targ 1	2		add	6		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:

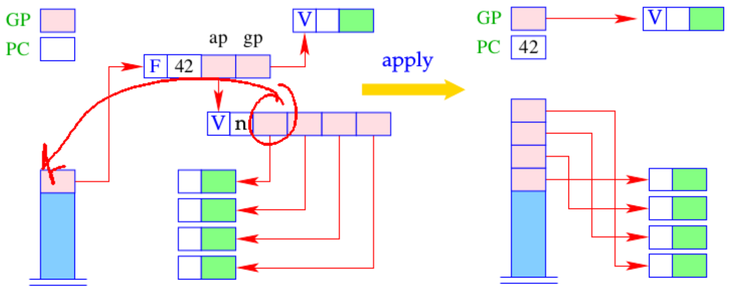


Different from the CMa, the instruction mark A already saves the return address:



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;
}
```


Caveat

- 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

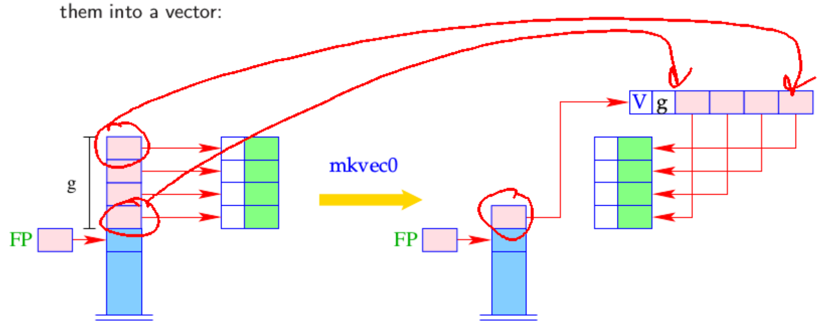
targ k is a complex instruction.

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
           }
```

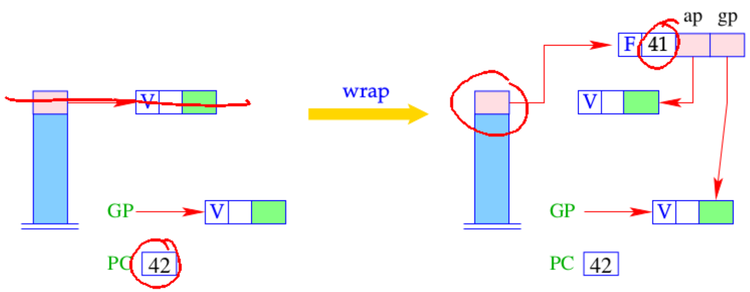
The combination of these steps into one instruction is a kind of optimization.

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



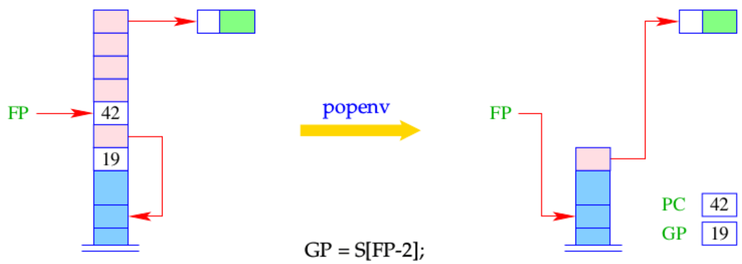

```
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```


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

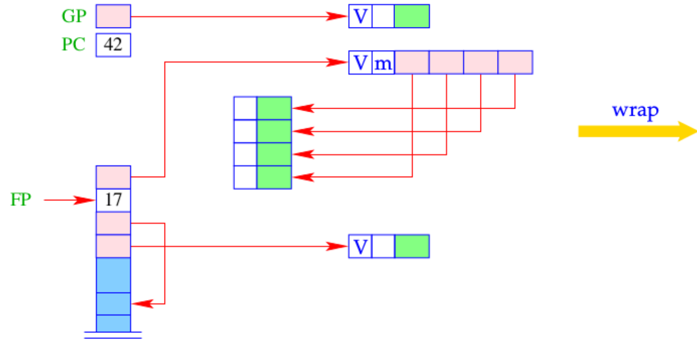


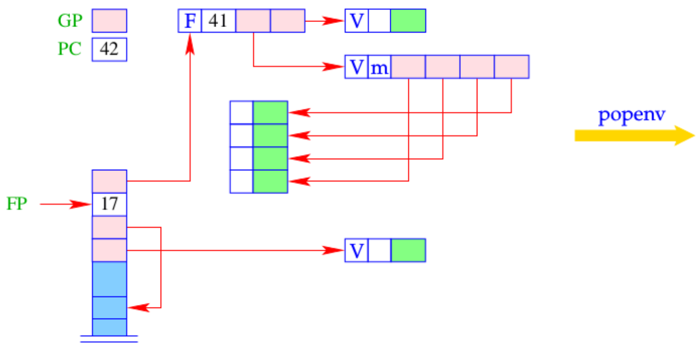
S[SP] = new (F, PC-1, S[SP], GP);

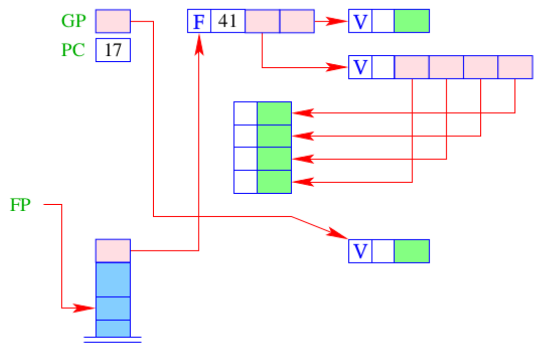
The instruction popenv finally releases the stack frame:



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





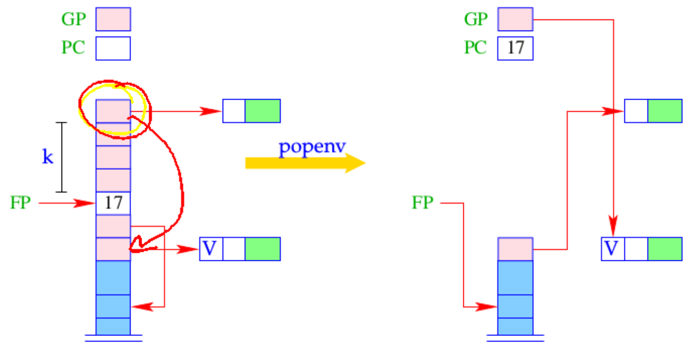


- The stack frame can be released after the execution of the body if exactly the right number of arguments was available.
- If there is an oversupply of arguments, the body must evaluate to a function, which consumes the rest of the arguments ...
- The check for this is done by return k:

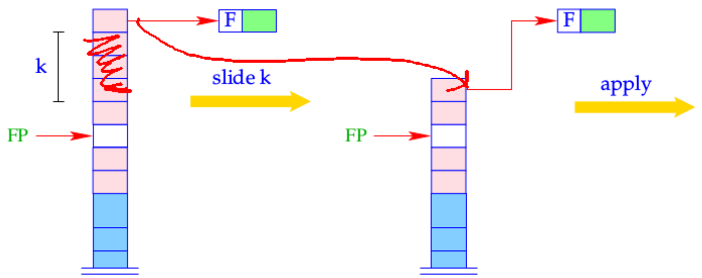
```
return k  =  if (SP − FP = k + 1)
                popenv;              // Done
             else {                  // There are more arguments
                slide k;
                apply;               // another application
             }
```

The execution of return k results in:

Case: Done



Case: Over-supply



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.

Caveat

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:

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

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 $e_1, \ldots, e_n$.

Caveat

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

Example

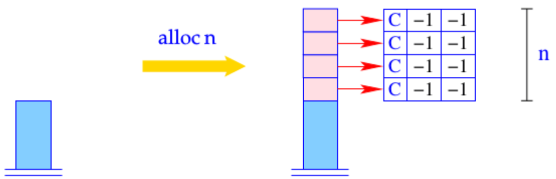
Consider the expression

$$e \equiv \textbf{let rec } f = \textbf{fun } x\ y \to \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f\,(x*y)\,(y-1) \textbf{ in } f\ 1$$

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

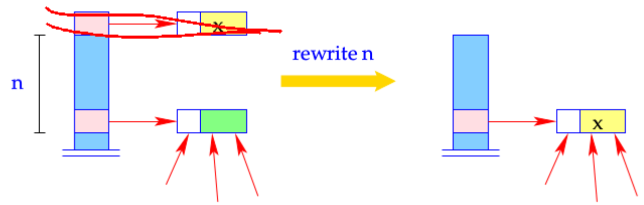
0	alloc 1	0	A:	targ 2	4		loadc 1
1	pushloc 0	0		...	5		mkbasic
2	mkvec 1	1		return 2	5		pushloc 4
2	mkfunval A	2	B:	rewrite 1	6		apply
2	jump B	1		mark C	2	C:	slide 1

The instruction alloc n reserves n cells on the stack and initialises them with n dummy nodes:




```
for (i=1; i<=n; i++)
    S[SP+i] = new (C,-1,-1);
SP = SP + n;
```


The instruction rewrite n overwrites the contents of the heap cell pointed to by the reference at S[SP–n]:




```
H[S[SP-n]] = H[S[SP]];
SP = SP - 1;
```

- The reference S[SP – n] remains unchanged!
- Only its contents is changed!

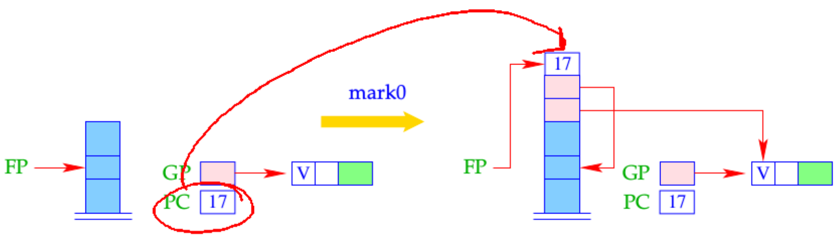
20 Closures and their Evaluation

- Closures are needed in the implementation of CBN for **let**-, **let**-**rec** expressions as well as for actual paramaters of functions.
- Before the value of a variable is accessed (with CBN), this value must be available.
- Otherwise, a stack frame must be created to determine this value.
- This task is performed by the instruction eval.

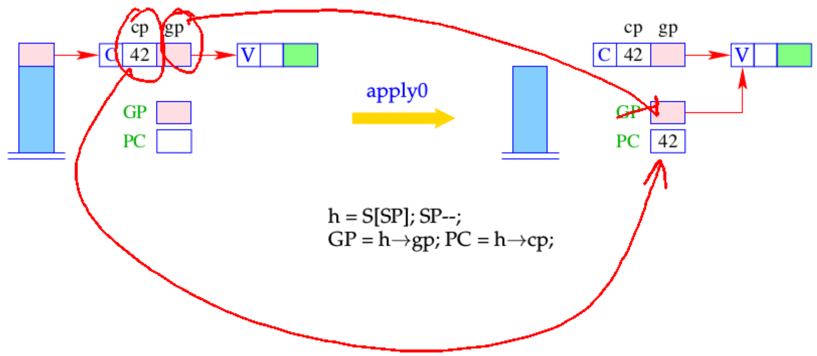
eval can be decomposed into small actions:

```
eval  =  if (H[S[SP]] ≡ (C, _, _)) {
             mark0;            // allocation of the stack frame
             pushloc 3;        // copying of the reference
             apply0;           // corresponds to apply
         }
```

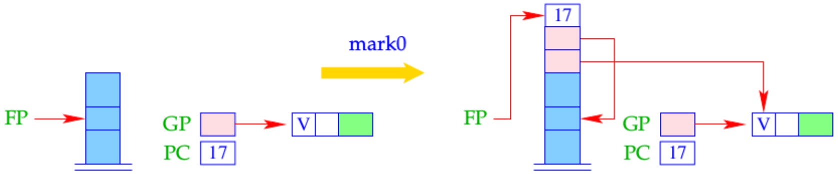
- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure means evaluation of an application of this function to 0 arguments.
- In constrast to mark A , mark0 dumps the current PC.
- The difference between apply and apply0 is that no argument vector is put on the stack.



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

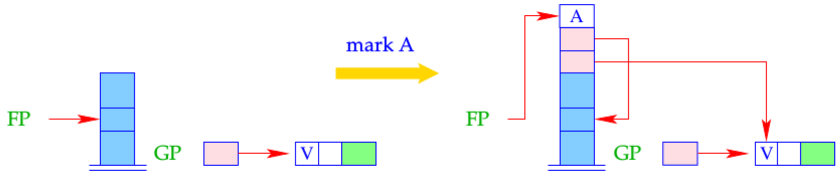


We thus obtain for the instruction eval:

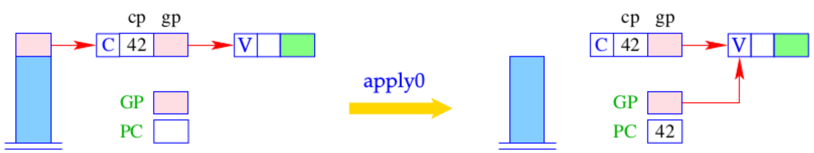


S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

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

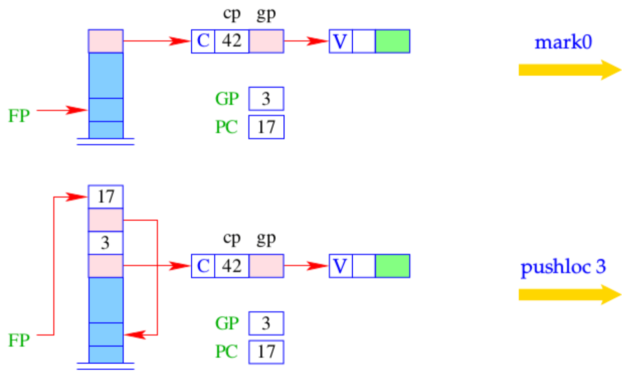


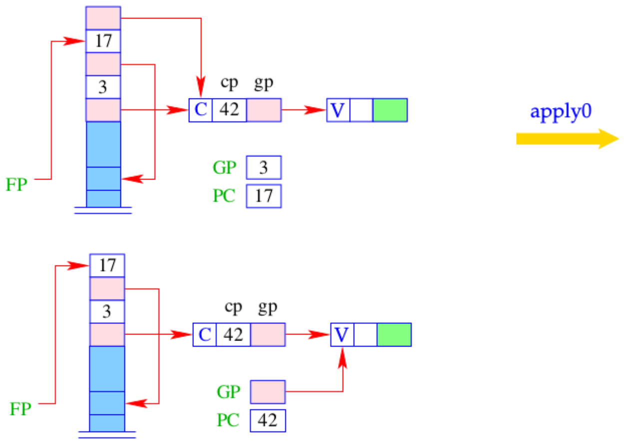
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = A;
~~FP = SP = SP + 3;~~



h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

We thus obtain for the instruction eval:





The construction of a closure for an expression e consists of:

- Packing the bindings for the free variables into a vector;
- Creation of a C-object, which contains a reference to this vector and to the code for the evaluation of e:

$$
\begin{array}{rrl}
\text{code}_C\ e\ \rho\ \text{sd} & = & \text{getvar}\ z_0\ \rho\ \text{sd} \\
 & & \text{getvar}\ z_1\ \rho\ (\text{sd}+1) \\
 & & \ldots \\
 & & \text{getvar}\ z_{g-1}\ \rho\ (\text{sd}+g-1) \\
 & & \text{mkvec}\ g \\
 & & \text{mkclos A} \\
 & & \text{jump B} \\
 & \text{A}: & \text{code}_V\ e\ \rho'\ 0 \\
 & & \text{update} \\
 & \text{B}: & \ldots
\end{array}
$$

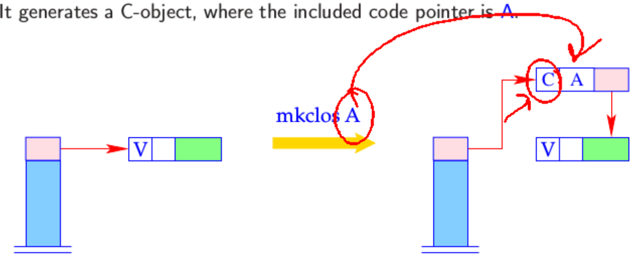
where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Example

Consider $e \equiv a * a$ with $\rho = \{a \mapsto (L, 0)\}$ and $\text{sd} = 1$. We obtain:

1	pushloc 1	0	A:	pushglob 0	2		getbasic
2	mkvec 1	1		eval	2		mul
2	mkclos A	1		getbasic	1		mkbasic
2	jump B	1		pushglob 0	1		update
		2		~~eval~~	2	B:	...

- The instruction mkclos A is analogous to the instruction mkfunval A.
- It generates a C-object, where the included code pointer is A.



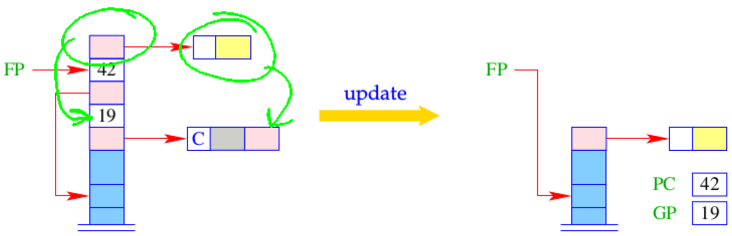
S[SP] = new (C, A, S[SP]);

In fact, the instruction update is the combination of the two actions:

popenv

rewrite 1

It overwrites the closure with the computed value.



In fact, the instruction update is the combination of the two actions:

popenv

rewrite 1

It overwrites the closure with the computed value.