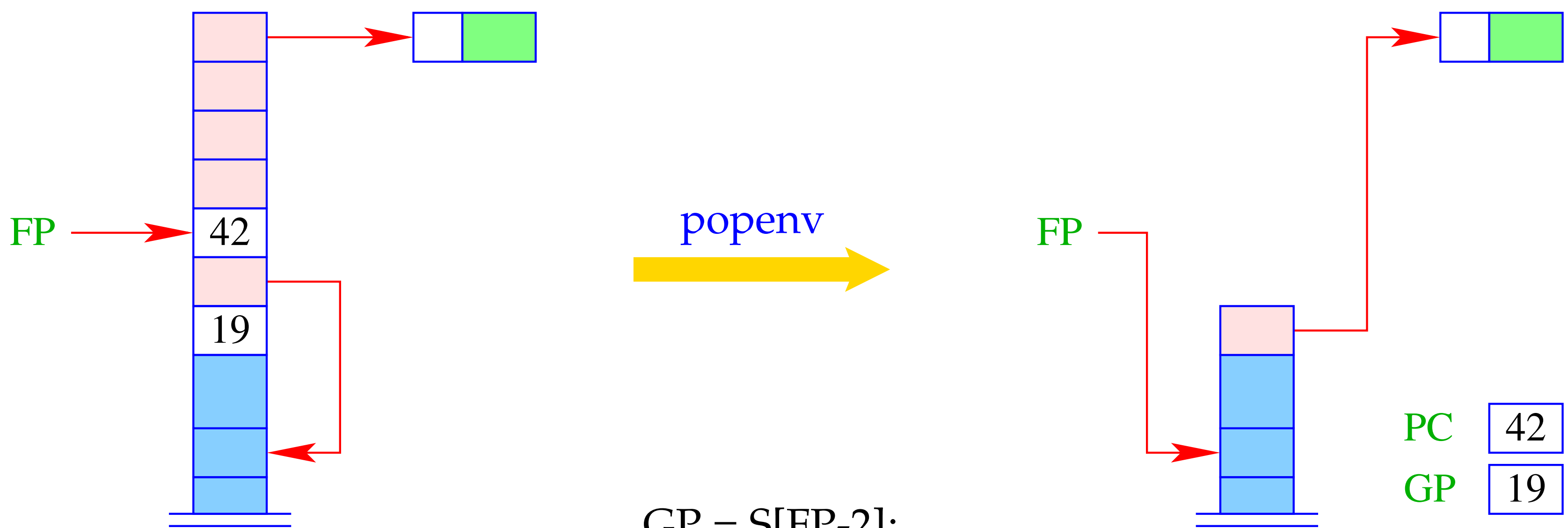
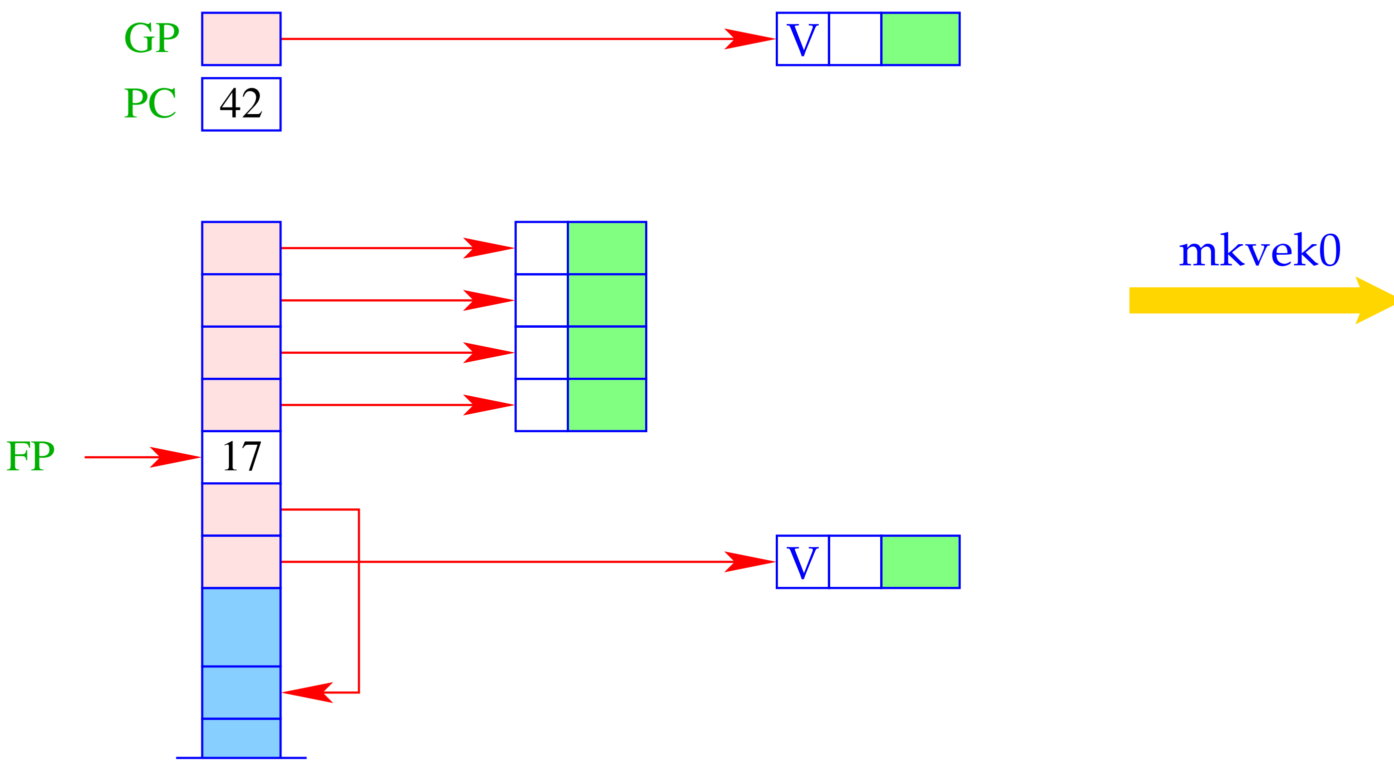
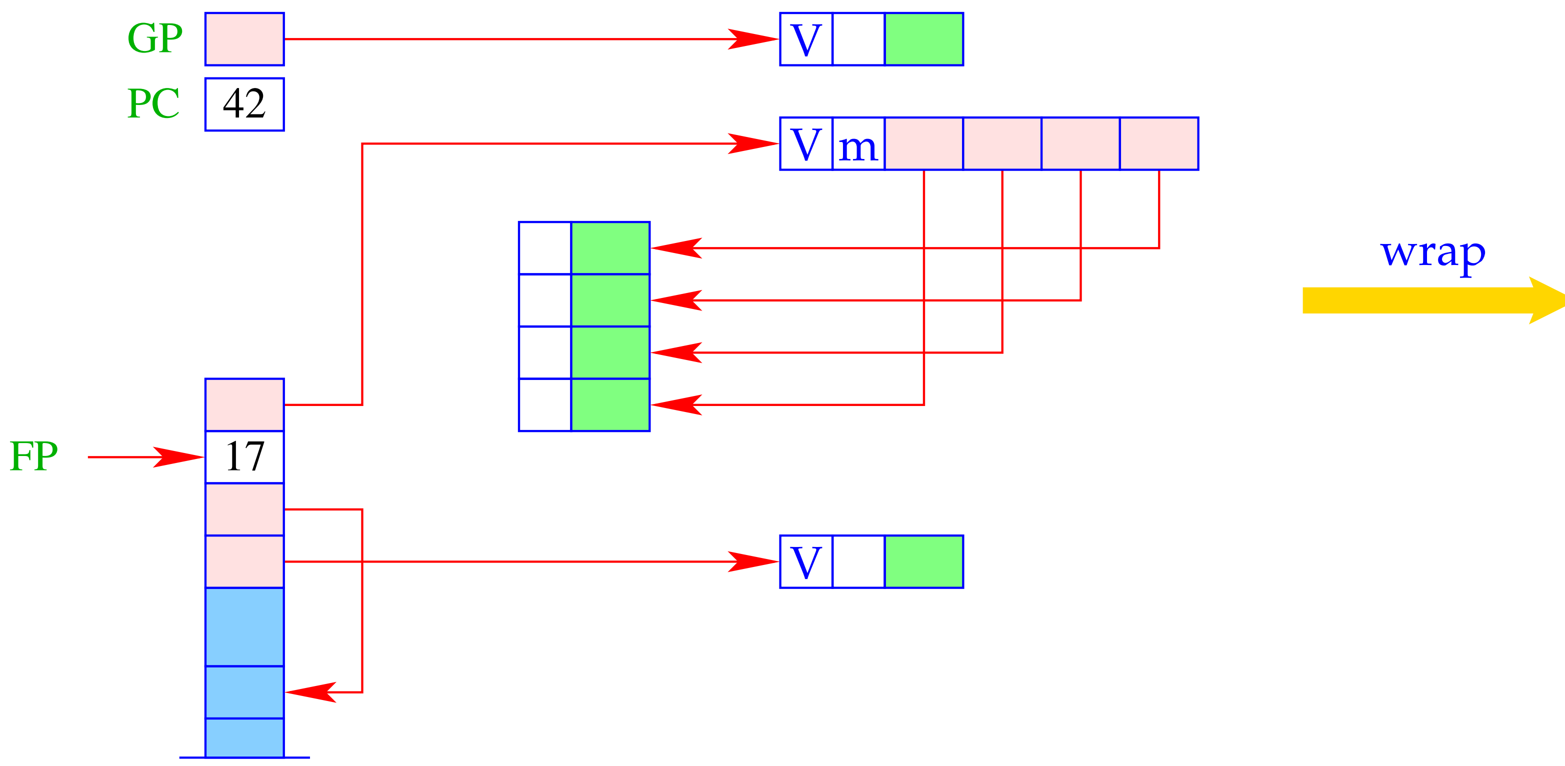
The instruction popenv finally releases the stack frame:

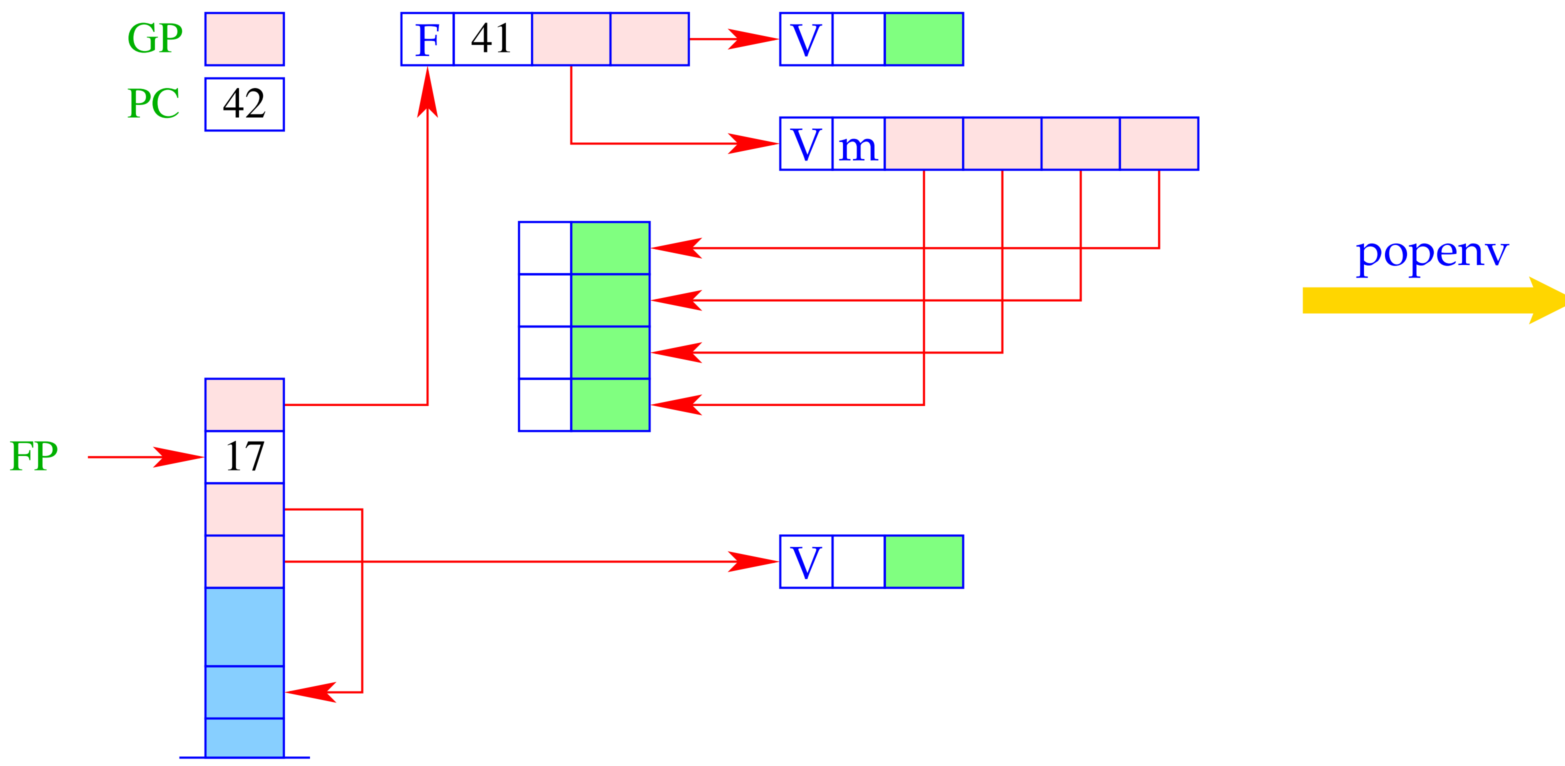


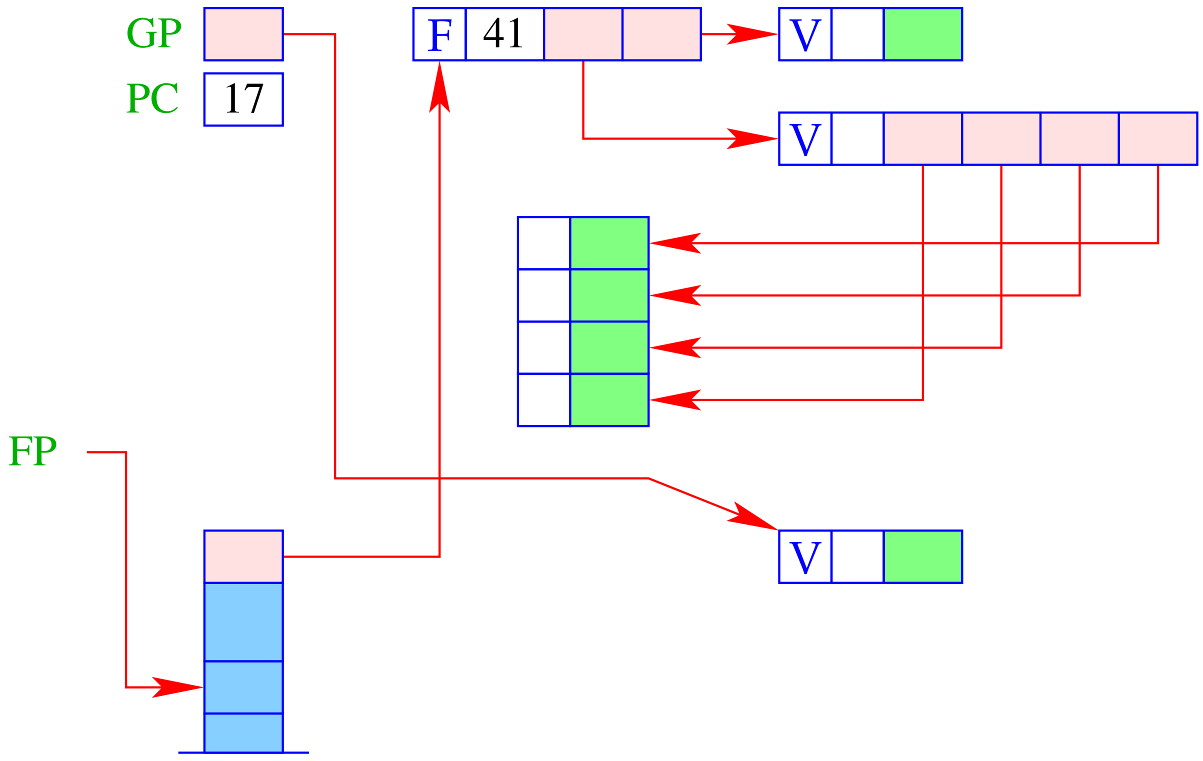
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];

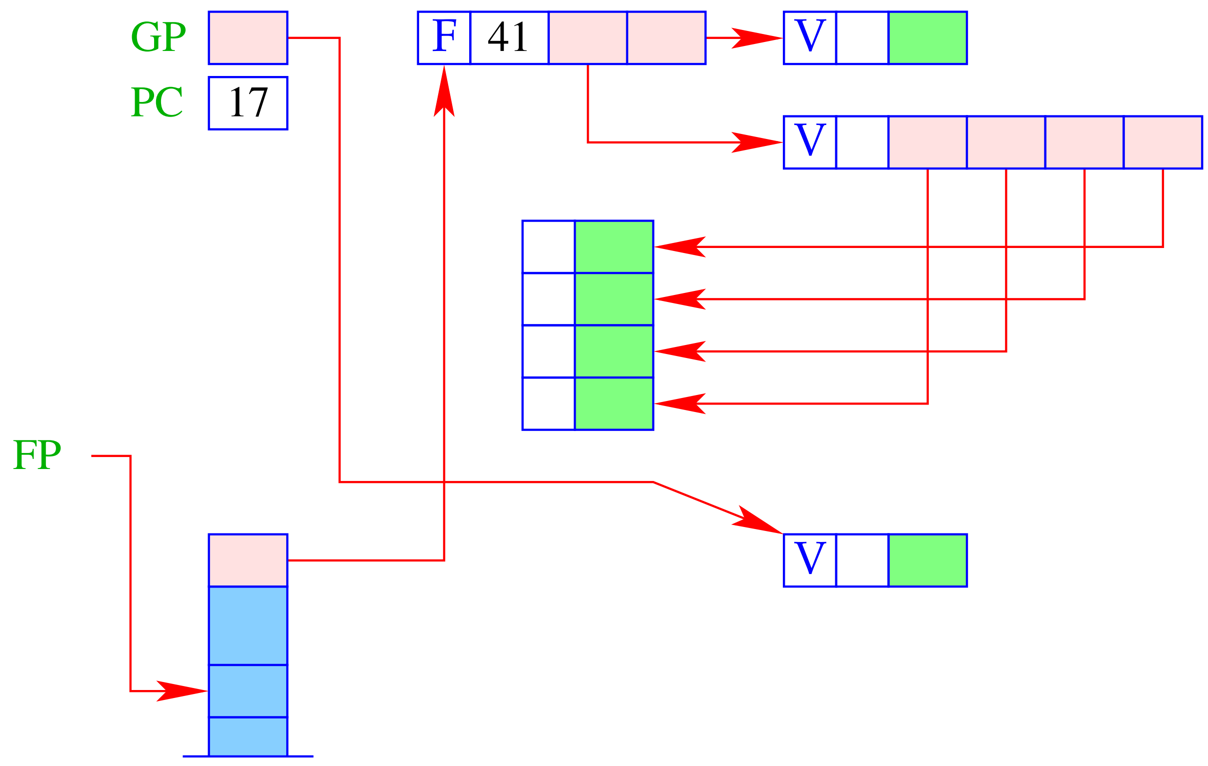
Thus, we obtain for targ k in the case of under supply:




GP
PC
42
V
V
m
wrap
FP
17
V


GP
PC
42
F
41
V
V
m
popenv
FP
17
V


GP
PC
17
F
41
V
V
FP
V

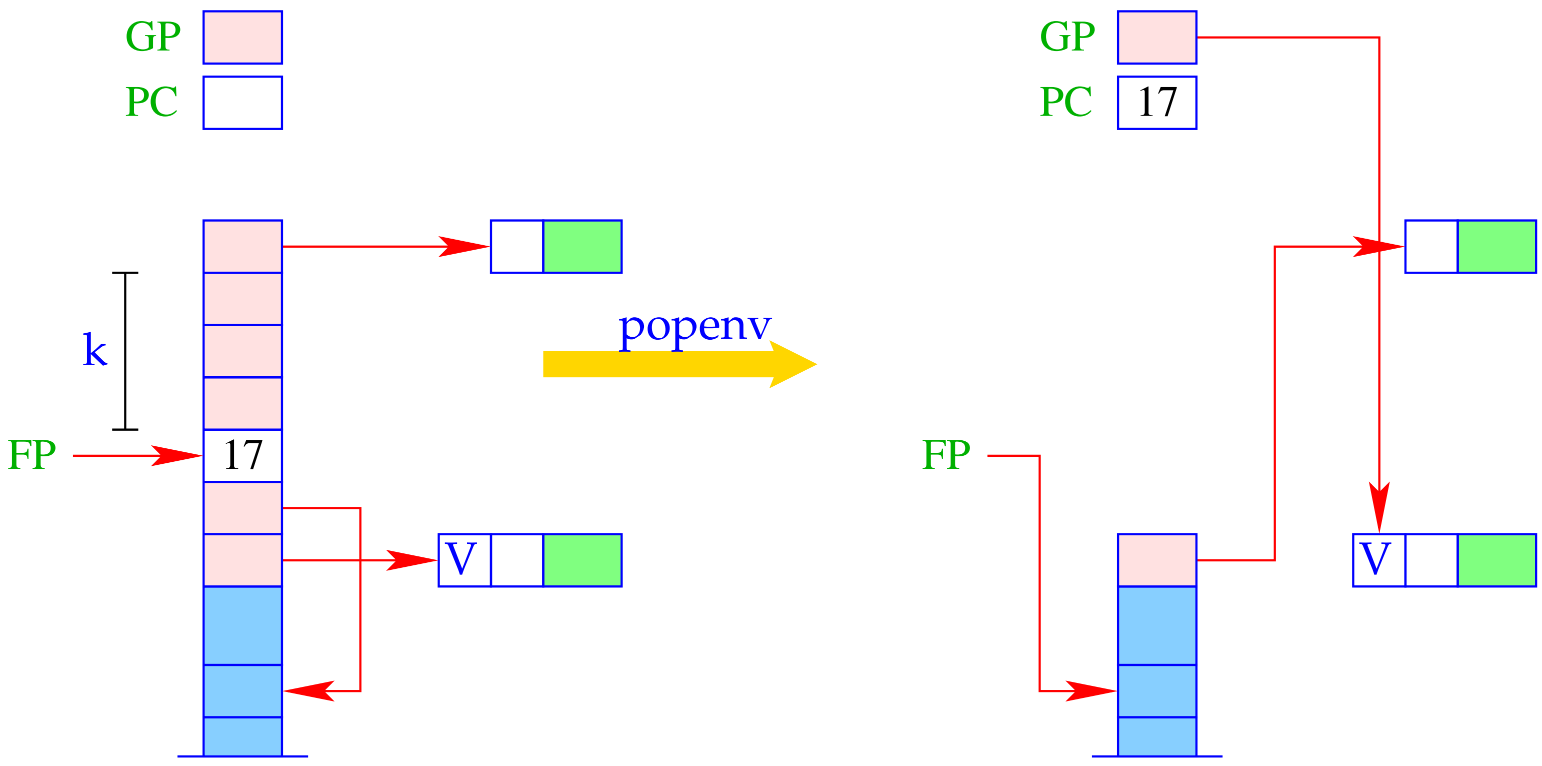

GP
PC
17
F
41
V
V
FP
V

- 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:

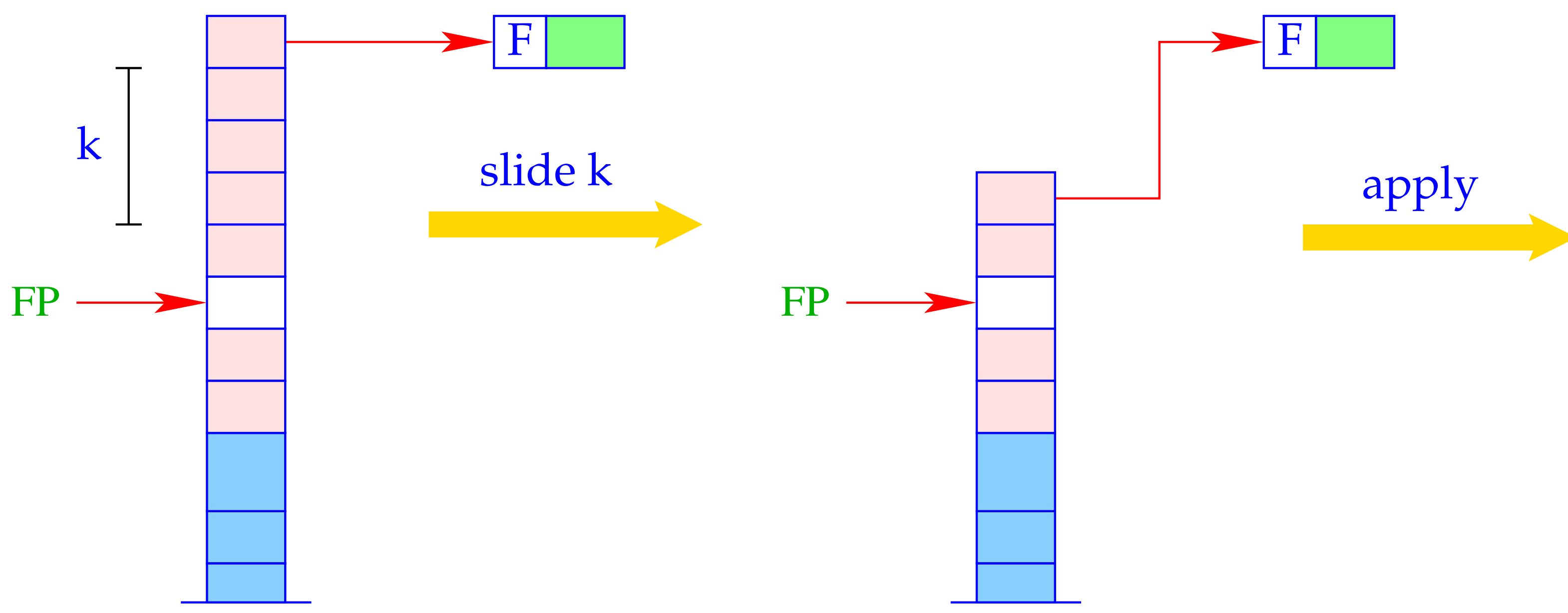
$$
\begin{array}{lcll}
\text{return k} & = & \text{if } (\text{SP} - \text{FP} = \text{k} + 1) & \\
 & & \quad \text{popenv;} & // \text{ Done} \\
 & & \text{else } \{ & // \text{ There are more arguments} \\
 & & \quad \text{slide k;} & \\
 & & \quad \text{apply;} & // \text{ another application} \\
 & & \} &
\end{array}
$$

The execution of return k results in:

Case: Done



Case: Over-supply



19 letrec-Expressions

Consider the expression $e \equiv \textbf{letrec}\ y_1 = e_1; \ldots; y_n = e_n\ \textbf{in}\ e_0$.

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:

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

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

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!!!

Example:

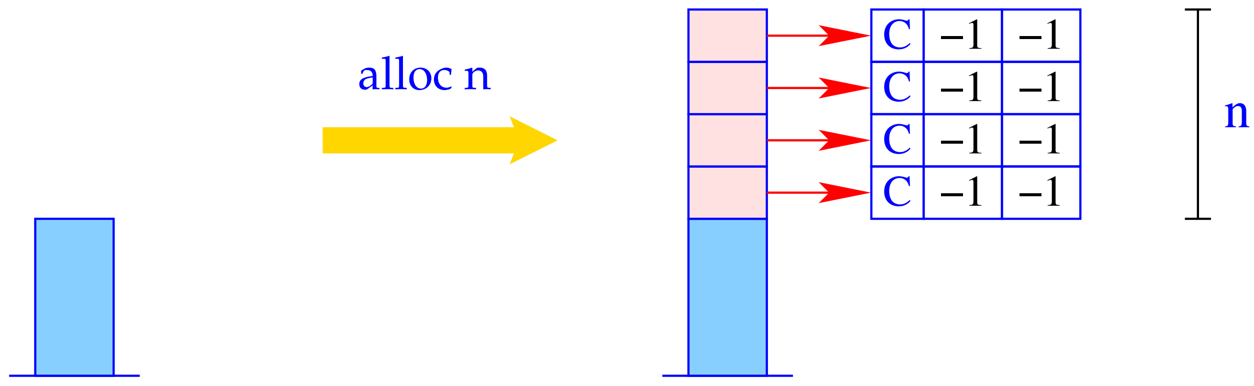
Consider the expression

$$e \equiv \textbf{letrec}\ f = \textbf{fn}\, x, y \Rightarrow \textbf{if}\, y \leq 1\ \textbf{then}\ x\ \textbf{else}\ f\,(x * y)(y - 1)\ \textbf{in}\ f\,1$$

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

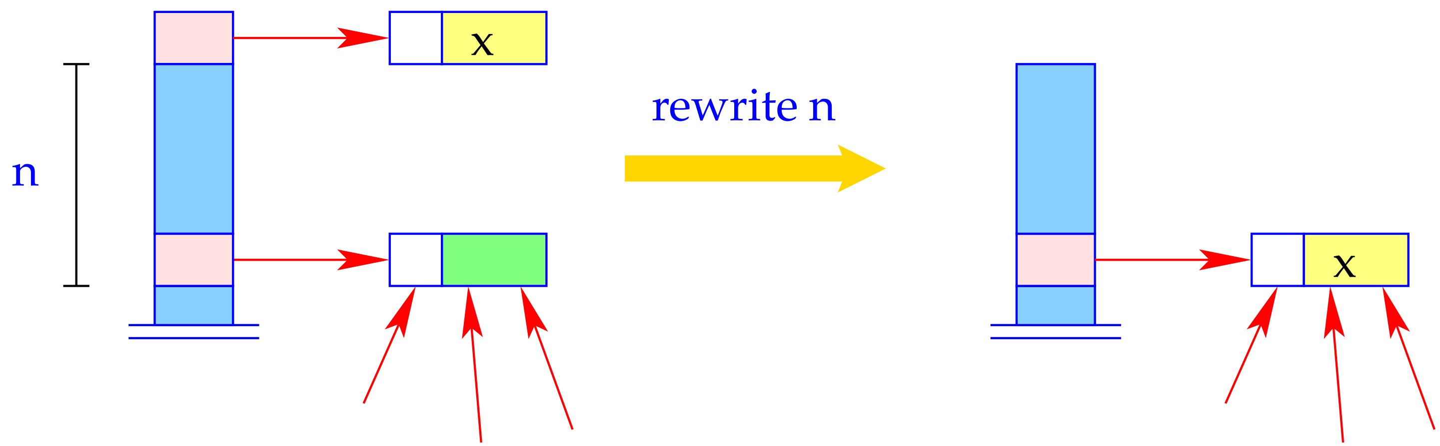
0	alloc 1	0	A:	targ 2	4		loadc 1
1	pushloc 0	0		...	5		mkbasic
2	mkvec 1	1		return 2	5		pushloc 4
2	mkfunval A	2	B:	rewrite 1	6		apply
2	jump B	1		mark C	2	C:	slide 1

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!

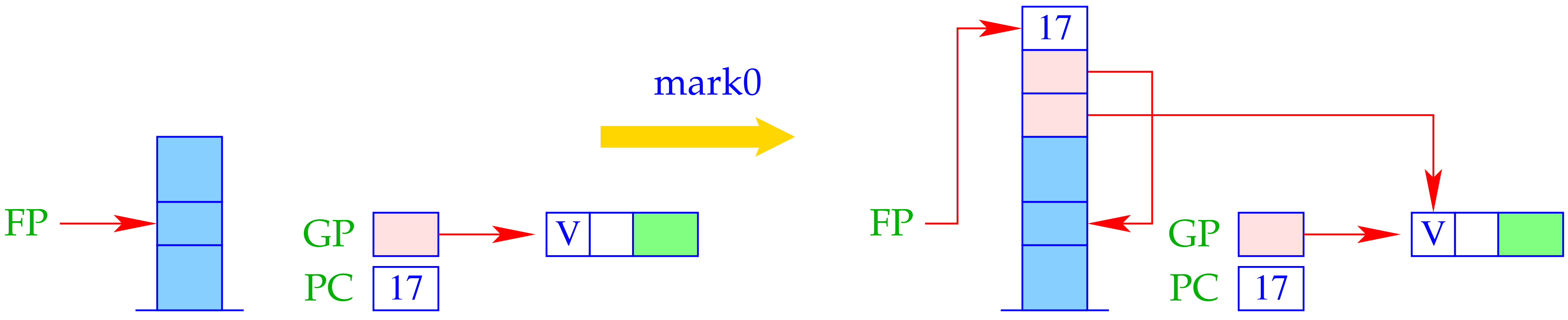
20 Closures and their Evaluation

- Closures are needed only for the implementation of CBN.
- 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.

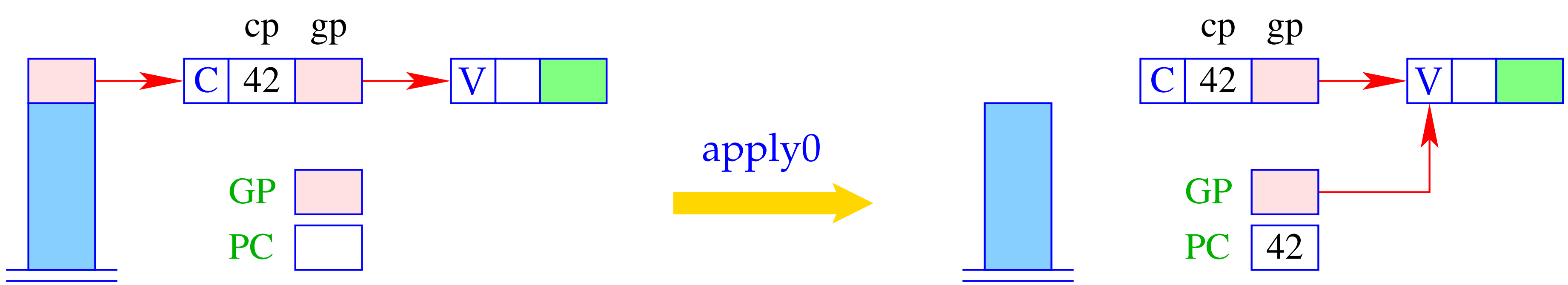
eval can be decompose into small actions:

$$\begin{array}{lll} \text{eval} & = & \text{if } (H[S[SP]] \equiv (C, \_, \_)) \; \{ \\ & & \quad \text{mark0}; \qquad\qquad // \text{ allocation of the stack frame} \\ & & \quad \text{pushloc } 3; \qquad // \text{ copying of the reference} \\ & & \quad \text{apply0}; \qquad\quad // \text{ corresponds to apply} \\ & & \} \end{array}$$

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure thus 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.

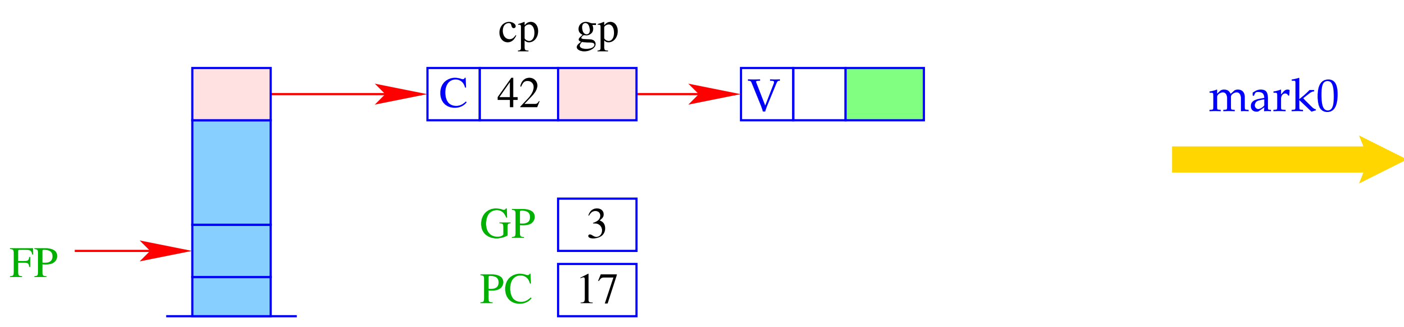


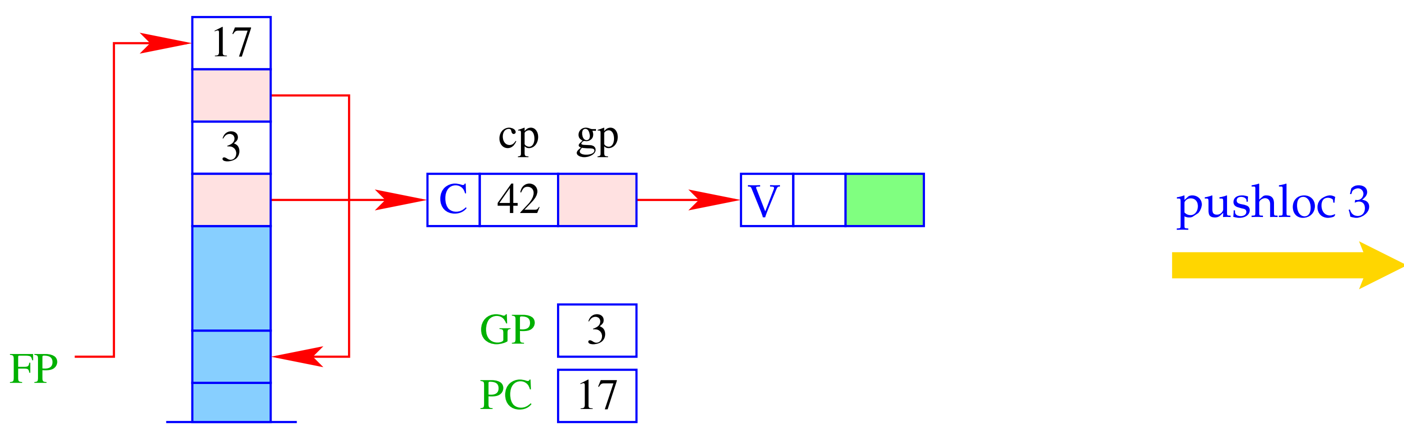
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

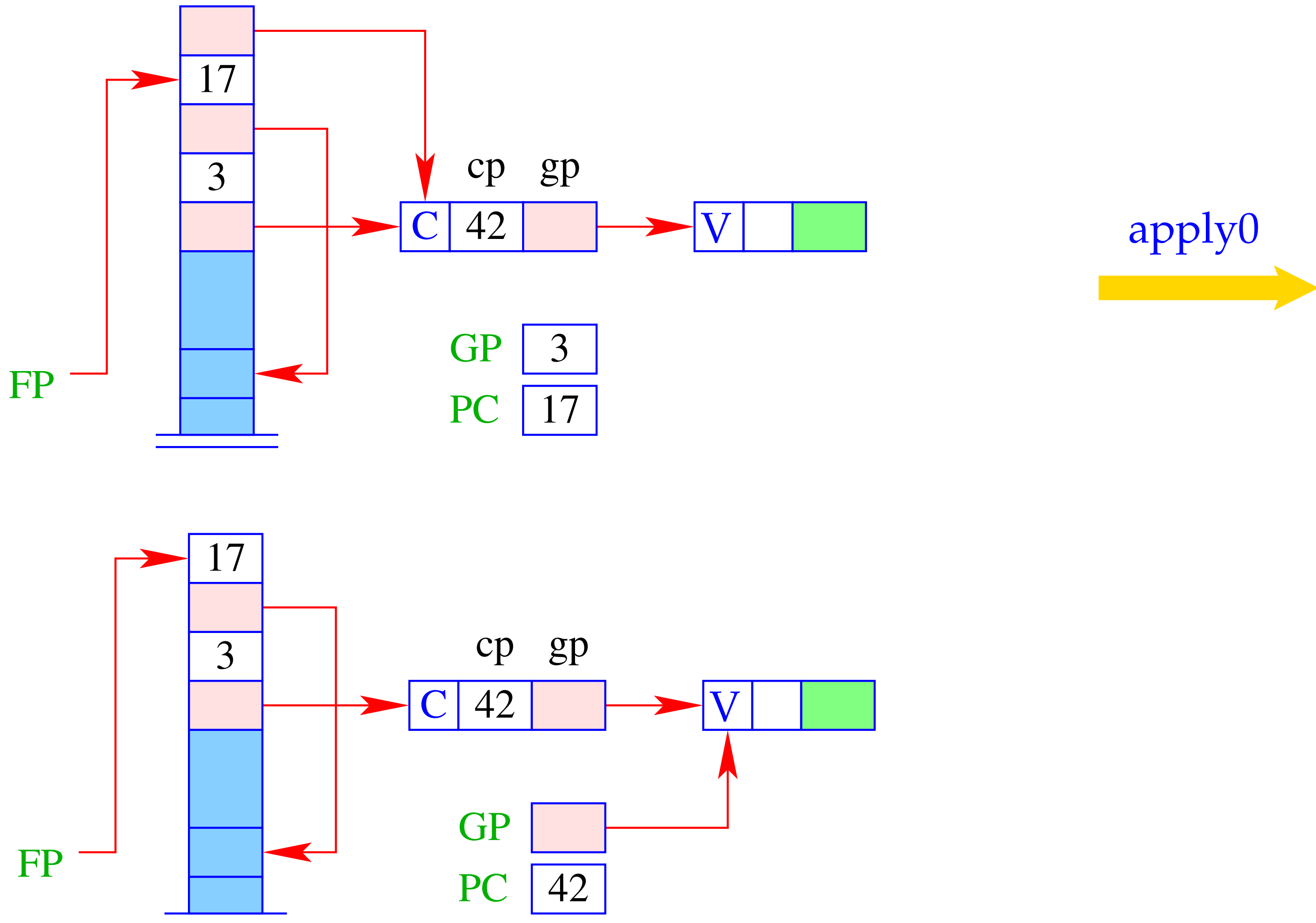


h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

We thus obtain for the instruction eval:


cp gp
C 42
V
FP
GP 3
PC 17
mark0


17
3
cp gp
C 42
V
FP
GP 3
PC 17
pushloc 3


17
3
cp gp
C 42
V
apply0
FP
GP 3
PC 17
17
3
cp gp
C 42
V
FP
GP
PC 42

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:

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

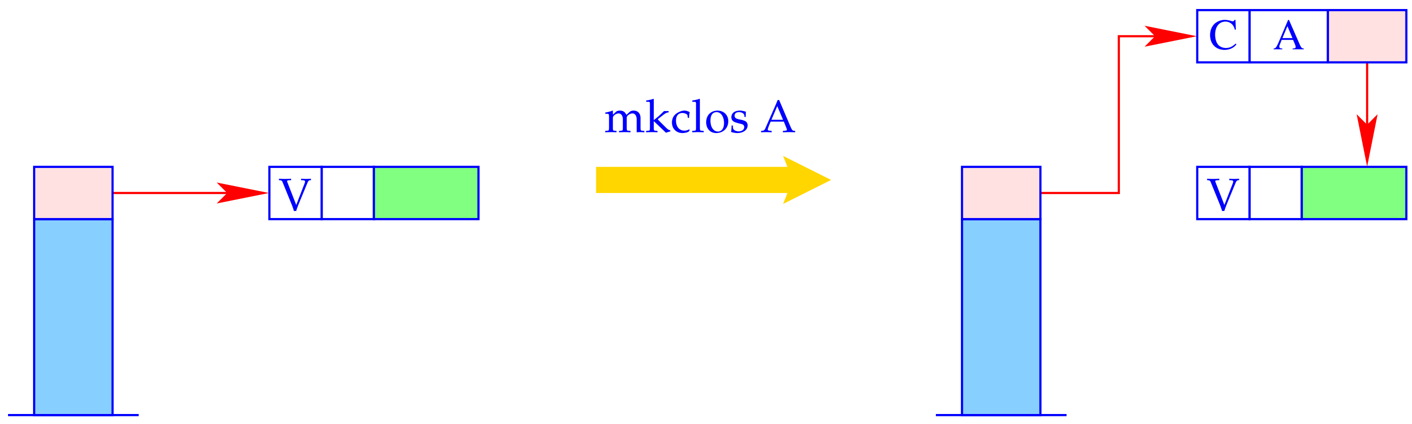
where $\quad \{z_0, \ldots, z_{g-1}\} = \mathit{free}(e) \quad$ and $\quad \rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Example:

Consider $e \equiv a * a$ with $\rho = \{a \mapsto (L, 0)\}$ and 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.