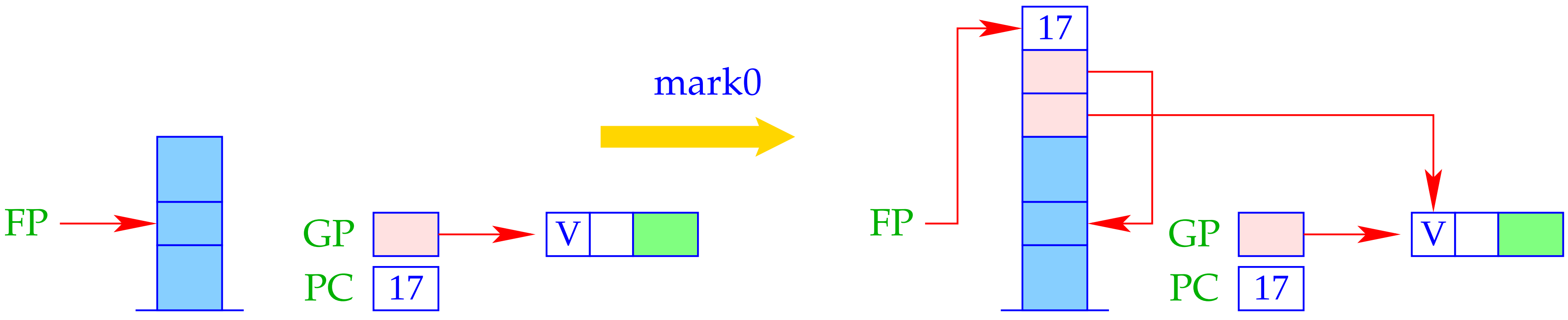
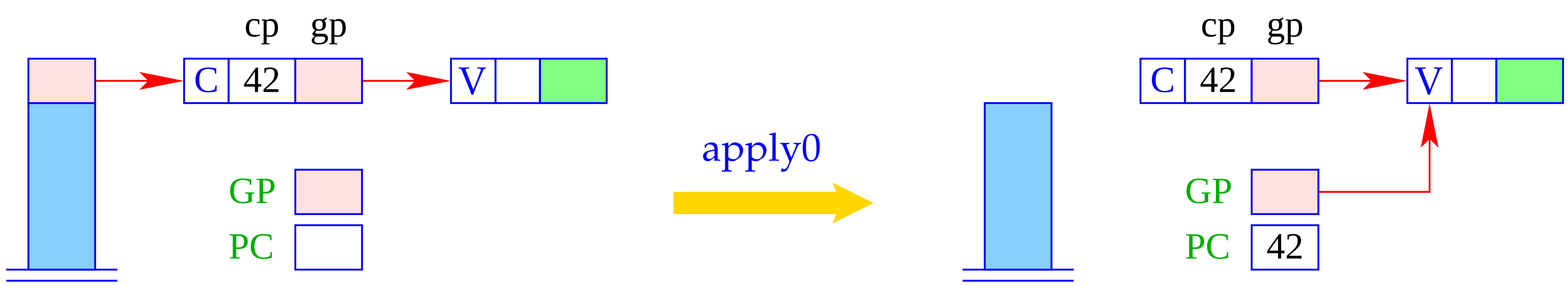
eval lässt sich wieder in übersichtlichere Bestandteile verlegen:

$$
\begin{array}{lll}
\text{eval} & = & \text{if } (H[S[SP]] \equiv (C, \_, \_)) \; \{ \\
 & & \quad \text{mark0}; \qquad\qquad\qquad // \text{ Anlegen des Kellerrahmens} \\
 & & \quad \text{pushloc } 3; \qquad\qquad\; // \text{ Kopieren des Verweises} \\
 & & \quad \text{apply0}; \qquad\qquad\quad\; // \text{ entspricht apply} \\
 & & \}
\end{array}
$$

- Ein Abschluss kann aufgefasst werden als eine parameterlose Funktion, bei der folglich auf die ap-Komponente verzichtet werden kann.
- Auswerten des Abschlusses heißt dann Auswerten einer Anwendung dieser Funktion auf 0 Argumente.
- Im Unterschied zu mark A rettet mark0 den aktuellen PC.
- Im Unterschied zu apply braucht apply0 keinen Argument-Vektor auf den Keller zu legen.

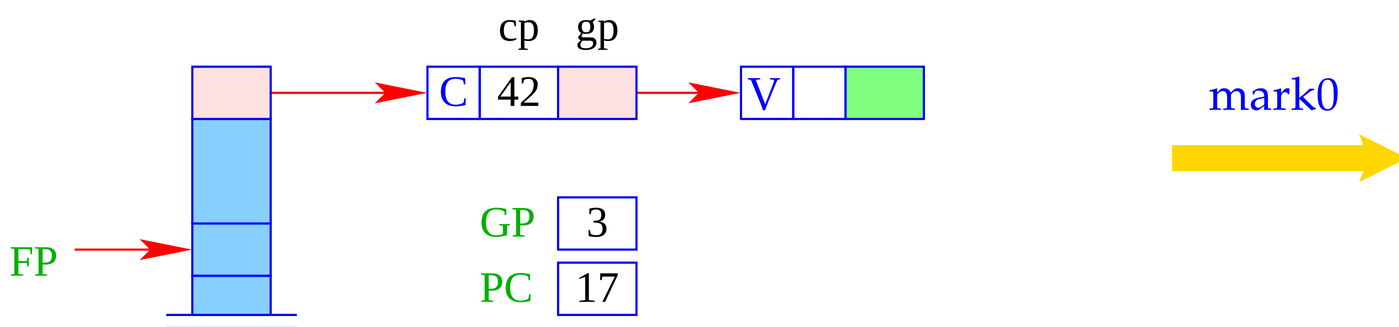


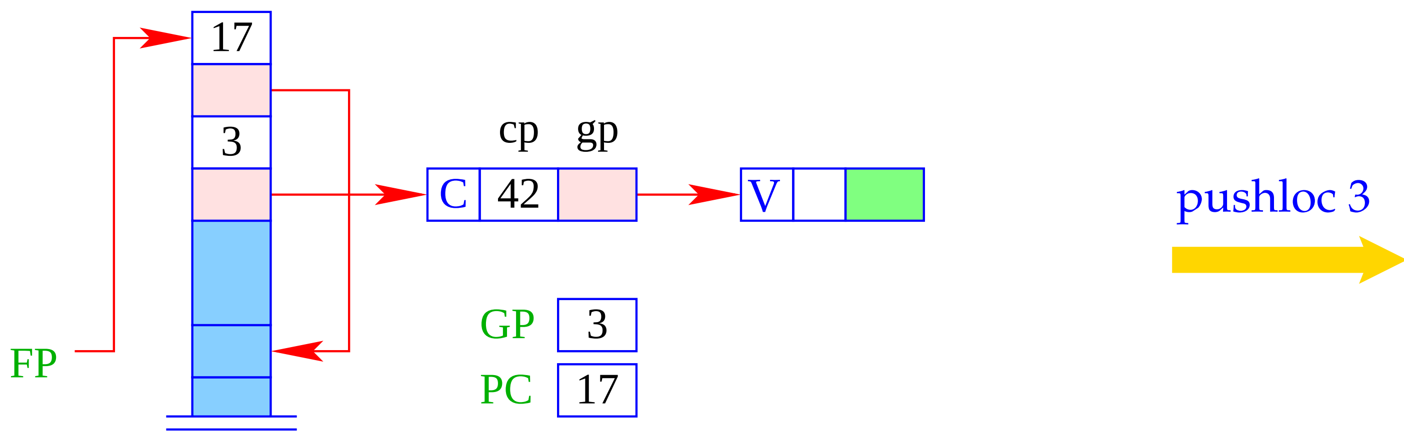
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

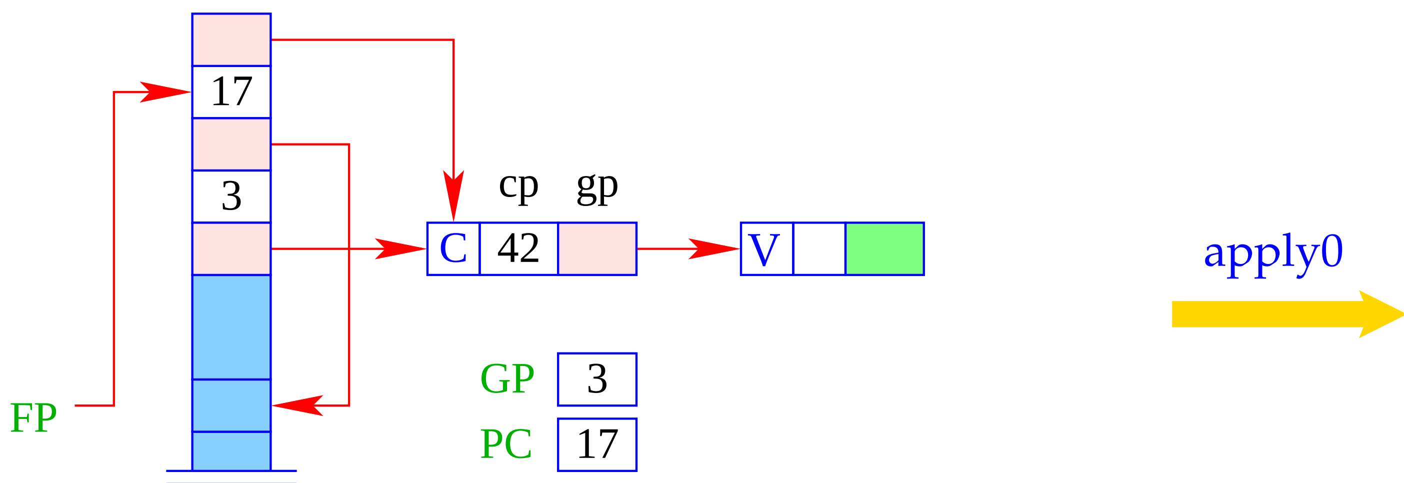


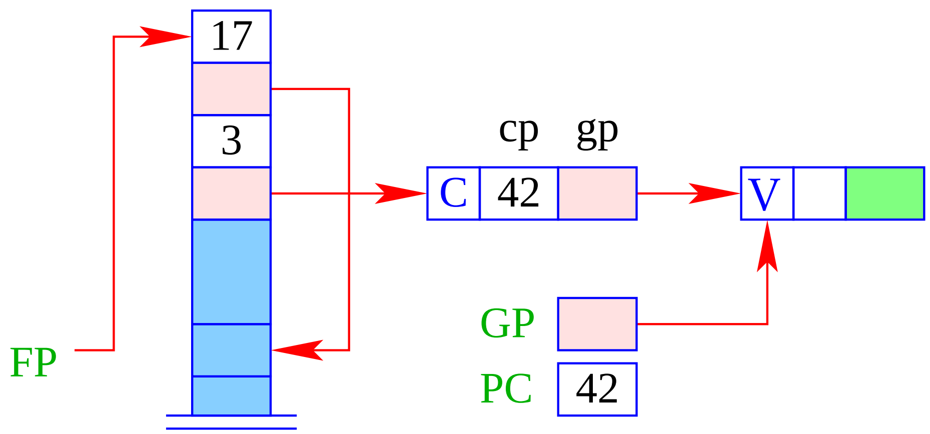
h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

Damit erhalten wir für die Instruktion eval:


cp
gp
C
42
V
FP
GP
3
PC
17
mark0


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


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



Die Herstellung eines Abschlusses für einen Ausdruck e erfordert:

- Einpacken der Bindungen für die freien Variablen;
- Erzeugen eines C-Objekts, das außerdem einen Verweis auf den Code zur Auswertung von e enthält:

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

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

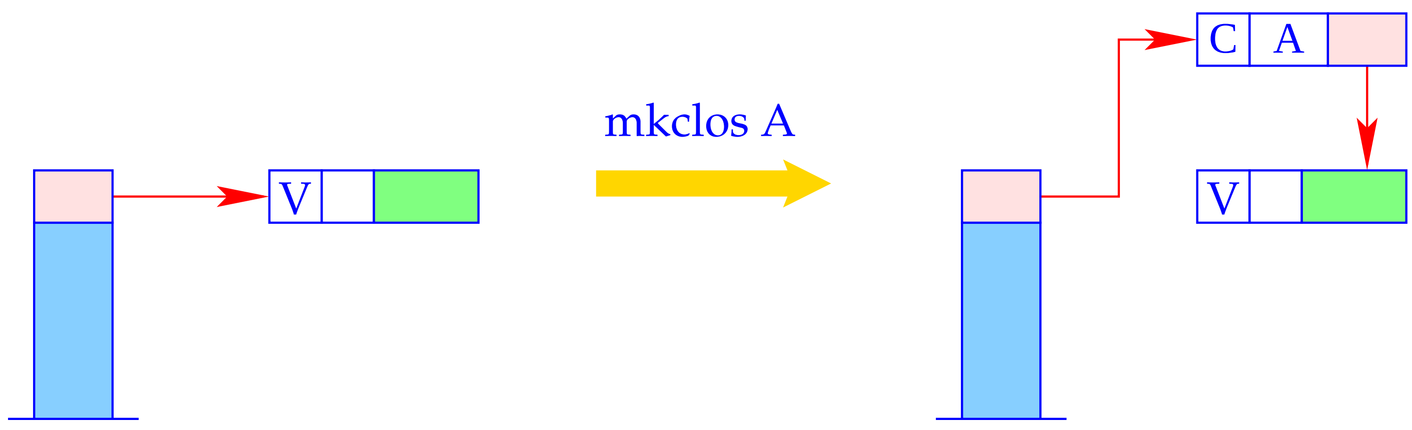
Beispiel:

Betrachte $e \equiv a * a$ mit $\rho = \{a \mapsto (L, 0)\}$ und kp $= 1$. Dann erhalten wir:

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:	...

- Die Instruktion mkclos A ist völlig analog der Instruktion mkfunval A.

- Sie erzeugt ein C-Objekt, wobei der eingepackte Code-Pointer gerade A ist.



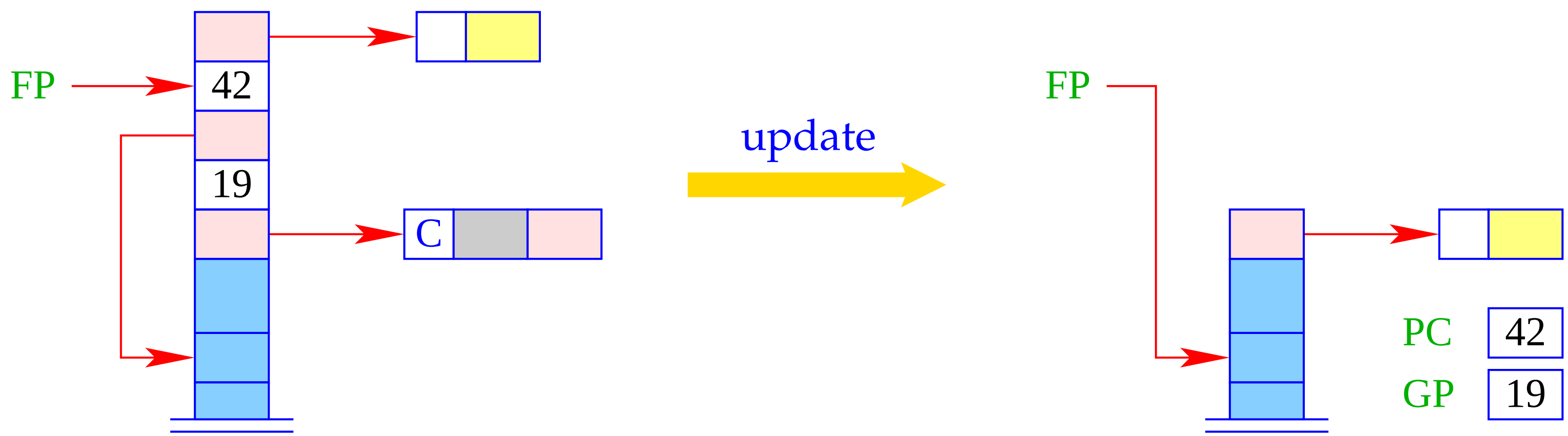
S[SP] = new (C, A, S[SP]);

Die Instruktion update ist in Wirklichkeit die Kombination der beiden Instruktionen:

popenv

rewrite 1

Sie überschreibt den Abschluss mit dem berechneten Wert.



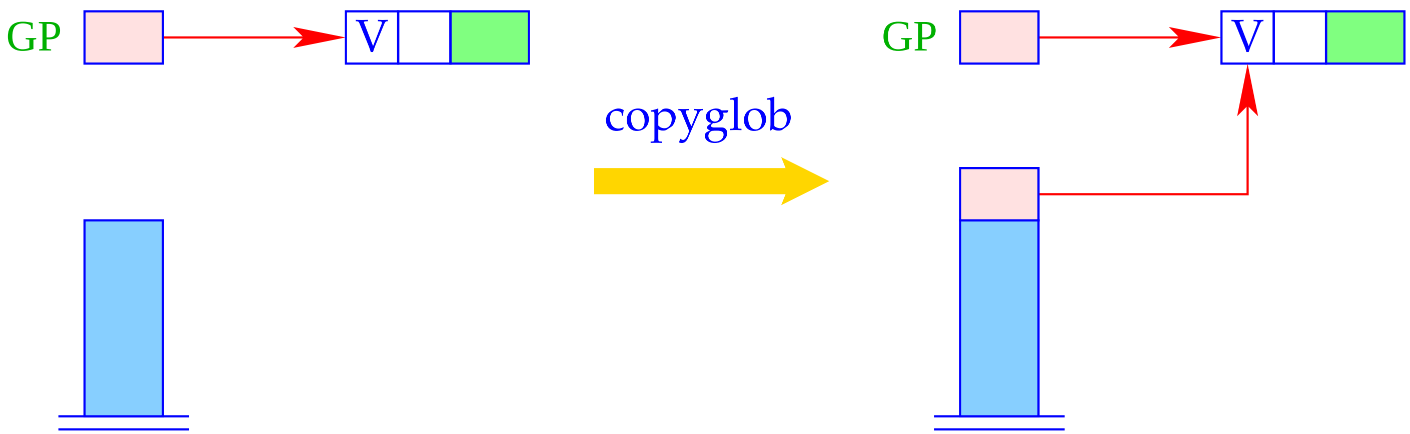
21 Optimierungen I: globale Variablen

Beobachtung:

- In funktionalen Programmen werden viele F- oder C-Objekte konstruiert.
- Dies erfordert das Einpacken sämtlicher globalen Variablen.

Idee:

- Gestatte die Mehrfachverwendung von Global-Vektoren!
- Das macht Sinn etwa bei der Übersetzung von **let**/**let-rec**-Ausdrücken oder Funktionsanwendungen.
- Lege mehrfach benutzte Global-Vektoren wie lokale Variablen in den Kellerrahmen!
- Gestatte auch den Zugriff auf den aktuellen Global-Vektor, z.B. mit einer Instruktion copyglob :



SP++;
S[SP] = GP;

- Die Optimierung ist häufiger anwendbar, wenn wir auch Global-Vektoren gestatten, die mehr Komponenten enthalten als nur die Variablen, die im Ausdruck vorkommen ...

Vorteil: Die Konstruktion von F- und C-Objekten wird "öfter" billiger :-)

Nachteil: Überflüssige Komponenten in Global-Vektoren behindern frühzeitige Freigabe nicht mehr benötigter Heap-Objekte $\Longrightarrow$ Space Leaks :-(

22 Optimierungen II: Abschlüsse

In einigen Fällen ist der Aufbau eines Abschlusses ganz überflüssig:

Basiswerte: Der Aufbau eines Abschlusses zur Berechnung des Werts ist mindestens so teuer, wie die Konstruktion eines B-Objekts selbst!

Darum:

$$\text{code}_C\ b\ \rho\ \text{kp} \quad = \quad \text{code}_V\ b\ \rho\ \text{kp} \quad = \quad \begin{array}{l}\text{loadc b}\\ \text{mkbasic}\end{array}$$

Dies ersetzt:

mkvec 0		jump B	mkbasic	B:	...
mkclos A	A:	loadc b	update		

Variablen: Variablen sind entweder an Werte oder bereits an C-Objekte gebunden. Erneute Konstruktion eines Abschlusses erscheint darum unsinnig. Deshalb:

$$\text{code}_C\ x\ \rho\ \text{kp} \quad = \quad \text{getvar}\ x\ \rho\ \text{kp}$$

Dies ersetzt:

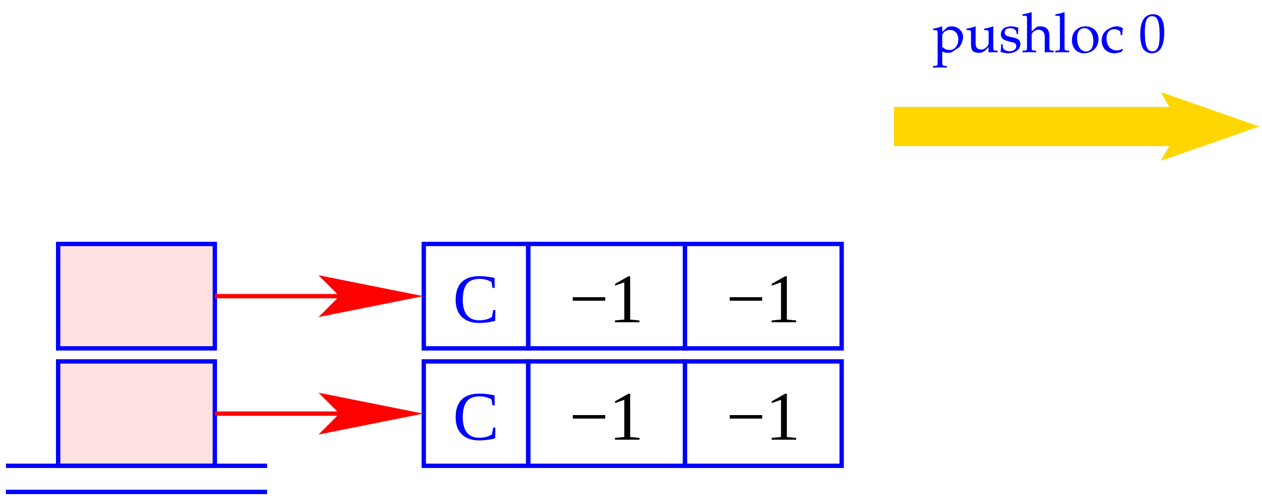
getvar x ρ kp	mkclos A	A:	pushglob 0		update
mkvec 1	jump B		eval	B:	...

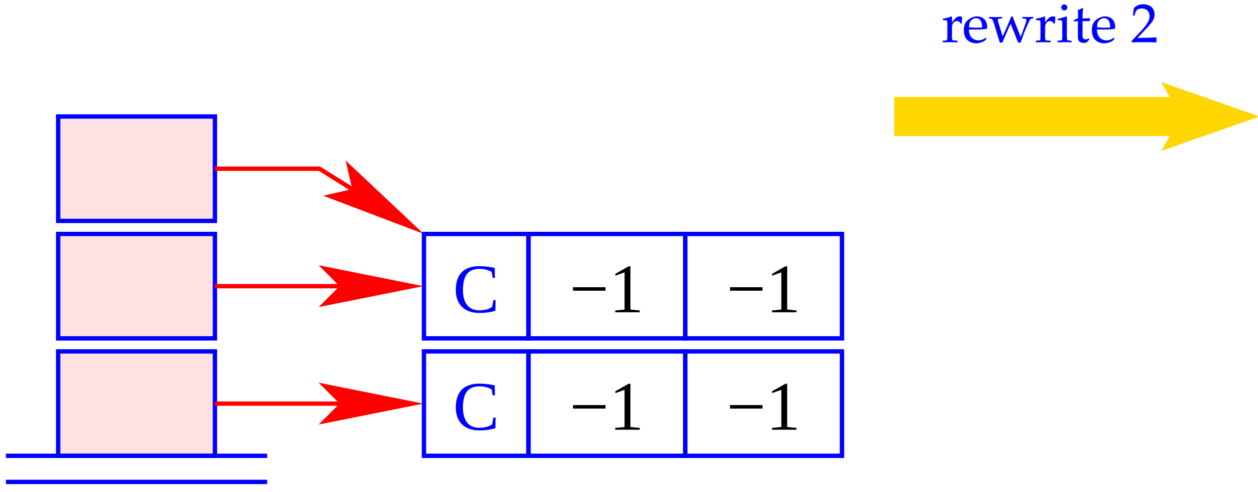
Beispiel: $e \equiv$ **let rec** $a = b$ **and** $b = 7$ **in** a. Dann liefert $\text{code}_V\ e\ \emptyset\ 0$:

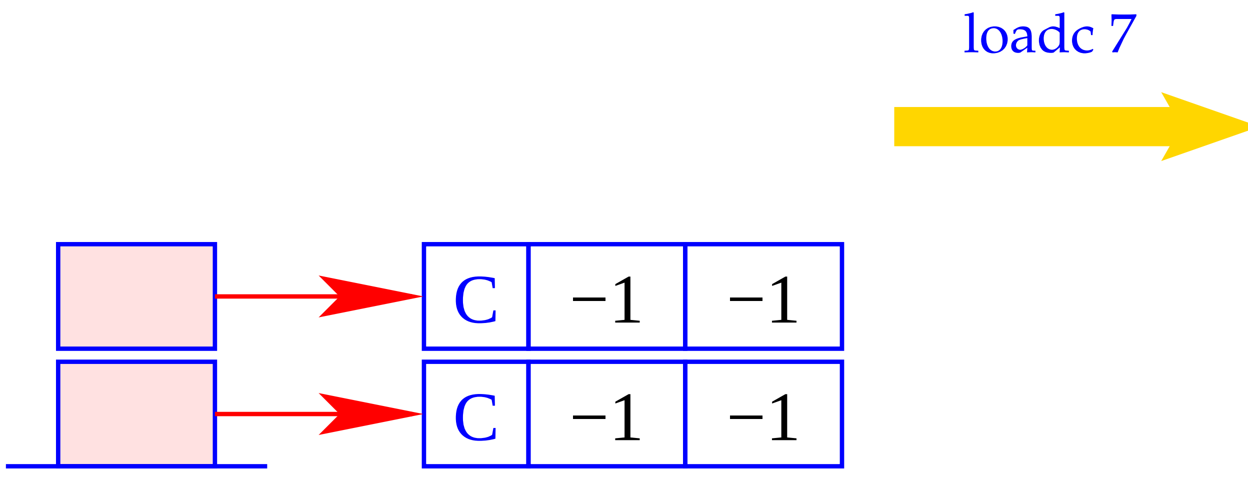
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

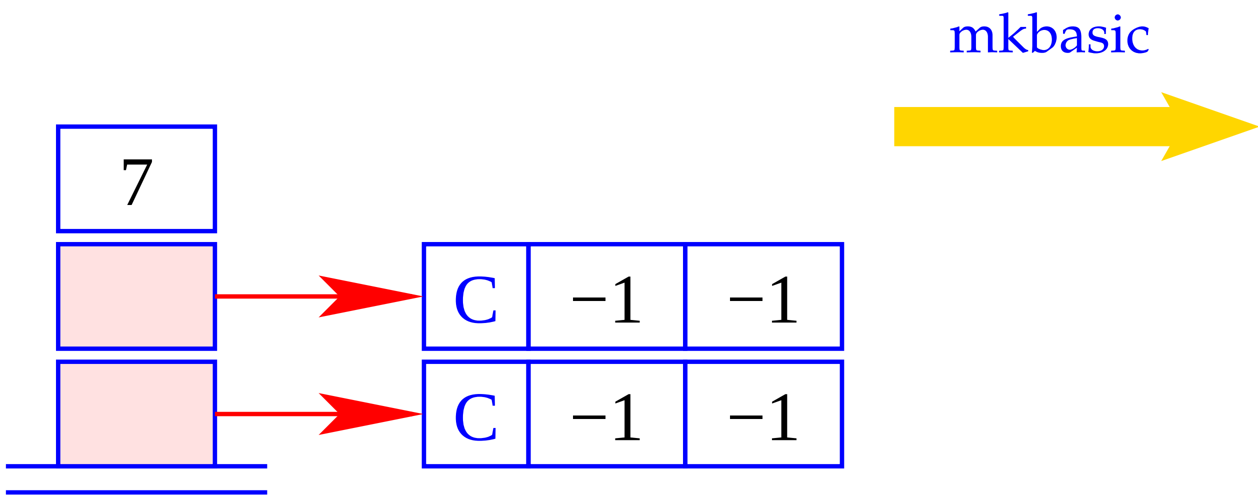
Ausführung dieser Folge sollte den Basiswert 7 liefern ...

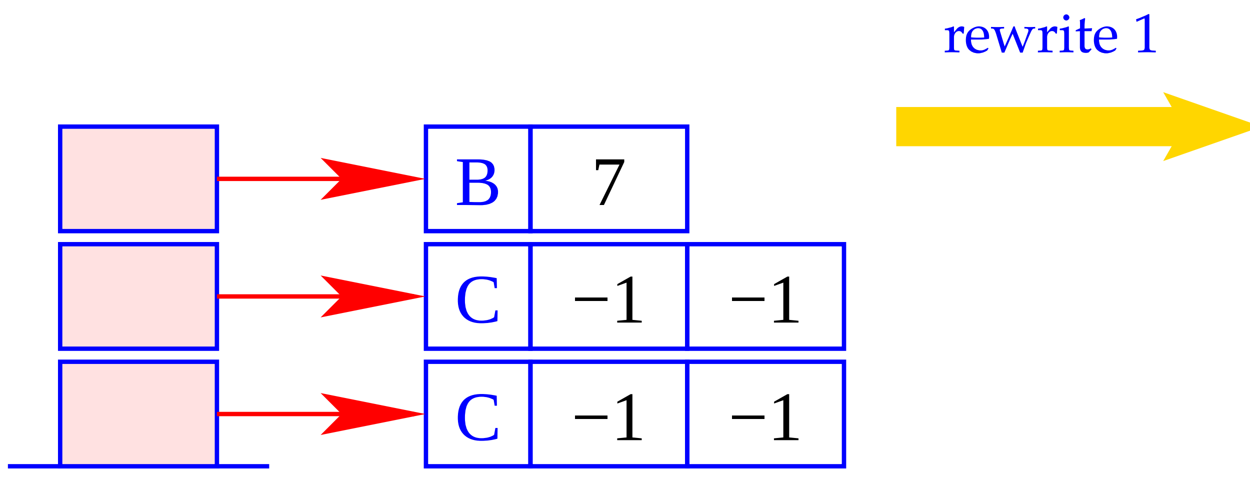
alloc 2

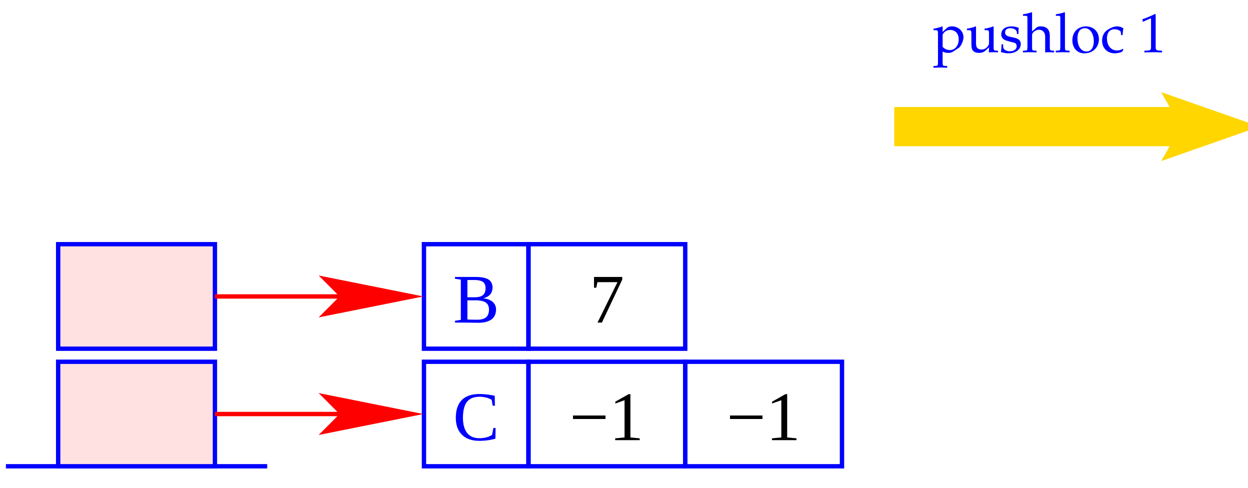

pushloc 0
C
−1
−1
C
−1
−1

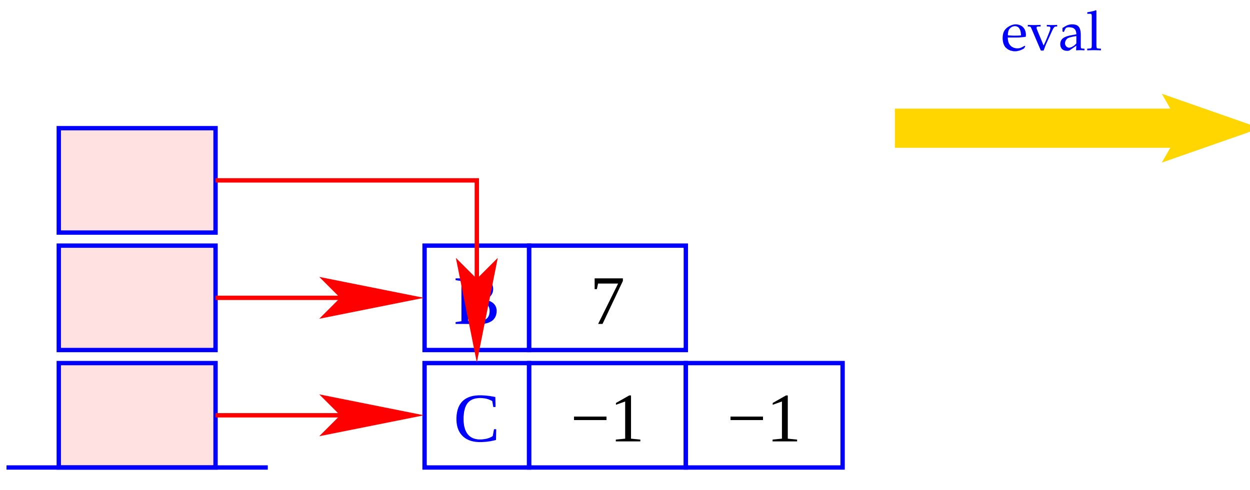

rewrite 2
C
−1
−1
C
−1
−1


loadc 7
C
−1
−1
C
−1
−1


mkbasic
7
C
−1
−1
C
−1
−1


rewrite 1
B
7
C
−1
−1
C
−1
−1


pushloc 1
B
7
C
−1
−1


eval
B
7
C
−1
−1

Segmentation Fault !!

Offenbar war die Optimierung nicht ganz korrekt :-(

Das Problem:

Bindungen von Variablen an Variablen, bei denen eine Variable y als rechte Seite verwendet wird, bevor ihr Dummy-Knoten überschrieben wurde!!

$$\Longrightarrow$$

Dieses Problem kann glücklicherweise stets umgangen werden durch:

- Zurückweisung zyklischer Variablen-Definitionen (wie $y = y$) sowie im azyklischen Fall
- Anordnung der Definitionen $y_i = y_j$ so, dass der Dummy-Knoten für die rechte Seite y_j stets bereits überschrieben ist.

Funktionen: Funktionen sind Werte, die nicht weiter evaluiert werden. Anstelle Code zu erzeugen, der ein F-Objekt erzeugt, können wir dieses auch direkt anlegen.

Darum:

$$\text{code}_C\ (\textbf{fun}\ x_0, \ldots, x_{k-1} \to e)\ \rho\ \text{kp} \quad = \quad \text{code}_V\ (\textbf{fun}\ x_0, \ldots, x_{k-1} \to e)\ \rho\ \text{kp}$$

Den Vergleich mit der entsprechenden un-optimierten Befehlsfolge überlassen wir einer Übungsaufgabe :-)

23 Die Übersetzung eines Programm-Ausdrucks

Die Programm-Ausführung für ein Programm e startet mit

$$\text{PC} = 0 \qquad \text{SP} = \text{FP} = \text{GP} = -1$$

Der Ausdruck e sollte keine freien Variablen enthalten.

Der Wert von e soll ermittelt und dann eine Instruktion halt ausgeführt werden:

$$\begin{array}{lll} \text{code}\ e & = & \text{code}_V\ e\ \emptyset\ 0 \\ & & \text{halt} \end{array}$$

Bemerkung:

- Die Code-Schemata, so wie wir sie bisher definiert haben, liefern Spaghetti-Code.
- Der Grund liegt darin, dass wir den Code zur Auswertung von Funktions-Rümpfen und Abschlüssen stets direkt hinter den Befehlen mkfunval bzw. mkclos ablegten.
- Dieser Code könnte aber auch an einer beliebigen anderen Stelle im Programm stehen, also z.B. hinter dem halt-Befehl:

  Vorteil: Die direkten Sprünge nach mkfunval und mkclos fallen weg.

  Nachteil: Die Code-Erzeugungs-Funktion wird umständlicher, da sie zusätzlich einen Code-Dump verwalten müsste, in dem die ausgelagerten Programm-Stücke akkumuliert werden.

$$\Longrightarrow$$

Überlasse die Sprung-Entwirrung einer separaten Optimierungs-Phase :-)

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

Entwirrung der Sprünge liefert:

0	loadc 17	2	mark B	3	B:	slide 2	1	pushloc 1
1	mkbasic	5	loadc 42	1		halt	2	eval
1	pushloc 0	6	mkbasic	0	A:	targ 1	2	getbasic
2	mkvec 1	6	pushloc 4	0		pushglob 0	2	add
2	mkfunval A	7	eval	1		eval	1	mkbasic
		7	apply	1		getbasic	1	return 1