

Die Übersetzung funktionaler Programmiersprachen

11 Die Sprache PuF

Wir betrachten hier nur die Mini-Sprache **PuF** (“Pure Functions”). Insbesondere verzichten wir (vorerst) auf:

- Seiteneffekte;
- Datenstrukturen;

Ein Programm ist ein Ausdruck e der Form:

$$\begin{aligned}
e ::= & \quad b \mid x \mid (\square_1 e) \mid (e_1 \square_2 e_2) \\
& \mid (\mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_3) \\
& \mid (e' e_0 \dots e_{k-1}) \mid (\mathbf{fun} \ x_0, \dots, x_{k-1} \rightarrow e) \\
& \mid (\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_0) \\
& \mid (\mathbf{let rec} \ x_1 = e_1 \ \mathbf{and} \dots \ \mathbf{and} \ x_n = e_n \ \mathbf{in} \ e_0)
\end{aligned}$$

Ein Ausdruck ist somit:

- ein Basiswert, eine Variable, eine Operator-Anwendung oder ein bedingter Ausdruck;
- eine **Funktions-Anwendung**;
- eine Funktion – d.h. aus einem Funktionsrumpf entstanden mithilfe von **Abstraktion** der formalen Parameter;
- ein **let**-Ausdruck, der **lokal Variablen-Definitionen** einführt, oder
- ein **let-rec**-Ausdruck, der lokal **rekursive** Variablen-Definitionen einführt.

Als Basis-Typen erlauben wir der Einfachheit halber nur **int**.

Beispiel:

Die folgende allseits bekannte Funktion berechnet die Fakultät:

```
fac    =   fun x → if x ≤ 1 then 1  
           else x · fac (x - 1)
```

Wie üblich, setzen wir nur da Klammern, wo sie zum Verständnis erforderlich sind :-)

Achtung:

Wir unterscheiden zwei Arten der Parameter-Übergabe:

CBV: Call-by-Value – die aktuellen Parameter werden ausgewertet bevor der Rumpf der Funktion ausgewertet wird (genau wie bei C ...);

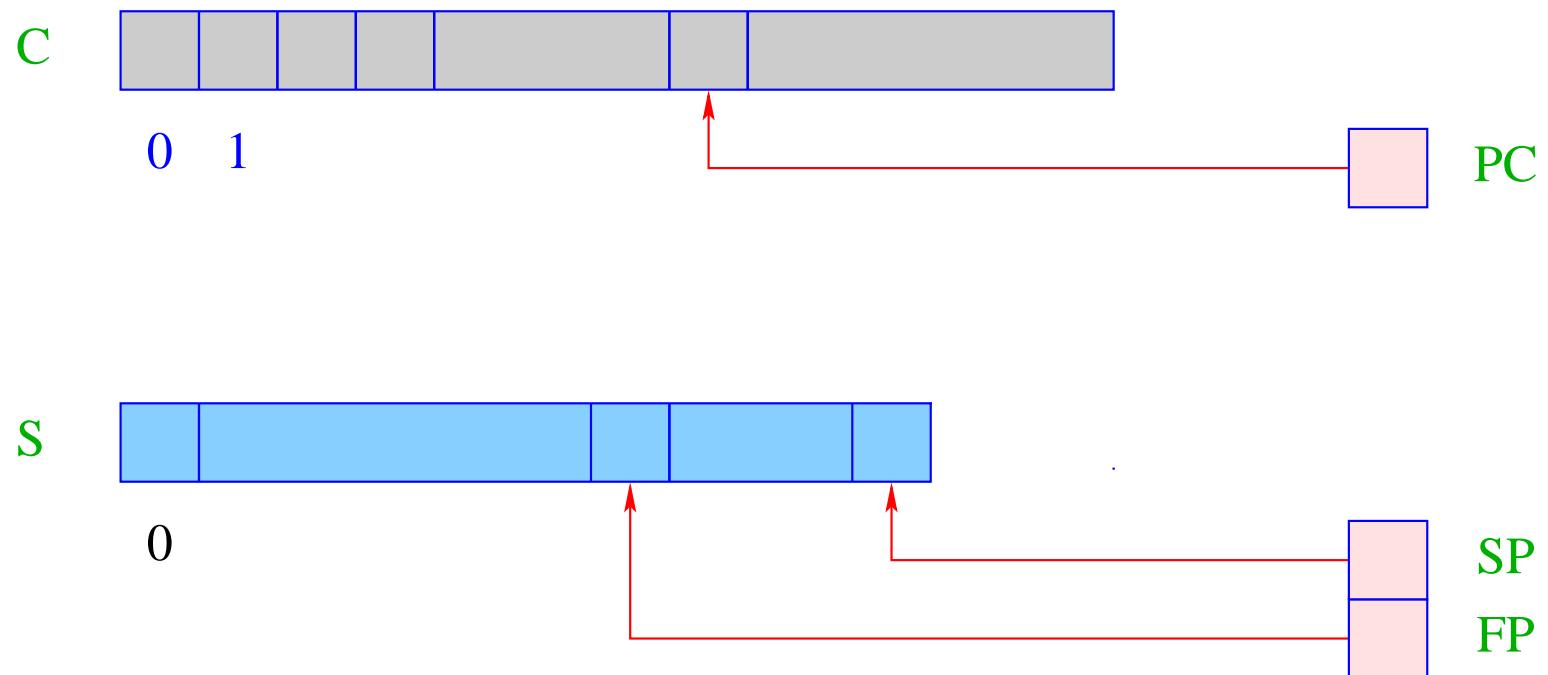
CBN: Call-by-Need – die aktuellen Parameter werden erst ausgewertet, wenn ihr Wert benötigt wird \implies spart manchmal Arbeit :-)

Beispiel:

```
let rec fac = ...
in let foo = fun x, y → x
in foo 1 (fac 1000)
```

- Die Funktion `foo` greift nur auf ihr erstes Argument zu.
- Die Auswertung des zweiten Arguments wird bei **CBN** vermieden `:)`
- Weil wir bei **CBN** nicht sicher sein können, ob der Wert einer Variablen bereits ermittelt wurde oder nicht, müssen wir `vor` jedem Variablen-Zugriff überprüfen, ob der Wert bereits vorliegt `:-(|`
- Liegt der Wert noch nicht vor, muss seine Berechnung angestoßen werden.

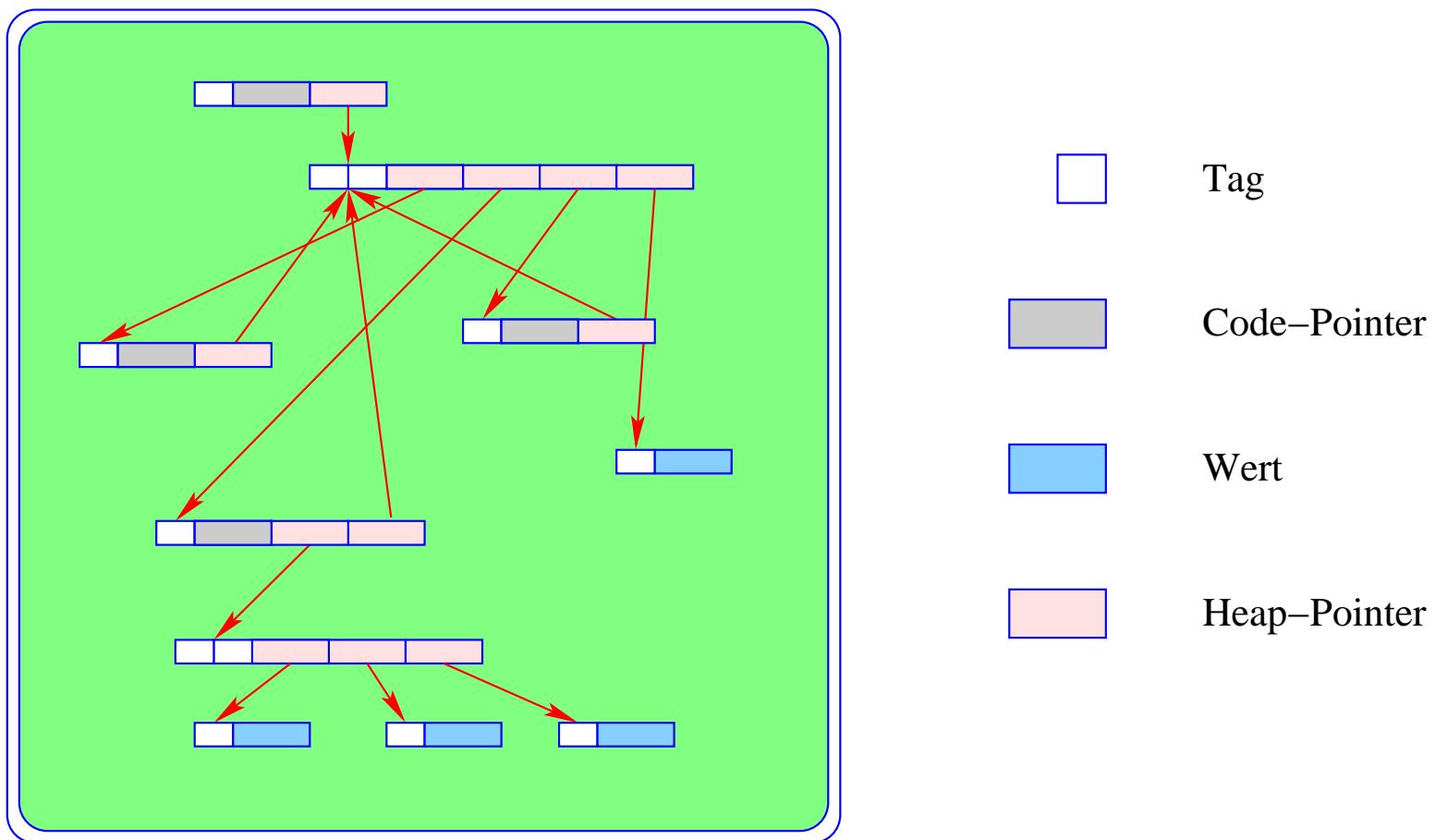
12 Architektur der MaMa:



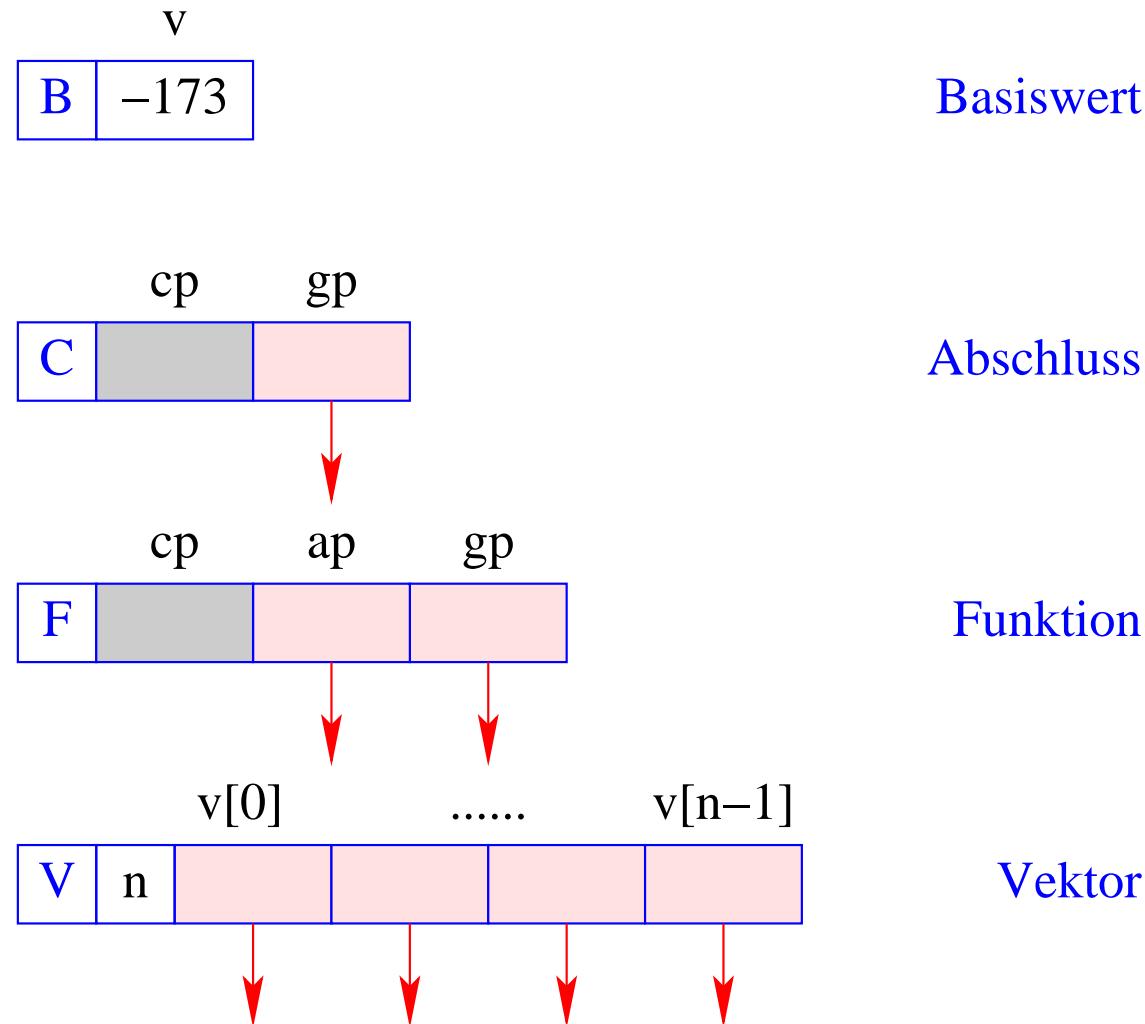
... das sind die uns bereits bekannten Datenstrukturen:

- C = Code-Speicher – enthält MaMa-Programm;
jede Zelle enthält einen Befehl;
- PC = Program Counter – zeigt auf nächsten auszuführenden Befehl;
- S = Runtime-Stack;
jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
- SP = Stack-Pointer – zeigt auf oberste belegte Zelle;
- FP = Frame-Pointer – zeigt auf den aktuellen Kellerrahmen.

Weiterhin benötigen wir eine Halde **H**:



... die wir nun als einen **abstrakten Datentyp** auffassen, in dem wir Daten-Objekte der folgenden Form ablegen können:



Die Funktion `new (tag, args)` des Laufzeit-Systems der MaMa erzeugt ein entsprechendes Objekt in \mathbf{H} und liefert eine Referenz darauf zurück.

Im Folgenden unterscheiden wir drei Arten von Code für einen Ausdruck e :

- $\text{code}_V e$ — berechnet den Wert von e , legt ihn in der Halde an und liefert auf dem Keller eine Referenz darauf zurück (der Normal-Fall);
- $\text{code}_B e$ — berechnet den Wert von e , und liefert ihn direkt oben auf dem Keller zurück (geht nur für Basistypen);
- $\text{code}_C e$ — wertet den Ausdruck e nicht aus, sondern legt einen Abschluss für e in der Halde an und liefert auf dem Stack eine Referenz auf diesen Abschluss zurück \Longrightarrow benötigen wir zur Implementierung von CBN.

Wir betrachten zuerst Übersetzungsschemata für die ersten beiden Code-Arten.

13 Einfache Ausdrücke

Ausdrücke, die nur Konstanten, Operator-Anwendungen und bedingte Verzweigungen enthalten, werden wie Ausdrücke in imperativen Sprachen übersetzt:

$$\text{code}_B b \rho \text{ kp} = \text{loadc } b$$

$$\text{code}_B (\square_1 e) \rho \text{ kp} = \text{code}_B e \rho \text{ kp}$$

op₁

$$\text{code}_B (e_1 \square_2 e_2) \rho \text{ kp} = \text{code}_B e_1 \rho \text{ kp}$$

$$\text{code}_B e_2 \rho (\text{kp} + 1)$$

op₂

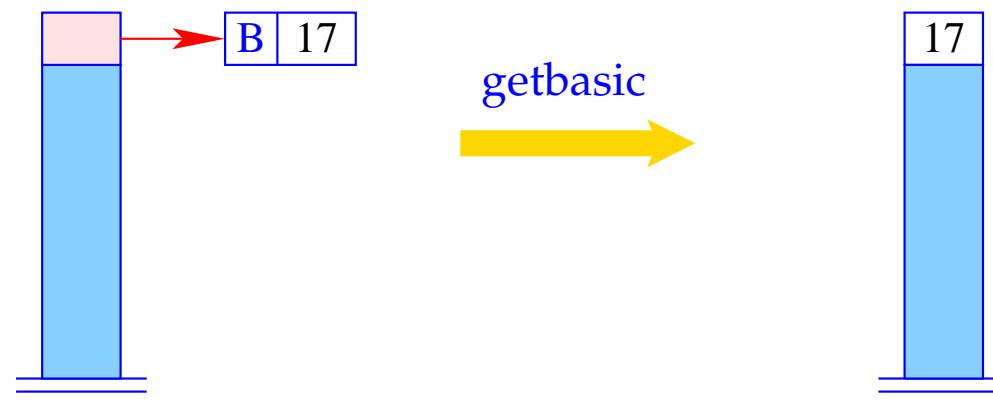
$$\text{code}_B (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \text{ kp} = \begin{array}{l} \text{code}_B e_0 \rho \text{ kp} \\ \text{jumpz A} \\ \text{code}_B e_1 \rho \text{ kp} \\ \text{jump B} \\ \text{A: code}_B e_2 \rho \text{ kp} \\ \text{B: ...} \end{array}$$

Bemerkungen:

- ρ bezeichnet die aktuelle Adress-Umgebung, in der der Ausdruck übersetzt wird.
- Das Extra-Argument **kp** zählt die Länge des lokalen Kellers mit \Rightarrow benötigen wir später zur Adressierung der Variablen.
- Die Instruktionen **op₁** und **op₂** implementieren die Operatoren \square_1 und \square_2 , so wie in der **CMa** die Operatoren **neg** und **add** die Negation bzw. die Addition implementieren.
- Für alle übrigen Ausdrücke berechnen wir erst den Wert im Heap und dereferenzieren dann:

$$\text{code}_B e \rho \text{ kp} = \begin{array}{l} \text{code}_V e \rho \text{ kp} \\ \text{getbasic} \end{array}$$

Dabei ist:



```
if (H[S[SP]] != (B,_))  
    Error "not basic!";  
else  
    S[SP] = H[S[SP]].v;
```

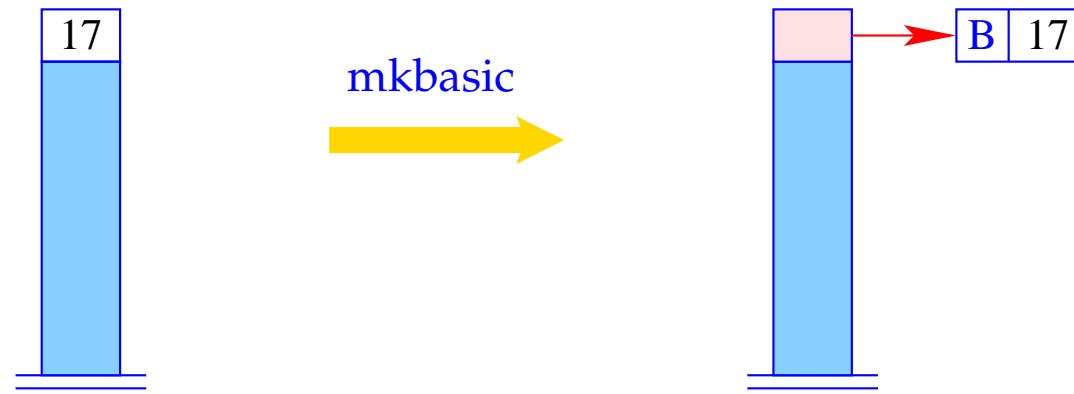
Für code_V und einfache Ausdrücke finden wir analog:

$$\text{code}_V b \rho \text{ kp} = \text{loadc } b; \text{ mkbasic}$$

$$\begin{aligned} \text{code}_V (\square_1 e) \rho \text{ kp} &= \text{code}_B e \rho \text{ kp} \\ &\quad \text{op}_1; \text{ mkbasic} \end{aligned}$$

$$\begin{aligned} \text{code}_V (e_1 \square_2 e_2) \rho \text{ kp} &= \text{code}_B e_1 \rho \text{ kp} \\ &\quad \text{code}_B e_2 \rho (kp + 1) \\ &\quad \text{op}_2; \text{ mkbasic} \end{aligned}$$

$$\begin{aligned} \text{code}_V (\mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2) \rho \text{ kp} &= \text{code}_B e_0 \rho \text{ kp} \\ &\quad \text{jumpz A} \\ &\quad \text{code}_V e_1 \rho \text{ kp} \\ &\quad \text{jump B} \\ &\quad \text{A: code}_V e_2 \rho \text{ kp} \\ &\quad \text{B: ...} \end{aligned}$$



$S[SP] = \text{new } (B, S[SP]);$

14 Der Zugriff auf Variablen

Beispiel: Betrachte die Funktion f :

```
fun a → let b = a * a  
          in b + c
```

Die Funktion f benutzt die **globale** Variable c sowie die **lokalen** Variablen a (als formalem Parameter) und b (eingeführt durch **let**).

Der Wert einer globalen Variable wird beim **Anlegen** der Funktion bestimmt (**Statische Bindung!**) und später nur nachgeschlagen.

Idee:

- Die Bindungen der globalen Variablen verwalten wir in einem Vektor im Heap ([Global Vector](#)).
- Beim Anlegen eines F-Objekts wird der Global Vector für die Funktion ermittelt und in der gp-Komponente abgelegt.
- Bei der Auswertung eines Ausdrucks zeigt das ([neue](#)) Register **GP** ([Global Pointer](#)) auf den aktuellen Global Vector.
- Die lokalen Variablen verwalten wir dagegen auf dem Keller.

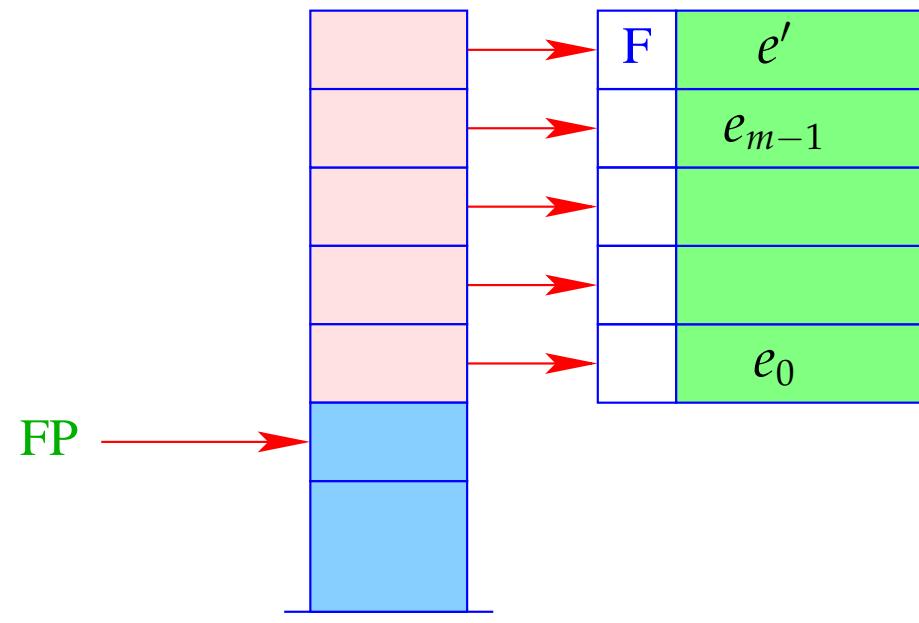
Adress-Umgebungen haben darum die Form:

$$\rho : \text{Vars} \rightarrow \{L, G\} \times \mathbb{Z}$$

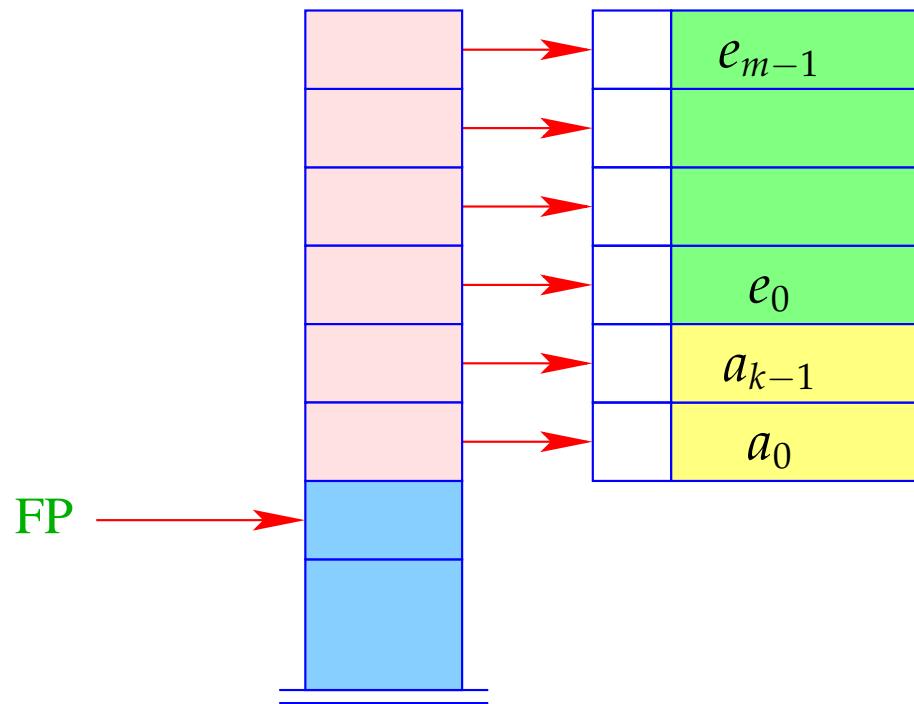
- Die globalen Variablen numerieren wir einfach geeignet durch.
- Für die Adressierung der lokalen Variablen gibt es zwei Möglichkeiten.

Sei $e \equiv e' e_0 \dots e_{m-1}$ die Anwendung einer Funktion e' auf Argumente e_0, \dots, e_{m-1} .

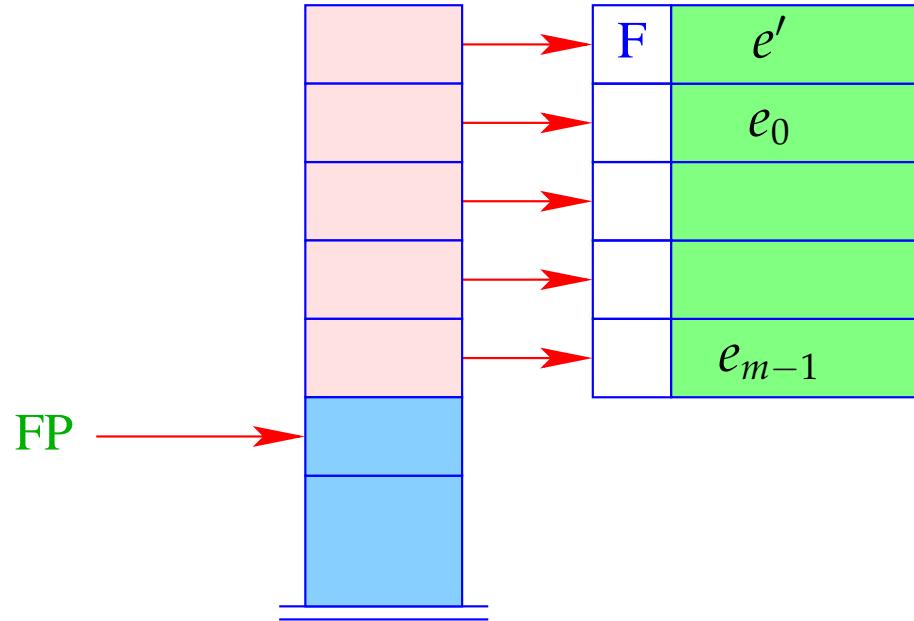
Mögliche Kellerorganisation:



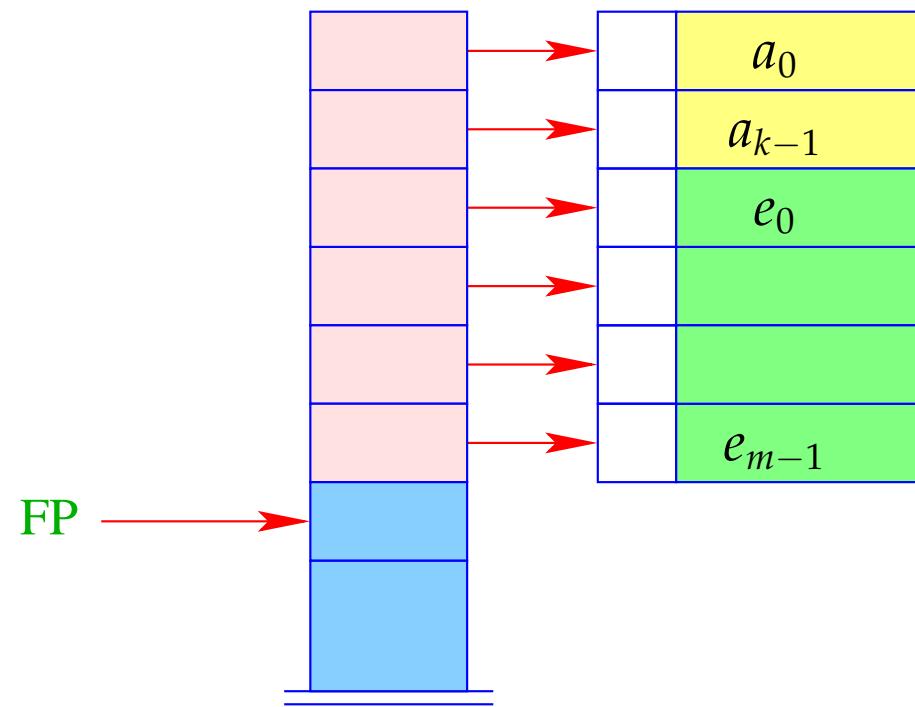
- + Adressierung der Parameter kann relativ zu FP erfolgen :-)
- Stellt sich heraus, dass sich e' zu einer Funktion evaluiert, die bereits partiell auf aktuelle Parameter a_0, \dots, a_{k-1} angewendet ist, müssen diese unterhalb von e_0 in den Keller hinein gefrickelt werden :-(



Alternative:



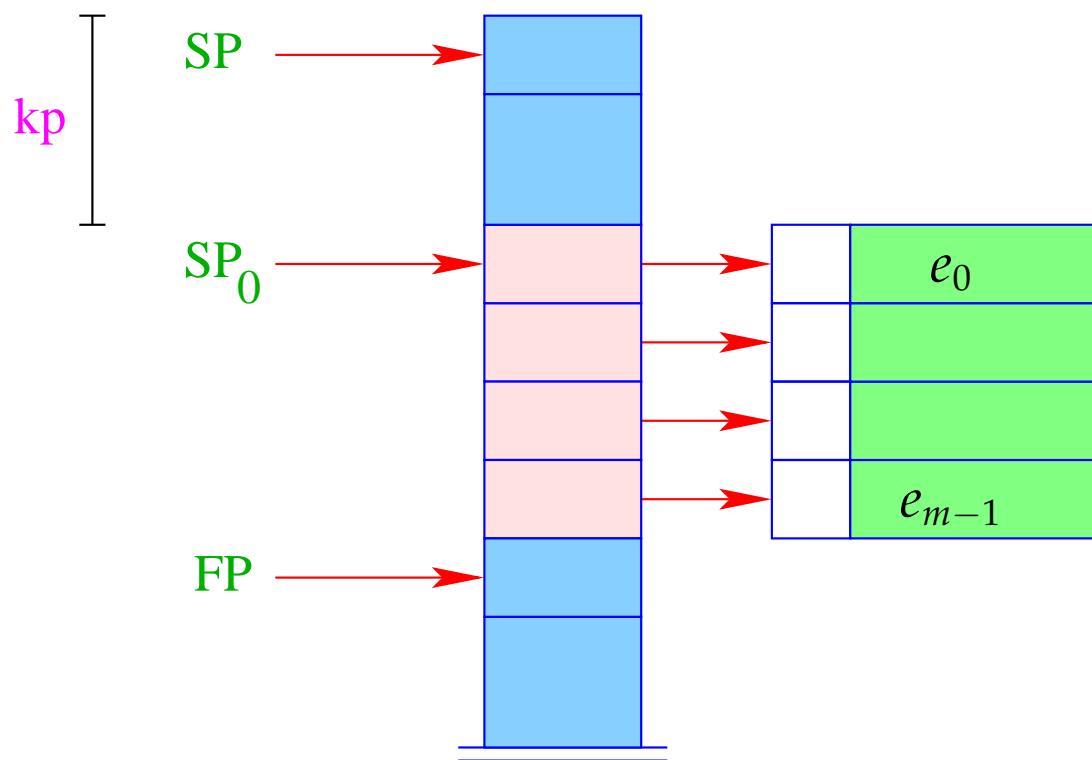
- + Die weiteren Argumente a_0, \dots, a_{k-1} wie auch die lokalen Variablen können einfach oben auf den Keller gelegt werden :-)



- Adressierung relativ zu **FP** ist aber leider nicht mehr möglich ... ;-?

Ausweg:

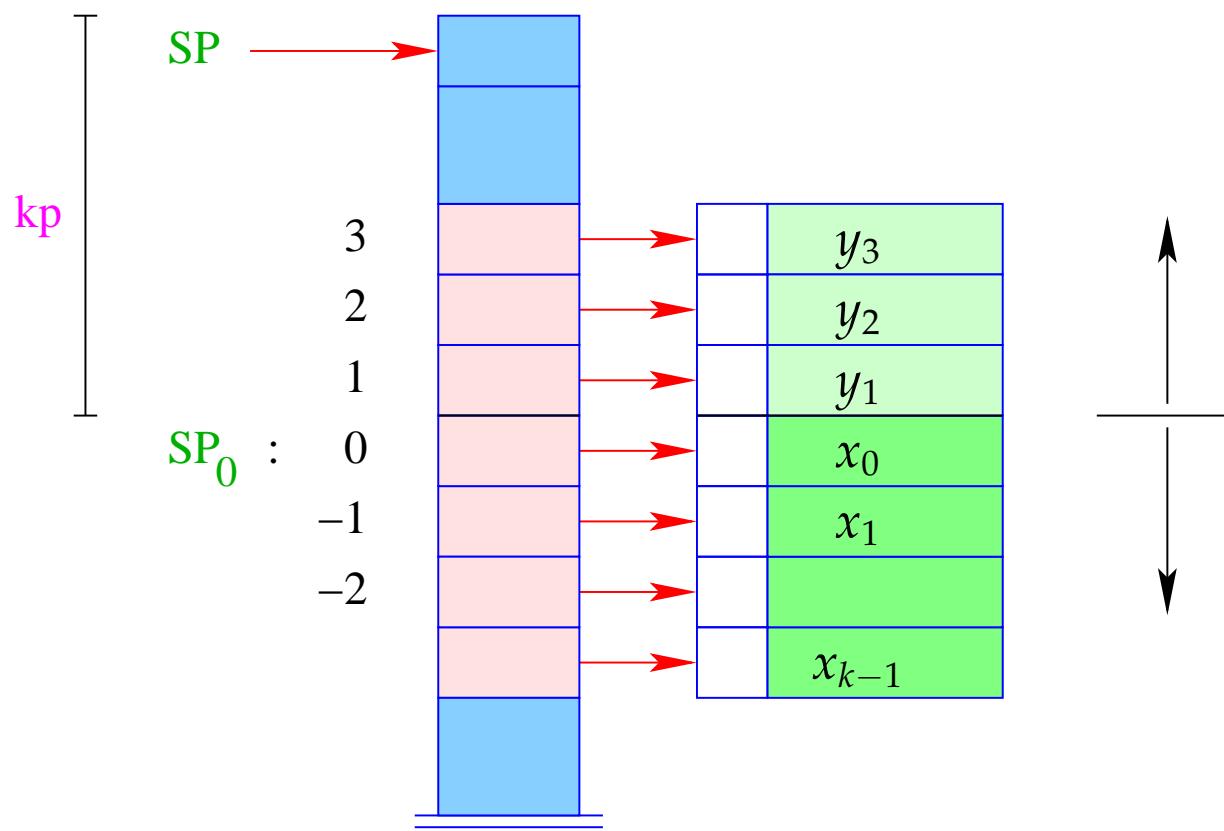
- Wir adressieren relativ zum Stackpointer SP !!!
- Leider ändert sich der Stackpointer während der Programm-Ausführung ...



- Die Abweichung des **SP** von seiner Position **SP₀** 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, \dots vergeben wir sukzessive die **nicht-positiven** Relativ-Adressen $0, -1, -2, \dots$, d.h. $\rho x_i = (L, -i)$.
- Die **absolute** Adresse des i -ten formalen Parameters ergibt sich dann als

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

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



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

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

Bei **CBN** erzeugen wir damit für einen Variablen-Zugriff:

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

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

Bei **CBV** können wir **eval** einfach streichen.

Das Macro **getvar** ist definiert durch:

$$\text{getvar } x \rho \text{ kp} = \begin{aligned} & \text{let } (t, i) = \rho x \text{ in} \\ & \text{match } t \text{ with} \\ & \quad L \rightarrow \text{pushloc } (\text{kp} - i) \\ & \quad G \rightarrow \text{pushglob } i \\ & \text{end} \end{aligned}$$