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:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \\
 & \mid & (\textbf{if}\ \ e_0\ \textbf{then}\ e_1\ \textbf{else}\ \ e_3) \\
 & \mid & (e'\ e_0 \ldots e_{k-1}) \mid (\textbf{fn}\ \ x_0, \ldots, x_{k-1} \Rightarrow e) \\
 & \mid & (\textbf{let}\ \ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ \ e_0) \\
 & \mid & (\textbf{letrec}\ \ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ \ e_0)
\end{array}
$$

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 **letrec**-Ausdruck, der lokal rekursive Variablen-Definitionen einführt.

Als Basis-Typen erlauben wir der Einfachkeit halber nur `int`.

Beispiel:

Die folgende allseits bekannte Funktion berechnet die Fakultät:

$$\begin{aligned} \text{fac} \quad = \quad & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & \textbf{else}\ x \cdot \text{fac}\ (x-1) \end{aligned}$$

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 ...);

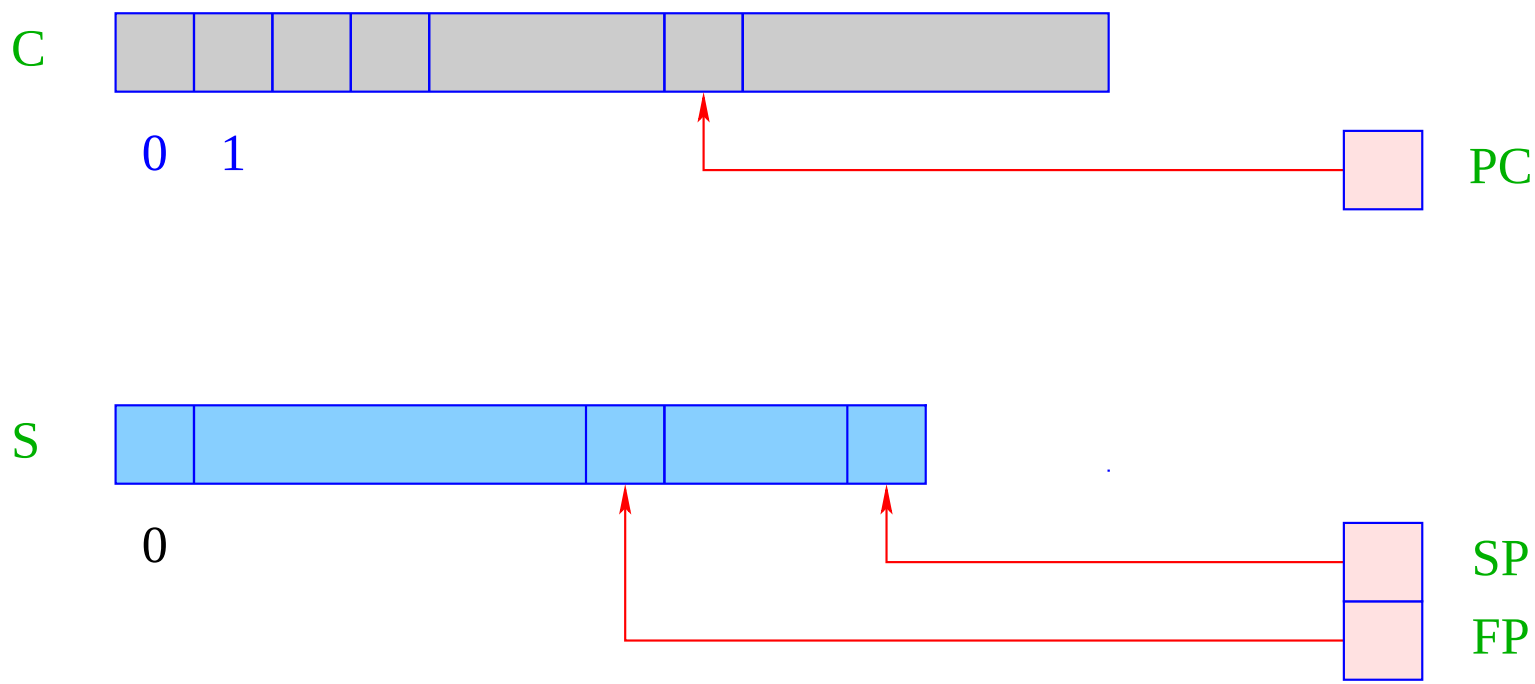
CBN: Call-by-Need – die aktuellen Parameter werden erst ausgewertet, wenn ihr Wert benötigt wird $\Longrightarrow$ spart manchmal Arbeit :-)

Beispiel:

$$
\begin{array}{llll}
\textbf{let} & \text{fac} & = & \ldots\,; \\
 & \text{foo} & = & \textbf{fn}\ x, y \Rightarrow x \\
\textbf{in} & \text{foo}\ 1\ (\text{fac}\ 1000) & &
\end{array}
$$

- 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.

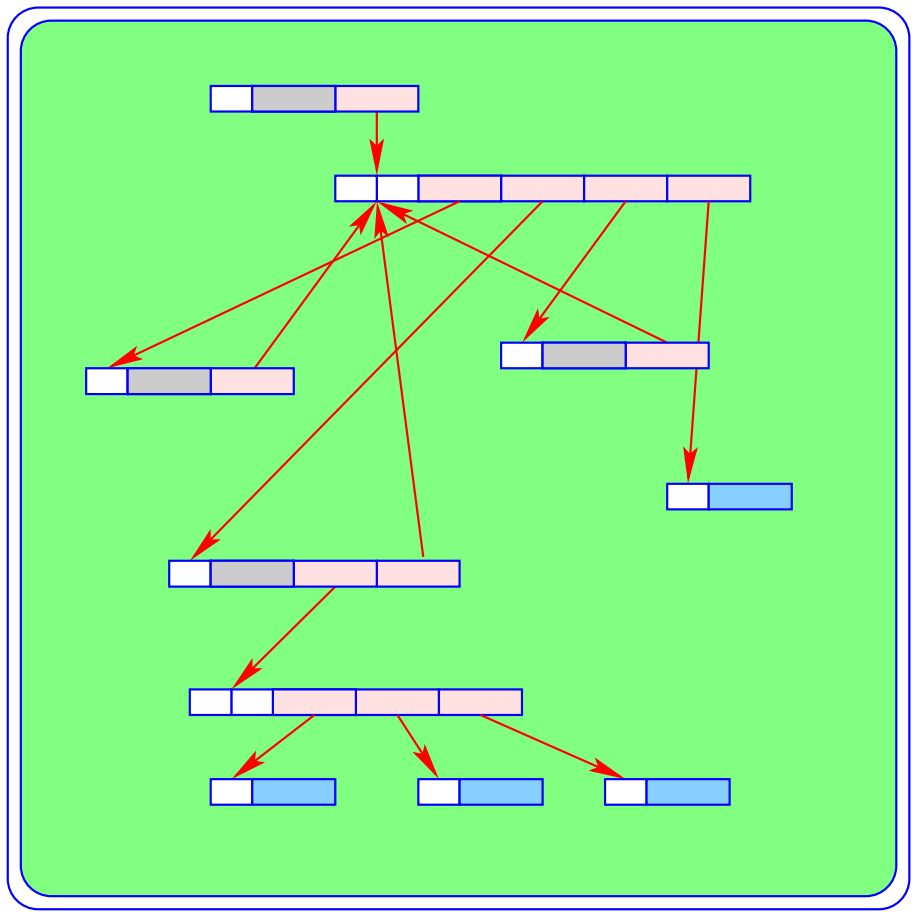
12 Architektur der MaMa:

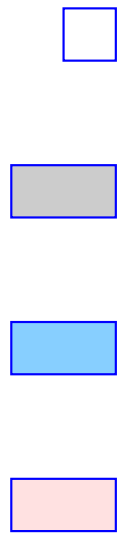


... das sind die uns bereits bekannten Datenstrukturen:

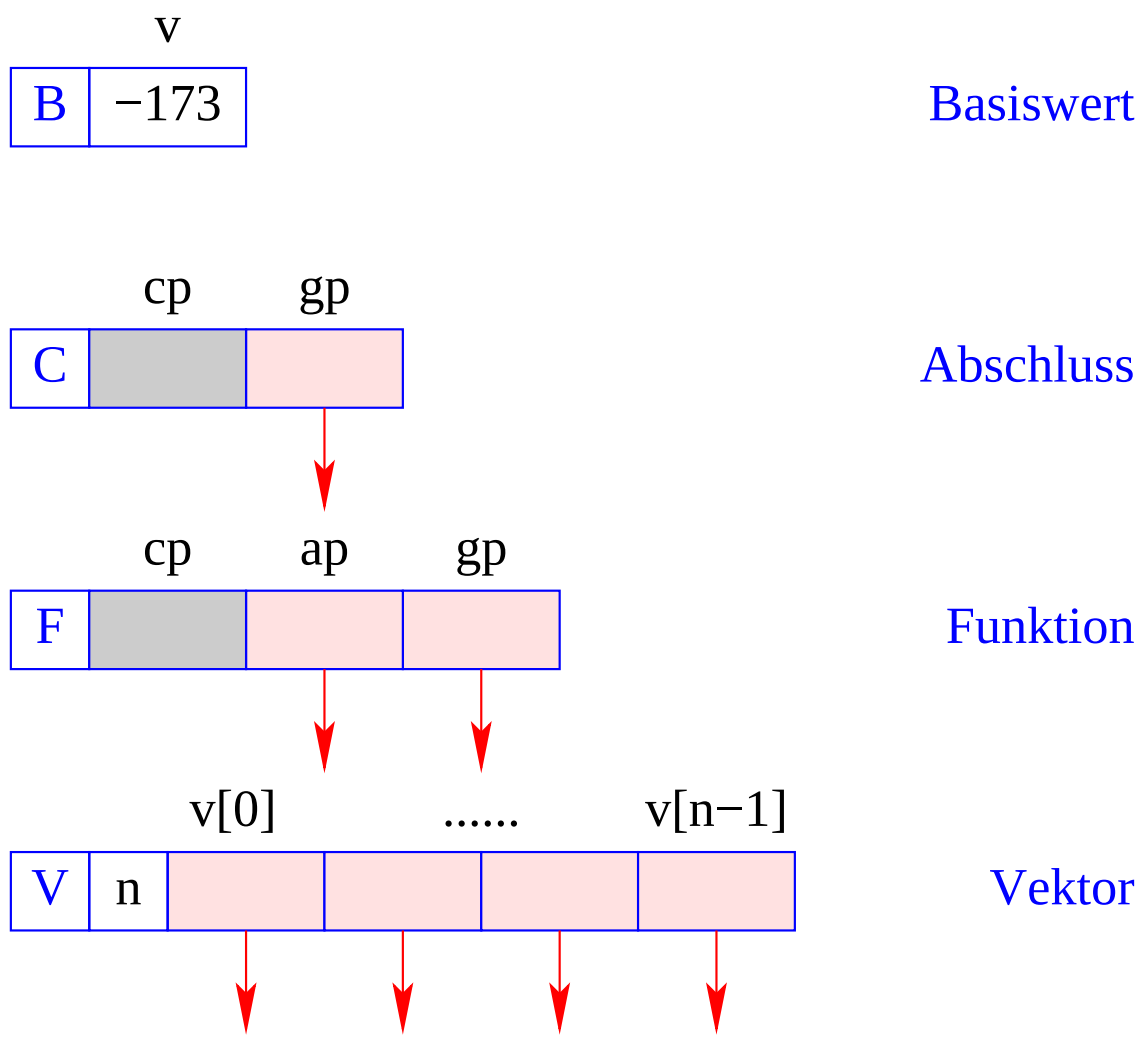
C	=	Code-Speicher – enthält MaMa-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter – zeigt auf nächsten auszuführenden Befehl;
S	=	Runtime-Stack;
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
SP	=	Stack-Pointer – zeigt auf oberste belegte Zelle;
FP	=	Frame-Pointer –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

$$
\begin{array}{lcl}
\mathrm{code}_B\ b\ \rho\ \mathrm{kp} & = & \mathrm{loadc\ b} \\
\mathrm{code}_B\ (\Box_1\ e)\ \rho\ \mathrm{kp} & = & \mathrm{code}_B\ e\ \rho\ \mathrm{kp} \\
 & & \mathrm{op}_1 \\
\mathrm{code}_B\ (e_1\ \Box_2\ e_2)\ \rho\ \mathrm{kp} & = & \mathrm{code}_B\ e_1\ \rho\ \mathrm{kp} \\
 & & \mathrm{code}_B\ e_2\ \rho\ (\mathrm{kp}+1) \\
 & & \mathrm{op}_2
\end{array}
$$

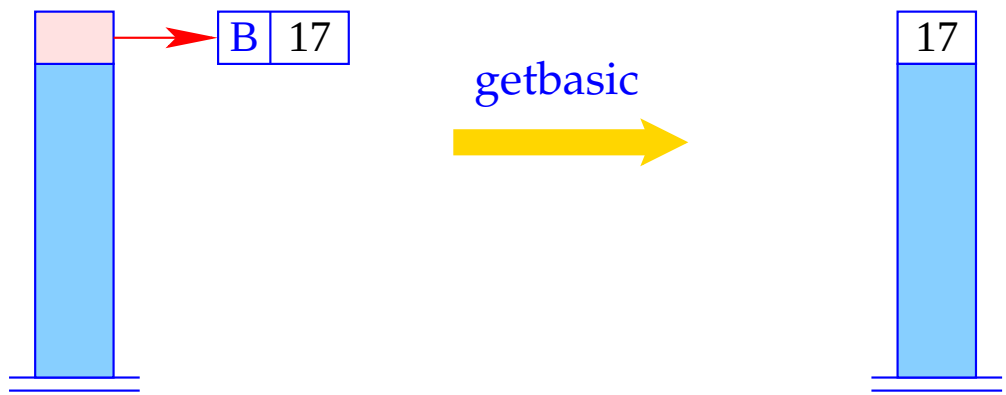
$$
\begin{array}{lcll}
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{kp} & = & & \text{code}_B\ e_0\ \rho\ \text{kp} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_B\ e_1\ \rho\ \text{kp} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{kp}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

Bemerkungen:

- ρ bezeichnet die aktuelle Adress-Umgebung, in der der Ausdruck übersetzt wird.
- Das Extra-Argument kp zählt die Länge des lokalen Kellers mit $\Longrightarrow$ benötigen wir später zur Adressierung der Variablen.
- Die Instruktionen op_1 und op_2 implementieren die Operatoren $\Box_1$ und $\Box_2$, 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:

$$\begin{array}{lcl} \mathrm{code}_B\, e\, \rho\, \mathrm{kp} & = & \mathrm{code}_V\, e\, \rho\, \mathrm{kp} \\ & & \mathrm{getbasic} \end{array}$$

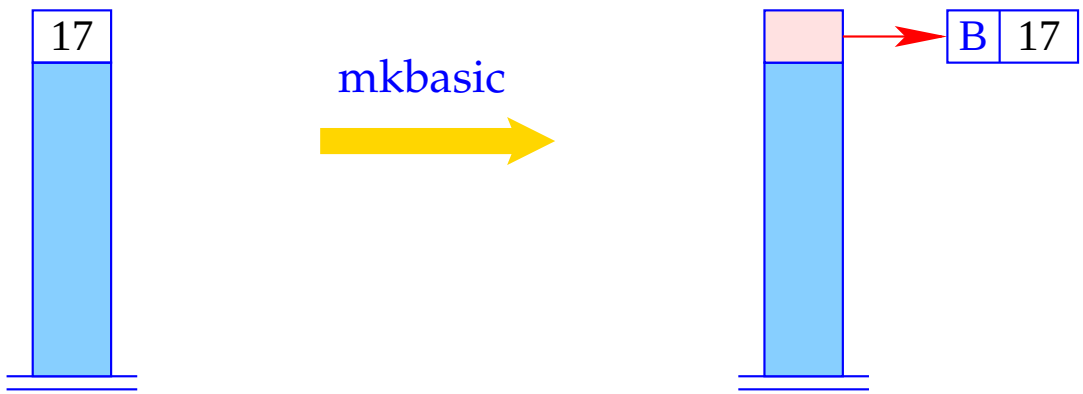
Dabei ist:




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

Für code_V und einfache Ausdrücke finden wir analog:

$$
\begin{array}{lcl}
\text{code}_V\ b\ \rho\ \text{kp} & = & \text{loadc b; mkbasic} \\
\text{code}_V\ (\Box_1\ e)\ \rho\ \text{kp} & = & \text{code}_B\ e\ \rho\ \text{kp} \\
 & & \text{op}_1\text{; mkbasic} \\
\text{code}_V\ (e_1\ \Box_2\ e_2)\ \rho\ \text{kp} & = & \text{code}_B\ e_1\ \rho\ \text{kp} \\
 & & \text{code}_B\ e_2\ \rho\ (\text{kp} + 1) \\
 & & \text{op}_2\text{; mkbasic} \\
\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{kp} & = & \text{code}_B\ e_0\ \rho\ \text{kp} \\
 & & \text{jumpz A} \\
 & & \text{code}_V\ e_1\ \rho\ \text{kp} \\
 & & \text{jump B} \\
 & \text{A:} & \text{code}_V\ e_2\ \rho\ \text{kp} \\
 & \text{B:} & \ldots
\end{array}
$$



S[SP] = new (B,S[SP]);

14 Der Zugriff auf Variablen

Beispiel: Betrachte die Funktion f :

$$\begin{aligned} \textbf{fn}\ a \quad \Rightarrow \quad & \textbf{let}\ b = a * a \\ & \textbf{in}\ b + c \end{aligned}$$

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.

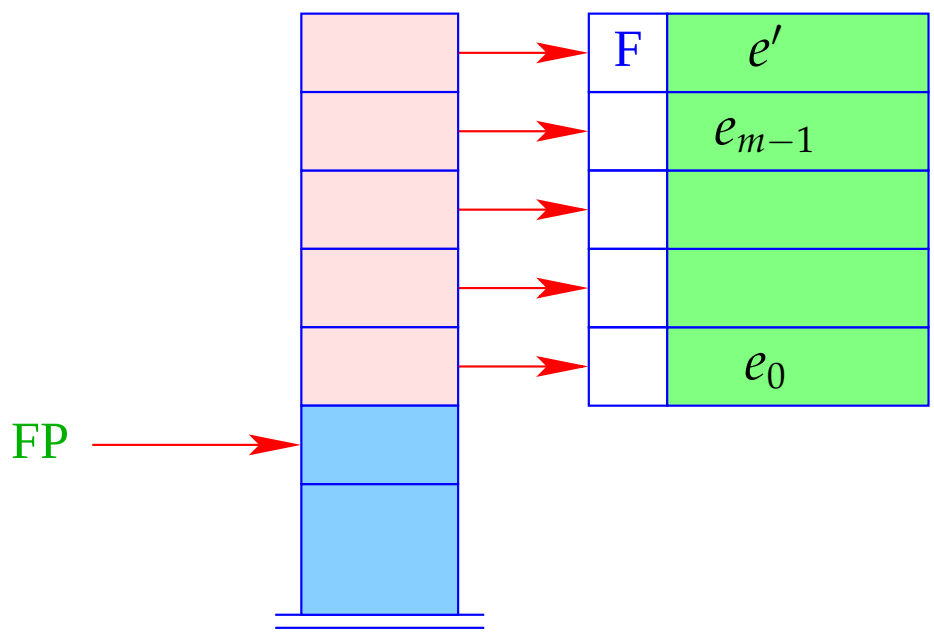
Adress-Umgebungen haben darum die Form:

$$\rho : \mathit{Vars} \to \{L, G\} \times \mathbb{Z}$$

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

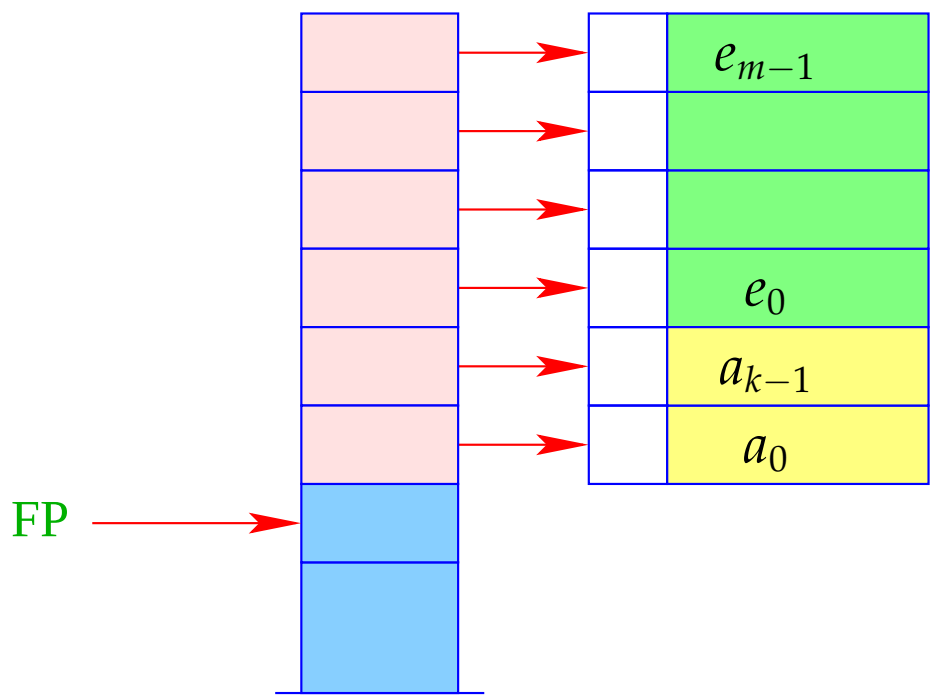
Sei $e \equiv e'\ e_0\ \ldots\ e_{m-1}$ die Anwendung einer Funktion e' auf Argumente $e_0, \ldots, e_{m-1}$.

Mögliche Kellerorganisation:

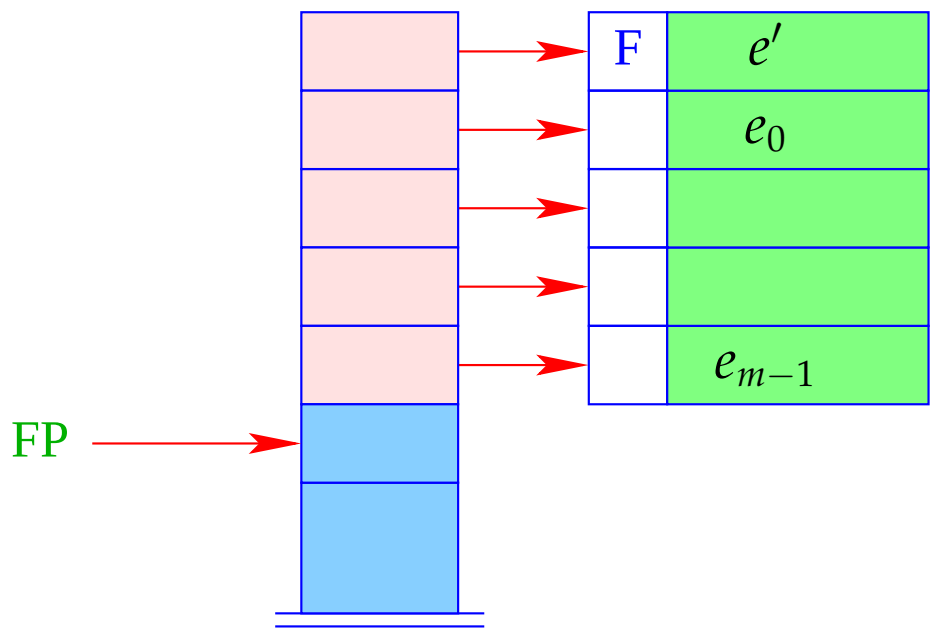


+ Adressierung der Parameter kann relativ zu FP erfolgen :-)

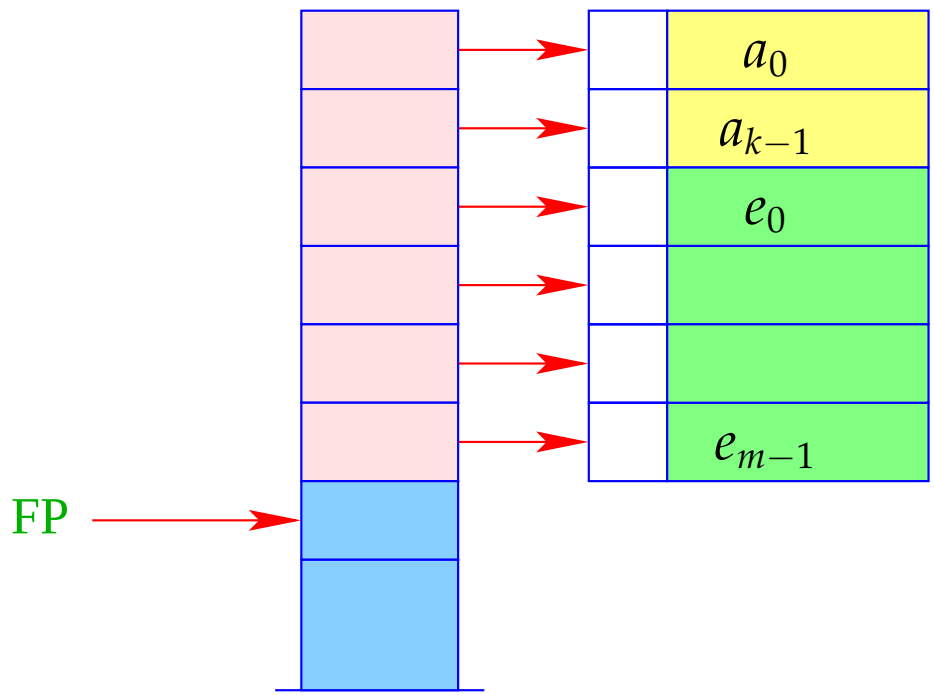
− Stellt sich heraus, dass sich e' zu einer Funktion evaluiert, die bereits partiell auf aktuelle Parameter $a_0, \ldots, a_{k-1}$ angewendet ist, müssen diese unterhalb von e_0 in den Keller hinein gefrickelt werden :-(



Alternative:



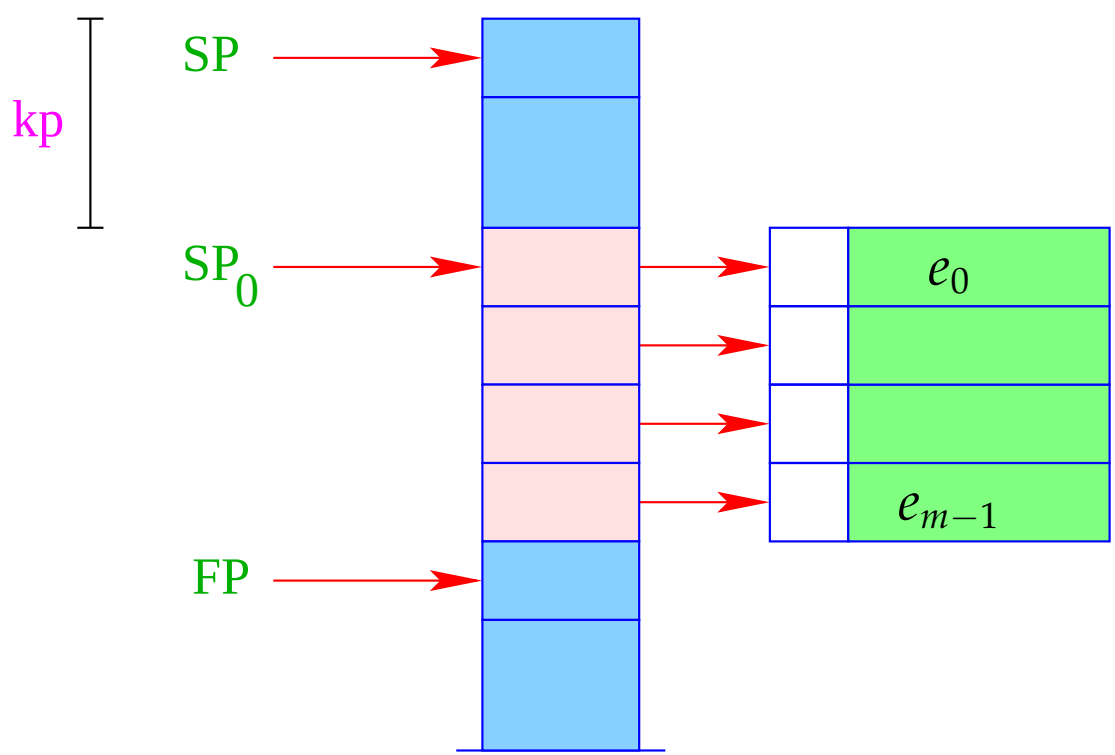
+ Die weiteren Argumente $a_0, \ldots, a_{k-1}$ wie auch die lokalen Variablen können einfach oben auf den Keller gelegt werden :-)



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

Ausweg:

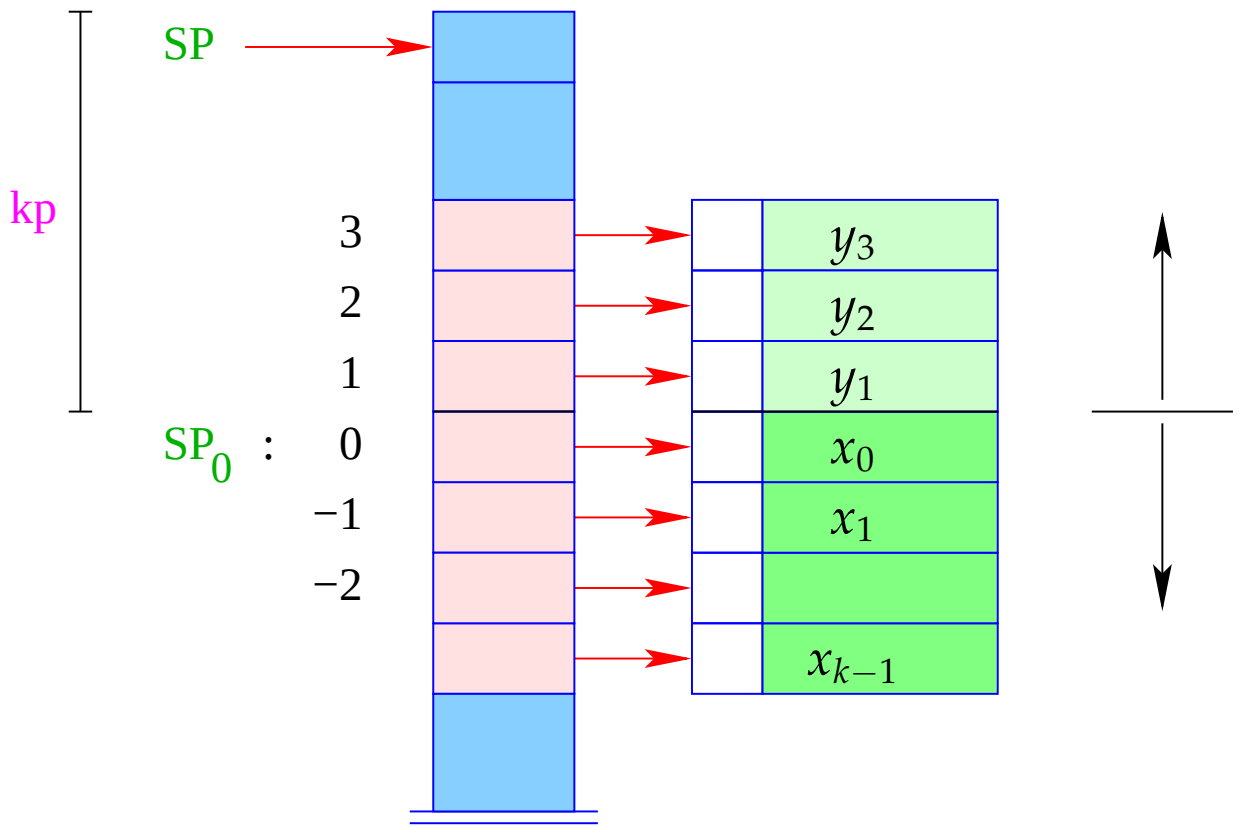
- Wir adressieren relativ zum Stackpointer SP !!!
- Leider ändert sich der Stackpointer während der Programm-Ausführung ...



- Die Abweichung des SP von seiner Position 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:

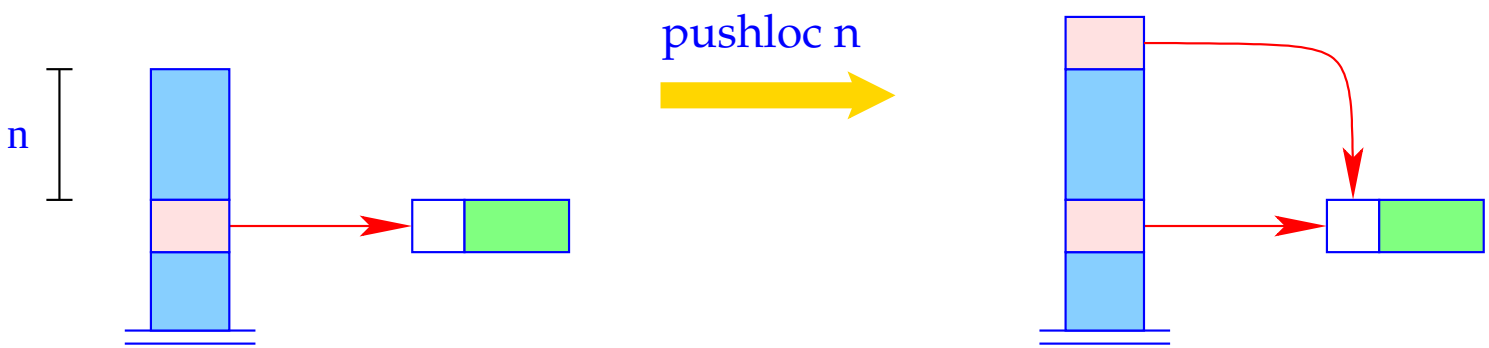
$$\begin{array}{lcl} \text{code}_V\ x\ \rho\ \text{kp} & = & \text{getvar}\ x\ \rho\ \text{kp} \\ & & \text{eval} \end{array}$$

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:

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



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

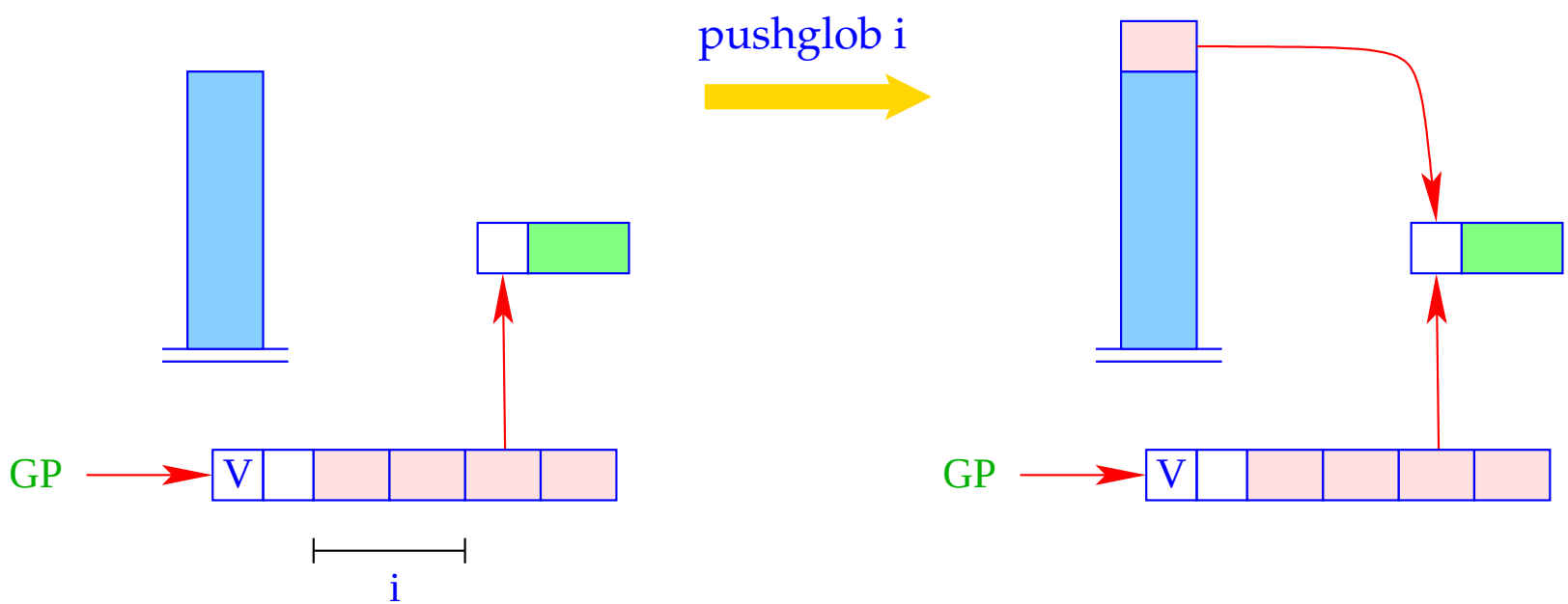
Zur Korrektheit:

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];