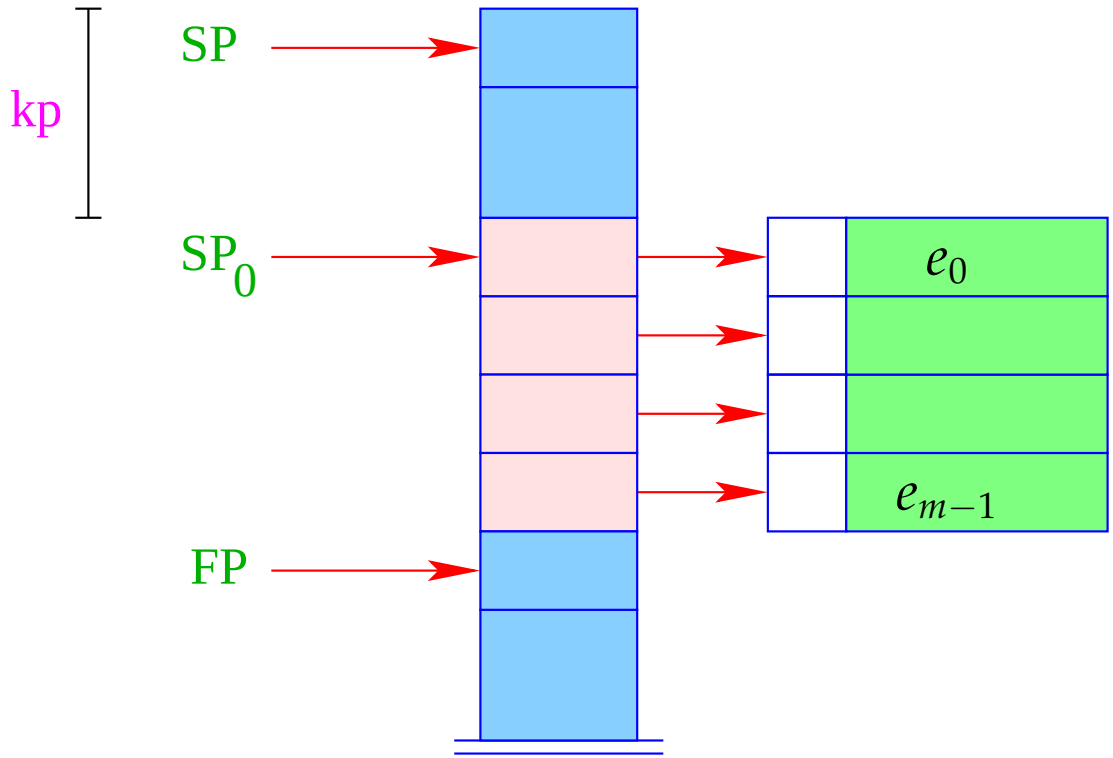
Ausweg:

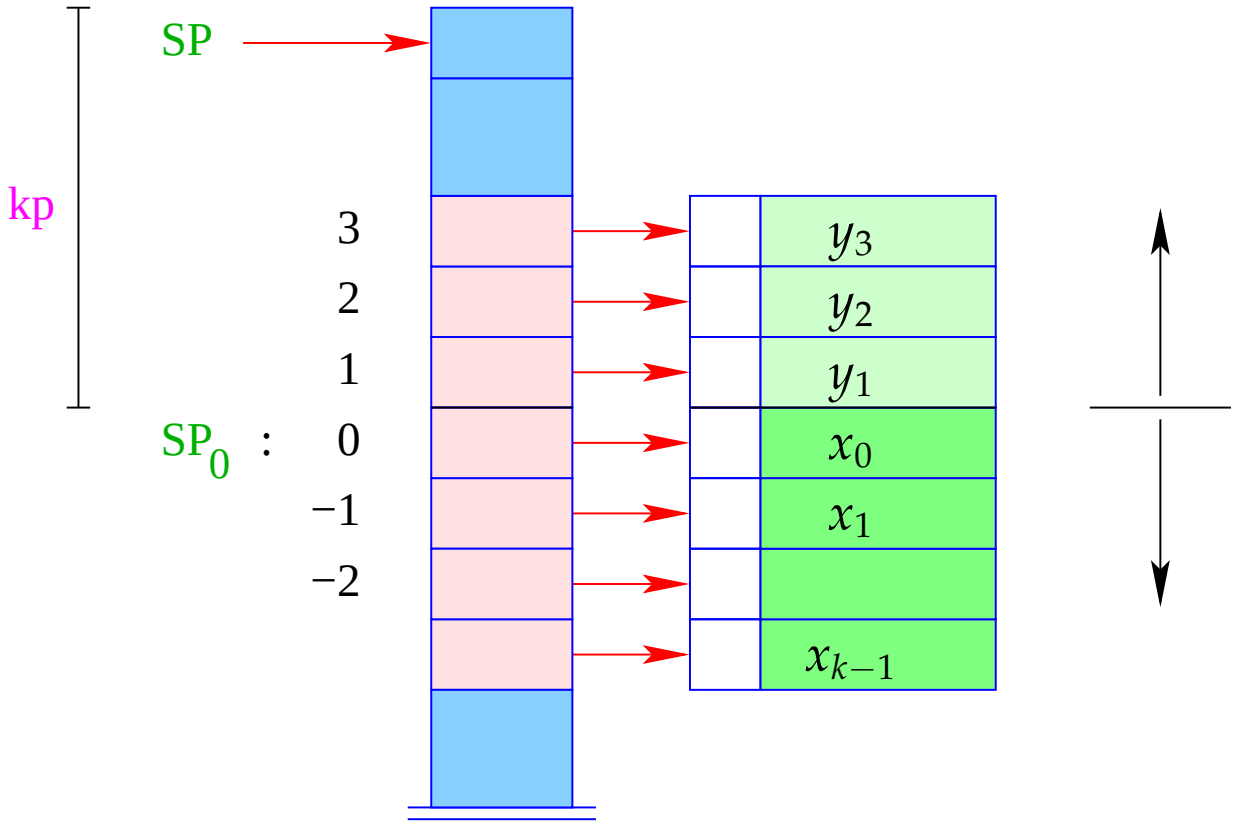
- Wir adressieren relativ zum Stackpointer SP !!!
- Leider ändert sich der Stackpointer während der Programm-Ausführung ...



- Die Abweichung des SP von seiner Position SP_0 nach Betreten eines Funktionsrumpfs nennen wir den Kellerpegel kp.
- Glücklicherweise können wir den Kellerpegel an jedem Programm-Punkt bereits zur Übersetzungszeit ermitteln :-)
- Für die formalen Parameter $x_0, x_1, x_2, \ldots$ vergeben wir sukzessive die nicht-positiven Relativ-Adressen $0, -1, -2, \ldots$, d.h. $\rho\ x_i = (L, -i)$.
- Die absolute Adresse des i-ten formalen Parameters ergibt sich dann als

$$\mathrm{SP}_0 - i = (\mathrm{SP} - \mathrm{kp}) - i$$

- Die lokalen **let**-Variablen $y_1, y_2, y_3, \ldots$ werden sukzessive oben auf dem Keller abgelegt:



- Die y_i erhalten darum positive Relativ-Adressen $1, 2, 3, \ldots$, hier: $\rho\ y_i = (L, i)$.
- Die absolute Adresse von y_i ergibt sich dann als

$$\mathrm{SP}_0 + i = (\mathrm{SP} - \mathrm{kp}) + i$$

Bei CBN erzeugen wir damit für einen Variablen-Zugriff:

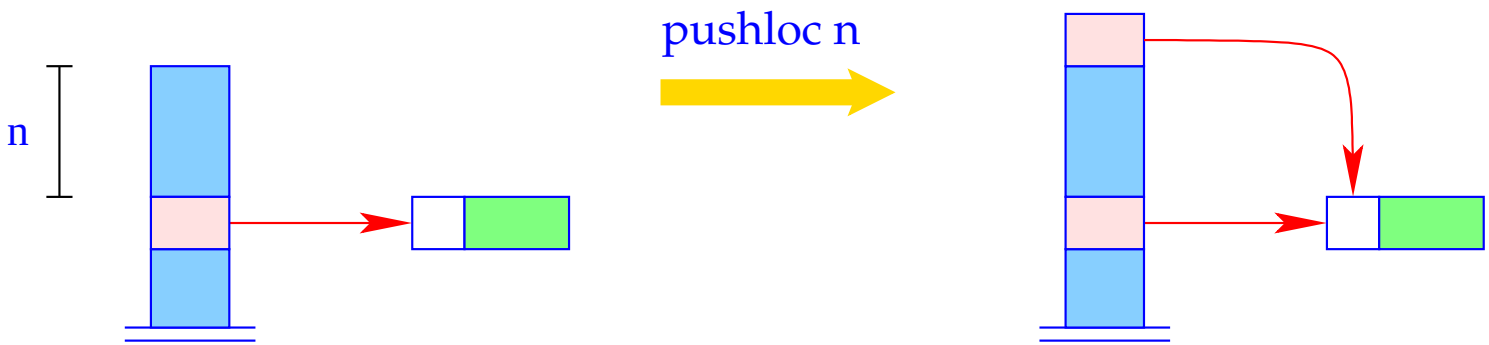
$$\begin{array}{lll} \text{code}_V\ x\ \rho\ \text{kp} & = & \text{getvar}\ x\ \rho\ \text{kp} \\ & & \text{eval} \end{array}$$

Die Instruktion eval überprüft, ob der Wert bereits berechnet wurde oder seine Auswertung erst durchgeführt werden muss ($\Longrightarrow$ kommt später :-)

Bei CBV können wir eval einfach streichen.

Das Macro getvar ist definiert durch:

$$\begin{array}{lll} \text{getvar}\ x\ \rho\ \text{kp} & = & \textbf{let}\ (t, i) = \rho\ x\ \textbf{in} \\ & & \textbf{case}\ t\ \textbf{of} \\ & & \quad L \Rightarrow \text{pushloc}\ (\text{kp} - i) \\ & & \quad G \Rightarrow \text{pushglob i} \\ & & \textbf{end} \end{array}$$



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

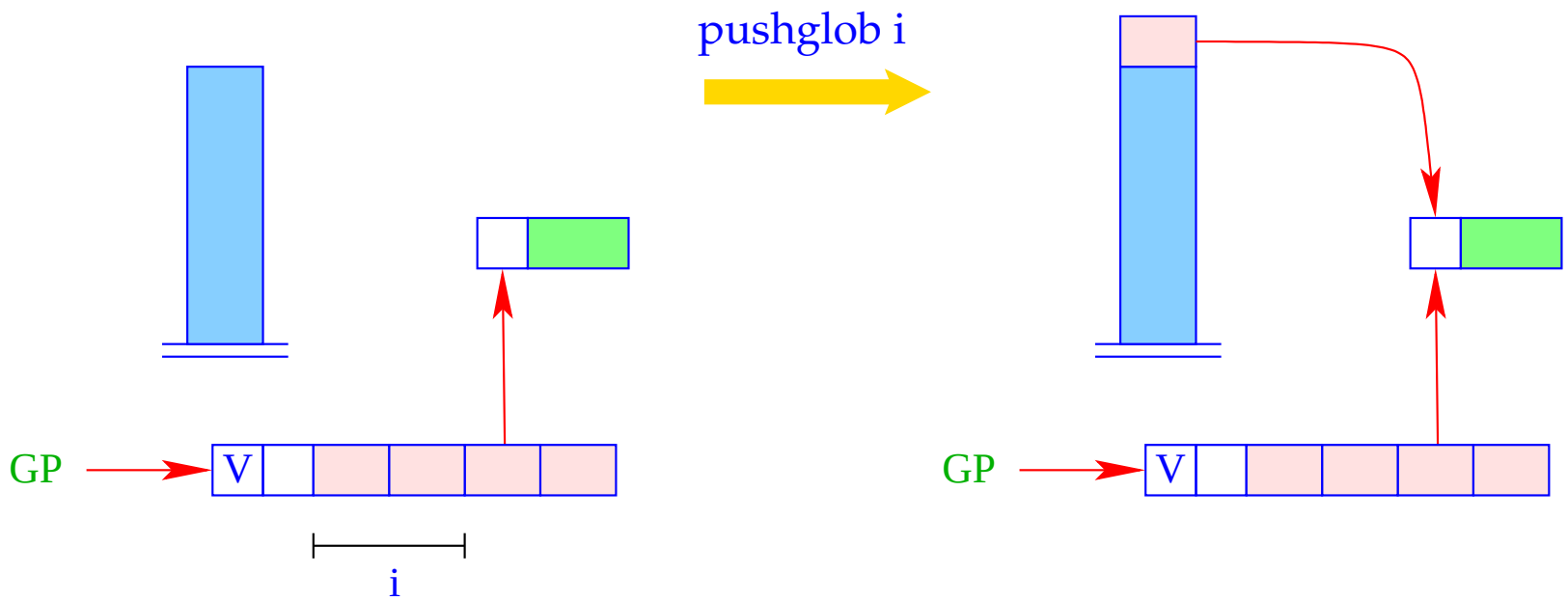
Zur Korrektheit:

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

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

... wie es auch sein soll :-)

Der Zugriff auf die globalen Variablen ist da viel einfacher:



SP = SP + 1;
S[SP] = GP→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:

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

15 let-Ausdrücke

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

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

- lokale Variablen $y_1, \ldots, y_n$ auf dem Stack 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.

Wir betrachten hier zuerst nur den nicht-rekursiven Fall, d.h. wo y_j nur von $y_1, \ldots, y_{j-1}$ abhängt. Dann erhalten wir für CBN:

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

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

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

Dann benutzen wir für die Ausdrücke $e_1, \ldots, e_n$ ebenfalls code_V.

Achtung!

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

Beispiel:

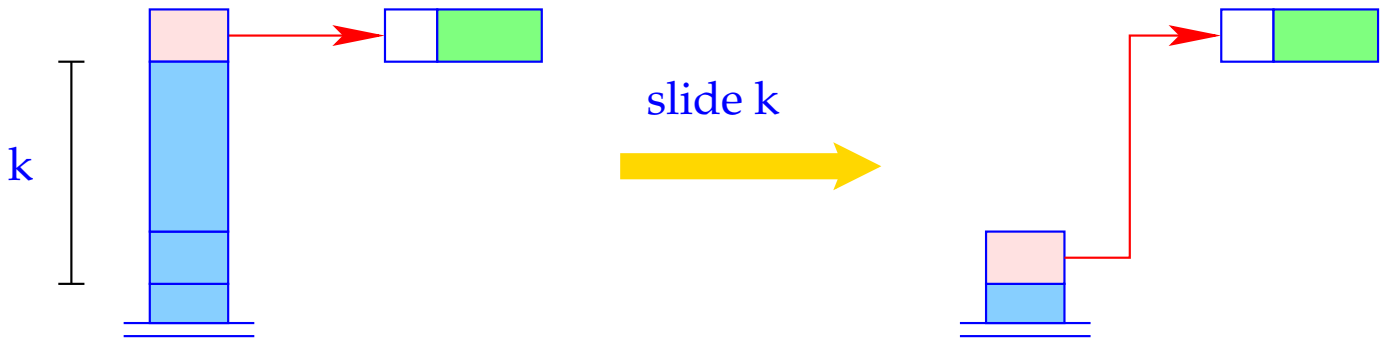
Betrachte den Ausdruck

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

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 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:

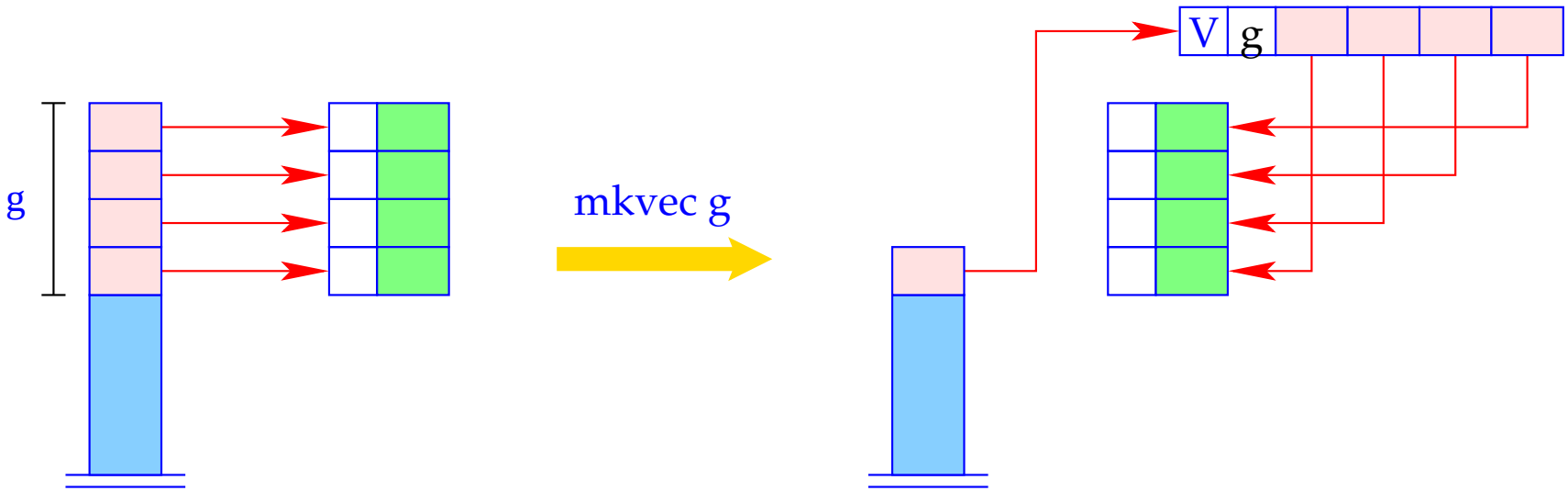
- Erzeugen des Global Vector mit den Bindungen der freien Variablen;
- Erzeugen eines (anfänglich leeren) Argument-Vektors;
- Erzeugen eines F-Objekts, das zusätzlich die Anfangs-Adresse des Codes zur Auswertung des Rumpfs enthält;
- Code zur Auswertung des Rumpfs.

Folglich:

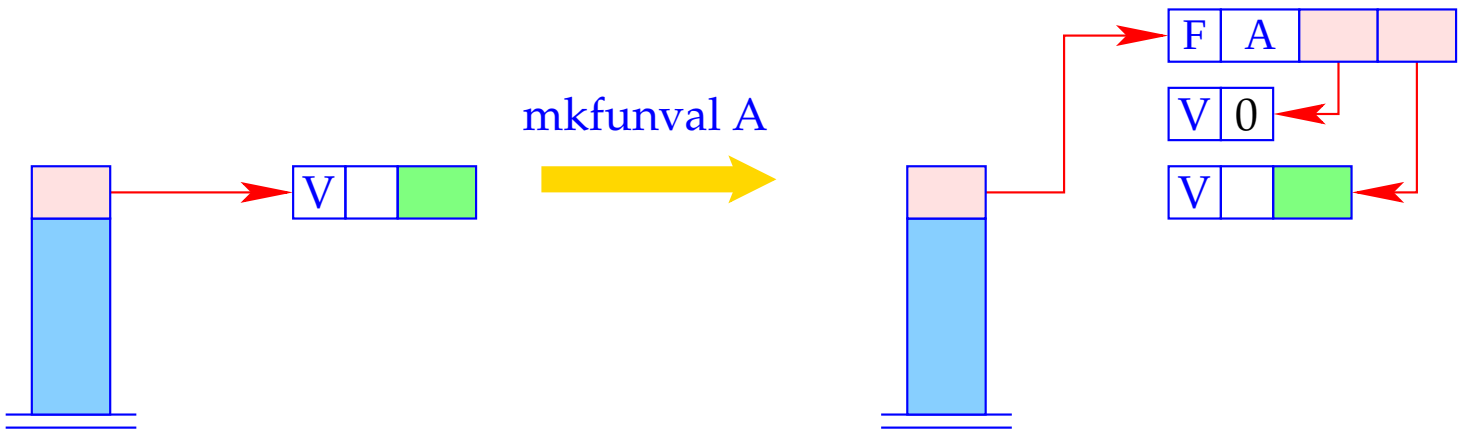
$$
\begin{array}{lll}
\text{code}_V\ (\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{kp} & = & \text{getvar}\ z_0\ \rho\ \text{kp} \\
 & & \text{getvar}\ z_1\ \rho\ (\text{kp}+1) \\
 & & \ldots \\
 & & \text{getvar}\ z_{g-1}\ \rho\ (\text{kp}+g-1) \\
 & & \text{mkvec g} \\
 & & \text{mkfunval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

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

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 $f \equiv \mathbf{fn}\ b \Rightarrow a + b$ für $\rho = \{a \mapsto (L, 1)\}$ und $\mathrm{kp} = 1$.

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

```
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 :  ...
```