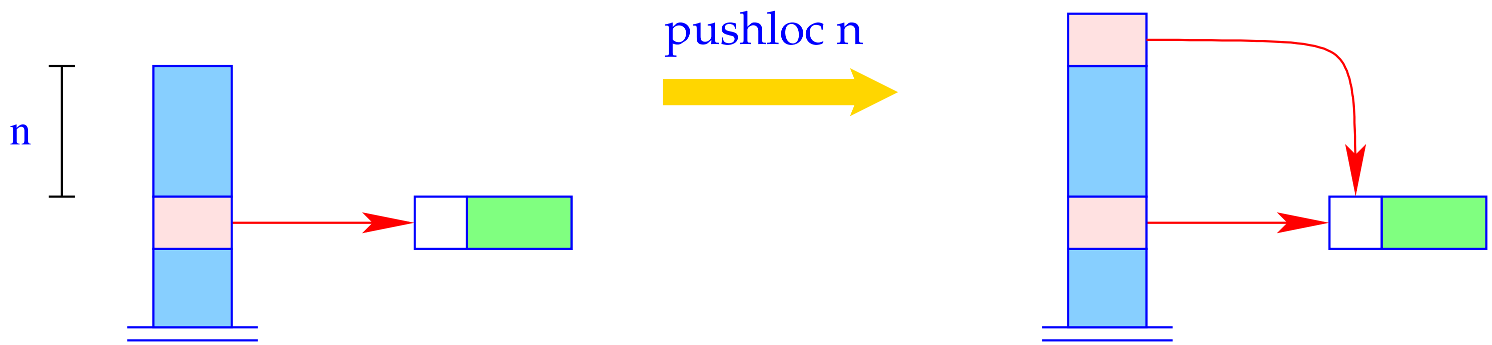


S[SP+1] =S[SP - n]; SP++;

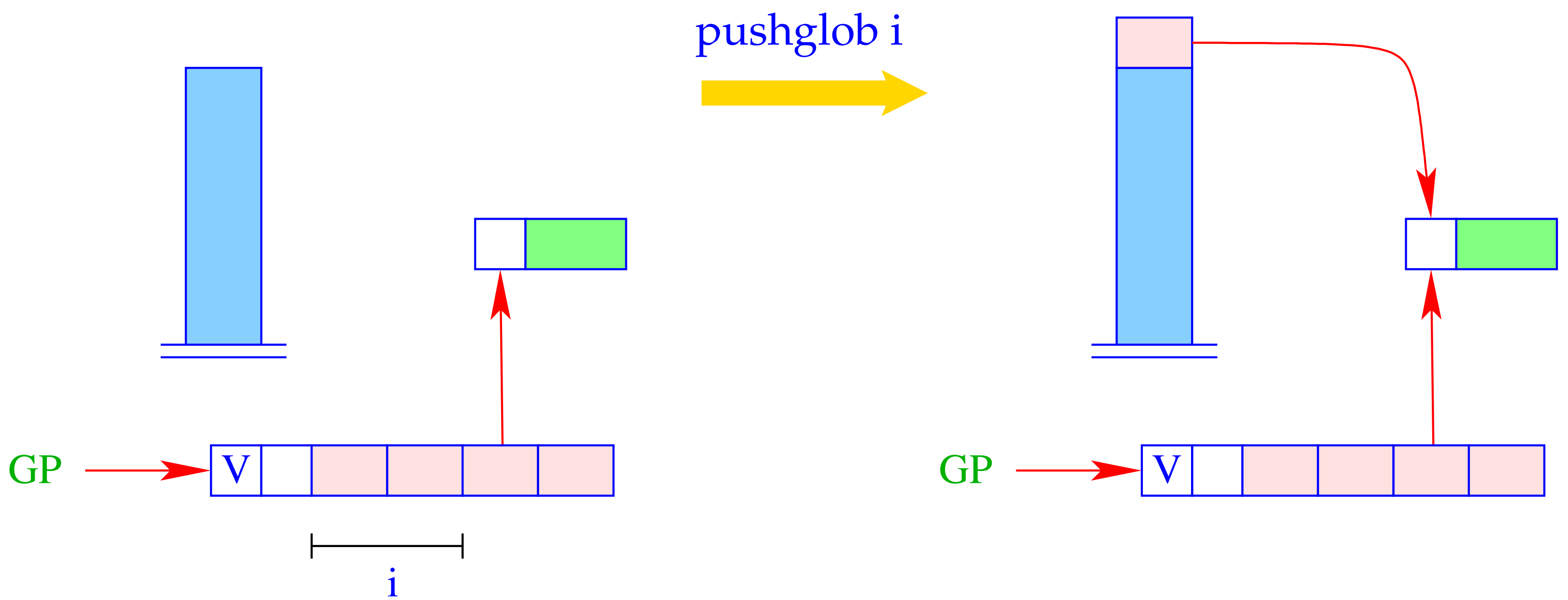
Zur Korrektheit:

Seien sp und kp die Werte des Stackpointers bzw. Kellerpegels vor der Ausführung der Instruktion. Dann wird der Wert $S[a]$ geladen für die Adresse

$$a = \text{sp} - (\text{kp} - i) = (\text{sp} - \text{kp}) + i = \text{sp}_0 + i$$

... wie es auch sein soll :-)

Der Zugriff auf die globalen Variablen ist da viel einfacher:



SP = SP + 1;
S[SP] = GP$\rightarrow$v[i];

Beispiel:

Betrachte $e \equiv (b + c)$ für $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ und $\mathrm{kp} = 1$.

Dann ist für CBN:

$$
\begin{array}{lclcll}
\mathrm{code}_V\; e\; \rho\; 1 & = & \mathrm{getvar}\; b\; \rho\; 1 & = & 1 & \mathrm{pushloc}\; 0 \\
 & & \mathrm{eval} & & 2 & \mathrm{eval} \\
 & & \mathrm{getbasic} & & 2 & \mathrm{getbasic} \\
 & & \mathrm{getvar}\; c\; \rho\; 2 & & 2 & \mathrm{pushglob}\; 0 \\
 & & \mathrm{eval} & & 3 & \mathrm{eval} \\
 & & \mathrm{getbasic} & & 3 & \mathrm{getbasic} \\
 & & \mathrm{add} & & 3 & \mathrm{add} \\
 & & \mathrm{mkbasic} & & 2 & \mathrm{mkbasic}
\end{array}
$$

15 let-Ausdrücke

Zum Aufwärmen betrachten wir zuerst die Behandlung lokaler Variablen :-)

Sei $e \equiv \textbf{let}\ y_1 = e_1\ \textbf{in let} \ldots \textbf{in let}\ y_k = e_n\ \textbf{in}\ e_0$ ein (geschachtelter) **let**-Ausdruck. Die Übersetzung von e muss eine Befehlsfolge liefern, die

- die lokalen Variablen $y_1, \ldots, y_n$ auf dem Stack anlegt;
- im Falle von
  CBV: die Ausdrücke e_j auswertet und y_j an die Werte bindet;
  CBN: Abschlüsse für die e_j herstellt und y_j daran bindet;
- den Ausdruck e_0 auswertet und schließlich dessen Wert zurück liefert.

Dann erhalten wir für CBN:

$$
\begin{array}{rcl}
\mathrm{code}_V\ e\ \rho_0\ \mathrm{kp} & = & \mathrm{code}_C\ e_1\ \rho_0\ \mathrm{kp} \\
& & \mathrm{code}_C\ e_2\ \rho_1\ (\mathrm{kp}+1) \\
& & \ldots \\
& & \mathrm{code}_C\ e_n\ \rho_{n-1}\ (\mathrm{kp}+n-1) \\
& & \mathrm{code}_V\ e_0\ \rho_1\ (\mathrm{kp}+n) \\
& & \mathrm{slide\ n} \qquad\qquad\qquad\qquad // \text{ gibt lok. Variable auf}
\end{array}
$$

wobei $\quad \rho_j = \rho_{j-1} \oplus \{y_1 \mapsto (L, \mathrm{kp}+j)\} \quad$ für $j = 1, \ldots, n$.

Im Falle von CBV müssen die Werte der Variablen y_j sofort ermittelt werden!

Dann benutzen wir für die Ausdaṙucke e_j ebenfalls code_V.

Achtung!

Die e_i müssen mit den gleichen Bindungen für die (nicht verdeckten) globalen Variablen versehen werden!

Beispiel:

Betrachte den Ausdruck

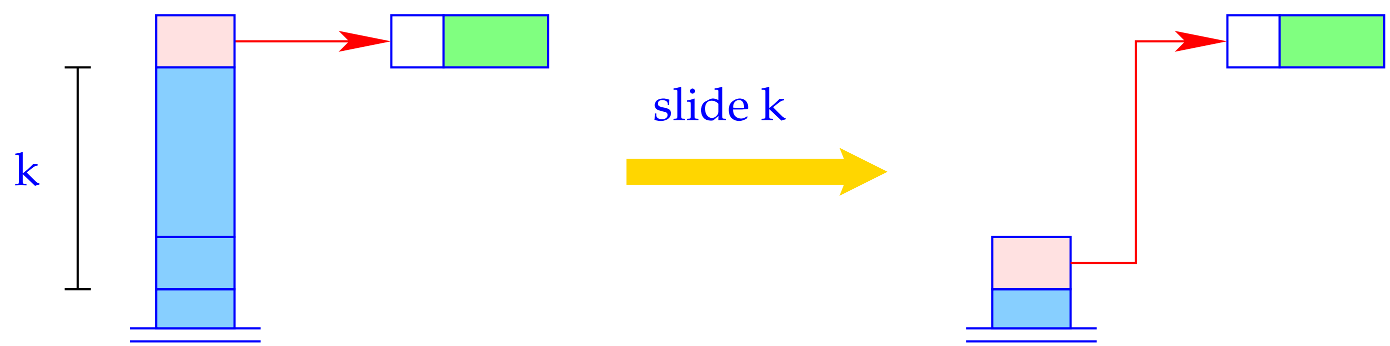
$$e \equiv \textbf{let } a = 19 \textbf{ in let } b = a * a \textbf{ in } a + b$$

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

0	loadc 19	3	getbasic	3	pushloc 1
1	mkbasic	3	mul	4	getbasic
1	pushloc 0	2	mkbasic	4	add
2	getbasic	2	pushloc 1	3	mkbasic
2	pushloc 1	3	getbasic	3	slide 2

Der Befehl slide k ist eine Abkürzung für slide 1 ... slide 1 (*k* mal).

Er gibt den Platz von k lokalen Variablen wieder auf:



S[SP-k] = S[SP];
SP = SP - k;

16 Funktions-Definitionen

Für eine Funktion f müssen wir Code erzeugen, die einen funktionalen Wert für f in der Halde anlegt. Das erfordert:

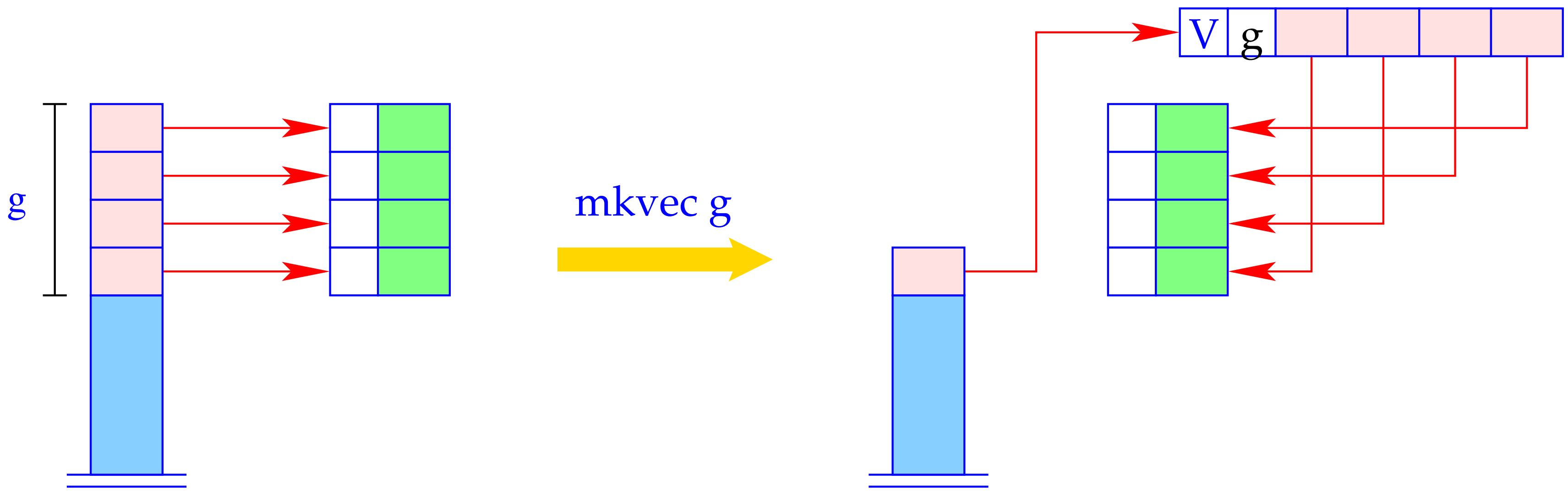
- Erzeugen des Global Vector mit den Bindungen der freien Variablen;
- Erzeugen eines (anfänglich leeren) Argument-Vektors;
- Erzeugen eines F-Objekts, das zusätzlich die Anfangs-Adresse des Codes zur Auswertung des Rumpfs enthält;
- Code zur Auswertung des Rumpfs.

Folglich:

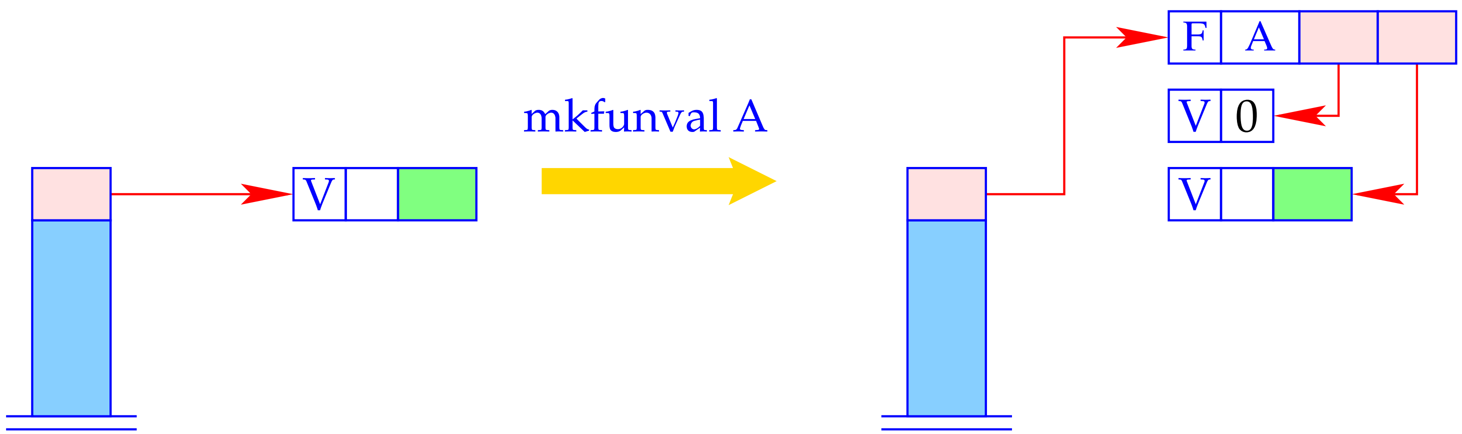
$$
\begin{array}{rcrl}
\mathrm{code}_V\ (\textbf{fun}\ x_0,\ldots,x_{k-1} \rightarrow e)\ \rho\ \mathrm{kp} & = & & \mathrm{getvar}\ z_0\ \rho\ \mathrm{kp} \\
& & & \mathrm{getvar}\ z_1\ \rho\ (\mathrm{kp}+1) \\
& & & \ldots \\
& & & \mathrm{getvar}\ z_{g-1}\ \rho\ (\mathrm{kp}+g-1) \\
& & & \mathrm{mkvec}\ \mathrm{g} \\
& & & \mathrm{mkfunval}\ \mathrm{A} \\
& & & \mathrm{jump}\ \mathrm{B} \\
& & \mathrm{A}: & \mathrm{targ}\ \mathrm{k} \\
& & & \mathrm{code}_V\ e\ \rho'\ 0 \\
& & & \mathrm{return}\ \mathrm{k} \\
& & \mathrm{B}: & \ldots
\end{array}
$$

wobei $\{z_0,\ldots,z_{g-1}\} = \mathit{free}(\textbf{fun}\ x_0,\ldots,x_{k-1} \rightarrow e)$

und $\rho' = \{x_i \mapsto (L,-i) \mid i = 0,\ldots,k-1\} \cup \{z_j \mapsto (G,j) \mid j = 0,\ldots,g-1\}$



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



a = new (V,0);
S[SP] = new (F, A, a, S[SP]);

Beispiel:

Betrachte $f \equiv \textbf{fun}\; b \rightarrow a + b$ für $\rho = \{a \mapsto (L, 1)\}$ und $\text{kp} = 1$.

Dann liefert $\text{code}_V\; f\; \rho\; 1$:

1		pushloc 0	0	pushglob 0	2		getbasic
2		mkvec 1	1	eval	2		add
2		mkfunval A	1	getbasic	1		mkbasic
2		jump B	1	pushloc 1	1		return 1
0	A :	targ 1	2	eval	2	B :	...

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 $e' \; e_0 \; \ldots \; e_{m-1}$ 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:

$$
\begin{array}{rcll}
\text{code}_V\,(e'\;e_0\;\ldots\;e_{m-1})\;\rho\;\text{kp} & = & \text{mark A} & //\;\text{Anlegen des Rahmens}\\
& & \text{code}_C\;e_{m-1}\;\rho\;(\text{kp}+3) & \\
& & \text{code}_C\;e_{m-2}\;\rho\;(\text{kp}+4) & \\
& & \ldots & \\
& & \text{code}_C\;e_0\;\rho\;(\text{kp}+m+2) & \\
& & \text{code}_V\;e'\;\rho\;(\text{kp}+m+3) & //\;\text{Auswerten der Funktion}\\
& & \text{apply} & //\;\text{entspricht call}\\
& A: & \ldots &
\end{array}
$$

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 :	...

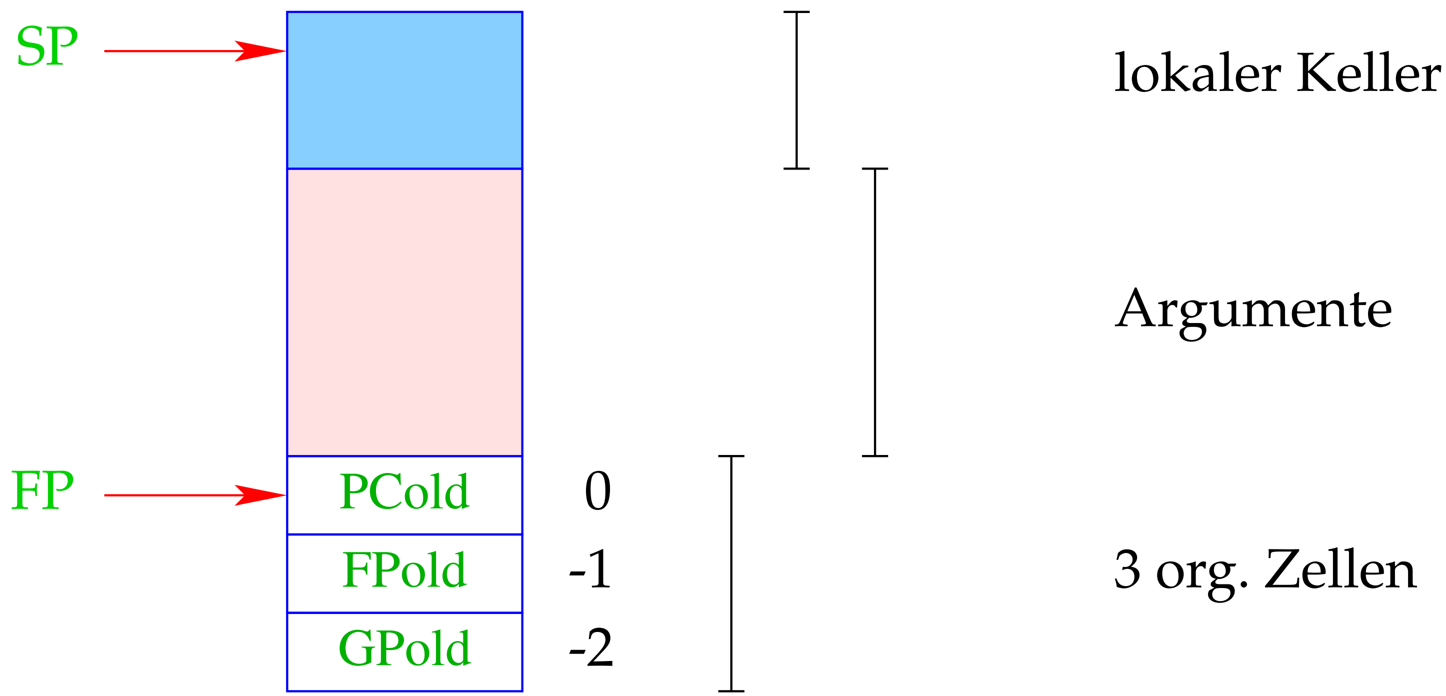
Ein etwas größeres Beispiel:

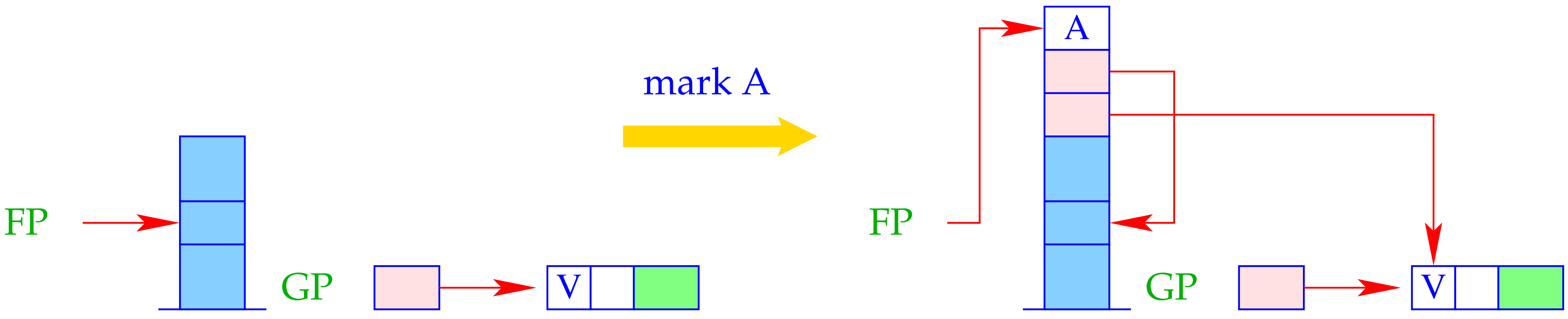
$$\textbf{let } a = 17 \textbf{ in let } f = \textbf{fun } b \rightarrow a + b \textbf{ in } f\ 42$$

Bei CBV erhalten wir für $\text{kp} = 0$:

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

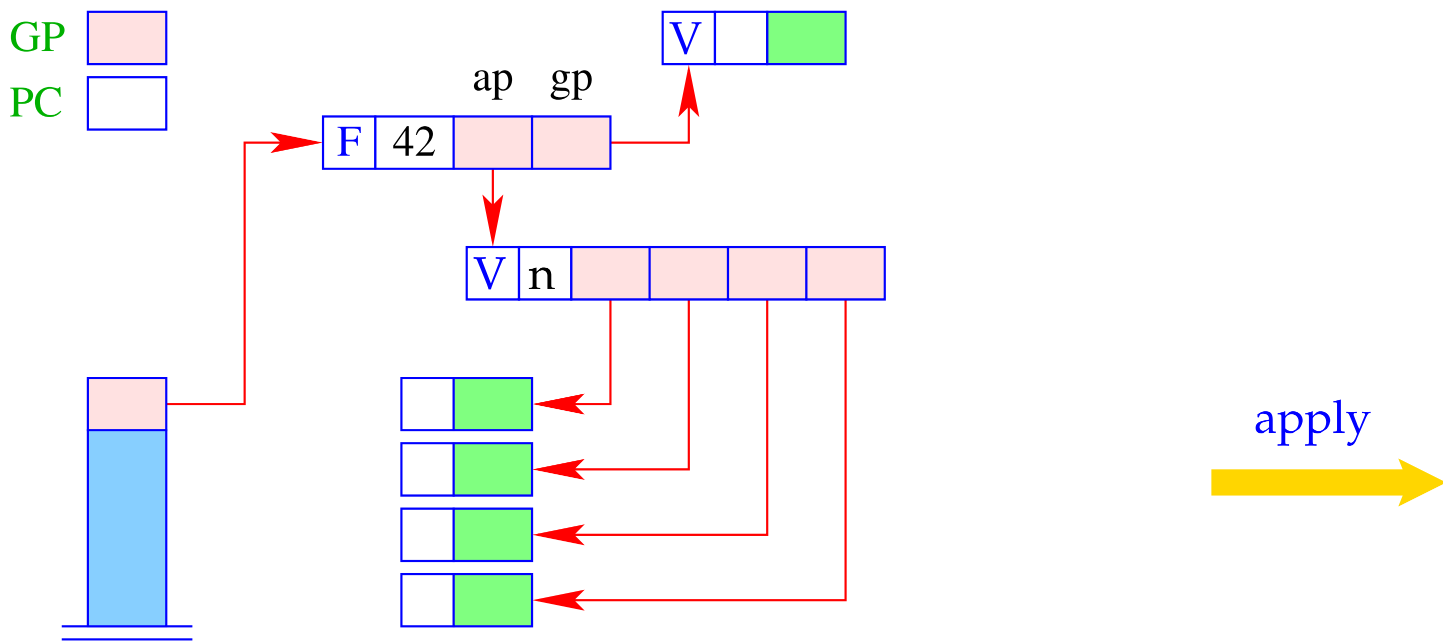
Vor der Implementierung der neuen Instruktionen müssen wir die Organisation eines Kellerrahmens festlegen:





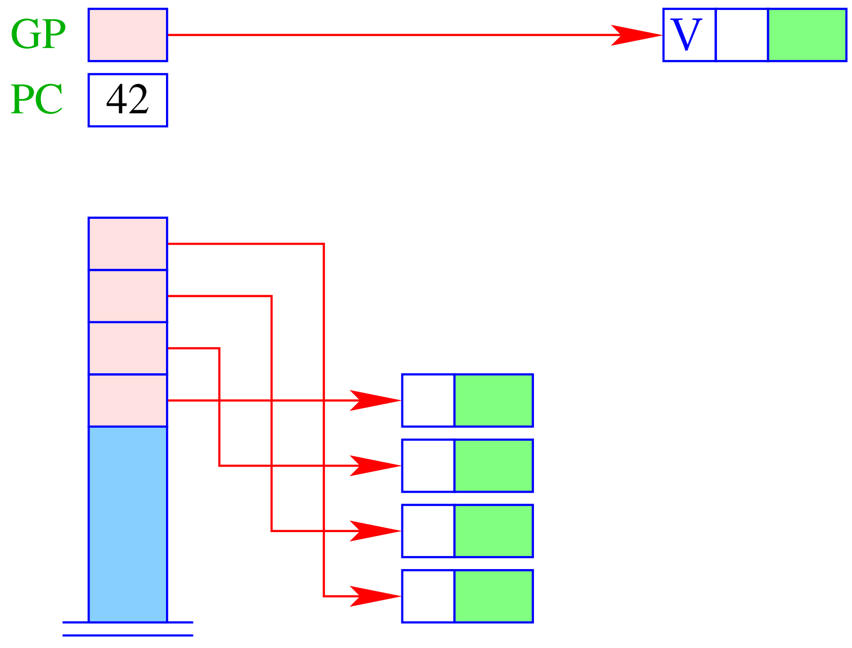
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = A;
FP = SP = SP + 3;

- 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;
}


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.

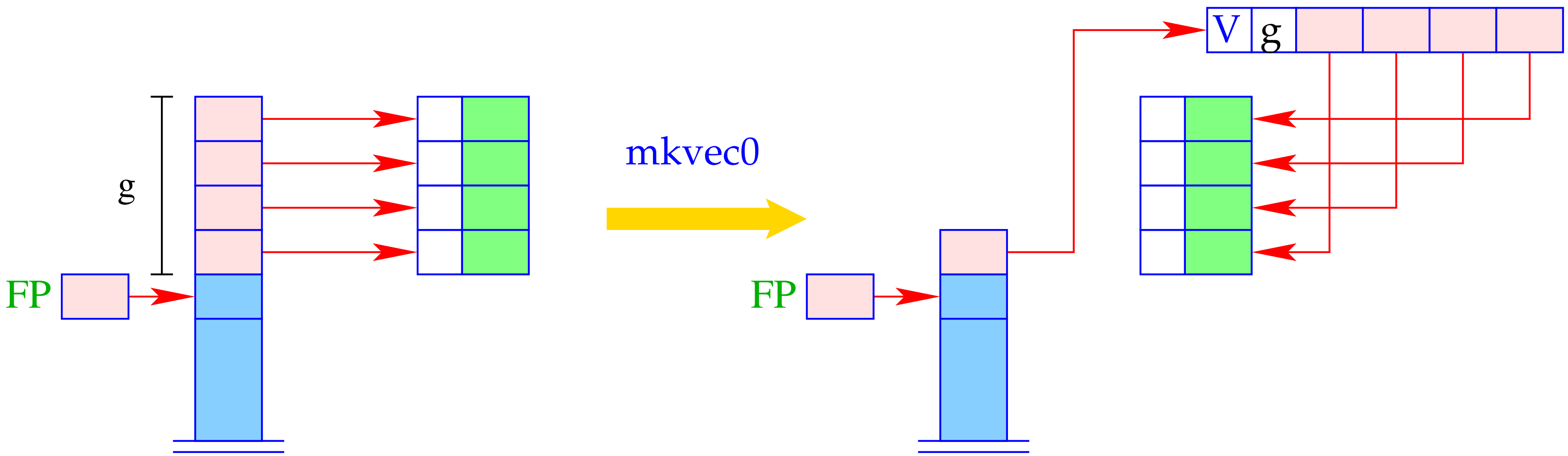
Er überprüft, ob bereits genügend Argumente vorliegen, um den Funktions-Rumpf auszuführen. Die Anzahl der Argumente ist: $SP - FP$.

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

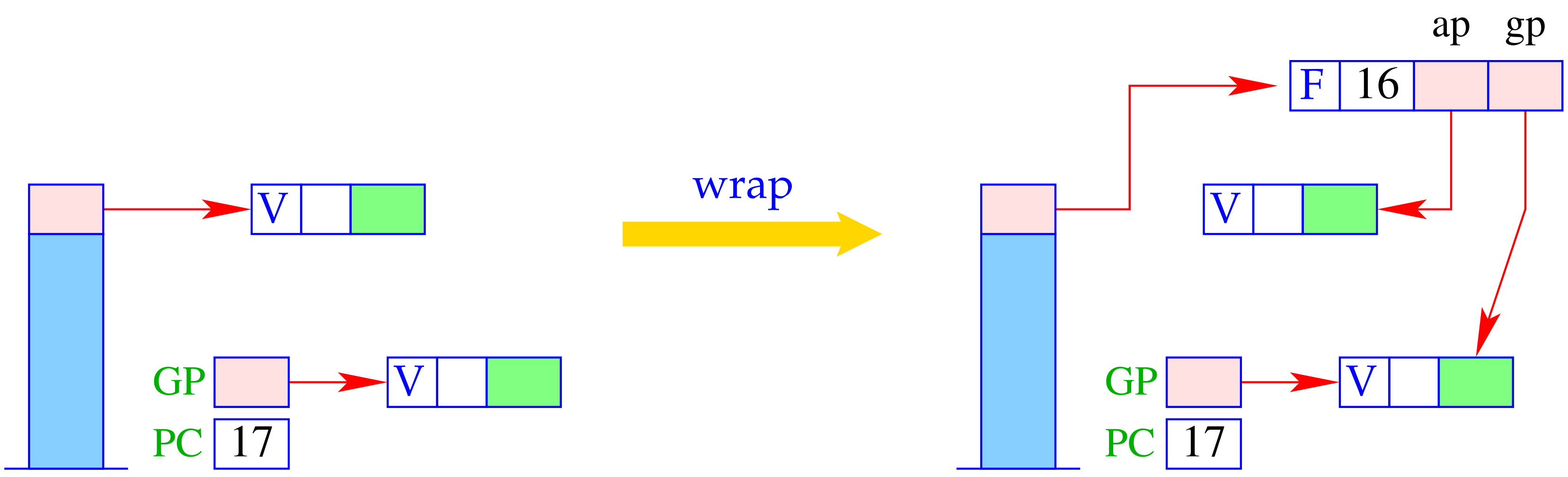
targ k   ist ein komplizierter Befehl. Darum zerlegen wir seine Ausführung in mehrere Schritte:

$$
\begin{array}{lll}
\text{targ } k & = & \text{if } (SP - FP < k)\ \{ \\
& & \quad \text{mkvec0}; \qquad\qquad // \text{ Anlegen des Argument} - \text{Vektors} \\
& & \quad \text{wrap}; \qquad\qquad\quad // \text{ Anlegen des F} - \text{Objekts} \\
& & \quad \text{popenv}; \qquad\qquad // \text{ Aufgeben des Kellerrahmens} \\
& & \}
\end{array}
$$

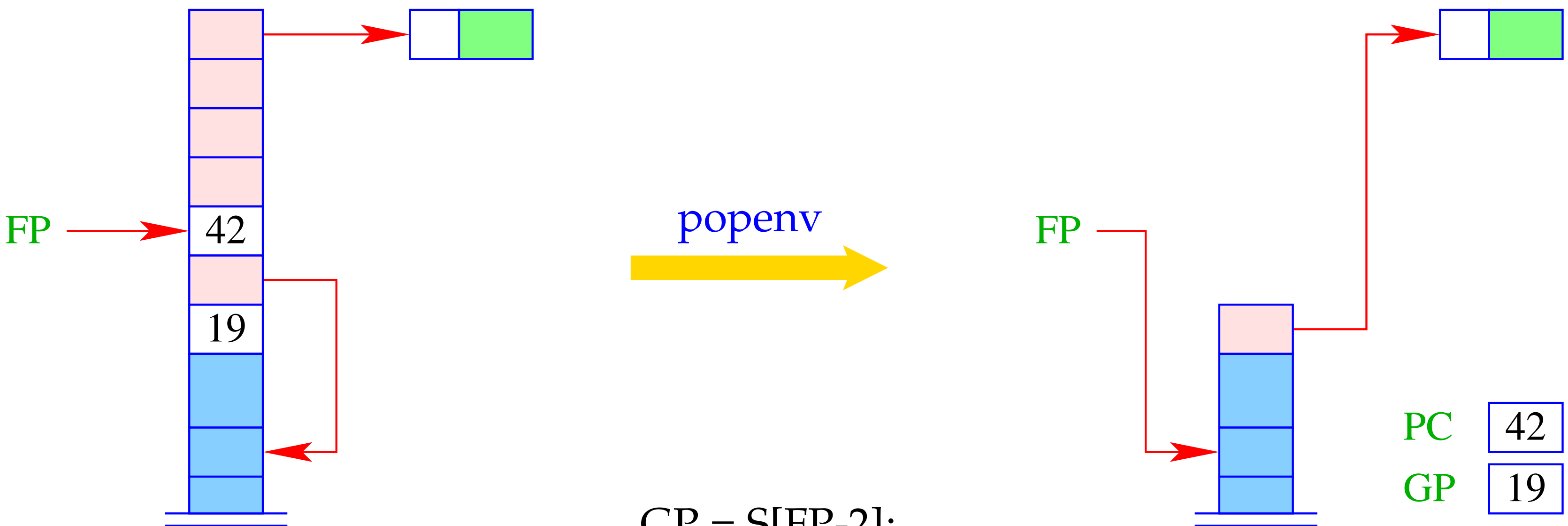
Die Zusammenfassung dieser festen Schritt-Abfolge zu einem Befehl kann als eine Art Optimierung verstanden werden   :-)



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;

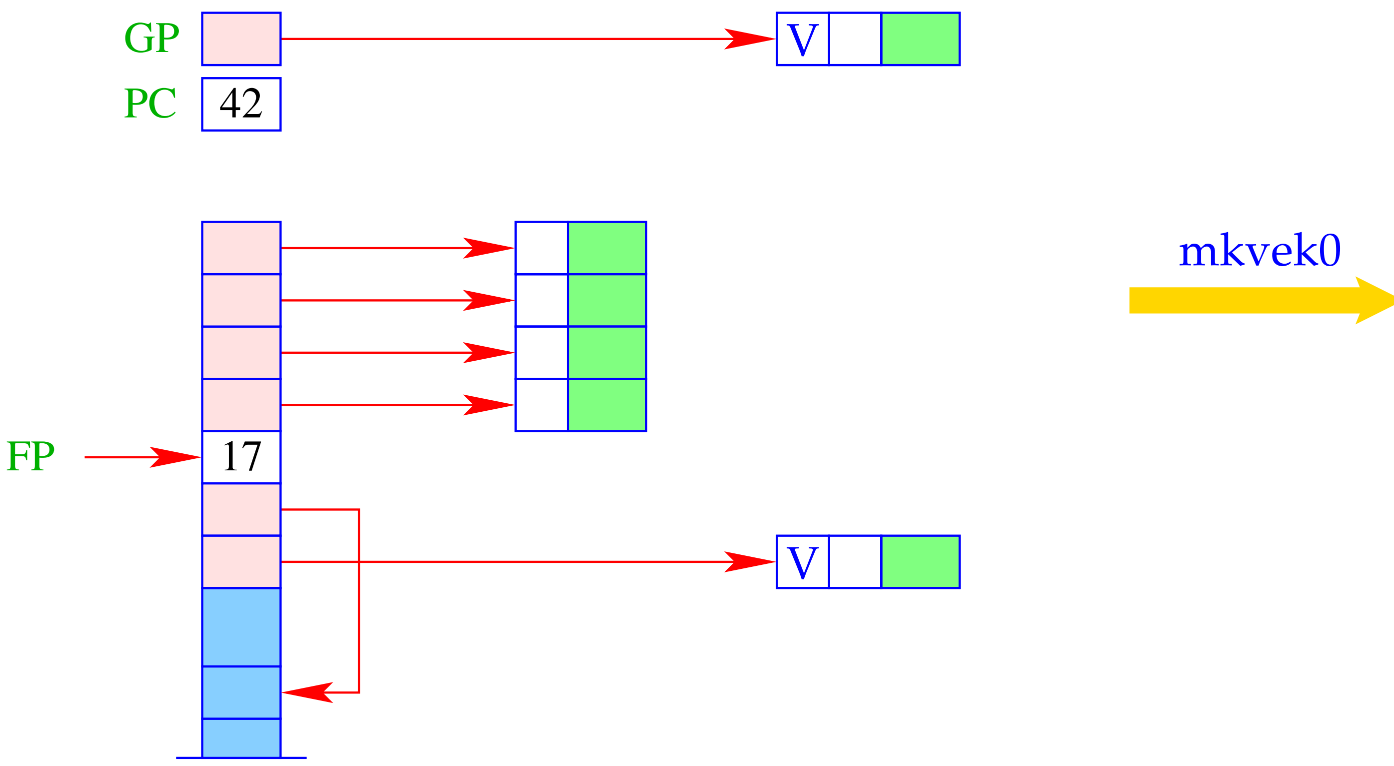


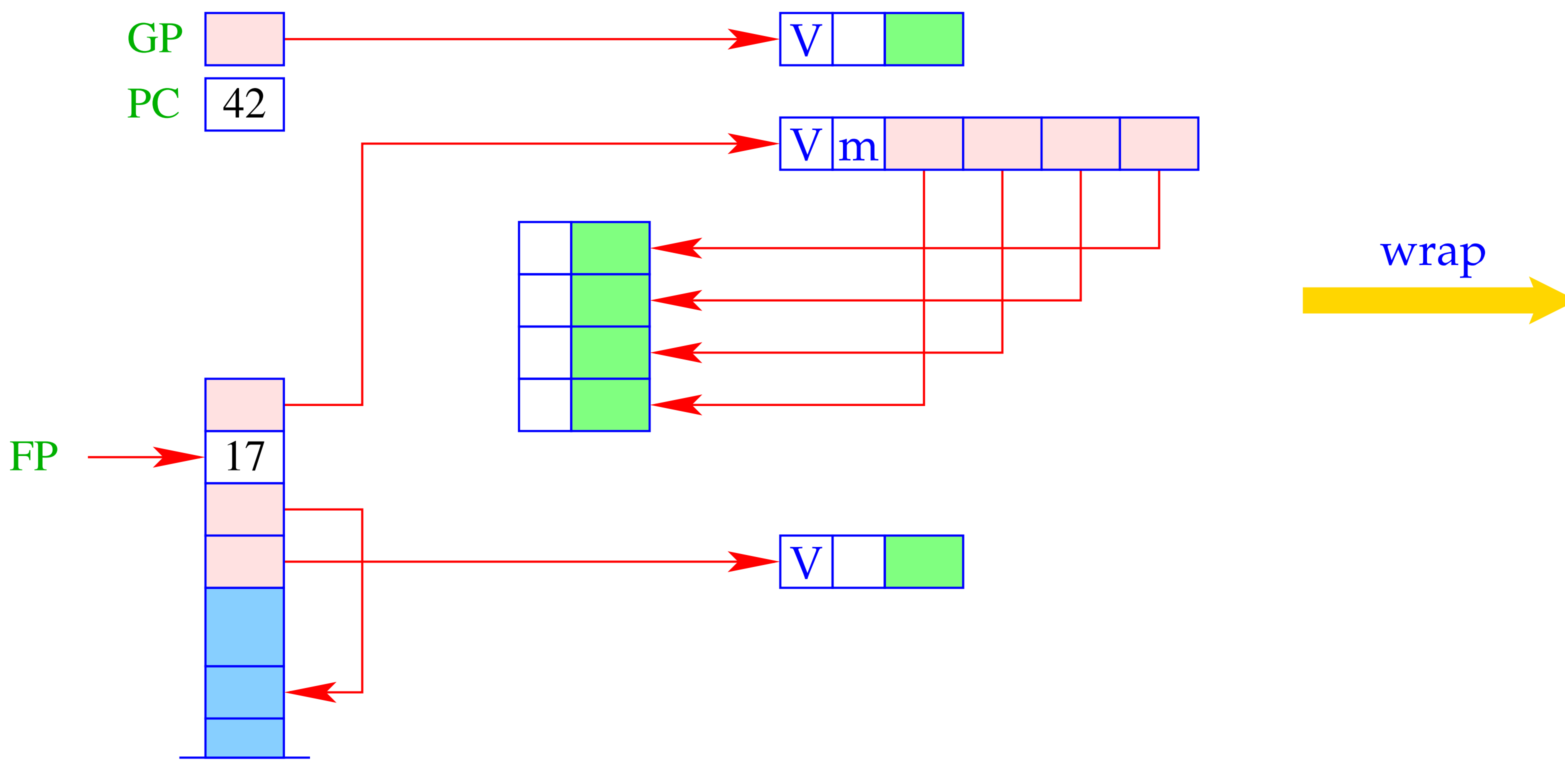
S[SP] = new (F, PC–1, S[SP], GP);

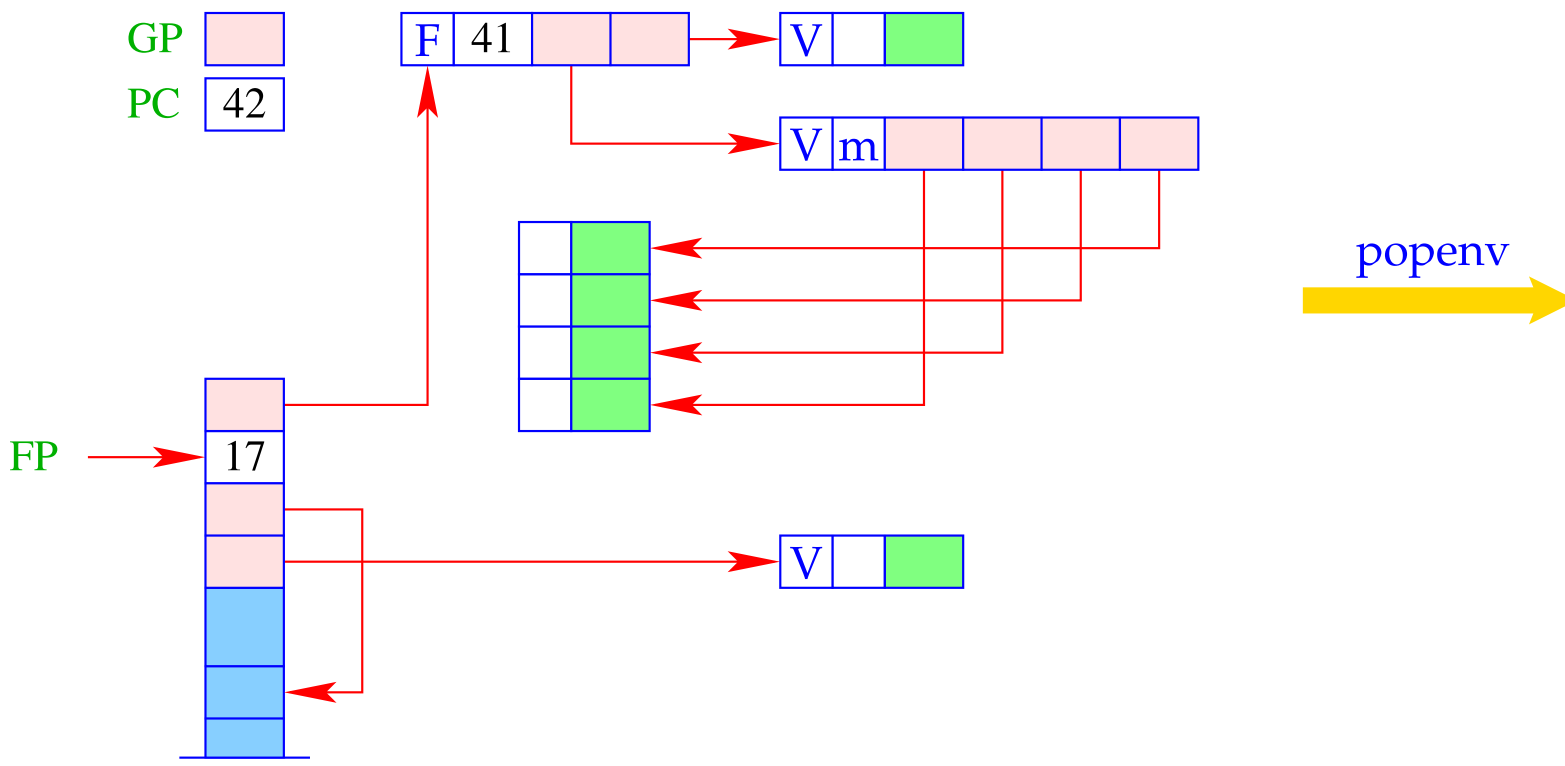


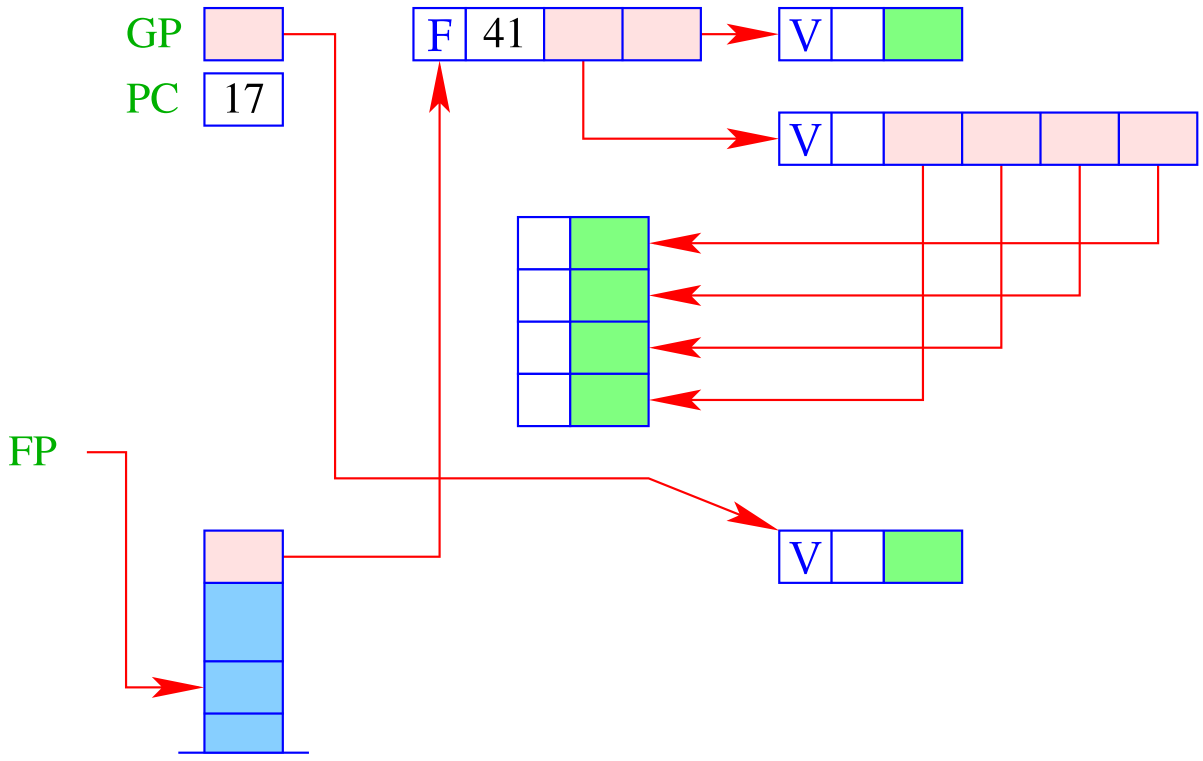
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];

Insgesamt erhalten wir damit für targ k:




GP
PC
42
V
V
m
wrap
FP
17
V

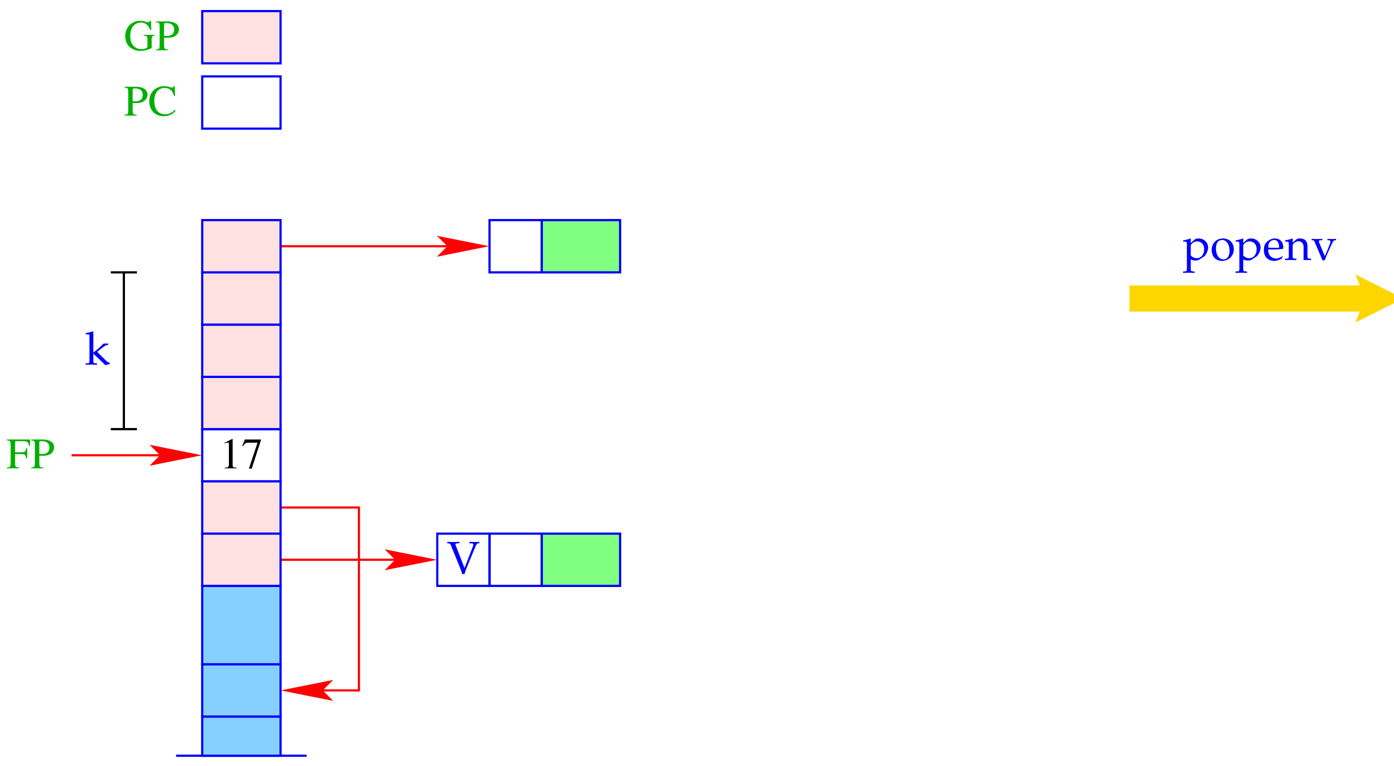

GP
PC
42
F
41
V
V
m
FP
17
V
popenv

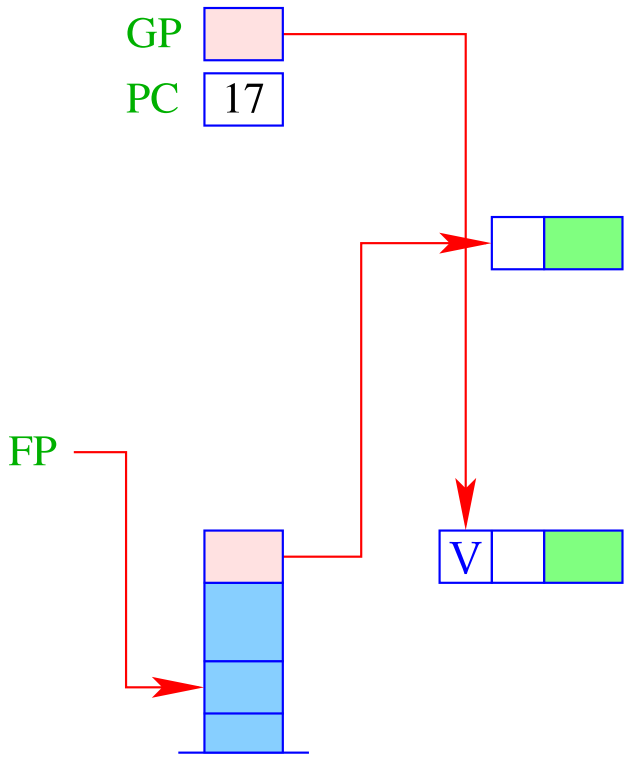

GP
PC
17
F
41
V
V
FP
V

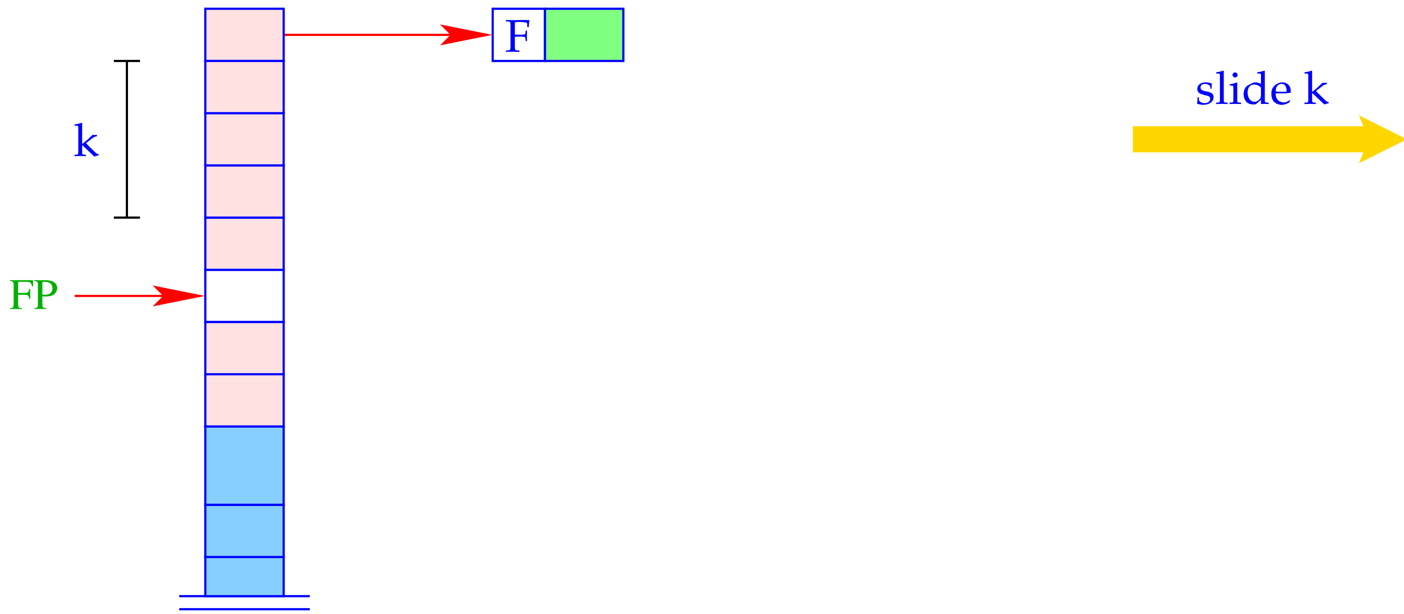
- Liegt exakt die richtige Argument-Anzahl vor, kann nach Abarbeitung des Rumpfs der Kellerrahmen aufgegeben werden.
- Liegt sogar Überversorgung mit Argumenten vor, muss der Rumpf sich offenbar erneut zu einer Funktion ausgewertet haben, die nun die restlichen Argumente konsumiert ...
- Für diese Überprüfung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

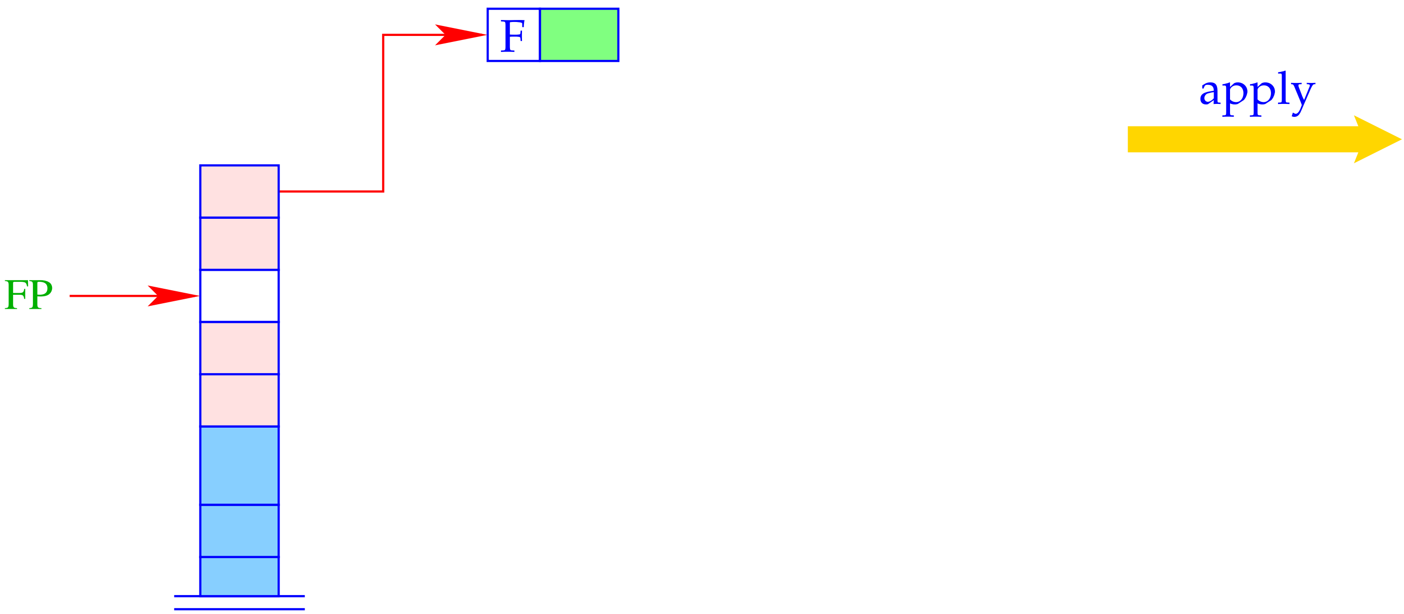
```
return k  =  if (SP − FP ≡ k + 1)
                popenv;          // Aufgeben des Kellerrahmens
             else {              // Es gibt noch weitere Argumente
                slide k;
                apply;           // erneuter Aufruf
             }
```

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
FP
apply

19 let-rec-Ausdrücke

Sei $e \equiv \textbf{let rec } y_1 = e_1 \textbf{ and} \ldots \textbf{and } y_n = e_n \textbf{ in } e_0$ ein **let-rec**-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 **let-rec**-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:

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

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

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!!!

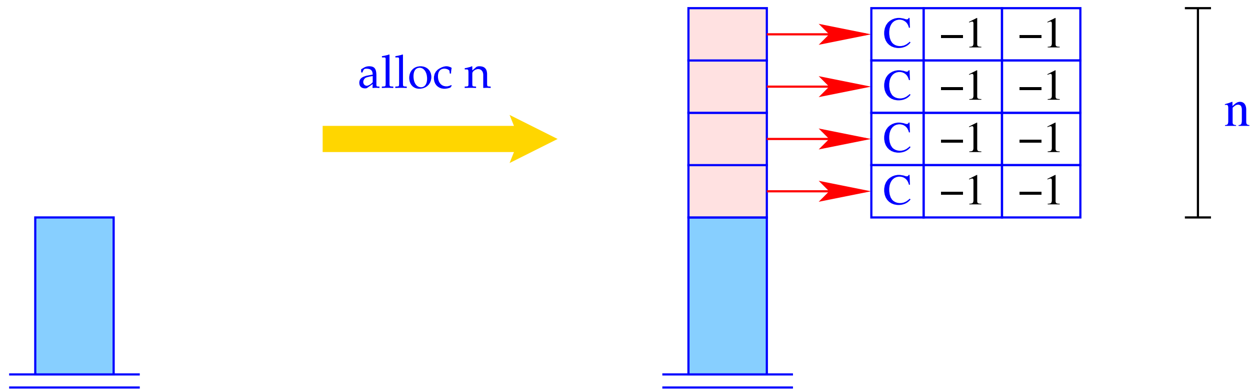
Beispiel:

Betrachte den Ausdruck

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

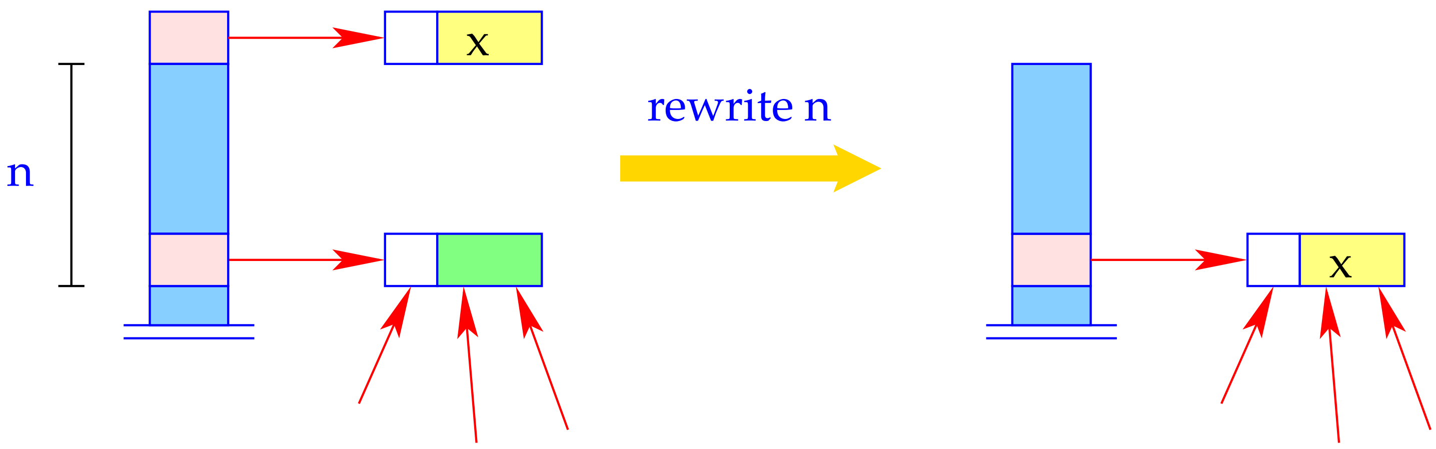
für $\rho = \emptyset$ und kp $= 0$. Dann ergibt sich (für CBV):

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!

20 Abschlüsse und ihre Auswertung

- Abschlüsse werden nur zur Implementierung von CBN benötigt.
- Bevor wir (bei CBN) auf den Wert einer Variablen zugreifen, müssen wir sicherstellen, dass der Wert bereits vorliegt.
- Ist das nicht der Fall, müssen wir einen Kellerrahmen anlegen, innerhalb dessen der Wert ermittelt wird.
- Diese Aufgabe erledigt der Befehl eval.