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.

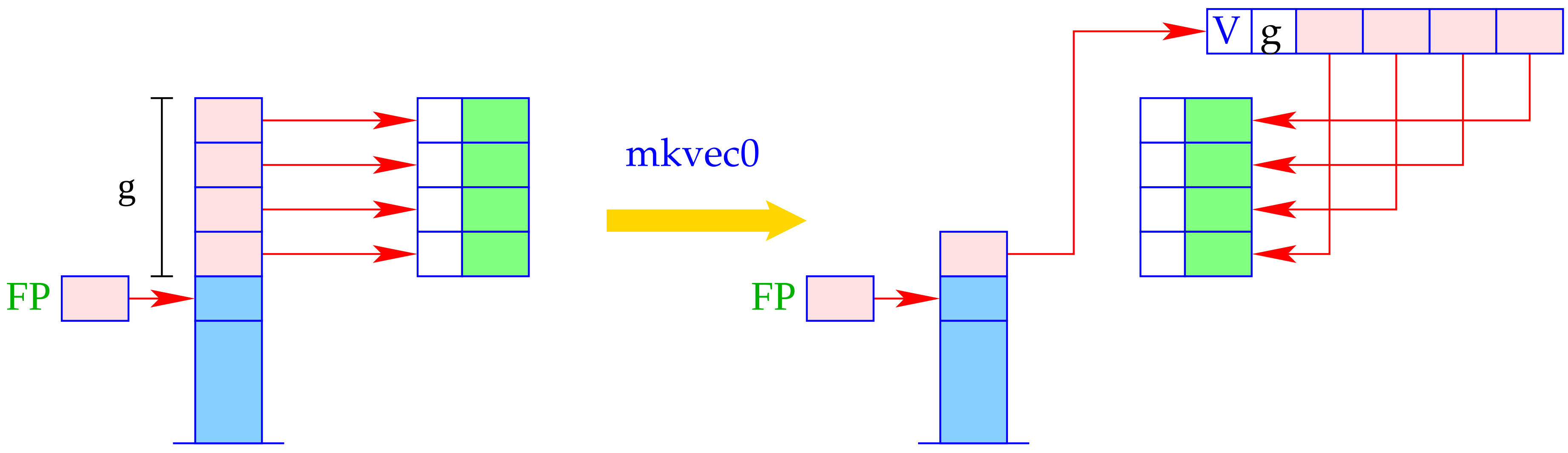
The test for number of arguments uses: SP – FP targ k   is a complex instruction.

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

$$
\begin{array}{lll}
\text{targ k} & = & \text{if (SP} - \text{FP} < \text{k) \{} \\
& & \quad \text{mkvec0;} \qquad\qquad // \text{ creating the argumentvector} \\
& & \quad \text{wrap;} \qquad\qquad\quad // \text{ wrapping into an } F - \text{object} \\
& & \quad \text{popenv;} \qquad\qquad // \text{ popping the stack frame} \\
& & \text{\}}
\end{array}
$$

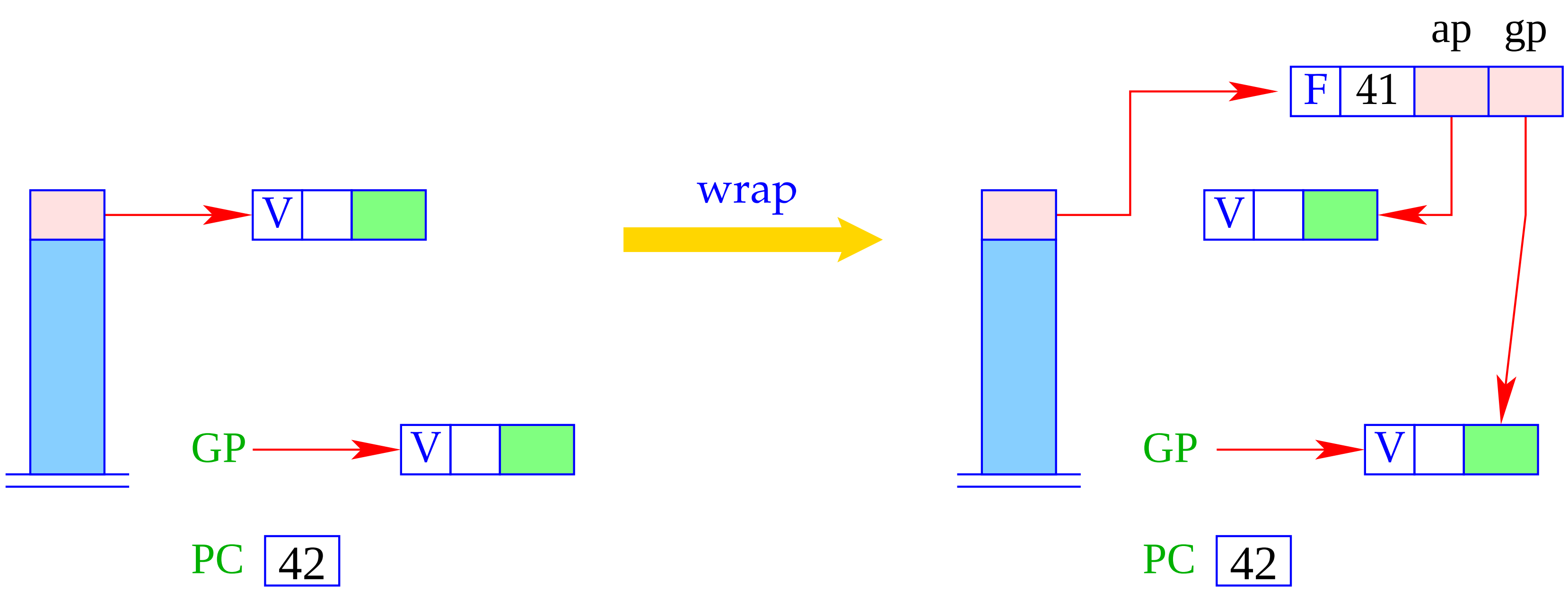
The combination of these steps into one instruction is a kind of optimization   :-)

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



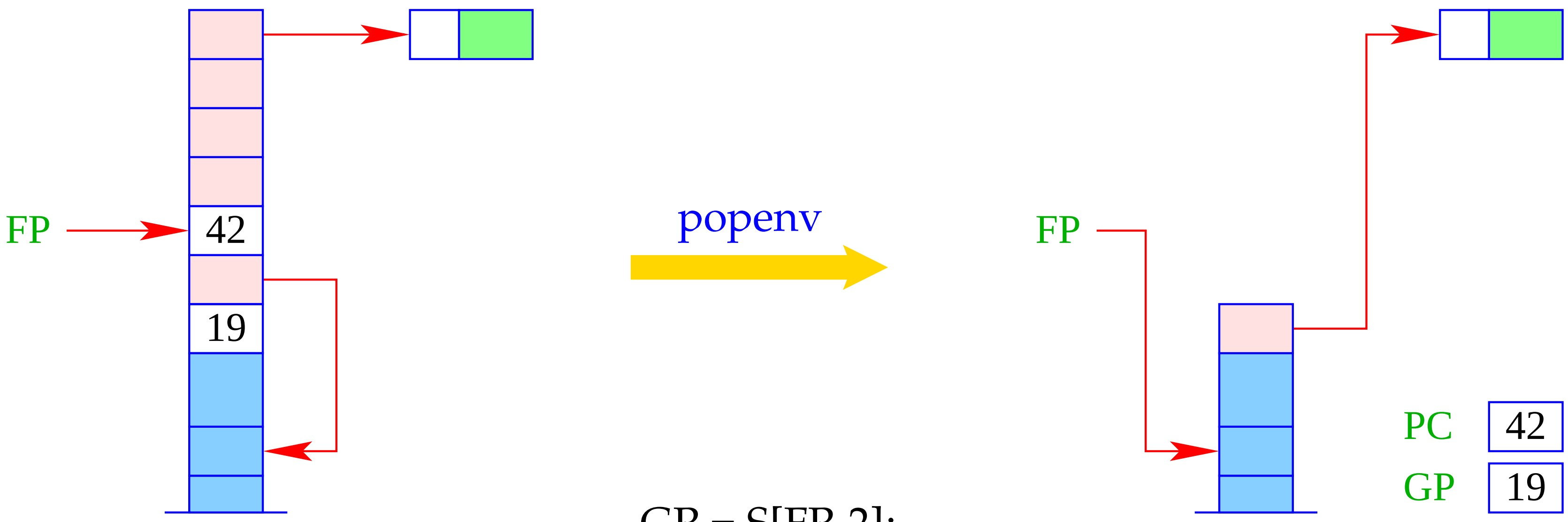
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;

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



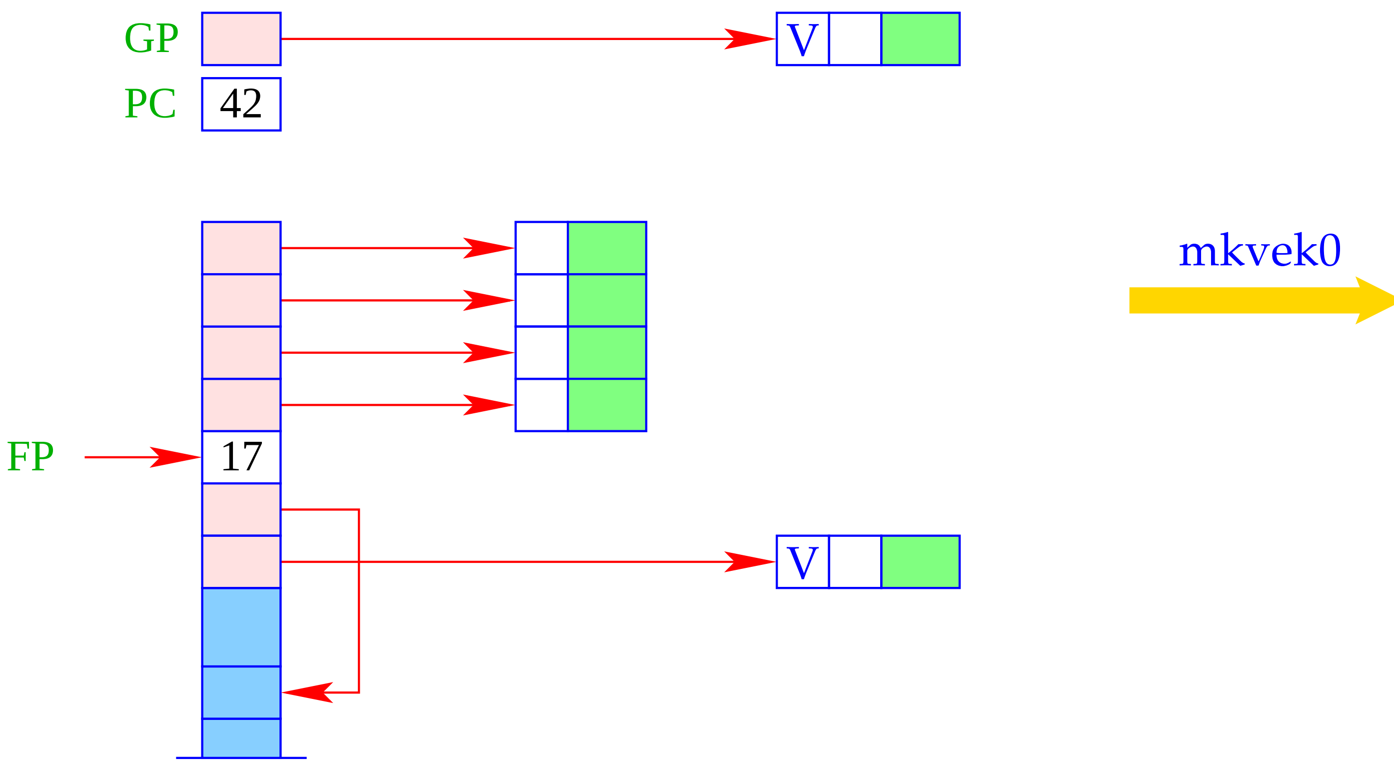
S[SP] = new (F, PC-1, S[SP], GP);

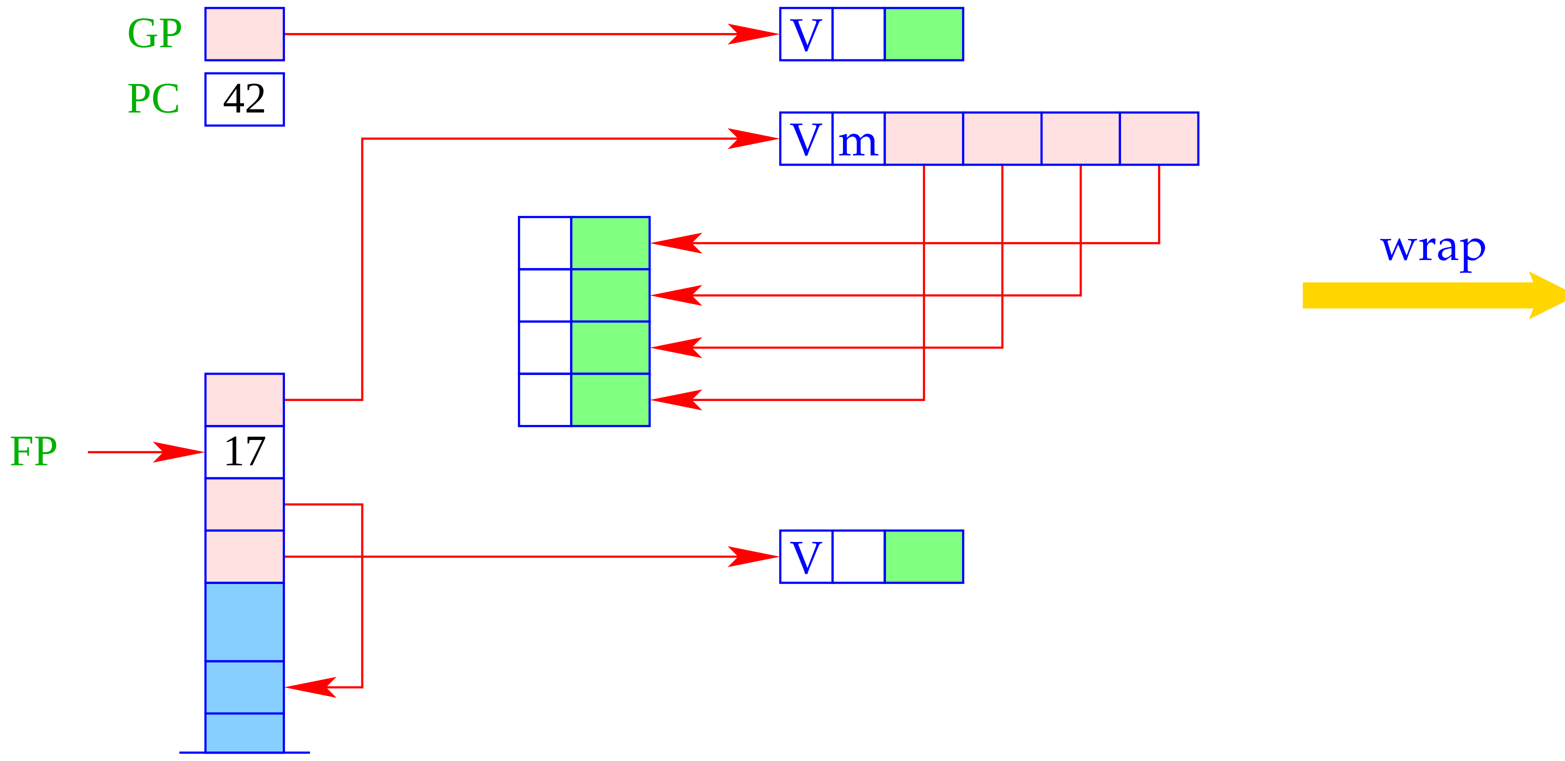
The instruction popenv finally releases the stack frame:

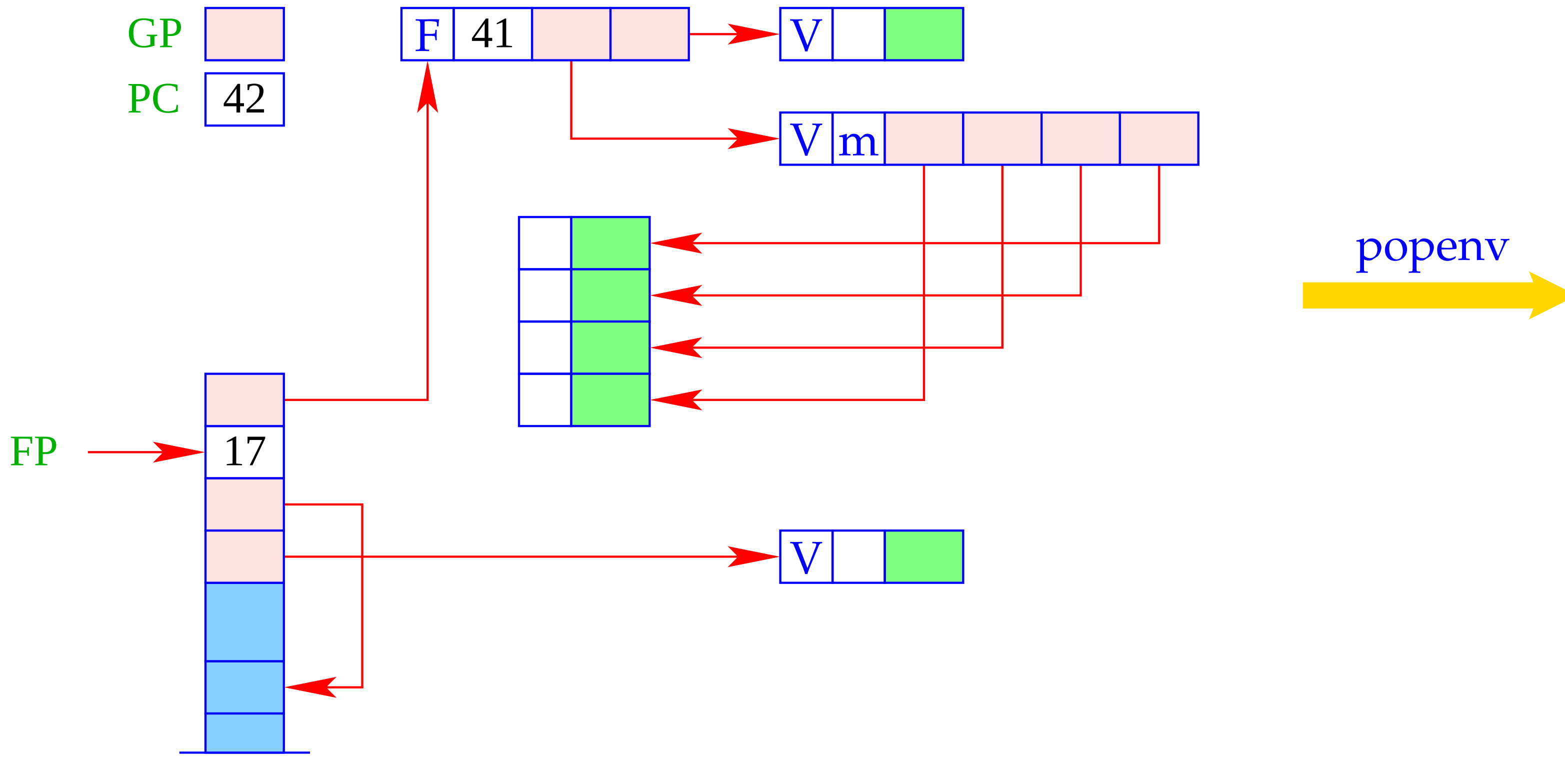


GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];

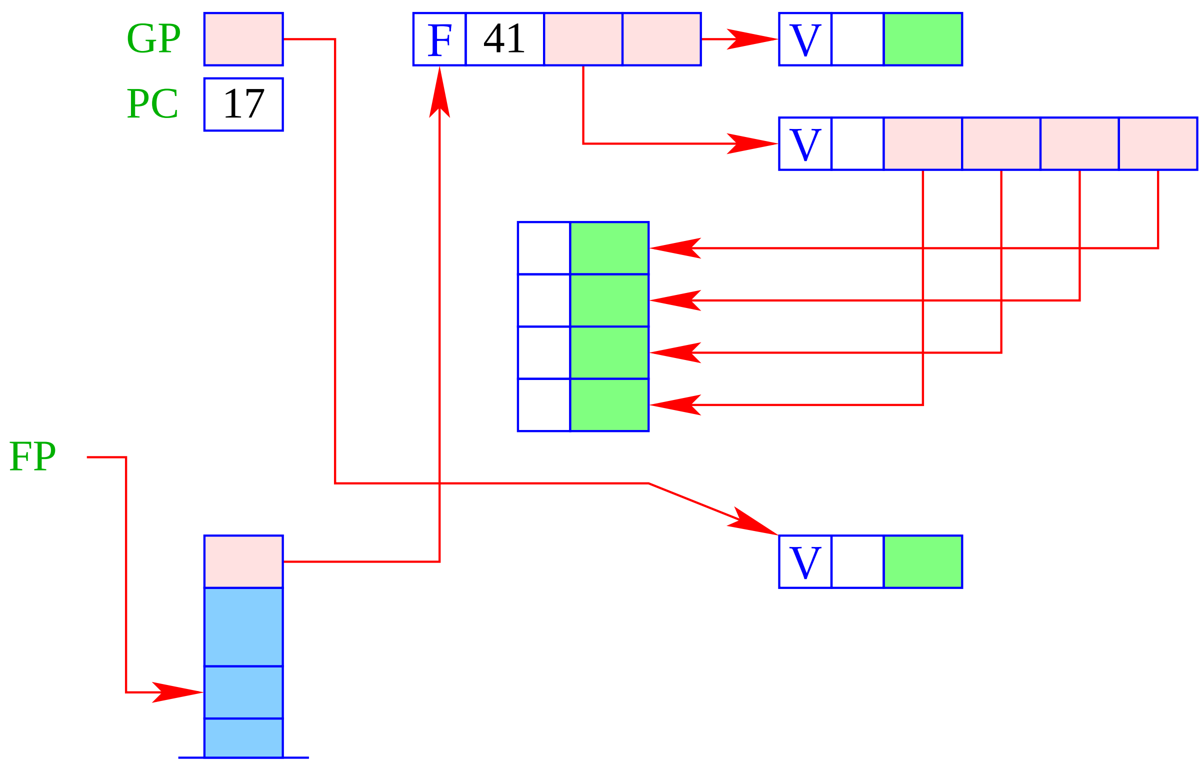
Thus, we obtain for targ k in the case of under supply:




GP
PC
42
V
V
m
wrap
FP
17
V



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

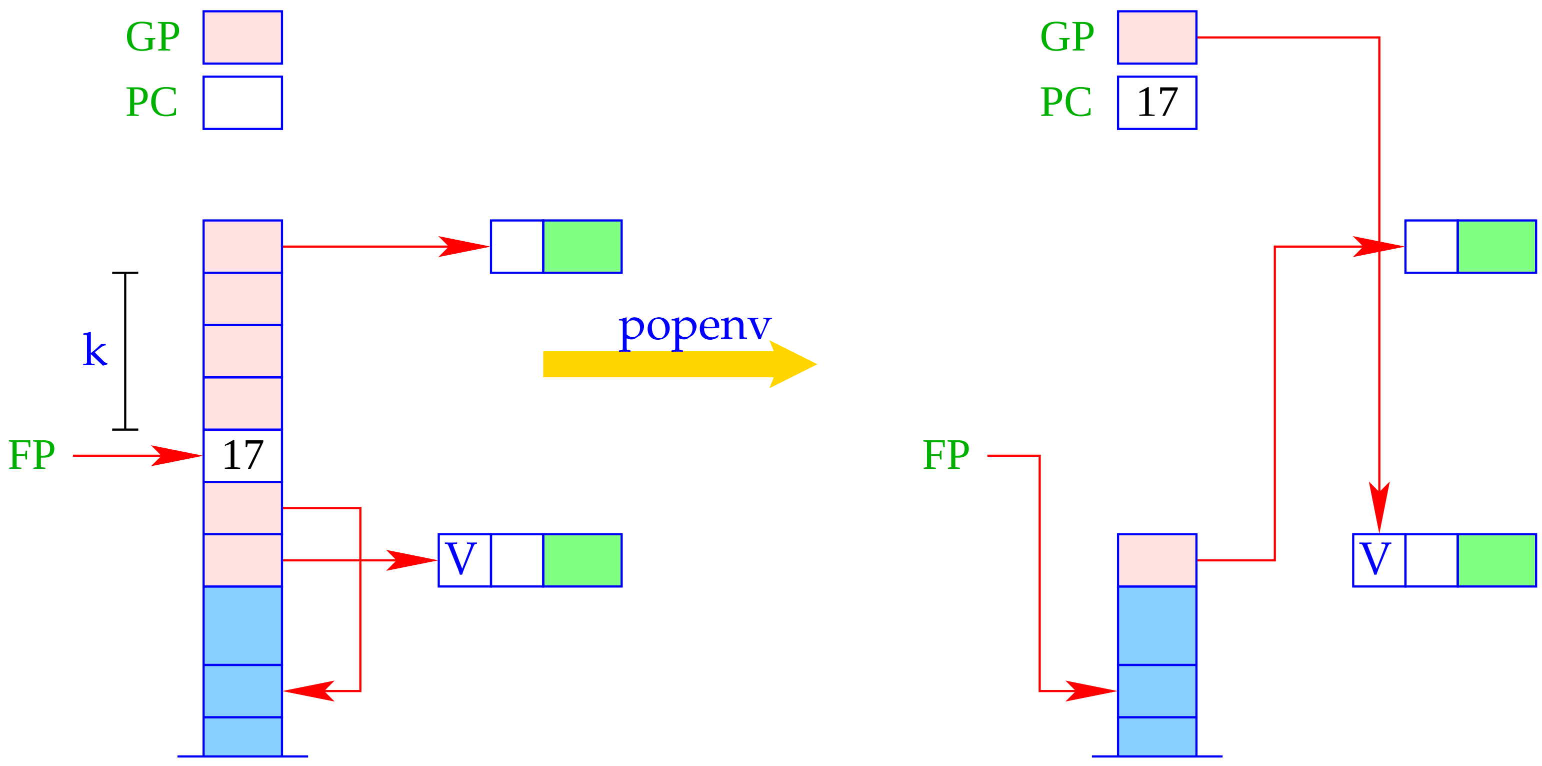

GP
PC
17
F
41
V
V
V
FP

- 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:

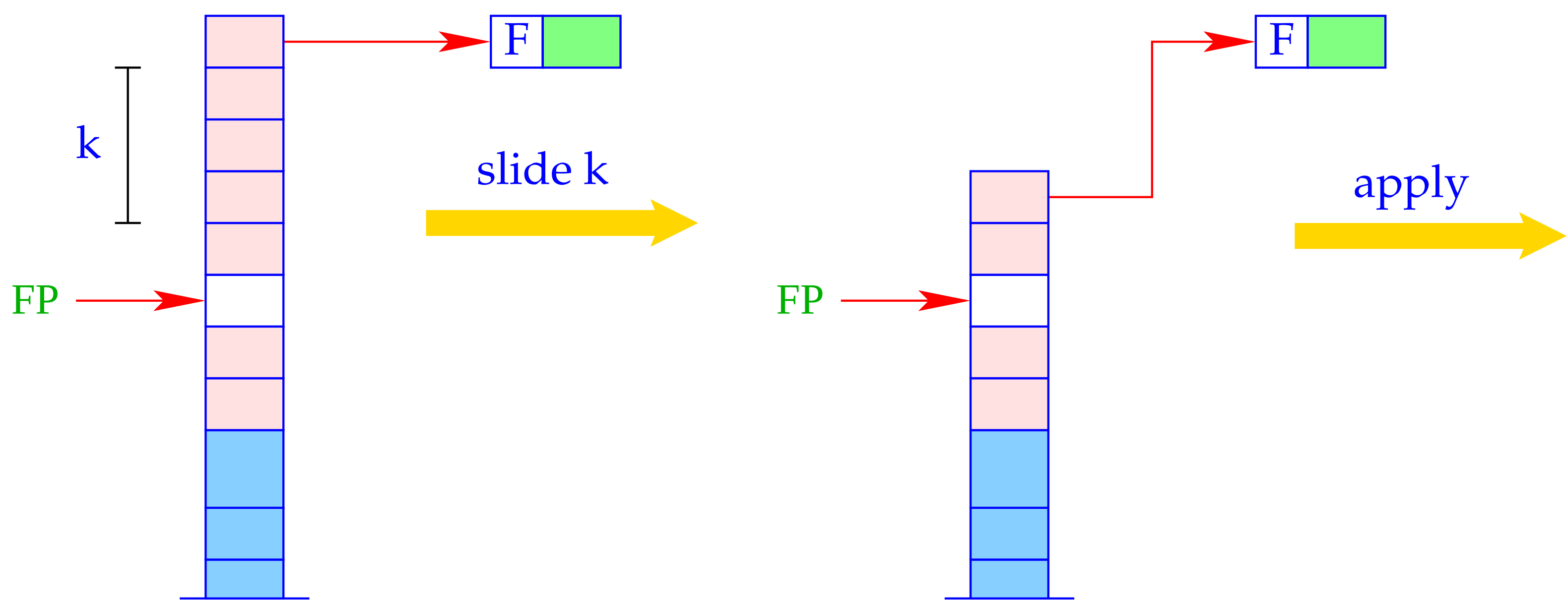
$$
\begin{array}{lll}
\text{return k} & = & \text{if } (\text{SP} - \text{FP} = \text{k} + 1) \\
& & \quad \text{popenv}; \qquad\qquad\qquad // \text{ Done} \\
& & \text{else } \{ \qquad\qquad\qquad\quad // \text{ There are more arguments} \\
& & \quad \text{slide k}; \\
& & \quad \text{apply}; \qquad\qquad\qquad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

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.

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!!!

Example:

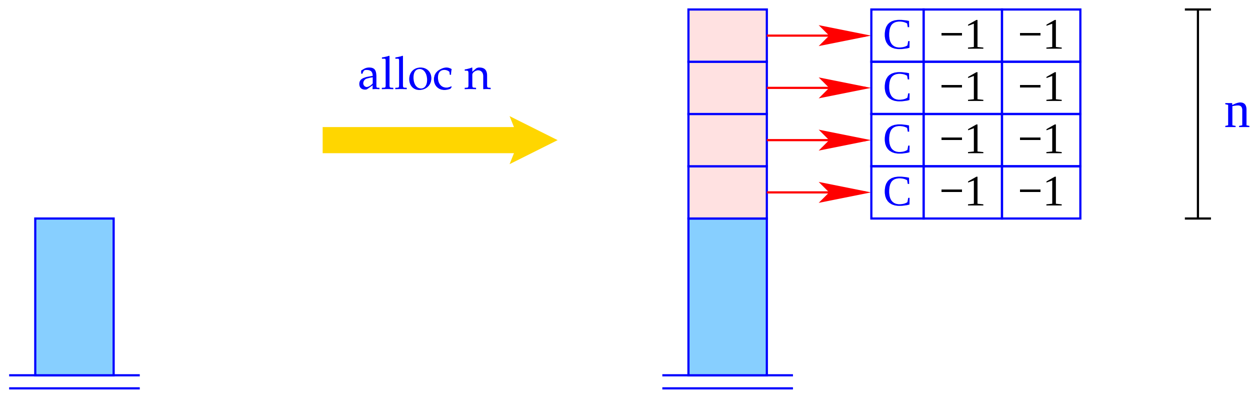
Consider the expression

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

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

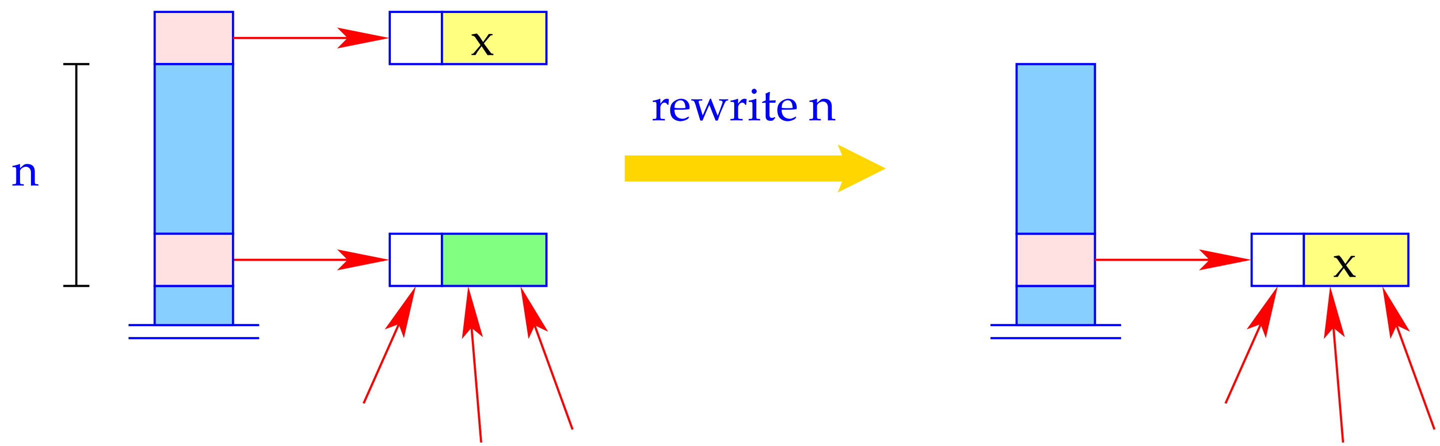
0		alloc 1	0	A:	targ 2	4		loadc 1
1		pushloc 0	0		...	5		mkbasic
2		mkvec 1	1		return 2	5		pushloc 4
2		mkfunval A	2	B:	rewrite 1	6		apply
2		jump B	1		mark C	2	C:	slide 1

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!

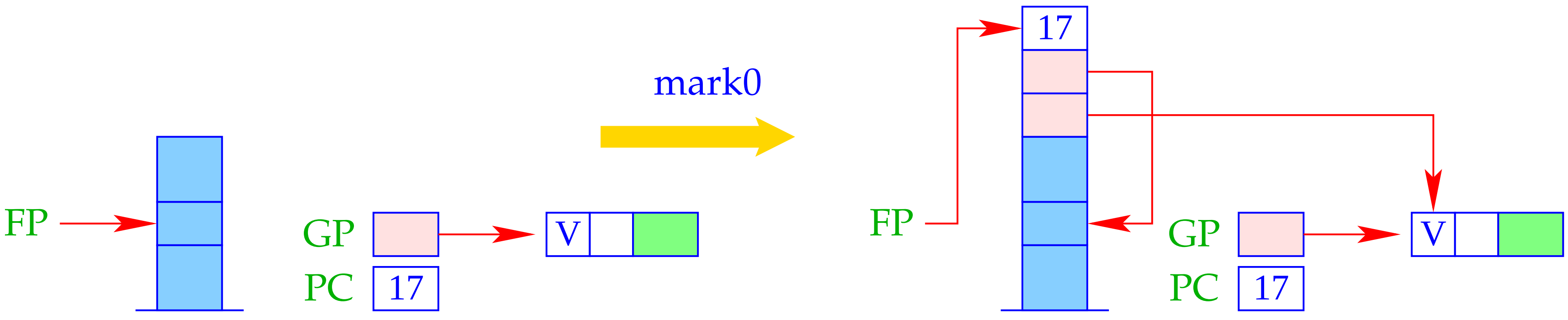
20 Closures and their Evaluation

- Closures are needed for the implementation of CBN and for functional paramaters.
- 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.

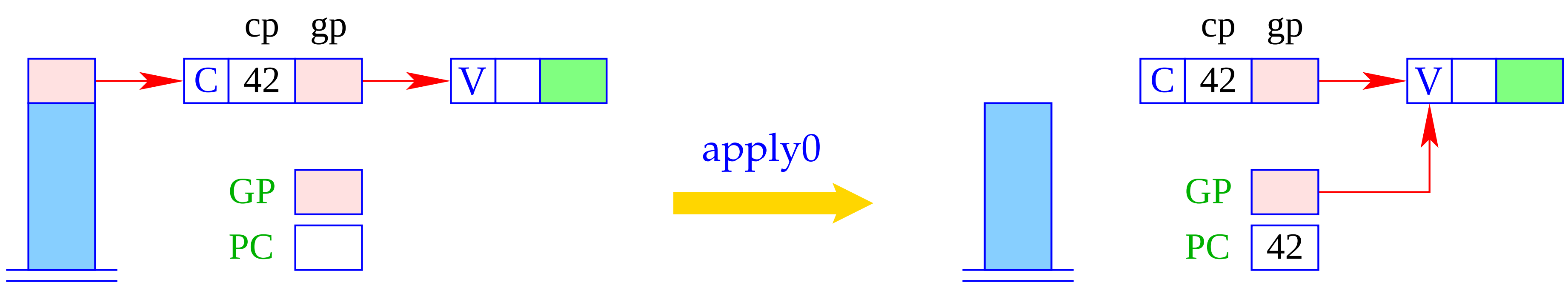
eval can be decomposed into small actions:

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

- 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.

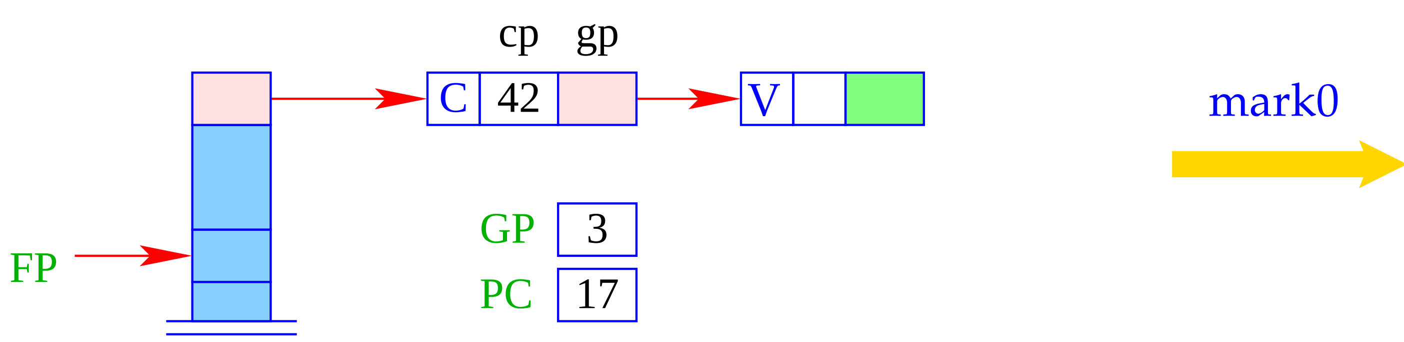


S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

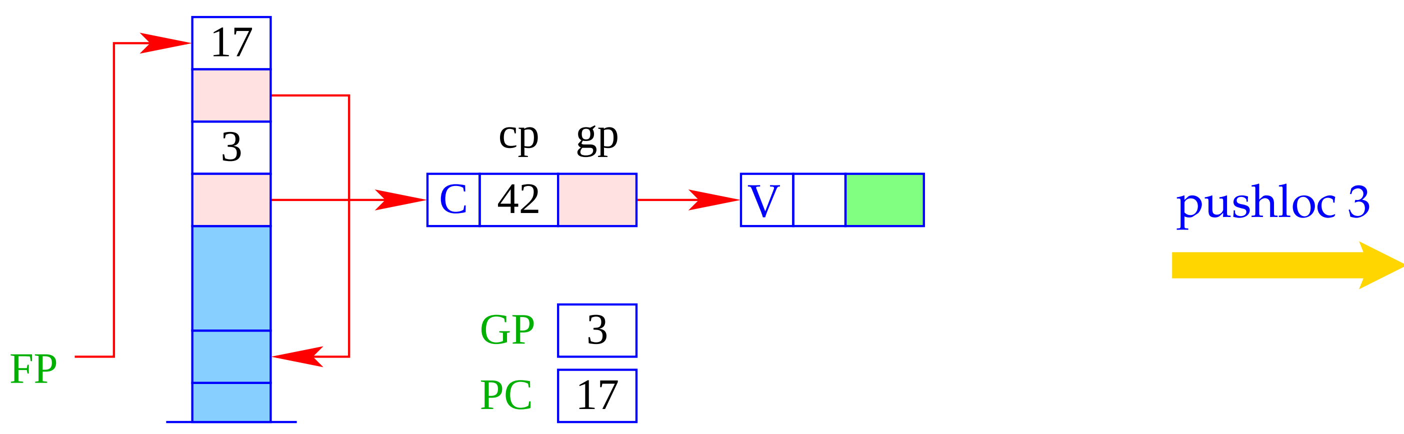


h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

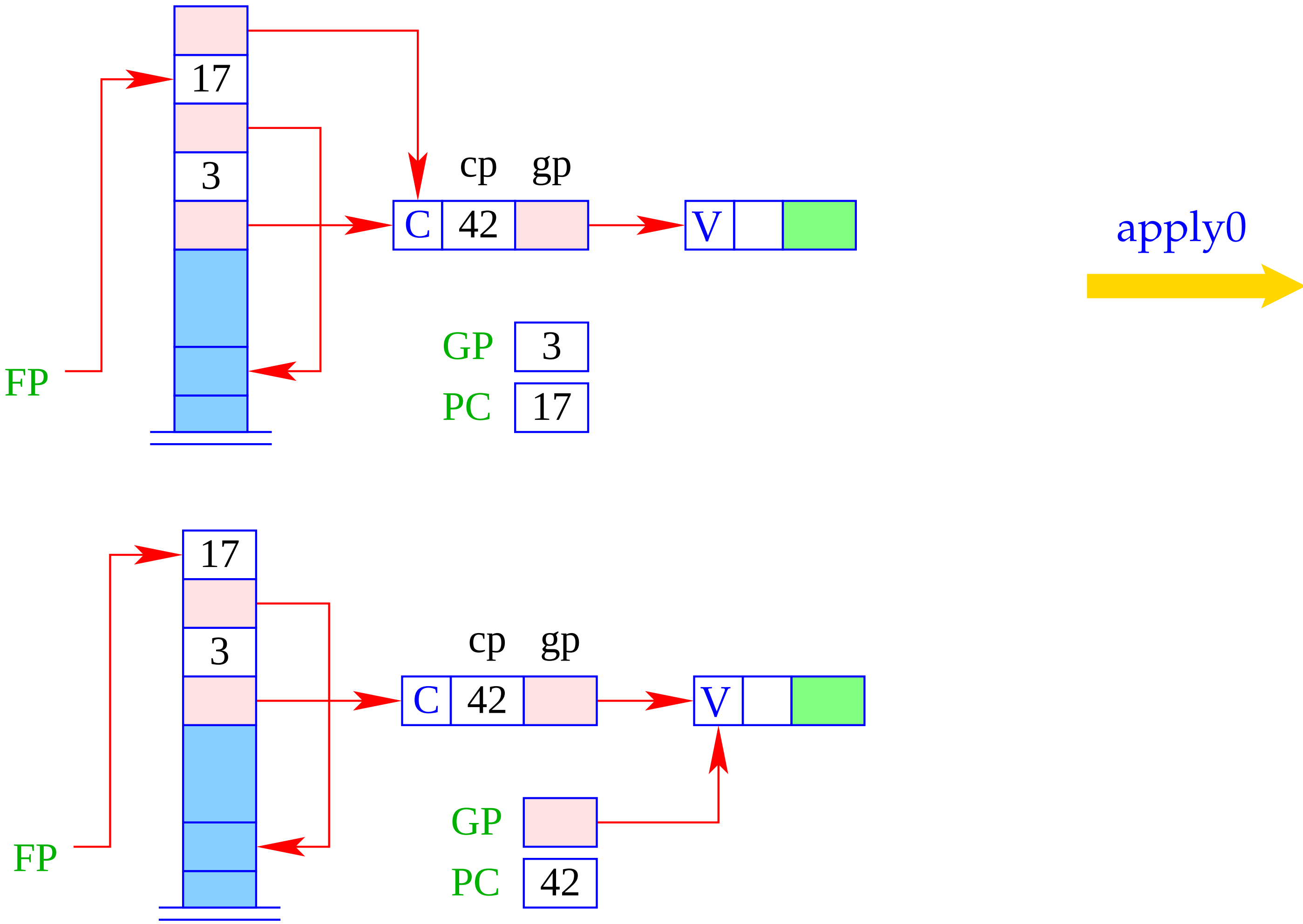
We thus obtain for the instruction eval:



cp gp
C 42
V
mark0
FP
GP 3
PC 17



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


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

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:

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

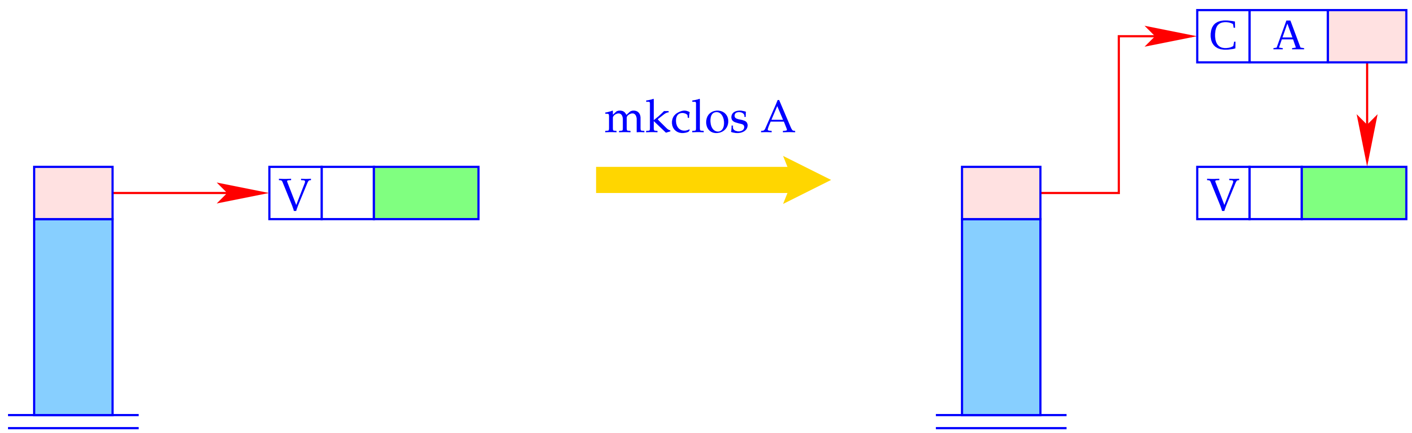
where $\{z_0, \ldots, z_{g-1}\} = \mathit{free}(e)$ and $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g-1\}$.

Example:

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



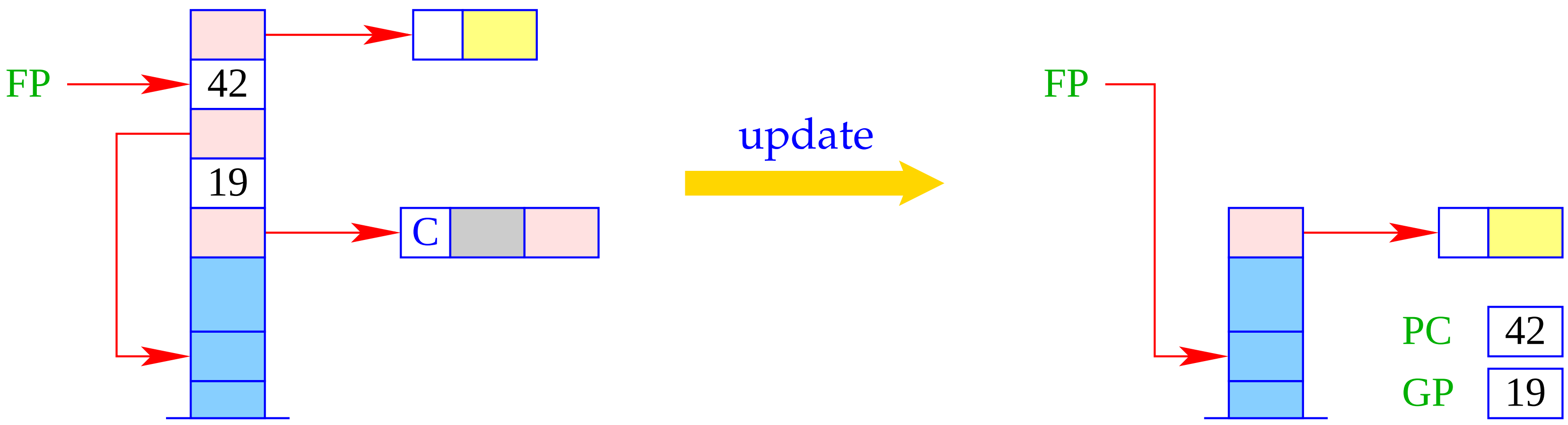
S[SP] = new (C, A, S[SP]);

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

popenv

rewrite 1

It overwrites the closure with the computed value.



21 Optimizations I: Global Variables

Observation:

- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables.

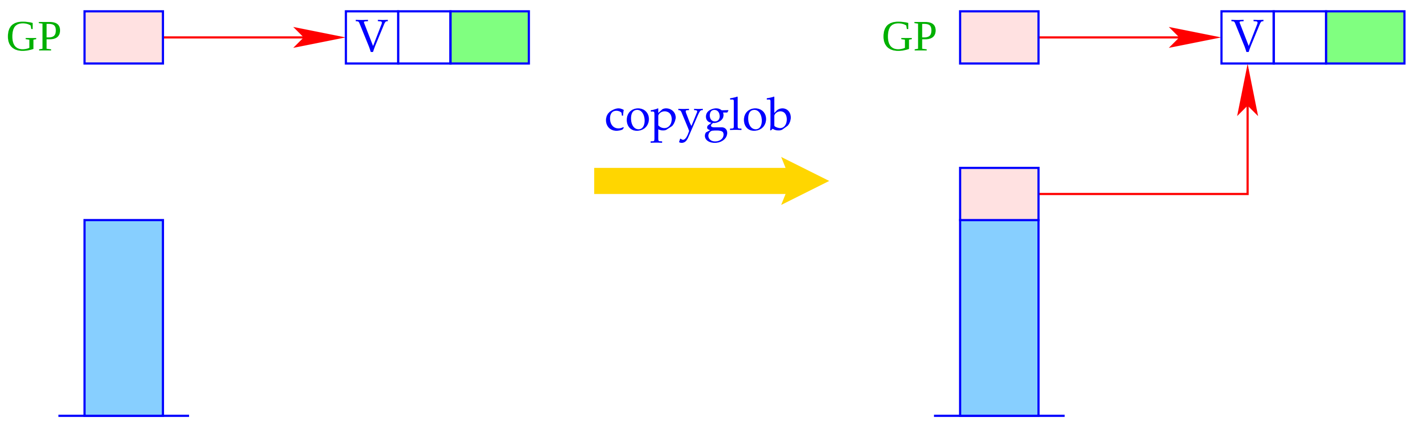
  Recall, e.g., the construction of a closure for an expression e ...

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

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

Idea:

- Reuse Global Vectors, i.e. share Global Vectors!
- Profitable in the translation of **let**-expressions or function applications: Build one Global Vector for the union of the free-variable sets of all let-definitions resp. all arguments.
- Allocate (references to) global vectors with multiple uses in the stack frame like local variables!
- Support the access to the current GP by an instruction copyglob :



SP++;
S[SP] = GP;

- The optimization will cause Global Vectors to contain more components than just references to the free the variables that occur in one expression ...

Disadvantage: Superfluous components in Global Vectors prevent the deallocation of already useless heap objects $\Longrightarrow$ Space Leaks :-(

Potential Remedy: Deletion of references at the end of their life time.