

Die Geheimnisse um `targ k` und `return k` lüften wir später.

17 Funktionsanwendungen

Funktions-Anwendungen entsprechen Funktions-Aufrufen in **C**. Notwendige Aktionen zur Auswertung von $e' e_0 \dots e_{m-1}$ sind:

- Anlegen eines Kellerrahmens;
- Parameter-Übergabe, d.h. bei:
 - CBV**: Auswerten der aktuellen Parameter;
 - CBN**: Anlegen von Abschlüssen für die aktuellen Parameter;
- Auswerten der Funktion e' zu einem F-Objekt;
- Anwenden der Funktion.

Folglich für **CBN**:

```

codeV (e' e0 ... em-1) ρ kp = mark A // Anlegen des Rahmens
codeC em-1 ρ (kp + 3)
codeC em-2 ρ (kp + 4)
...
codeC e0 ρ (kp + m + 2)
codeV e' ρ (kp + m + 3) // Auswerten der Funktion
apply // entspricht call
A : ...

```

Wenn wir **CBV** implementieren wollen, müssen die Argumente **vor** dem Funktions-Aufruf ausgewertet werden.

Dann benutzen wir **code_V** anstelle von **code_C** für die Argumente e_i :-)

Beispiel:

Für $(f\ 42)$, $\rho = \{f \mapsto (L, 2)\}$ und $kp = 2$ liefert das bei **CBV**:

2	mark A	6	mkbasic	7	apply
5	loadc 42	6	pushloc 4	3	A : ...

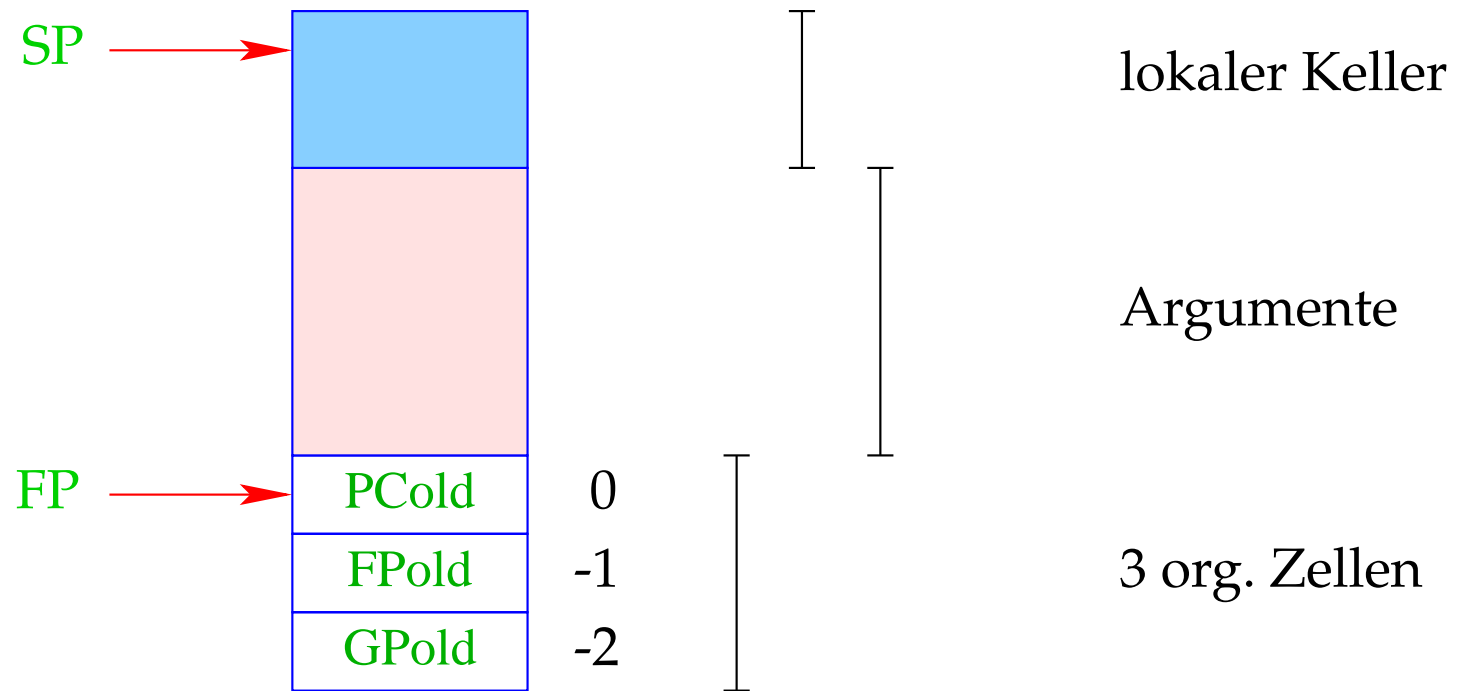
Ein etwas größeres Beispiel:

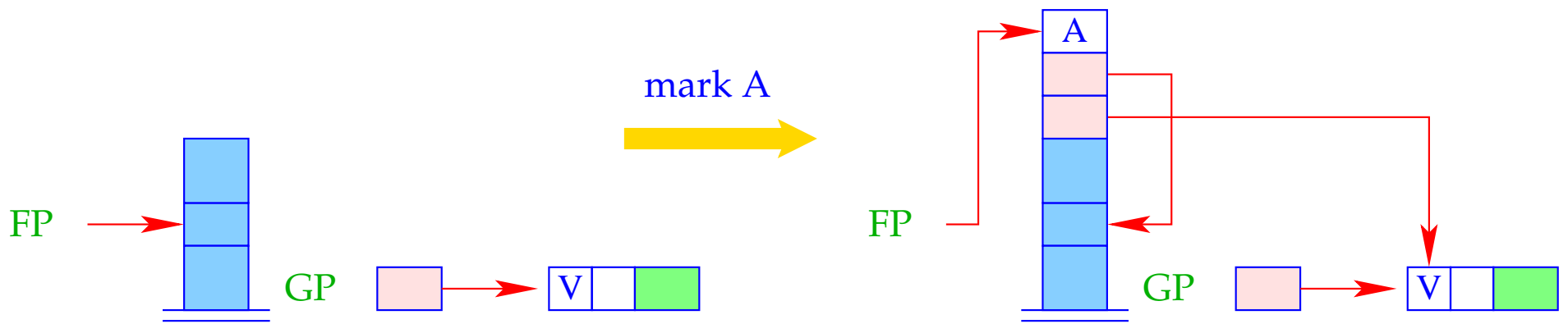
let $a = 17$; $f = \mathbf{fn}$ $b \Rightarrow a + b$ **in** f 42

Bei **CBV** erhalten wir für $kp = 0$:

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

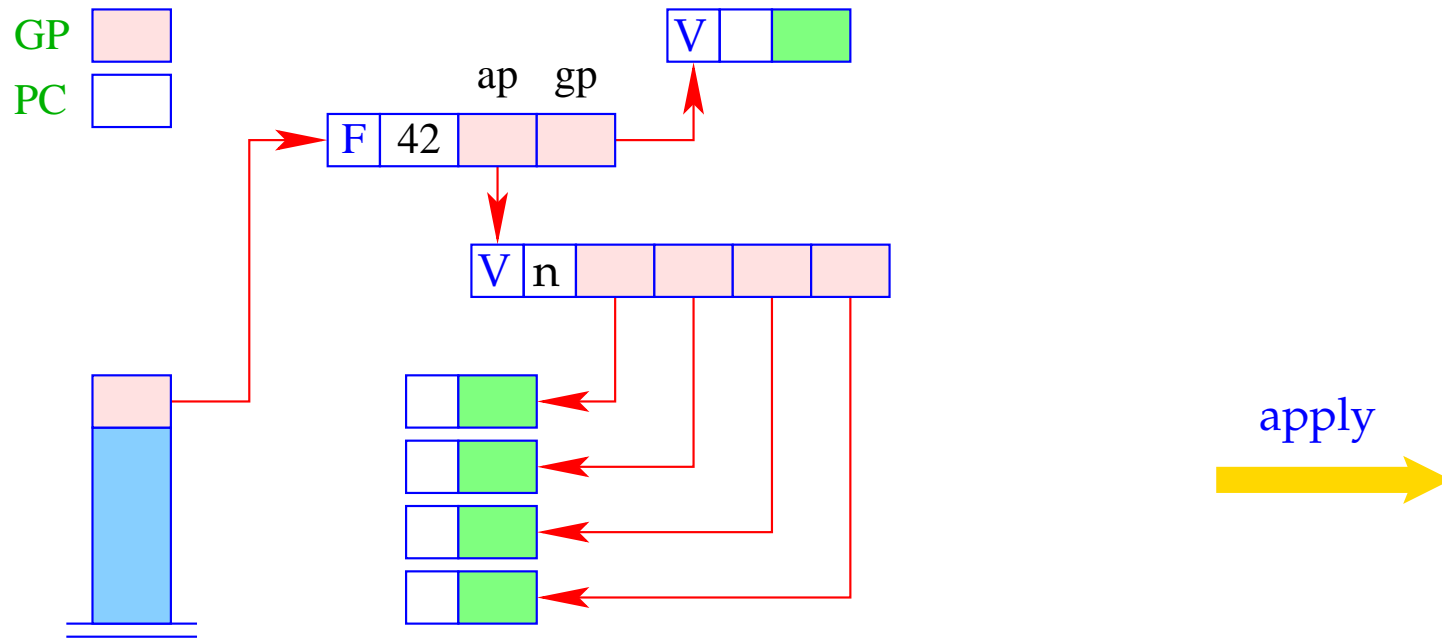
Vor der Implementierung der neuen Instruktionen müssen wir die Organisation eines Kellerrahmens festlegen:





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

- 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.



```

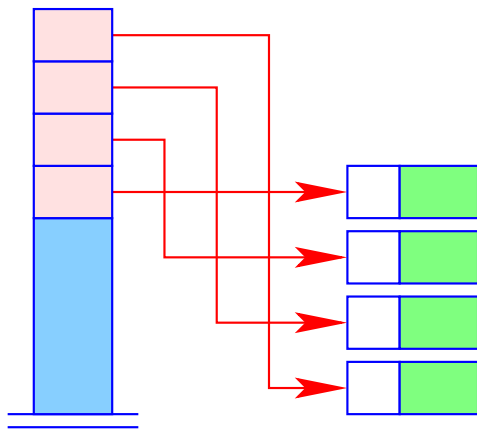
h = S[SP];
if (H[h] != (F,_,_))
    Error "no fun";
else {

```

```

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;
}

```

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`.

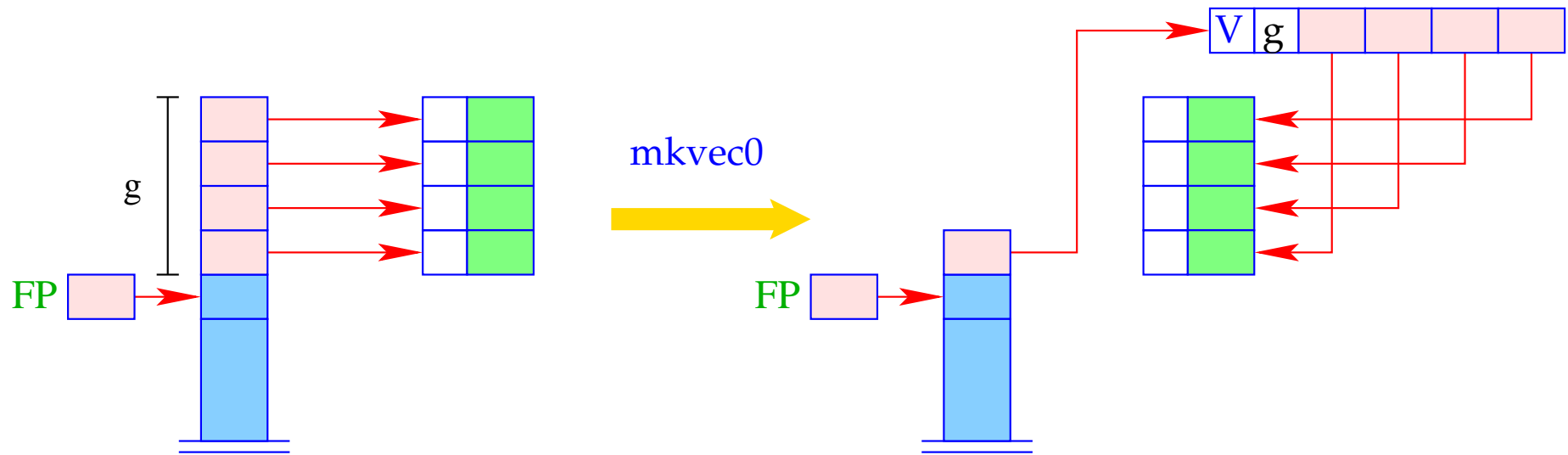
Er überprüft, ob bereits genügend Argumente vorliegen, um den Funktions-Rumpf auszuführen. Die Anzahl der Argumente ist: $SP - FP$.

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  
}
```

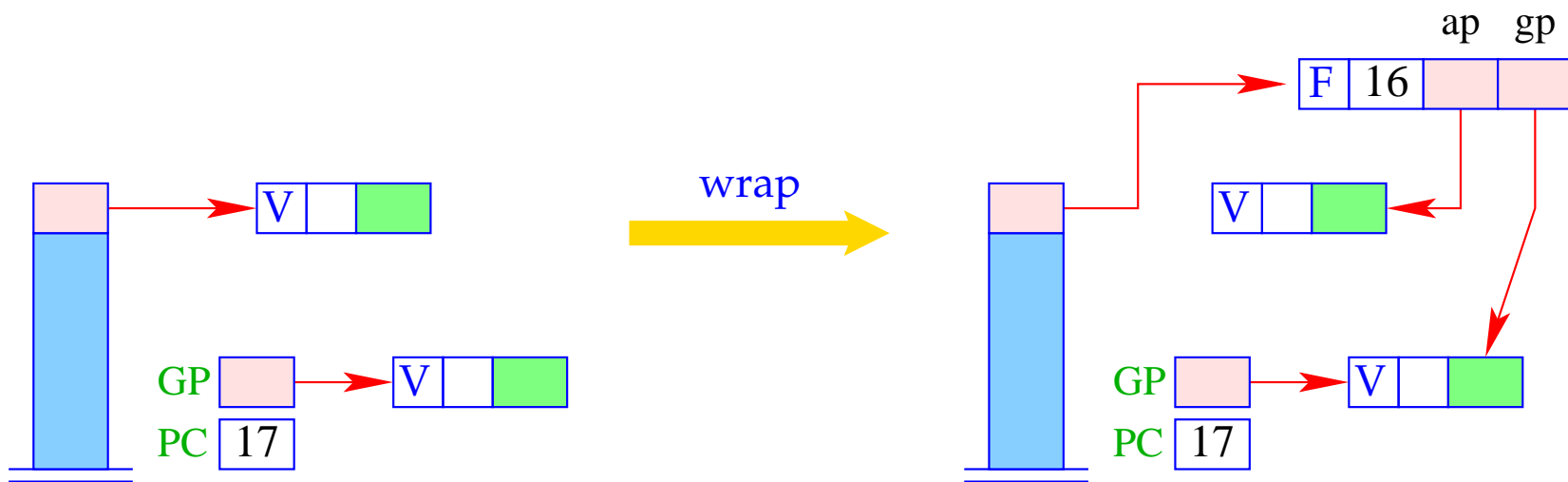
Die Zusammenfassung dieser festen Schritt-Abfolge zu einem Befehl kann als eine Art Optimierung verstanden werden :-)



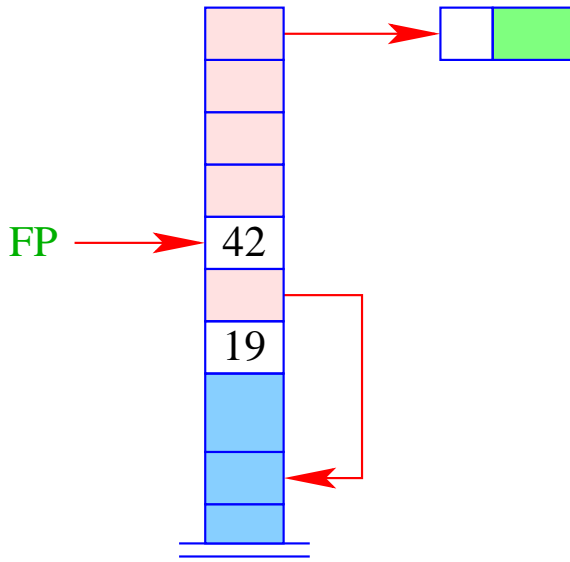
```

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;

```

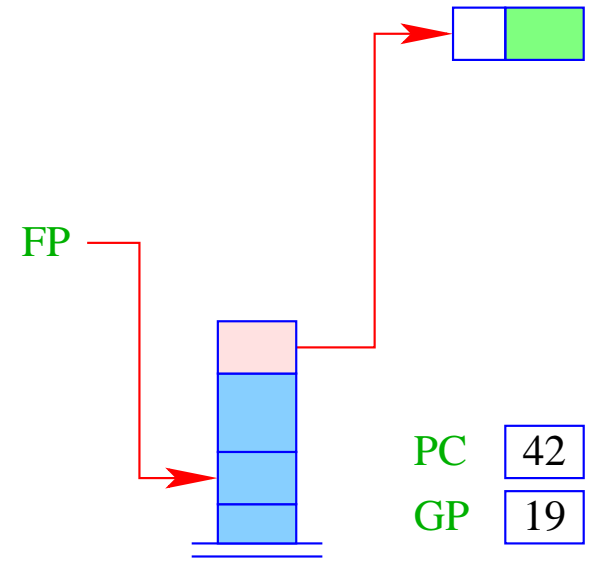


$S[SP] = \text{new}(F, PC-1, S[SP], GP);$

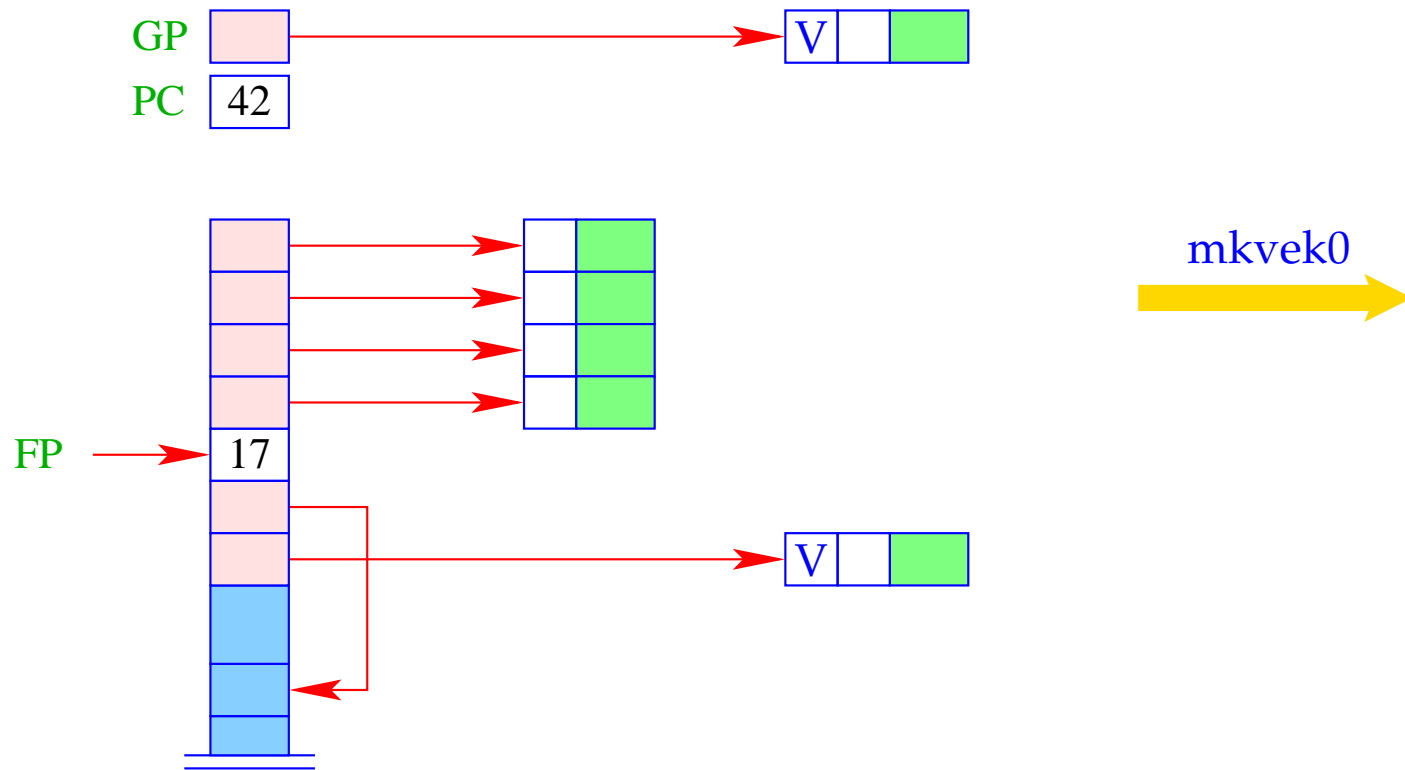


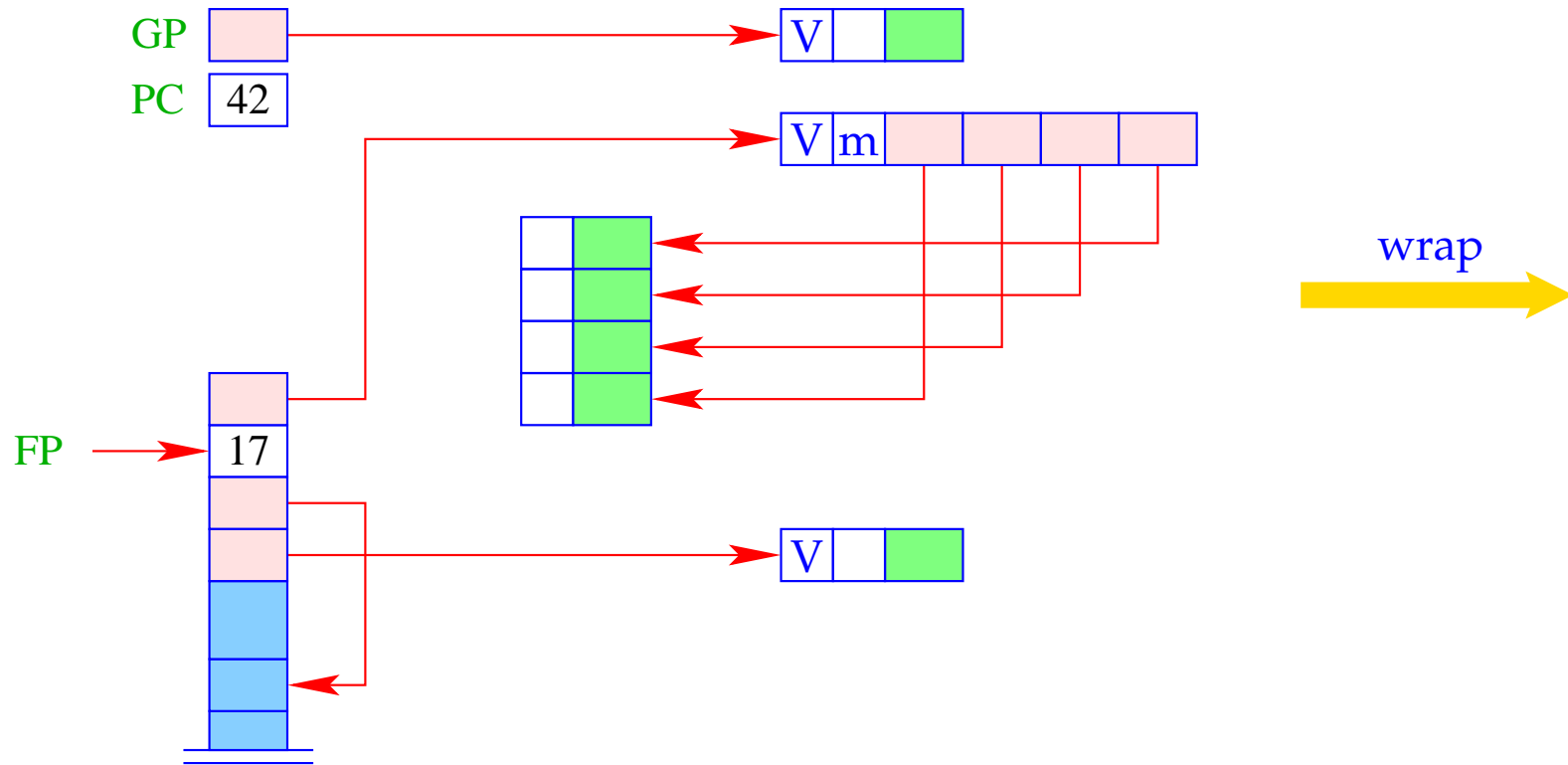
popenv

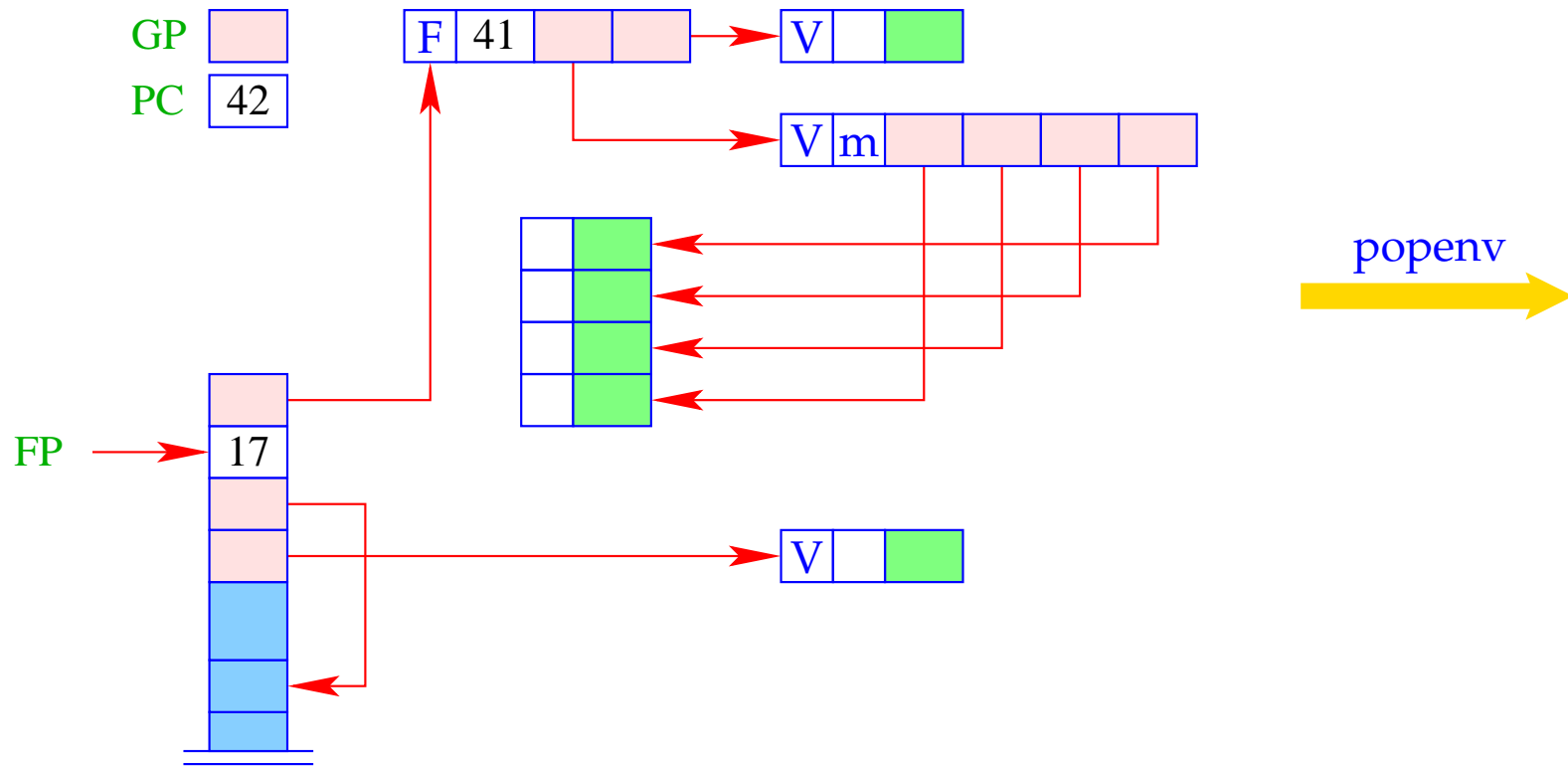
```
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];
```

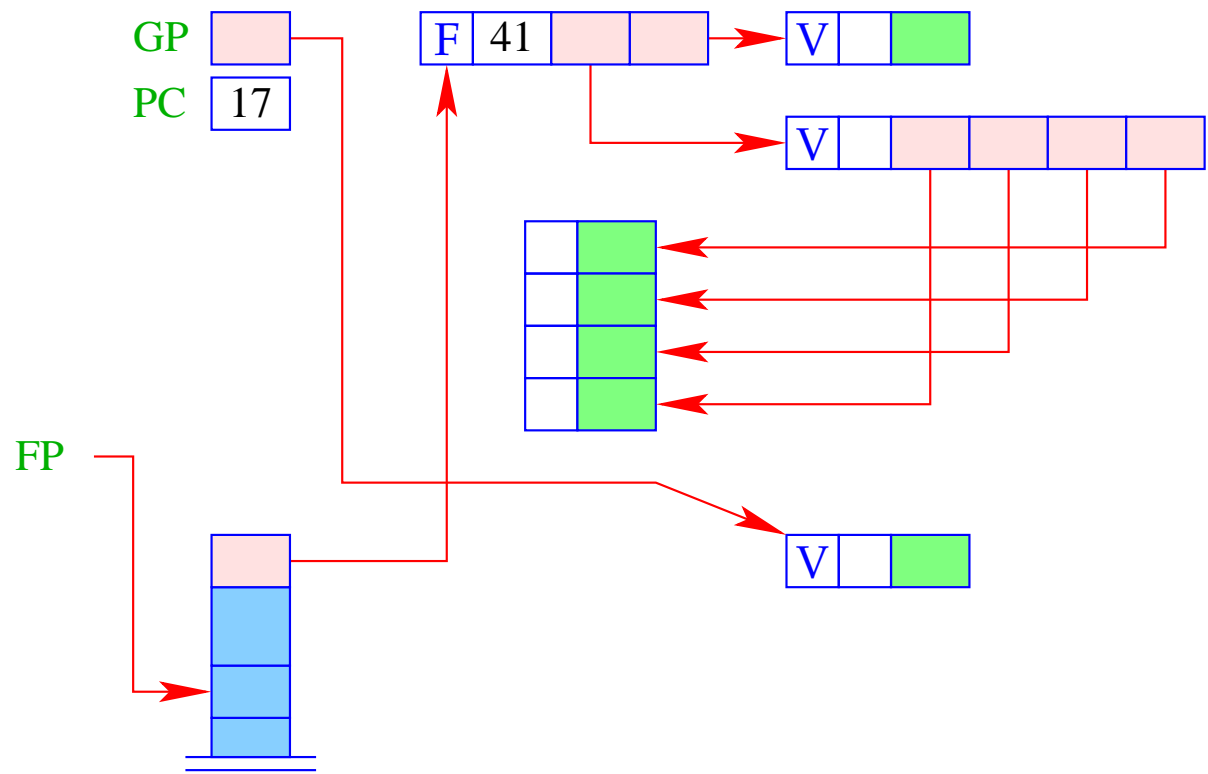


Insgesamt erhalten wir damit für `targ k`:









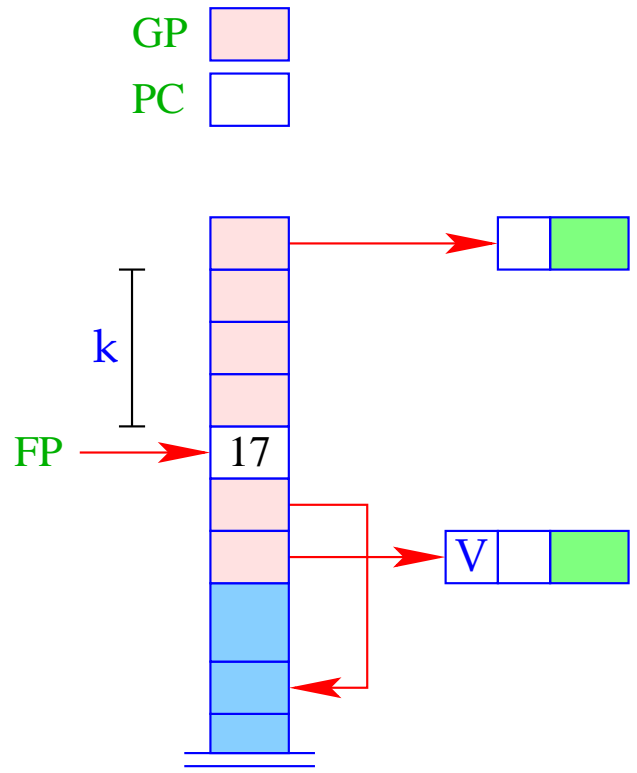
- Liegt exakt die richtige Argument-Anzahl vor, kann **nach Abarbeitung des Rumpfs** der Kellerrahmen aufgegeben werden.
- Liegt sogar **Übersorgung** mit Argumenten vor, muss der Rumpf sich offenbar erneut zu einer Funktion ausgewertet haben, die nun die restlichen Argumente konsumiert ...
- Für diese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

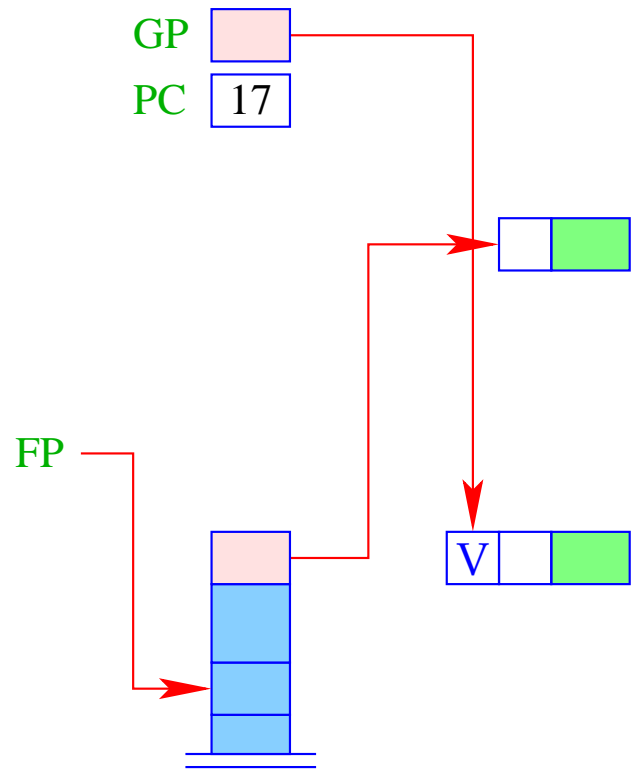
```

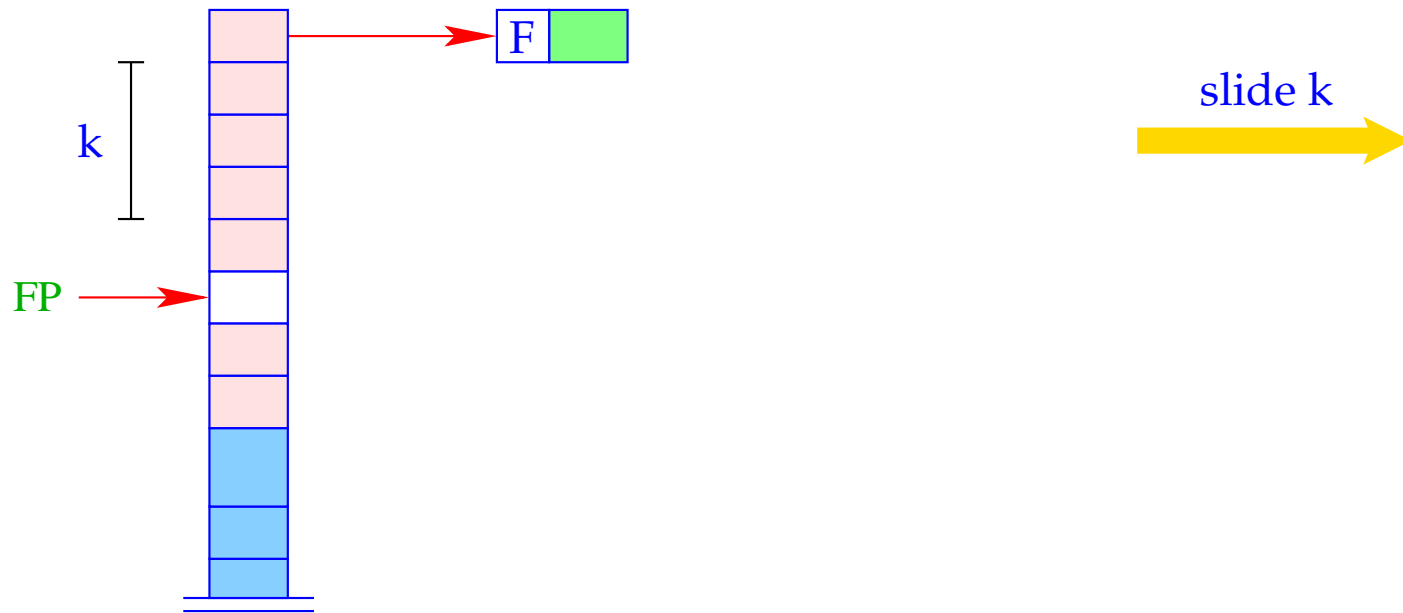
return k = if (SP - FP ≡ k + 1)
            popenv;           // Aufgeben des Kellerrahmens
        else {                // Es gibt noch weitere Argumente
            slide k;
            apply;           // erneuter Aufruf
        }

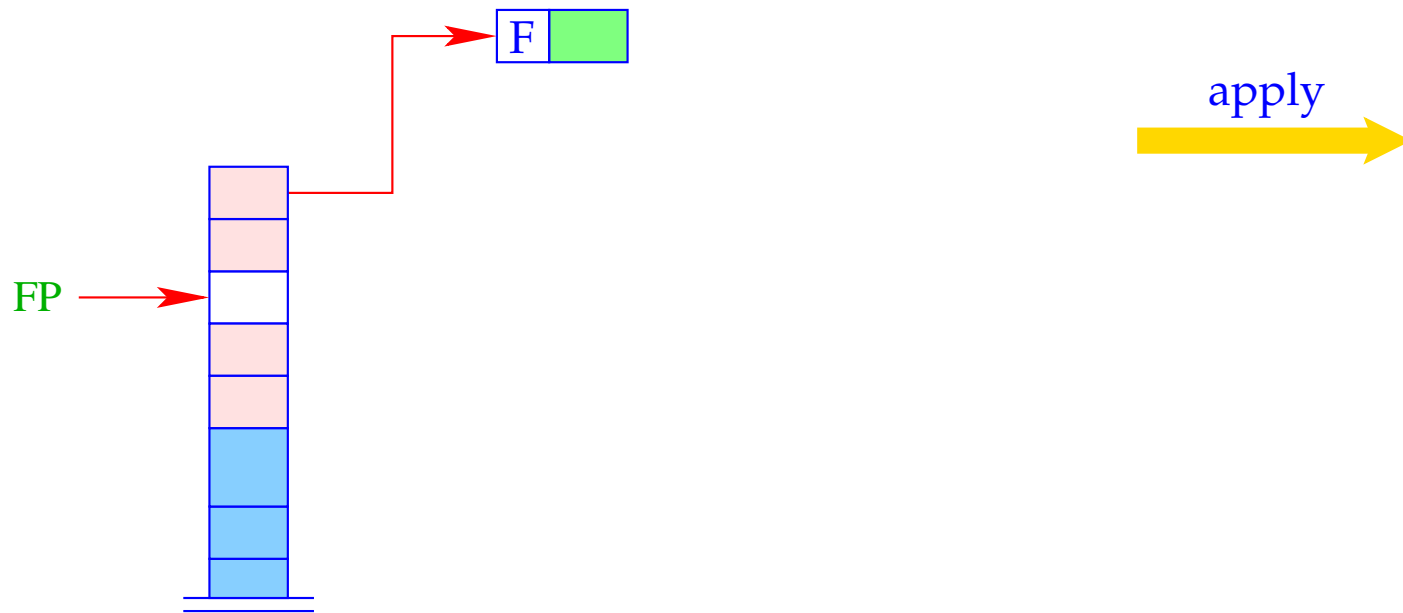
```

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









19 letrec-Ausdrücke

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

- lokale Variablen y_1, \dots, y_n anlegt;
- im Falle von
 - CBV**: e_1, \dots, e_n auswertet und die y_i an deren Werte bindet;
 - CBN**: Abschlüsse für e_1, \dots, 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! \implies Vor der eigentlichen Definition werden **Dummy**-Werte auf den Stack gelegt.

Für **CBN** erhalten wir:


```

codeV e ρ kp = alloc n // legt lok. Variablen an
                codeC e1 ρ' (kp + n)
                rewrite n
                ...
                codeC en ρ' (kp + n)
                rewrite 1
                codeV e0 ρ' (kp + n)
                slide n // gibt lok. Variablen auf

```

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

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

Achtung:

Rekursive Definition von Basiswerten ist bei **CBV undefiniert!!!**

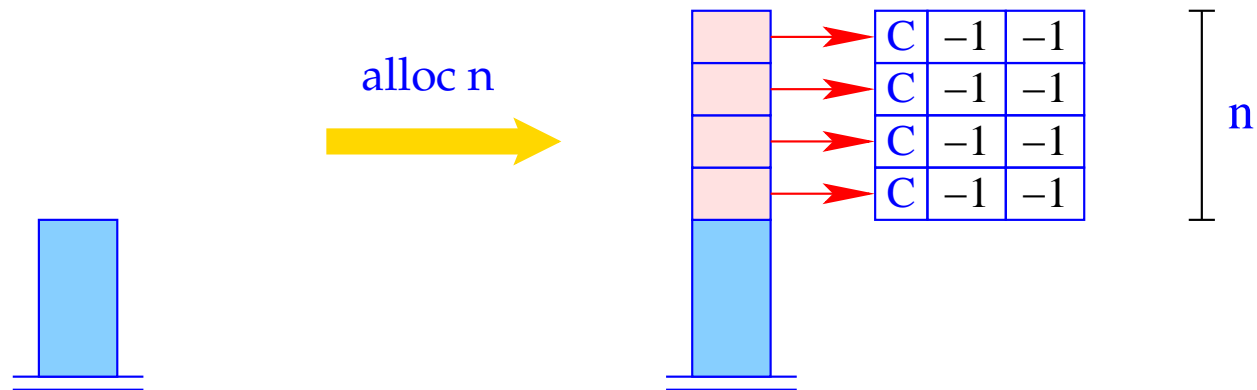
Beispiel:

Betrachte den Ausdruck

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

für $\rho = \emptyset$ und $\mathbf{kp} = 0$. 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

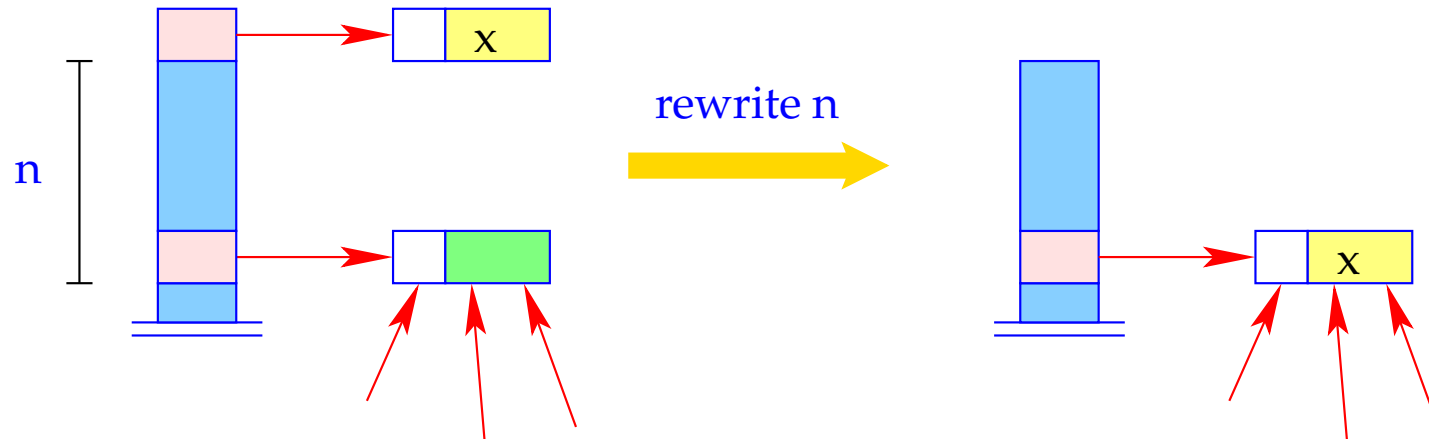


```

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**!