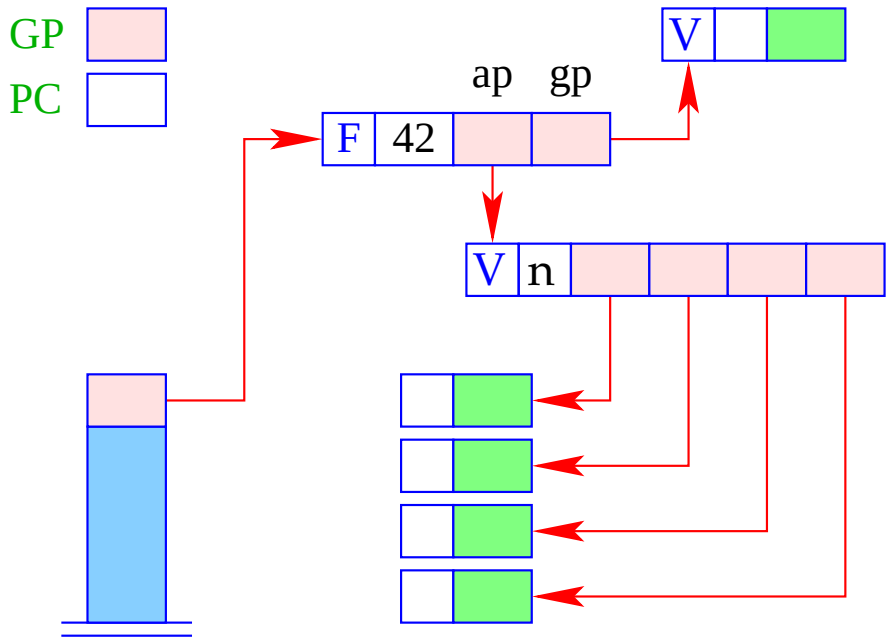
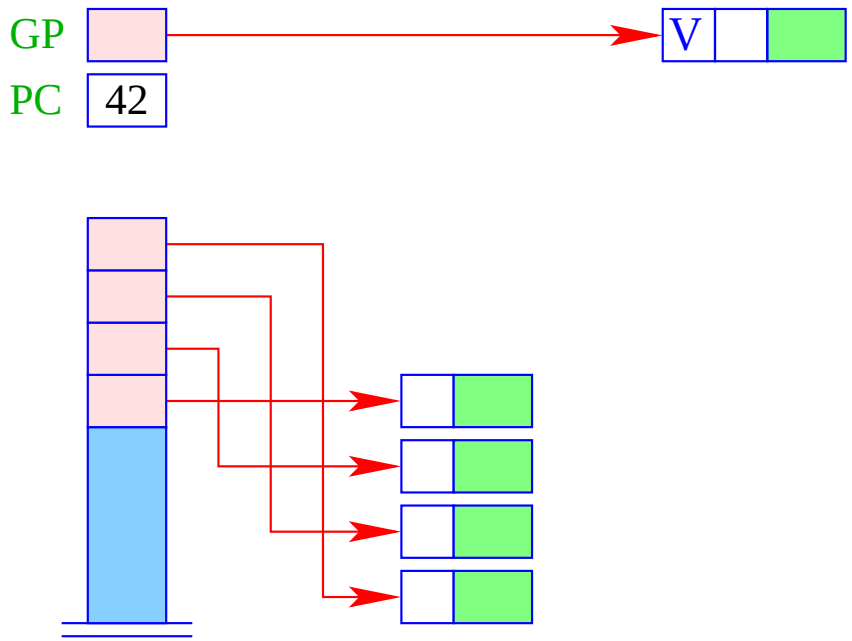
- Im Gegensatz zur CMa rettet hier bereits der mark-Befehl die Adresse, an der die Programm-Ausführung nach der Abarbeitung der Funktions-Anwendung fortfahren soll.
- Der apply-Befehl muss das F-Objekt, auf das (hoffentlich) oben auf dem Keller ein Verweis liegt, auspacken und an der dort angegebenen Adresse fortfahren.



apply

```
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;
}
```


GP
PC
42
V

Achtung!

- Das 0-te Element des Argument-Vektors legen wir zuerst auf den Keller. Dieses muss also die äußerste Argument-Referenz darstellen.
- Das müssen wir berücksichtigen, wenn wir die Argumente einer unterversorgten Funktions-Anwendung zu einem F-Objekt einpacken!!!

18 Unter- und Überversorgung mit Argumenten

Der erste Befehl, der nach einem apply ausgeführt wird, ist targ k.

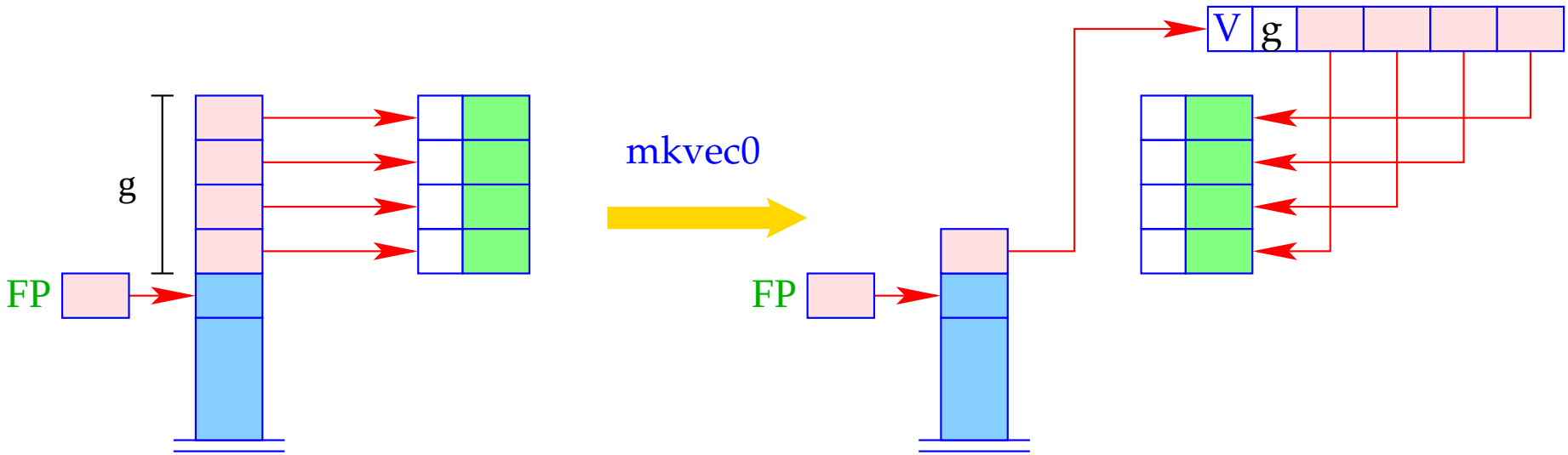
Er überprüft, ob bereits genügend Argumente vorliegen, um den Funktions-Rumpf auszuführen. Die Anzahl der Argumente ist: SP – FP .

Sind nicht genügend Argumente vorhanden, wird als Ergebnis ein neues F-Objekt zurückgeliefert. Andernfalls soll der Rumpf normal betreten werden.

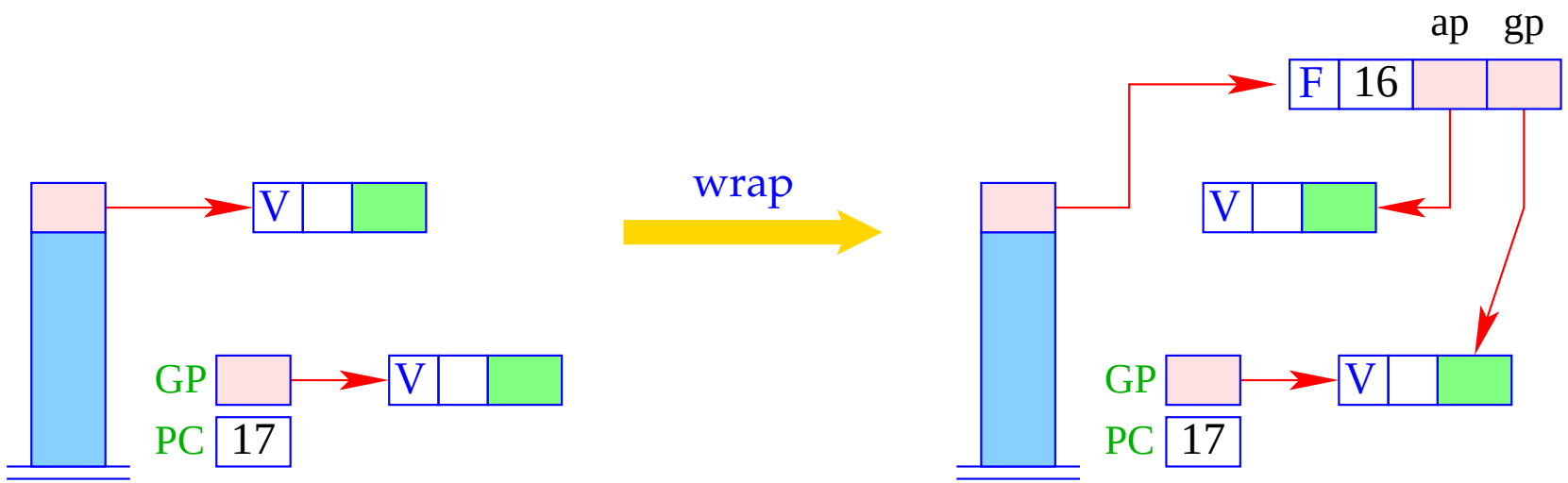
targ k ist ein komplizierter Befehl. Darum zerlegen wir seine Ausführung in mehrere Schritte:

```
targ k  =  if (SP – FP < k) {
             mkvec0;          // Anlegen des Argument – Vektors
             wrap;            // Anlegen des F – Objekts
             popenv;          // Aufgeben des Kellerrahmens
           }
```

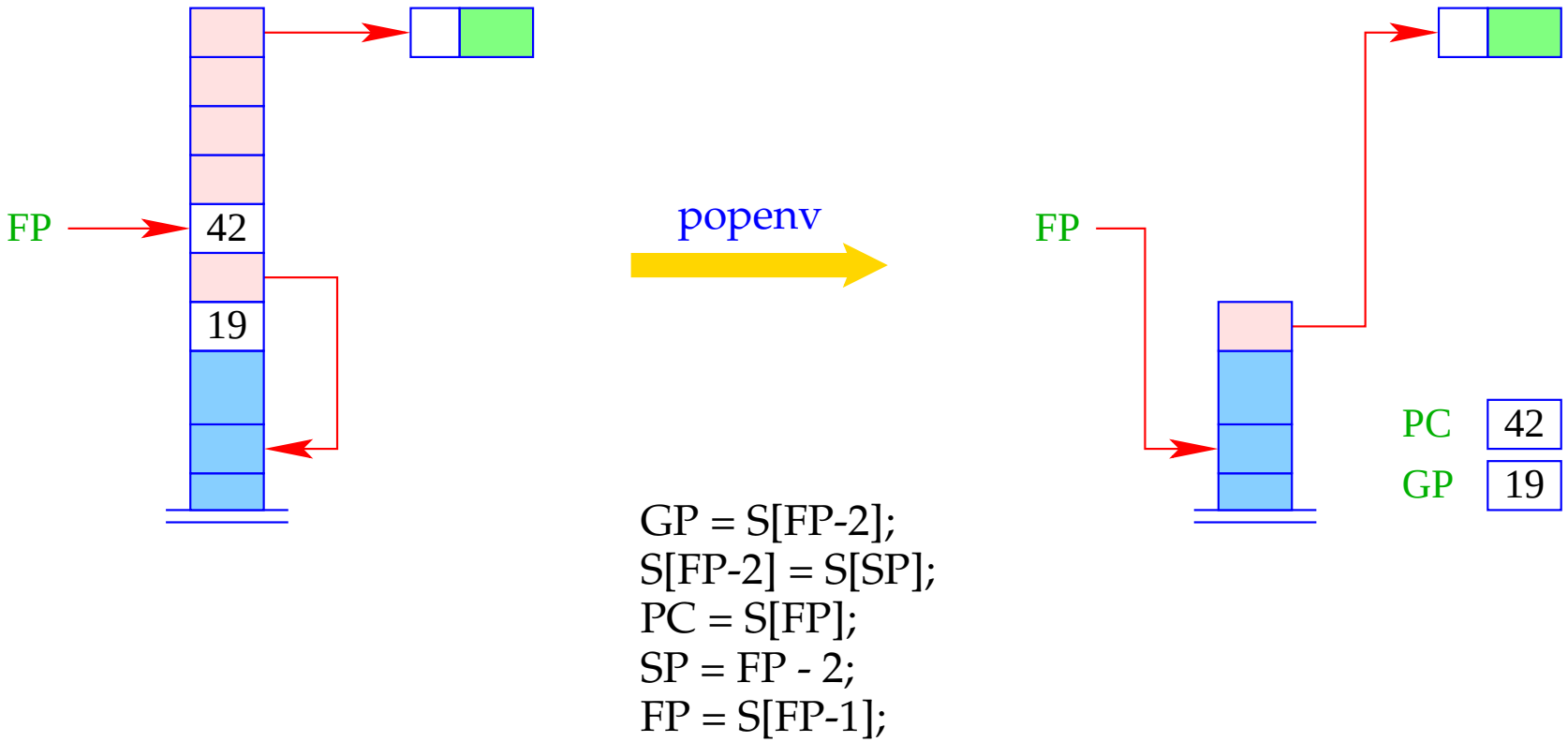
Die Zusammenfassung dieser festen Schritt-Abfolge zu einem Befehl kann als eine Art Optimierung verstanden werden :-)




```
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;
```

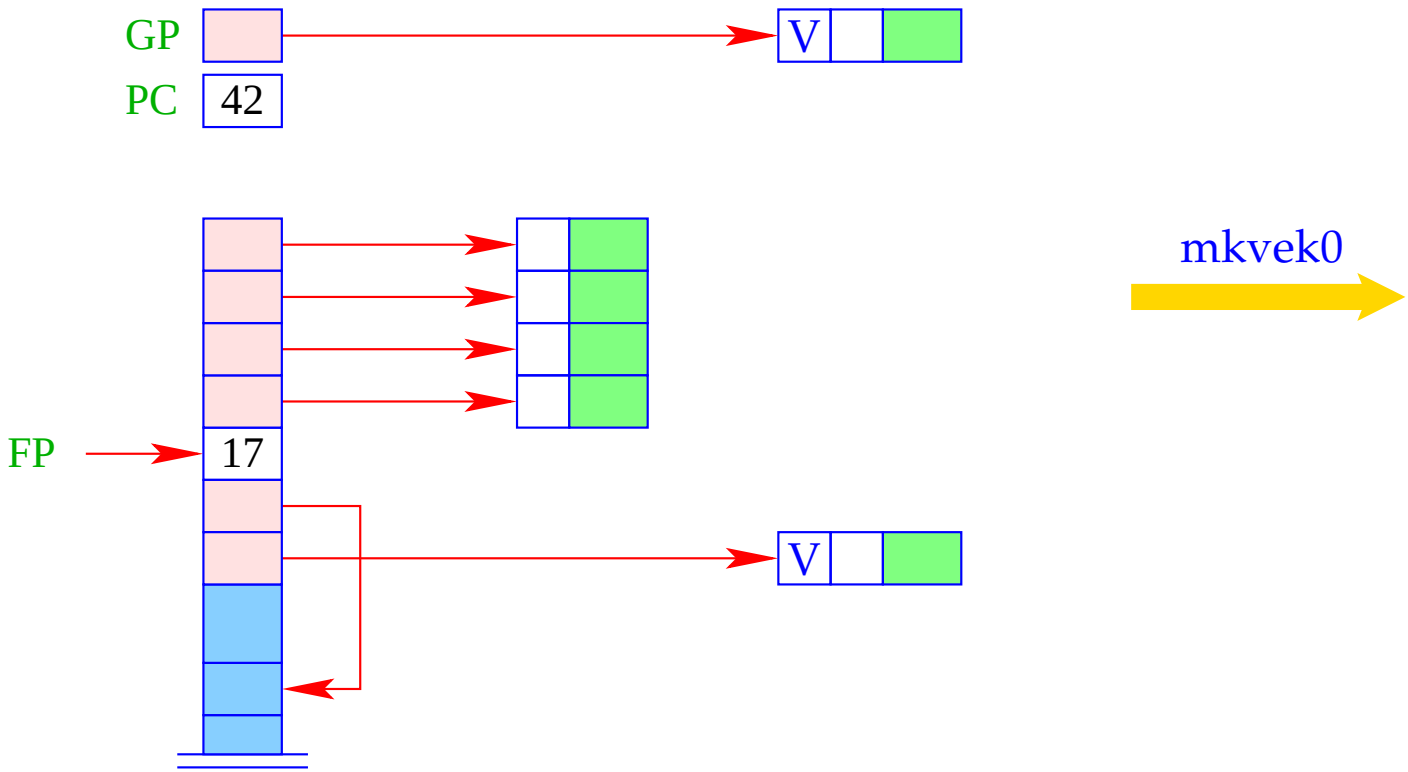


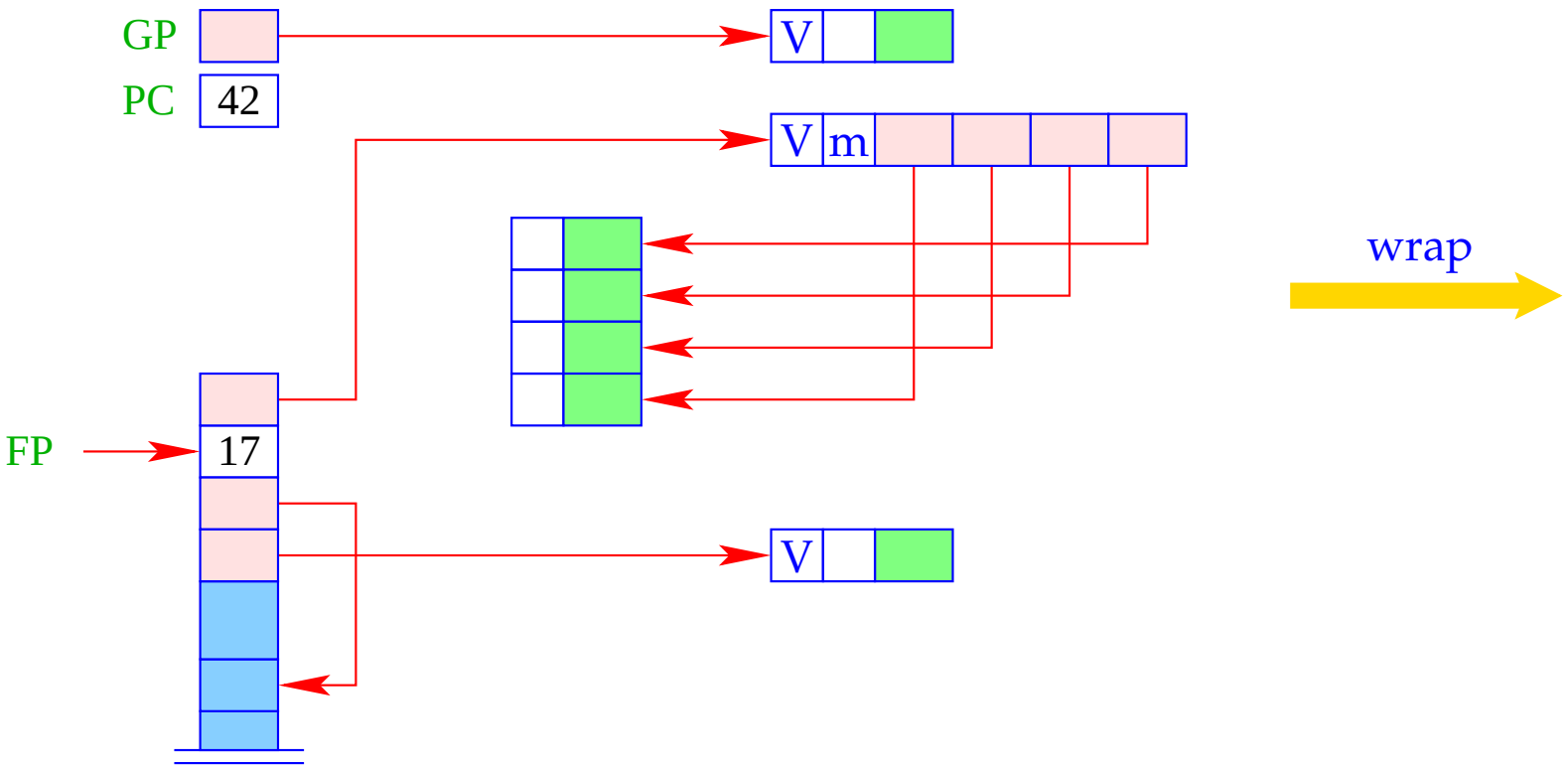
S[SP] = new (F, PC–1, S[SP], GP);

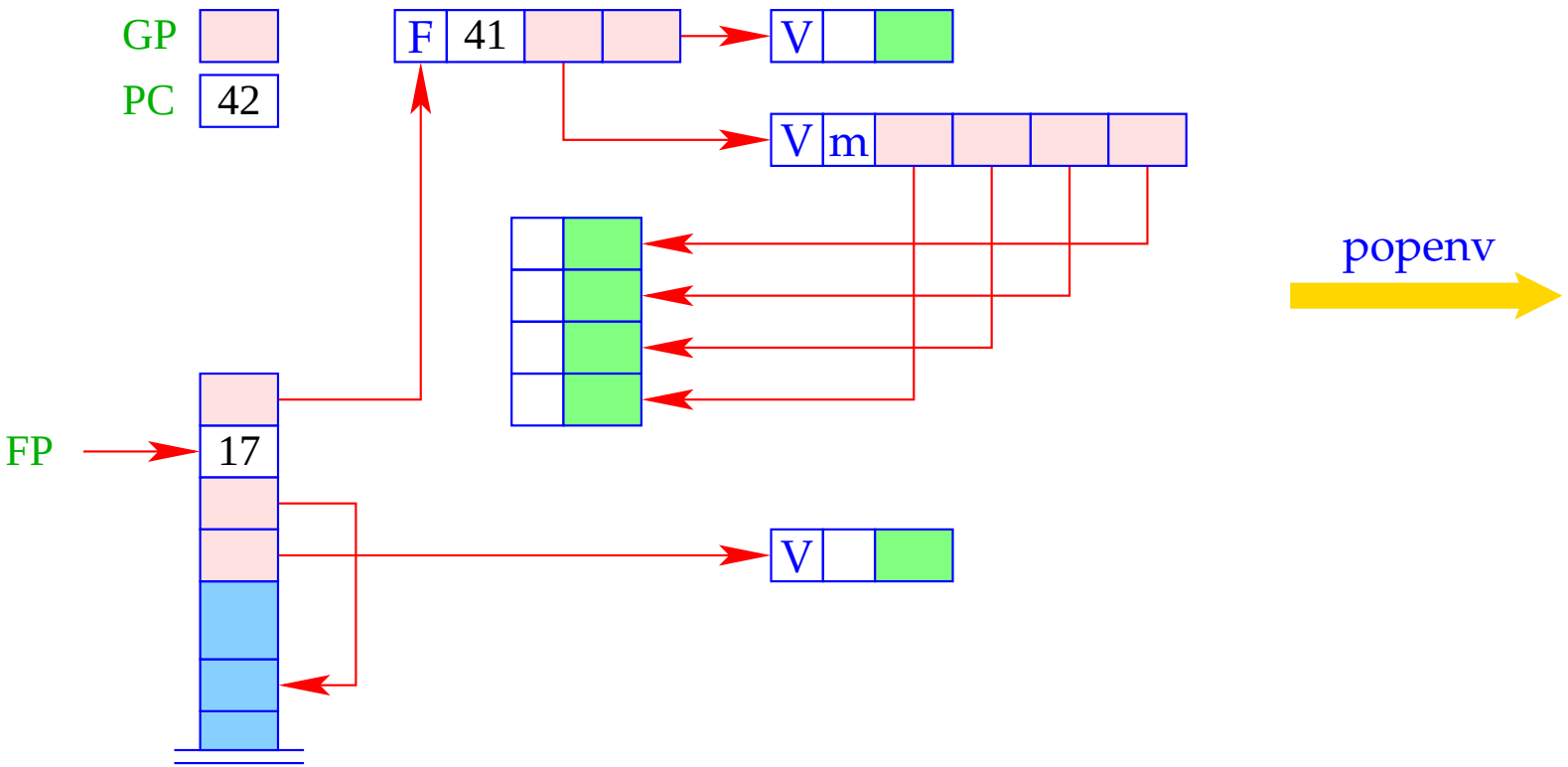


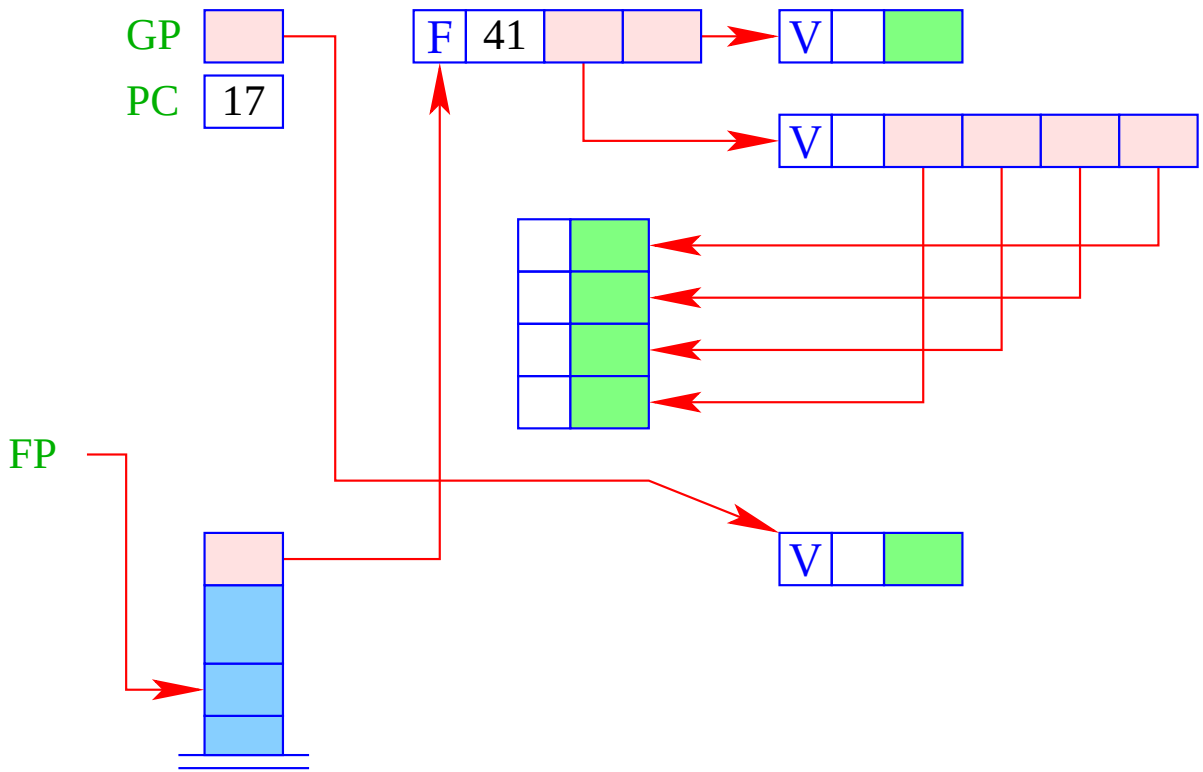
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];

Insgesamt erhalten wir damit für targ k:




GP
PC
42
V
V
m
wrap
FP
17
V

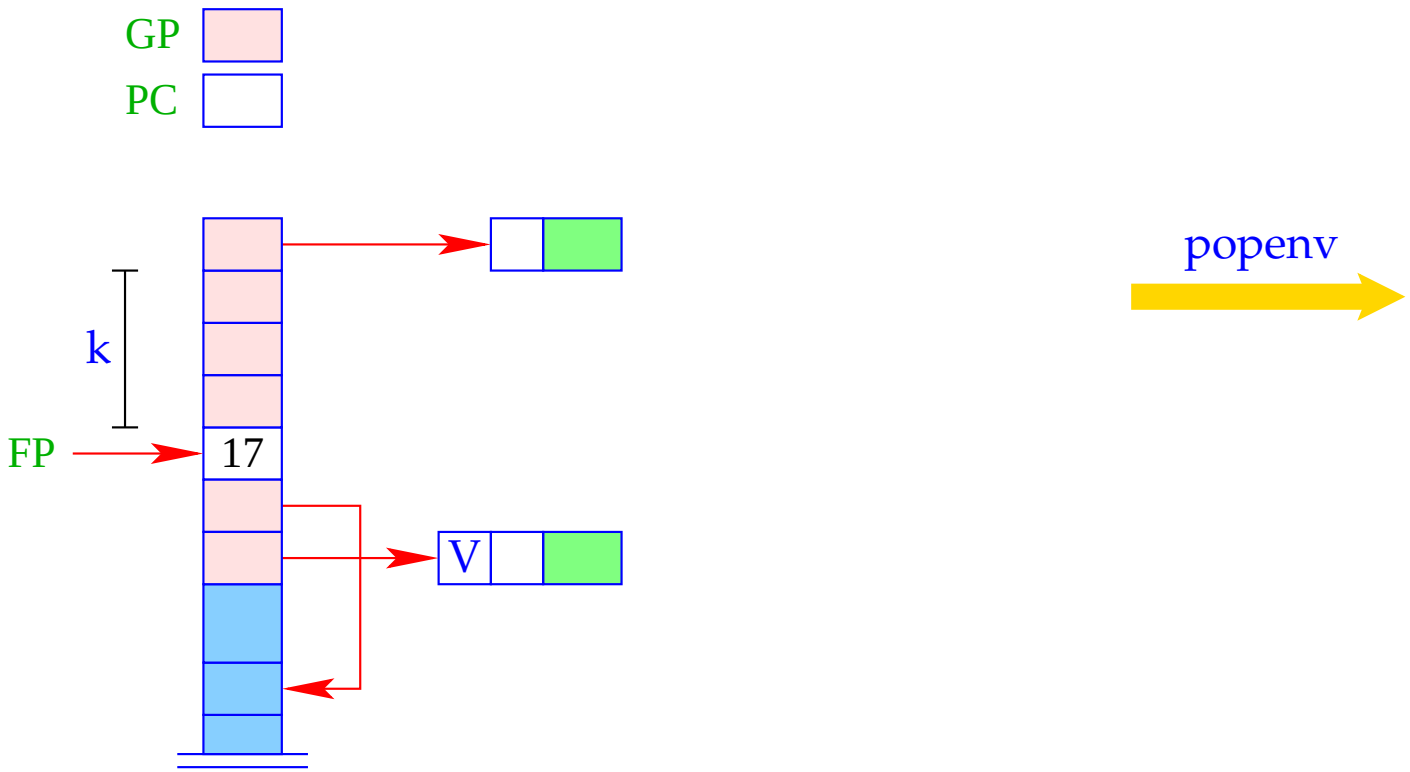

GP
PC
42
F
41
V
V
m
popenv
FP
17
V

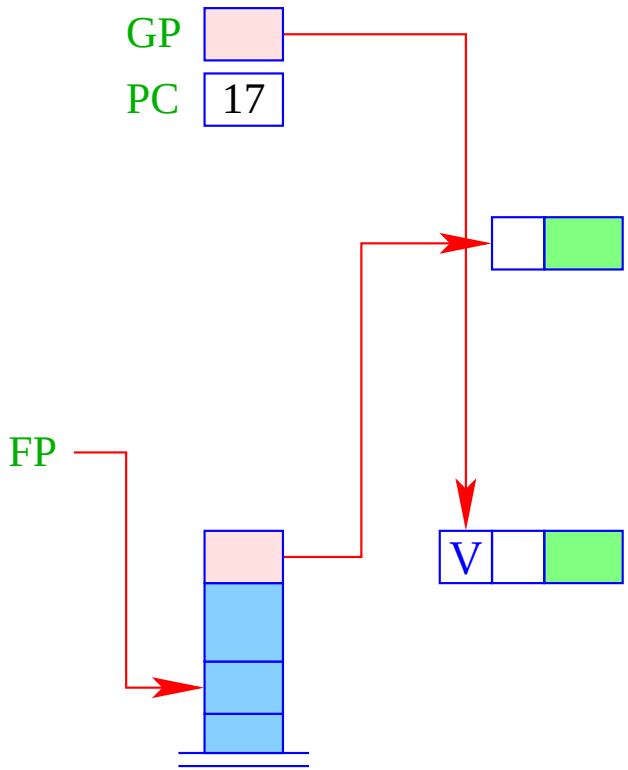

GP
PC
17
F
41
V
V
V
FP

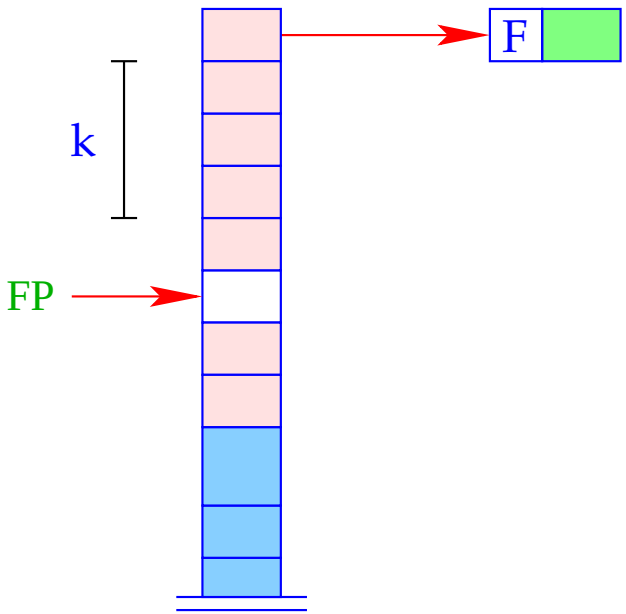
- Liegt exakt die richtige Argument-Anzahl vor, kann nach Abarbeitung des Rumpfs der Kellerrahmen aufgegeben werden.
- Liegt sogar Überversorgung mit Argumenten vor, muss der Rumpf sich offenbar erneut zu einer Funktion ausgewertet haben, die nun die restlichen Argumente konsumiert ...
- Für diese Überprüfung ist `return k` zuständig:

```
return k  =  if (SP − FP ≡ k + 1)
               popenv;              // Aufgeben des Kellerrahmens
             else {                 // Es gibt noch weitere Argumente
               slide k;
               apply;               // erneuter Aufruf
             }
```

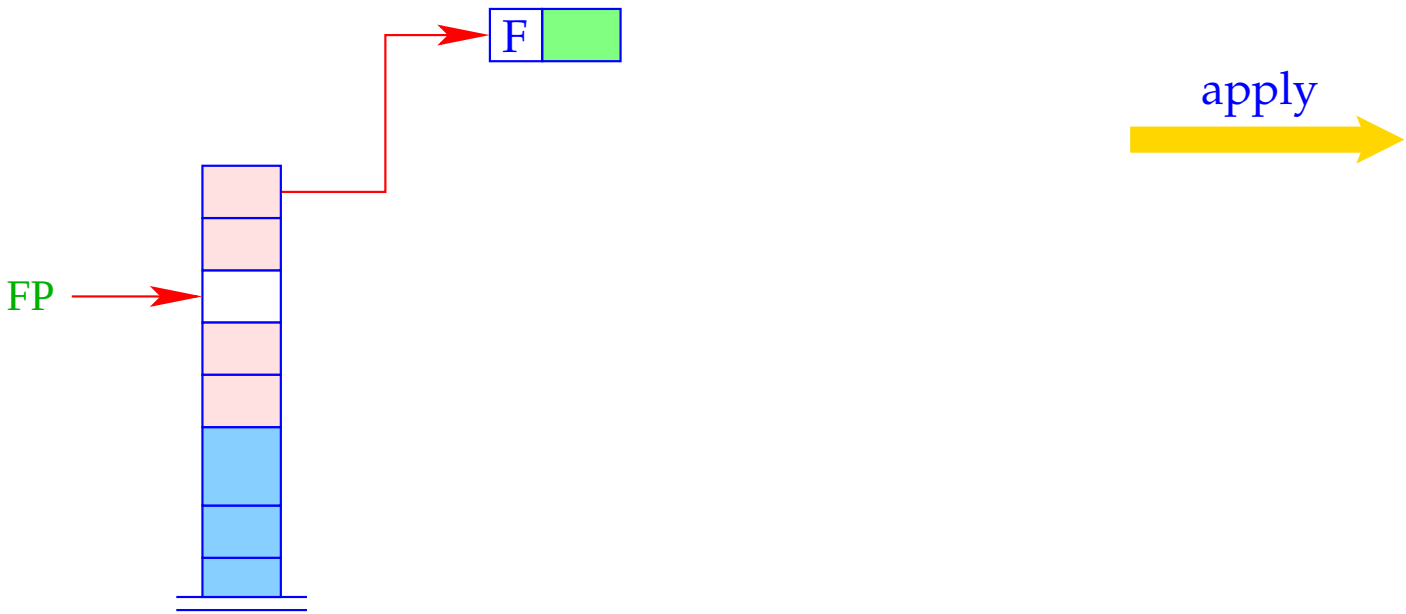
Damit erhalten wir etwa bei der Ausführung von `return k`:


GP
PC
k
FP
17
V
popenv


GP
PC
17
FP
V


F
k
FP


slide k


F
apply
FP

19 letrec-Ausdrücke

Sei $e \equiv \textbf{letrec}\ y_1 = e_1; \ldots; y_n = e_n\ \textbf{in}\ e_0$ ein **letrec**-Ausdruck. Die Übersetzung von e muss eine Befehlsfolge liefern, die

- lokale Variablen $y_1, \ldots, y_n$ anlegt;
- im Falle von
  CBV: $e_1, \ldots, e_n$ auswertet und die y_i an deren Werte bindet;
  CBN: Abschlüsse für $e_1, \ldots, e_n$ herstellt und die y_i daran bindet;
- den Ausdruck e_0 auswertet und schließlich dessen Wert zurück liefert.

Achtung!

In einem **letrec**-Ausdruck können wir bei der Definition der Werte bereits Variablen verwenden, die erst später angelegt werden! $\Longrightarrow$ Vor der eigentlichen Definition werden Dummy-Werte auf den Stack gelegt.

Für CBN erhalten wir:

$$
\begin{array}{lcll}
\text{code}_V\ e\ \rho\ \text{kp} & = & \text{alloc n} & // \text{ legt lok. Variablen an} \\
 & & \text{code}_C\ e_1\ \rho'\ (\text{kp}+n) & \\
 & & \text{rewrite n} & \\
 & & \ldots & \\
 & & \text{code}_C\ e_n\ \rho'\ (\text{kp}+n) & \\
 & & \text{rewrite 1} & \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{kp}+n) & \\
 & & \text{slide n} & // \text{ gibt lok. Variablen auf}
\end{array}
$$

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

Im Falle von CBV benutzen wir für die Ausdrücke $e_1, \ldots, e_n$ ebenfalls code_V.

Achtung:

Rekursive Definition von Basiswerten ist bei CBV undefiniert!!!

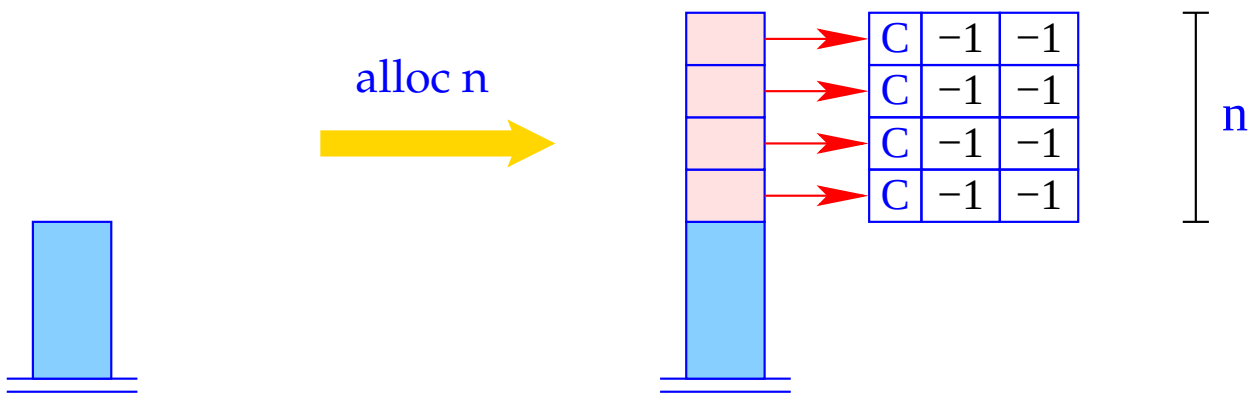
Beispiel:

Betrachte den Ausdruck

$$e \equiv \textbf{letrec } f = \textbf{fn } x, y \Rightarrow \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f(x * y)(y - 1) \textbf{ in } f1$$

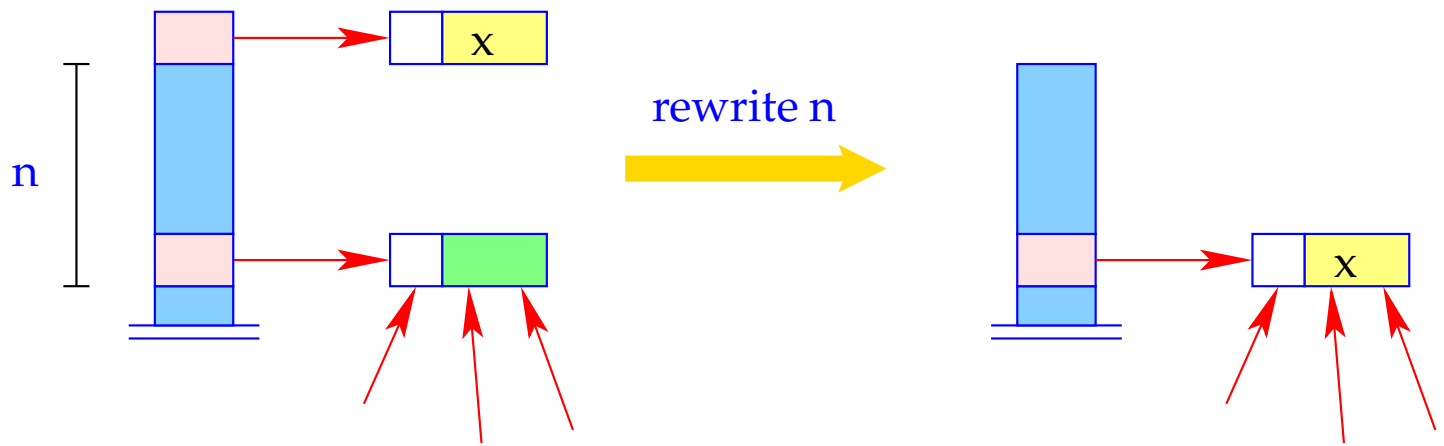
für $\rho = \emptyset$ und kp $= 0$. Dann ergibt sich (für CBV):

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




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

Die Instruktion alloc n reserviert n Zellen auf dem Keller und initialisiert diese mit n Dummy-Knoten.



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

- Die Referenz S[SP – n] bleibt erhalten!
- Was überschrieben wird, ist nur ihr Inhalt!