

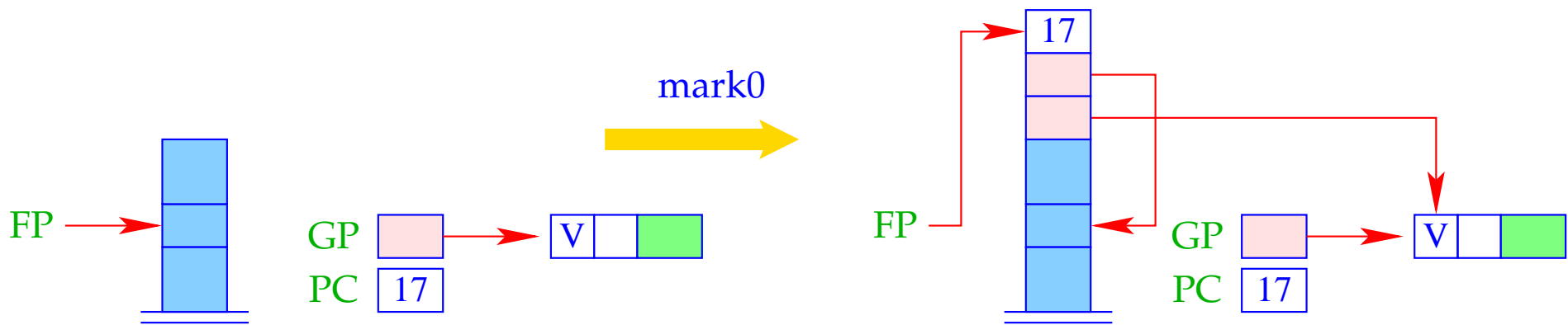
## 20 Abschlüsse und ihre Auswertung

- 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`.

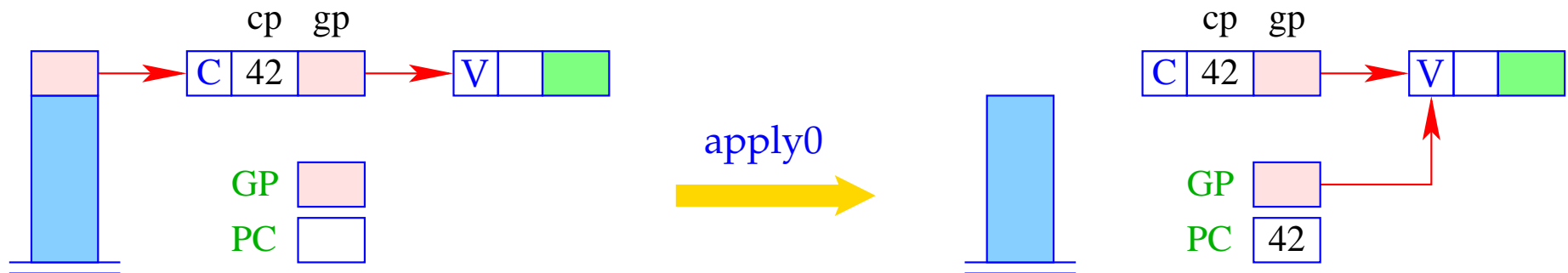
`eval` lässt sich wieder in übersichtlichere Bestandteile verlegen:

```
eval = if (H[S[SP]] ≡ (C, _, _)) {
    mark0;           // Anlegen des Kellerrahmens
    pushloc 3;      // Kopieren des Verweises
    apply0;         // entspricht apply
}
```

- 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.

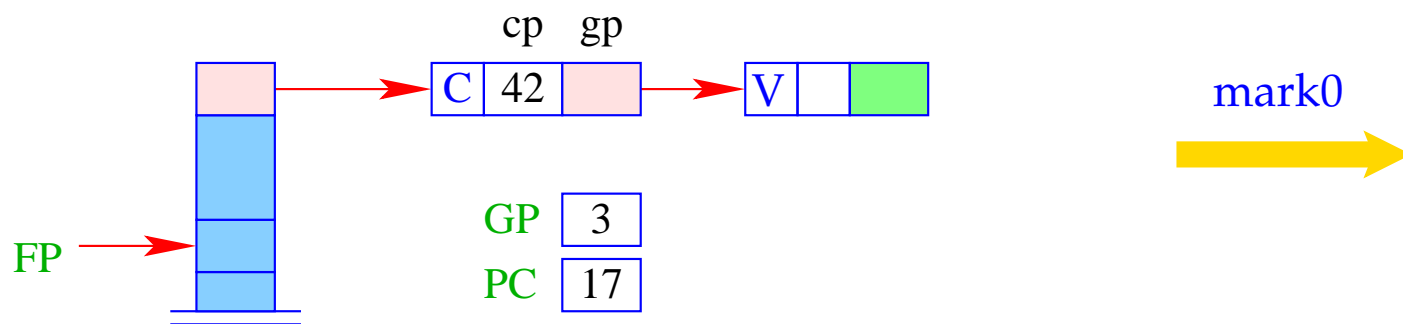


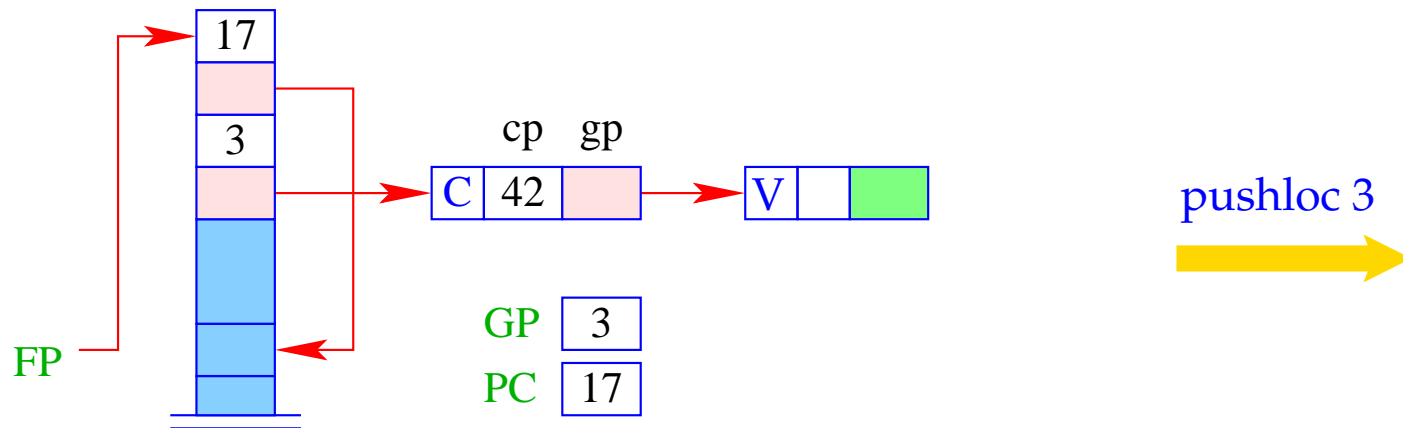
$S[SP+1] = GP;$   
 $S[SP+2] = FP;$   
 $S[SP+3] = PC;$   
 $FP = SP = SP + 3;$

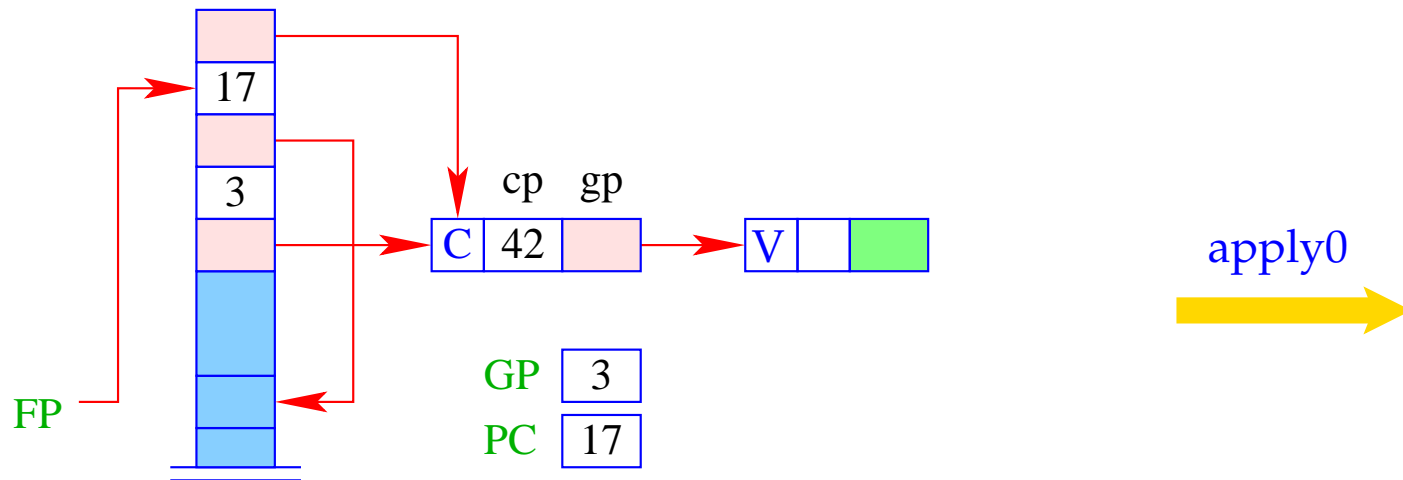


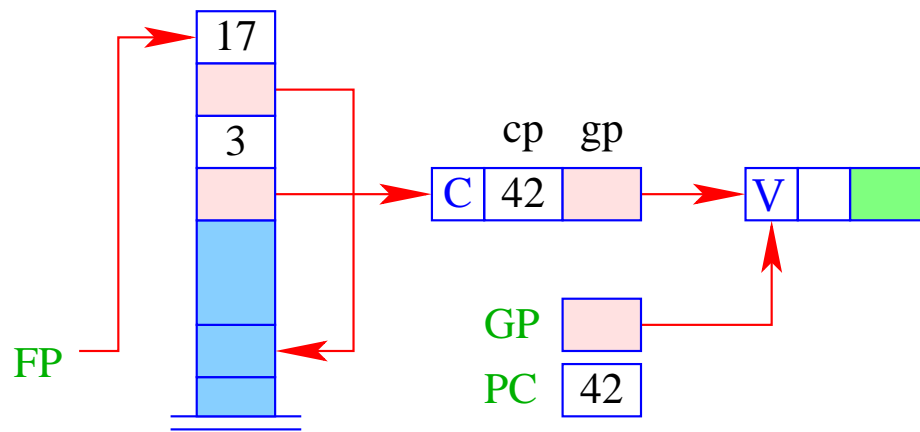
$h = S[SP]; SP--;$   
 $GP = h \rightarrow gp; PC = h \rightarrow cp;$

Damit erhalten wir für die Instruktion `eval`:









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:

```

codeC e ρ kp =   getvar z0 ρ kp
                  getvar z1 ρ (kp + 1)
                  ...
                  getvar zg-1 ρ (kp + g - 1)
                  mkvec g
                  mkclos A
                  jump B
A : codeV e ρ' 0
    update
B : ...

```

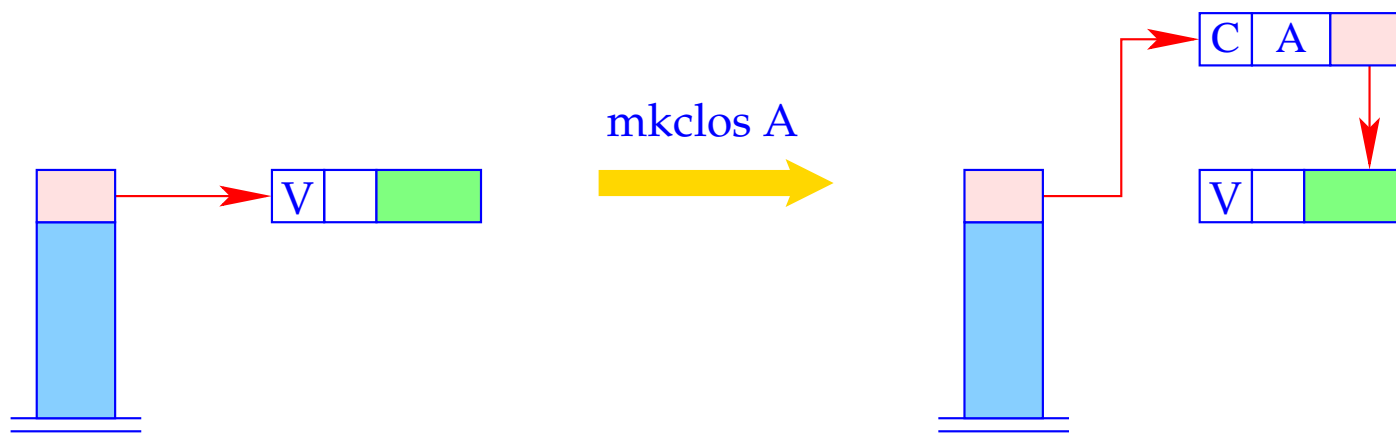
wobei  $\{z_0, \dots, z_{g-1}\} = \text{free}(e)$  und  $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \dots, g - 1\}$ .

## 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	mkcloc 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.



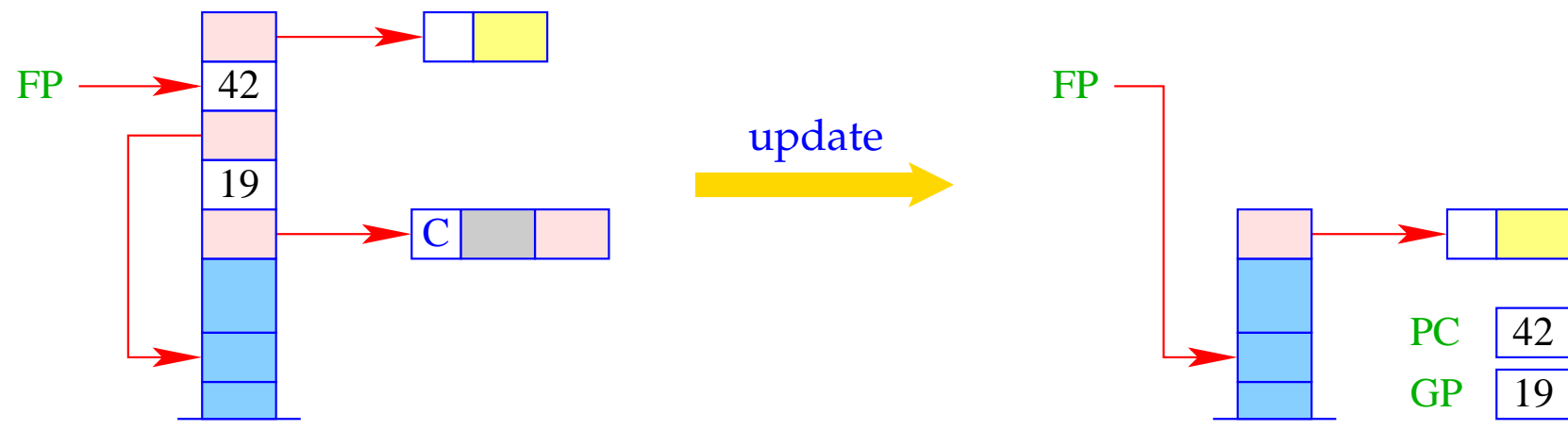
`S[SP] = new (C, A, S[SP]);`

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

`popenv`

`rewrite 1`

Sie überschreibt den Abschluss mit dem berechneten Wert.



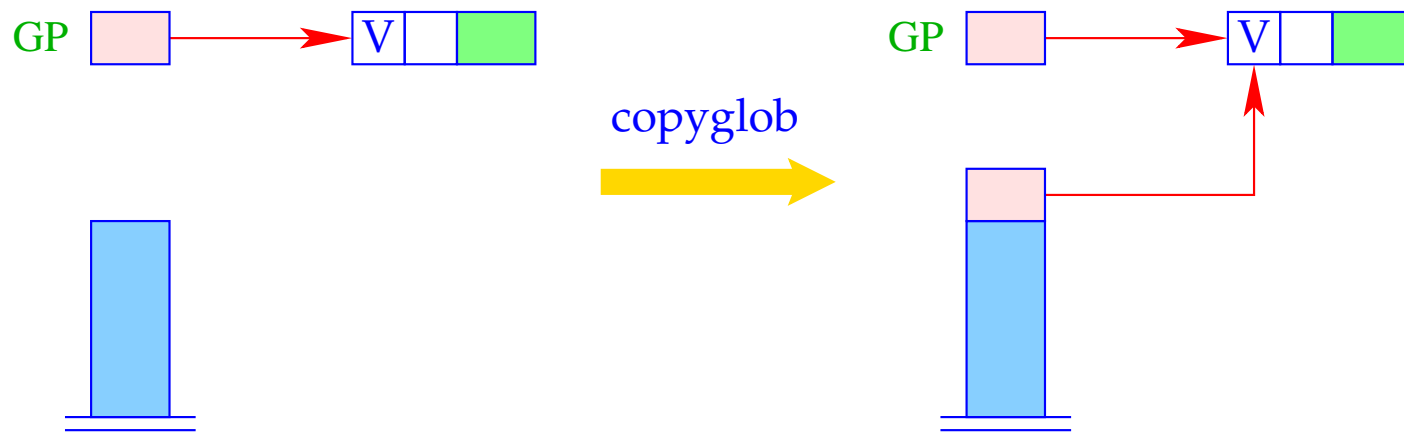
## 21 Optimierungen I: globale Variablen

### 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/letrec**-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  $\implies$  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:

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

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:

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

Dies ersetzt:

<b>getvar</b> $x \rho \text{kp}$	mkclos A	A:	pushglob 0	update
mkvec 1	jump B		eval	B: ...

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

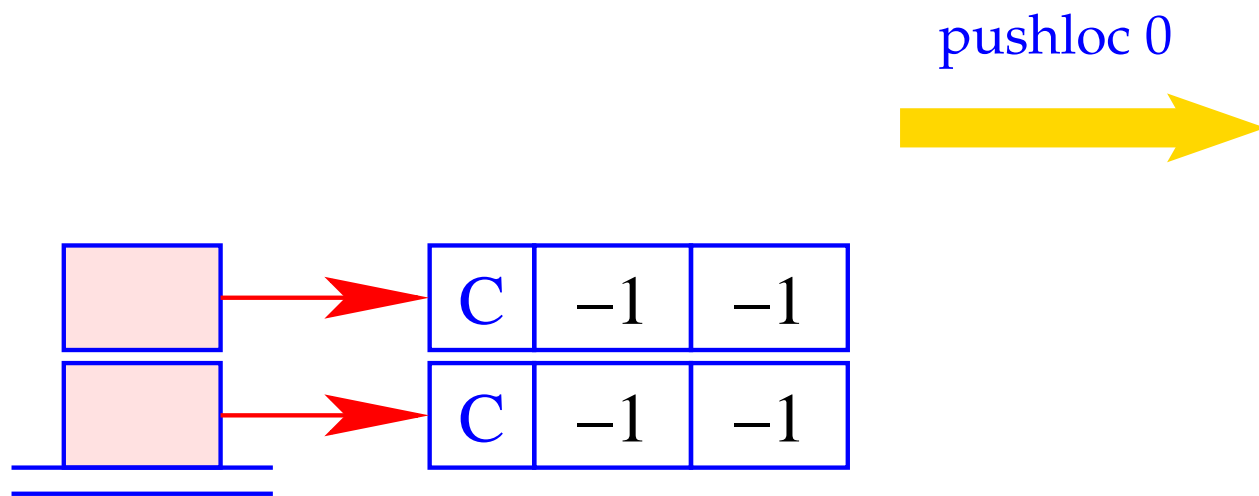
0	alloc 2	3	rewrite 2	3	mkbasic	2	pushloc 1
2	pushloc 0	2	loadc 7	3	rewrite 1	3	eval
						3	slide 2

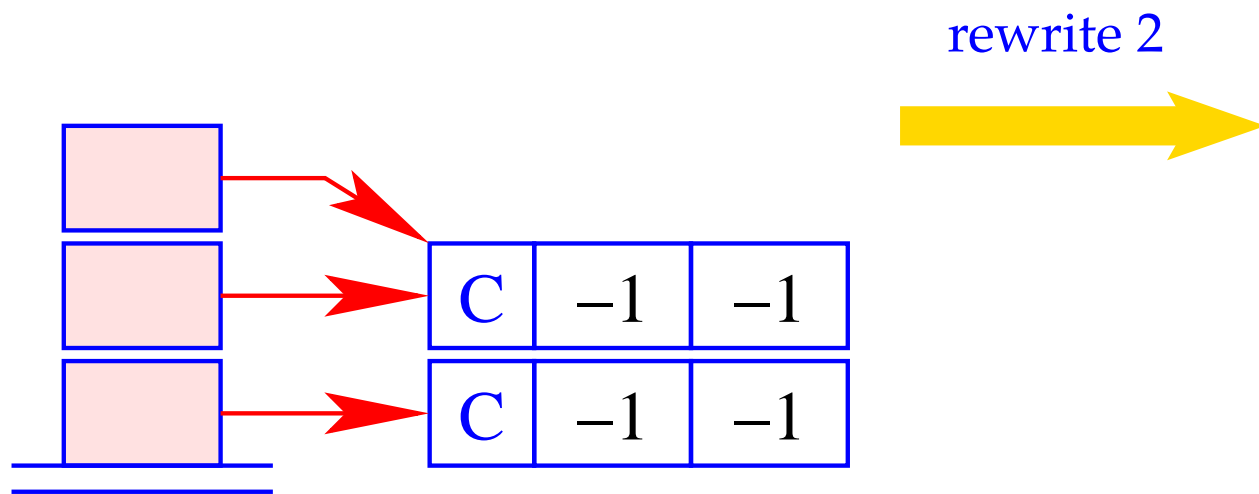
Ausführung dieser Folge sollte den Basiswert 7 liefern ...

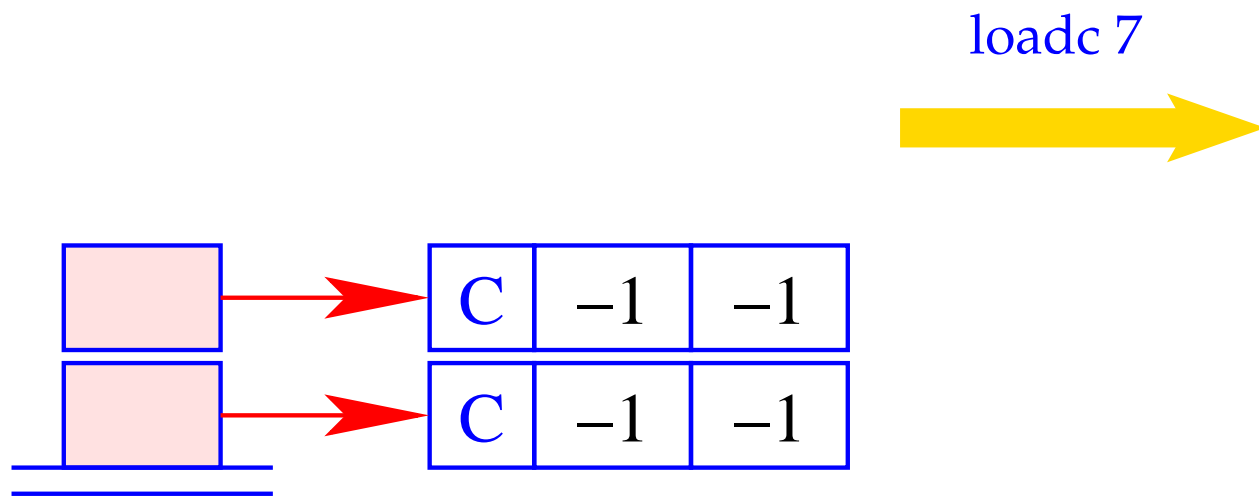


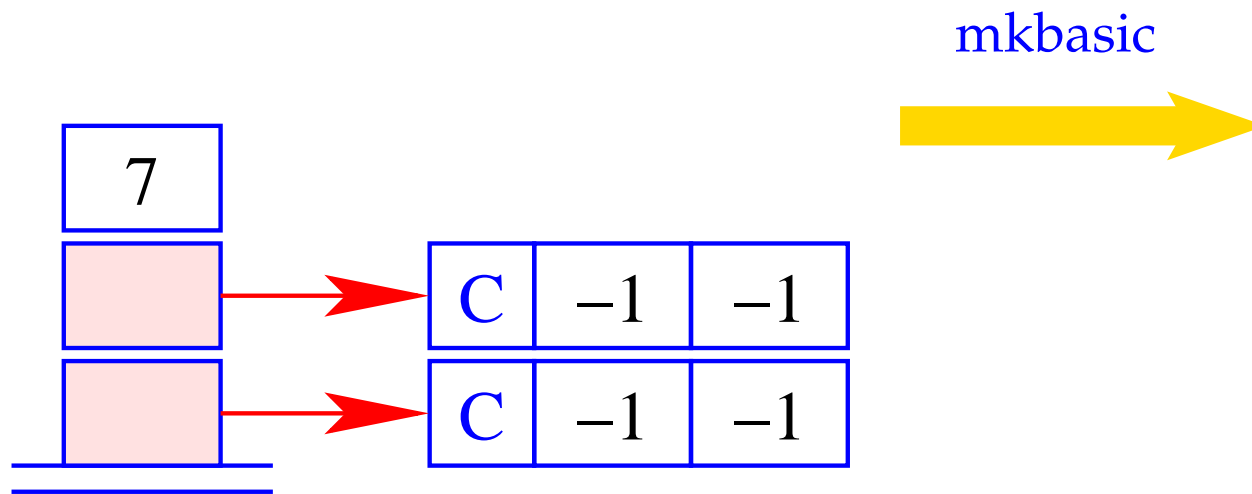
alloc 2

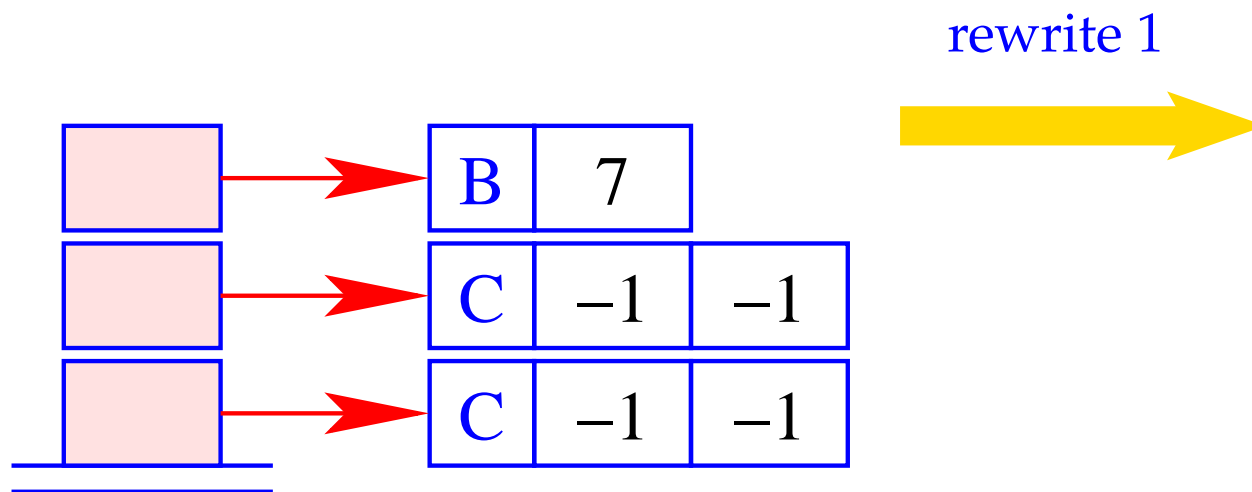


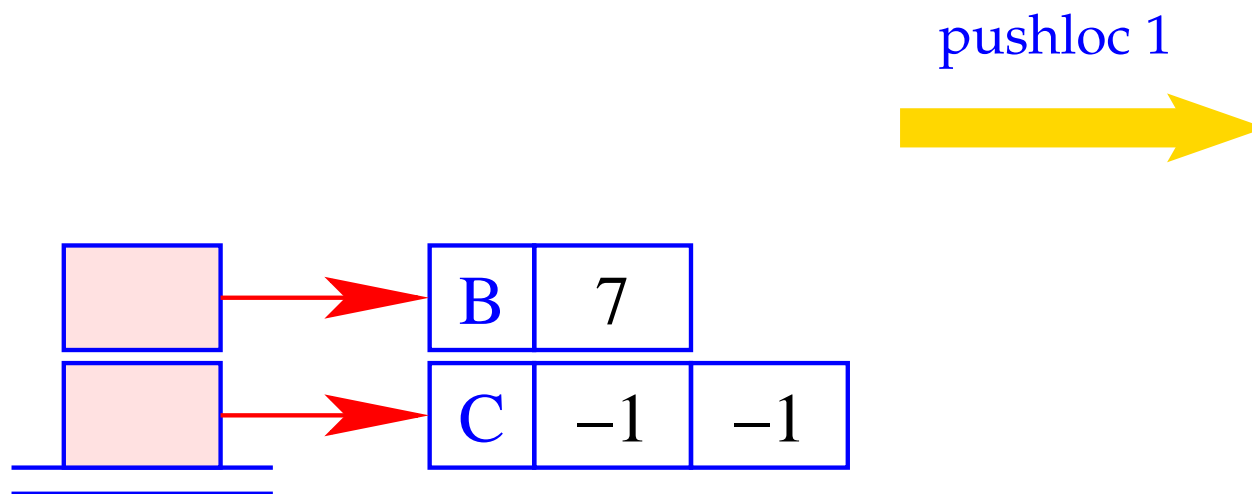


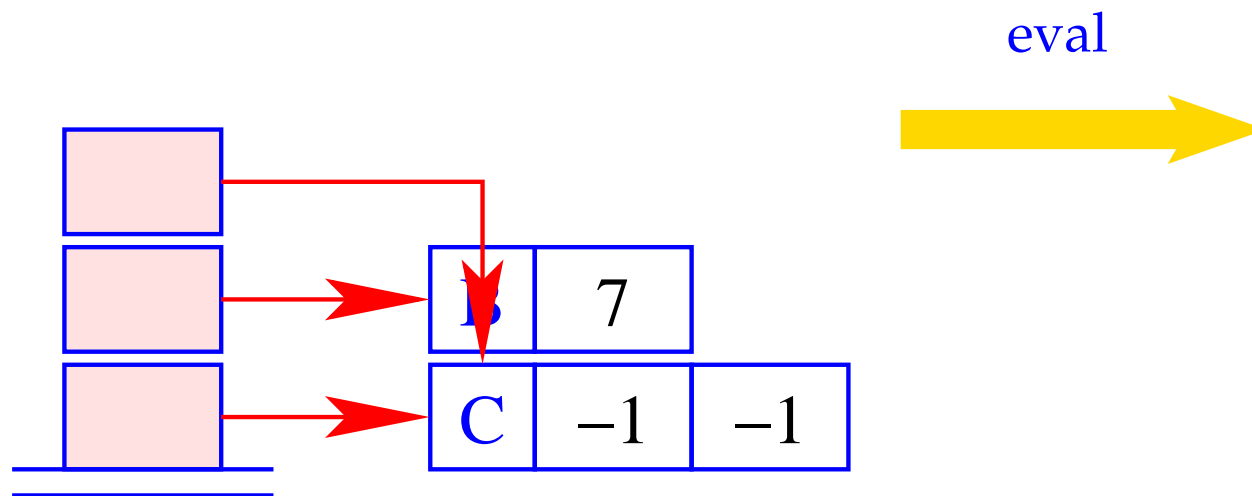












Segmentation Fault !!

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!!



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:

$$\text{code}_C (\text{fn } x_0, \dots, x_{k-1} \Rightarrow e) \rho \text{kp} = \text{code}_V (\text{fn } x_0, \dots, x_{k-1} \Rightarrow e) \rho \text{kp}$$

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

$$PC = 0 \quad SP = FP = GP = -1$$

Der Ausdruck  $e$  sollte **keine freien Variablen** enthalten.

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

$$\text{code } e = \text{code}_V e \ \emptyset \ 0 \\ \text{halt}$$

## 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.



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

**Beispiel:** `let a = 17; f = fn b => a + b in f 42`

Entwerrung 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

## 24 Strukturierte Daten

Im Folgenden wollen wir unsere funktionale Programmiersprache um einfache Datentypen erweitern. Wir beginnen mit der Einführung von **Tupeln**.

### 24.1 Tupel

Tupel werden mithilfe  $k$ -stelliger Konstruktoren ( $k \geq 0$ )  $(., \dots, .)$  aufgebaut und mithilfe der Projektions-Funktionen  $\#j$  ( $j \in \mathbb{N}_0$ ) wieder zerlegt.

Entsprechend erweitern wir die Syntax unserer Programm-Ausdrücke durch:

$$e ::= \dots \mid (e_0, \dots, e_{k-1}) \mid \#j e \\ \quad \mathbf{let} (x_0, \dots, x_{k-1}) = e_1 \mathbf{in} e_0$$

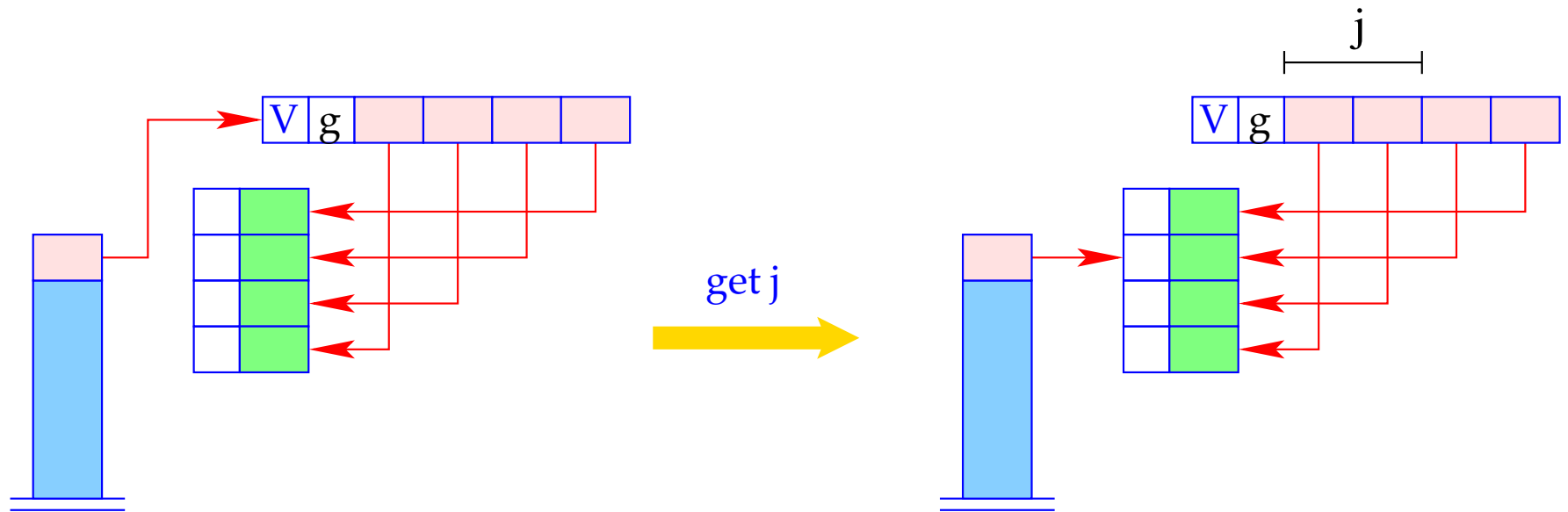
- Tupel werden angelegt, indem erst die Folge der Referenzen auf ihre Komponenten auf dem Keller gesammelt und dann mithilfe der Operation `mkvec` in den Heap gelegt werden.
- Auf Komponenten greifen wir zu, indem wir innerhalb des Tupels einen indizierten Zugriff vornehmen.

$$\begin{aligned}
 \text{code}_V (e_0, \dots, e_{k-1}) \rho \text{kp} &= \text{code}_C e_0 \rho \text{kp} \\
 &\quad \text{code}_C e_1 \rho (\text{kp} + 1) \\
 &\quad \dots \\
 &\quad \text{code}_C e_{k-1} \rho (\text{kp} + k - 1) \\
 &\quad \text{mkvec } k
 \end{aligned}$$

$$\begin{aligned}
 \text{code}_V (\#j e) \rho \text{kp} &= \text{code}_V e \rho \text{kp} \\
 &\quad \text{get } j \\
 &\quad \text{eval}
 \end{aligned}$$

Im Falle von `CBV` können natürlich direkt die Werte der  $e_i$  berechnet werden.

Dabei ist:



```
if (S[SP] == (V,g,v))  
    S[SP] = v[j];  
else Error "Vector expected!";
```

Möchte man nicht auf einzelne, sondern gegebenenfalls alle Komponenten eines Tupels zugreifen, kann man dies mithilfe des Ausdrucks

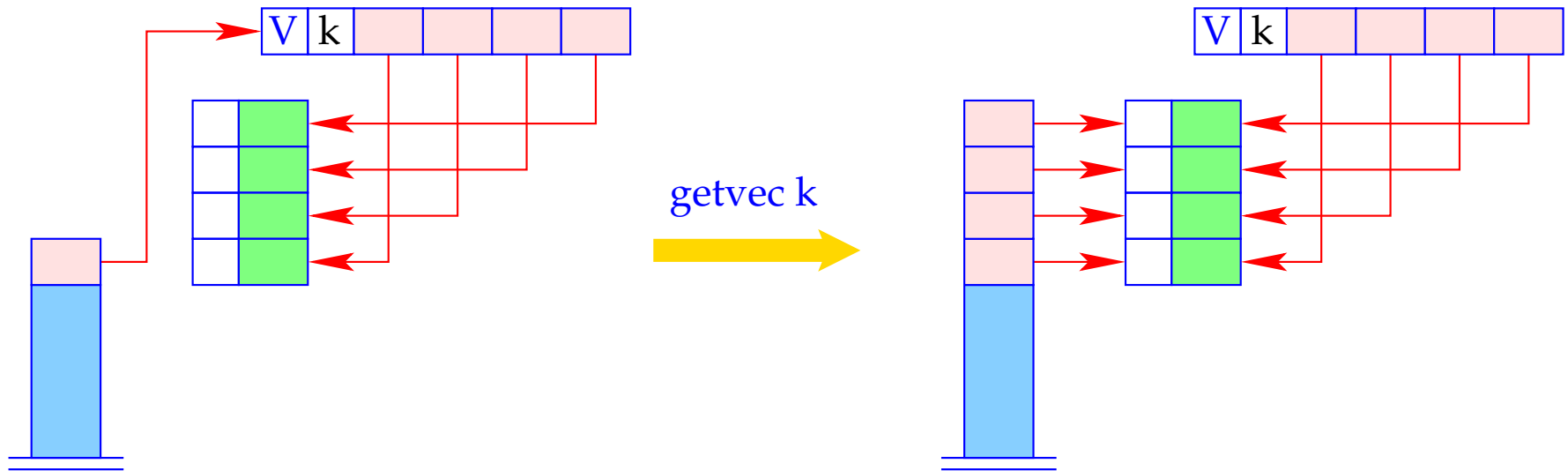
$e \equiv \mathbf{let} (y_0, \dots, y_{k-1}) = e_1 \mathbf{in} e_0$  tun.

Diesen übersetzen wir wie folgt:

$$\begin{aligned} \mathbf{code}_V e \rho \mathbf{kp} &= \mathbf{code}_V e_1 \rho \mathbf{kp} \\ &\quad \mathbf{getvec} \mathbf{k} \\ &\quad \mathbf{code}_V e_0 \rho' (\mathbf{kp} + \mathbf{k}) \\ &\quad \mathbf{slide} \mathbf{k} \end{aligned}$$

wobei  $\rho' = \rho \oplus \{y_i \mapsto \mathbf{kp} + i + 1 \mid i = 0, \dots, k - 1\}$ .

Der Befehl `getvec k` legt die Komponenten eines Vektors der Länge  $k$  auf den Keller:



```

if (S[SP] == (V,k,v)) {
    SP--;
    for(i=0; i<k; i++) {
        SP++; S[SP] = v[i];
    }
} else Error "Vector expected!";

```

## 24.2 Listen

Als Beispiel eines weiteren Datentyps betrachten wir [Listen](#).

Listen werden aus Listen-Elementen mithilfe der Konstante `[]` (“Nil” – die leere Liste) und des rechts-assoziativen Operators `:` (“Cons” – dem Listen-Konstruktor) aufgebaut.

Ein **case**-Ausdruck gestattet den Zugriff auf die Komponenten einer Liste.

**Beispiel:** Die Append-Funktion `app`:

```
app = fn l, y => case l of
      []      -> y
      h : t   -> h : (app t y)
```

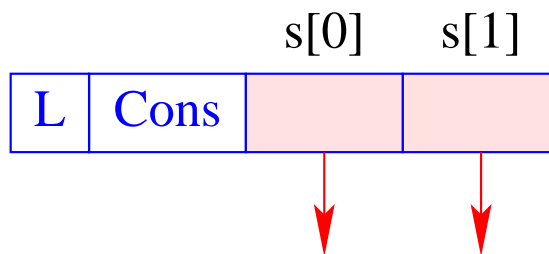
Folglich erweitern wir die Syntax von Ausdrücken  $e$  um:

$$e ::= \dots \mid [] \mid (e_1 : e_2) \\ \mid (\mathbf{case} \ e_0 \ \mathbf{of} \ [] \rightarrow e_1; \ h : t \rightarrow e_2)$$

Neue Halden-Objekte:



leere Liste



nicht-leere Liste

## 24.3 Der Aufbau von Listen

Für das Anlegen von Listen-Knoten führen wir die Befehle `nil` und `cons` ein.

Damit erhalten wir für **CBN**:

$$\begin{aligned}\text{code}_V [] \rho \text{kp} &= \text{nil} \\ \text{code}_V (e_1 : e_2) \rho \text{kp} &= \text{code}_C e_1 \rho \text{kp} \\ &\quad \text{code}_C e_2 \rho (\text{kp} + 1) \\ &\quad \text{cons}\end{aligned}$$

**Beachte:**

- Bei **CBN** werden für die Argumente von “:” Abschlüsse angelegt.
- Bei **CBV** müssen sie dagegen erst ausgewertet werden.