Helmut Seidl

Compilerbau
+
Abstrakte Maschinen

München

Sommersemester 2004

Organisatorisches

Der erste Abschnitt Die Übersetzung von C ist den Vorlesungen Compilerbau und Abstrakte Maschinen gemeinsam :-)

Er findet darum zu beiden Vorlesungsterminen statt :-)

Zeiten:

Vorlesung Compilerbau:	Mo. 12:15-13:45 Uhr
	Mi. 10:15-11:45 Uhr
Vorlesung Abstrakte Maschinen:	Mi. 13:15-14:45 Uhr
Übung Compilerbau:	Di./Do. 12:15-13:45 Uhr
	Di./Fr. 10:15-11:45 Uhr
Übung Abstrakte Maschinen:	Do. 14:15-15:45 Uhr

Einordnung:

Diplom-Studierende:

Compilerbau:	Wahlpflichtveranstaltung
Abstrakte Maschinen:	Vertiefende Vorlesung

Bachelor-Studierende:

Compilerbau:	8 ETCS-Punkte
Abstrakte Maschinen:	nicht anrechenbar

Scheinerwerb:

Diplom-Studierende:

- 50% der Punkte;
- zweimal Vorrechnen :-)

Bachelor-Studierende:

- Klausur
- Erfolgreiches Lösen der Aufgaben wird zu 20% angerechnet :-))

Material:

- Literaturliste (im Netz)
- Aufzeichnung der Vorlesungen
  (Folien + Annotationen + Ton + Bild)
- die Folien selbst :-)
- Tools zur Visualisierung der Abstrakten Maschinen :-))
- Tools, um Komponenten eines Compilers zu generieren ...

Weitere Veranstaltungen:

- Seminar Programmanalyse — Di., 14:00-16:00 Uhr
- Wahlpflicht-Praktika:

SS 2004:	Oberflächengenerierung (Frau Höllerer)
WS 2004/05:	Konstruktion eines Compilers (Frau Höllerer)

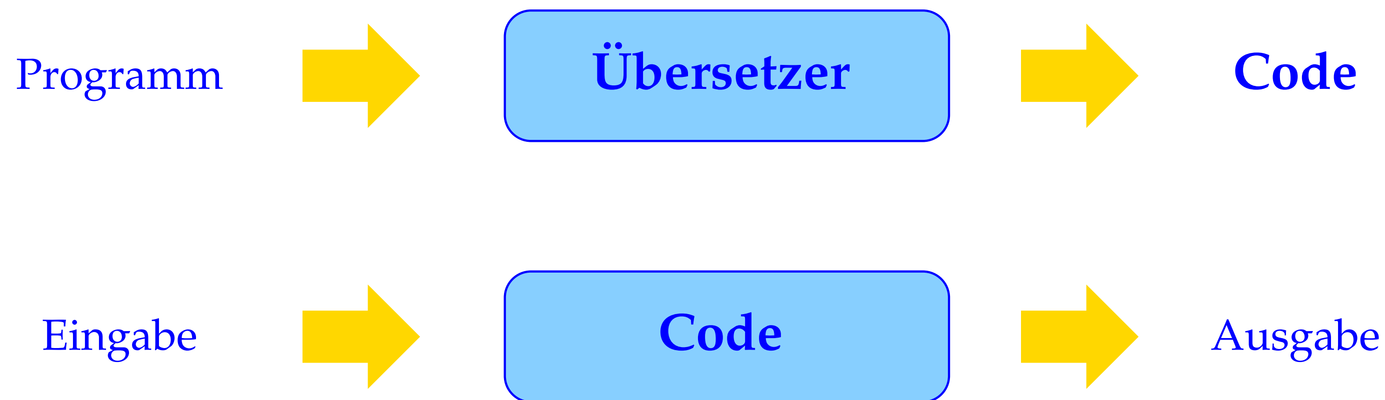
0 Einführung

Prinzip eines Interpreters:

Vorteil: Keine Vorberechnung auf dem Programmtext erforderlich $\Longrightarrow$ keine/geringe Startup-Zeit :-)

Nachteil: Während der Ausführung werden die Programm-Bestandteile immer wieder analysiert $\Longrightarrow$ längere Laufzeit :-(

Prinzip eines Übersetzers:



Zwei Phasen:

- Übersetzung des Programm-Texts in ein Maschinen-Programm;
- Ausführung des Maschinen-Programms auf der Eingabe.

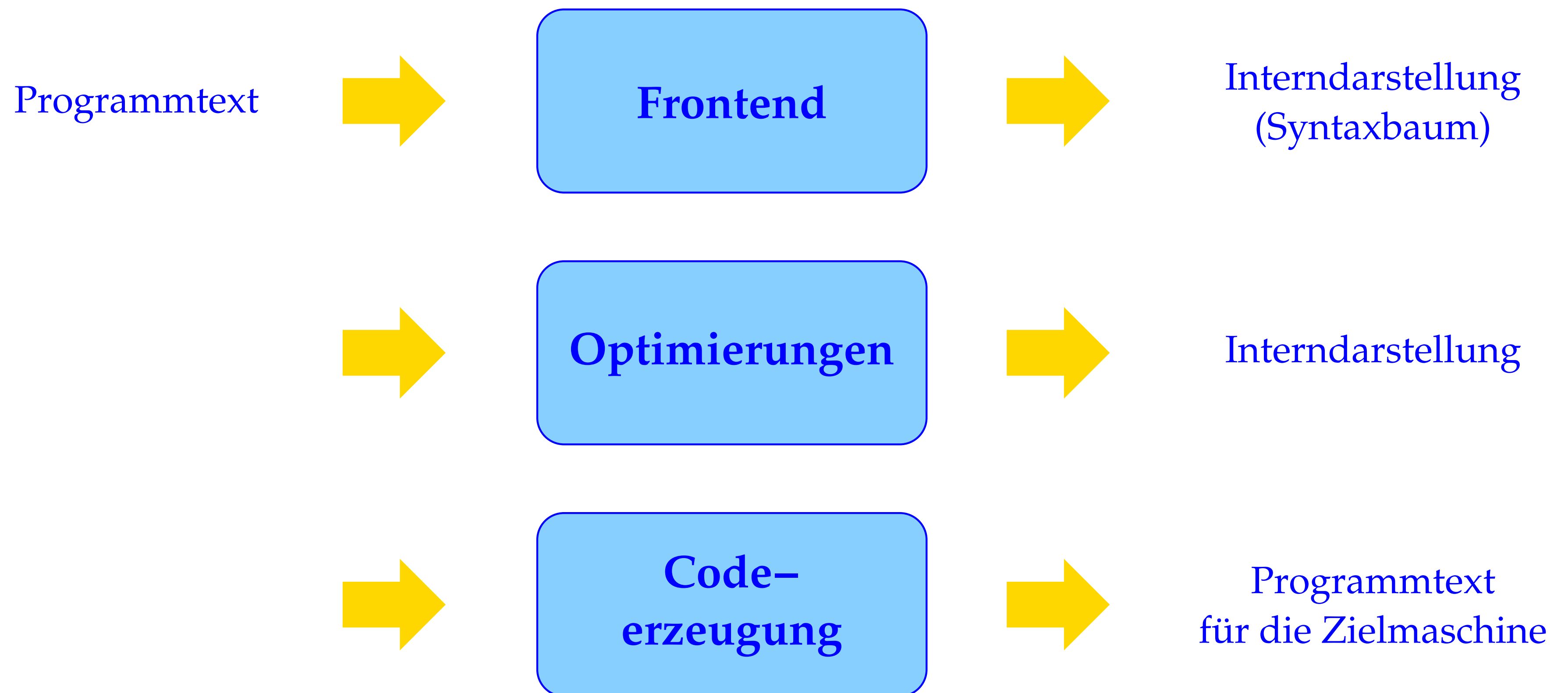
Eine Vorberechnung auf dem Programm gestattet u.a.

- eine geschickte(re) Verwaltung der Variablen;
- Erkennung und Umsetzung globaler Optimierungsmöglichkeiten.

Nachteil: Die Übersetzung selbst dauert einige Zeit :-(

Vorteil: Die Ausführung des Programme wird effizienter $\Longrightarrow$ lohnt sich bei aufwendigen Programmen und solchen, die mehrmals laufen ...

Aufbau eines Übersetzers:



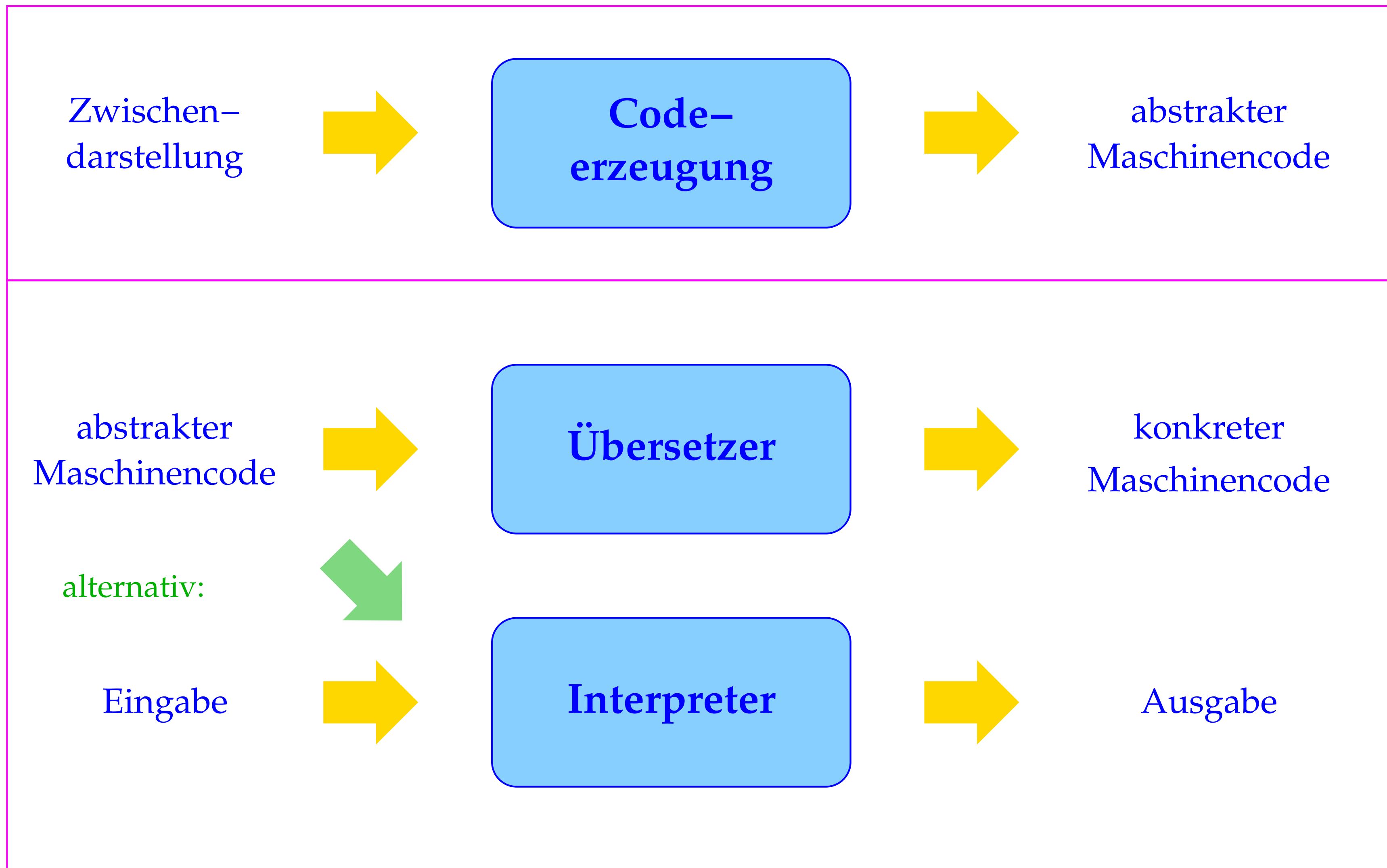
Aufgaben der Code-Erzeugung:

Ziel ist eine geschickte Ausnutzung der Möglichkeiten der Hardware. Das heißt u.a.:

1. Instruction Selection: Auswahl geeigneter Instruktionen;
2. Registerverteilung: optimale Nutzung der vorhandenen (evt. spezialisierten) Register;
3. Instruction Scheduling: Anordnung von Instruktionen (etwa zum Füllen einer Pipeline).

Weitere gegebenenfalls auszunutzende spezielle Hardware-Features können mehrfache Recheneinheiten sein, verschiedene Caches, . . .

Weil konkrete Hardware so vielgestaltig ist, wird die Code-Erzeugung oft erneut in zwei Phasen geteilt:


Zwischen–
darstellung
Code–
erzeugung
abstrakter
Maschinencode
abstrakter
Maschinencode
Übersetzer
konkreter
Maschinencode
alternativ:
Eingabe
Interpreter
Ausgabe

Eine abstrakte Maschine ist eine idealisierte Hardware, für die sich einerseits “leicht” Code erzeugen lässt, die sich andererseits aber auch “leicht” auf realer Hardware implementieren lässt.

Vorteile:

- Die Portierung auf neue Zielarchitekturen vereinfacht sich;
- der Compiler wird flexibler;
- die Realisierung der Programmkonstrukte wird von der Aufgabe entkoppelt, Hardware-Features auszunutzen.

Programmiersprachen, deren Übersetzungen auf abstrakten Maschinen beruhen:

Pascal	$\longrightarrow$	P-Maschine	
Smalltalk	$\longrightarrow$	Bytecode	
Prolog	$\longrightarrow$	WAM	(“Warren Abstract Machine”)
SML, Haskell	$\longrightarrow$	STGM	
Java	$\longrightarrow$	JVM	

Hier werden folgende Sprachen und abstrakte Maschinen betrachtet:

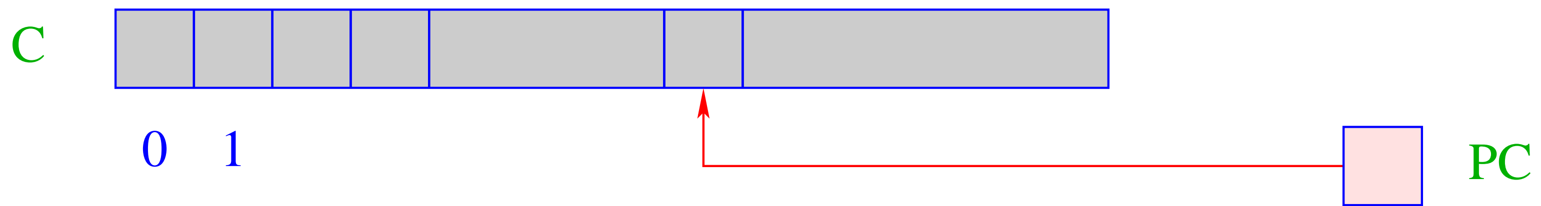
C	$\longrightarrow$	CMa	//	*imperativ*
PuF	$\longrightarrow$	MaMa	//	*funktional*
PuP	$\longrightarrow$	WiM	//	*logikbasiert*
Threaded C	$\longrightarrow$	CMa+Threads	//	*nebenläufig*

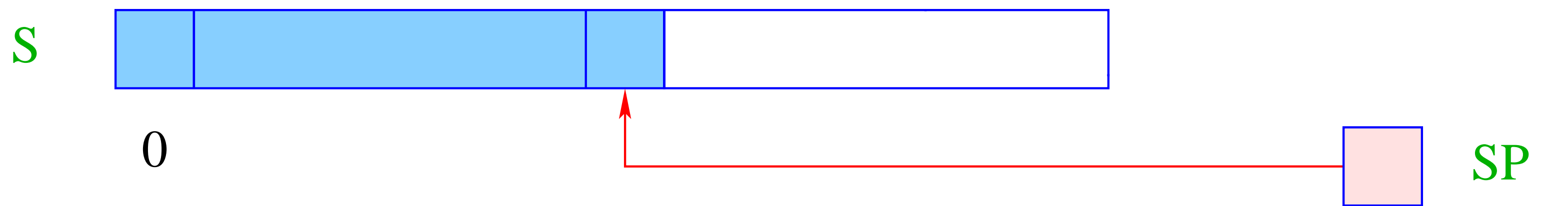
Die Übersetzung von C

1 Die Architektur der CMa

- Jede abstrakte Maschine stellt einen Satz abstrakter Instruktionen zur Verfügung.
- Instruktionen werden auf der abstrakten Hardware ausgeführt.
- Die abstrakte Hardware fassen wir als eine Menge von Datenstrukturen auf, auf die die Instruktionen zugreifen
- ... und die vom Laufzeitsystem verwaltet werden.

Für die CMa benötigen wir:


C
0
1
PC


S
0
SP

- S ist der (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können $\Longrightarrow$ Keller/Stack.
- SP ($\hat{=}$ Stack Pointer) ist ein Register, das die Adresse der obersten belegten Zelle enthält.

  Vereinfachung: Alle Daten passen jeweils in eine Zelle von S.
- C ist der Code-Speicher, der das Programm enthält.

  Jede Zelle des Felds C kann exakt einen abstrakten Befehl aufnehmen.
- PC ($\hat{=}$ Program Counter) ist ein Register, das die Adresse des nächsten auszuführenden Befehls enthält.
- Vor Programmausführung enthält der PC die Adresse 0

  $\Longrightarrow$ C[0] enthält den ersten auszuführenden Befehl.

Die Ausführung von Programmen:

- Die Maschine lädt die Instruktion aus C[PC] in ein Instruktions-Register IR und führt sie aus.
- Vor der Ausführung eines Befehls wird der PC um 1 erhöht.

```
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```

- Der PC muss vor der Ausführung der Instruktion erhöht werden, da diese möglicherweise den PC überschreibt :-)
- Die Schleife (der Maschinen-Zyklus) wird durch Ausführung der Instruktion halt verlassen, die die Kontrolle an das Betriebssystem zurückgibt.
- Die weiteren Instruktionen führen wir nach Bedarf ein :-)

2 Einfache Ausdrücke und Wertzuweisungen

Aufgabe: werte den Ausdruck $(1+7)*3$ aus!

Das heißt: erzeuge eine Instruktionsfolge, die

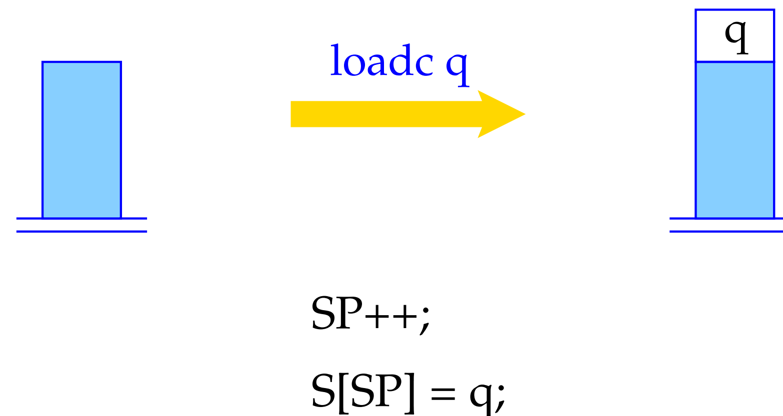
- den Wert des Ausdrucks ermittelt und dann
- oben auf dem Keller ablegt...

Idee:

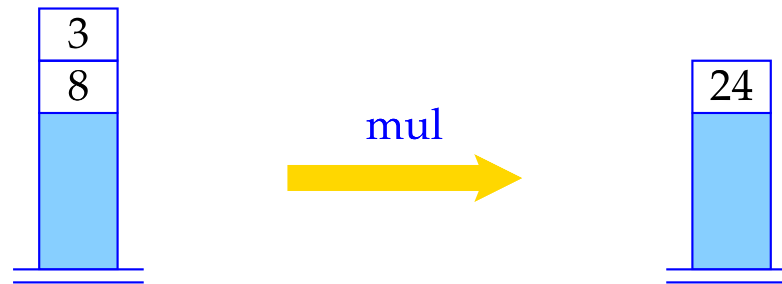
- berechne erst die Werte für die Teilausdrücke;
- merke diese Zwischenergebnisse oben auf dem Keller;
- wende dann den Operator an!

Generelles Prinzip:

- die Argumente für Instruktionen werden oben auf dem Keller erwartet;
- die Ausführung einer Instruktion konsumiert ihre Argumente;
- möglicherweise berechnete Ergebnisse werden oben auf dem Keller wieder abgelegt.



Die Instruktion loadc q benötigt keine Argumente, legt dafür aber als Wert die Konstante q oben auf dem Stack ab.



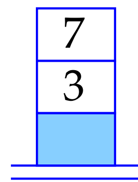
SP--;

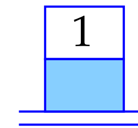
S[SP] = S[SP] $*$ S[SP+1];

mul erwartet zwei Argumente oben auf dem Stack, konsumiert sie und legt sein Ergebnis oben auf dem Stack ab.

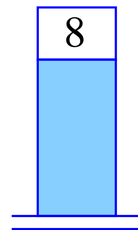
... analog arbeiten auch die übrigen binären arithmetischen und logischen Instruktionen add, sub, div, mod, and, or und xor, wie auch die Vergleiche eq, neq, le, leq, gr und geq.

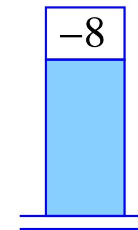
Beispiel: Der Operator leq





Einstellige Operatoren wie neg und not konsumieren dagegen ein Argument und erzeugen einen Wert:



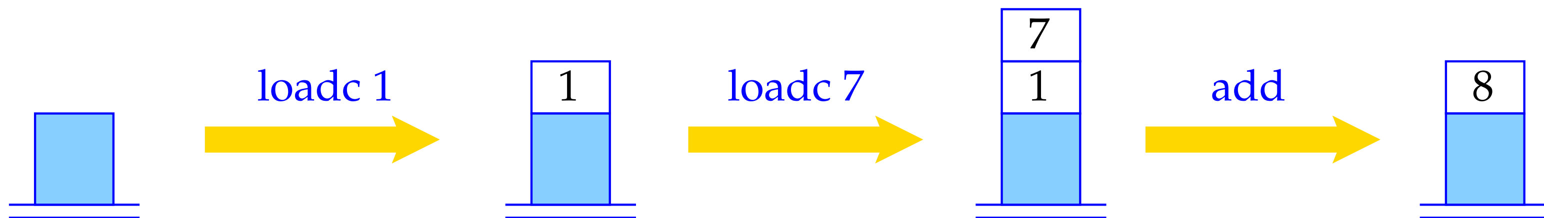


S[SP] = – S[SP];

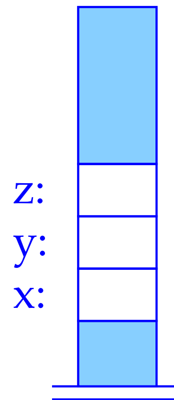
Beispiel: Code für $1 + 7$:

loadc 1 loadc 7 add

Ausführung dieses Codes:



Variablen ordnen wir Speicherzellen in S zu:



Die Übersetzungsfunktionen benötigen als weiteres Argument eine Funktion ρ, die für jede Variable x die (Relativ-)Adresse von x liefert. Die Funktion ρ heißt Adress-Umgebung (Address Environment).

Variablen können auf zwei Weisen verwendet werden.

Beispiel: $x = y + 1$

Für y sind wir am Inhalt der Zelle, für x an der Adresse interessiert.

L-Wert von x	=	Adresse von x
R-Wert von x	=	Inhalt von x

$\text{code}_\text{R}\ e\ \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
$\text{code}_\text{L}\ e\ \rho$	analog für den L-Wert

Achtung:

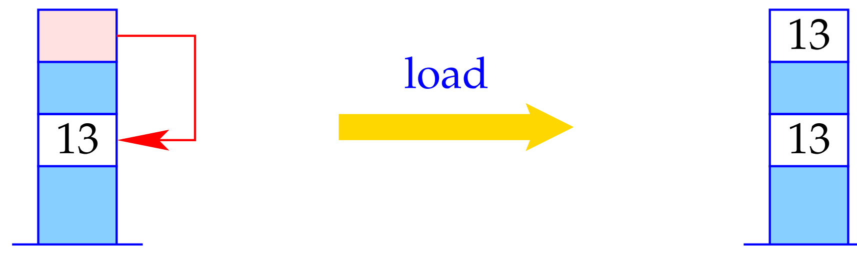
Nicht jeder Ausdruck verfügt über einen L-Wert (Bsp.: $x + 1$).

Wir definieren:

$$
\begin{array}{lcl}
\text{code}_\text{R}\ (e_1 + e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{add} \\
 & & \quad \text{... analog für die anderen binären Operatoren} \\
\text{code}_\text{R}\ (-e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\
 & & \text{neg} \\
 & & \quad \text{... analog für andere unäre Operatoren} \\
\text{code}_\text{R}\ q\ \rho & = & \text{loadc q} \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

$$\mathrm{code_R}\ x\ \rho \quad = \quad \mathrm{code_L}\ x\ \rho$$
$$\qquad\qquad\qquad\qquad \mathrm{load}$$

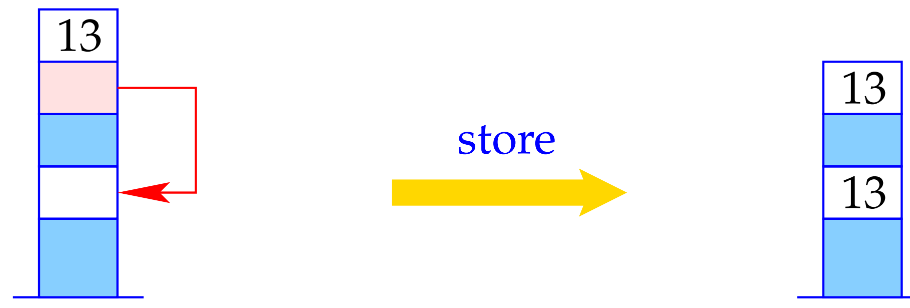
Die Instruktion load lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



S[SP] = S[S[SP]];

$$\text{code}_\text{R}\ (x = e)\ \rho \quad = \quad \text{code}_\text{L}\ x\ \rho$$
$$\text{code}_\text{R}\ e\ \rho$$
$$\text{store}$$

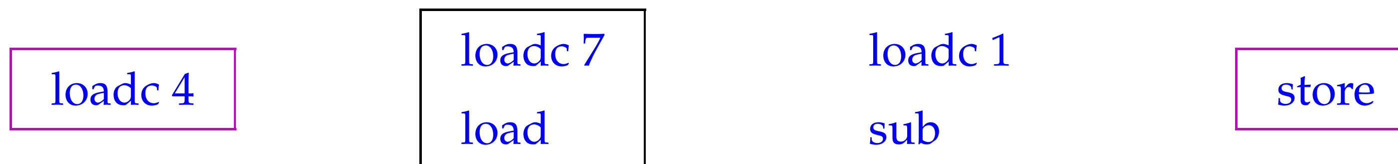
Die Instruktion store schreibt den Inhalt der obersten Speicherzelle in die Speicherzelle, deren Adresse darunter auf dem Keller steht, lässt den geschriebenen Wert aber oben auf dem Keller liegen :-)



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

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

Beispiel: Code für $e \equiv x = y - 1$ mit $\rho = \{x \mapsto 4, y \mapsto 7\}$.
Dann liefert $\text{code}_\text{R}\ e\ \rho$:



Optimierungen:

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

loada q	=	loadc q
		load
bla; storea q	=	loadc q; bla
		store

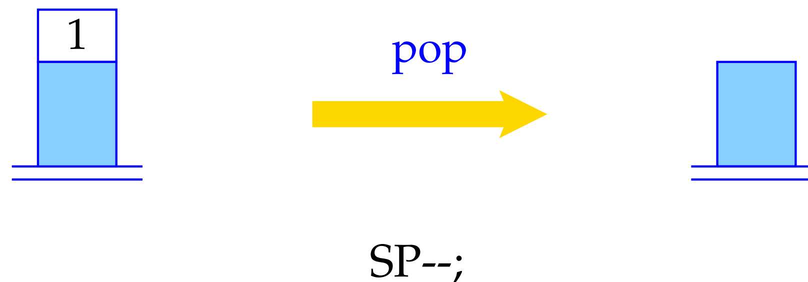
3 Anweisungen und Anweisungsfolgen

Ist e ein Ausdruck, dann ist $e;$ eine Anweisung (Statement).

Anweisungen liefern keinen Wert zurück. Folglich muss der SP vor und nach der Ausführung des erzeugten Codes gleich sein:

$$\text{code}\ e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

Die Instruktion pop wirft das oberste Element des Kellers weg ...



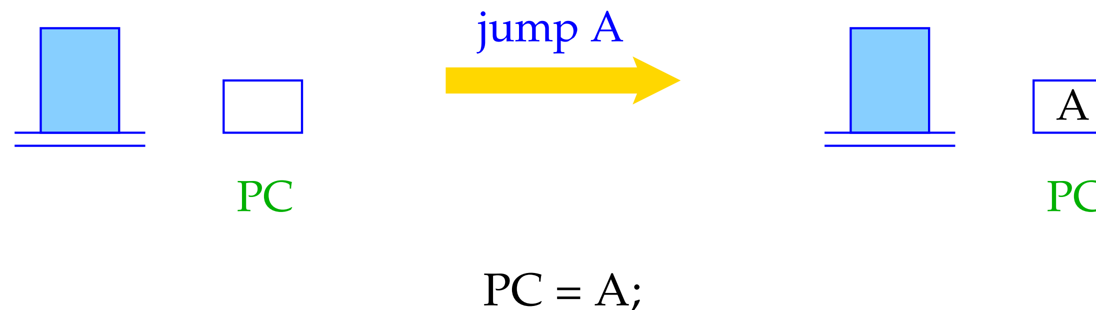
SP--;

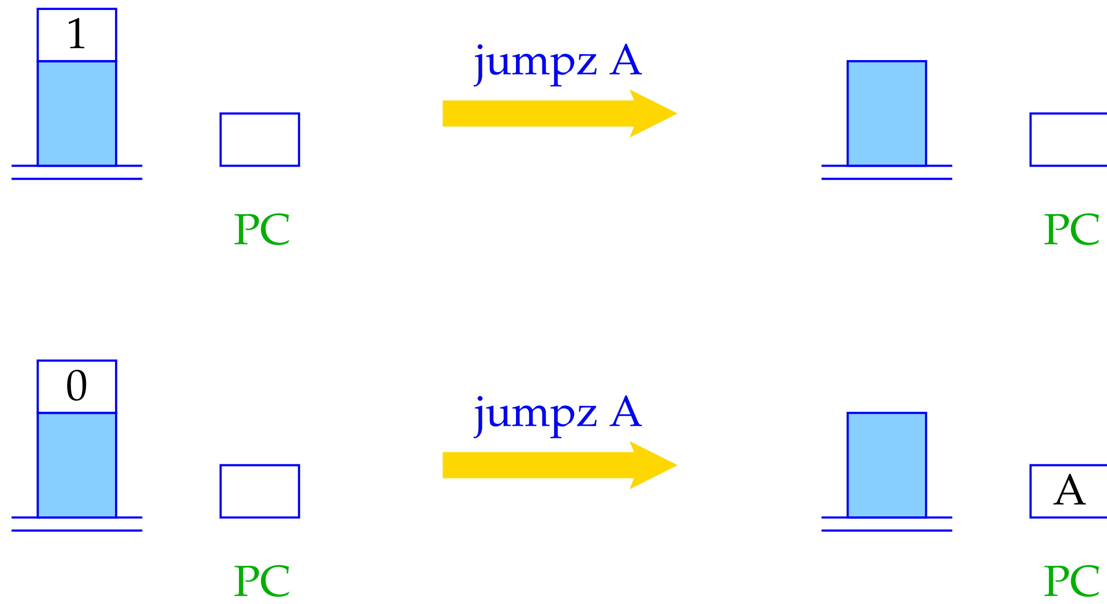
Der Code für eine Statement-Folge ist die Konkatenation des Codes for die einzelnen Statements in der Folge:

$$\begin{array}{lll} \text{code}\ (s\, ss)\ \rho & = & \text{code}\ s\ \rho \\ & & \text{code}\ ss\ \rho \\ \text{code}\ \varepsilon\ \rho & = & \quad //\ \textit{leere Folge von Befehlen} \end{array}$$

4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:





if (S[SP] == 0) PC = A;

SP--;

Der Übersichtlichkeit halber gestatten wir die Verwendung von symbolischen Sprungzielen. In einem zweiten Pass können diese dann durch absolute Code-Adressen ersetzt werden.

Statt absoluter Code-Adressen könnte man auch relative Adressen benutzen, d. h. Sprungziele relativ zum aktuellen PC angeben.

Vorteile:

- kleinere Adressen reichen aus;
- der Code wird relokierbar, d. h. kann im Speicher unverändert hin und her geschoben werden.

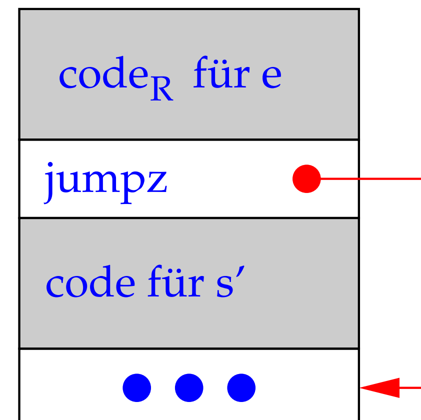
4.1 Bedingte Anweisung, einseitig

Betrachten wir zuerst $s \equiv \textbf{if}\ (e)\ s'$.

Idee:

- Lege den Code zur Auswertung von e und s' hintereinander in den Code-Speicher;
- Dekoriere mit Sprung-Befehlen so, dass ein korrekter Kontroll-Fluss gewährleistet ist!

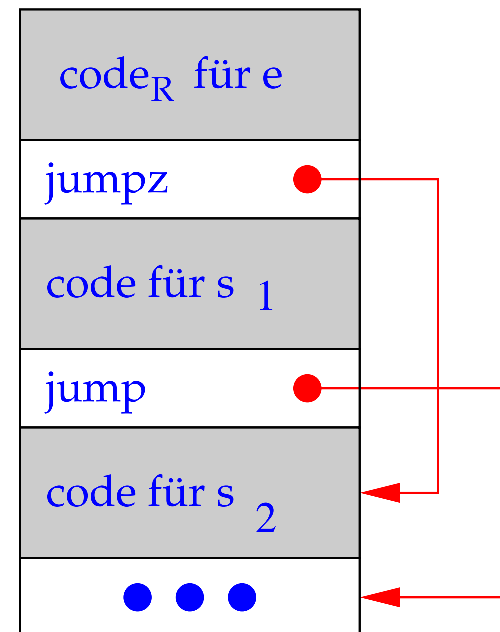
$$
\begin{aligned}
\text{code}\ s\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\
& \text{jumpz A} \\
& \text{code}\ s'\ \rho \\
\text{A}: \quad & \ldots
\end{aligned}
$$



4.2 Zweiseitiges if

Betrachte nun $s \equiv \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2$. Die gleiche Strategie liefert:

$$
\begin{array}{rrl}
\text{code}\ s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code}\ s_1\ \rho \\
 & & \text{jump B} \\
 & \text{A}: & \text{code}\ s_2\ \rho \\
 & \text{B}: & \dots
\end{array}
$$



Beispiel: Sei $\rho = \{x \mapsto 4, y \mapsto 7\}$ und

$$
\begin{array}{lll}
s & \equiv & \textbf{if}\,(x > y) \qquad (i) \\
 & & \quad x = x - y; \qquad (ii) \\
 & & \textbf{else}\; y = y - x; \qquad (iii)
\end{array}
$$

Dann liefert code $s\ \rho$:

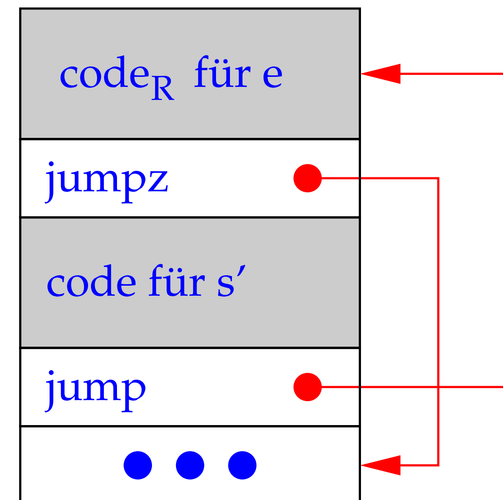
```
   loada 4        loada 4        A:  loada 7
   loada 7        loada 7            loada 4
   gr             sub                sub
   jumpz A        storea 4           storea 7
                  pop                pop
                  jump B         B:  ...
```

(i)	(ii)	(iii)

4.3 while-Schleifen

Betrachte schließlich die Schleife $s \equiv \mathbf{while}\ (e)\ s'$. Dafür erzeugen wir:

$$
\begin{array}{lll}
\text{code}\ s\ \rho & = & \\
 & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$



Beispiel: Sei $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ und s das Statement:

$$\textbf{while}\ (a > 0)\ \{c = c + 1;\ a = a - b;\}$$

Dann liefert code $s\ \rho$ die Folge:

A:	loada 7	loada 9	loada 7	B:	...
	loadc 0	loadc 1	loada 8		
	gr	add	sub		
	jumpz B	storea 9	storea 7		
		pop	pop		
			jump A		

4.4 for-Schleifen

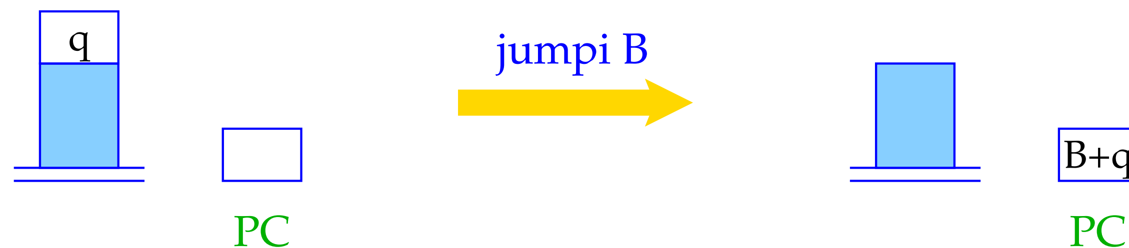
Die **for**-Schleife $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ ist äquivalent zu der Statementfolge $e_1;$ **while** $(e_2)\ \{s'\ e_3;\}$ – sofern s' keine **continue**-Anweisung enthält.
Darum übersetzen wir:

$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_\text{R}\ e_1 \\
 & & \text{pop} \\
 & \text{A}: & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{code}_\text{R}\ e_3\ \rho \\
 & & \text{pop} \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$

4.5 Das switch-Statement

Idee:

- Unterstütze Mehrfachverzweigung in konstanter Zeit!
- Benutze Sprungtabelle, die an der *i*-ten Stelle den Sprung an den Anfang der *i*-tem Alternative enthält.
- Eine Möglichkeit zur Realisierung besteht in der Einführung von indizierten Sprüngen.



PC = B + S[SP];

SP--;

Vereinfachung:

Wir betrachten nur **switch**-Statements der folgenden Form:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \quad \textbf{case}\ 0\text{:} \quad ss_0\ \textbf{break;} \\
 & & \quad \textbf{case}\ 1\text{:} \quad ss_1\ \textbf{break;} \\
 & & \qquad \vdots \\
 & & \quad \textbf{case}\ k-1\text{:} \quad ss_{k-1}\ \textbf{break;} \\
 & & \quad \textbf{default:} \quad ss_k \\
 & & \}
\end{array}
$$

Dann ergibt sich für s die Instruktionsfolge:

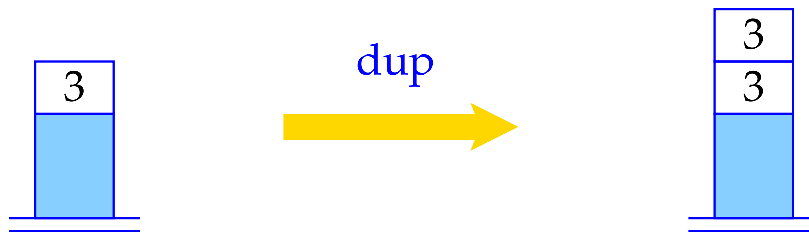
code s ρ	=	code$_R$ e ρ	C_0:	code ss_0 ρ	B:	jump C_0
		check 0 k B		jump D		...
				...		jump C_k
			C_k:	code ss_k ρ	D:	...
				jump D		

- Das Macro check 0 k B überprüft, ob der R-Wert von e im Intervall $[0, k]$ liegt, und führt einen indizierten Sprung in die Tabelle B aus.
- Die Sprungtabelle enthält direkte Sprünge zu den jeweiligen Alternativen.
- Am Ende jeder Alternative steht ein Sprung hinter das **switch**-Statement.

check 0 k B =

```
dup         dup             jumpi B
loadc 0     loadc k     A:  pop
geq         leq             loadc k
jumpz A     jumpz A         jumpi B
```

- Weil der R-Wert von e noch zur Indizierung benötigt wird, muss er vor jedem Vergleich kopiert werden.
- Dazu dient der Befehl dup.
- Ist der R-Wert von e kleiner als 0 oder größer als k, ersetzen wir ihn vor dem indizierten Sprung durch k.



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

SP++;

Achtung:

- Die Sprung-Tabelle könnte genauso gut direkt hinter dem Macro check liegen. Dadurch spart man ein paar unbedingte Sprünge, muss aber evt. das **switch**-Statement zweimal durchsuchen.
- Beginnt die Tabelle mit u statt mit 0, müssen wir den R-Wert von e um u vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von e sicher im Intervall $[0, k]$, können wir auf check verzichten.

5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x statisch, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse $\rho\, x$ zu!

Annahmen:

- Variablen von Basistypen wie **int**, ... erhalten eine Speicherzelle.
- Variablen werden in der Reihenfolge im Speicher abgelegt, wie sie deklariert werden, und zwar ab Adresse 1.

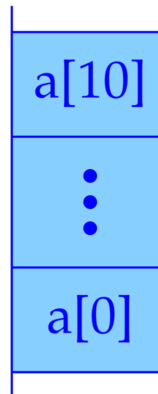
Folglich erhalten wir für die Deklaration $d \equiv t_1\; x_1; \;\ldots\; t_k\; x_k;$ (t_i einfach) die Adress-Umgebung ρ mit

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Felder

Beispiel: **int** [11] a;

Das Feld a enthält 11 Elemente und benötigt darum 11 Zellen.
$\rho\, a$ ist die Adresse des Elements $a[0]$.



Notwendig ist eine Funktion sizeof (hier: $|\cdot|$), die den Platzbedarf eines Typs berechnet:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ einfach} \\ k \cdot |t'| & \text{falls } t \equiv t'[k] \end{cases}$$

Dann ergibt sich für die Deklaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{array}{lll} \rho\, x_1 & = & 1 \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| \qquad \text{für } i > 1 \end{array}$$

Weil $|\cdot|$ zur Übersetzungszeit berechnet werden kann, kann dann auch ρ zur Übersetzungszeit berechnet werden.

Aufgabe:

Erweitere code_L und code_R auf Ausdrücke mit indizierten Feldzugriffen.

Sei $t[c]\ a;$ die Deklaration eines Feldes a.

Um die Anfangsadresse der Datenstruktur $a[i]$ zu bestimmen, müssen wir $\rho\, a + |t| * (\textit{R-Wert von } i)$ ausrechnen. Folglich:

$$
\begin{aligned}
\text{code}_L\ a[e]\ \rho \quad = \quad & \text{loadc}\ (\rho\, a) \\
& \text{code}_R\ e\ \rho \\
& \text{loadc}\ |t| \\
& \text{mul} \\
& \text{add}
\end{aligned}
$$

... oder allgemeiner:

$$\begin{aligned} \text{code}_L\ e_1[e_2]\ \rho \quad = \quad & \text{code}_R\ e_1\ \rho \\ & \text{code}_R\ e_2\ \rho \\ & \text{loadc}\ |t| \\ & \text{mul} \\ & \text{add} \end{aligned}$$

Bemerkung:

- In C ist ein Feld ein Zeiger. Ein deklariertes Feld a ist eine Zeiger-Konstante, deren R-Wert die Anfangsadresse des Feldes ist.
- Formal setzen wir für ein Feld e: $\quad \text{code}_R\ e\ \rho = \text{code}_L\ e\ \rho$
- In C sind äquivalent (als L-Werte):

$$2[a] \qquad a[2] \qquad a+2$$

5.2 Strukturen

In Modula heißen Strukturen Records.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwandt.
Alternativ könnte man zu jedem Struktur-Typ *st* eine separate
Komponenten-Umgebung ρ_{st} verwalten :-)

Sei **struct** { **int** a; **int** b; } x; Teil einer Deklarationsliste.

- x erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen relativ zum Anfang der Struktur, hier $a \mapsto 0$, $b \mapsto 1$.

Sei allgemein $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. Dann ist

$$
\begin{aligned}
|t| &= \sum_{i=1}^{k} |t_i| \\
\rho\, c_1 &= 0 \quad \text{und} \\
\rho\, c_i &= \rho\, c_{i-1} + |t_{i-1}| \quad \text{für } i > 1
\end{aligned}
$$

Damit erhalten wir:

$$
\begin{aligned}
\text{code}_\text{L}\ (e.c)\ \rho \quad = \quad & \text{code}_\text{L}\ e\ \rho \\
& \text{loadc}\ (\rho\, c) \\
& \text{add}
\end{aligned}
$$

Beispiel:

Sei **struct** { **int** a; **int** b; } x; mit $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

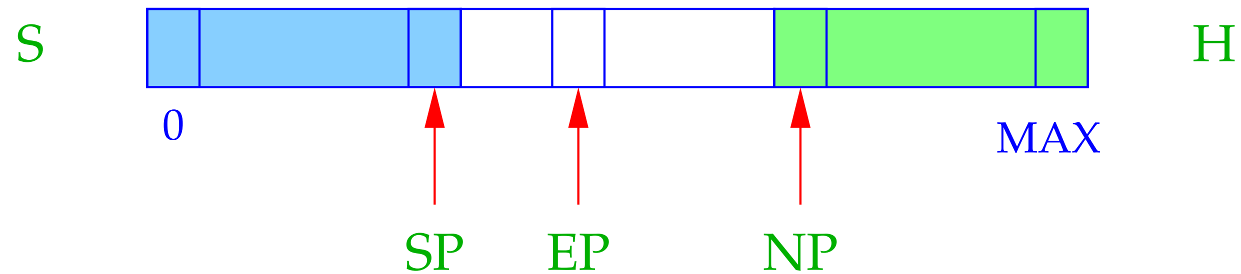
Dann ist

$$\text{code}_\text{L}\ (x.b)\ \rho \quad = \quad \begin{array}{l} \text{loadc } 13 \\ \text{loadc } 1 \\ \text{add} \end{array}$$

6 Zeiger und dynamische Speicherverwaltung

Zeiger (Pointer) gestatten den Zugriff auf anonyme, dynamisch erzeugte Datenelemente, deren Lebenszeit nicht dem LIFO-Prinzip unterworfen ist.

$\Longrightarrow$ Wir benötigen eine weitere potentiell beliebig große Datenstruktur H – den Heap (bzw. die Halde):



NP	$\widehat{=}$	New Pointer; zeigt auf unterste belegte Haldenzelle.
EP	$\widehat{=}$	Extreme Pointer; zeigt auf die Zelle, auf die der SP maximal zeigen kann (innerhalb der aktuellen Funktion).

Idee dabei:

- Chaos entsteht, wenn Stack und Heap sich überschneiden (Stack Overflow).
- Eine Überschneidung kann bei jeder Erhöhung von SP, bzw. jeder Erniedrigung des NP eintreten.
- EP erspart uns die Überprüfungen auf Überschneidung bei den Stackoperationen :-)
- Die Überprüfungen bei Heap-Allokationen bleiben erhalten :-(.

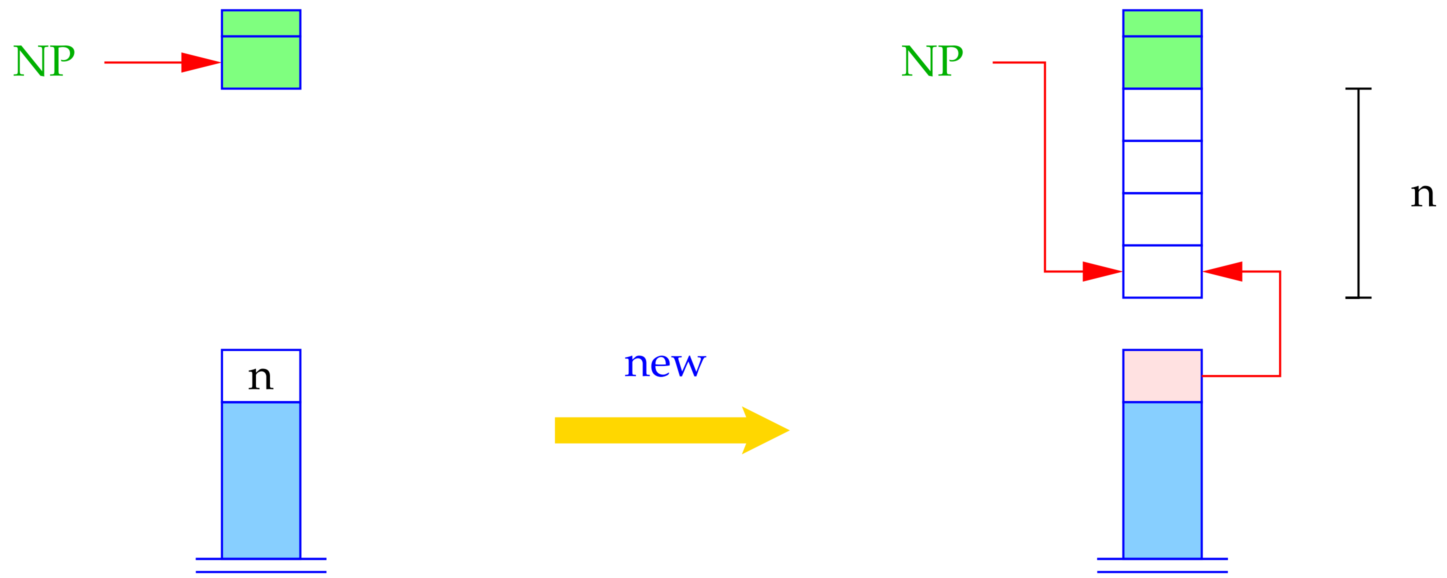
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- Zeiger zu erzeugen, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu dereferenzieren, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Es gibt zwei Arten, Zeiger zu erzeugen:

(1) Ein Aufruf von **malloc** liefert einen Zeiger auf eine Heap-Zelle:

$$
\begin{aligned}
\text{code}_{\text{R}}\ \textbf{malloc}\,(e)\ \rho \quad = \quad & \text{code}_{\text{R}}\ e\ \rho \\
& \text{new}
\end{aligned}
$$


NP
n
new
NP
n

```
if (NP - S[SP] ≤ EP)
        S[SP] = NULL;
else {
        NP = NP - S[SP];
        S[SP] = NP;
}
```

- NULL ist eine spezielle Zeigerkonstante (etwa 0 :-)
- Im Falle einer Kollision von Stack und Heap wird der NULL-Zeiger zurückgeliefert.

(2) Die Anwendung des Adressoperators & liefert einen Zeiger auf eine Variable, d. h. deren Adresse ($\widehat{=}$ L-Wert). Deshalb:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$

Dereferenzieren von Zeigern:

Die Anwendung des Operators $*$ auf den Ausdruck e liefert den Inhalt der Speicherzelle, deren Adresse der R-Wert von e ist:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

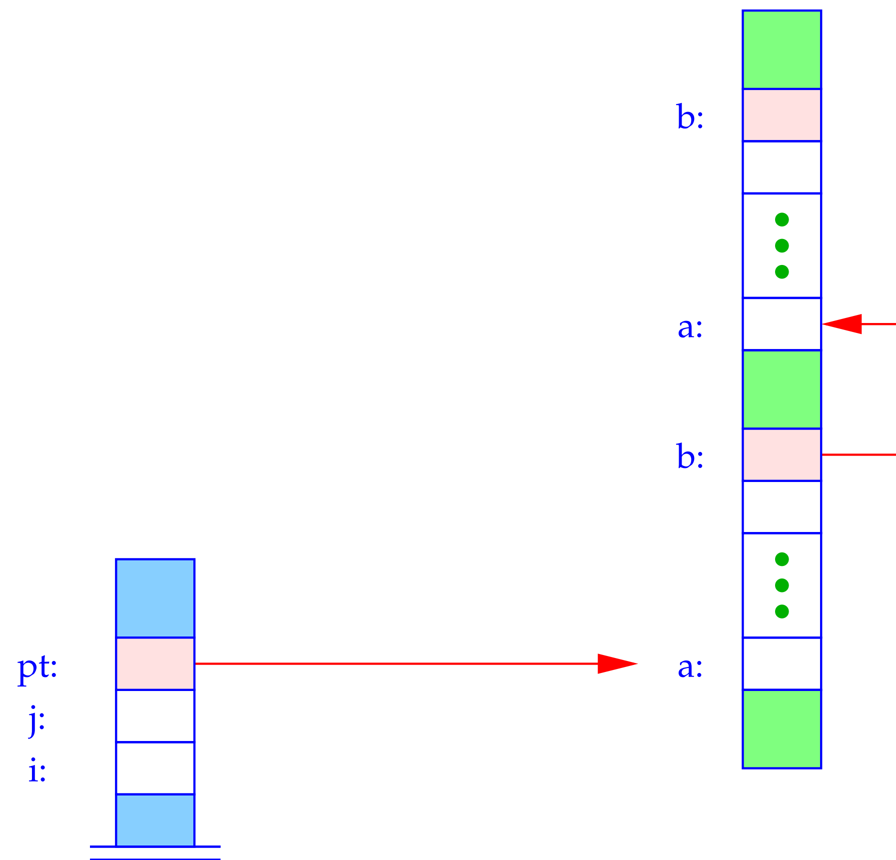
Beispiel: Betrachte für

$$\textbf{struct}\ t\ \{\ \textbf{int}\ a[7];\ \textbf{struct}\ t\ *b;\ \};$$
$$\textbf{int}\ i, j;$$
$$\textbf{struct}\ t\ *pt;$$

den Ausdruck $e \equiv ((pt \to b) \to a)[i+1]$

Wegen $e \to a \equiv (*e).a$ gilt:

$$\text{code}_\text{L}\ (e \to a)\ \rho \quad = \quad \text{code}_\text{R}\ e\ \rho$$
$$\text{loadc}\ (\rho\, a)$$
$$\text{add}$$


b:
a:
b:
pt:
a:
j:
i:

Sei $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Dann ist:

$$
\begin{array}{lllll}
\text{code}_L\ e\ \rho & = & \text{code}_R\ ((pt \to b) \to a)\ \rho & = & \text{code}_R\ ((pt \to b) \to a)\ \rho \\
 & & \text{code}_R\ (i+1)\ \rho & & \text{loada 1} \\
 & & \text{loadc 1} & & \text{loadc 1} \\
 & & \text{mul} & & \text{add} \\
 & & \text{add} & & \text{loadc 1} \\
 & & & & \text{mul} \\
 & & & & \text{add}
\end{array}
$$

Für Felder ist der R-Wert gleich dem L-Wert. Deshalb erhalten wir:

$\text{code}_R\ ((pt \to b) \to a)\ \rho$	=	$\text{code}_R\ (pt \to b)\ \rho$	=	loada 3
		loadc 0		loadc 7
		add		add
				load
				loadc 0
				add

Damit ergibt sich insgesamt die Folge:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

7 Zusammenfassung

Stellen wir noch einmal die Schemata zur Übersetzung von Ausdrücken zusammen.

$$
\begin{array}{lll}
\text{code}_L\ (e_1[e_2])\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add} \qquad\qquad \text{sofern } e_1 \text{ Typ } t\,[\,] \text{ hat}
\end{array}
$$

$$
\begin{array}{lll}
\text{code}_L\ (e.a)\ \rho & = & \text{code}_L\ e\ \rho \\
 & & \text{loadc}\ (\rho\, a) \\
 & & \text{add}
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) & \\
\text{code}_\text{R}\ (\&e)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
\text{code}_\text{R}\ (\text{malloc}\,(e))\ \rho & = & \text{code}_\text{R}\ e\ \rho & \\
 & & \text{new} & \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \text{falls } e \text{ ein Feld ist} \\
\text{code}_\text{R}\ (e_1 \Box e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{op} & \text{op Befehl zu Operator '}\Box\text{'}
\end{array}
$$

$$\text{code}_R\ q\ \rho \quad = \quad \text{loadc q} \qquad q \text{ Konstante}$$

$$\begin{aligned}\text{code}_R\ (e_1 = e_2)\ \rho \quad = \quad & \text{code}_L\ e_1\ \rho \\ & \text{code}_R\ e_2\ \rho \\ & \text{store}\end{aligned}$$

$$\begin{aligned}\text{code}_R\ e\ \rho \quad = \quad & \text{code}_L\ e\ \rho \\ & \text{load} \qquad \text{sonst}\end{aligned}$$

Beispiel: **int** $a[10], *b;$ mit $\rho = \{a \mapsto 7, b \mapsto 17\}$.

Betrachte das Statement: $s_1 \equiv *a = 5;$

Dann ist:

$$\text{code}_L\ (*a)\ \rho \quad = \quad \text{code}_R\ a\ \rho \quad = \quad \text{code}_L\ a\ \rho \quad = \quad \text{loadc } 7$$

$$\text{code}\ s_1\ \rho \quad = \quad \begin{array}{l} \text{loadc } 7 \\ \text{loadc } 5 \\ \text{store} \\ \text{pop} \end{array}$$

Zur Übung übersetzen wir auch noch:

$$s_2 \equiv b = (\&a) + 2; \qquad \text{und} \qquad s_3 \equiv *(b + 3) = 5;$$

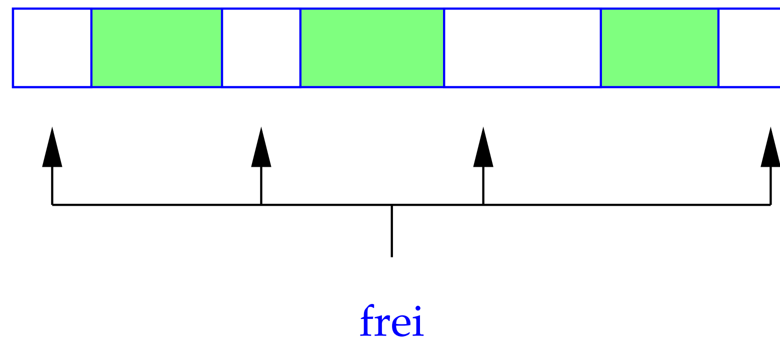
$\text{code}\ (s_2 s_3)\ \rho \quad = \quad$

```
loadc 17                      loadc 17
loadc 7                       load
loadc 2                       loadc 3
loadc 1    //  Skalierung     loadc 1    //  Skalierung
mul                           mul
add                           add
store                         loadc 5
pop        //  Ende von s2    store
                              pop        //  Ende von s3
```

8 Freigabe von Speicherplatz

Probleme:

- Der freigegebene Speicherbereich wird noch von anderen Zeigern referenziert (dangling references).
- Nach einiger Freigabe könnte der Speicher etwa so aussehen (fragmentation):



Mögliche Auswege:

- Nimm an, der Programmierer weiß, was er tut. Verwalte dann die freien Abschnitte (etwa sortiert nach Größe) in einer speziellen Datenstruktur;

  $\Longrightarrow$ **malloc** wird teuer :-(

- Tue nichts, d.h.:

$$\text{code } \textbf{free}\,(e);\ \rho \quad = \quad \text{code}_{\text{R}}\ e\ \rho$$
$$\text{pop}$$

  $\Longrightarrow$ einfach und (i.a.) effizient :-)

- Benutze eine automatische, evtl. “konservative” Garbage-Collection, die gelegentlich sicher nicht mehr benötigten Heap-Platz einsammelt und dann **malloc** zur Verfügung stellt.

9 Funktionen

Die Definition einer Funktion besteht aus

- einem Namen, mit dem sie aufgerufen werden kann;
- einer Spezifikation der formalen Parameter;
- evtl. einem Ergebnistyp;
- einem Anweisungsteil.

In C gilt:

$$\text{code}_R\ f\ \rho \quad = \quad \text{load c _}f \quad = \quad \text{Anfangsadresse des Codes für } f$$

$\Longrightarrow$ Auch Funktions-Namen müssen in der Adress-Umgebung verwaltet werden!

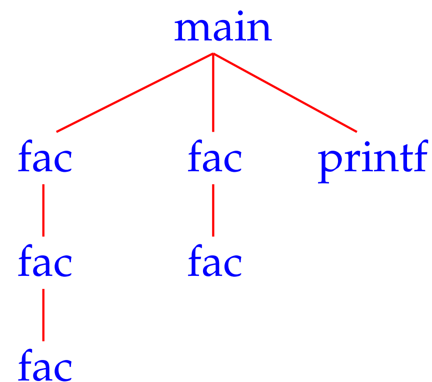
Beispiel:

```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf ("%d", n);
}
```

Zu einem Ausführungszeitpunkt können mehrere Instanzen (Aufrufe) der gleichen Funktion aktiv sein, d. h. begonnen, aber noch nicht beendet sein.

Der Rekursionsbaum im Beispiel:



Wir schließen:

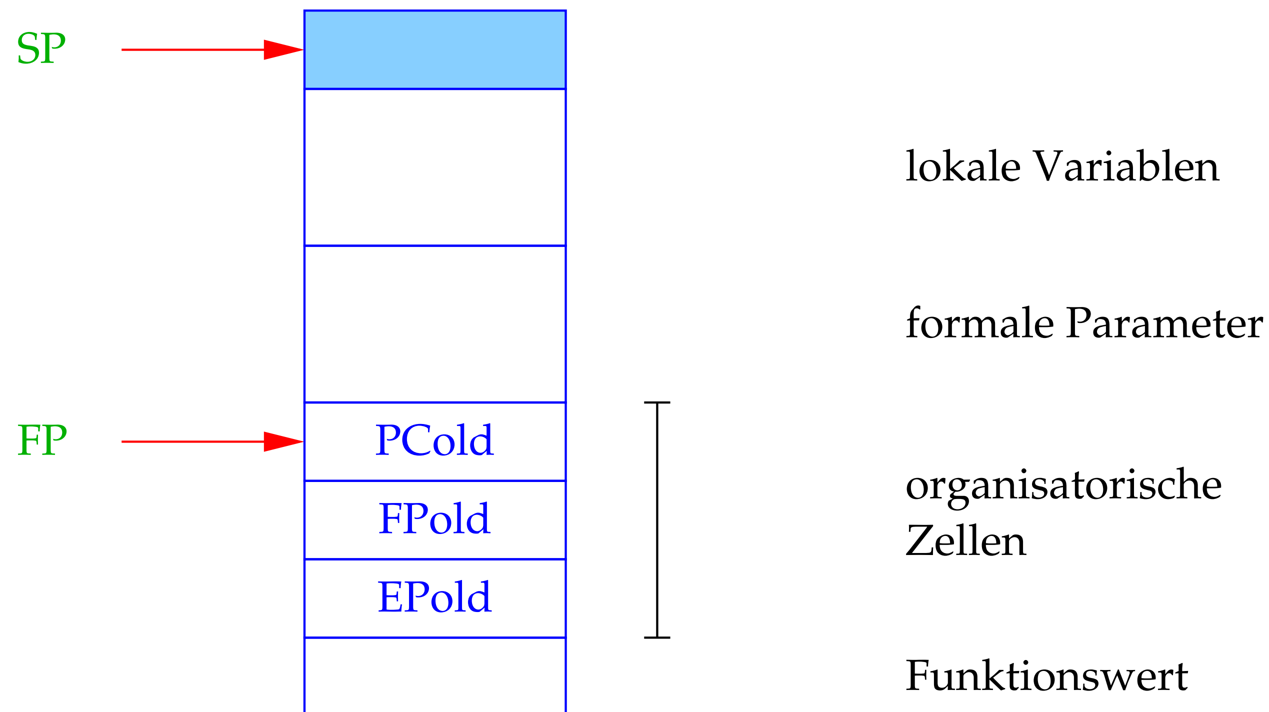
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee:

Lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.

In sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden. Deshalb heißen sie auch Keller-Rahmen (oder Stack Frame).

9.1 Speicherorganisation für Funktionen



FP $\widehat{=}$ Frame Pointer; zeigt auf die letzte organisatorische Zelle und wird zur Adressierung der formalen Parameter und lokalen Variablen benutzt.

- Die lokalen Variablen und formalen Parameter adressieren wir relativ zu FP.
- Bei einem Funktions-Aufruf muss der FP in eine organisatorische Zelle gerettet werden.
- Weiterhin müssen gerettet werden:
  - die Fortsetzungsadresse nach dem Aufruf;
  - der aktuelle EP.

Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Unsere Übersetzungsaufgaben für Funktionen:

- Erzeuge Code für den Rumpf!
- Erzeuge Code für Aufrufe!

9.2 Bestimmung der Adress-Umgebung

Wir müssen zwei Arten von Variablen unterscheiden:

1. globale/externe, die außerhalb von Funktionen definiert werden;
2. lokale/interne/automatische (inklusive formale Parameter), die innerhalb von Funktionen definiert werden.

Die Adress-Umgebung ρ ordnet den Namen Paare $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ zu.

Achtung:

- Tatsächlich gibt es i.a. weitere verfeinerte Abstufungen der Sichtbarkeit von Variablen.
- Bei der Übersetzung eines Programms gibt es i.a. für verschiedene Programmteile verschiedene Adress-Umgebungen!

Beispiel:

```
[0]  int i;
     struct list {
         int info;
         struct list * next;
     } * l;

[1]  int ith (struct list * x, int i) {
         if (i ≤ 1) return x →info;
         else return ith (x →next, i − 1);
     }

[2]  main () {
         int k;
         scanf ("%d", &i);
         scanlist (&l);
         printf ("\n\t%d\n", ith (l,i));
     }
```

Vorkommende Adress-Umgebungen in dem Programm:

0 Außerhalb der Funktions-Definitionen:

$$\begin{array}{llll} \rho_0 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \ldots & \end{array}$$

1 Innerhalb von ith:

$$\begin{array}{llll} \rho_1 : & i & \mapsto & (L, 2) \\ & x & \mapsto & (L, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \ldots & \end{array}$$

[2] Innerhalb von main:

$$\begin{array}{rrcl} \rho_2 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & k & \mapsto & (L, 1) \\ & \text{ith} & \mapsto & (G, \text{_ith}) \\ & \text{main} & \mapsto & (G, \text{_main}) \\ & & \ldots & \end{array}$$

9.3 Betreten und Verlassen von Funktionen

Sei f die aktuelle Funktion, d. h. der Caller, und f rufe die Funktion g auf, d. h. den Callee.

Der Code für den Aufruf muss auf den Caller und den Callee verteilt werden. Die Aufteilung kann nur so erfolgen, dass der Teil, der von Informationen des Callers abhängt, auch dort erzeugt wird (analog für den Callee).

Aktionen beim Betreten von g:

1. Retten von FP, EP } mark
2. Bestimmung der aktuellen Parameter
3. Bestimmung der Anfangsadresse von g
4. Setzen des neuen FP
5. Retten von PC und
   Sprung an den Anfang von g } call
6. Setzen des neuen EP } enter
7. Allokieren der lokalen Variablen } alloc

(Schritte 1–5: stehen in f; Schritte 6–7: stehen in g)

Aktionen beim Verlassen von g:

1. Rücksetzen der Register FP, EP, SP
2. Rücksprung in den Code von f, d. h.
   Restauration des PC } return

Damit erhalten wir für einen Aufruf:

$$
\begin{aligned}
\text{code}_\text{R}\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \text{mark} \\
& \text{code}_\text{R}\ e_1\ \rho \\
& \ldots \\
& \text{code}_\text{R}\ e_n\ \rho \\
& \text{code}_\text{R}\ g\ \rho \\
& \text{call m}
\end{aligned}
$$

wobei m der Platz für die aktuellen Parameter ist.

Beachte:

- Von jedem Ausdruck, der als aktueller Parameter auftritt, wird jeweils der R-Wert berechnet $\Longrightarrow$ Call-by-Value-Parameter-Übergabe.
- Die Funktion g kann auch ein Ausdruck sein, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...

- Ähnlich deklarierten Feldern, werden Funktions-Namen als konstante Zeiger auf Funktionen aufgefasst. Dabei ist der R-Wert dieses Zeigers gleich der Anfangs-Adresse der Funktion.
- Achtung! Für eine Variable **int** $(*)()\ g;$ sind die beiden Aufrufe

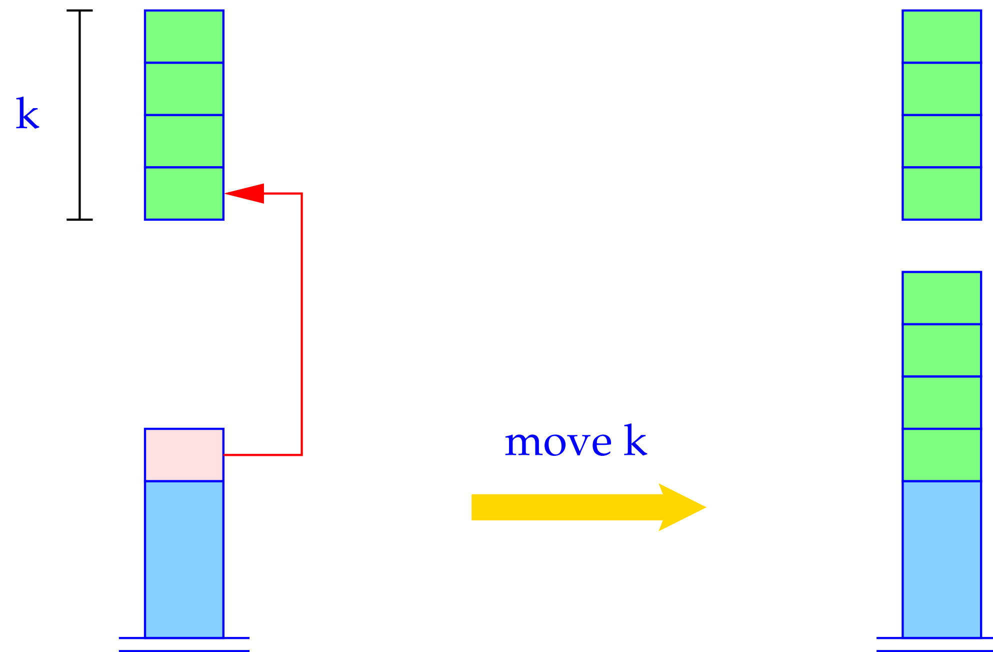
  $$(*g)() \qquad \text{und} \qquad g()$$

  äquivalent! Per Normalisierung, muss man sich hier vorstellen, werden Dereferenzierungen eines Funktions-Zeigers ignoriert :-)
- Bei der Parameter-Übergabe von Strukturen werden diese kopiert.

Folglich:

$\text{code}_R\ f\ \rho$	=	$\text{loadc}\ (\rho\ f)$	f ein Funktions-Name
$\text{code}_R\ (*e)\ \rho$	=	$\text{code}_R\ e\ \rho$	e ein Funktions-Zeiger
$\text{code}_R\ e\ \rho$	=	$\text{code}_L\ e\ \rho$	
		move k	e eine Struktur der Größe k

Dabei ist:

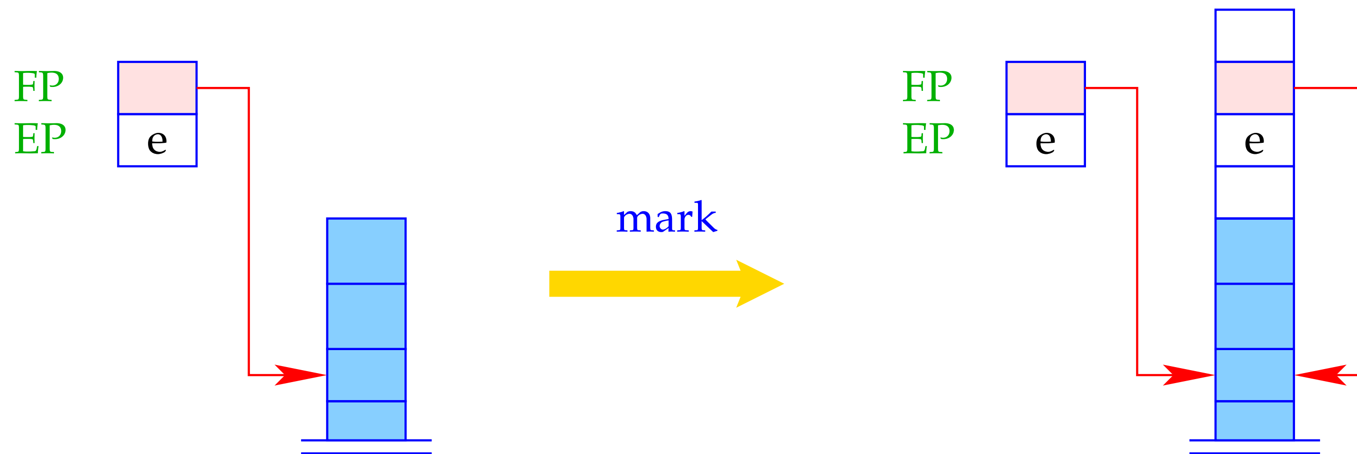


for (i = k-1; i≥0; i--)

S[SP+i] = S[S[SP]+i];

SP = SP+k–1;

Der Befehl mark legt Platz für Rückgabewert und organisatorische Zellen an und rettet FP und EP.

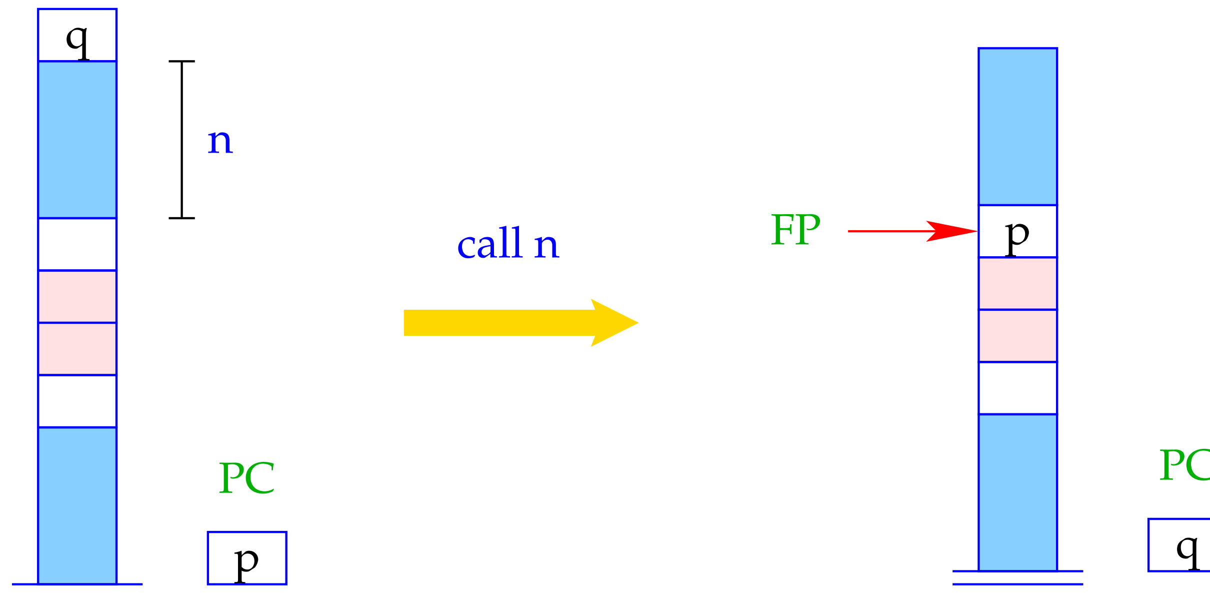


S[SP+2] = EP;

S[SP+3] = FP;

SP = SP + 4;

Der Befehl call n rettet die Fortsetzungs-Adresse und setzt FP, SP und PC auf die aktuellen Werte.



FP = SP - n - 1;

S[FP] = PC;

PC = S[SP];

SP--;

Entsprechend übersetzen wir eine Funktions-Definition:

$$\text{code} \quad t \text{ f } (specs)\{V_defs \quad ss\} \quad \rho \quad =$$

```
_f:     enter q         //  setzen des EP
        alloc k         //  Anlegen der lokalen Variablen
        code ss ρf
        return          //  Verlassen der Funktion
```

wobei q = $max + \text{k}$ wobei

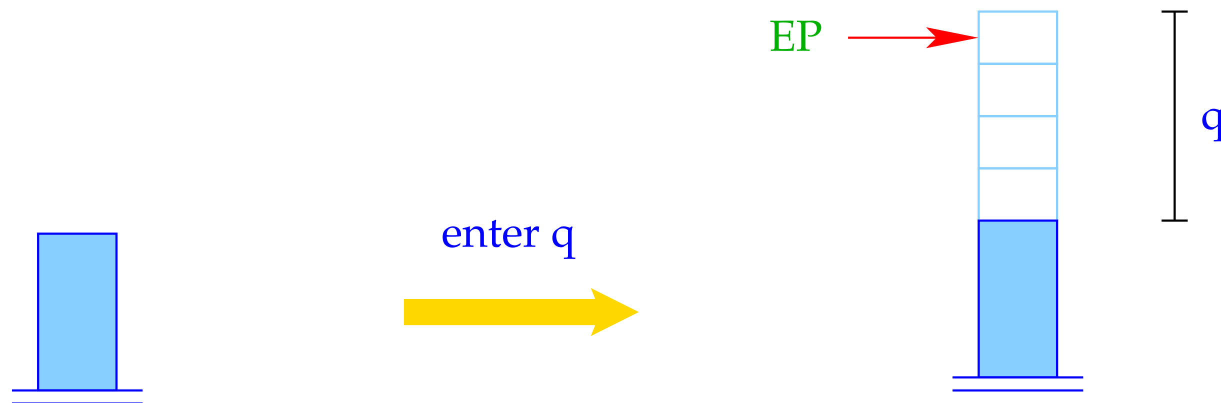
max = maximale Länge des lokalen Kellers

k = Platz für die lokalen Variablen

ρ_f = Adress-Umgebung für f

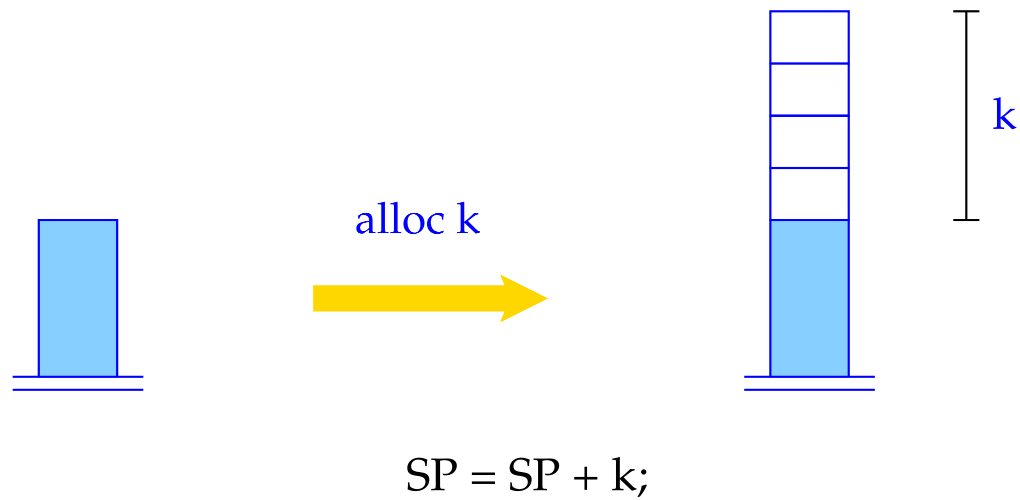
// *berücksichtigt specs, V_defs* und ρ

Der Befehl enter q setzt den EP auf den neuen Wert. Steht nicht mehr genügend Platz zur Verfügung, wird die Programm-Ausführung abgebrochen.



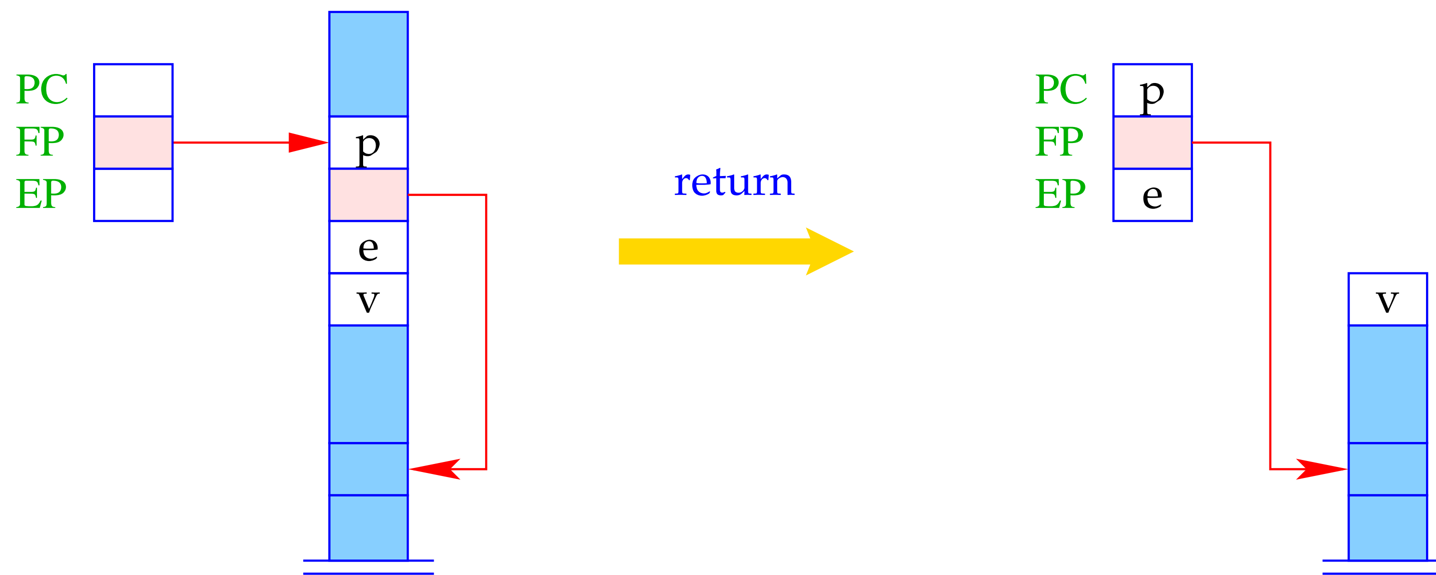

```
EP = SP + q;
if (EP ≥ NP)
    Error ("Stack Overflow");
```

Der Befehl alloc k reserviert auf dem Keller Platz für die lokalen Variablen.



SP = SP + k;

Der Befehl return gibt den aktuellen Keller-Rahmen auf. D.h. er restauriert die Register PC, EP und FP und hinterlässt oben auf dem Keller den Rückgabe-Wert.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error (“Stack Overflow”);

SP = FP-3; FP = S[SP+2];

9.4 Zugriff auf Variablen, formale Parameter und Rückgabe von Werten

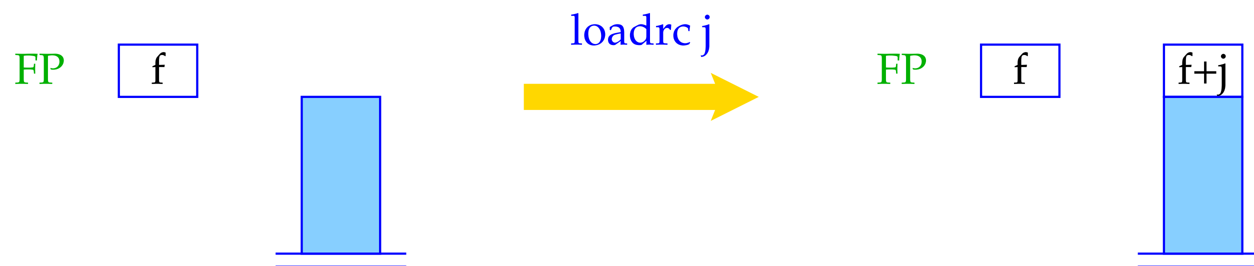
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir code_L für Variablen-Namen.

Für $\rho\, x = (tag, j)$ definieren wir

$$\text{code}_L\ x\ \rho = \begin{cases} \text{loadc}\ j & tag = G \\ \text{loadrc}\ j & tag = L \end{cases}$$

Der Befehl loadrc j berechnet die Summe von FP und j.



SP++;

S[SP] = FP+j;

Als Optimierung führt man analog zu loada j und storea j die Befehle loadr j und storer j ein:

```
loadr j        =  loadrc j
                  load

bla; storer j  =  loadrc j; bla
                  store
```

Der Code für **return** e; entspricht einer Zuweisung an eine Variable mit Relativadresse -3.

$$\text{code } \textbf{return}\, e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

storer -3

return

Beispiel: Für die Funktion

```
int fac (int x) {
   if (x ≤ 0) return 1;
   else return x ∗ fac (x − 1);
}
```

erzeugen wir:

```
_fac:   enter q       loadc 1       A:  loadr 1              mul
        alloc 0       storer -3         mark                 storer -3
        loadr 1       return            loadr 1              return
        loadc 0       jump B            loadc 1          B:  return
        leq                             sub
        jumpz A                         loadc _fac
                                        call 1
```

Dabei ist $\rho_{\text{fac}} : x \mapsto (L, 1)$ und $q = 1 + 6 = 7$.

10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

$$\text{SP} = -1 \qquad \text{FP} = \text{EP} = 0 \qquad \text{PC} = 0 \qquad \text{NP} = \text{MAX}$$

Sei $p \equiv V_defs \quad F_def_1 \ldots F_def_n,$ ein Programm, wobei F_def_i eine Funktion f_i definiert, von denen eine main heißt.

Der Code für das Programm p enthält:

- Code für die Funktions-Definitionen F_def_i;
- Code zum Anlegen der globalen Variablen;
- Code für den Aufruf von main();
- die Instruktion halt.

Dann definieren wir:

$$
\begin{array}{rrl}
\text{code}\ p\ \emptyset & = & \text{enter}\ (k+6) \\
 & & \text{alloc}\ (k+1) \\
 & & \text{mark} \\
 & & \text{loadc _main} \\
 & & \text{call}\ 0 \\
 & & \text{pop} \\
 & & \text{halt} \\
 & _f_1: & \text{code}\ F_def_1\ \rho \\
 & & \vdots \\
 & _f_n: & \text{code}\ F_def_n\ \rho
\end{array}
$$

wobei	$\emptyset$	$\widehat{=}$	leere Adress-Umgebung;
	ρ	$\widehat{=}$	globale Adress-Umgebung;
	k	$\widehat{=}$	Platz für globale Variablen

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:

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

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 ...);

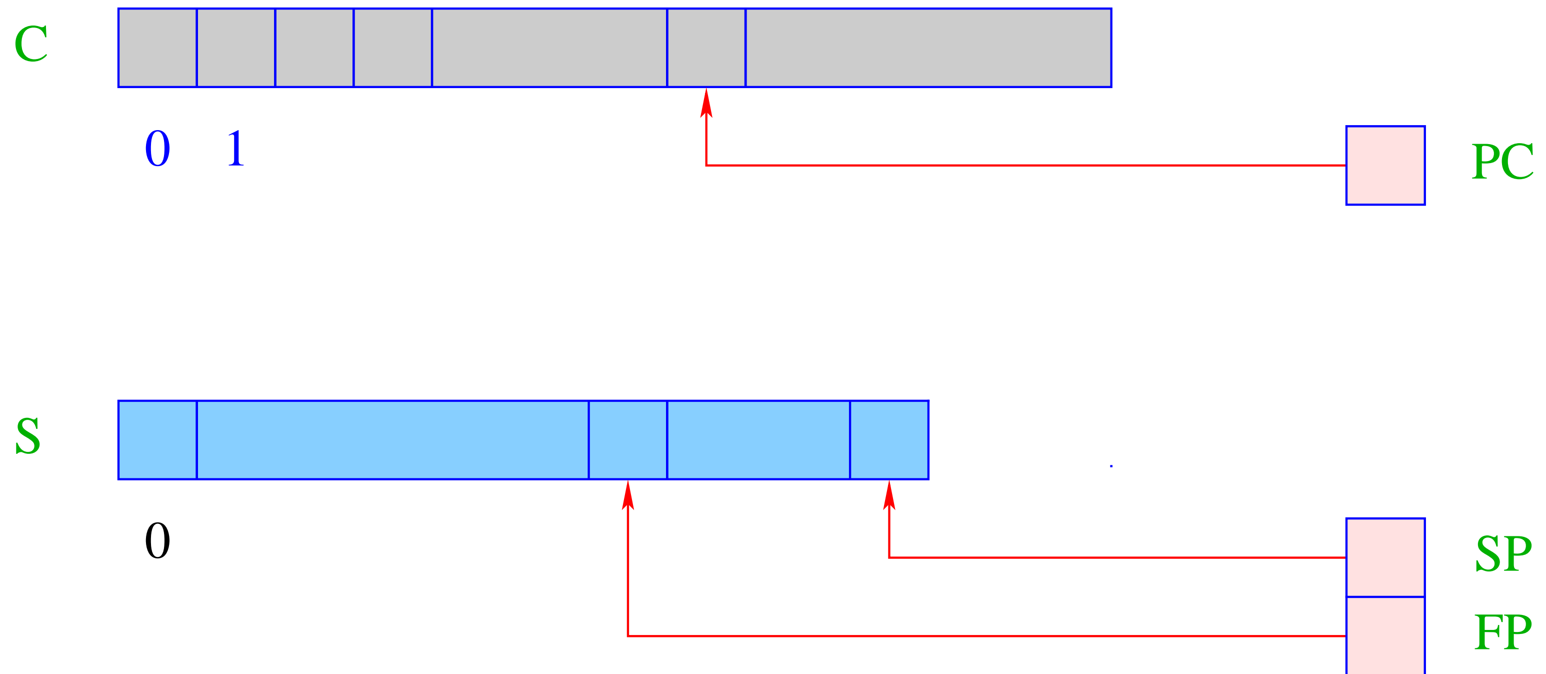
CBN: Call-by-Need – die aktuellen Parameter werden erst ausgewertet, wenn ihr Wert benötigt wird $\Longrightarrow$ spart manchmal Arbeit :-)

Beispiel:

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

- 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.

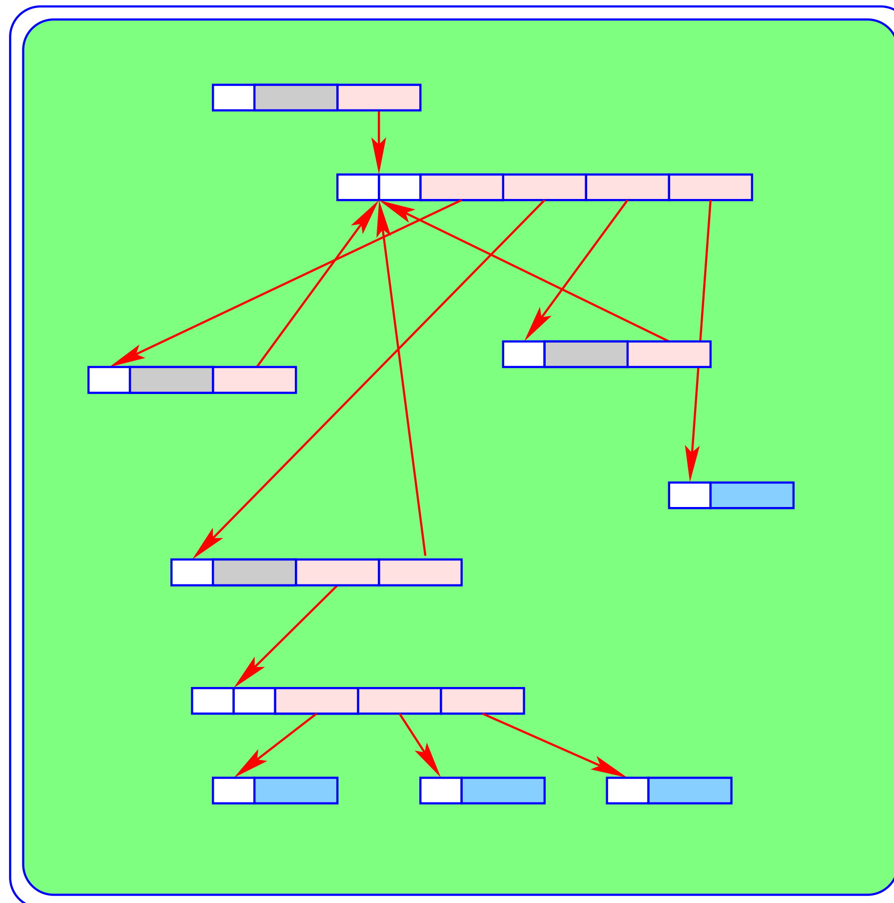
12 Architektur der MaMa:

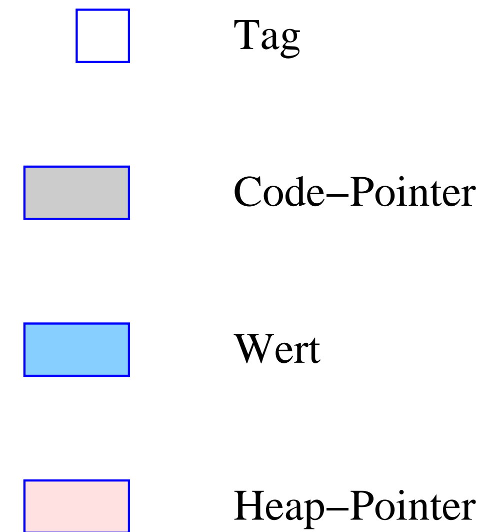


... das sind die uns bereits bekannten Datenstrukturen:

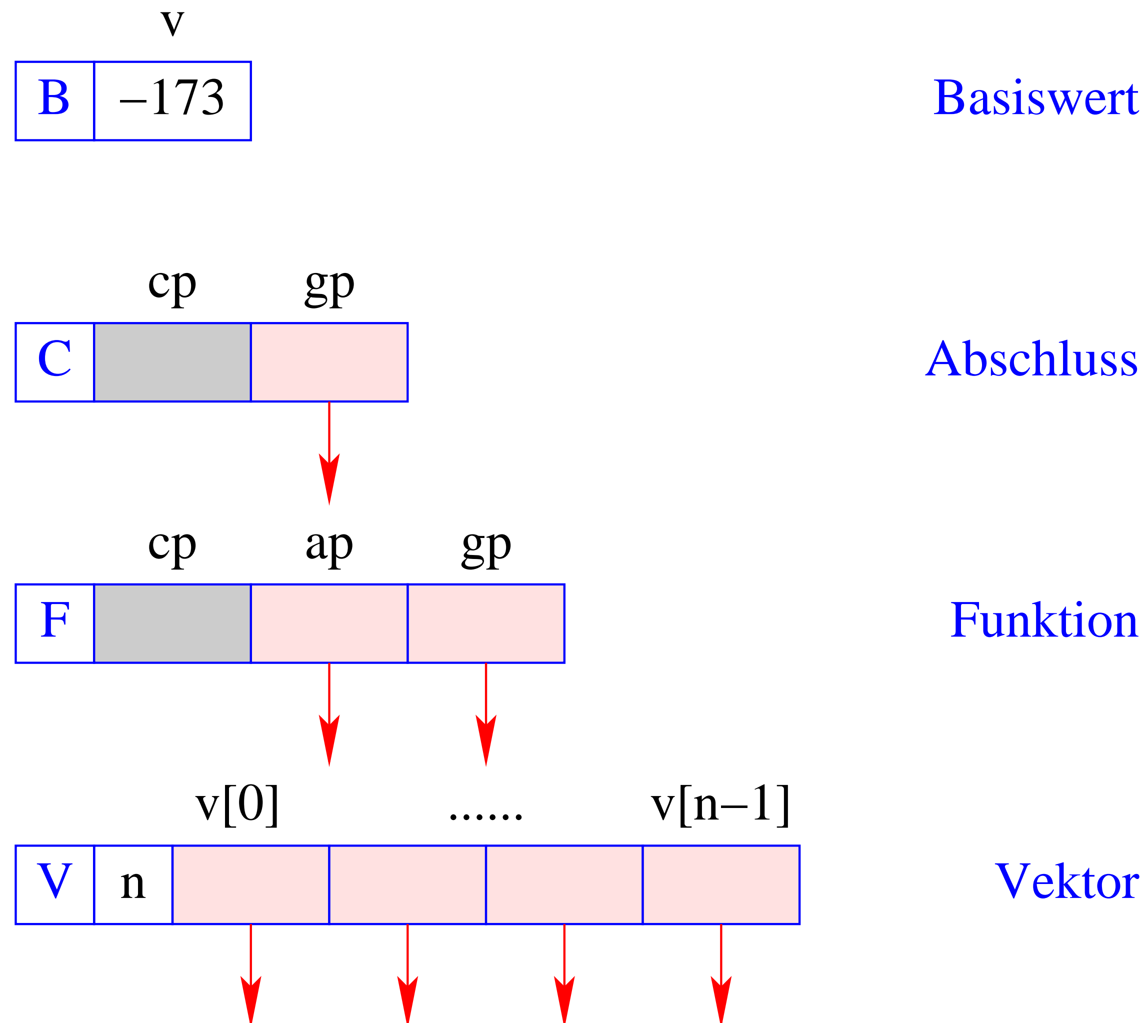
C	=	Code-Speicher – enthält MaMa-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter – zeigt auf nächsten auszuführenden Befehl;
S	=	Runtime-Stack;
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
SP	=	Stack-Pointer – zeigt auf oberste belegte Zelle;
FP	=	Frame-Pointer –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

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

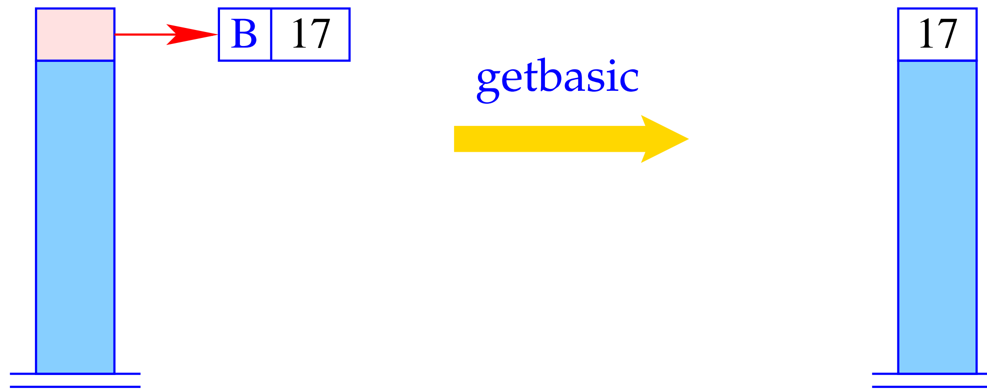
$$
\begin{array}{rrl}
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{kp} & = & \text{code}_B\ e_0\ \rho\ \text{kp} \\
 & & \text{jumpz A} \\
 & & \text{code}_B\ e_1\ \rho\ \text{kp} \\
 & & \text{jump B} \\
 & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{kp} \\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

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 $\square_1$ 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:

$$\begin{aligned} \text{code}_B\, e\, \rho\, \text{kp} \quad = \quad & \text{code}_V\, e\, \rho\, \text{kp} \\ & \text{getbasic} \end{aligned}$$

Dabei ist:



if (H[S[SP]] != (B,_))

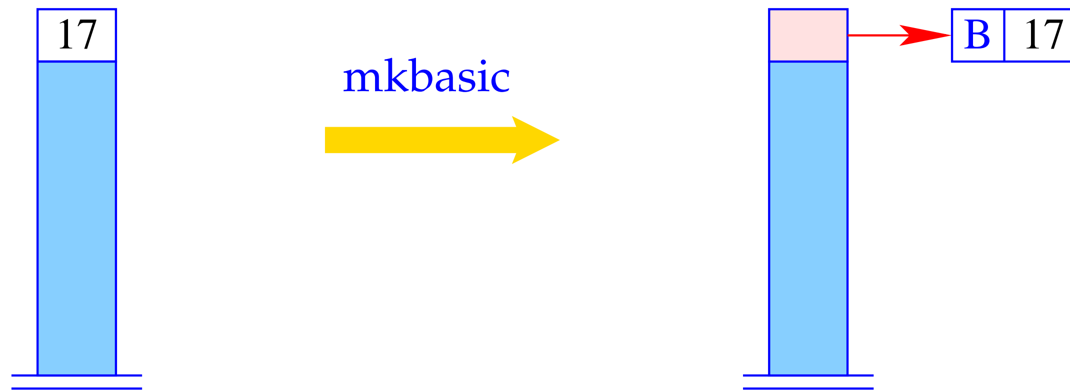
Error “not basic!”;

else

S[SP] = H[S[SP]].v;

Für code_V und einfache Ausdrücke finden wir analog:

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



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

14 Der Zugriff auf Variablen

Beispiel: Betrachte die Funktion f :

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

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.

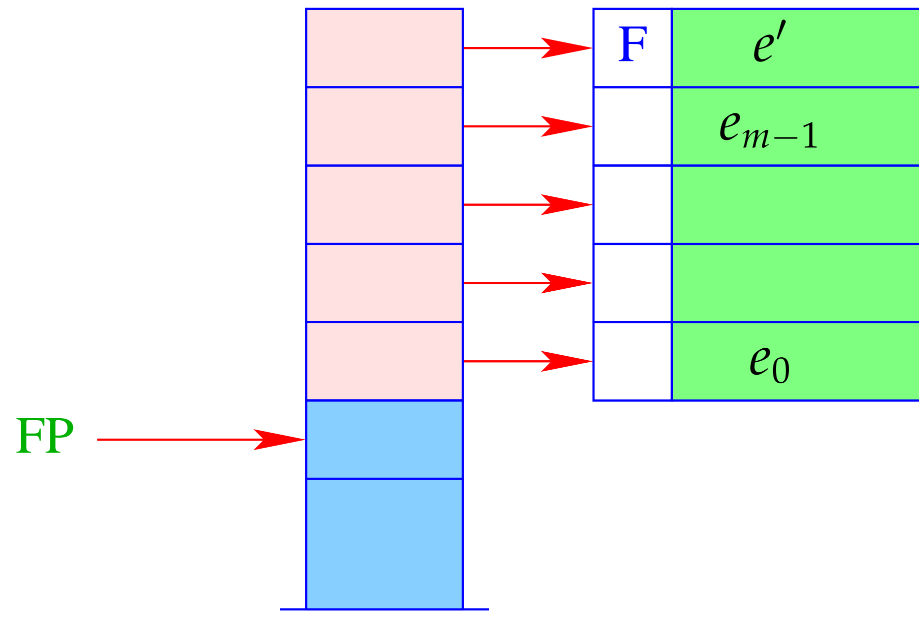
Adress-Umgebungen haben darum die Form:

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

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

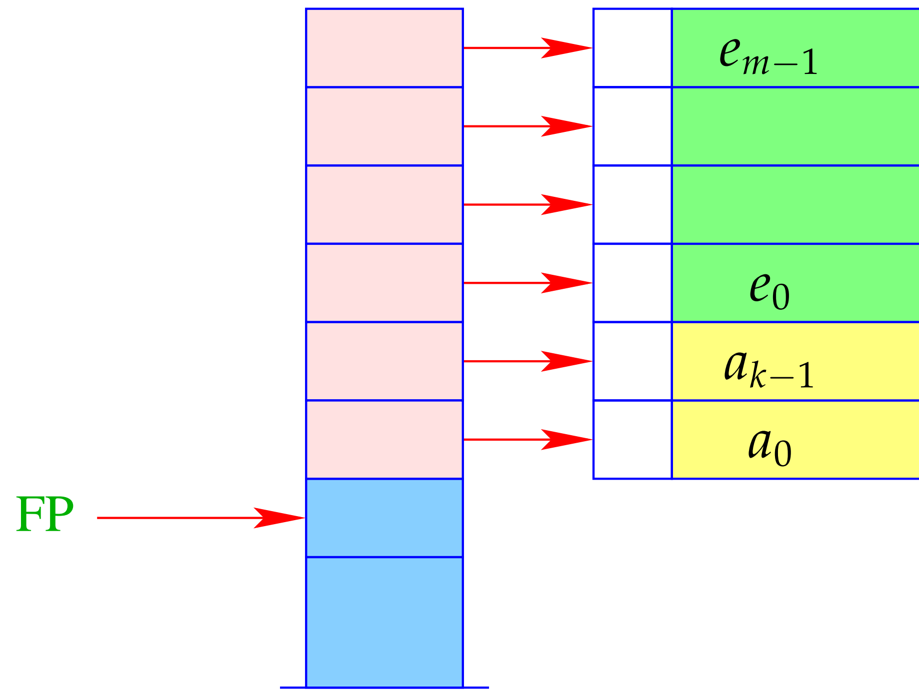
Sei $e \equiv e' \; e_0 \; \ldots \; e_{m-1}$ die Anwendung einer Funktion e' auf Argumente $e_0, \ldots, e_{m-1}$.

Mögliche Kellerorganisation:

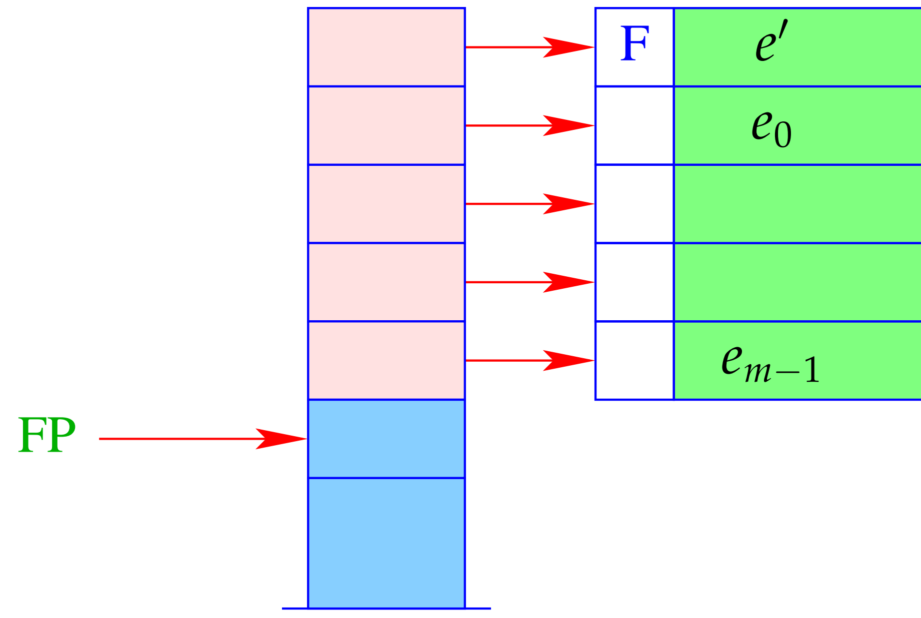


+ Adressierung der Parameter kann relativ zu FP erfolgen :-)

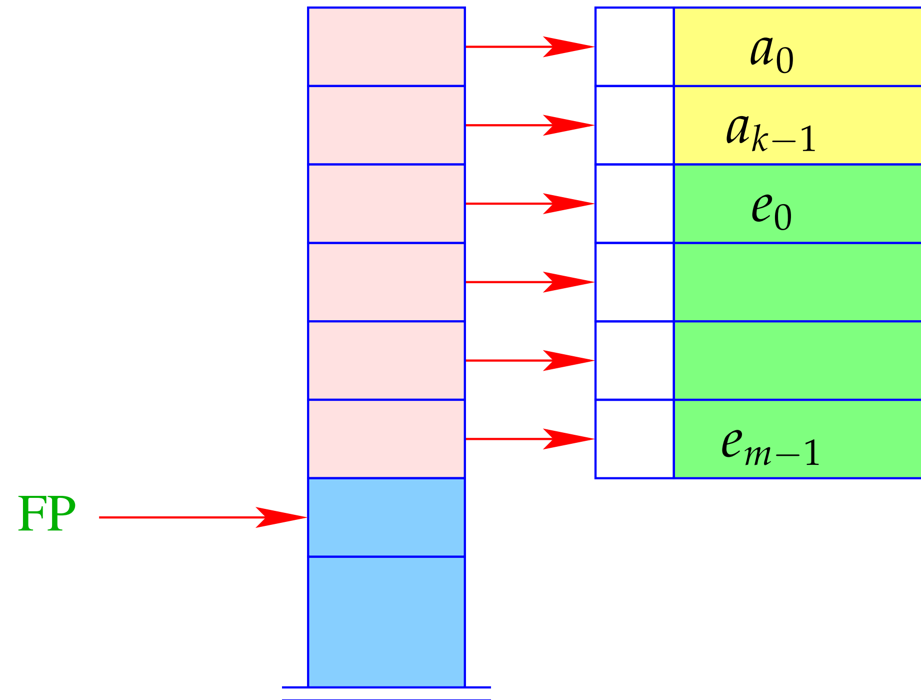
− Stellt sich heraus, dass sich e' zu einer Funktion evaluiert, die bereits partiell auf aktuelle Parameter $a_0, \ldots, a_{k-1}$ angewendet ist, müssen diese unterhalb von e_0 in den Keller hinein gefrickelt werden :-(



Alternative:



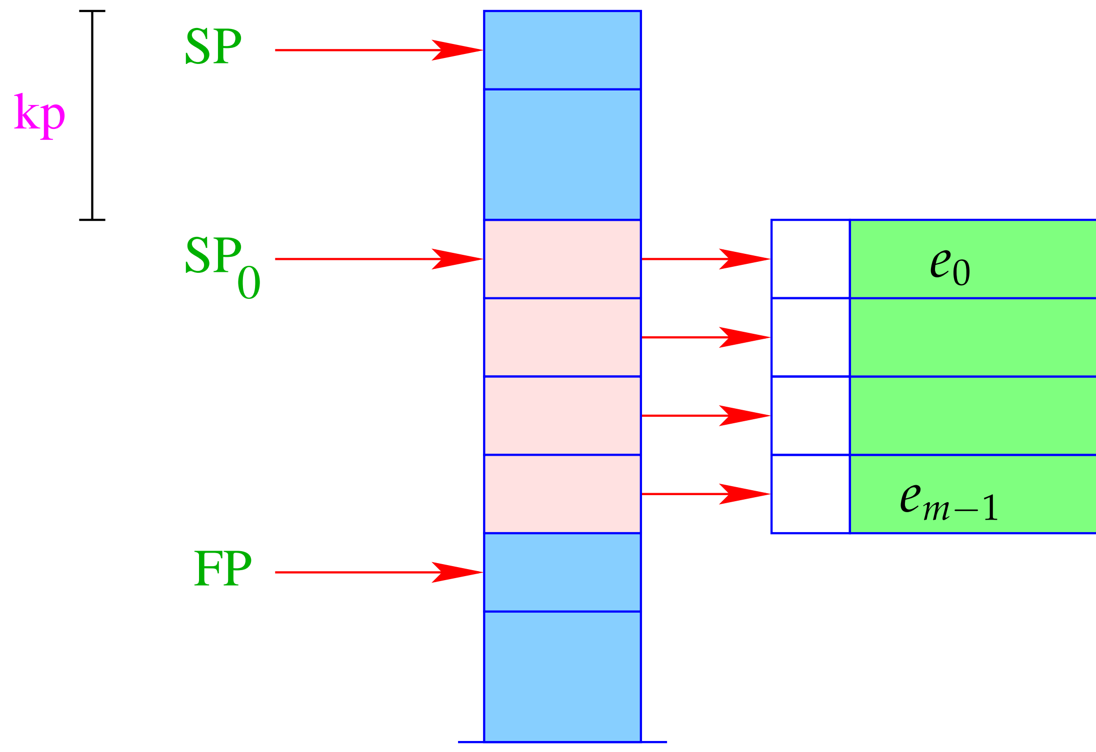
+ Die weiteren Argumente $a_0, \ldots, a_{k-1}$ wie auch die lokalen Variablen können einfach oben auf den Keller gelegt werden :-)



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

Ausweg:

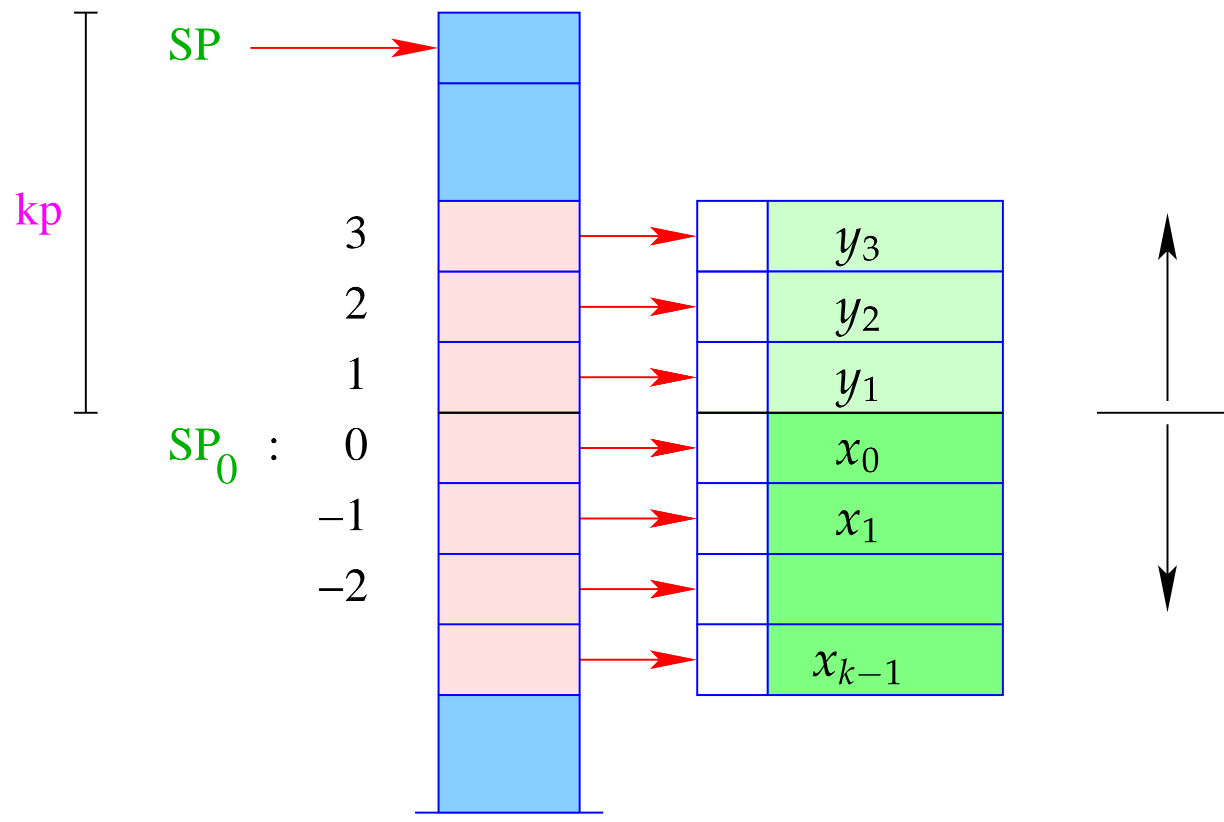
- Wir adressieren relativ zum Stackpointer SP !!!
- Leider ändert sich der Stackpointer während der Programm-Ausführung ...



- 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

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

- 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:

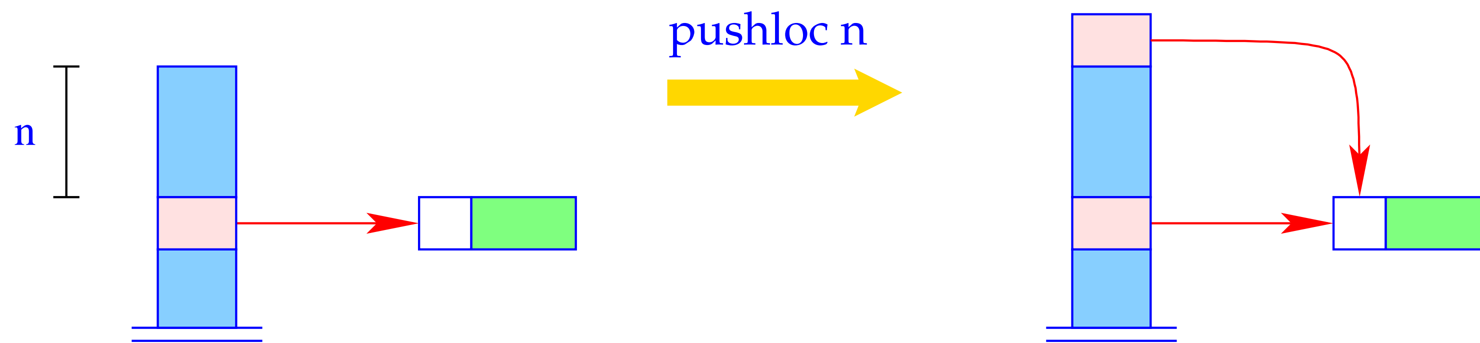
$$\begin{aligned} \text{code}_V\ x\ \rho\ \text{kp} \quad = \quad & \text{getvar}\ x\ \rho\ \text{kp} \\ & \text{eval} \end{aligned}$$

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

Bei CBV können wir eval einfach streichen.

Das Macro getvar ist definiert durch:

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



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

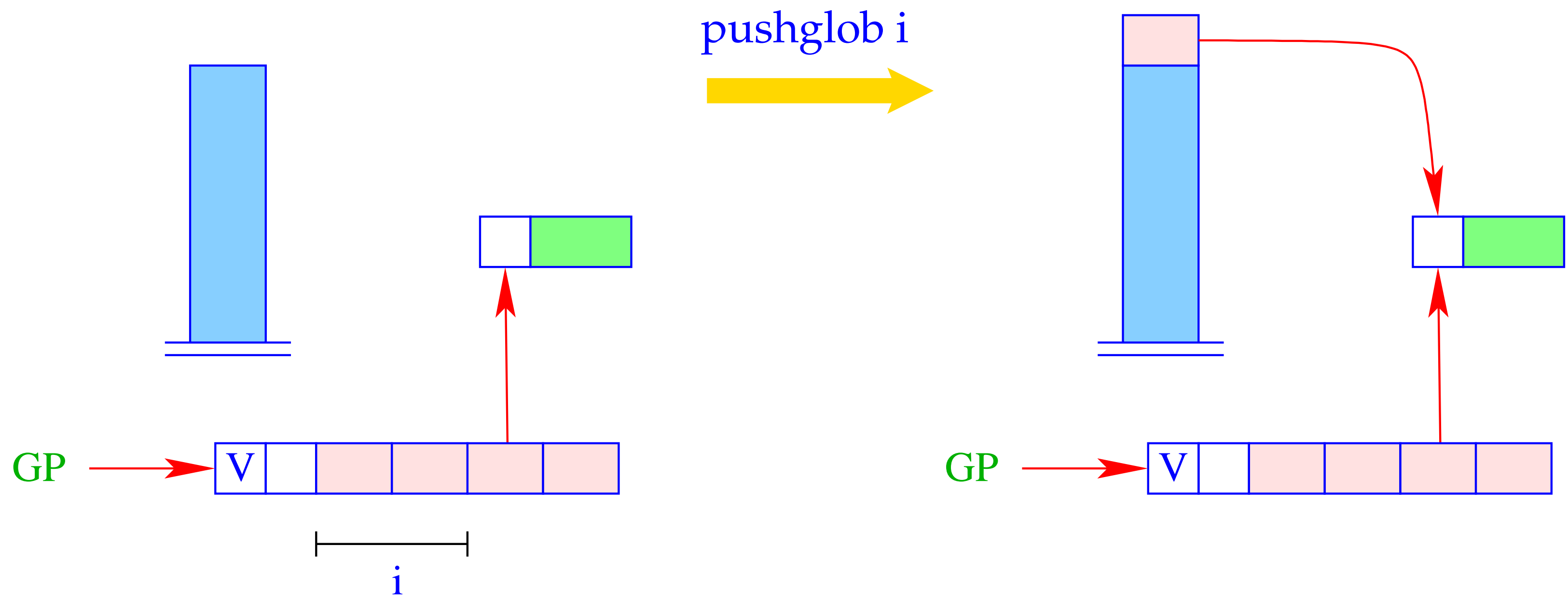
Zur Korrektheit:

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 kp $= 1$.

Dann ist für CBN:

$\text{code}_V\ e\ \rho\ 1$	$=$	getvar $b\ \rho$ 1	$=$	1	pushloc 0
		eval		2	eval
		getbasic		2	getbasic
		getvar $c\ \rho$ 2		2	pushglob 0
		eval		3	eval
		getbasic		3	getbasic
		add		3	add
		mkbasic		2	mkbasic

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:

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

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:

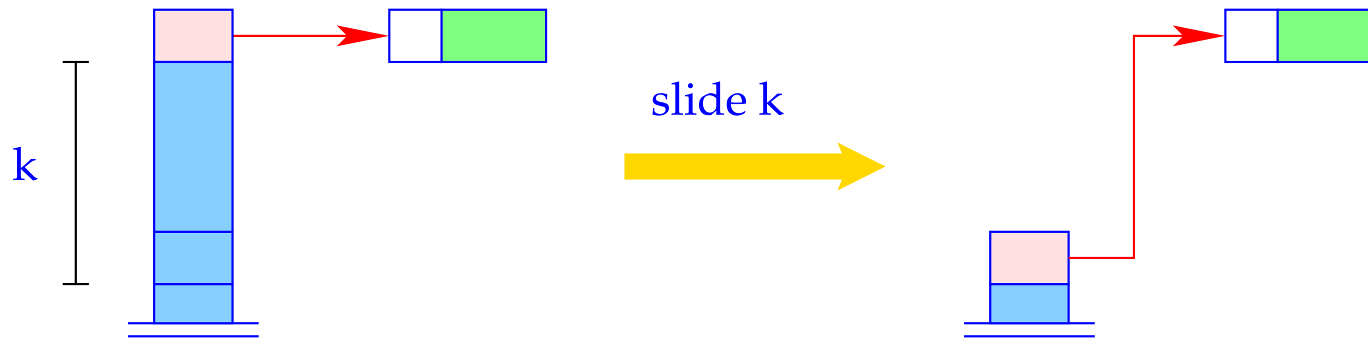
Betrachte den Ausdruck

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

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:

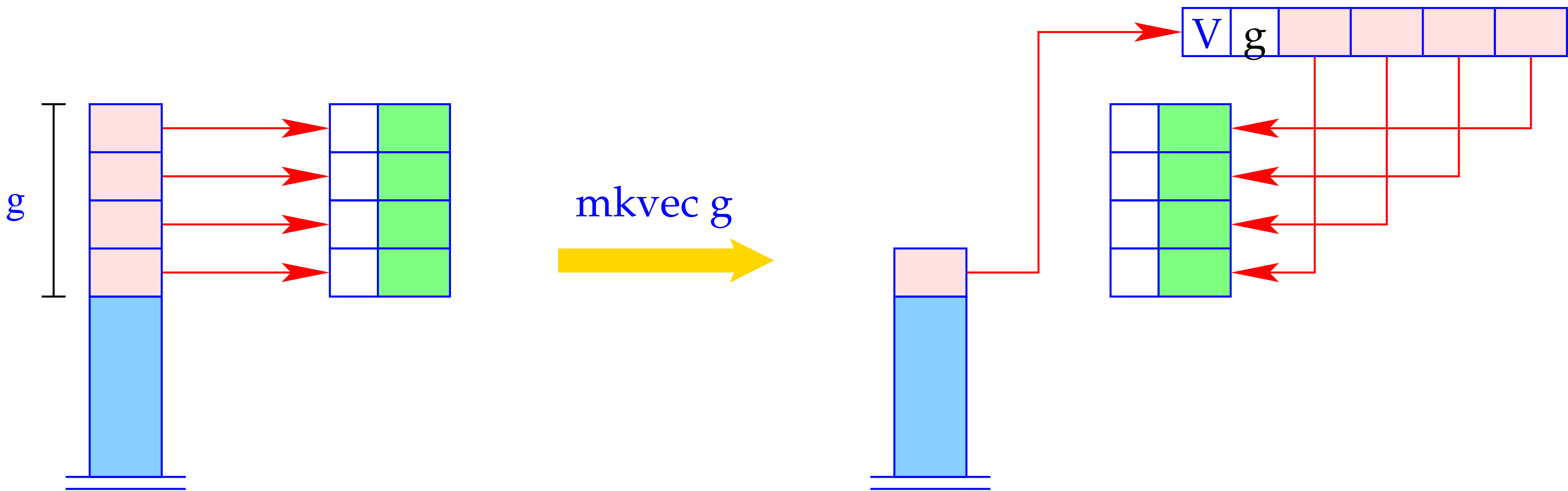
- Erzeugen des Global Vector mit den Bindungen der freien Variablen;
- Erzeugen eines (anfänglich leeren) Argument-Vektors;
- Erzeugen eines F-Objekts, das zusätzlich die Anfangs-Adresse des Codes zur Auswertung des Rumpfs enthält;
- Code zur Auswertung des Rumpfs.

Folglich:

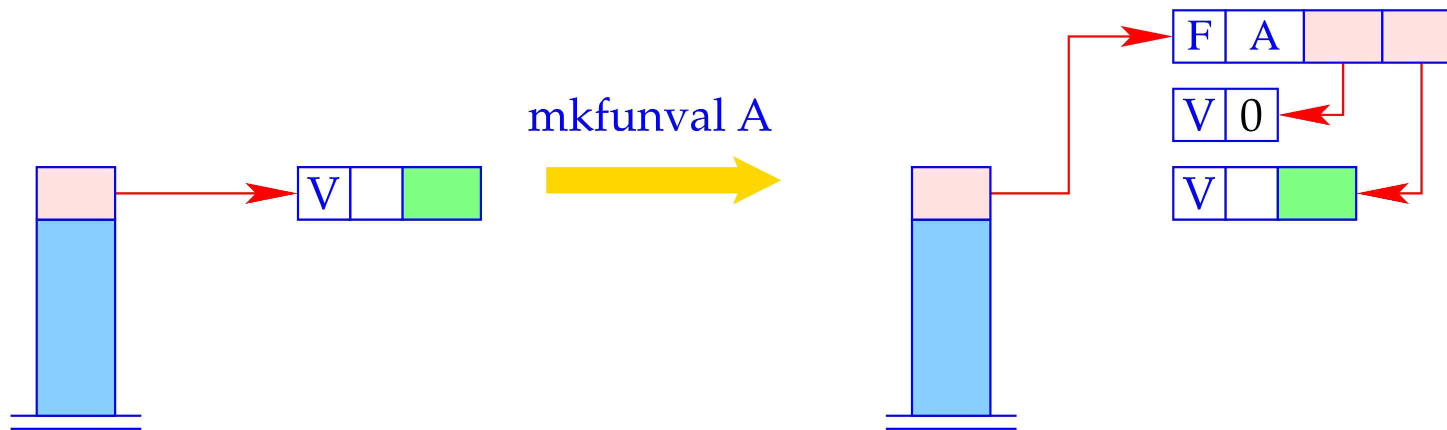
$$
\begin{array}{rrl}
\text{code}_V\ (\textbf{fn}\ x_0, \ldots, x_{k-1} \Rightarrow e)\ \rho\ \text{kp} & = & \text{getvar}\ z_0\ \rho\ \text{kp} \\
 & & \text{getvar}\ z_1\ \rho\ (\text{kp} + 1) \\
 & & \ldots \\
 & & \text{getvar}\ z_{g-1}\ \rho\ (\text{kp} + g - 1) \\
 & & \text{mkvec g} \\
 & & \text{mkfunval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

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

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

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 :	...

Die Geheimnisse um targ k und return k lüften wir später.

17 Funktionsanwendungen

Funktions-Anwendungen entsprechen Funktions-Aufrufen in C. Notwendige Aktionen zur Auswertung von $e' \; e_0 \; \ldots \; e_{m-1}$ sind:

- Anlegen eines Kellerrahmens;
- Parameter-Übergabe, d.h. bei:
  CBV: Auswerten der aktuellen Parameter;
  CBN: Anlegen von Abschlüssen für die aktuellen Parameter;
- Auswerten der Funktion e' zu einem F-Objekt;
- Anwenden der Funktion.

Folglich für CBN:

$$
\begin{array}{rcll}
\text{code}_V\ (e'\ e_0\ \ldots\ e_{m-1})\ \rho\ \text{kp} & = & \text{mark A} & //\ \text{Anlegen des Rahmens} \\
 & & \text{code}_C\ e_{m-1}\ \rho\ (\text{kp} + 3) & \\
 & & \text{code}_C\ e_{m-2}\ \rho\ (\text{kp} + 4) & \\
 & & \ldots & \\
 & & \text{code}_C\ e_0\ \rho\ (\text{kp} + m + 2) & \\
 & & \text{code}_V\ e'\ \rho\ (\text{kp} + m + 3) & //\ \text{Auswerten der Funktion} \\
 & & \text{apply} & //\ \text{entspricht call} \\
 & A: & \ldots &
\end{array}
$$

Wenn wir CBV implementieren wollen, müssen die Argumente vor dem Funktions-Aufruf ausgewertet werden.

Dann benutzen wir code_V anstelle von code_C für die Argumente e_i :-)

Beispiel:

Für $(f\ 42)$, $\rho = \{f \mapsto (L, 2)\}$ und kp $= 2$ liefert das bei CBV:

2	mark A	6	mkbasic	7		apply
5	loadc 42	6	pushloc 4	3	A :	...

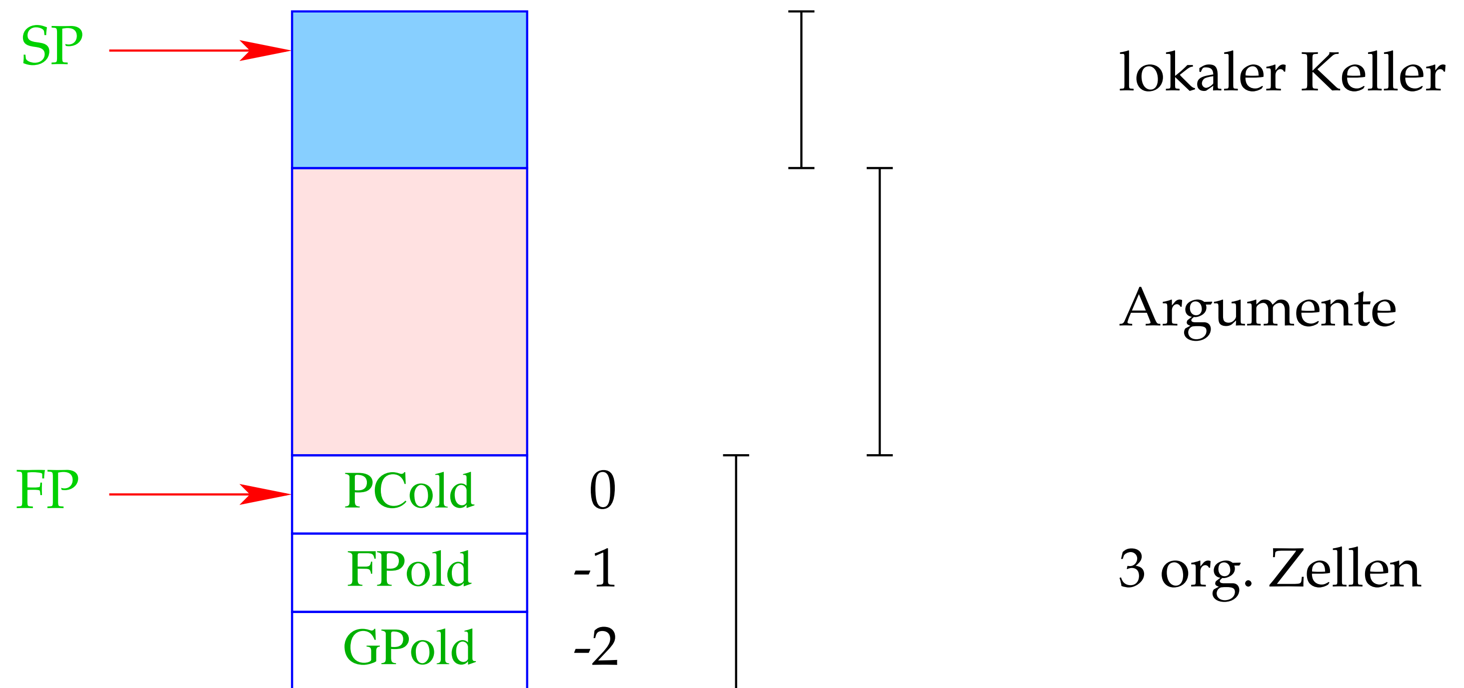
Ein etwas größeres Beispiel:

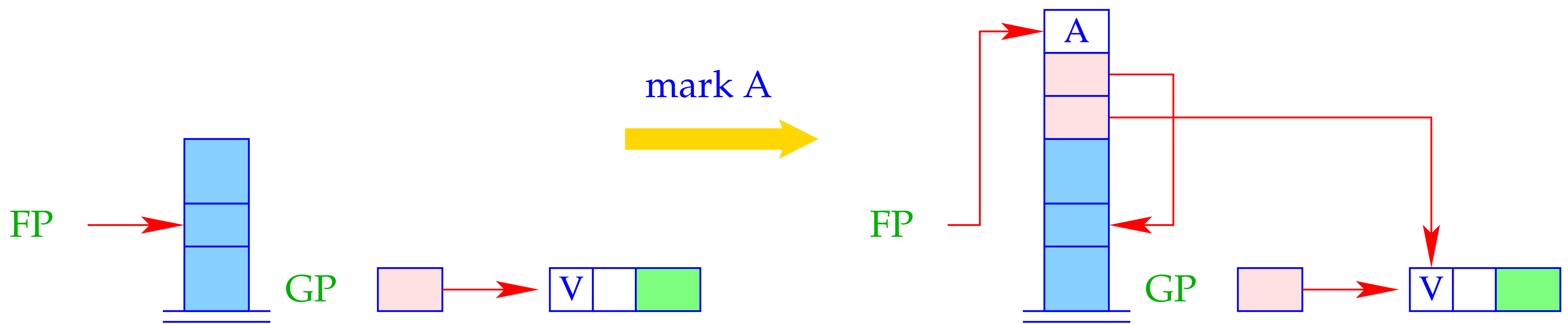
$$\textbf{let } a = 17;\ f = \textbf{fn } b \Rightarrow a + b \textbf{ in } f\ 42$$

Bei CBV erhalten wir für $\;\text{kp} = 0\;$:

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

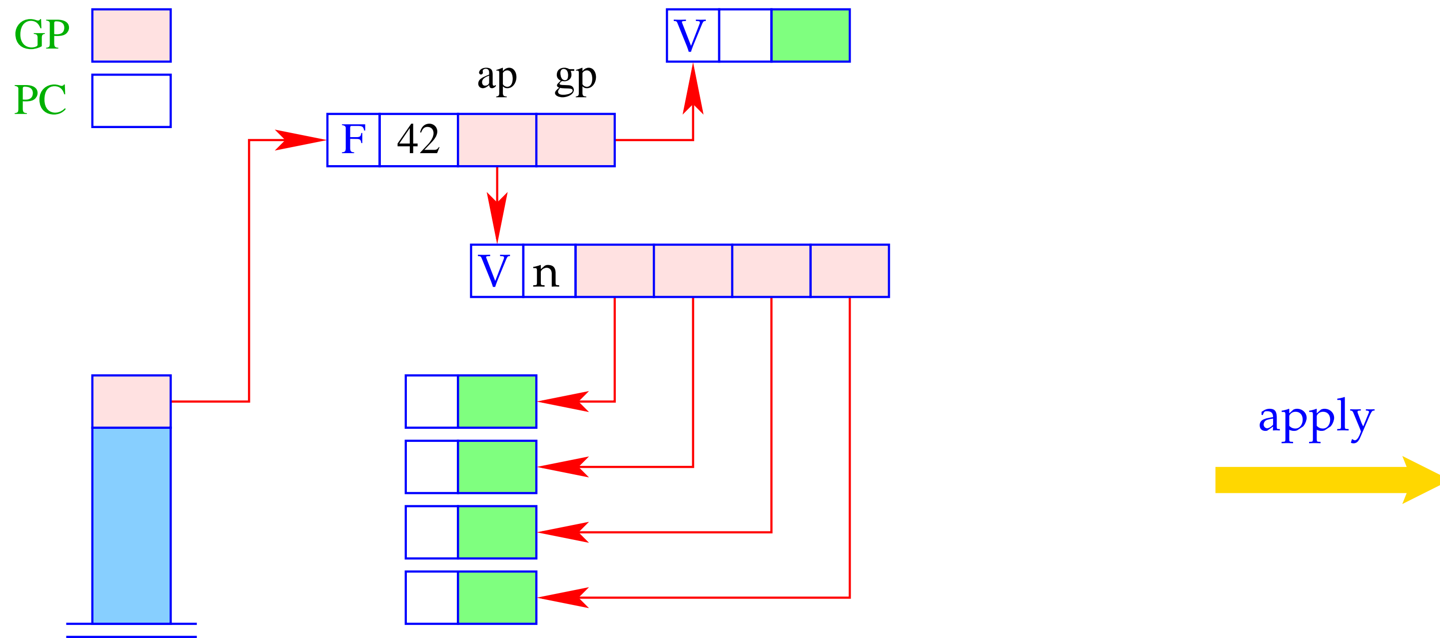
Vor der Implementierung der neuen Instruktionen müssen wir die Organisation eines Kellerrahmens festlegen:





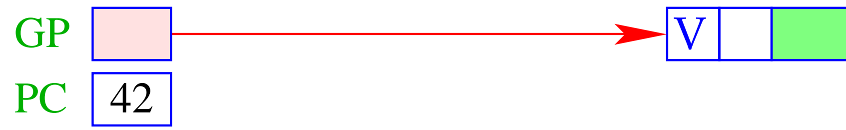
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = A;
FP = SP = SP + 3;

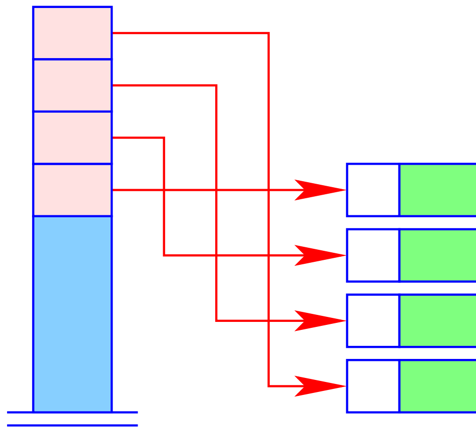
- Im Gegensatz zur CMa rettet hier bereits der mark-Befehl die Adresse, an der die Programm-Ausführung nach der Abarbeitung der Funktions-Anwendung fortfahren soll.
- Der apply-Befehl muss das F-Objekt, auf das (hoffentlich) oben auf dem Keller ein Verweis liegt, auspacken und an der dort angegebenen Adresse fortfahren.




```
h = S[SP];
if (H[h] != (F,_,_))
    Error "no fun";
else {
```

```
GP = h→gp; PC = h→cp;
for (i=0; i< h→ap→n; i++)
    S[SP+i] = h→ap→v[i];
SP = SP + h→ap→n – 1;
}
```


GP
PC
42
V



Achtung!

- Das 0-te Element des Argument-Vektors legen wir zuerst auf den Keller. Dieses muss also die äußerste Argument-Referenz darstellen.
- Das müssen wir berücksichtigen, wenn wir die Argumente einer unterversorgten Funktions-Anwendung zu einem F-Objekt einpacken!!!

18 Unter- und Überversorgung mit Argumenten

Der erste Befehl, der nach einem apply ausgeführt wird, ist targ k.

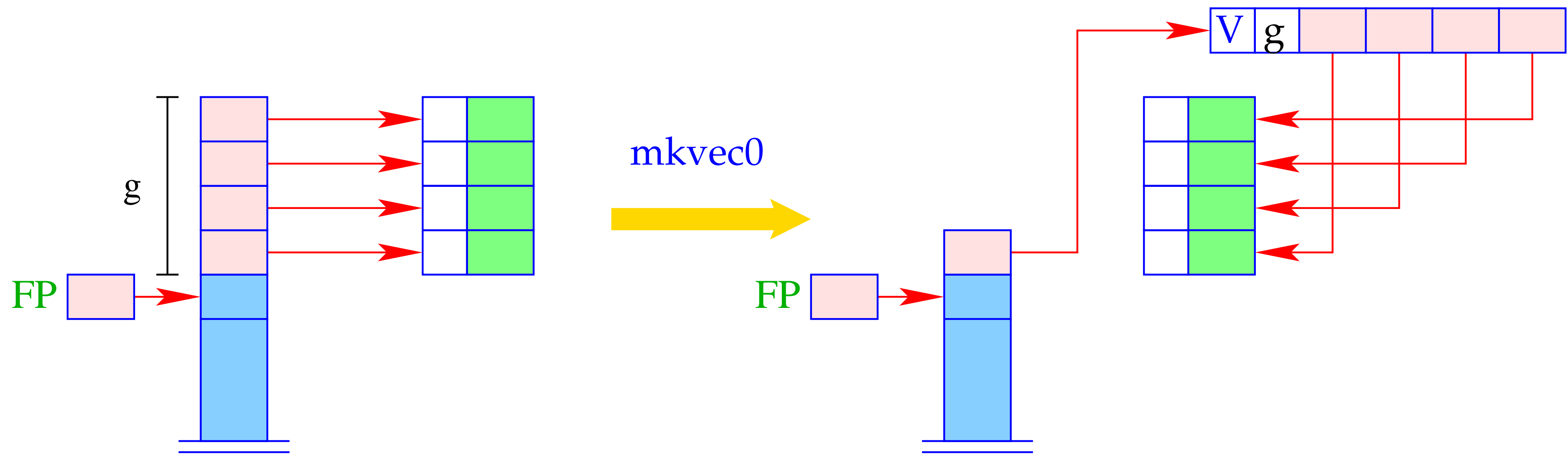
Er überprüft, ob bereits genügend Argumente vorliegen, um den Funktions-Rumpf auszuführen. Die Anzahl der Argumente ist: $SP - FP$.

Sind nicht genügend Argumente vorhanden, wird als Ergebnis ein neues F-Objekt zurückgeliefert. Andernfalls soll der Rumpf normal betreten werden.

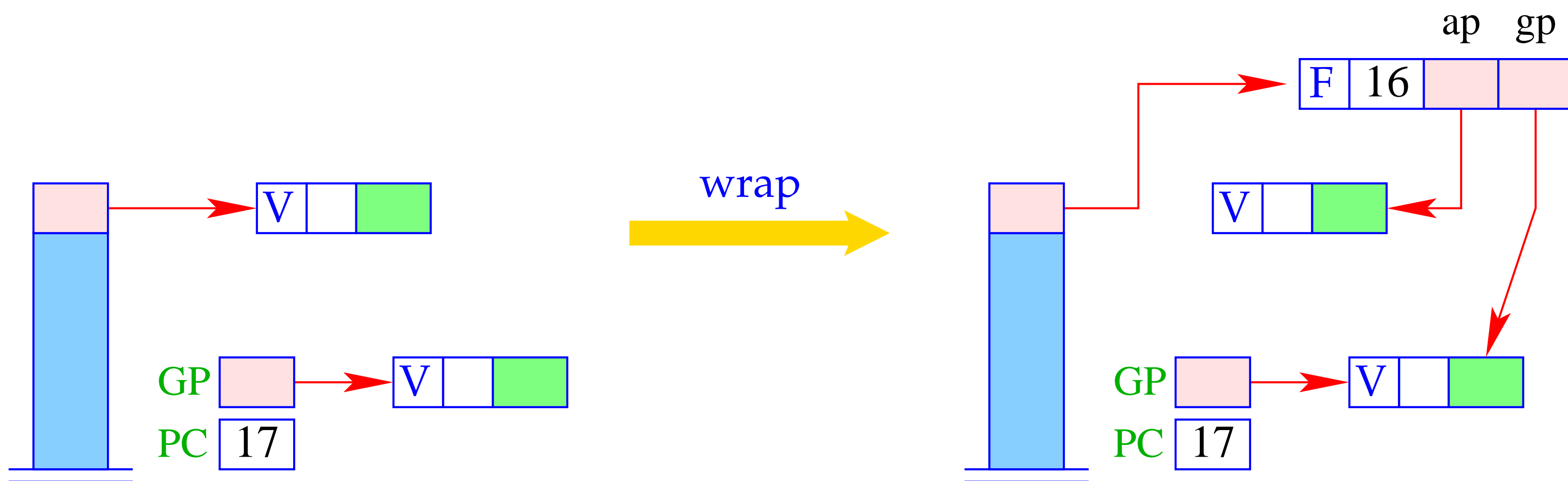
targ k ist ein komplizierter Befehl. Darum zerlegen wir seine Ausführung in mehrere Schritte:

$$
\begin{array}{lll}
\text{targ } k & = & \text{if } (SP - FP < k) \, \{ \\
& & \quad \text{mkvec0}; \qquad // \text{ Anlegen des Argument} - \text{Vektors} \\
& & \quad \text{wrap}; \qquad // \text{ Anlegen des F} - \text{Objekts} \\
& & \quad \text{popenv}; \qquad // \text{ Aufgeben des Kellerrahmens} \\
& & \}
\end{array}
$$

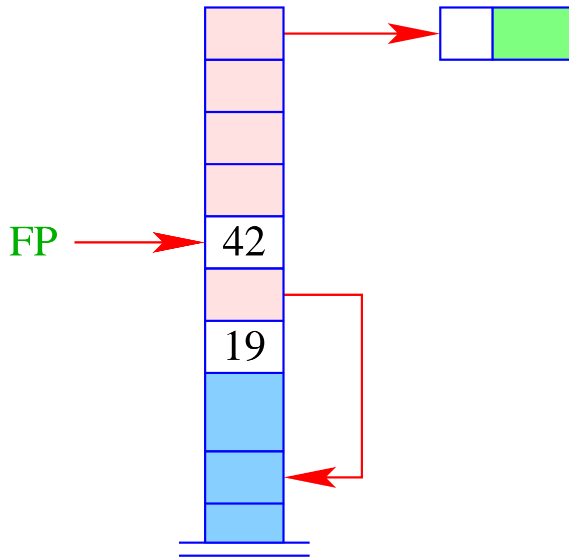
Die Zusammenfassung dieser festen Schritt-Abfolge zu einem Befehl kann als eine Art Optimierung verstanden werden :-)



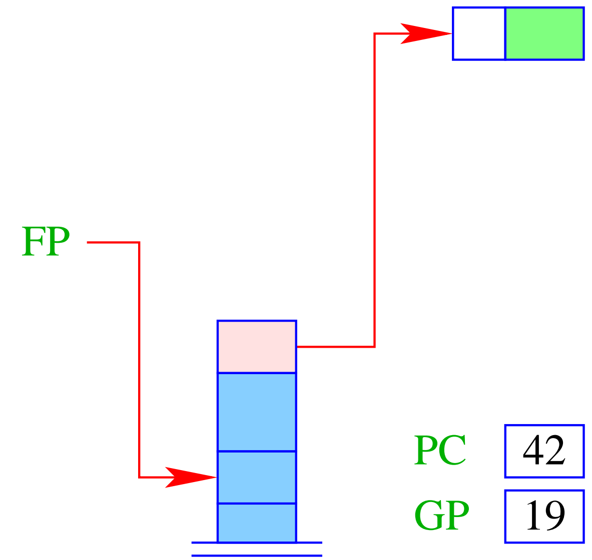

```
g = SP–FP; h = new (V, g);
SP = FP+1;
for (i=0; i<g; i++)
    h→v[i] = S[SP + i];
S[SP] = h;
```



S[SP] = new (F, PC–1, S[SP], GP);

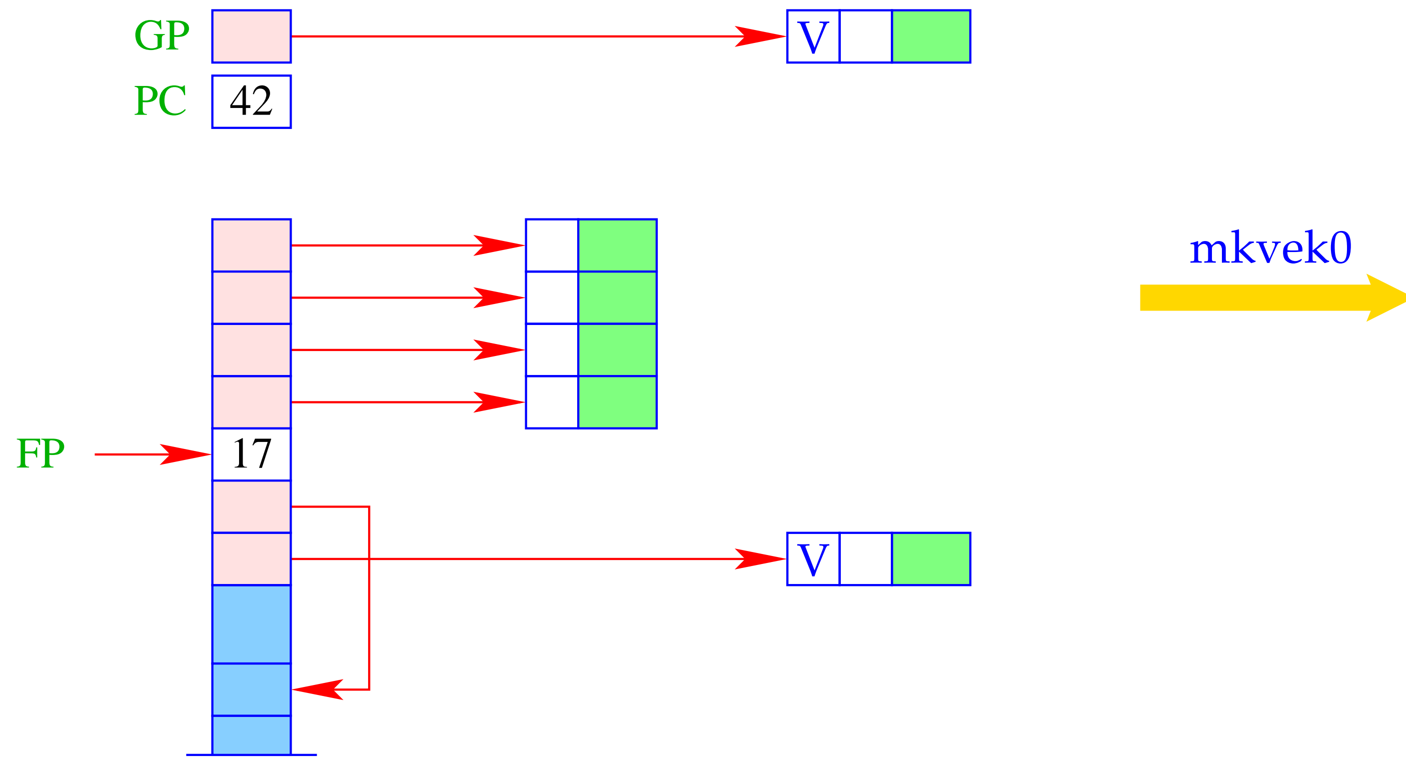


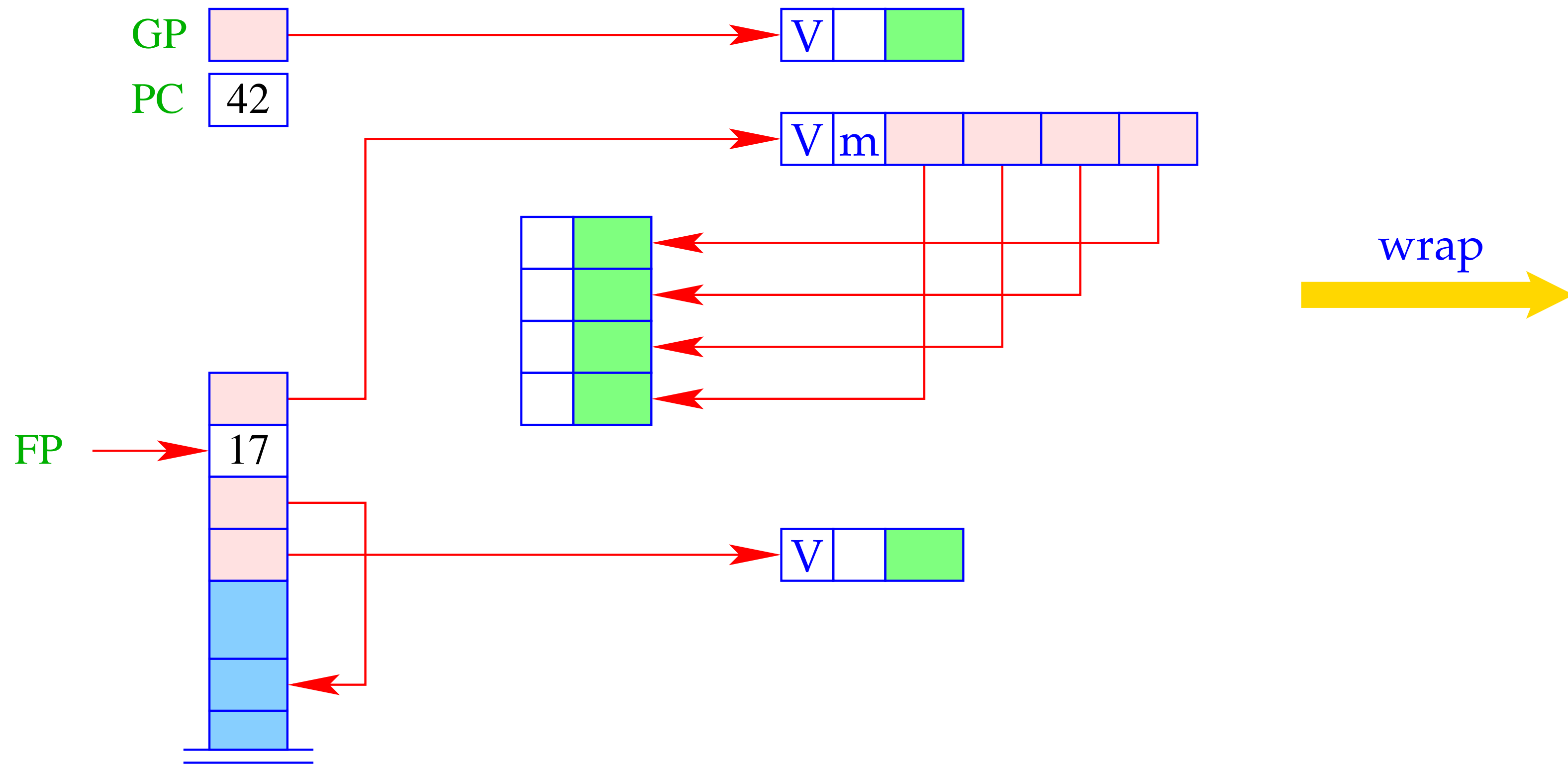
popenv



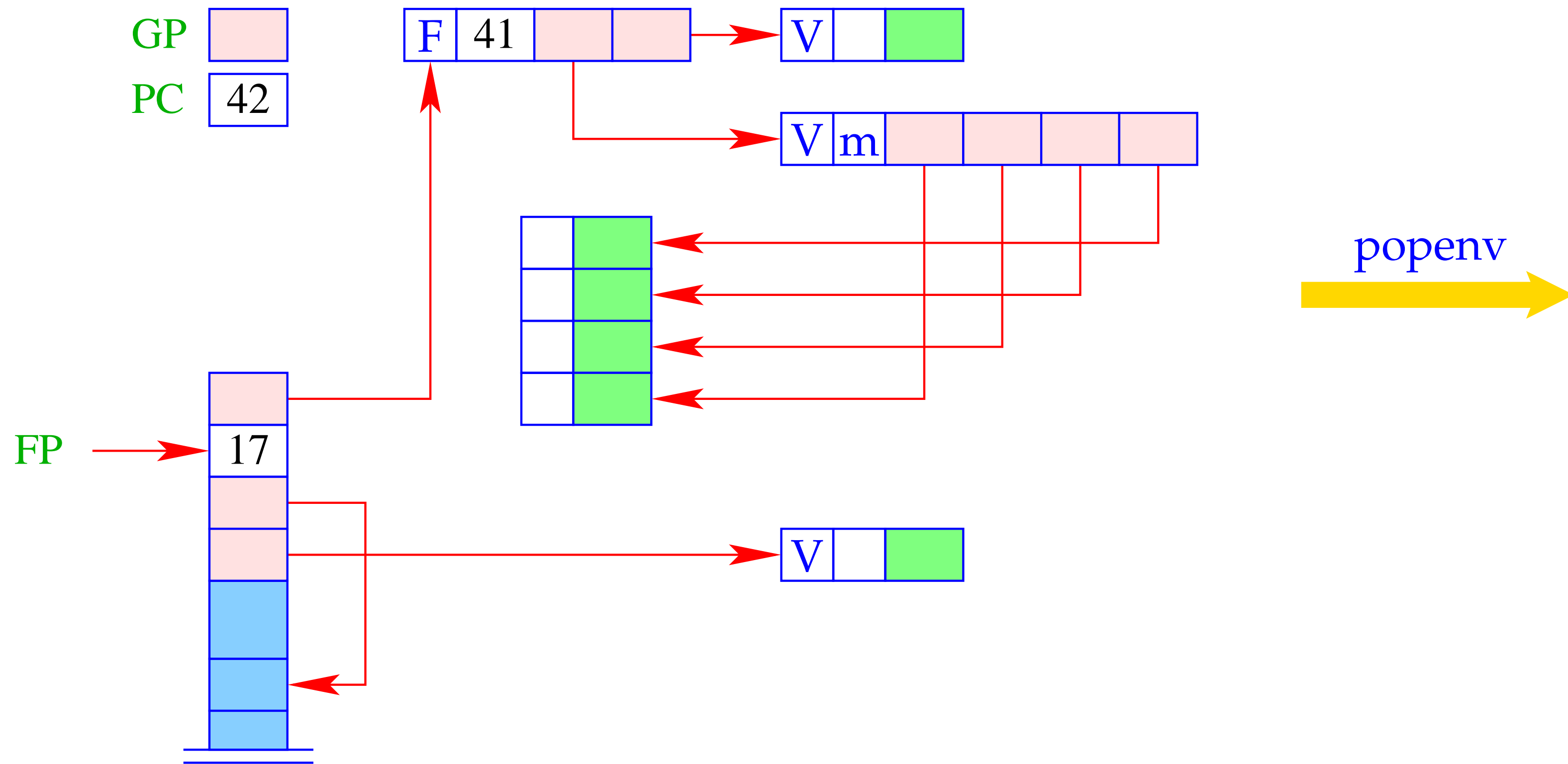
GP = S[FP-2];
S[FP-2] = S[SP];
PC = S[FP];
SP = FP - 2;
FP = S[FP-1];

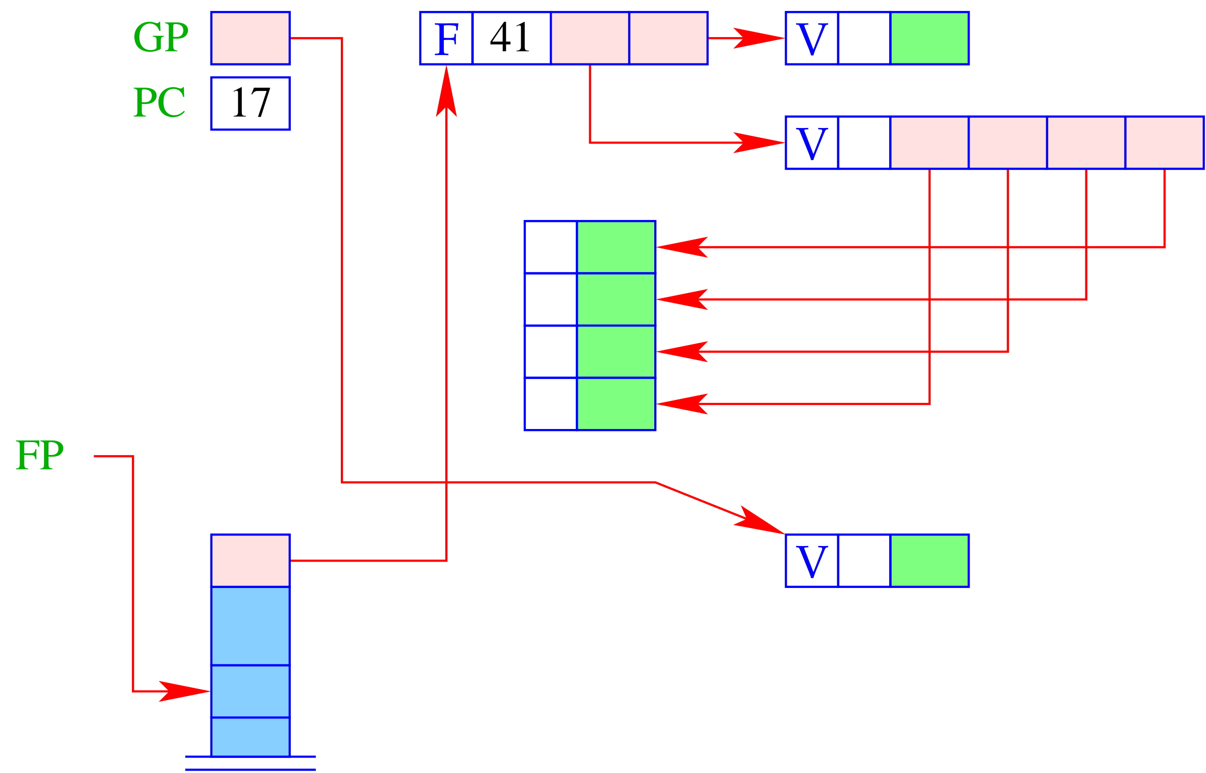
Insgesamt erhalten wir damit für targ k:





GP
PC
42
V
V
m
wrap
FP
17
V

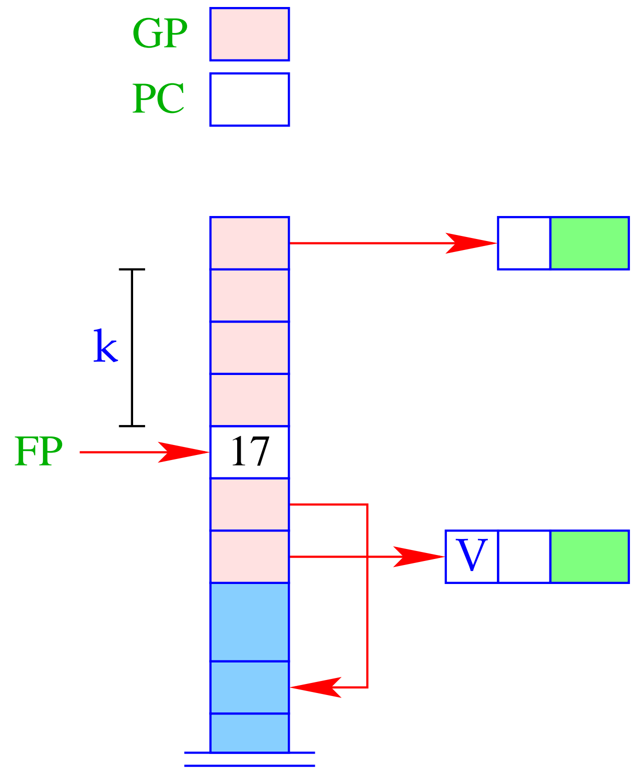

GP
PC
42
F
41
V
V
m
popenv
FP
17
V


GP
PC
17
F
41
V
V
FP
V

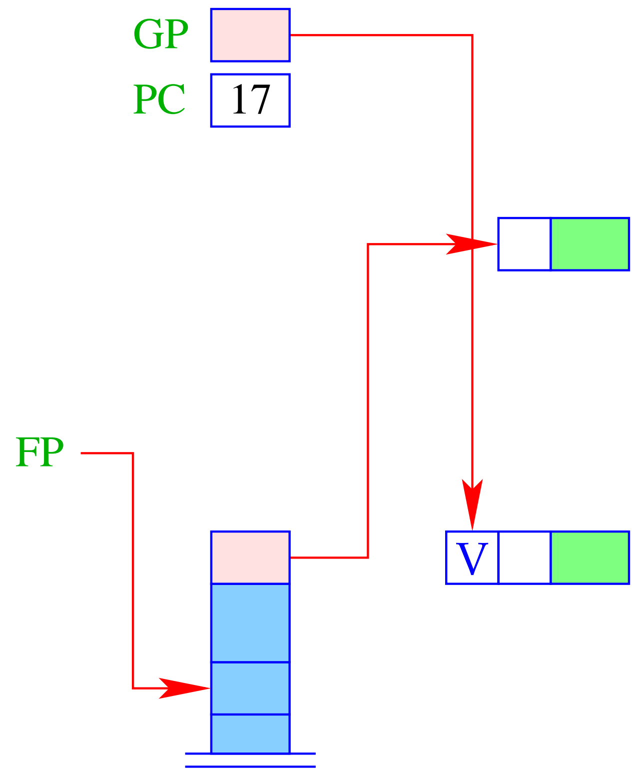
- Liegt exakt die richtige Argument-Anzahl vor, kann nach Abarbeitung des Rumpfs der Kellerrahmen aufgegeben werden.
- Liegt sogar Überversorgung mit Argumenten vor, muss der Rumpf sich offenbar erneut zu einer Funktion ausgewertet haben, die nun die restlichen Argumente konsumiert ...
- Für diese Überprüfung ist return k zuständig:

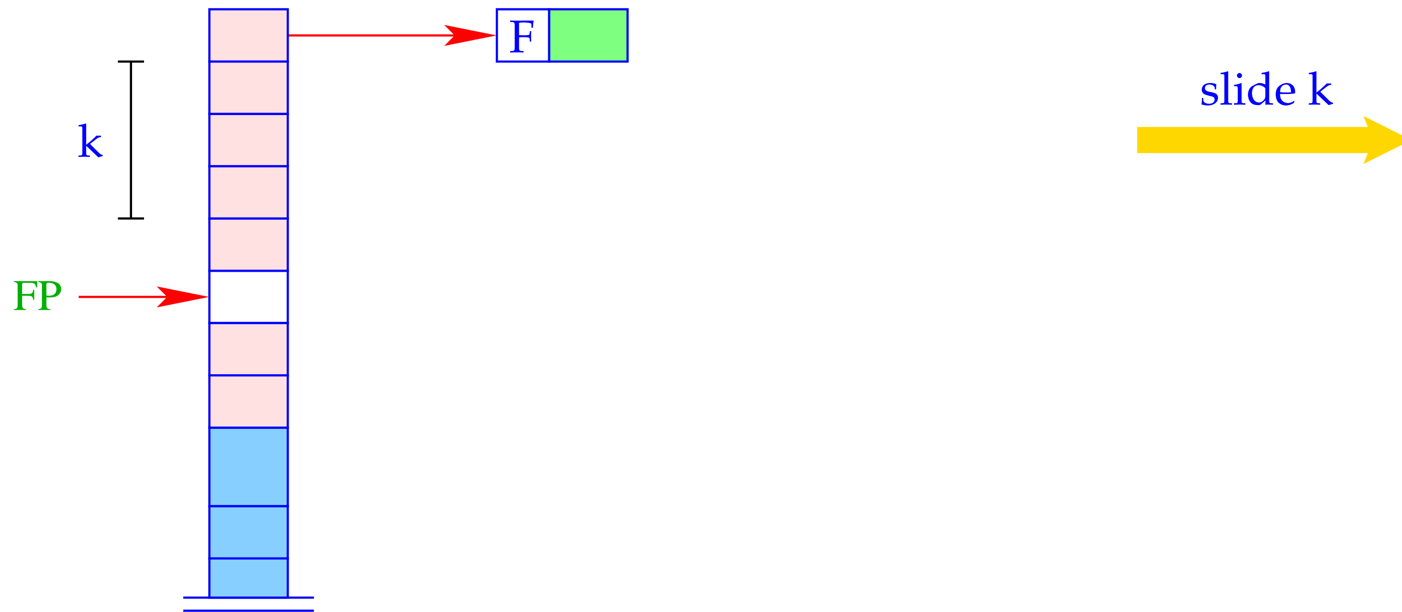
```
return k  =  if (SP − FP ≡ k + 1)
                popenv;           // Aufgeben des Kellerrahmens
             else {               // Es gibt noch weitere Argumente
                slide k;
                apply;            // erneuter Aufruf
             }
```

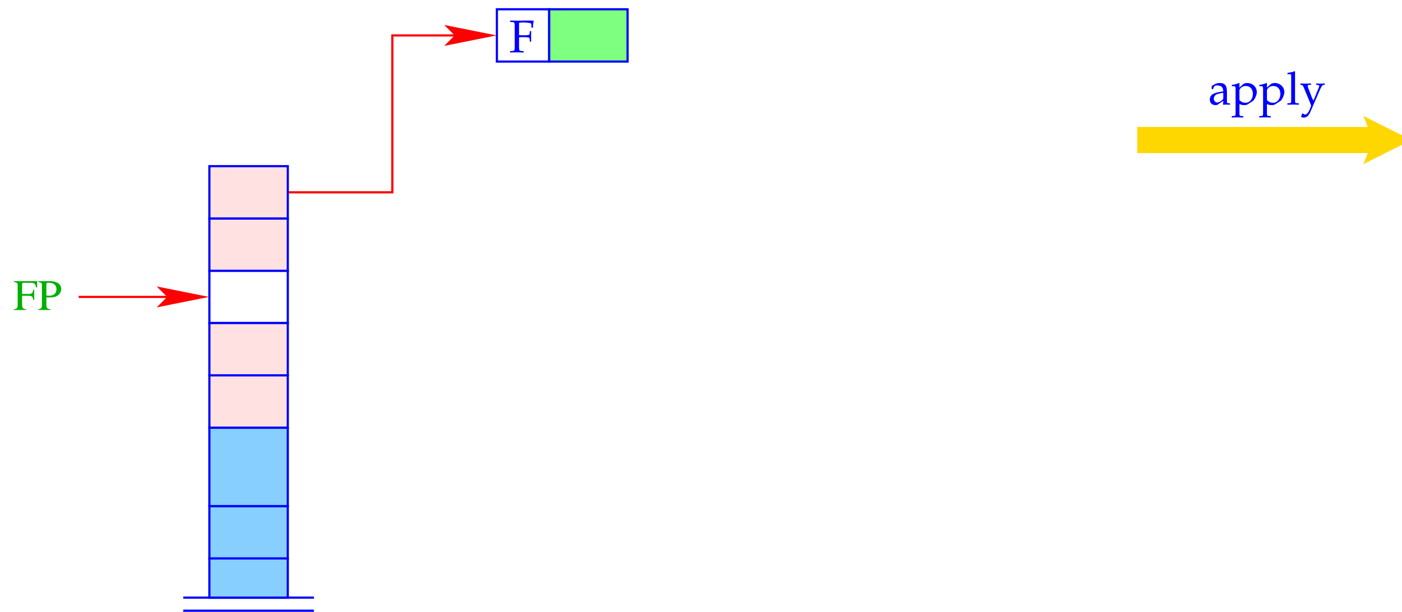
Damit erhalten wir etwa bei der Ausführung von return k:



popenv


GP
PC
17
FP
V


F
k
FP
slide k


F
FP
apply

19 letrec-Ausdrücke

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

- lokale Variablen $y_1, \ldots, y_n$ 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.

Achtung!

In einem **letrec**-Ausdruck können wir bei der Definition der Werte bereits Variablen verwenden, die erst später angelegt werden! $\Longrightarrow$ Vor der eigentlichen Definition werden Dummy-Werte auf den Stack gelegt.

Für CBN erhalten wir:

$$
\begin{array}{lcll}
\text{code}_V\ e\ \rho\ \text{kp} & = & \text{alloc n} & //\ \text{legt lok. Variablen an} \\
 & & \text{code}_C\ e_1\ \rho'\ (\text{kp}+n) & \\
 & & \text{rewrite n} & \\
 & & \ldots & \\
 & & \text{code}_C\ e_n\ \rho'\ (\text{kp}+n) & \\
 & & \text{rewrite 1} & \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{kp}+n) & \\
 & & \text{slide n} & //\ \text{gibt lok. Variablen auf}
\end{array}
$$

wobei $\qquad \rho' = \rho \oplus \{y_i \mapsto (L, \text{kp}+i) \mid i = 1, \ldots, n\}$.

Im Falle von CBV benutzen wir für die Ausdrücke $e_1, \ldots, e_n$ ebenfalls code_V.

Achtung:

Rekursive Definition von Basiswerten ist bei CBV undefiniert!!!

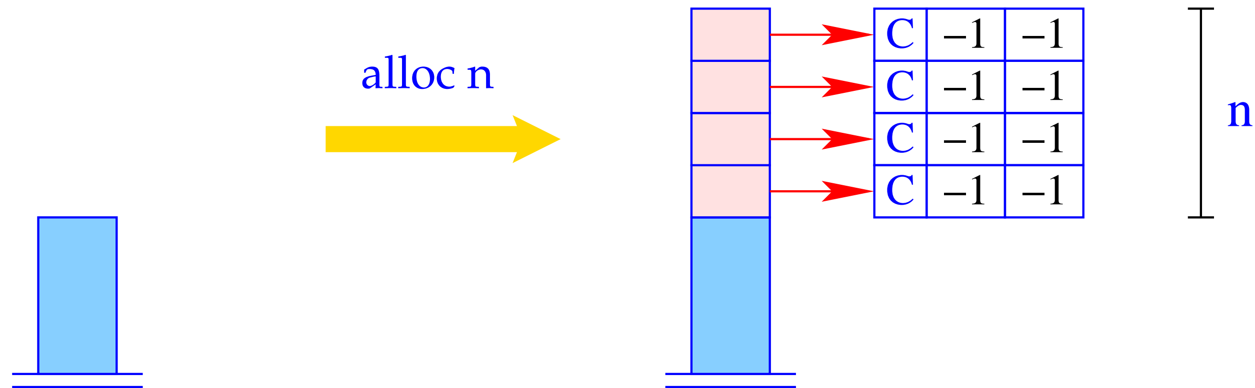
Beispiel:

Betrachte den Ausdruck

$$e \equiv \textbf{letrec } f = \textbf{fn } x, y \Rightarrow \textbf{if } y \leq 1 \textbf{ then } x \textbf{ else } f(x * y)(y - 1) \textbf{ in } f 1$$

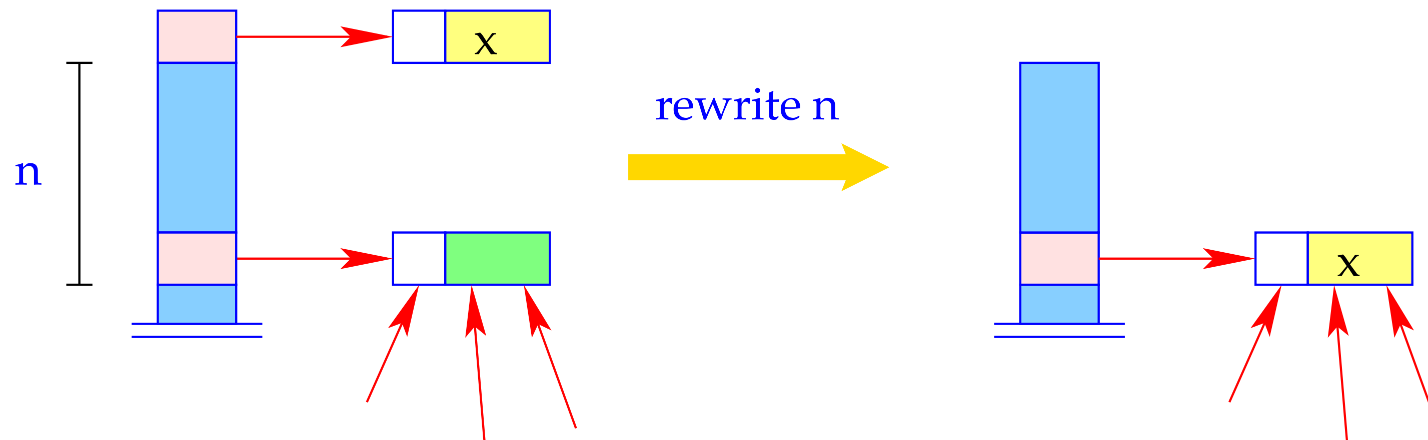
für $\rho = \emptyset$ und kp $= 0$. Dann ergibt sich (für CBV):

0	alloc 1	0	A:	targ 2	4		loadc 1
1	pushloc 0	0		...	5		mkbasic
2	mkvec 1	1		return 2	5		pushloc 4
2	mkfunval A	2	B:	rewrite 1	6		apply
2	jump B	1		mark C	2	C:	slide 1




```
for (i=1; i≤n; i++)
     S[SP+i] = new (C,-1,-1);
SP = SP + n;
```

Die Instruktion alloc n reserviert n Zellen auf dem Keller und initialisiert diese mit n Dummy-Knoten.




```
H[S[SP-n]] = H[S[SP]];
SP = SP - 1;
```

- Die Referenz S[SP – n] bleibt erhalten!
- Was überschrieben wird, ist nur ihr Inhalt!

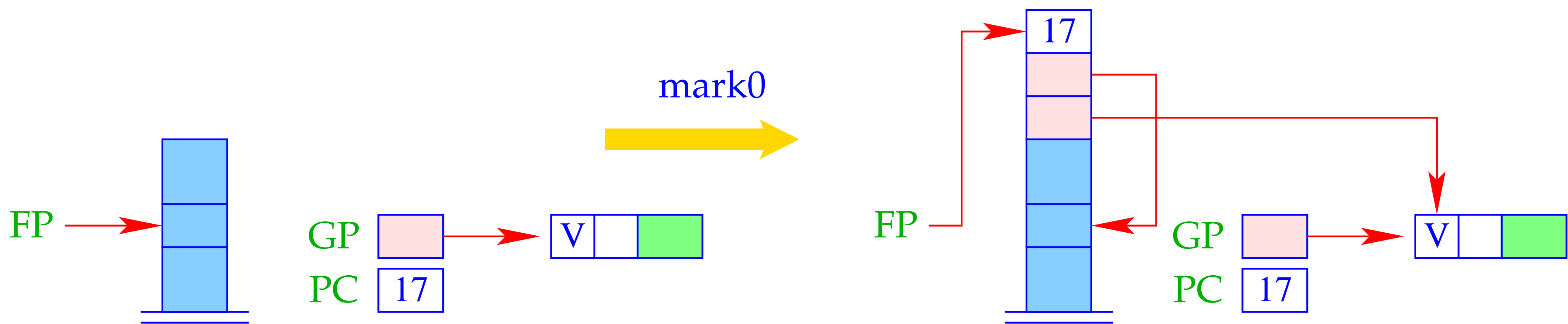
20 Abschlüsse und ihre Auswertung

- 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.

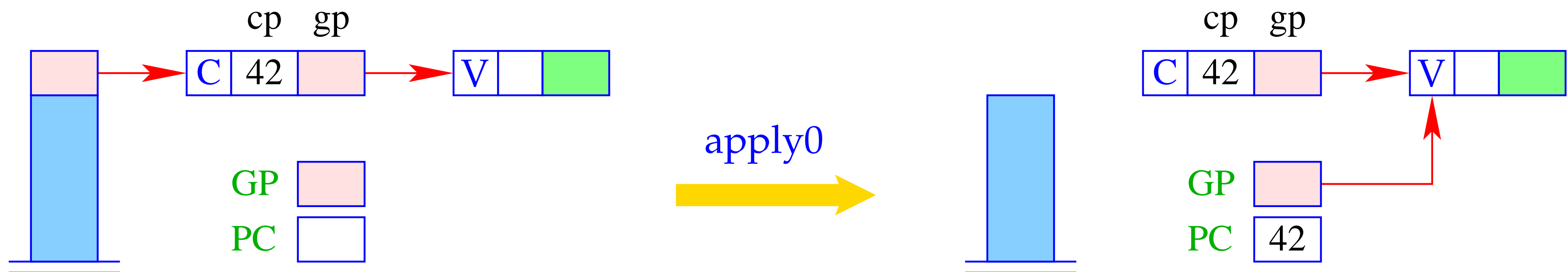
eval lässt sich wieder in übersichtlichere Bestandteile verlegen:

```
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.

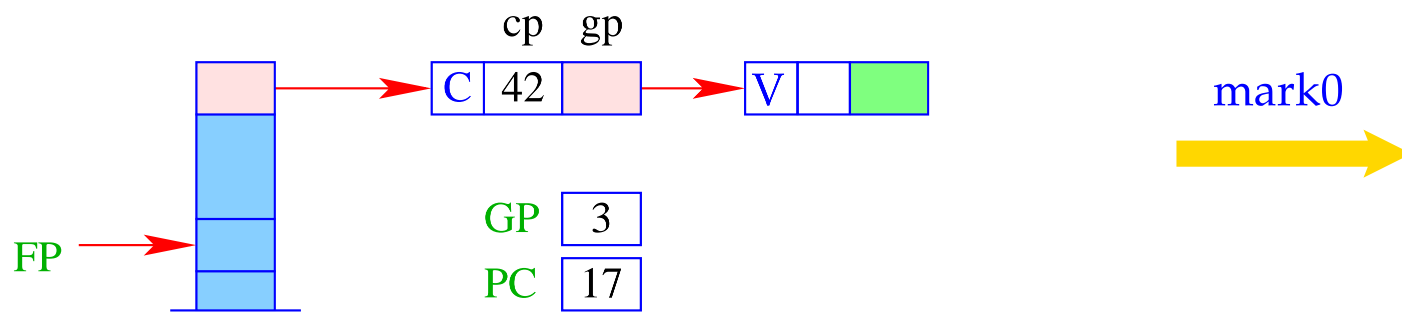


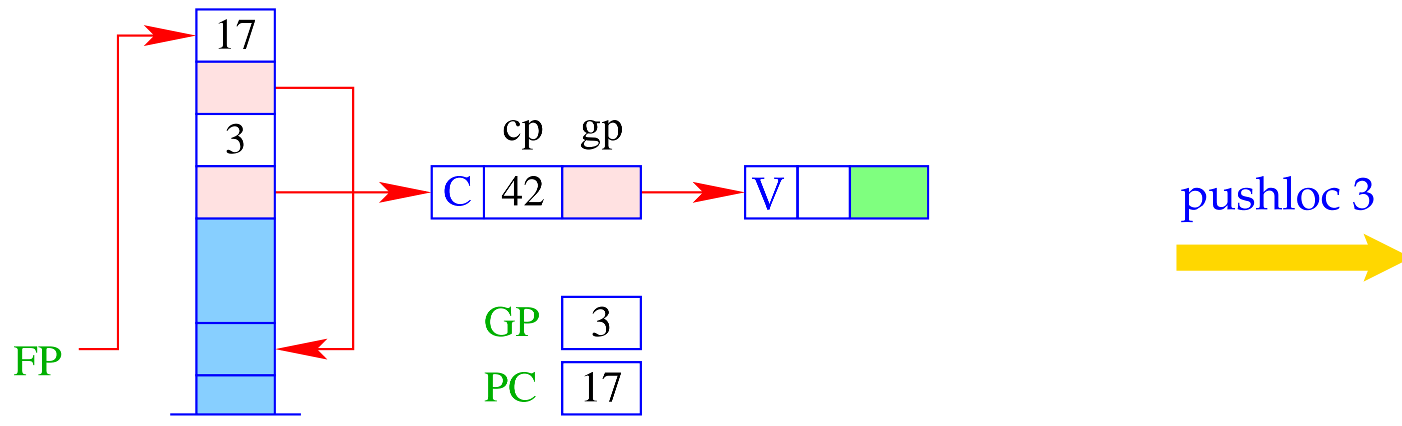
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

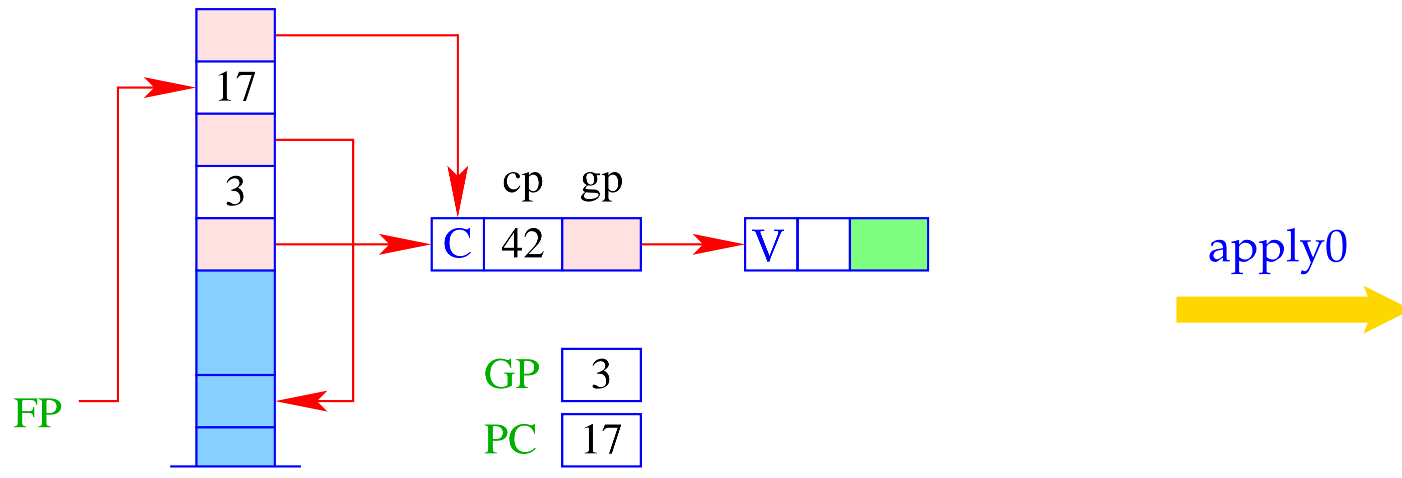


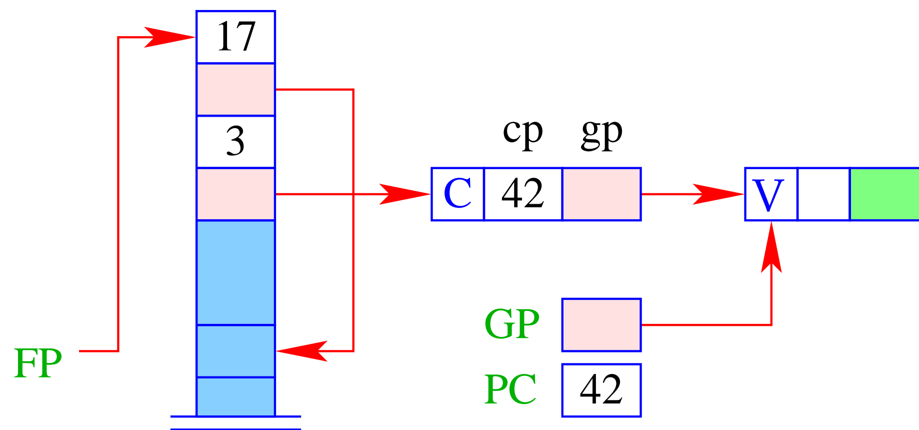
h = S[SP]; SP--;
GP = h→gp; PC = h→cp;

Damit erhalten wir für die Instruktion eval:


cp
gp
C
42
V
mark0
GP
3
PC
17
FP


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

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

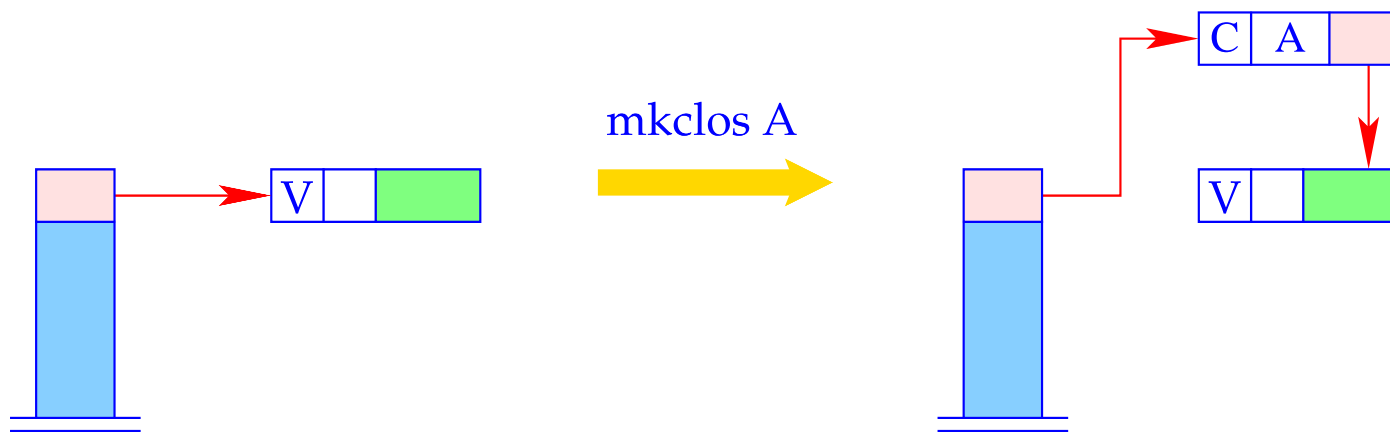
wobei $\{z_0, \dots, z_{g-1}\} = \mathit{free}(e)$ und $\rho' = \{z_i \mapsto (G, i) \mid i = 0, \dots, g-1\}$.

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	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.



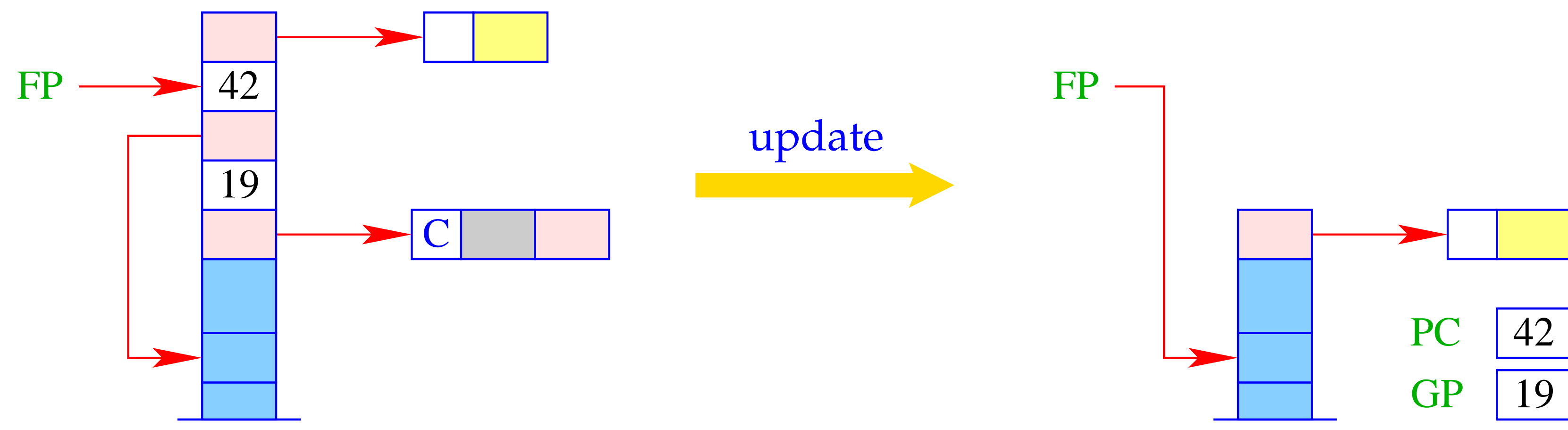
S[SP] = new (C, A, S[SP]);

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

popenv

rewrite 1

Sie überschreibt den Abschluss mit dem berechneten Wert.



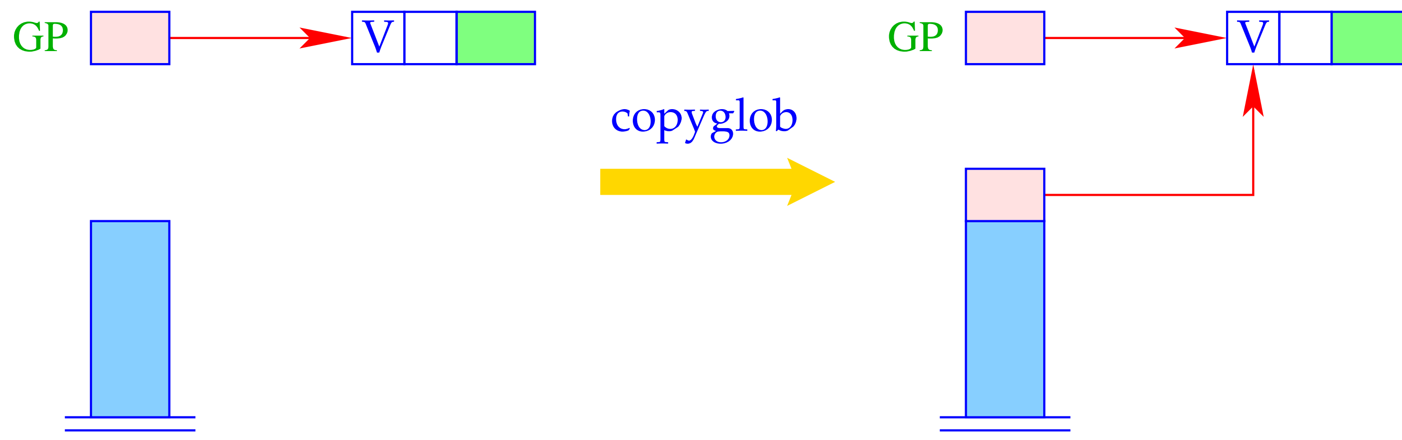
21 Optimierungen I: globale Variablen

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**/**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 $\Longrightarrow$ 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:

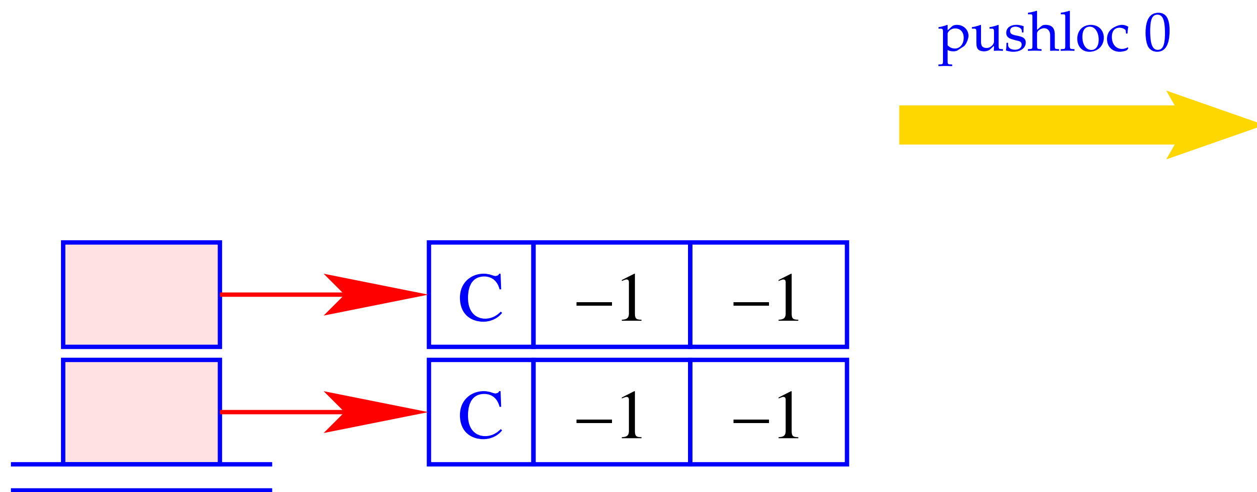
getvar x ρ kp	mkclos A	A:	pushglob 0		update
mkvec 1	jump B		eval	B:	...

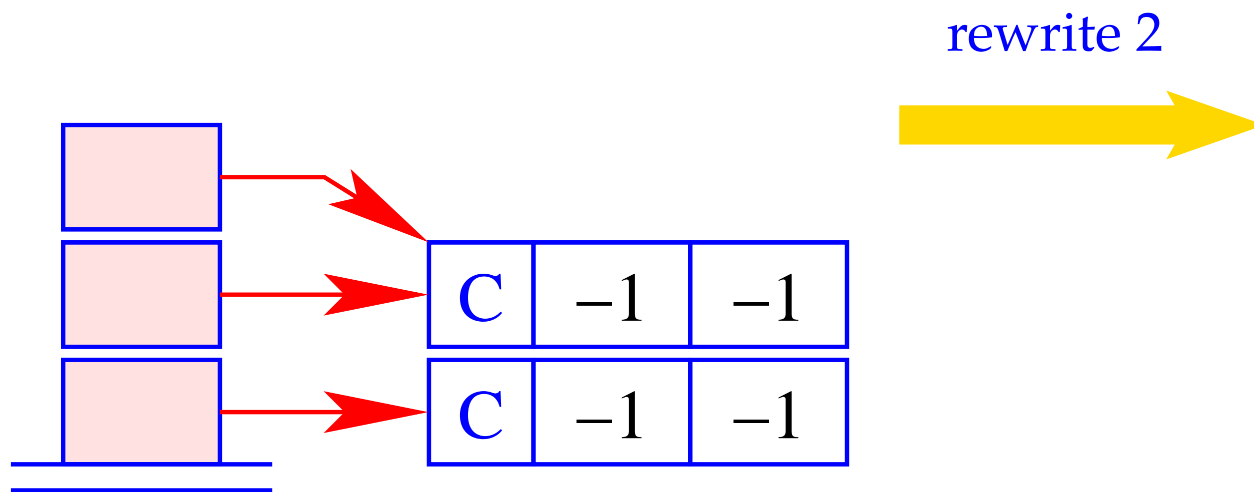
Beispiel: $e \equiv \textbf{letrec}\ a = b; b = 7\ \textbf{in}\ a.$ Dann liefert $\text{code}_V\ e\ \emptyset\ 0$:

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

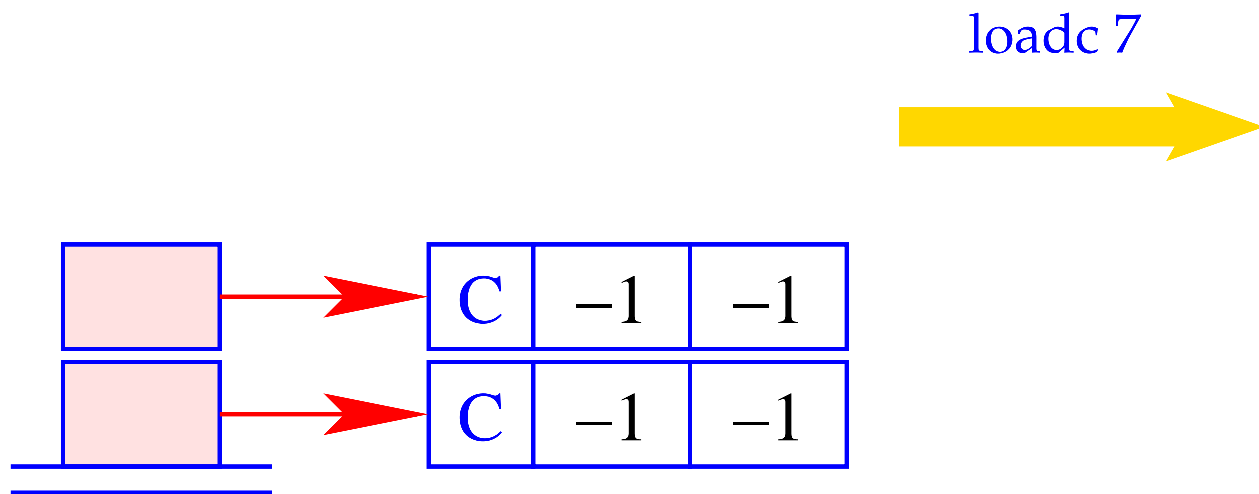
Ausführung dieser Folge sollte den Basiswert 7 liefern ...

alloc 2

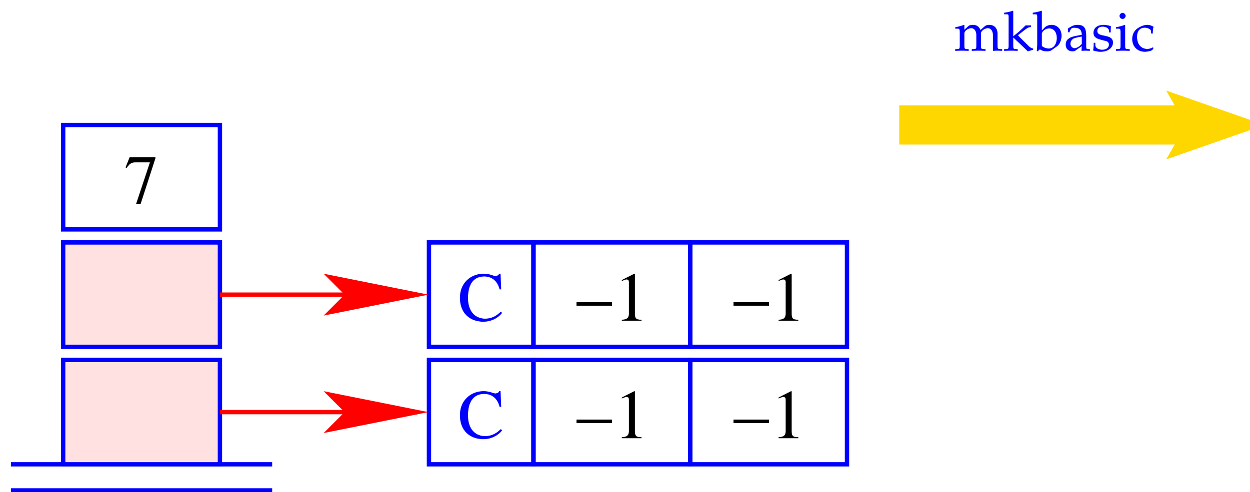

pushloc 0
C
−1
−1
C
−1
−1

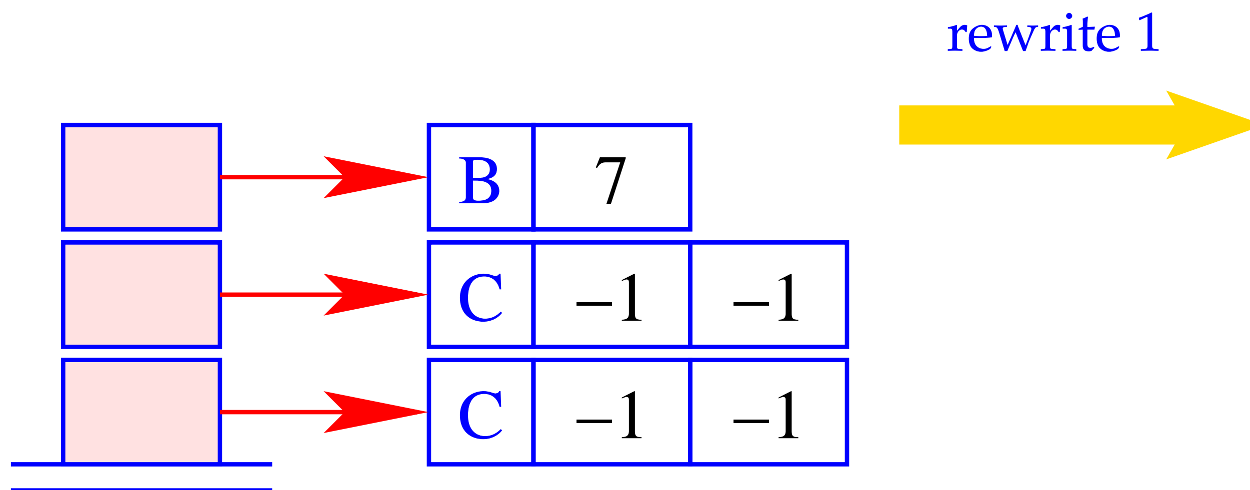


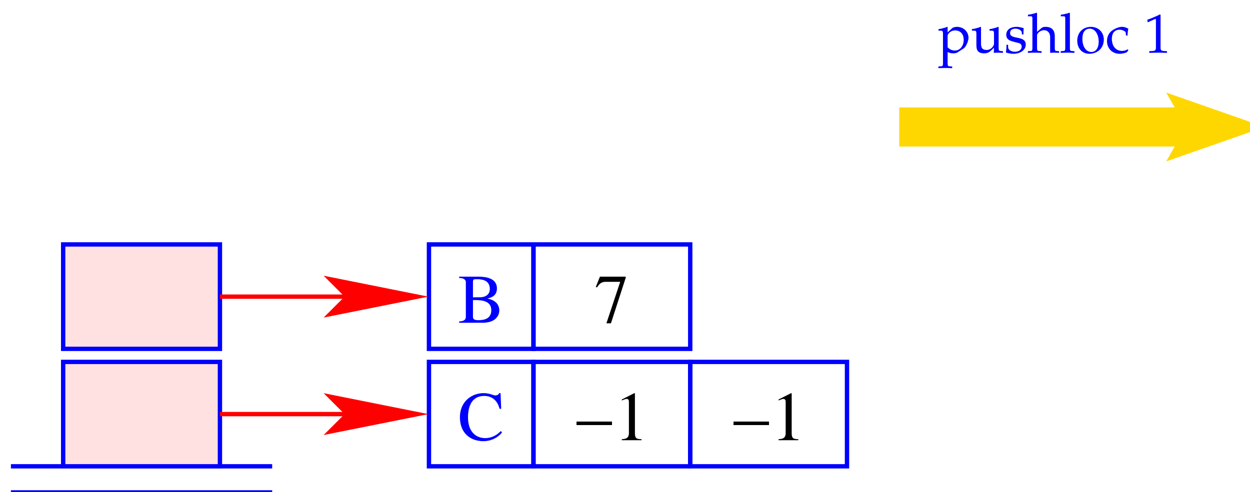
rewrite 2
C
–1
–1
C
–1
–1

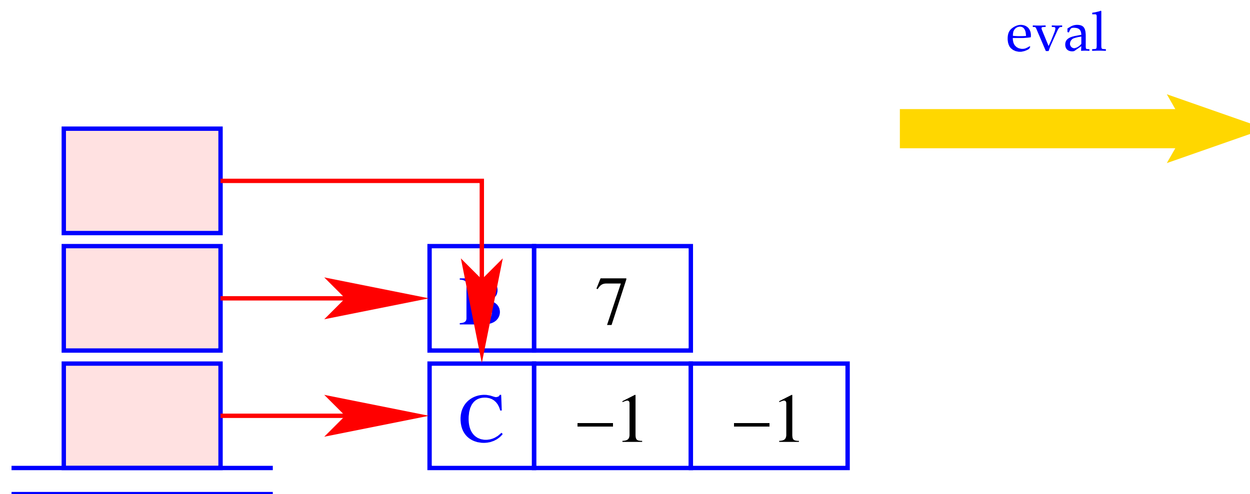


loadc 7
C
−1
−1
C
−1
−1


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!!

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:

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

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

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

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.

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

Beispiel: $\textbf{let}\ a = 17;\ f = \textbf{fn}\ b \Rightarrow a + b\ \textbf{in}\ f\ 42$

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

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

Entsprechend erweitern wir die Syntax unserer Programm-Ausdrücke durch:

$$
\begin{array}{rcl}
e & ::= & \ldots \mid (e_0, \ldots, e_{k-1}) \mid \#j\ e \\
 & & \textbf{let}\ (x_0, \ldots, x_{k-1}) = e_1\ \textbf{in}\ e_0
\end{array}
$$

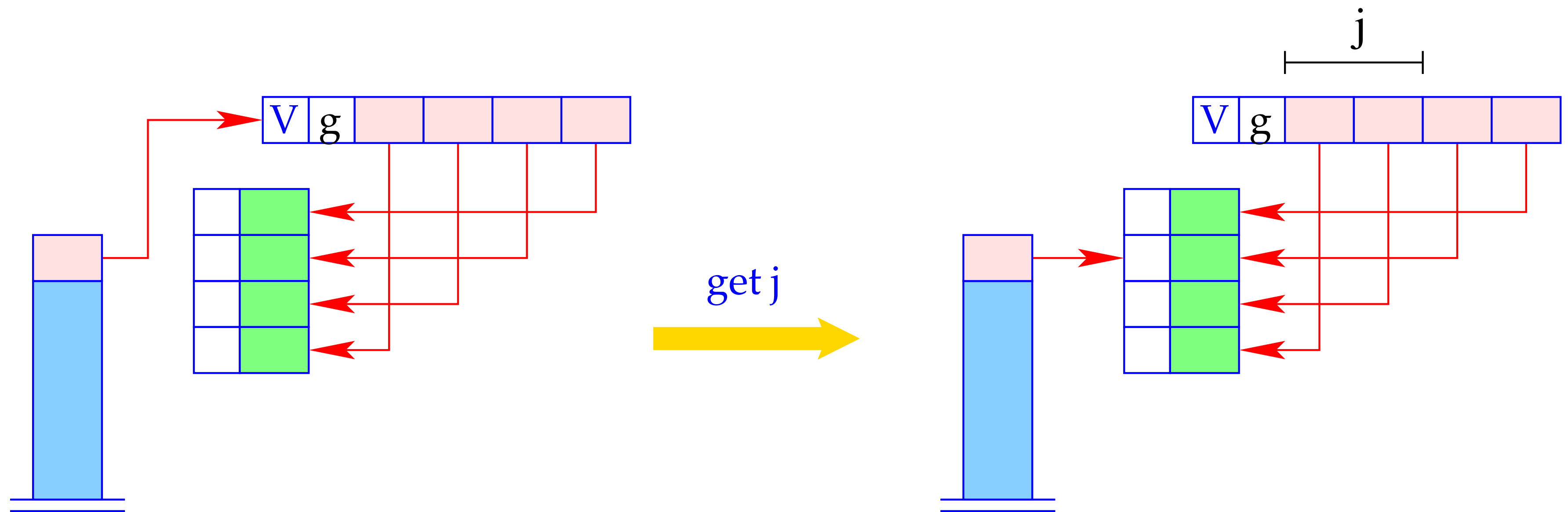
- Tupel werden angelegt, indem erst die Folge der Referenzen auf ihre Komponenten auf dem Keller gesammelt und dann mithilfe der Operation mkvec in den Heap gelegt werden.
- Auf Komponenten greifen wir zu, indem wir innerhalb des Tupels einen indizierten Zugriff vornehmen.

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

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

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!";

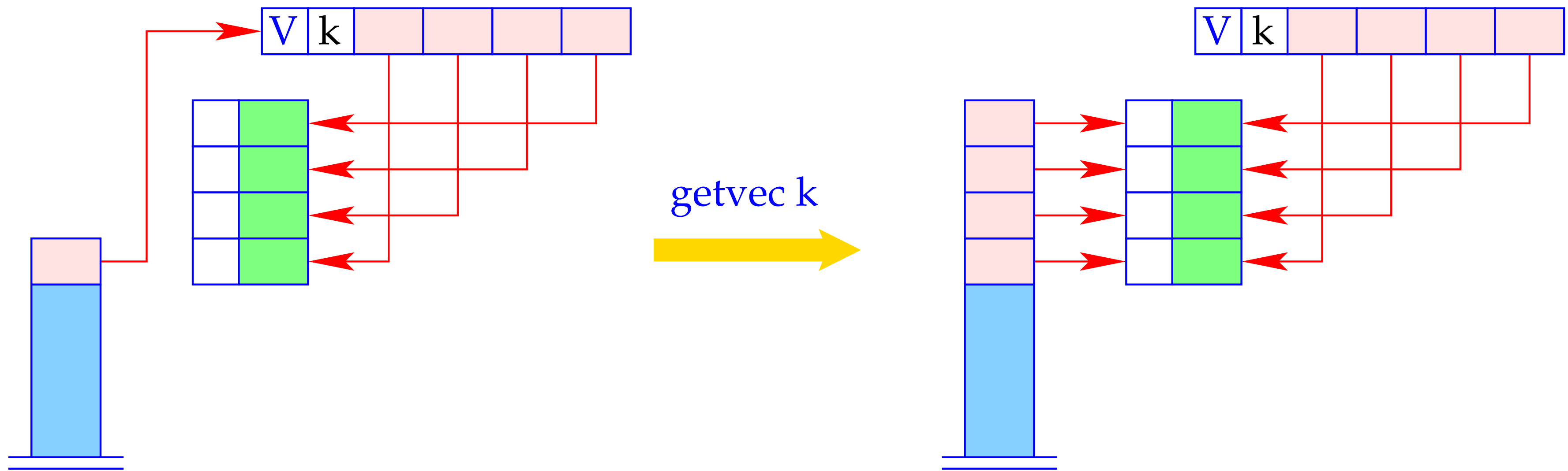
Möchte man nicht auf einzelne, sondern gegebenenfalls alle Komponenten eines Tupels zugreifen, kann man dies mithilfe des Ausdrucks
$e \equiv \textbf{let}\ (y_0, \ldots, y_{k-1}) = e_1\ \textbf{in}\ e_0$ tun.

Diesen übersetzen wir wie folgt:

$$
\begin{array}{lll}
\text{code}_V\ e\ \rho\ \text{kp} & = & \text{code}_V\ e_1\ \rho\ \text{kp} \\
 & & \text{getvec k} \\
 & & \text{code}_V\ e_0\ \rho'\ (\text{kp} + k) \\
 & & \text{slide k}
\end{array}
$$

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

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:

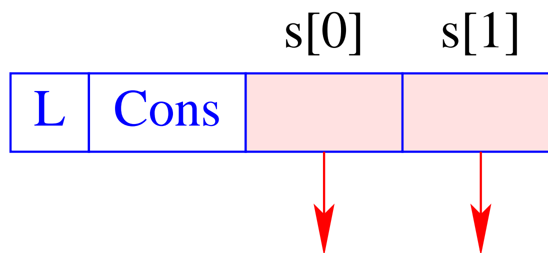
$$
\begin{array}{lllll}
\text{app} & = & \textbf{fn}\ l, y \Rightarrow \textbf{case}\ l\ \textbf{of} & & \\
 & & [] & \rightarrow & y \\
 & & h : t & \rightarrow & h : (\text{app}\ t\ y)
\end{array}
$$

Folglich erweitern wir die Syntax von Ausdrücken e um:

$$\begin{array}{lll} e & ::= & \ldots \mid [] \mid (e_1 : e_2) \\ & & \mid (\textbf{case}\ e_0\ \textbf{of}\ [] \to e_1;\ \ h : t \to e_2) \end{array}$$

Neue Halden-Objekte:

leere Liste



nicht–leere Liste

24.3 Der Aufbau von Listen

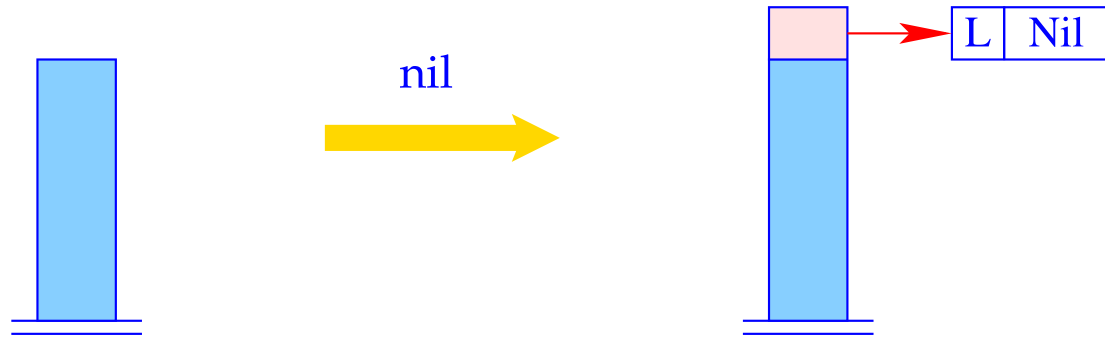
Für das Anlegen von Listen-Knoten führen wir die Befehle nil und cons ein.

Damit erhalten wir für CBN:

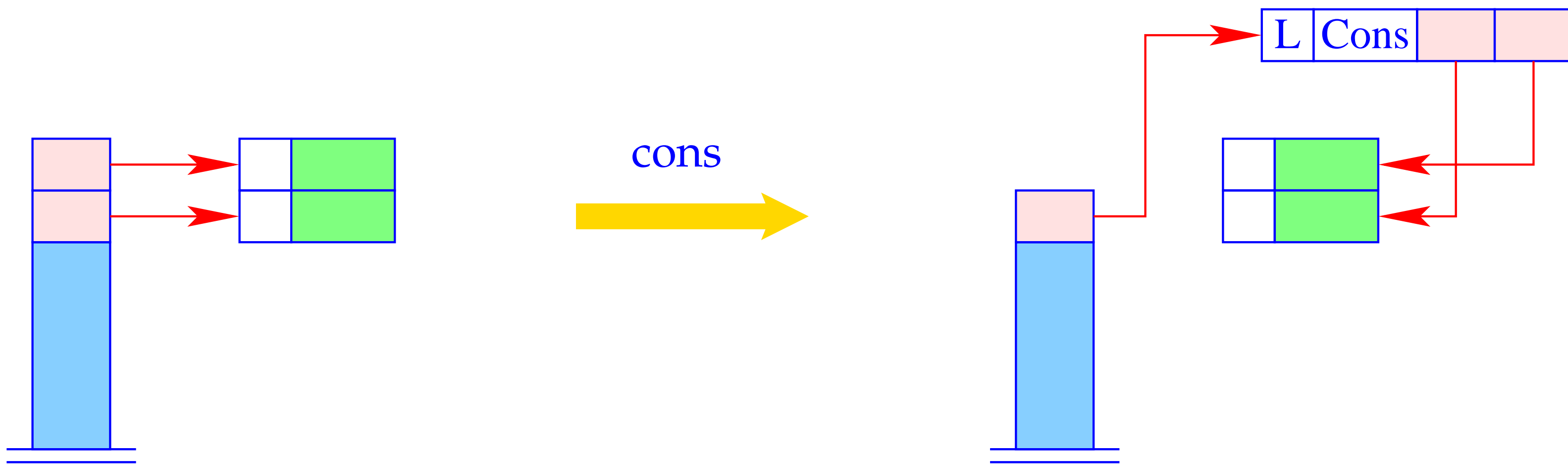
$$
\begin{array}{lcl}
\text{code}_V\ []\ \rho\ \text{kp} & = & \text{nil} \\
\text{code}_V\ (e_1 : e_2)\ \rho\ \text{kp} & = & \text{code}_C\ e_1\ \rho\ \text{kp} \\
& & \text{code}_C\ e_2\ \rho\ (\text{kp}+1) \\
& & \text{cons}
\end{array}
$$

Beachte:

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



S[SP] = SP++; S[SP] = new (L,Nil);



S[SP-1] = new (L,Cons, S[SP-1], S[SP]);
SP- -;

24.4 Pattern-Matching

Betrachte den Ausdruck $e \equiv \textbf{case}\ e_0\ \textbf{of}\ [\,] \rightarrow e_1;\ \ h : t \rightarrow e_2$.

Auswertung von e erfordert:

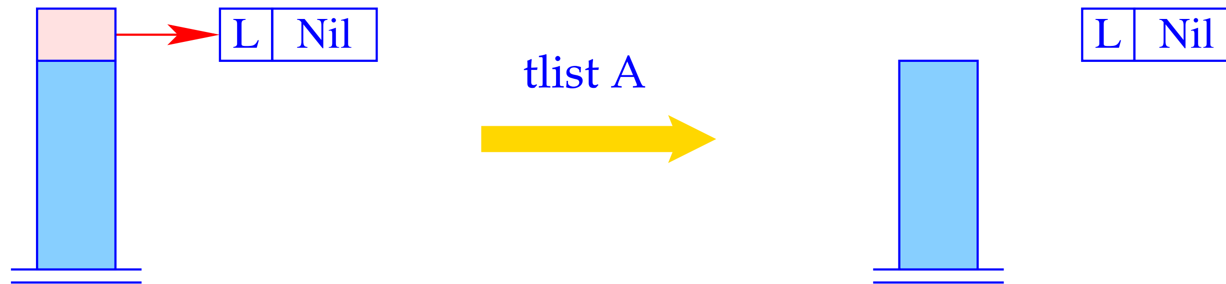
- Auswertung von e_0;
- Überprüfung, ob e_0 ein L-Objekt ist;
- Falls e_0 gleich der leeren Liste ist, Auswertung von e_1 ...
- ... andernfalls Kellern der Verweise des L-Objekts und Auswertung von e_2.

Folglich erhalten wir (für CBN wie CBV):

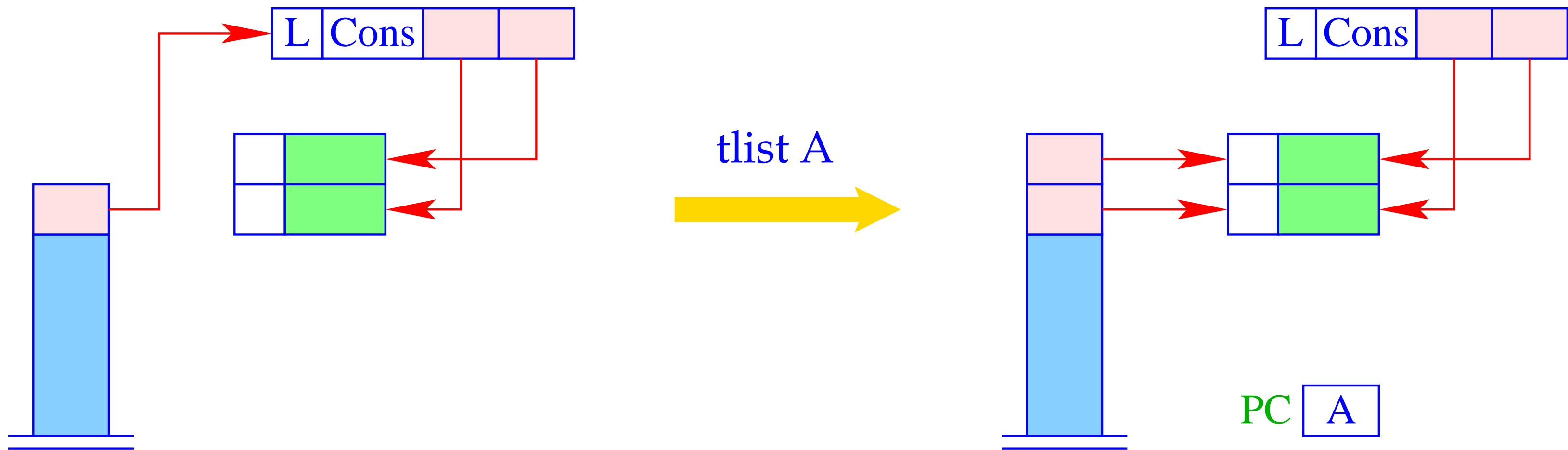
$$
\begin{array}{rrl}
\text{code}_V\; e\; \rho\; \text{kp} & = & \text{code}_V\; e_0\; \rho\; \text{kp} \\
& & \text{tlist A} \\
& & \text{code}_V\; e_1\; \rho\; \text{kp} \\
& & \text{jump B} \\
& \text{A}: & \text{code}_V\; e_2\; \rho'\; (\text{kp}+2) \\
& & \text{slide 2} \\
& \text{B}: & \ldots
\end{array}
$$

wobei $\rho' = \rho \oplus \{h \mapsto (L, kp+1), t \mapsto (L, kp+2)\}$.

Der neue Befehl tlist A führt die notwendigen Überprüfungen durch und legt (im Cons-Fall) zwei neue lokale Variablen an:




```
h = S[SP];
if (H[h] != (L,...)
    Error “no list!”;
if (H[h] == (_,Nil)) SP- -;
      ...
```




```
... else {
   S[SP+1] = S[SP]→s[1];
   S[SP] = S[SP]→s[0];
   SP++; PC = A;
}
```

Beispiel: Der (entwirrte) Rumpf der Funktion app mit $\text{app} \mapsto (G, 0)$:

0		targ 2	3		pushglob 0	0	C:	mark D
0		pushloc 0	4		pushloc 2	3		pushglob 2
1		eval	5		pushloc 6	4		pushglob 1
1		tlist A	6		mkvec 3	5		pushglob 0
0		pushloc 1	4		mkclos C	6		eval
1		eval	4		cons	6		apply
1		jump B	3		slide 2	1	D:	update
2	A:	pushloc 1	1	B:	return 2			

Beachte:

Hat man Datentypen mit mehr als zwei Konstruktoren, benötigt man eine Verallgemeinerung des tlist-Befehls, der einer `switch`-Anweisung entspricht :-)

24.5 Abschlüsse von Tupeln und Listen

Das generelle Schema für code_C lässt sich auch bei Tupeln und Listen optimieren:

$$
\begin{array}{lllll}
\text{code}_C\ (e_0, \ldots, e_{k-1})\ \rho\ \text{kp} & = & \text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{kp} & = & \text{code}_C\ e_0\ \rho\ \text{kp} \\
 & & & & \text{code}_C\ e_1\ \rho\ (\text{kp} + 1) \\
 & & & & \ldots \\
 & & & & \text{code}_C\ e_{k-1}\ \rho\ (\text{kp} + k - 1) \\
 & & & & \text{mkvec k} \\
\text{code}_C\ []\ \rho\ \text{kp} & = & \text{code}_V\ []\ \rho\ \text{kp} & = & \text{nil} \\
\text{code}_C\ (e_1 : e_2)\ \rho\ \text{kp} & = & \text{code}_V\ (e_1 : e_2)\ \rho\ \text{kp} & = & \text{code}_C\ e_1\ \rho\ \text{kp} \\
 & & & & \text{code}_C\ e_2\ \rho\ (\text{kp} + 1) \\
 & & & & \text{cons}
\end{array}
$$

25 Letzte Aufrufe

Das Aufruf-Vorkommen $l \equiv e'\ e_0 \ldots e_{m-1}$ heißt letzt in einem Ausdruck e, falls die Auswertung von l den Wert für e liefern kann.

Beispiele:

$r\ t\ (h : y)$ ist letzt in	$\textbf{case}\ x\ \textbf{of}\ []\ \rightarrow y;\ \ h : t\ \rightarrow r\ t\ (h : y)$
$f\ (x - 1)$ ist nicht letzt in	$\textbf{if}\ x \leq 1\ \textbf{then}\ 1\ \textbf{else}\ x * f\ (x - 1)$

Beobachtung:

Letzte Aufrufe eines Funktions-Rumpfs benötigen keinen neuen Kellerrahmen!

Automatische Transformation von Tail Recursion in Schleifen !!!

Der Code für einen letzten Aufruf $l \equiv (e'\ e_0 \ldots e_{m-1})$ in einer Funktion f mit k Argumenten muss:

- die aktuellen Parameter e_i anlegen und die Funktion e' bestimmen;
- die lokalen Variablen sowie die k verbrauchten Argumente von f frei geben;
- apply ausführen.

$$
\begin{array}{lll}
\mathrm{code}_V\ l\ \rho\ \mathrm{kp} & = & \mathrm{code}_C\ e_{m-1}\ \rho\ \mathrm{kp} \\
& & \mathrm{code}_C\ e_{m-2}\ \rho\ (\mathrm{kp}+1) \\
& & \ldots \\
& & \mathrm{code}_C\ e_0\ \rho\ (\mathrm{kp}+m-1) \\
& & \mathrm{code}_V\ e'\ \rho\ (\mathrm{kp}+m) \qquad // \text{ Auswerten der Funktion} \\
& & \mathrm{move}\ r\ (m+1) \qquad // \text{ Freigabe} \\
& & \mathrm{apply}
\end{array}
$$

wobei $r = kp + k$ die Anzahl der freizugebenden stack-Zellen ist.

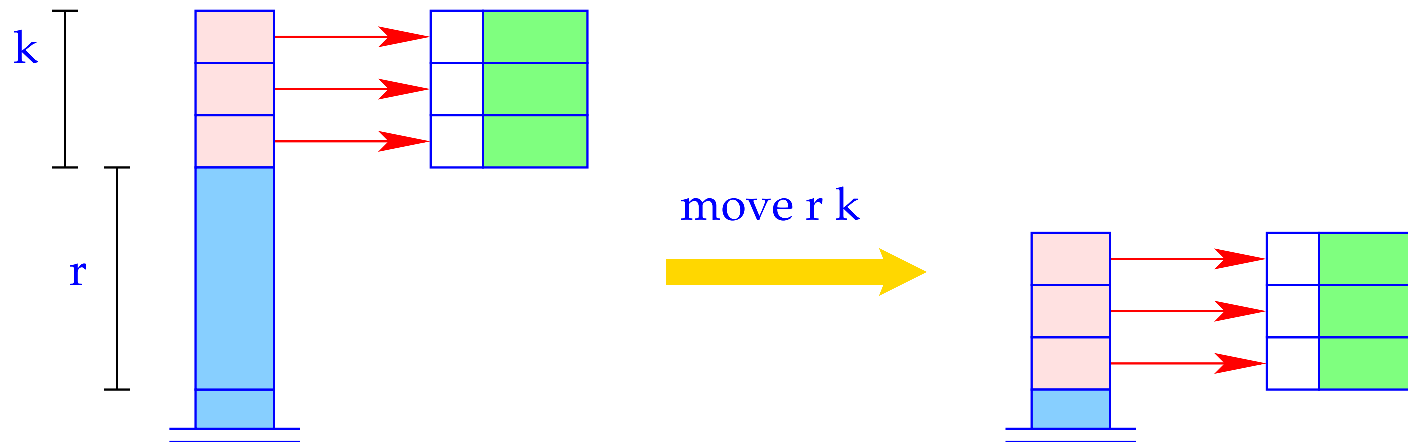
Beispiel:

Der Rumpf der Funktion

$$r = \textbf{fn}\, x, y \;\Rightarrow\; \textbf{case}\; x\; \textbf{of}\; [\,] \;\rightarrow y;\; h : t \;\rightarrow r\; t\; (h : y)$$

0		targ 2	1		jump B	4		pushglob 0
0		pushloc 0				5		eval
1		eval	2	A:	pushloc 1	5		move 4 3
1		tlist A	3		pushloc 4			apply
0		pushloc 1	4		cons			slide 2
1		eval	3		pushloc 1	1	B:	return 2

Da der alte Kellerrahmen beibehalten wird, wird return 2 nur über den direkten Sprung am Ende der []-Alternative erreicht.




```
SP = SP – k – r;
for (i=1; i≤k; i++)
    S[SP+i] = S[SP+i+r];
SP = SP + k;
```

Die Übersetzung logischer Programmiersprachen

26 Die Sprache PuP

Wir betrachten hier nur die Mini-Sprache PuP (“Pure Prolog”). Insbesondere verzichten wir auf:

- Arithmetik;
- den Cut-Operator (vorerst :-)
- Selbst-Modifikation von Programmen mittels assert und retract.

Beispiel:

$$
\begin{array}{lll}
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Y) \\
\mathsf{is_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Z), \mathsf{is_bigger}(Z, Y) \\
? \quad \mathsf{is_bigger}(\mathit{elephant}, \mathit{dog}) & &
\end{array}
$$

Ein realistischeres Beispiel:

$$\begin{array}{lll} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,], \; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'], \; Z = [H|Z'], \; \mathsf{app}(X', Y, Z') \end{array}$$

$$? \quad \mathsf{app}(X, [Y, c], [a, b, Z])$$

Ein realistischeres Beispiel:

$$\begin{array}{lll} \mathsf{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathsf{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X',Y,Z') \\ ? \quad \mathsf{app}(X,[Y,c],[a,b,Z]) & & \end{array}$$

Bemerkung:

$[\,]$	$==$	das Atom leere Liste
$[H|Z]$	$==$	binäre Konstructor-Anwendung
$[a,b,Z]$	$==$	Abkürzung für: $[a|[b|[Z|[\,]]]]$

Ein Programm p ist darum wie folgt aufgebaut:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid _ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 . \ldots . c_m ? g
\end{array}
$$

- Ein Term t ist entweder ein Atom, eine (evt. anonyme) Variable oder eine Konstruktor-Anwendung.
- Ein Ziel g ist entweder ein Literal, d.h. ein Prädikats-Aufruf, oder eine Unifikation.
- Eine Klausel c besteht aus einem Kopf $p(X_1, \ldots, X_k)$ mit Prädikats-Namen und Liste der formalen Parameter sowie einer Folge von Zielen als Rumpf.
- Ein Programm besteht aus einer Folge von Klauseln sowie einem Ziel als Anfrage.

Prozedurale Sicht auf PuP-Programme:

Ziel	$==$	Prozedur-Aufruf
Prädikat	$==$	Prozedur
Definition	$==$	Rumpf
Term	$==$	Wert
Unifikation	$==$	elementarer Berechnungschritt
Bindung von Variablen	$==$	Seiteneffekt

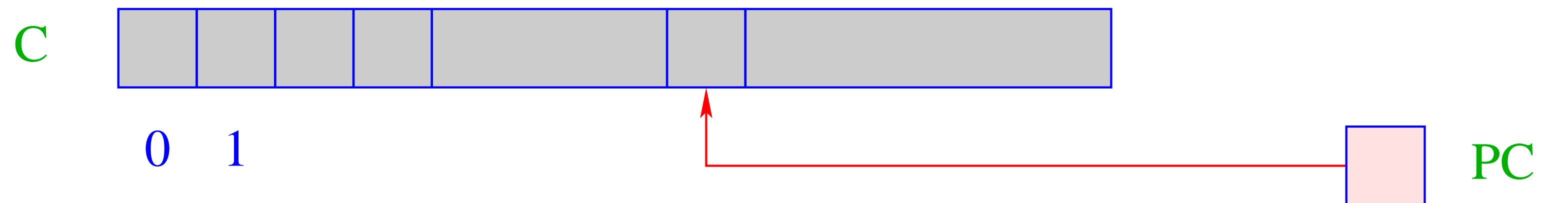
Achtung: Prädikat-Aufrufe ...

- ... liefern keinen Rückgabewert!
- ... beeinflussen den Aufrufer einzig durch Seiteneffekte :-)
- ... können fehlschlagen. Dann wird die nächste Definition probiert :-))

$\Longrightarrow$ backtracking

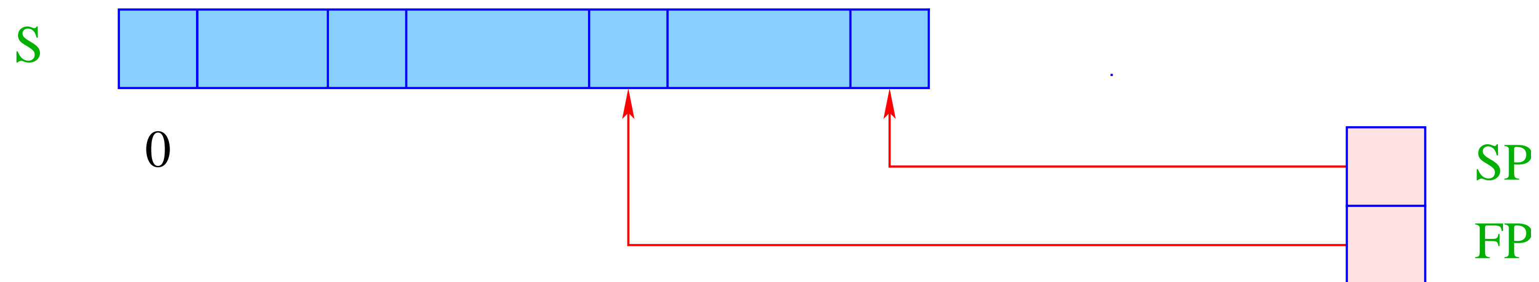
27 Architektur der WiM:

Der Code-Speicher:



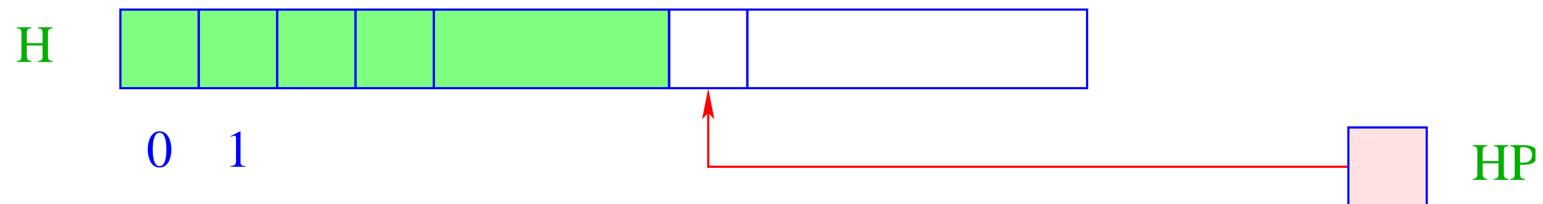
C	=	Code Speicher – enthält WiM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter – zeigt auf nächsten auszuführenden Befehl.

Der Laufzeit-Keller:



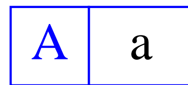
S	=	Laufzeit-Stack – jede Zelle enthält einen Wert oder eine Adresse;
SP	=	Stack Pointer – zeigt auf die oberste belegte Zelle;
FP	=	Frame Pointer – zeigt auf den aktuellen Kellerrahmen.
		Rahmen werden für jeden Prädikat-Aufruf erzeugt,
		enthalten Platz für die Variablen der aktuellen Klausel.

Der Heap:

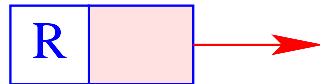


H	=	Heap für dynamisch erzeugte Terme;
HP	=	Heap-Pointer – zeigt auf die erste freie Zelle.

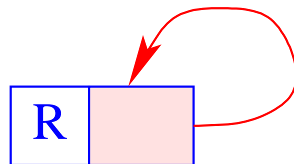
- Der Heap wird ebenfalls wie ein Keller verwaltet :-)
- Ein new-Befehl allokiert ein Objekt in H.
- Objekte sind mit ihrem Typ markiert (wie bei der MaMa) ...



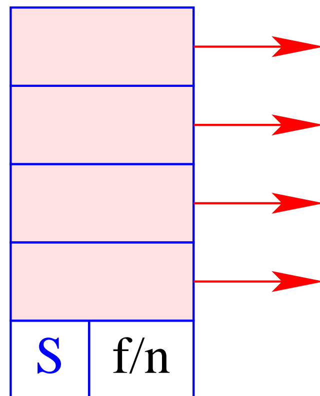
Atom — 1 Zelle



Variable — 1 Zelle



ungebundene Variable — 1 Zelle



Struktur — (n+1) Zellen

28 Anlegen von Termen in der Halde

Argumente von Zielen (Aufrufen) werden vor Übergabe im Heap aufgebaut.

Nehmen wir an, wir hätten eine Adress-Umgebung ρ, die für jede Variable X die Adresse (relativ zu FP) auf dem Keller liefert.

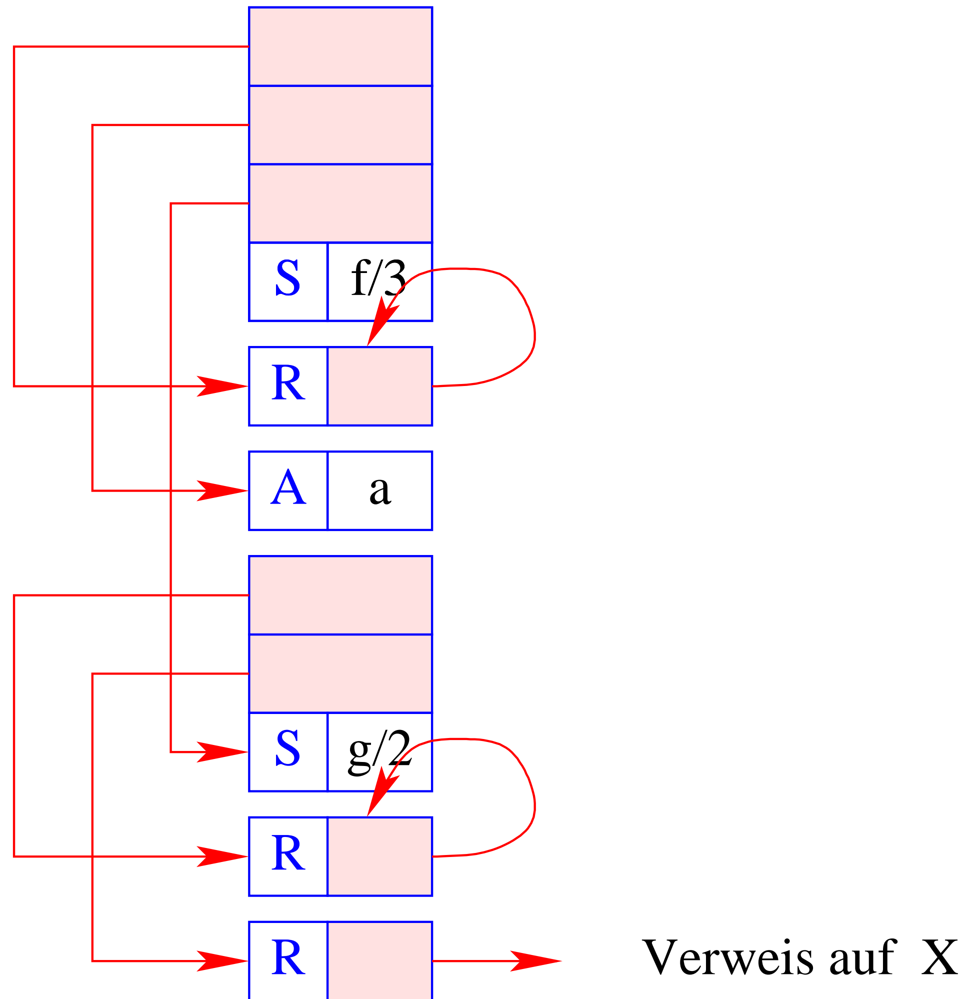
Dann wollen wir für einen Term t eine Folge $\text{code}_A\ t\ \rho$ von Instruktionen erzeugen, die (eine Repräsentation von) t im Heap aufbaut.

Idee:

- Baue den Baum in einer post-order Traversierung von t auf;
- erfinde dafür Befehle, die entsprechend einzelne neue Knoten anlegen!

Beispiel: $t \equiv f(g(X, Y), a, Z)$.

Dabei soll X bereits initialisiert sein, d.h. $S[FP + \rho\, X]$ eine Referenz enthalten, Y und Z aber noch nicht. Dann soll der Heap wie folgt erweitert werden:


S
f/3
R
A
a
S
g/2
R
R
Verweis auf X

Zur Unterscheidung kennzeichnen wir Vorkommen bereits initialisierter Variablen mit einem Oberstrich (z.B. $\bar{X}$).

Beachte: Argumente sind immer initialisiert, anonyme Variablen nie!

Dann definieren wir:

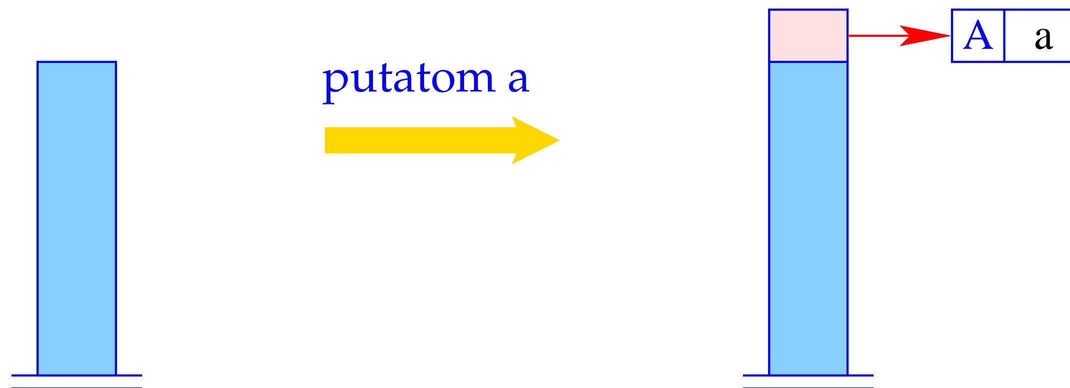
$$
\begin{array}{lcl}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\, X) \\
\text{code}_A\ \bar{X}\ \rho & = & \text{putref}\ (\rho\, X) \\
\text{code}_A\ _\ \rho & = & \text{putanon}
\end{array}
$$

Im Beispiel liefert das für $f(g(\bar{X}, Y), a, Z)$ mit $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ die Folge:

putref 1	putatom a
putvar 2	putvar 3
putstruct g/2	putstruct f/3

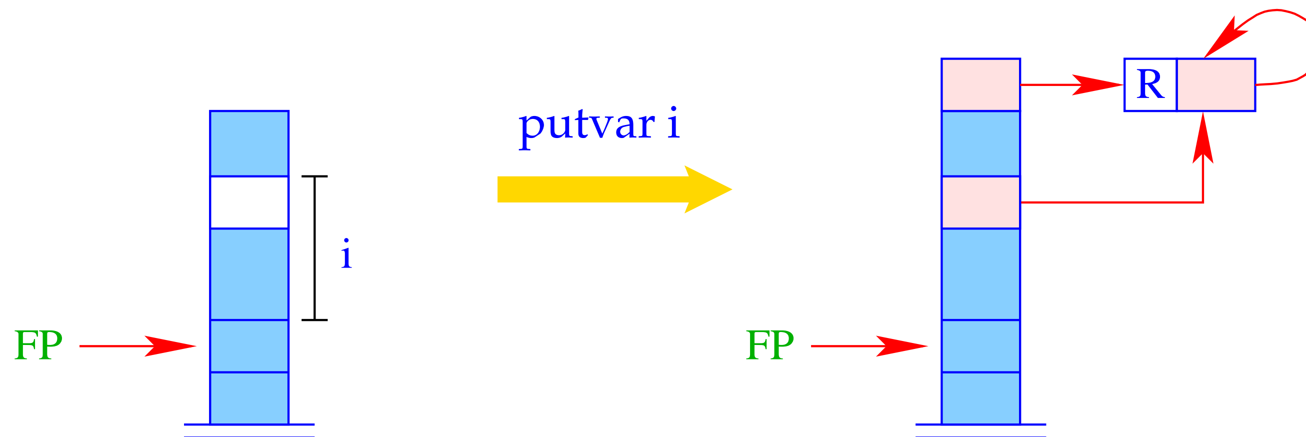
Dabei bedeuten die Befehle:

Die Instruktion putatom a legt ein Atom auf der Halde an:



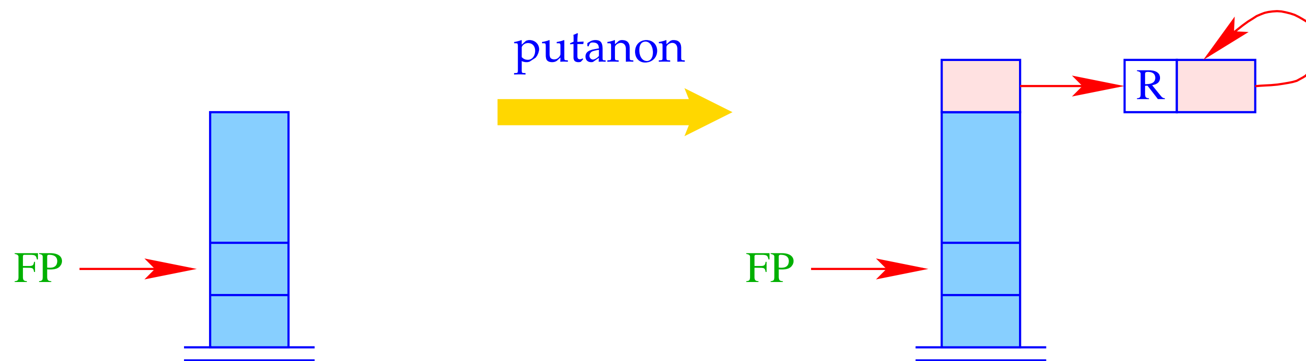
SP++; S[SP] = new (A,a);

Die Instruktion putvar i erzeugt eine neue uninitialisierte Variable und initialisiert zusätzlich die entsprechende Zelle im Kellerrahmen:



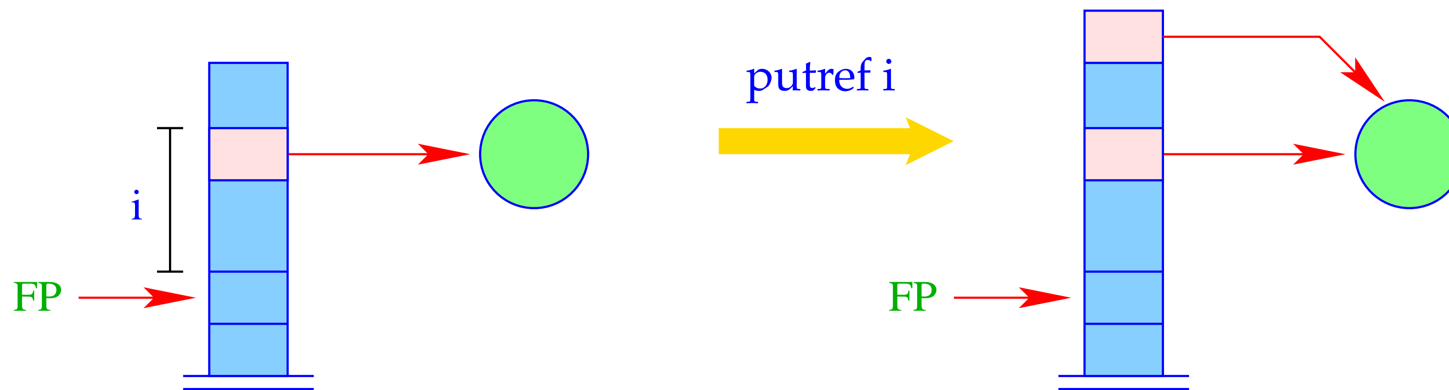
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];

Die Instruktion putanon führt eine neue ungebundene Variable ein, aber speichert keine Referenz darauf im Kellerrahmen:



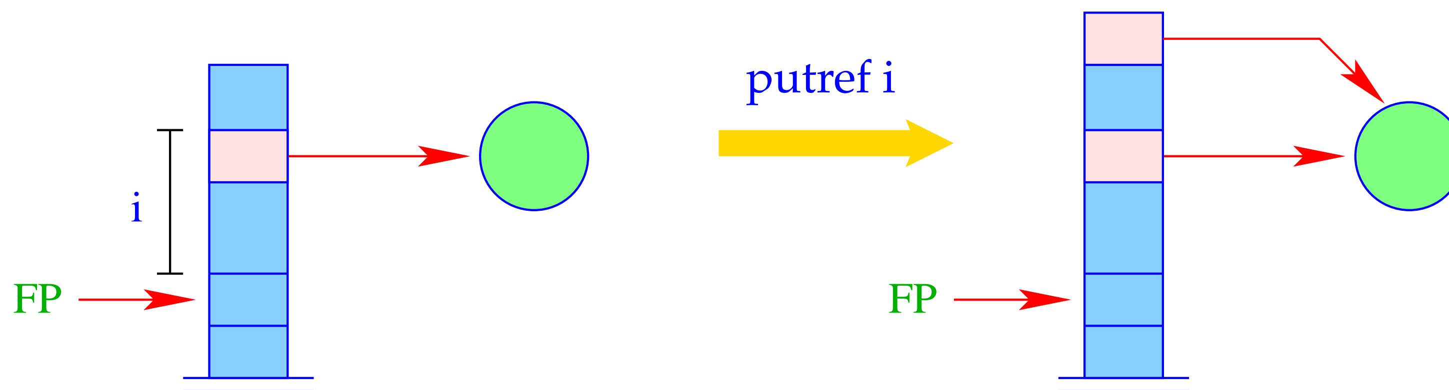
SP = SP + 1;
S[SP] = new (R, HP);

Die Instruktion putref i kopiert den Wert der Variable oben auf den Keller:



SP = SP + 1;
S[SP] = deref S[FP + i];

Die Instruktion putref i kopiert den Wert der Variable oben auf den Keller:

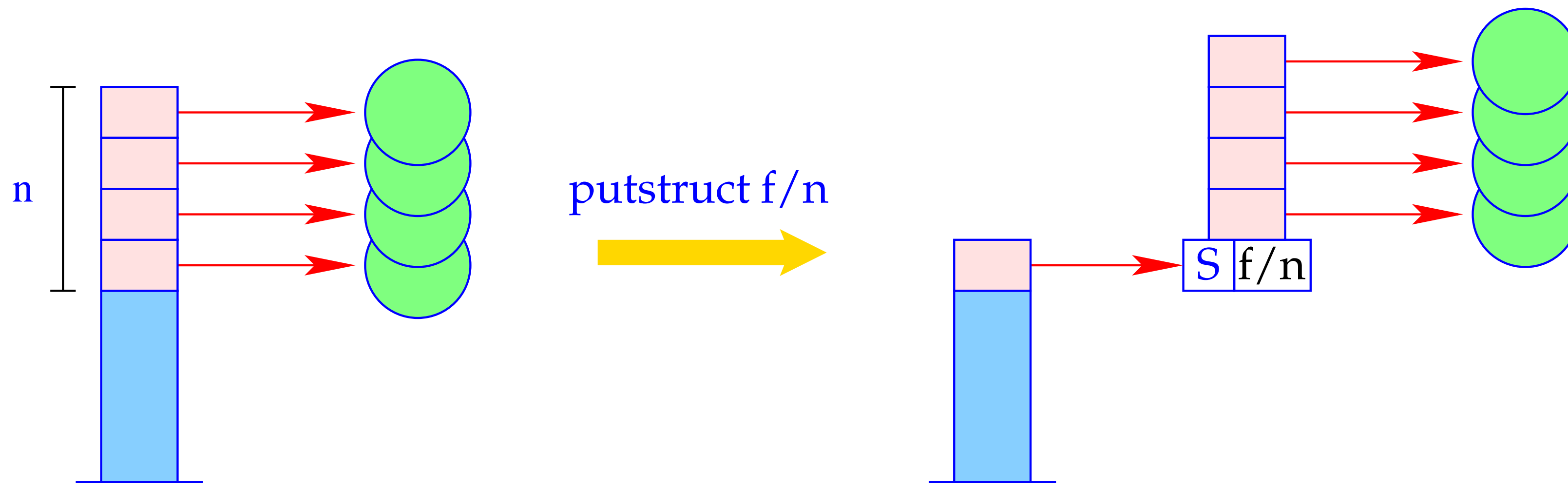


SP = SP + 1;
S[SP] = deref S[FP + i];

Die Hilfsfunktion `deref()` verkürzt dabei Referenz-Ketten:

```
ref deref (ref v) {
      if (H[v]==(R,w) && v!=w) return deref (w);
      else return v;
}
```

Die Instruktion putstruct f/n erzeugt eine Konstruktor-Anwendung in der Halde:




```
v = new (S, f, n);
SP = SP - n + 1;
for (i=1; i<=n; i++)
    H[v + i] = S[SP + i -1];
S[SP] = v;
```

Bemerkungen:

- Der Befehl putref i kopiert nicht einfach den Verweis aus $S[FP + i]$ oben auf den Keller, sondern dereferenziert ihn vorher so oft wie möglich $\Longrightarrow$ maximale Verkürzung der Referenz-Ketten.

- In den aufgebauten Termen zeigen die Referenzen stets auf kleinere Heap-Adressen.

  Dies wird zwar auch sonst oft, leider aber nicht immer zu garantieren sein :-(

29 Die Übersetzung von Literalen

Idee:

- Literale behandeln wir wie Prozedur-Aufrufe.
- Erst legen wir einen Kellerrahmen an.
- Dann konstruieren wir die aktuellen Parameter in der Halde
- ... und speichern Verweise darauf im Kellerrahmen.
- Dann springen wir den Code für die Prozedure/das Prädikat an.

Folglich:

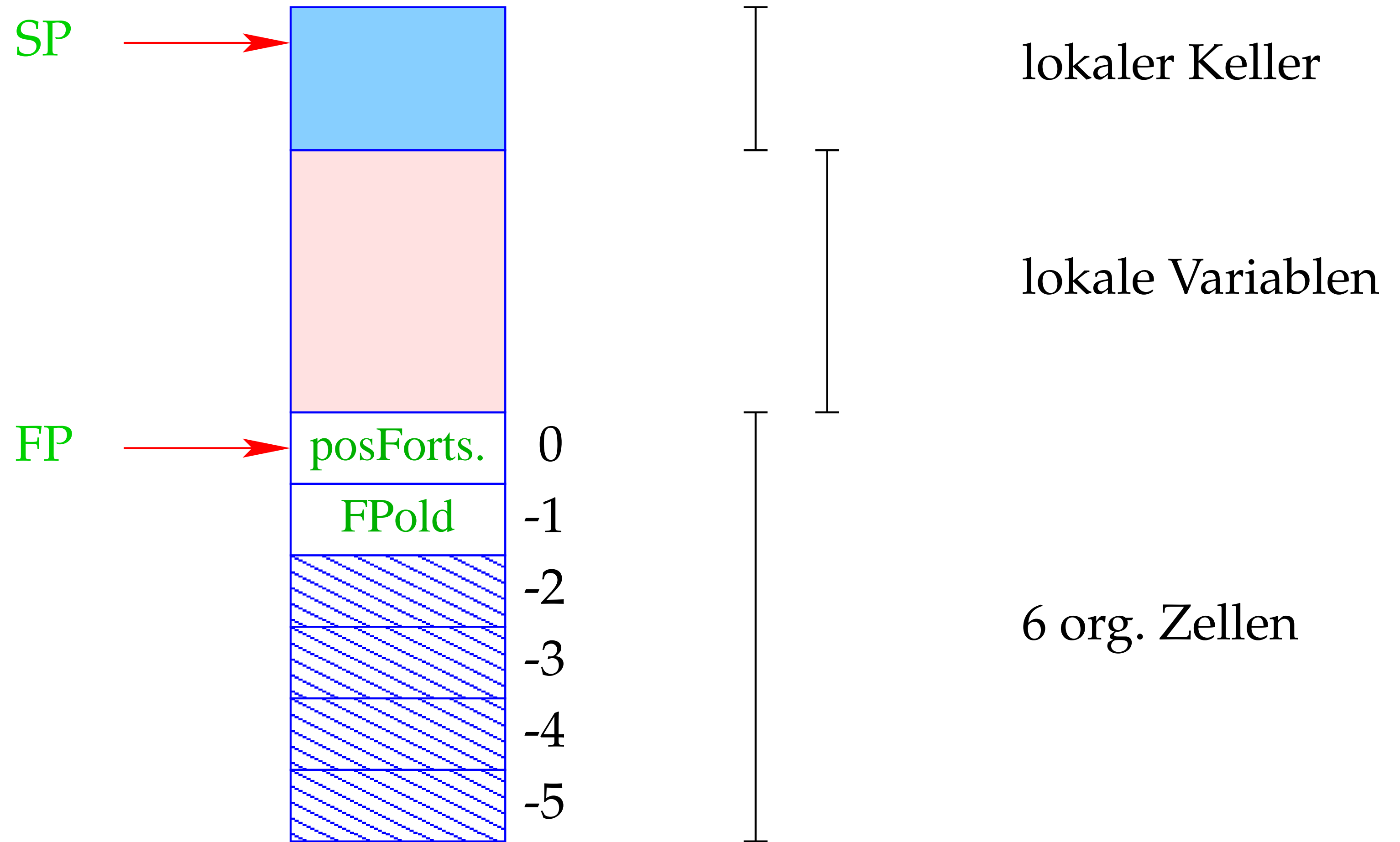
$$
\begin{array}{rll}
\text{code}_G\ p(t_1,\ldots,t_k)\ \rho \quad = & \text{mark B} & \text{// legt einen Kellerrahmen an} \\
& \text{code}_A\ t_1\ \rho & \\
& \ldots & \\
& \text{code}_A\ t_k\ \rho & \\
& \text{call p/k} & \text{// ruft Prozedur p/k auf} \\
\text{B}: & \ldots &
\end{array}
$$

$$\text{code}_G\ p(t_1, \ldots, t_k)\ \rho \quad = \quad \begin{array}{ll} & \text{mark B} \qquad\qquad // \text{ legt einen Kellerrahmen an} \\ & \text{code}_A\ t_1\ \rho \\ & \ldots \\ & \text{code}_A\ t_k\ \rho \\ & \text{call p/k} \qquad\qquad // \text{ ruft Prozedur p/k auf} \\ \text{B}: & \ldots \end{array}$$

Beispiel: $g \equiv p(a, X, g(\bar{X}, Y))$. Mit $\rho = \{X \mapsto 1, Y \mapsto 2\}$ gibt das:

```
   mark B          putref 1          call p/3
   putatom a       putvar 2      B:  ...
   putvar 1        putstruct g/2
```

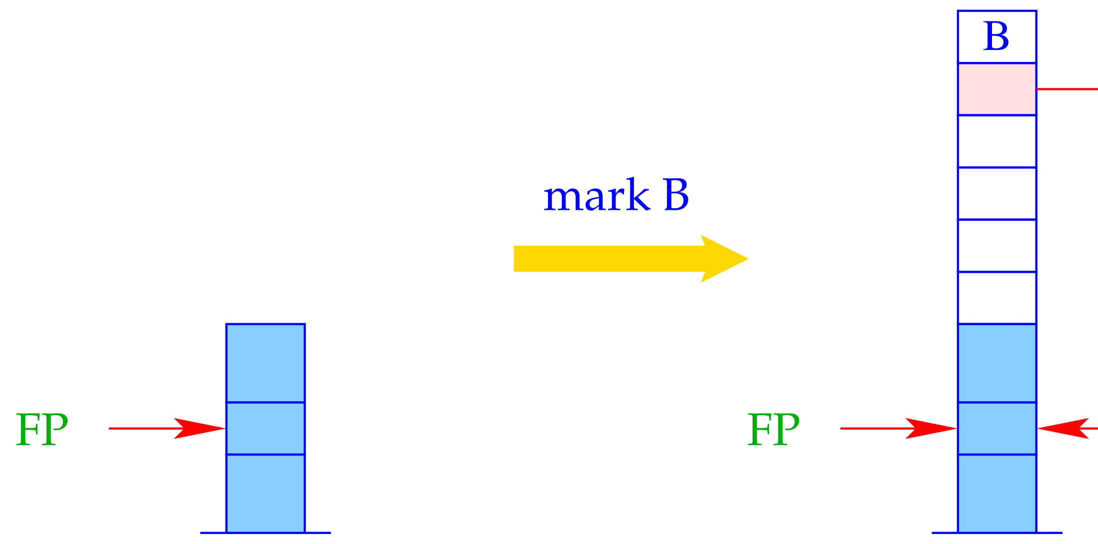
Ein Kellerrahmen der WiM hat den folgenden Aufbau:


SP
lokaler Keller
lokale Variablen
FP
posForts.
0
FPold
-1
-2
-3
-4
-5
6 org. Zellen

Bemerkungen:

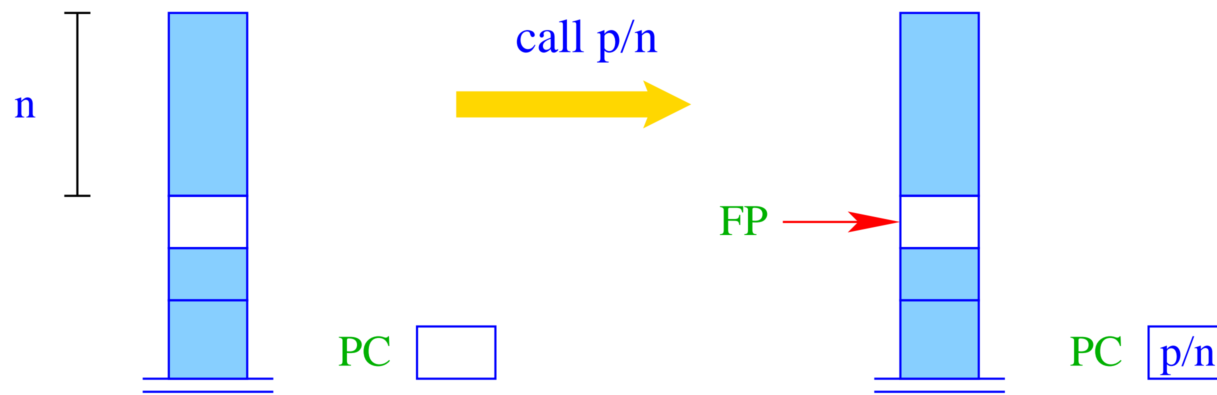
- Die positive Fortsetzungsadresse gibt an, an welcher Stelle im Code fortgefahren werden soll, wenn das Ziel erfolgreich abgearbeitet wurde.
- Die Zelle FPold enthält einen Verweis auf den Rahmen des Aufrufers.
- Die zusätzlichen organisatorischen Zellen benötigen wir zur Implementierung des Backtracking $\Longrightarrow$ Diese werden erst bei der Übersetzung von Prädikaten eingeführt, gesetzt und modifiziert :-)

Die Instruktion mark B allokiert einen neuen Kellerrahmen:



SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

Die Instruktion call p/n ruft das n-stellige Prädikat p auf:



FP = SP − n;
PC = p/n;

30 Unifikation

Konventionen:

- Mit $\tilde{X}$ bezeichnen wir ein Vorkommen von X, das entweder initialisiert ist oder nicht.
- Wir führen die Abkürzung put $\tilde{X}$ ρ ein:

$$\begin{array}{lll} \text{put } X\ \rho & = & \text{putvar } (\rho\, X) \\ \text{put } _\ \rho & = & \text{putanon} \\ \text{put } \bar{X}\ \rho & = & \text{putref } (\rho\, X) \end{array}$$

Wir wollen nun $\tilde{X} = t$ übersetzen.

Idee 1:

- Kellere eine Referenz auf (die Bindung von) X;
- konstruiere den Term t auf der Halde;
- erfinde eine neue Instruktion, die die Unifikation implementiert :-)

Wir wollen nun $\tilde{X} = t$ übersetzen.

Idee 1:

- Kellere eine Referenz auf (die Bindung von) X;
- konstruiere den Term t auf der Halde;
- erfinde eine neue Instruktion, die die Unifikation implementiert :-)

$$\begin{array}{lll} \text{code}_G\ (\tilde{X} = t)\ \rho & = & \text{put}\ \tilde{X}\ \rho \\ & & \text{code}_A\ t\ \rho \\ & & \text{unify} \end{array}$$

Beispiel:

Betrachte die Gleichung:

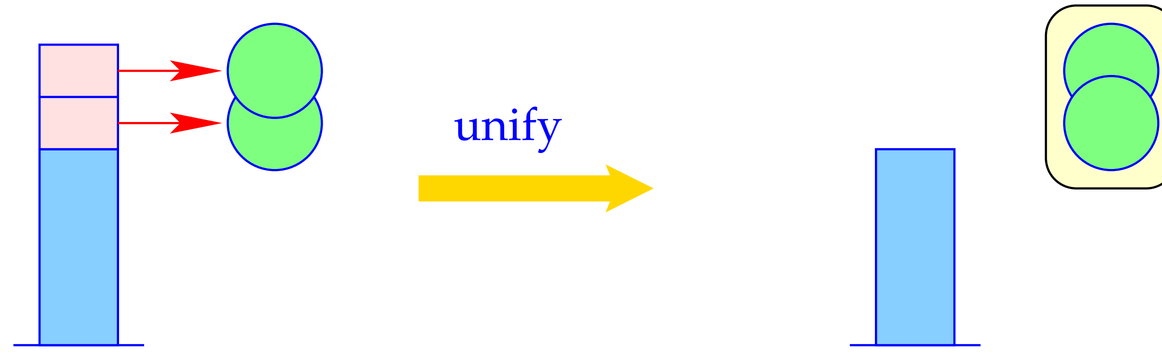
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Dann erhalten wir für die Adress-Umgebung

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

putref 4	putref 1	putatom a	unify
	putvar 2	putvar 3	
	putstruct g/2	putstruct f/3	

Die Instruktion unify ruft die Hilfsfunktion `unify()` für die beiden obersten Kellerreferenzen auf:



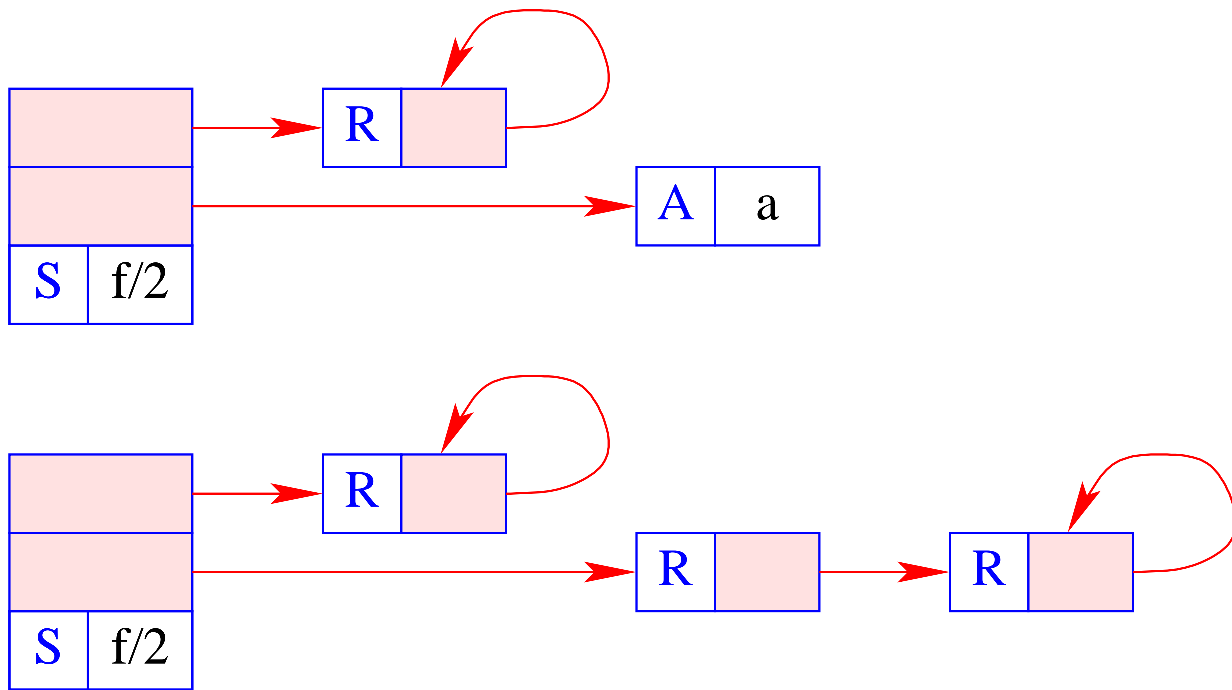

```
unify (S[SP-1], S[SP]);
SP = SP–2;
```

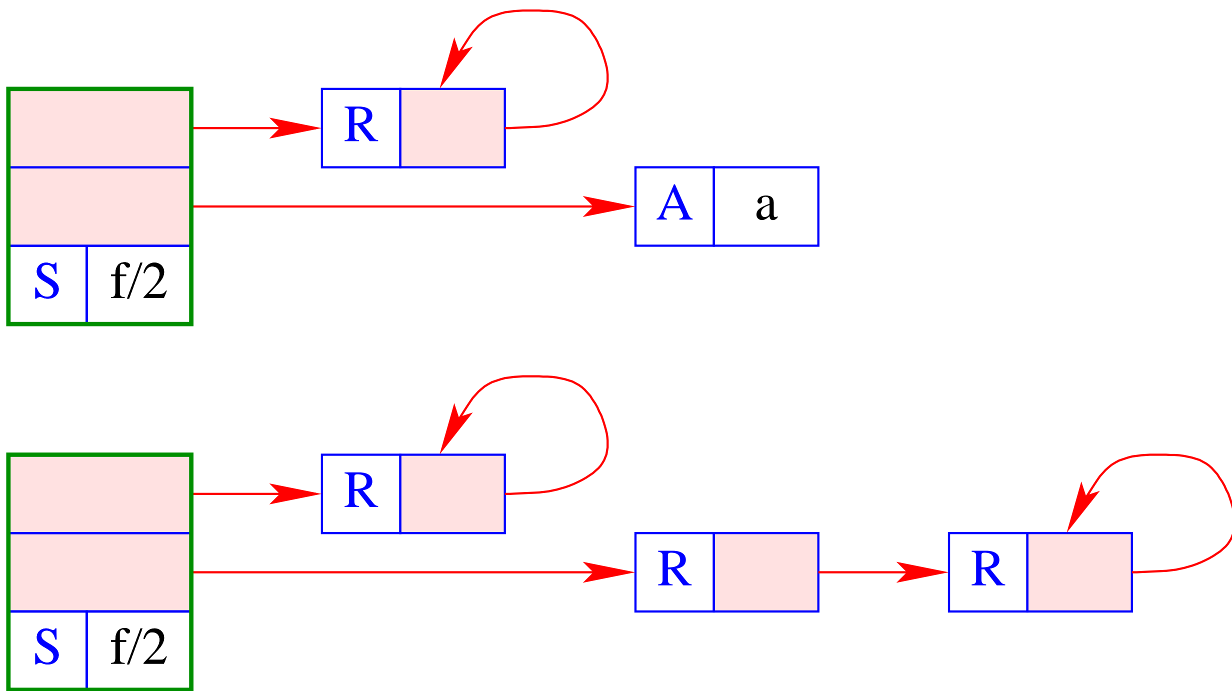
Die Funktion `unify()` des Laufzeit-Systems erhält als Input zwei Heap-Adressen und führt die Unifikation durch. Dabei beachtet sie, dass

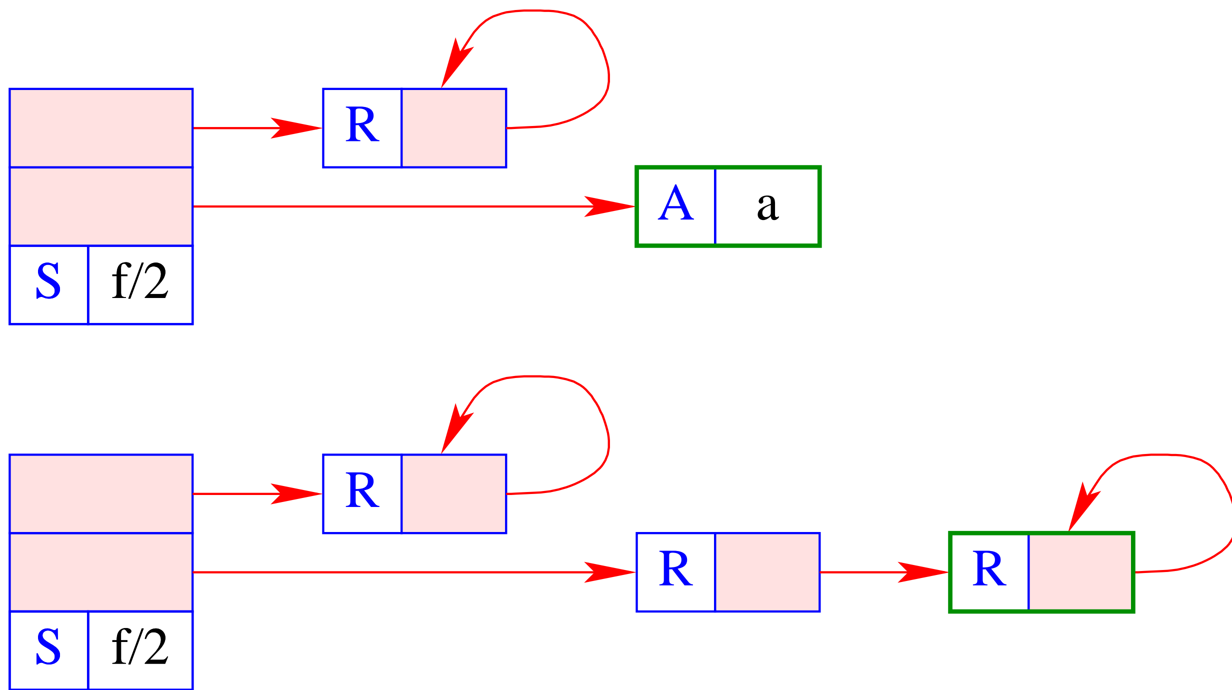
- gleiche Heapadressen bereits unifiziert sind :-)
- beim Binden zweier Variablen aneinander die jüngere (größere Adresse) an die ältere (kleinere Adresse) gebunden wird;
- beim Binden einer Variablen an einen Term diese Variable nicht auch noch im Term vorkommt (Occur Check);
- eingegangene Bindungen mitprotokolliert werden;
- beim Fehlschlagen Backtracking ausgelöst wird.

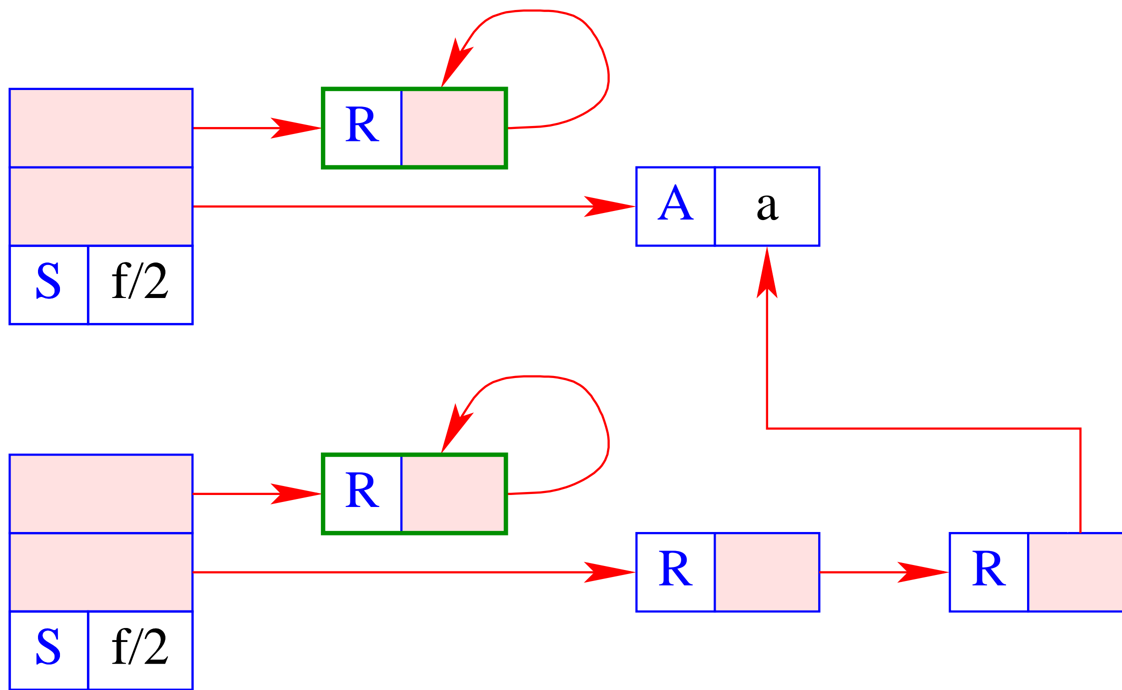
```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u] = (R,v); trail (u); return true;
      } else {
         backtrack(); return false;
      }
   }
   ...
```

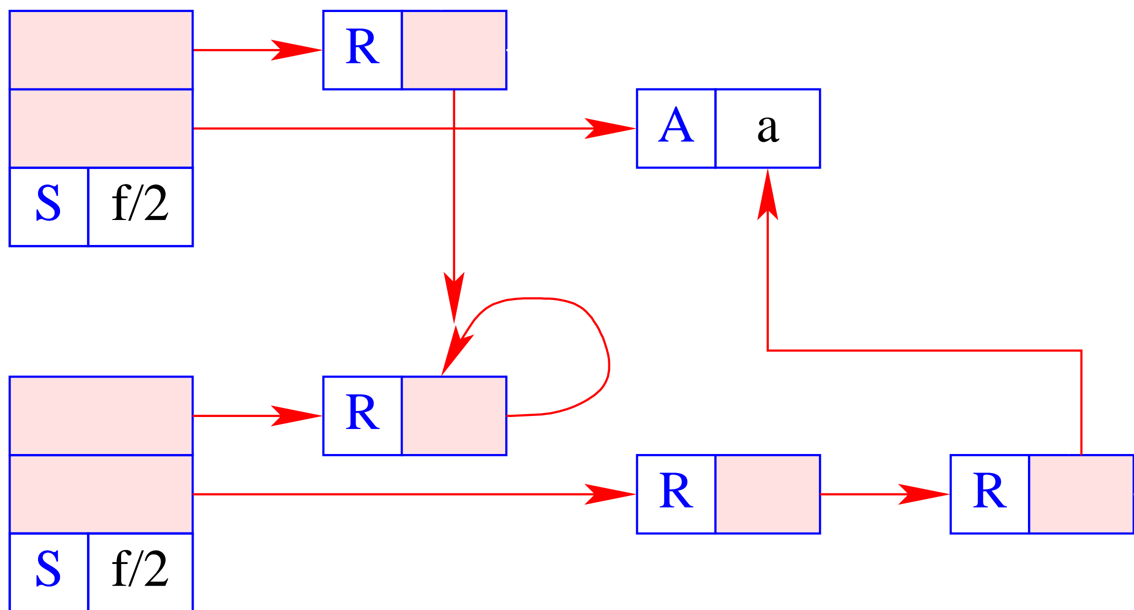
```
...
 if ((H[v] == (R,_)) {
      if (check (v,u)) {
         H[v] = (R,u); trail (v); return true;
      } else {
         backtrack(); return false;
      }
   }
   if (H[u]==(A,a) && H[v]==(A,a))
      return true;
   if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
      for (int i=1; i<=n; i++)
         if(!unify (deref (H[u+i]), deref (H[v+i])) return false;
      return true;
   }
   backtrack(); return false;
}
```


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2


R
A
a
S
f/2
R
R
R
S
f/2

- Die Hilfsfunktion `trail()` vermerkt möglicherweise neue Bindungen.
- Die Hilfsfunktion `backtrack()` initiiert backtracking.
- Die Hilfsfunktion `check()` führt den Occur-Check durch, d.h. überprüft, ob eine Variable in einem Term vorkommt ...
- Oft wird dieser jedoch weggelassen, d.h.

```
bool check (ref u, ref v) { return true;}
```

Ansonsten könnte man ihn wie folgt implementieren:

```
bool check (ref u, ref v) {
   if (u == v) return false;
   if (H[v] == (S, f/n)) {
      for (int i=1; i<=n; i++)
         if (!check(u, deref (H[v+i])))
            return false;
   return true;
}
```

Diskussion:

- Die Übersetzung der Gleichung $\tilde{X} = t$ ist sehr einfach :-)
- Oft werden die gerad konstruierte Halden-Objekte sofort Müll :-(

Idee 2:

- Lege einen Verweis auf die aktuelle Bindung von X oben auf den Keller.
- Vermeide die Konstruktion von Termen soweit möglich!
- Übersetze t in eine Instruktions-Folge, die die Unifikation mit t implementiert !!!

Diskussion:

- Die Übersetzung der Gleichung $\tilde{X} = t$ ist sehr einfach :-)
- Oft werden die gerad konstruierte Halden-Objekte sofort Müll :-(

Idee 2:

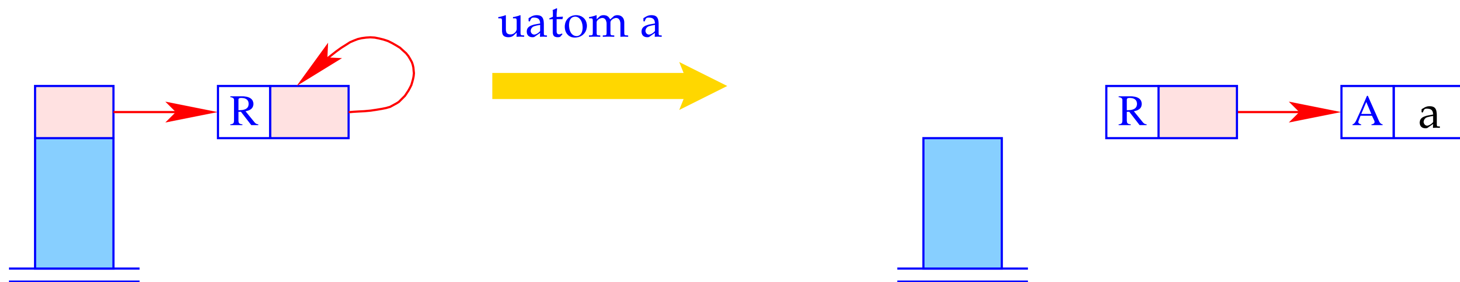
- Lege einen Verweis auf die aktuelle Bindung von X oben auf den Keller.
- Vermeide die Konstruktion von Termen soweit möglich!
- Übersetze t in eine Instruktions-Folge, die die Unifikation mit t implementiert !!!

$$\begin{array}{lll} \text{code}_G\ (\tilde{X} = t)\ \rho & = & \text{put}\ \tilde{X}\ \rho \\ & & \text{code}_U\ t\ \rho \end{array}$$

Betrachten wir zuerst Unifikation nur für Atome und Variablen:

$$
\begin{array}{lll}
\text{code}_U\ a\ \rho & = & \text{uatom a} \\
\text{code}_U\ X\ \rho & = & \text{uvar}\ (\rho\, X) \\
\text{code}_U\ _\ \rho & = & \text{pop} \\
\text{code}_U\ \bar{X}\ \rho & = & \text{uref}\ (\rho\, X) \\
 & \ldots & //\ \text{wird fortgesetzt} \quad :\!-)
\end{array}
$$

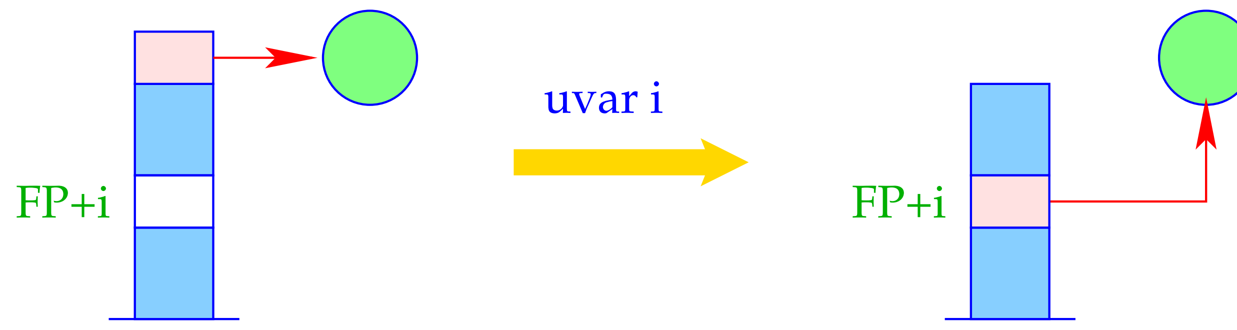
Die Instruktion uatom a implementiert die Unifikation mit dem Atom a:




```
v = S[SP]; SP--;
switch (H[v]) {
case (A, a):     break;
case (R,_) :     H[v] = (R, new (A, a));
                 trail (v); break;
default:         backtrack();
}
```

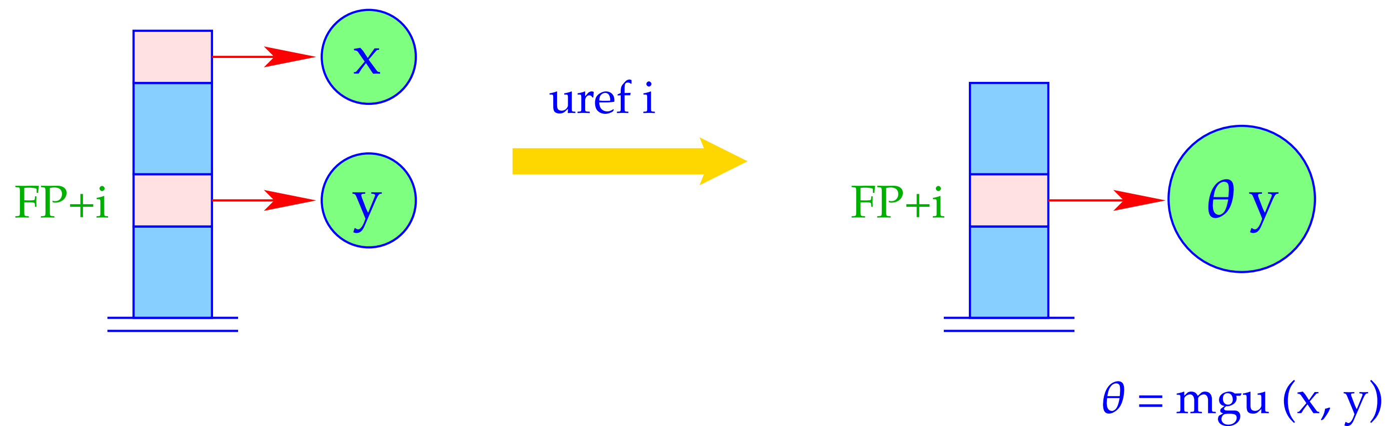
- Der Aufruf von `trail()` vermerkt eine potentiell neue Bindung.
- Der Aufruf von `backtrack()` initiiert backtracking.

Die Instruktion uvar i implementiert die Unifikation mit der *i*-ten Variable, die ungebunden ist. Diese schlägt nie fehl :-)



S[FP+i] = S[SP]; SP− −;

Die Instruktion uref i implementiert die Unifikation mit der *i*-ten Variable, die gebunden ist.



unify (S[SP], deref (S[FP+i]));
SP– –;

Nur hier wird die Laufzeit-Unifikation unify() aufgerufen :-)

- Der Unifikations-Code führt einen pre-order Durchlauf über t durch.
- Trifft er auf eine ungebundene Variable, schalten wir von Testen auf Aufbauen um :-)

```
codeU f(t1,...,tn) ρ  =       ustruct f/n A
                              son 1                      // rekursiver Abstieg
                              codeU t1 ρ
                              ...
                              son n                      // rekursiver Abstieg
                              codeU tn ρ
                              up B                       // Rückkehr zum Vater
                         A :  check ivars(f(t1,...,tn)) ρ   // Occur-Check
                              codeA f(t1,...,tn) ρ
                              bind                       // stellt die Bindung her
                         B :  ...
```

Der Aufbaublock

Vor der Konstruktion der neuen Teilterme t' für die Bindung müssen wir ausschließen, dass sie die Variable X' oben auf dem Keller enthalten !!!

Dies ist genau dann der Fall, wenn eine der Bindungen der in t' vorkommenden Variablen X' enthalten.

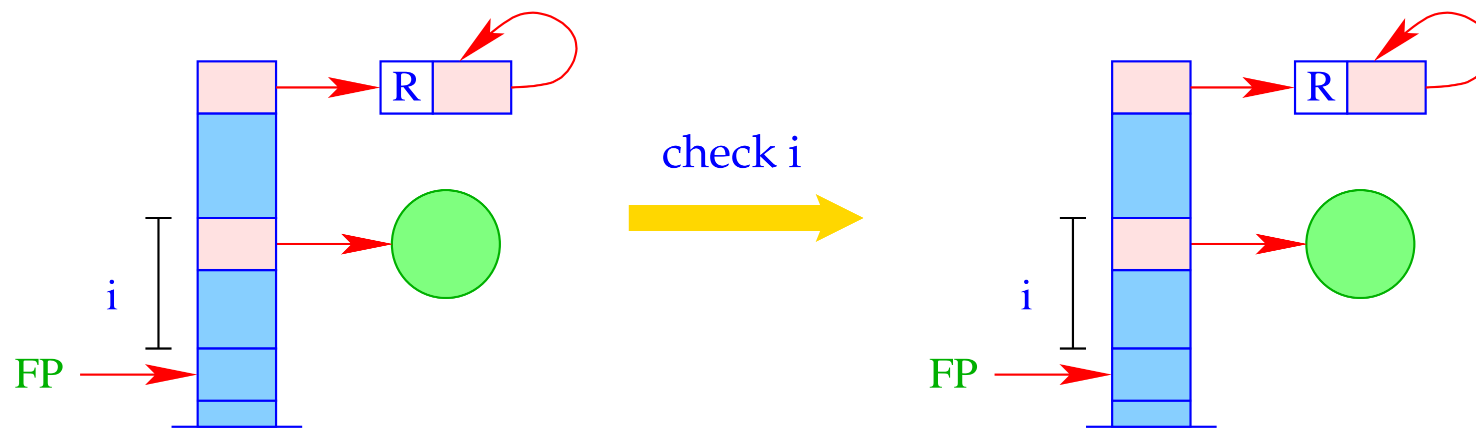
$\Longrightarrow$ $\mathit{ivars}(t')$ liefert die Menge der bereits initialisierten Variablen in t'.

$\Longrightarrow$ Das Macro check $\{Y_1, \ldots, Y_d\}\ \rho$ erzeugt die notwendigen Tests der Variablen $Y_1, \ldots, Y_d$:

$$\begin{array}{lll} \text{check } \{Y_1, \ldots, Y_d\}\ \rho & = & \text{check } (\rho\ Y_1) \\ & & \text{check } (\rho\ Y_2) \\ & & \ldots \\ & & \text{check } (\rho\ Y_d) \end{array}$$

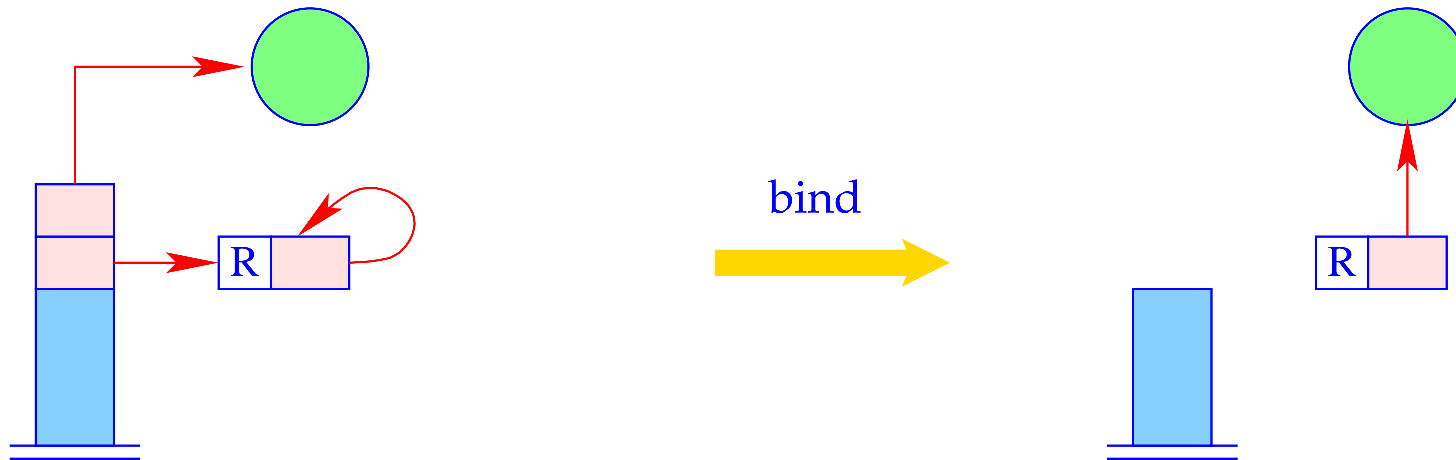
Die Instruktion check i überprüft, ob die (ungebundene) Variable oben auf dem Keller innerhalb des Term vorkommt, an den die Variable i gebunden ist.

Ist dies der Fall, wird Backtracking ausgelöst:




```
if (!check (S[SP], deref S[FP+i]))
    backtrack();
```

Die Instruktion bind schließt den Term-Aufbau ab. Sie bindet die (ungebundene) Variable an den konstruierten Term:

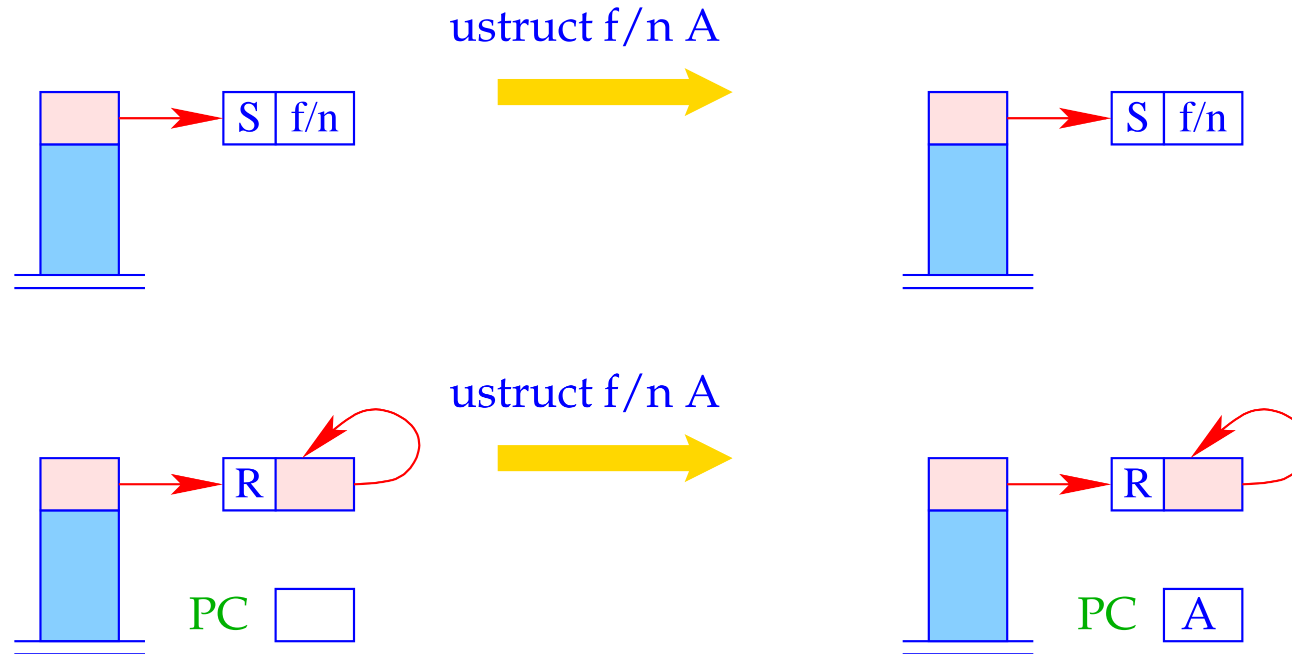



```
H[SP-1] = (R, S[SP]);
trail (S[SP-1]);
SP = SP – 2;
```

Der Pre-Order Durchlauf:

- Zuerst testen wir, ob die oberste Referenz eine ungebundene Variable ist.
  Ist das der Fall, springen wir zum Aufbaublock.
- Andernfalls vergleichen wir den Wurzelknoten mit dem Konstruktor f/n.
- Dann steigen wir rekursiv zu den Kindern ab.
- Anschließend poppen wir den Keller und fahren hinter dem Unifikations-Code fort:

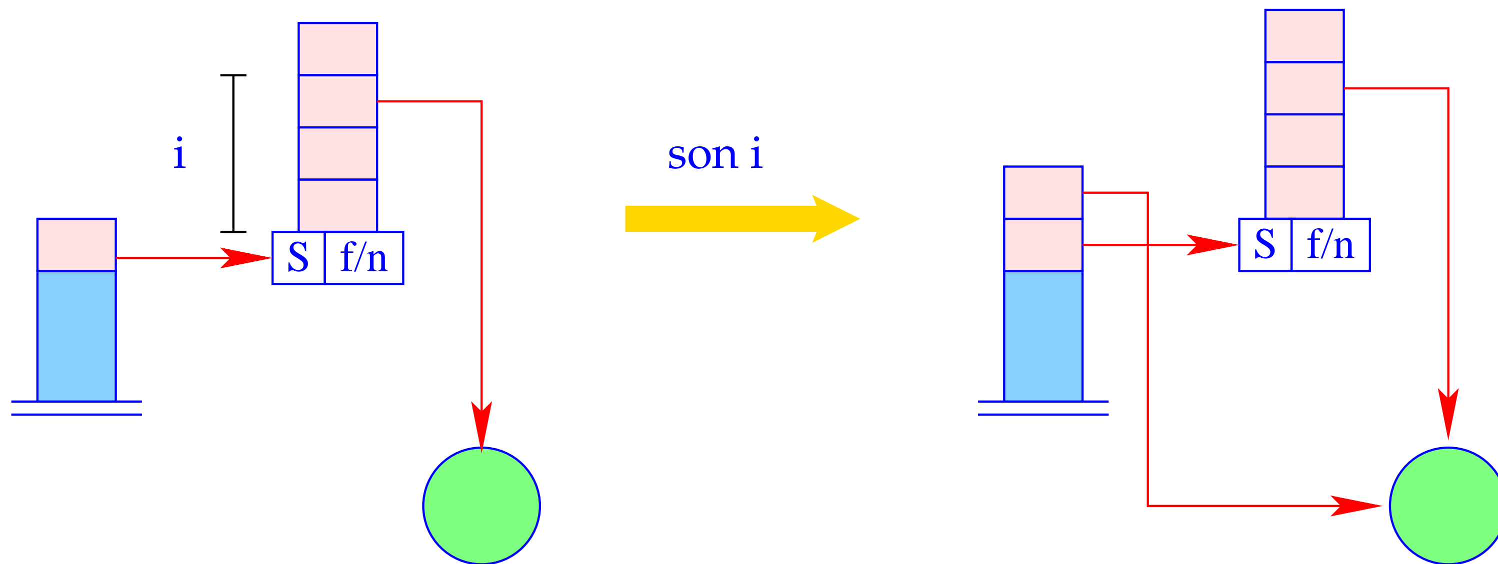
Der Befehl ustruct i implementiert den Test des Wurzel-Knotens der Struktur:




```
switch (H[S[SP]]) {
case (S, f/n):    break;
case (R,_) :      PC = A; break;
default:          backtrack();
}
```

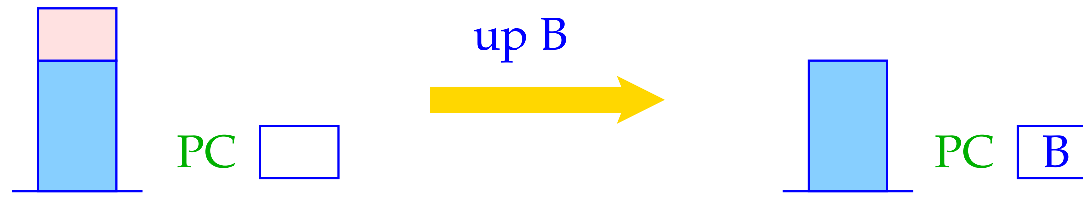
... die Argument-Referenz wurde noch nicht gepoppt :-)

Der Befehl son i kellert (die Referenz auf) den *i*-ten Teilterm der Struktur, auf die die oberste Referenz zeigt:



S[SP+1] = deref (H[S[SP]+i]); SP++;

Es ist schließlich der Befehl up B der die Referenz auf die Struktur poppt:



SP– –; PC = B;

Die Fortsetzungsadresse B ist die erste Adresse hinter dem Aufbaublock.

Beispiel:

Für den Term $t \equiv f(g(\bar{X}, Y), a, Z)$ und die Adress-Umgebung $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ erhalten wir:

ustruct f/3 A_1		up B_2	B_2:	son 2		putvar 2
son 1				uatom a		putstruct g/2
ustruct g/2 A_2	A_2:	check 1		son 3		putatom a
son 1		putref 1		uvar 3		putvar 3
uref 1		putvar 2		up B_1		putstruct f/3
son 2		putstruct g/2	A_1:	check 1		bind
uvar 2		bind		putref 1	B_1:	...

Für tiefe Terme kann die Code-Größe beträchtlich sein. Tiefe Terme sind in der Praxis allerdings “selten” :-)

31 Klauseln

Der Code für eine Klausel muss:

- Platz für die lokalen Variablen allokieren;
- den Rumpf auswerten;
- Kellerplatz freigeben (wann immer möglich :-)

Sei r die Klausel $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$.

Sei $\{X_1, \ldots, X_m\}$ die Menge der lokalen Variablen von r sowie ρ die Adress-Umgebung mit

$$\rho\ X_i = i$$

Bemerkung: Die ersten k lokalen Variablen sind die formalen Parameter :-)

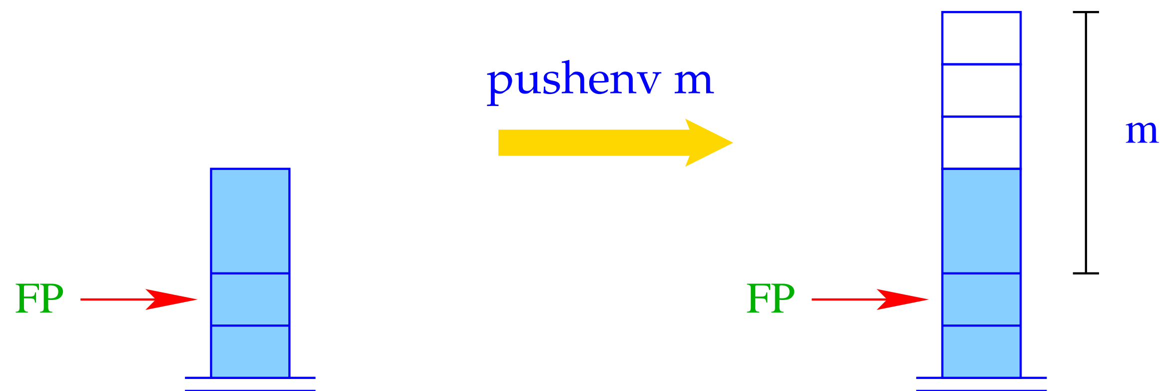
Dann übersetzen wir:

$$\begin{array}{lcll} \text{code}_C\ r & = & \text{pushenv m} & \text{//reserviert Platz für lokale Vars.} \\ & & \text{code}_G\ g_1\ \rho & \\ & & \ldots & \\ & & \text{code}_G\ g_n\ \rho & \\ & & \text{popenv} & \end{array}$$

Der Befehl popenv restauriert FP und PC und versucht den aktuellen Kellerrahmen frei zu geben.

Das sollte gelingen, sofern die Programm-Ausführung nie mehr zu diesem Kellerrahmen zurückkehrt :-)

Der Befehl pushenv m setzt den SP:



SP = FP + m;

Beispiel:

Betrachte die Klausel r:

$$\mathsf{a}(X, Y) \leftarrow \mathsf{f}(\bar{X}, X_1), \mathsf{a}(\bar{X}_1, \bar{Y})$$

Dann liefert $\text{code}_C\ r$:

pushenv 3	mark A	A:	mark B	B:	popenv
	putref 1		putref 3		
	putvar 3		putref 2		
	call $\mathsf{f}/2$		call $\mathsf{a}/2$		

32 Prädikate

Ein Prädikat q/k ist definiert durch eine Folge von Klauseln $rr \equiv r_1 \ldots r_f$.

Die Übersetzung von q/k enthält Übersetzungen der einzelnen Klauseln r_i.

Insbesondere haben wir für $f = 1$:

$$\text{code}_P\; rr \quad = \quad \text{code}_C\; r_1$$

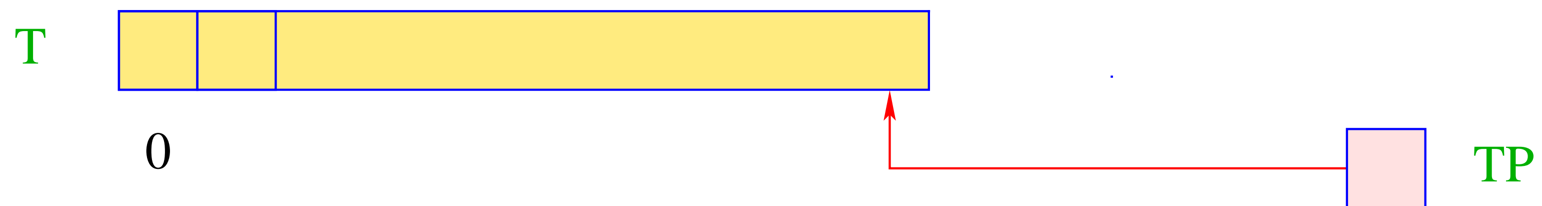
Falls q/k durch mehrere Klauseln definiert ist, muss die erste Alternative ausprobiert werden.

Bei Fehlschlag wird die nächste Alternative probiert ...

$\Longrightarrow$ backtracking :-)

32.1 Backtracking

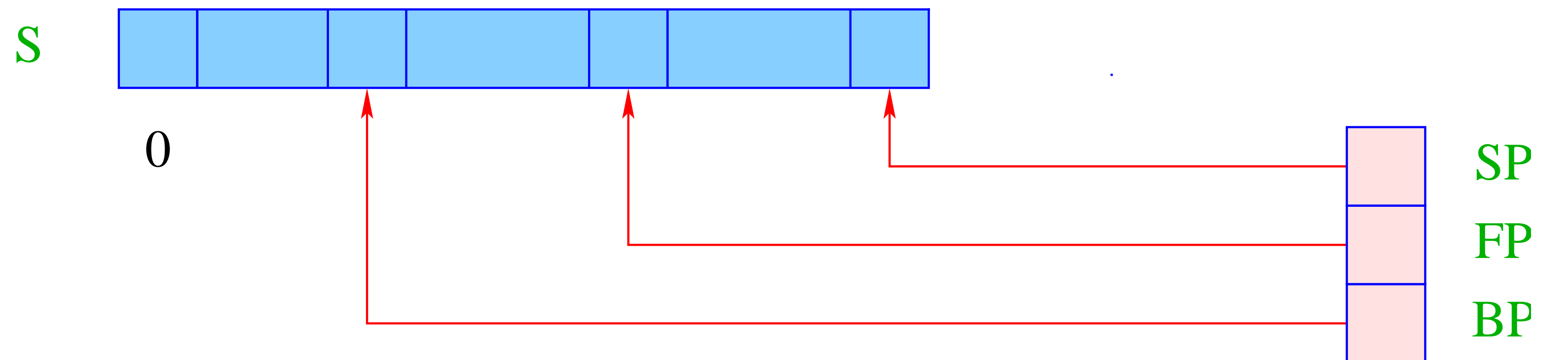
- Wenn die Unifikation fehl schlug, rufen wir die Laufzeit-Funktion `backtrack()` auf.
- Das Ziel ist, die gesamte Berechnung bis zum (dynamisch :-) letzten Ziel rückgängig zu machen, wo eine alternative Klausel gewählt werden kann $\Longrightarrow$ der aktuelle Rücksetz-Punkt.
- Um zwischenzeitlich eingegangene Variablen-Bindungen aufzuheben, haben wir eingegangene neue Bindungen mithilfe der Laufzeit-Funktion `trail()` mitprotokolliert.
- `trail()` speichert Variablen in der Datenstruktur trail:



TP $\quad==\quad$ Trail Pointer

zeigt auf die oberste belegte Trail-Zelle.

Das neue Register BP zeigt auf den aktuellen Rücksetz-Punkt, d.h. den Kellerrahmen, wohin Backtracking aktuell zurück kehren sollte:

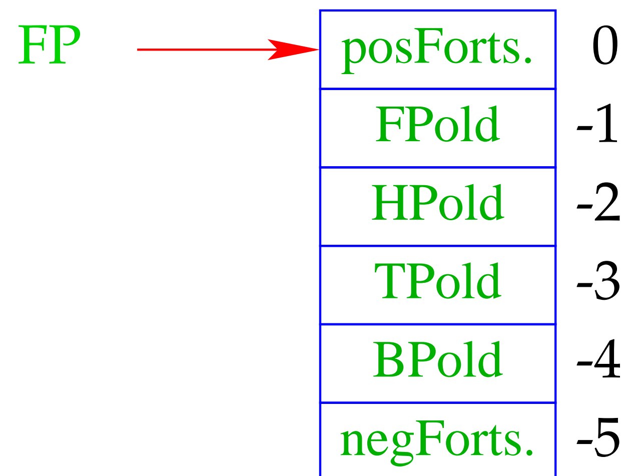


Beachte: Ein neuer BP erhält den Wert des aktuellen FP :-)

Im Kellerrahmen eines Rücksetz-Punkts benötigen wir:

- die Code-Adresse für die nächste Alternative (negative Fortsetzungs-Adresse);
- die alten Werte der Register HP, TP und BP.

Dafür dienen die vier zusätzlichen organisatorischen Zellen:



Zur besseren Lesbarkeit führen wir die folgenden nützlichen Makros ein:

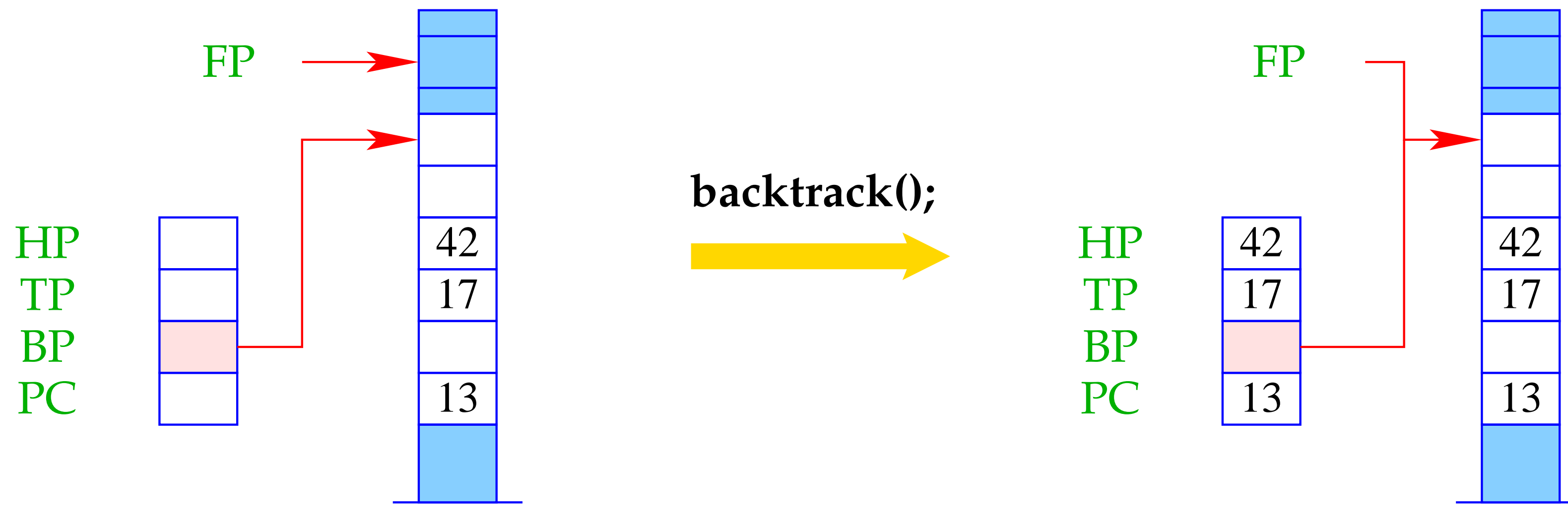
$$\begin{array}{lll} \text{posCont} & \equiv & S[\text{FP}] \\ \text{FPold} & \equiv & S[\text{FP} - 1] \\ \text{HPold} & \equiv & S[\text{FP} - 2] \\ \text{TPold} & \equiv & S[\text{FP} - 3] \\ \text{BPold} & \equiv & S[\text{FP} - 4] \\ \text{negCont} & \equiv & S[\text{FP} - 5] \end{array}$$

Bemerkung:

Vorkommen auf der linken Seite $\implies$ retten der Register

Vorkommen auf der rechten Seite $\implies$ restaurieren der Register

Ein Aufruf der Laufzeit-Funktion `void backtrack()` bewirkt:




```
void backtrack() {
   FP = BP; HP = S[FP-2];
   reset (S[FP-3], TP);
   TP = S[FP-3]; PC = S[FP-4];
}
```

wobei die Laufzeit-Funktion `reset()` die Bindungen der Variablen rückgängig macht, die seit dem Rücksetz-Punkt eingegangen wurden.

32.2 Rücksetzen von Variablen

Idee:

- Die seit dem letzten Rücksetz-Punkt angelegten Variablen und eingegangenen Bindungen beseitigen wir durch Poppen des Heaps !!! :-)
- Leider ist das nur ausreichend, wenn jüngere Variablen stets auf ältere Objekte zeigen.
- Bindungen an jüngere Objekte wurden darum im Trail mitprotokolliert.
- Diese müssen nun einzeln zu Fuß rückgesetzt werden :-(

Die Funktionen `void trail(ref u)` und `void reset (ref y, ref x)` lauten deshalb:

```
void trail (ref u) {
   if (u < S[BP-2]) {
      TP = TP+1;
      T[TP] = u;
   }
}
```

```
void reset (ref x, ref y) {
   for (ref u=y; x<u; u--)
      H[T[u]] = (R,T[u]);
}
```

`S[BP-2]` enthält dabei den Heap-Pointer bei Anlegen des letzten Rücksetz-Punkts.

32.3 Zusammenfügung

Nehmen wir an, das Prädikat q/k sei durch die Klauseln $rr \equiv r_1, \ldots, r_f$ $(f > 1)$ definiert.
Wir benötigen code, um

- den Rücksetz-Punkt zu initialisieren;
- sukzessive die Alternativen zu probieren;
- den Rücksetz-Punkt aufzugeben.

Das bedeutet:

$$
\begin{array}{lcll}
\text{code}_P\ rr & = & \text{q/k}: & \text{setbtp} \\
 & & & \text{try}\ A_1 \\
 & & & \ldots \\
 & & & \text{try}\ A_{f-1} \\
 & & & \text{delbtp} \\
 & & & \text{jump}\ A_f \\
 & & A_1: & \text{code}_C\ r_1 \\
 & & & \ldots \\
 & & A_f: & \text{code}_C\ r_f
\end{array}
$$

Beachte:

- Wir geben den Rücksetz-Punkt vor der letzten Alternative auf :-)
- Wir springen die letzte Alternative direkt an — um nie mehr zu diesem Rahmen zurückzukehren :-))

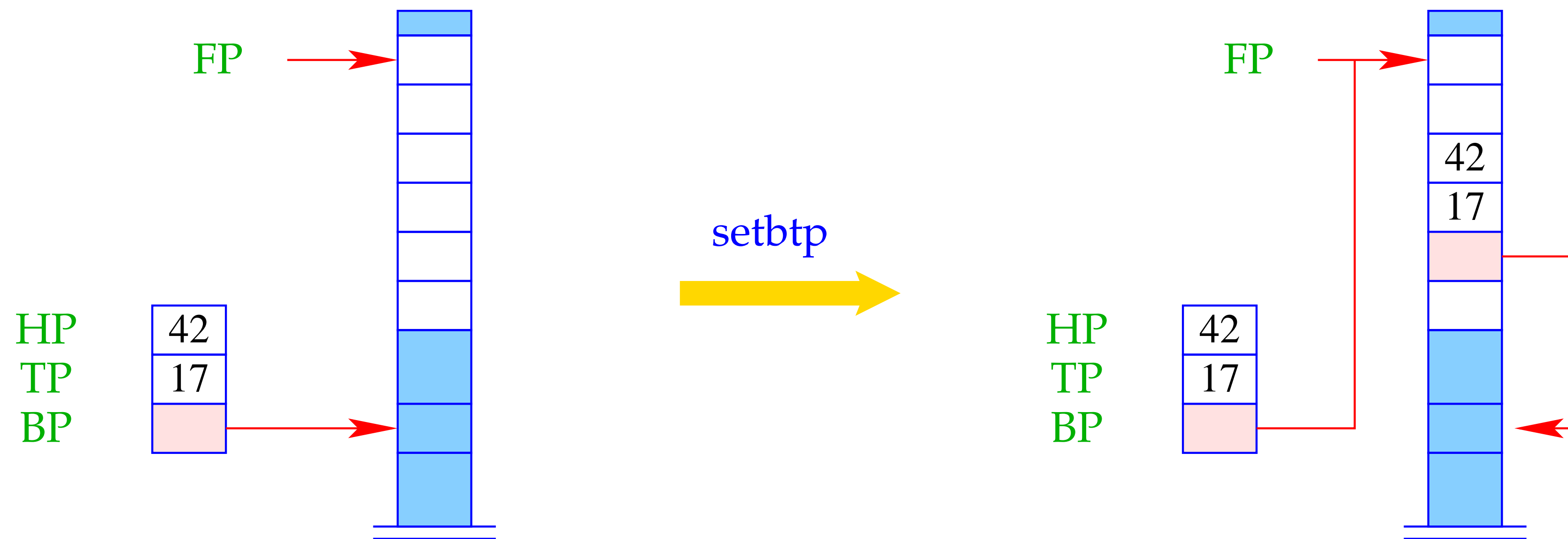
Beispiel:

$$\mathsf{s}(X) \quad \leftarrow \quad \mathsf{t}(\bar{X})$$
$$\mathsf{s}(X) \quad \leftarrow \quad \bar{X} = a$$

Die Übersetzung des Prädikats s liefert:

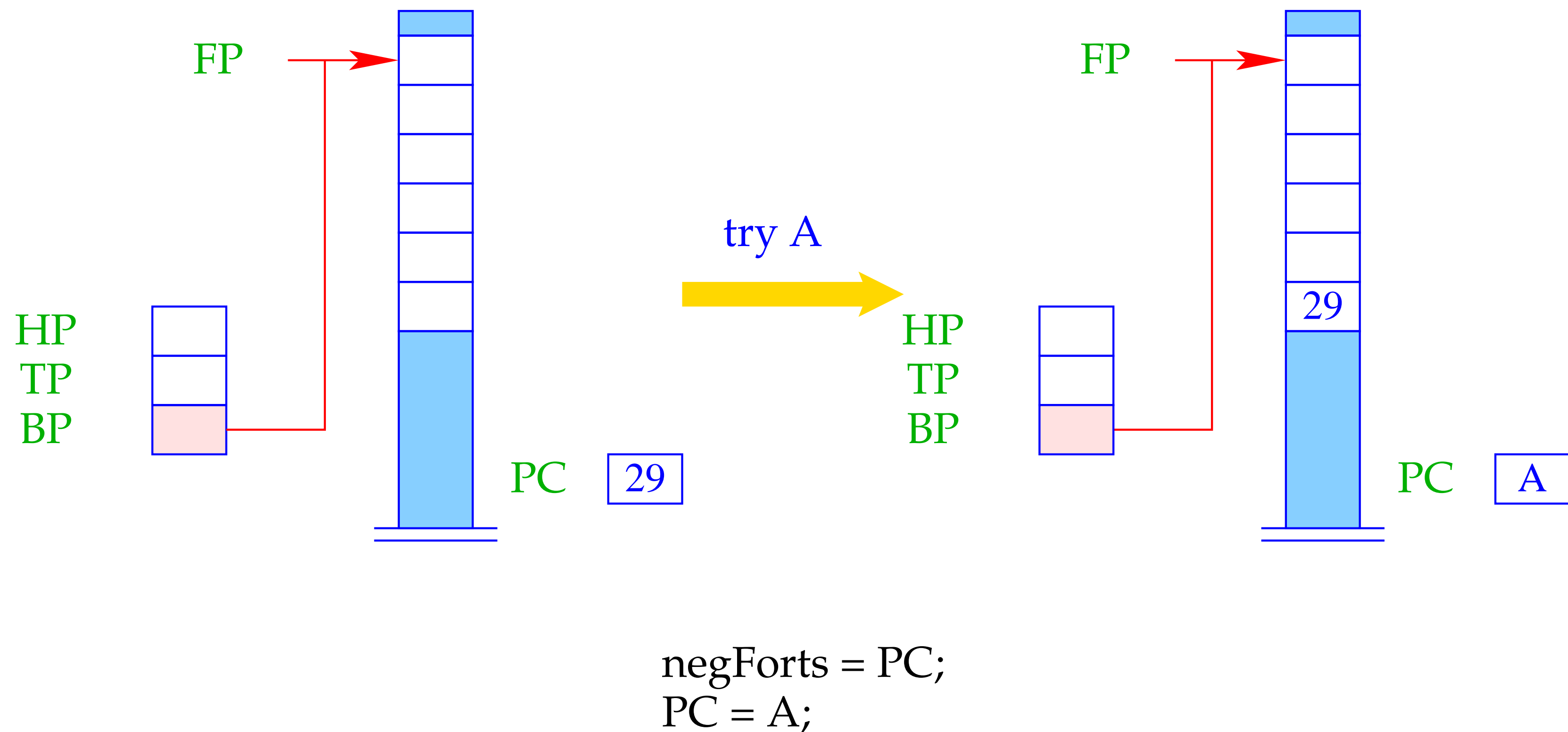
s/1:	setbtp	A:	pushenv 1	B:	pushenv 1
	try A		mark C		putref 1
	delbtp		putref 1		uatom a
	jump B		call t/1		popenv
		C:	popenv		

Der Befehl setbtp rettet die Register HP, TP, BP:



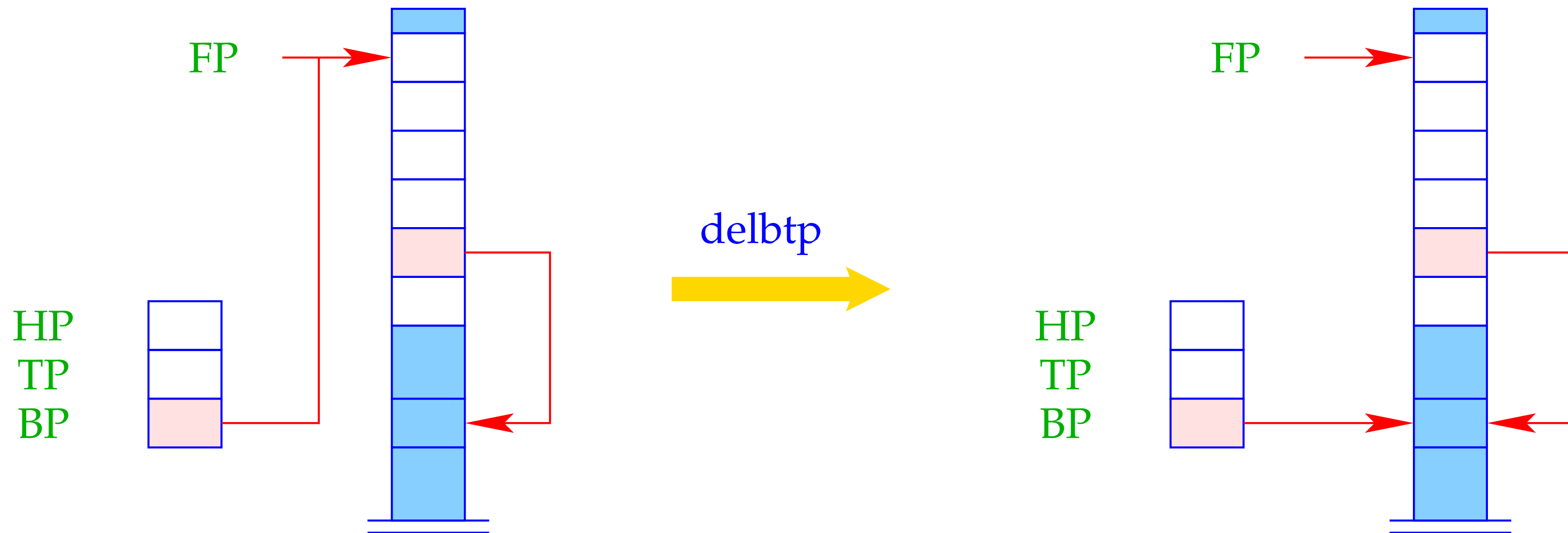
HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;

Der Befehl try A versucht die Alternative an der Adresse A und ersetzt die negative Fortsetzungs-Adress mit dem aktuellen PC:



negForts = PC;
PC = A;

Der Befehl delbtp restauriert den alten Backtrack-Pointer:



BP = S[FP-4];

32.4 Endbehandlung von Klauseln

Erinnern wir uns an die Übersetzung von Klauseln:

$$\begin{aligned} \text{code}_C\ r \quad = \quad & \text{pushenv m} \\ & \text{code}_G\ g_1\ \rho \\ & \ldots \\ & \text{code}_G\ g_n\ \rho \\ & \text{popenv} \end{aligned}$$

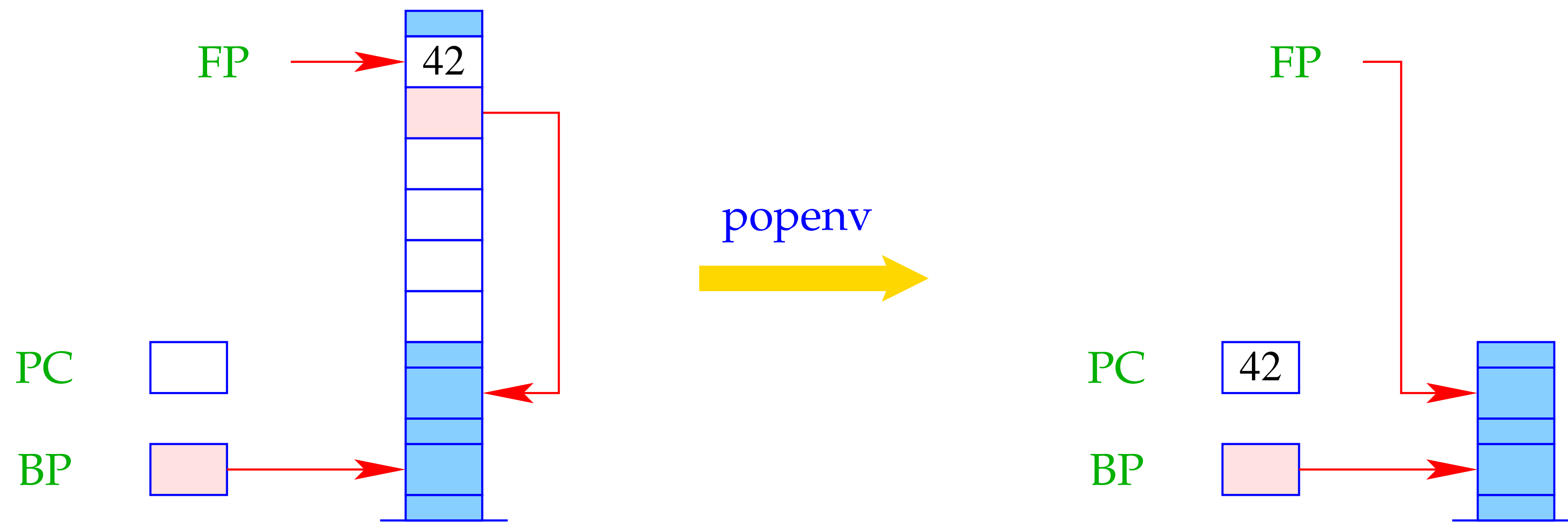
Der aktuelle Kellerrahmen kann aufgegeben werden ...

- falls die aktuelle Klausel die letzte oder einzige ist; und
- falls alle Ziele im Rumpf definitiv beendet sind.

$\Longrightarrow$ der aktuelle Rücksetz-Punkt ist älter :-)

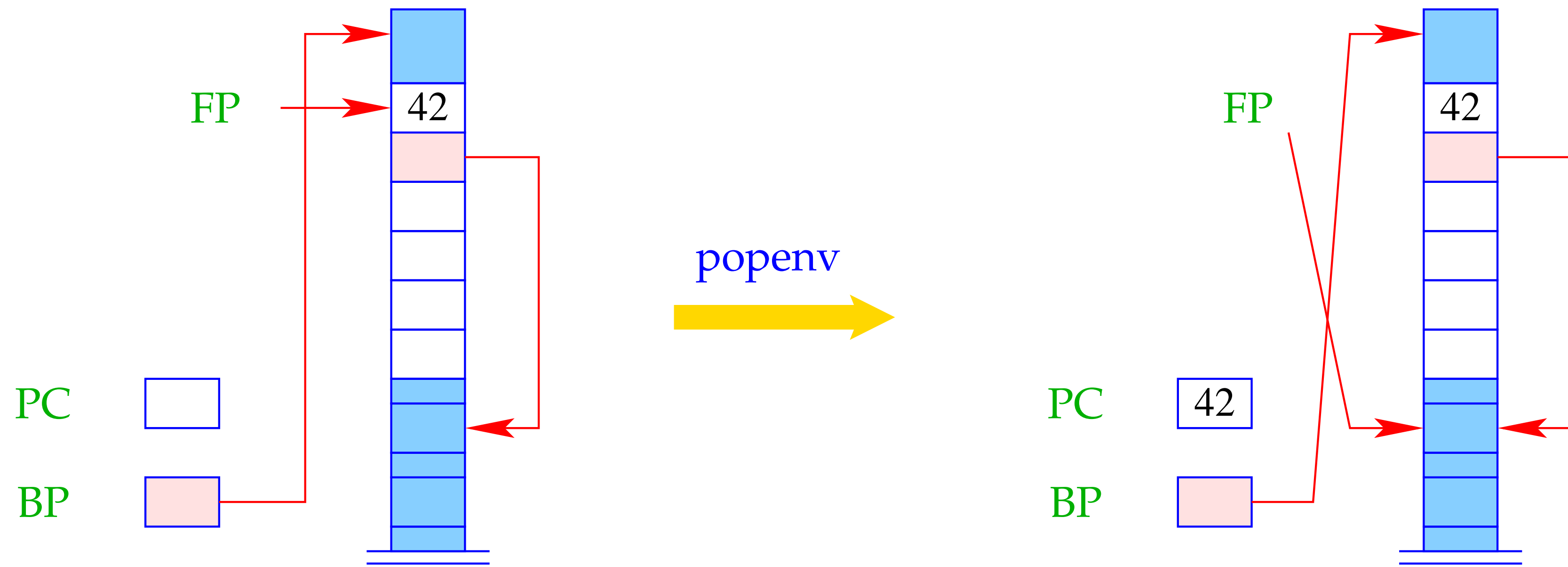
$\Longrightarrow$ FP > BP

Der Befehl popenv restauriert die Register FP und PC und versucht, den Kellerrahmen frei zu geben:



if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;

Achtung: popenv gibt den Kellerrahmen nicht immer auf :



if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;

Falls die Freigabe scheitert, werden weitere Daten oben auf dem Keller allokiert. Bei Rückkehr zu dem Rahmen können die lokalen Variablen aber immer noch über den FP adressiert werden :-))

33 Anfragen und Programme

Die Übersetzung des Programms $p \equiv rr_1 . \ldots . rr_h \; ?g$ besteht aus:

- Code zur Auswertung der Anfrage $?g$;
- einer Instruktion no für Fehlschlag; sowie
- Code für die Prädikate rr_i.

Vor Auswertung der Anfrage müssen zusätzlich die Register korrekt initialisiert und ein erster Kellerrahmen auf dem Keller anglegt werden.

Danach muss die Ergebnis-Substitution zurück gegeben (oder Fehlschlag gemeldet) werden:

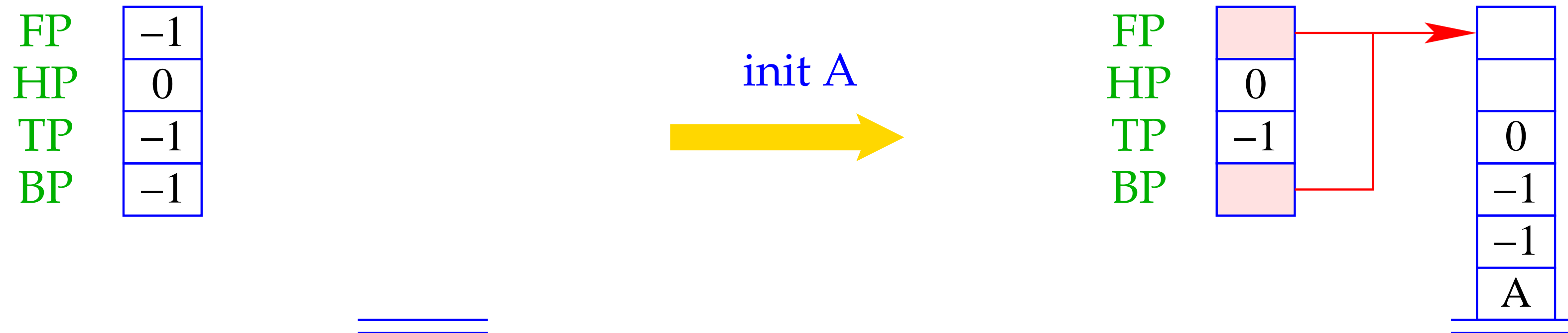
$$
\begin{array}{rrl}
\text{code } p & = & \text{init A} \\
& & \text{pushenv d} \\
& & \text{code}_G\ g\ \rho \\
& & \text{halt d} \\
& \text{A:} & \text{no} \\
& & \text{code}_P\ rr_1 \\
& & \ldots \\
& & \text{code}_P\ rr_h
\end{array}
$$

wobei $free(g) = \{X_1, \ldots, X_d\}$ und die Adress-Umgebung $\rho\ X_i = i$ ist.

Die Instruktion halt d beendet die Programm-Ausführung und stellt die

- ... beendet die Programm-Ausführung;
- ... stellt die Bindungen der d Variablen der Welt zur Verfügung;
- ... löst backtracking aus – sofern von der Benutzerin gefordert :-)

Die Instruktion init A ist definiert durch:



BP = FP = SP = 5;
S[0] = A;
S[1] = S[2] = -1;
S[3] = 0;
BP = FP;

An der Adresse "A" für das Fehlschlagen des Ziels haben wir die Instruktion no untergebracht, welche *no* auf die Standard-Ausgabe schreibt und dann hält.

Das ultimative Beispiel:

$$t(X) \leftarrow \bar{X} = b \qquad q(X) \leftarrow s(\bar{X}) \qquad s(X) \leftarrow \bar{X} = a$$

$$p \leftarrow q(X), t(\bar{X}) \qquad s(X) \leftarrow t(\bar{X}) \qquad ?\ \ p$$

Die Übersetzung liefert:

```
        init N
        pushenv 0
        mark A
        call p/0
A:      halt 0
N:      no
t/1:    pushenv 1
        putref 1
        uatom b
        popenv
p/0:    pushenv 1
        mark B
        putvar 1
        call q/1
B:      mark C
        putref 1
        call t/1
C:      popenv
q/1:    pushenv 1
        mark D
        putref 1
        call s/1
D:      popenv
s/1:    setbtp
        try E
        delbtp
        jump F
E:      pushenv 1
        mark G
        putref 1
        call t/1
G:      popenv
F:      pushenv 1
        putref 1
        uatom a
        popenv
```

34 Letzte Ziele

Betrachte das Prädikat app aus der Einleitung:

$$\begin{array}{lll} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \end{array}$$

Wir beobachten:

- Der rekursive Aufruf ist das letzte Ziel der Klausel.
- Ein solches Ziel ist ein letzter Aufruf :-)

  $\Longrightarrow$ wir versuchen, es im aktuellen Kellerrahmen auszuwerten !!!

  $\Longrightarrow$ nach (erfolgreicher) Beendigung werden wir nicht mehr zum gegenwärtigen Aufrufer zurückkehren !!!

Betrachten wir die Klausel r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$ mit m lokalen Variablen, wobei $g_n \equiv q(t_1, \ldots, t_h)$. Das Zusammenspiel zwischen code_C und code_G:

$$
\begin{array}{rcll}
\text{code}_C\ r & = & & \text{pushenv m} \\
& & & \text{code}_G\ g_1\ \rho \\
& & & \ldots \\
& & & \text{code}_G\ g_{n-1}\ \rho \\
& & & \text{mark B} \\
& & & \text{code}_A\ t_1\ \rho \\
& & & \ldots \\
& & & \text{code}_A\ t_h\ \rho \\
& & & \text{call q/h} \\
& & \text{B}: & \text{popenv}
\end{array}
$$

Ersetzung:	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	lastcall q/h m

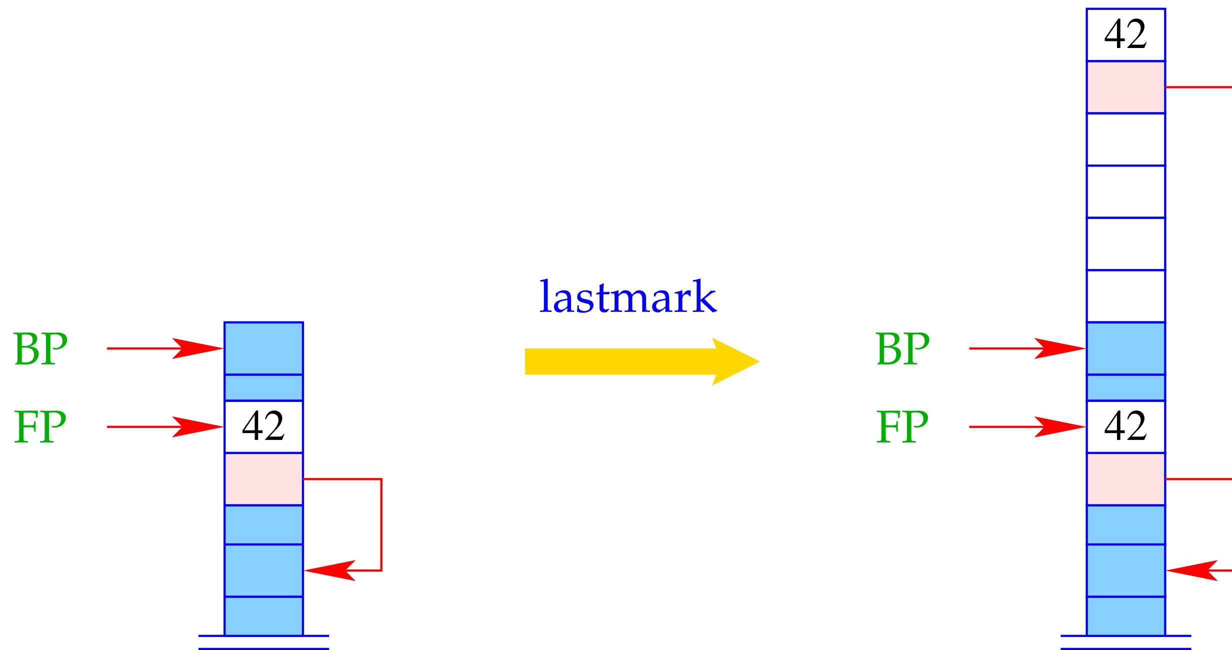
Betrachten wir die Klausel r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$
mit m lokalen Variablen, wobei $g_n \equiv q(t_1, \ldots, t_h)$. Das Zusammenspiel zwischen code_C und code_G:

$$
\begin{array}{lll}
\text{code}_C\ r & = & \text{pushenv m} \\
 & & \text{code}_G\ g_1\ \rho \\
 & & \ldots \\
 & & \text{code}_G\ g_{n-1}\ \rho \\
 & & \text{lastmark} \\
 & & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_h\ \rho \\
 & & \text{lastcall}\ \ \text{q/h}\ \ \text{m}
\end{array}
$$

Ersetzung:	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	lastcall q/h m

Falls die gegenwärtige Klausel nicht letzt ist oder die $g_1, \ldots, g_{n-1}$ Rücksetz-Punkte erzeugt haben, ist FP $\leq$ BP :-)

lastmark legt einen neuen Rahmen an mit einer Referenz auf den Vorgänger:




```
if (FP ≤ BP) {
    SP = SP + 6;
    S[SP] = posCont; S[SP-1] = FPold;
}
```

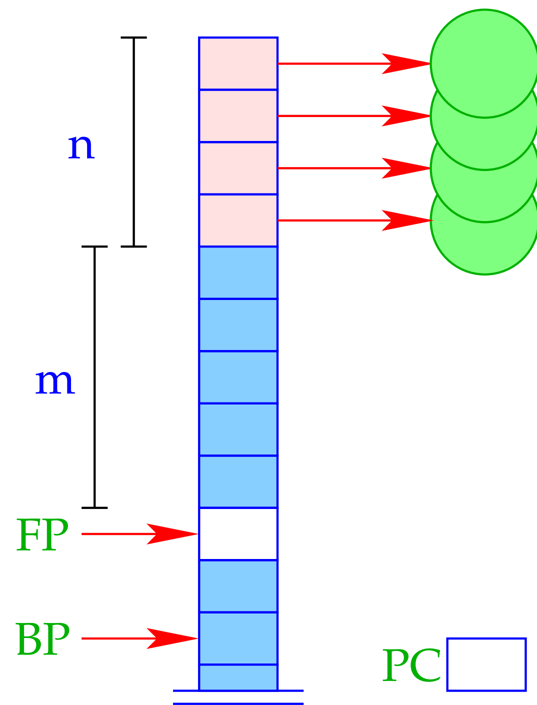
Falls FP $>$ BP dann tut lastmark nichts :-)

Falls $FP \leq BP$, führt lastcall p/h m ein normales call p/h aus.

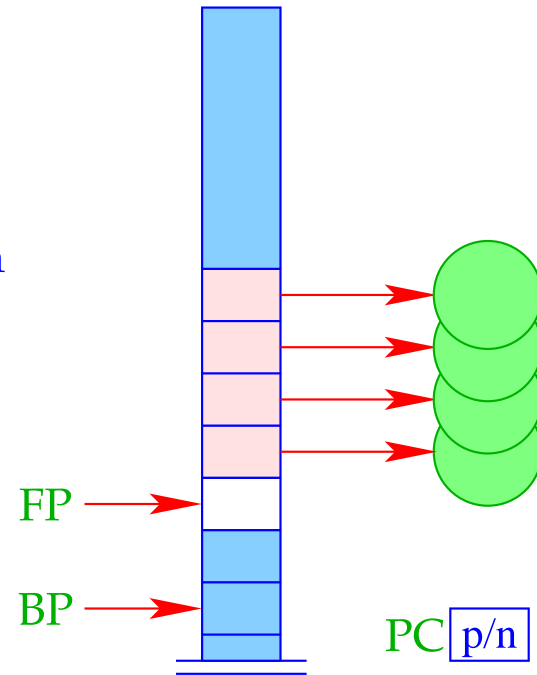
Andernfalls werden die aktuellen Parameter in den Zellen S[FP+1], S[FP+2], . . ., S[FP+h] ausgetauscht und p/h angesprungen.

```
lastcall p/h m   =   if (FP ≤ BP)  call p/h;
                     else {
                        move m h;
                        jump p/h;
                     }
```

Die Differenz zwischen den alten und neuen Adressen der verschobenen Parameter $m = SP–n–FP$ ist gerade gleich der Anzahl der lokalen Variablen im Kellerrahmen.



lastcall p/n m



Beispiel:

Betrachten wir die Klausel r, von eben:

$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Die Optimierung letzter Ziele liefert für $\text{code}_C\ r$:

	mark A	A:	lastmark
pushenv 3	putref 1		putref 3
	putvar 3		putref 2
	call f/2		lastcall a/2 3

Beachte: Ist das letzte Literal der letzten Klausel gleichzeitig das einzige in dieser Klausel, können wir auf lastmark verzichten und lastcall p/n m durch die Folge move m n; jump p/n ersetzen.

Beispiel:

Betrachte die letzte Klausel des Prädikats app:

$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad \bar{X} = [H|X'],\ \bar{Z} = [\bar{H}|Z'],\ \mathsf{app}(\bar{X}', \bar{Y}, \bar{Z}')$$

Hier ist das letzte Ziel das einzige :-) Folglich erhalten wir:

```
A:  pushenv 6
    putref 1
    ustruct [|]/2 B
    son 1
    uvar 4
    son 2
    uvar 5
    up C

B:  putvar 4
    putvar 5
    putstruct [|]/2
    bind

C:  putref 3
    ustruct [|]/2 D
    son 1
    uref 4
    son 2
    uvar 6
    up E

D:  check 4
    putref 4
    putvar 6
    putstruct [|]/2
    bind

E:  putref 5
    putref 2
    putref 6
    move 6 3
    jump app/3
```

35 Verkleinerung von Kellerrahmen

Idee:

- Ordne lokale Variablen gemäß ihrer Lebensdauer;
- Beseitige tote Variablen — falls möglich :-}

35 Verkleinerung von Kellerrahmen

Idee:

- Ordne lokale Variablen gemäß ihrer Lebensdauer;
- Beseitige tote Variablen — falls möglich   :-}

Beispiel:

Betrachte die Klausel:

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

35 Verkleinerung von Kellerrahmen

Idee:

- Ordne lokale Variablen gemäß ihrer Lebensdauer;
- Beseitige tote Variablen — falls möglich :-}

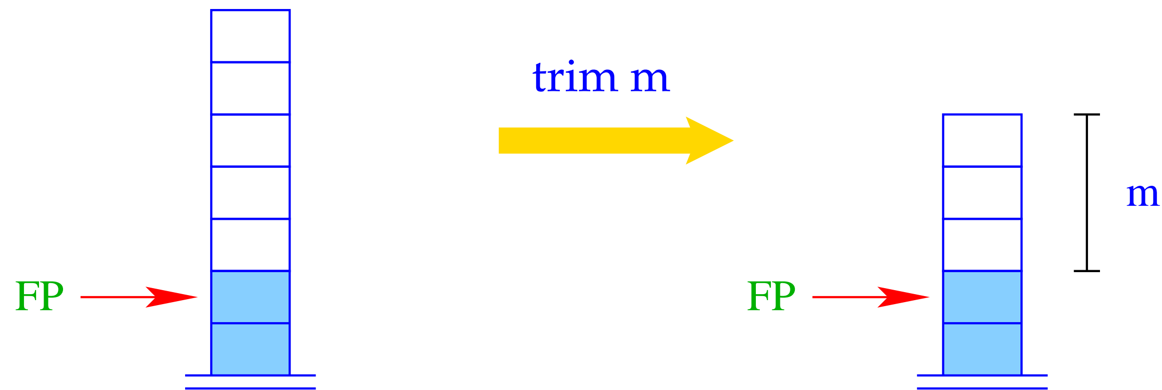
Beispiel:

Betrachte die Klausel:

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

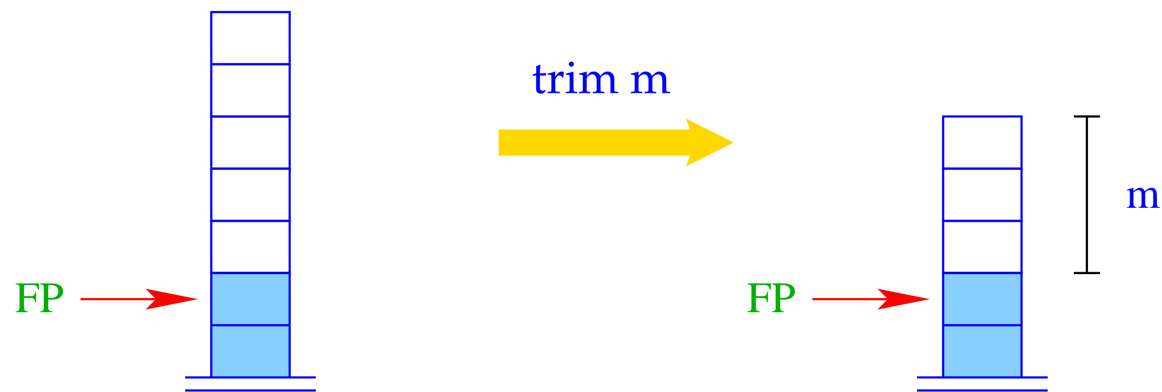
Nach der Anfrage $\mathsf{p}_2(\bar{X}_1, X_2)$ ist die Variable X_1 tot.
Nach der Anfrage $\mathsf{p}_3(\bar{X}_2, X_3)$ ist die Variable X_2 tot :-)

Nach jedem nicht-letzten Ziel mit toten Variablen fügen wir Instruktionen
trim m ein:



if (FP $\geq$ BP)
 SP = FP + m;

Nach jedem nicht-letzten Ziel mit toten Variablen fügen wir die Instruktion trim ein:



if (FP $\geq$ BP)
SP = FP + m;

Die toten lokalen Variablen dürfen nur eliminiert werden, wenn keine Rücksetz-Punkte angelegt wurden :-)

Beispiel (Forts.):

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

Anordnung der Variablen:

$$\rho = \{X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_2 \mapsto 4, X_1 \mapsto 5\}$$

Der resultierende Code:

pushenv 5	A:	mark B		mark C	lastmark
mark A		putref 5		putref 4	putref 3
putref 1		putvar 4		putvar 3	putref 2
putvar 5		call $\mathsf{p}_2/2$		call $\mathsf{p}_3/2$	lastcall $\mathsf{p}_4/2$ 3
call $\mathsf{p}_1/2$	B:	trim 4	C:	trim 3	

36 Klausel-Indizierung

Beobachtung:

Oft werden Prädikate durch Fallunterscheidung nach dem ersten Argument definiert.

$\Longrightarrow$ Berücksichtigung des ersten Arguments kann viele Alternativen ausschließen :-)

$\Longrightarrow$ Fehlschlag wird früher entdeckt :-)

$\Longrightarrow$ Rücksetz-Punkte werden früher beseitigt :-))

$\Longrightarrow$ Kellerrahmen werden früher gepoppt :-)))

Beispiel: Das app-Prädikat:

$$\begin{array}{lll} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z') \end{array}$$

- Falls der Wurzel-Konstruktor $[\,]$ ist, ist nur die erste Klausel anwendbar.
- Falls der Wurzel-Konstruktor $[|]$ ist, ist nur die zweite Klausel anwendbar.
- Jeder andere Wurzel-Konstruktor sollte fehlschlagen !!
- Nur wenn das erste Argument eine ungebundene Variable ist, müssen beide Alternativen probiert werden ;-)

Idee:

- Führe getrennte Try-Ketten für jeden möglichen Konstruktor ein.
- Besichtige den Wurzelknoten des ersten Arguments.
- Abhängig vom Ergebnis, führe einen indizierten Sprung zu der entsprechenden Kette durch.

Angenommen, das Prädikat p/k sei durch die Folge rr von Klauseln $r_1 \ldots r_m$ definiert.

Sei tchains rr die Folge der Try-Ketten entsprechend den Wurzel-Konstruktoren in Unifikationen $X_1 = t$.

Beispiel:

Betrachten wir erneut das app-Prädikat und nehmen an, der Code der beiden Klauseln beginne an den Adressen A_1 und A_2.

Dann erhalten wir die folgenden vier Ketten:

```
VAR:      setbtp      // Variablen      NIL:       jump A1   // Atom [ ]
          try A1
          delbtp                        CONS:      jump A2   // Konstruktor [|]
          jump A2
                                        DEFAULT:   fail      // Default
```

fail ist für alle Konstruktoren außer [ ] und [|] zuständig ...

Beispiel:

Betrachten wir erneut das app-Prädikat und nehmen an, der Code der beiden Klauseln beginne an den Adressen A_1 und A_2.

Dann erhalten wir die folgenden vier Ketten:

VAR:	setbtp	// Variablen	NIL:	jump A_1	// Atom [ ]
	try A_1				
	delbtp		CONS:	jump A_2	// Konstruktor [|]
	jump A_2				
			DEFAULT:	fail	// Default

Die neue Instruktion fail ist für alle Konstruktoren außer [ ] und [|] zuständig ...

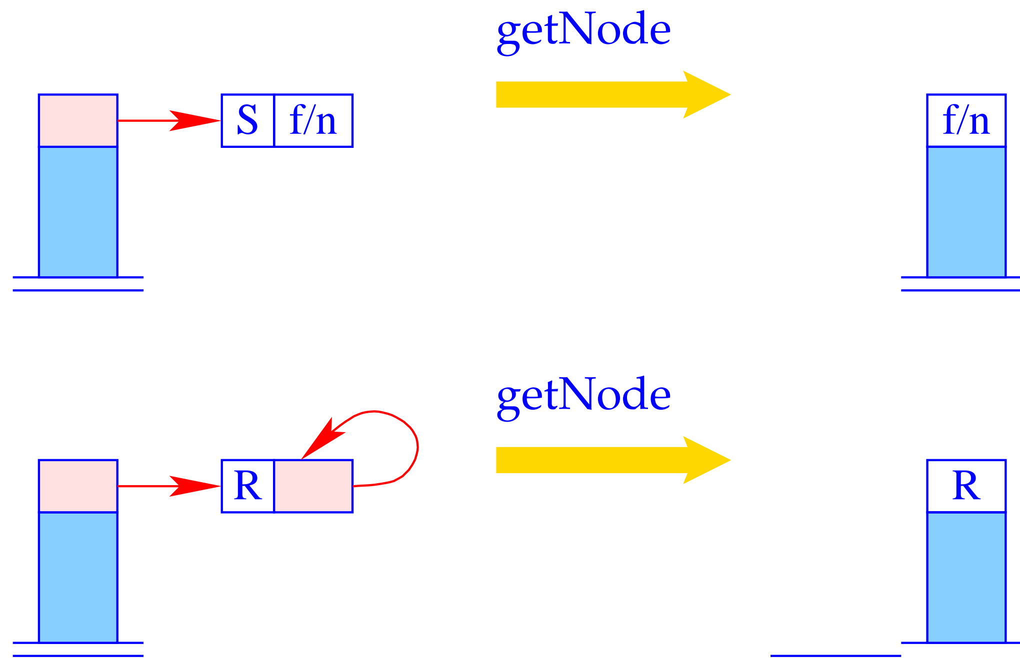
fail = `backtrack()`

... löst direkt backtracking aus :-)

Dann erzeugen wir für das Prädikat p/k:

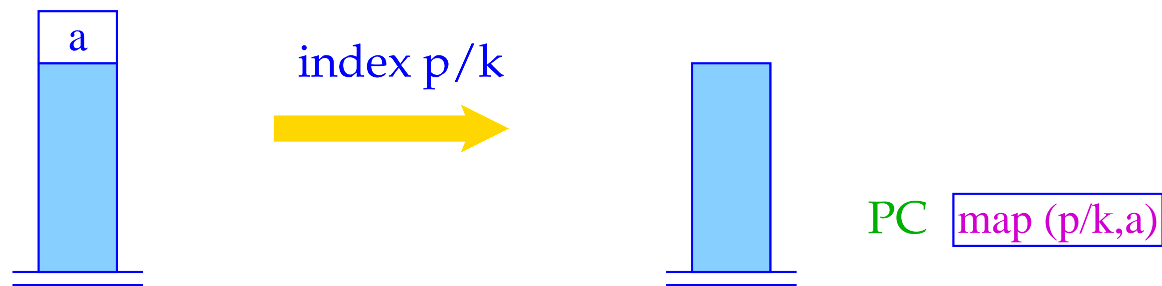
$\text{code}_P\ rr$	=		putref 1
			getNode // extrahiert die Wurzel-Beschriftung
			index p/k // springe zum Try-Block
			tchains rr
		A_1 :	$\text{code}_C\ r_1$
			...
		A_m :	$\text{code}_C\ r_m$

Der Befehl getNode liefert “R”, falls der Verweis oben auf dem Keller auf eine ungebundene Variable zeigt. Andernfalls liefert er den Inhalt des Heap-Objekts:



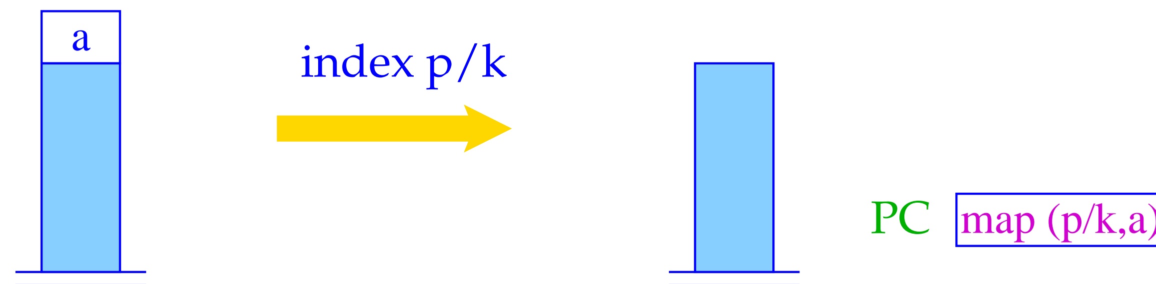

```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```

Der Befehl index p/k führt einen indizierten Sprung an die entsprechende Try-Kette durch:



PC = map (p/k,S[SP]);
SP– –;

Der Befehl index p/k führt einen indizierten Sprung an die entsprechende Try-Kette durch:



PC = map (p/k,S[SP]);
SP– –;

Die Funktion map() liefert zu gegebenem Prädikat und Knoten-Inhalt die Start-Adresse der entsprechenden Try-Kette :-)

Sie wird typischerweise mit einer Hash-Tabelle implementiert :-)

37 Erweiterung: der Cut Operator

Wirkliches Prolog stellt zusätzlich einen Operator “!” (Cut) zur Verfügung, der es erlaubt, den Suchraum für Backtracking explizit zu beschneiden.

Beispiel:

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

Sobald die Anfragen vor dem Cut erfolgreich waren, sind alle getroffenen Auswahlen fest (committed):

Backtracking wird nur noch zu Rücksetz-Punkten vor der Abarbeitung der linken Seite zurück kehren ...

Die grundlegende Idee:

- Wir restaurieren den oldBP des aktuellen Kellerrahmens;
- Wir beseitigen alle Kellerrahmen oberhalb der lokalen Variablen.

Folglich übersetzen wir den Cut in die Folge:

```
prune
pushenv m
```

wobei m die Anzahl der (noch benötigten) localen Variablen der Klausel ist.

Beispiel:

Betrachten wir unser Beispiel:

$$\begin{array}{lcl} \mathsf{branch}(X, Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X, Y) \\ \mathsf{branch}(X, Y) & \leftarrow & \mathsf{q}_2(X, Y) \end{array}$$

Dann erhalten wir:

```
      setbtp     A:  pushenv 2   C:  prune        lastmark          B:  pushenv 2
      try A          mark C          pushenv 2    putref 1              putref 1
      delbtp         putref 1                     putref 2              putref 2
      jump B         call p/1                     lastcall q1/2 2       move 2 2
                                                                        jump q2/2
```

Beispiel:

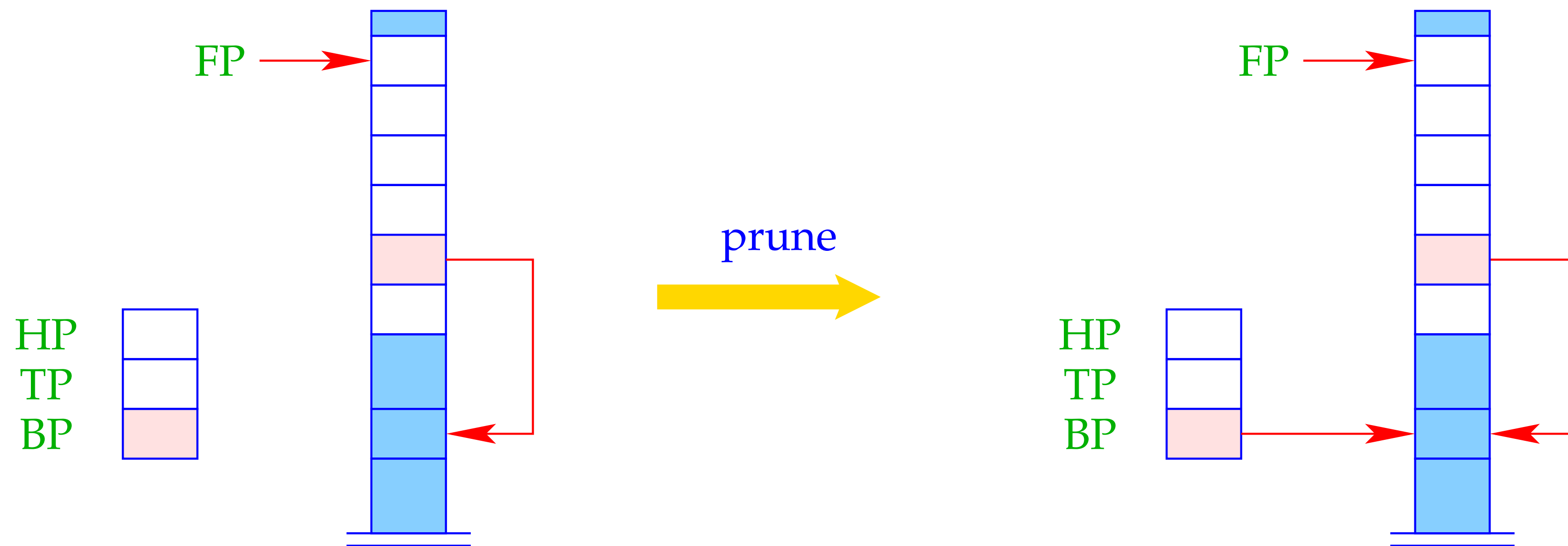
Betrachten wir unser Beispiel:

$$\begin{array}{lcl} \mathsf{branch}(X, Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X, Y) \\ \mathsf{branch}(X, Y) & \leftarrow & \mathsf{q}_2(X, Y) \end{array}$$

Eine optimierte Übersetzung liefert hier sogar:

```
setbtp      A:  pushenv 2   C:  prune       putref 1      B:  pushenv 2
try A           mark C          pushenv 2   putref 2          putref 1
delbtp          putref 1                    move 2 2          putref 2
jump B          call p/1                    jump q1/2         move 2 2
                                                              jump q2/2
```

Die neue Instruktion prune restauriert einfach den BP:



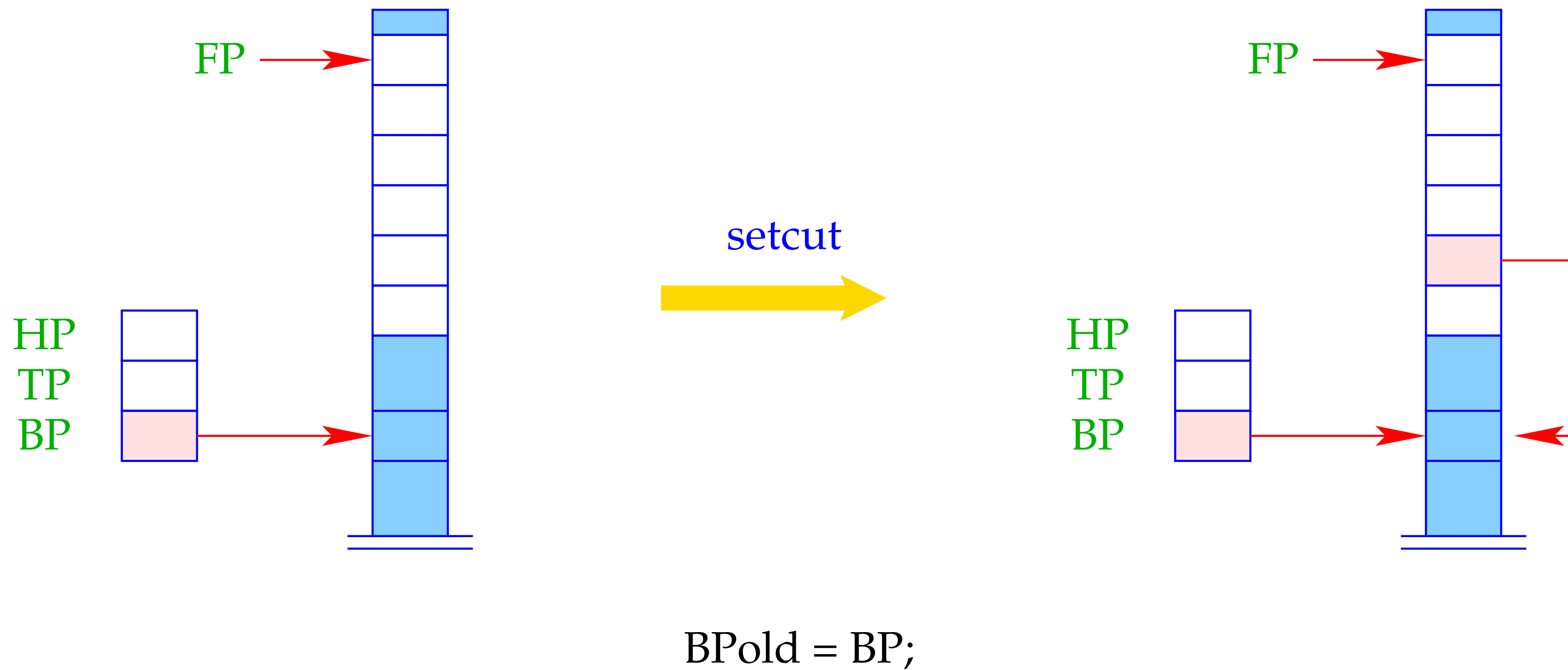
BP = BPold;

Problem:

Ist eine Klausel einzeln, dann haben wir (zumindest bisher ;-) den alten BP noch nicht innerhalb des Kellerrahmens abgelegt :-(

Damit der Cut auch für Prädikate mit einer einzigen Klausel gilt bzw. für Try-Ketten der Länge 1, fügen wir eine extra Instruktion setcut vor dem Klausel-Code (bzw. dem unbedingten Sprung) ein:

Der Befehl setcut rettet den aktuellen Wert des BP:



BPold = BP;

Das allerletzte Beispiel: Negierung durch Fehlschlag

Das Prädikat notP sollte erfolgreich sein, wann immer p fehlschlägt (und umgekehrt :-)

$$\mathsf{notP}(X) \leftarrow \mathsf{p}(X), !, \mathsf{fail}$$
$$\mathsf{notP}(X) \leftarrow$$

wobei das Ziel fail immer fehlschlägt. Dann erhalten wir für notP :

```
     setbtp    A:  pushenv 1   C:  prune       B:  pushenv 1
     try A         mark C          pushenv 1       popenv
     delbtp        putref 1        fail
     jump B        call p/1        popenv
```

38 Garbage Collection

- Sowohl bei der Ausführung eines MaMa- wie eines WiM-Programms können Objekte in der Halde auftreten, auf die es keine Verweise mehr gibt.
- Diese Objekte heißen Müll (Garbage) und können offenbar die weitere Programm-Ausführung nicht mehr beeinflussen.
- Ihr Speicherplatz sollte frei gegeben und für das Anlegen anderer Objekte wiederverwendet werden.

Achtung:

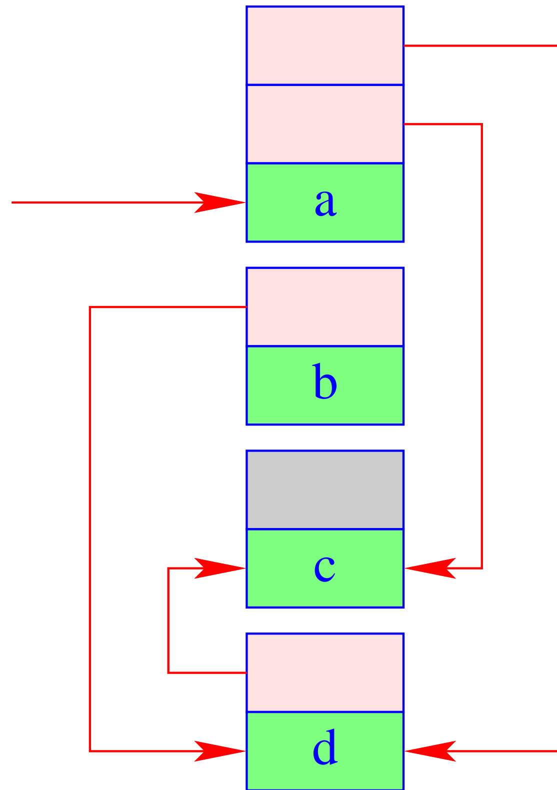
Die WiM verfügt zwar über eine Art von Speicherplatz-Freigabe. Diese gibt jedoch nur den Platz fehlgeschlagener Alternativen frei !!!

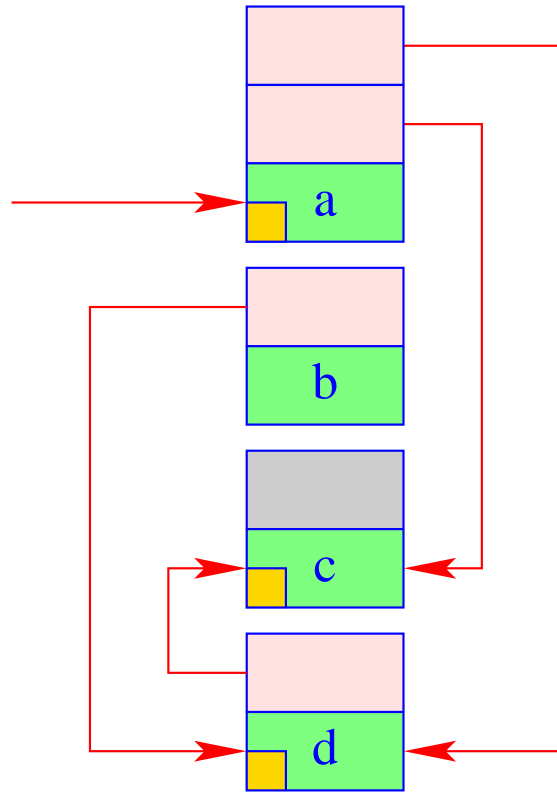
Vorgehen eines kopierenden Kollektors:

(1) Auffinden der noch lebendigen Objekte:

- alle Referenzen im Keller zeigen auf lebendige Objekte;
- jede Referenz eines lebendigen Objekts zeigt auf ein lebendiges Objekt.

Graph-Erreichbarkeit.

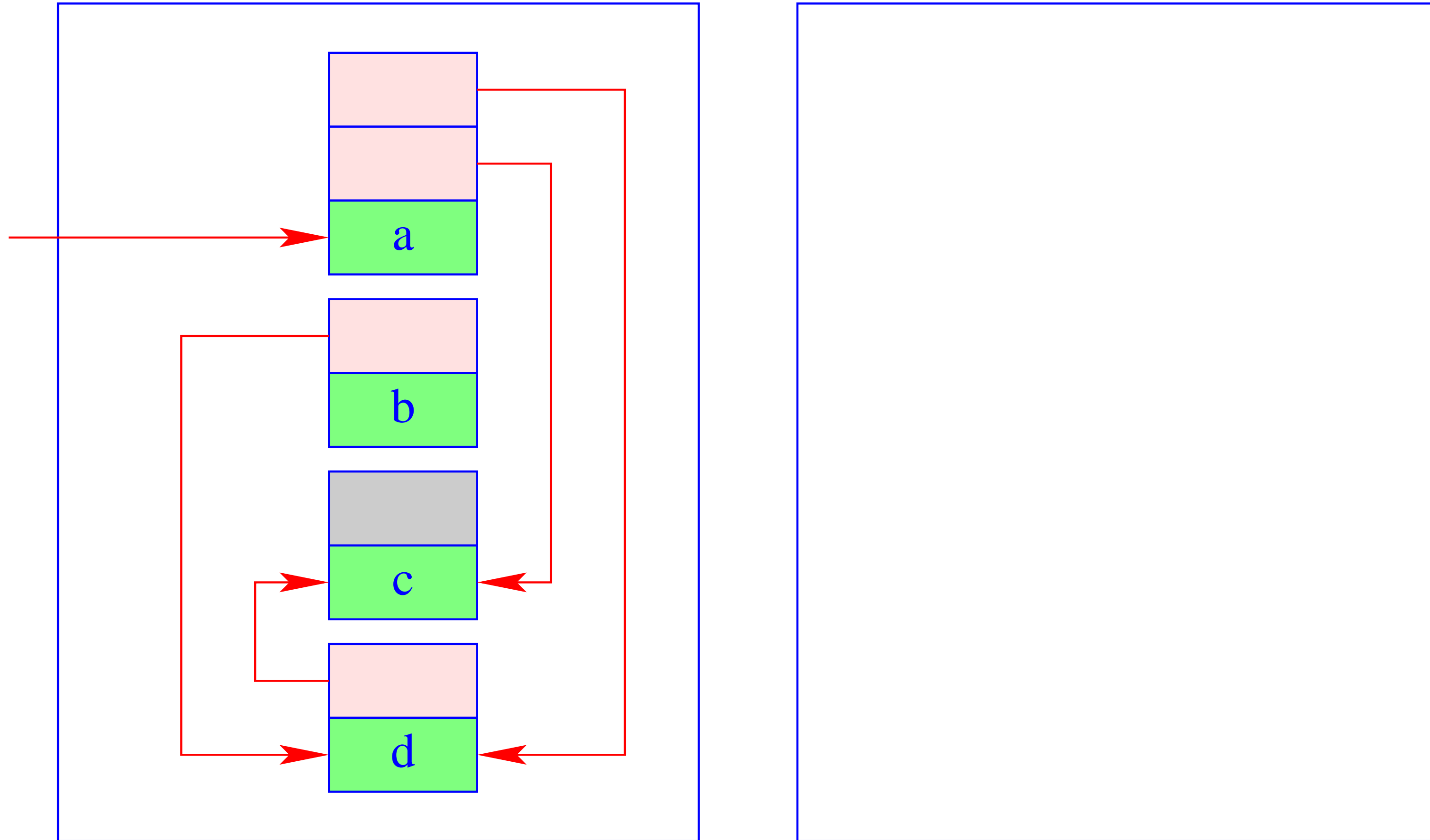

a
b
c
d

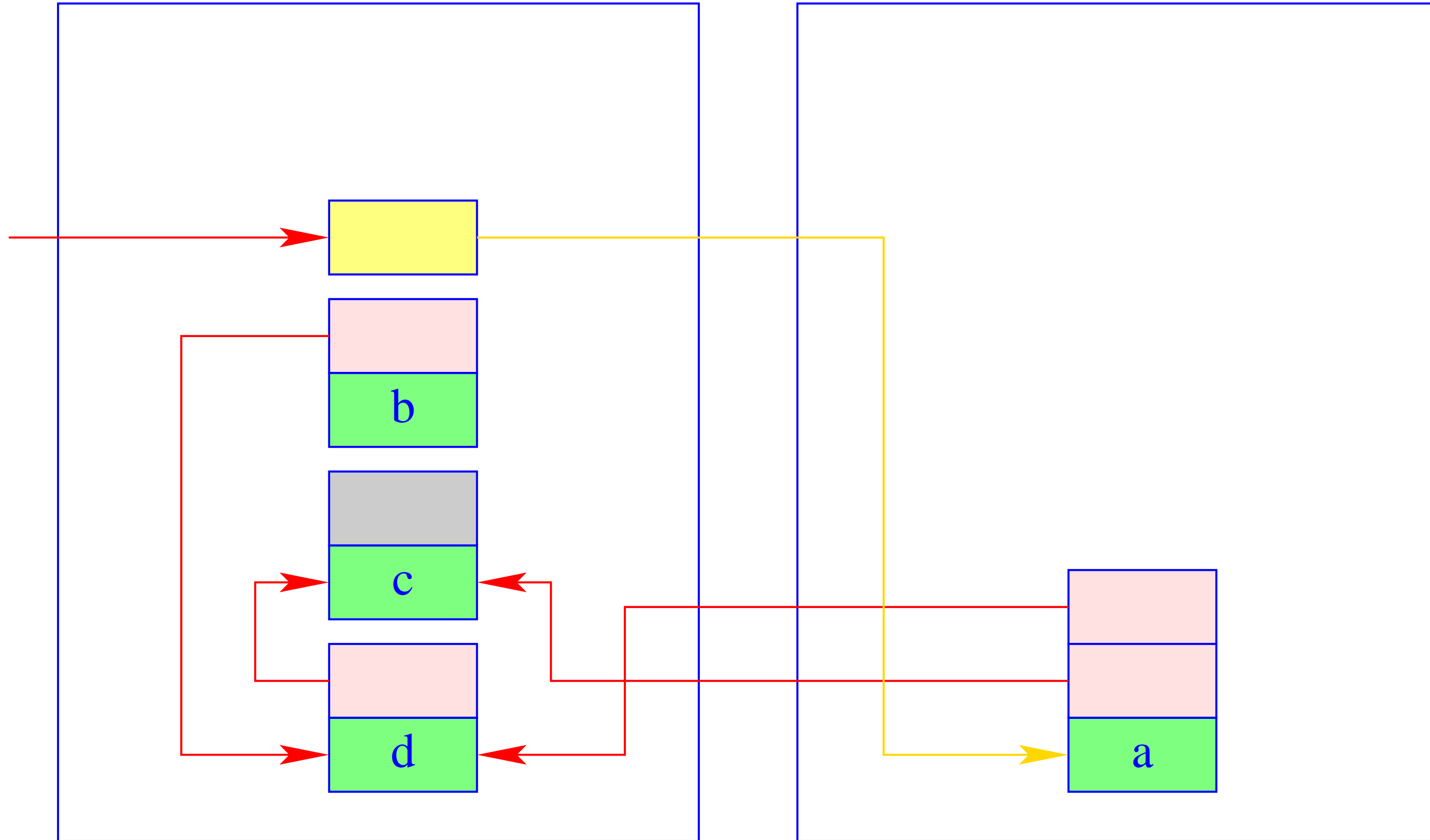

a
b
c
d

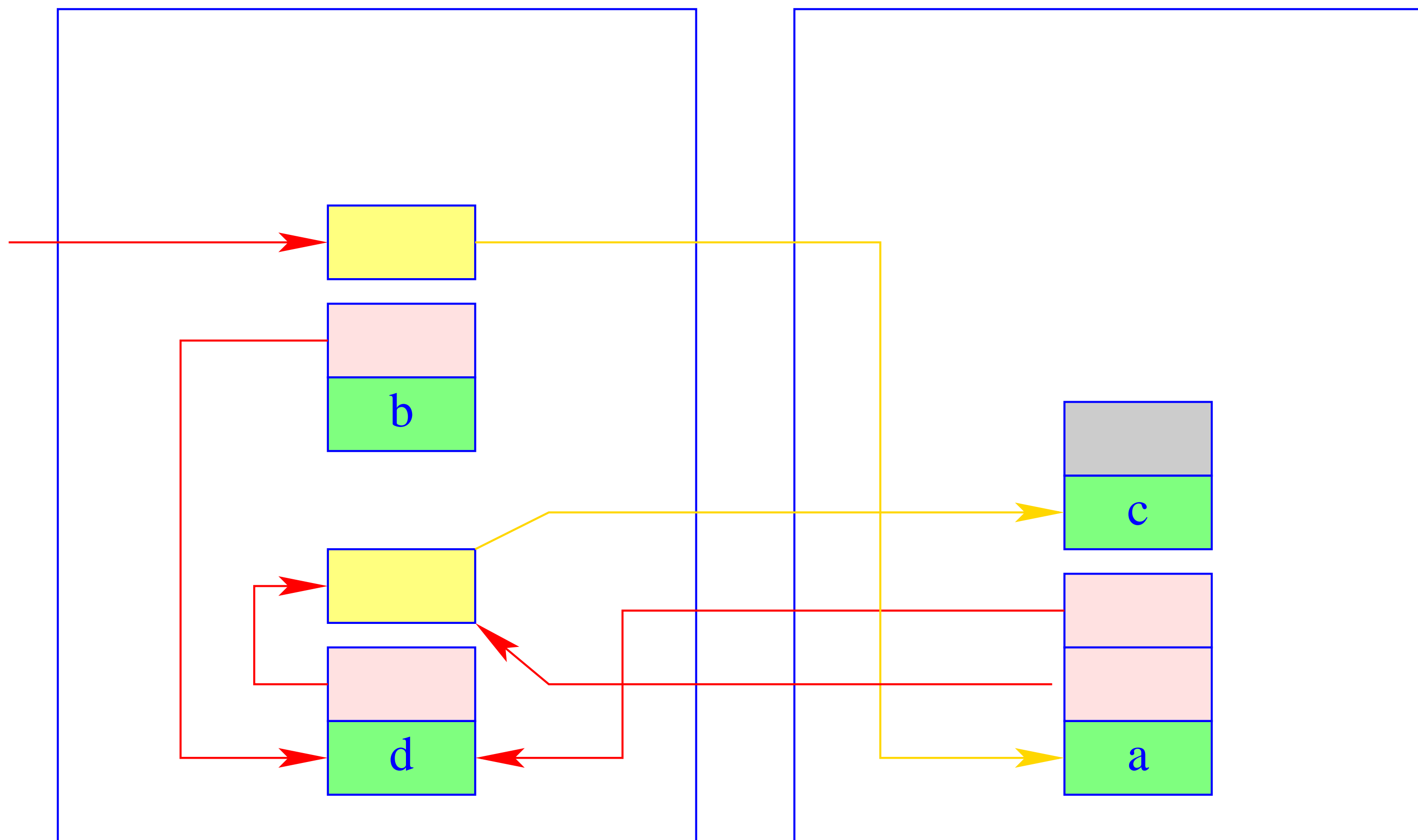
(2) Kopieren der lebendigen Objekte aus dem alten Speicherbereich From-Space in einen neuen BereichTo-Space. Das heißt für jedes aufgefundene Objekt:

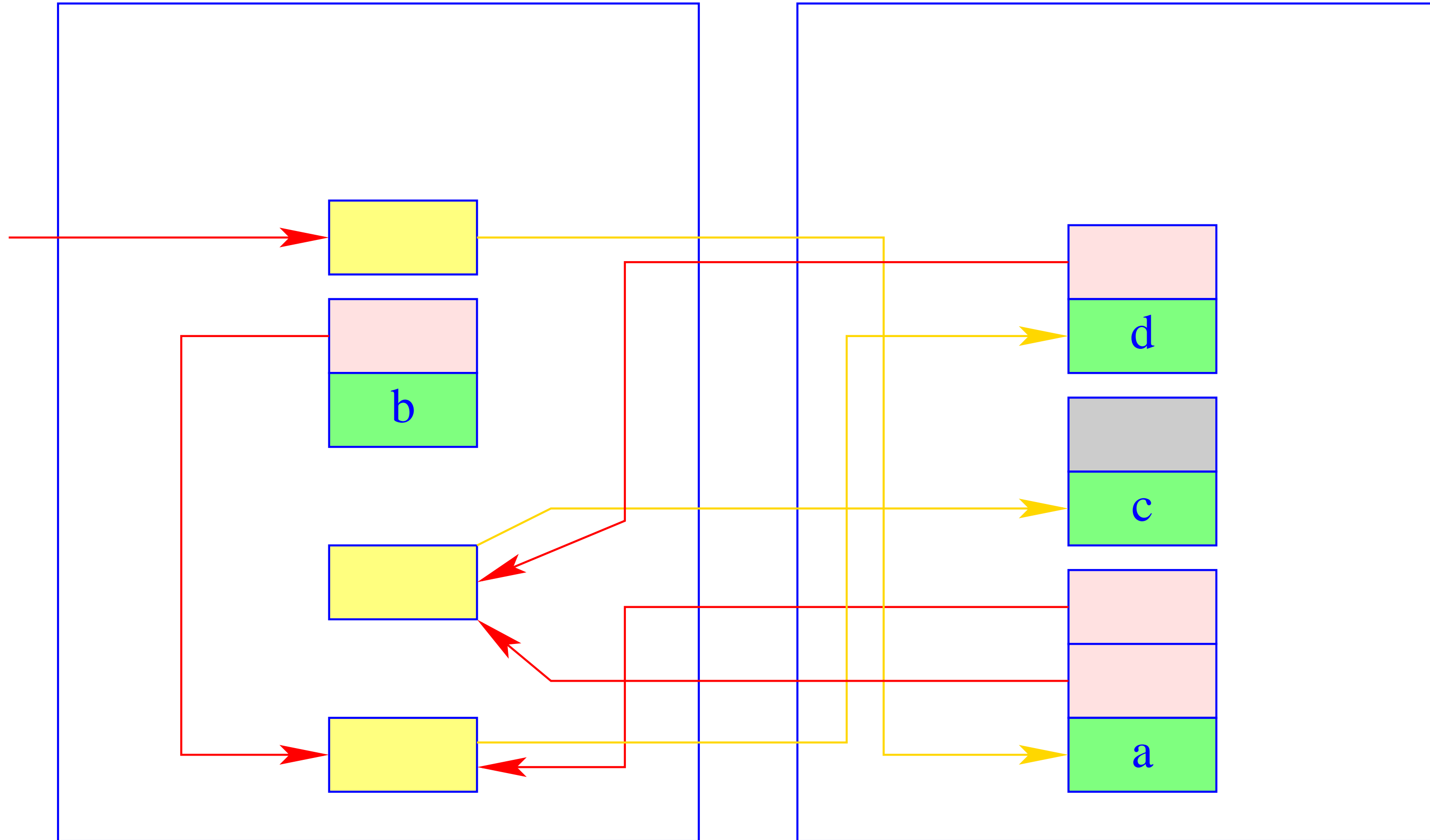
- Kopieren des Objekts;
- Vermerk des neuen Platzes an der alten Stelle (Forwärts-Referenz).

alle Verweise der kopierten Objekte zeigen auf die Forwärts-Referenzen im From-Space.

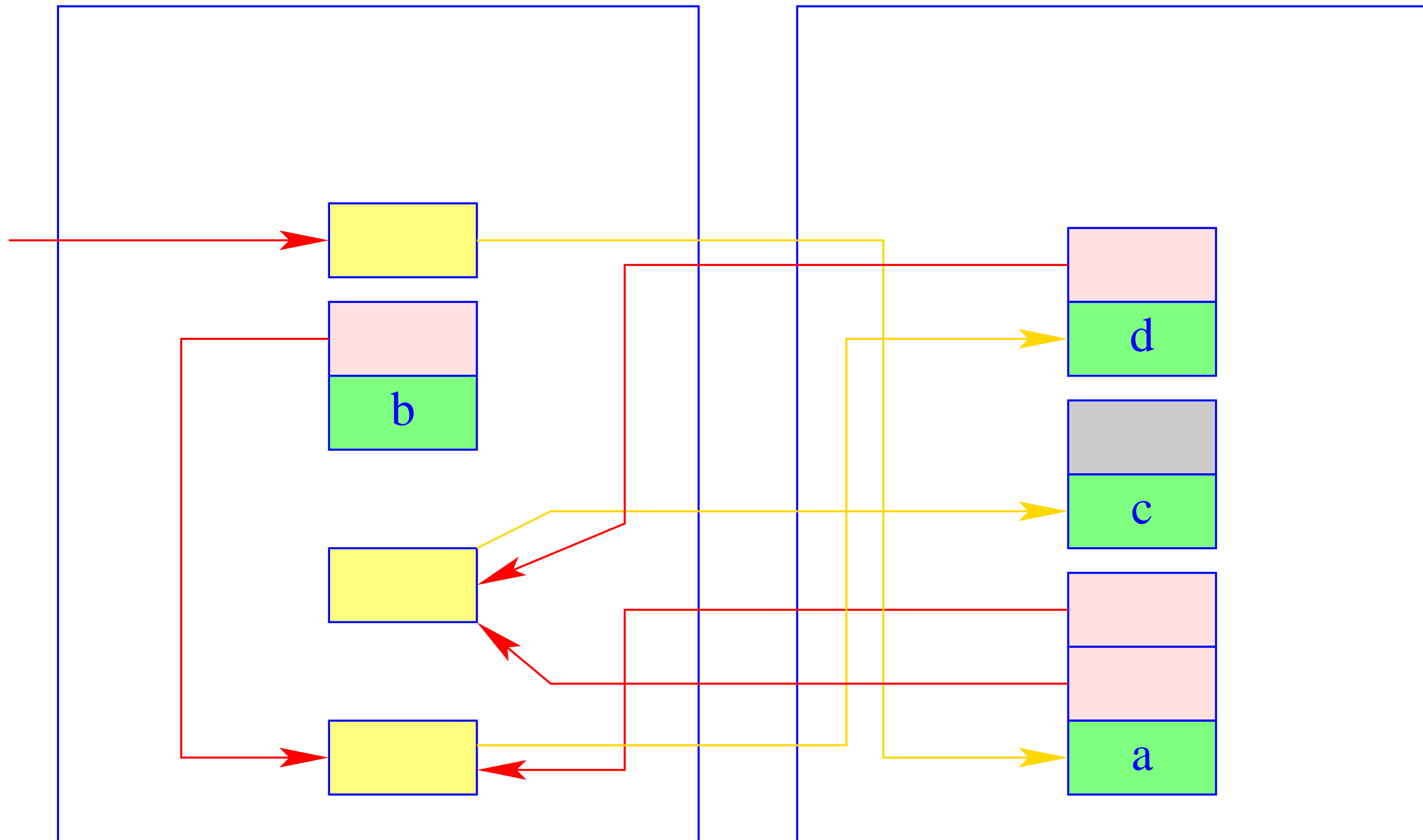

a
b
c
d

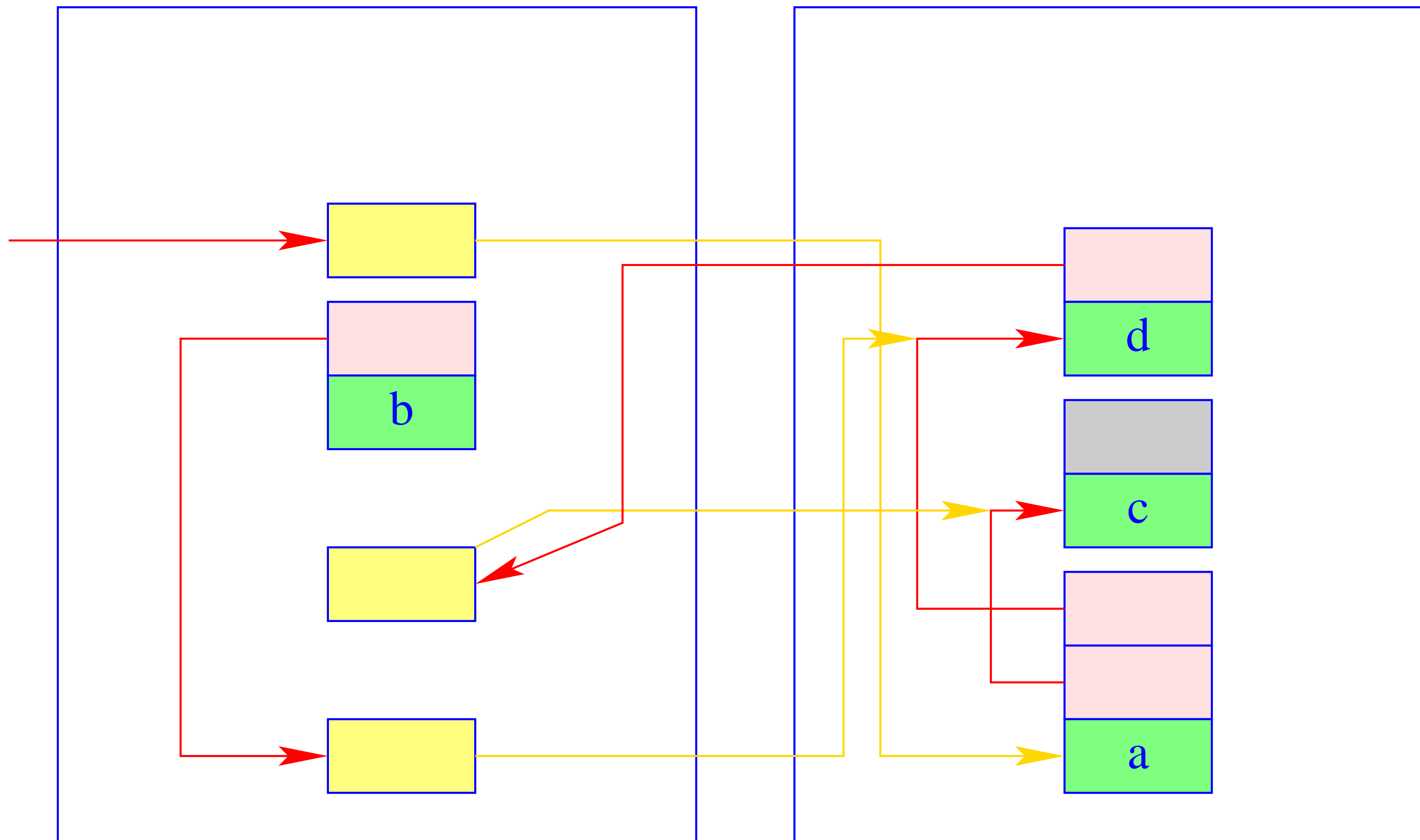

b
c
d
a


b
c
d
a

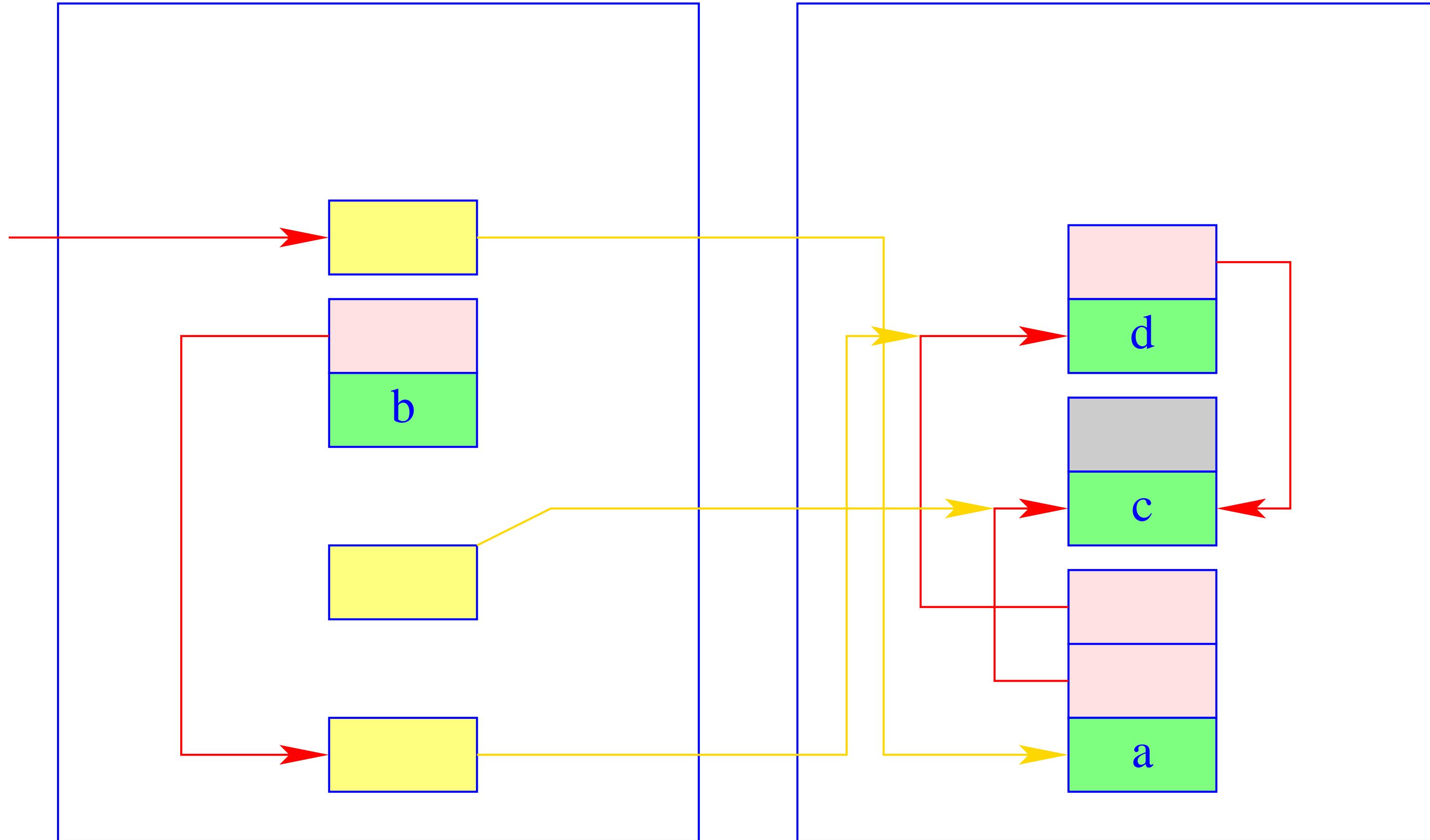

d
b
c
a

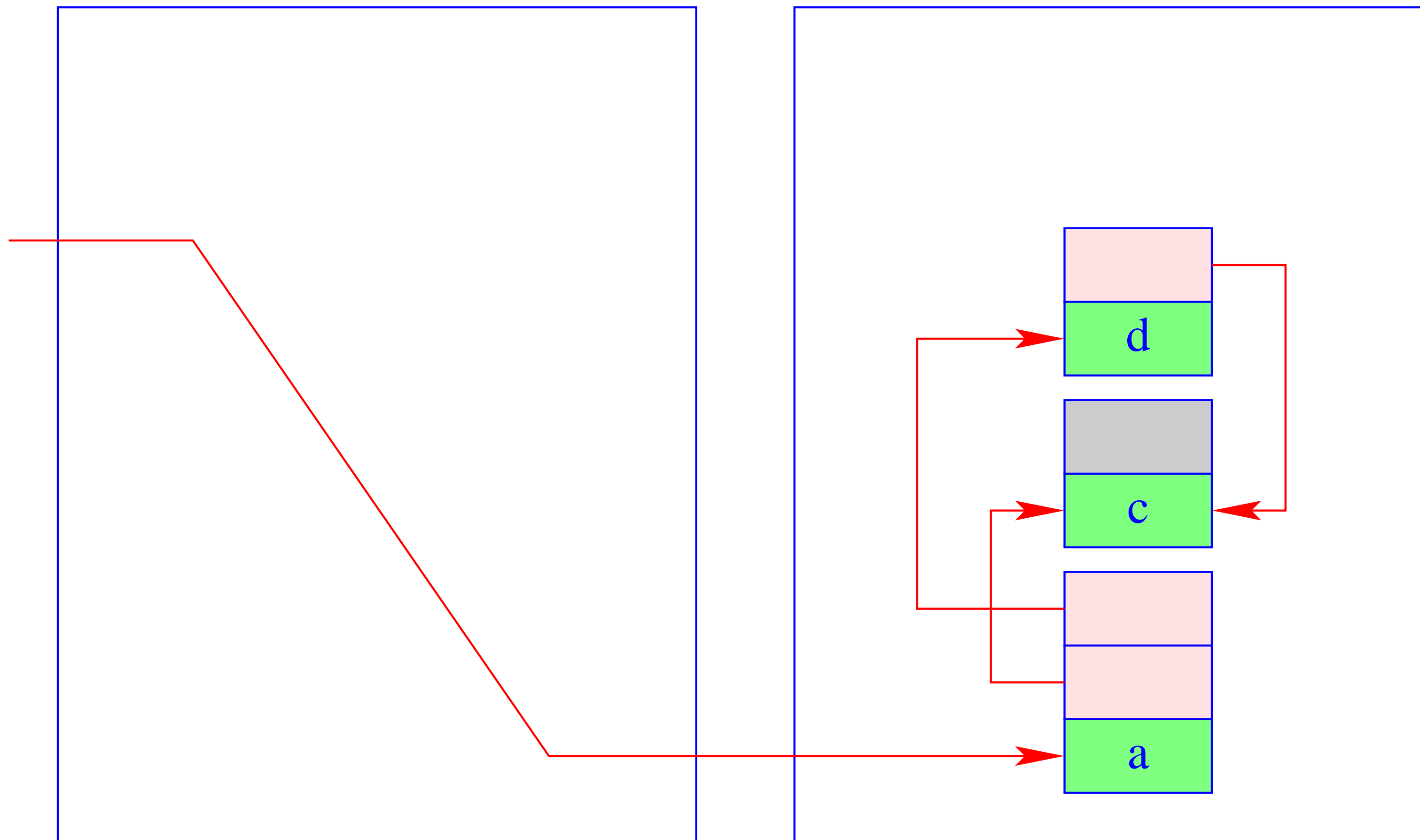
(3) Traversieren des To-Space zur Korrektur der Referenzen.



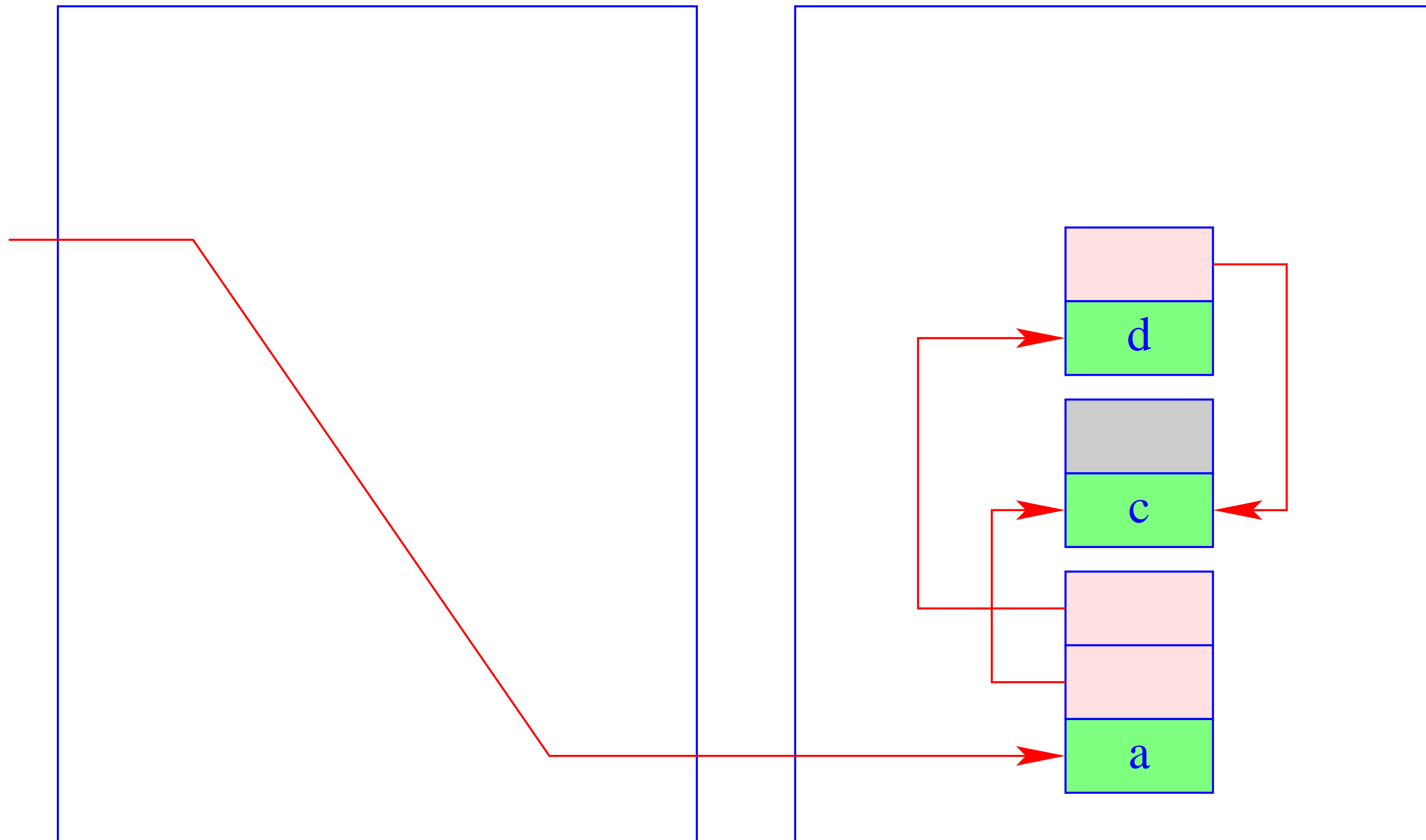


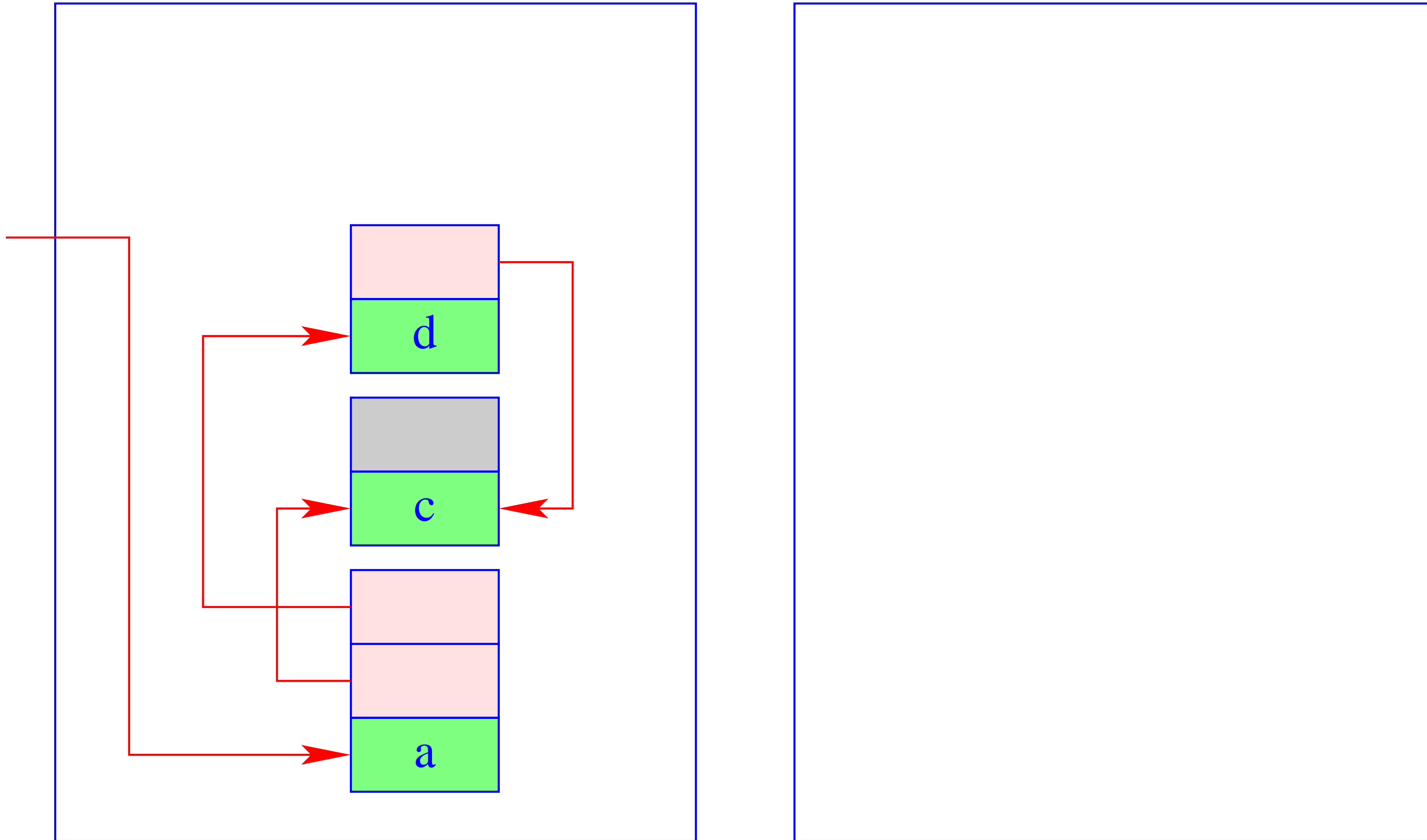
b
d
c
a


b
d
c
a


d
c
a

(4) Vertauschen von To- und From-Space.



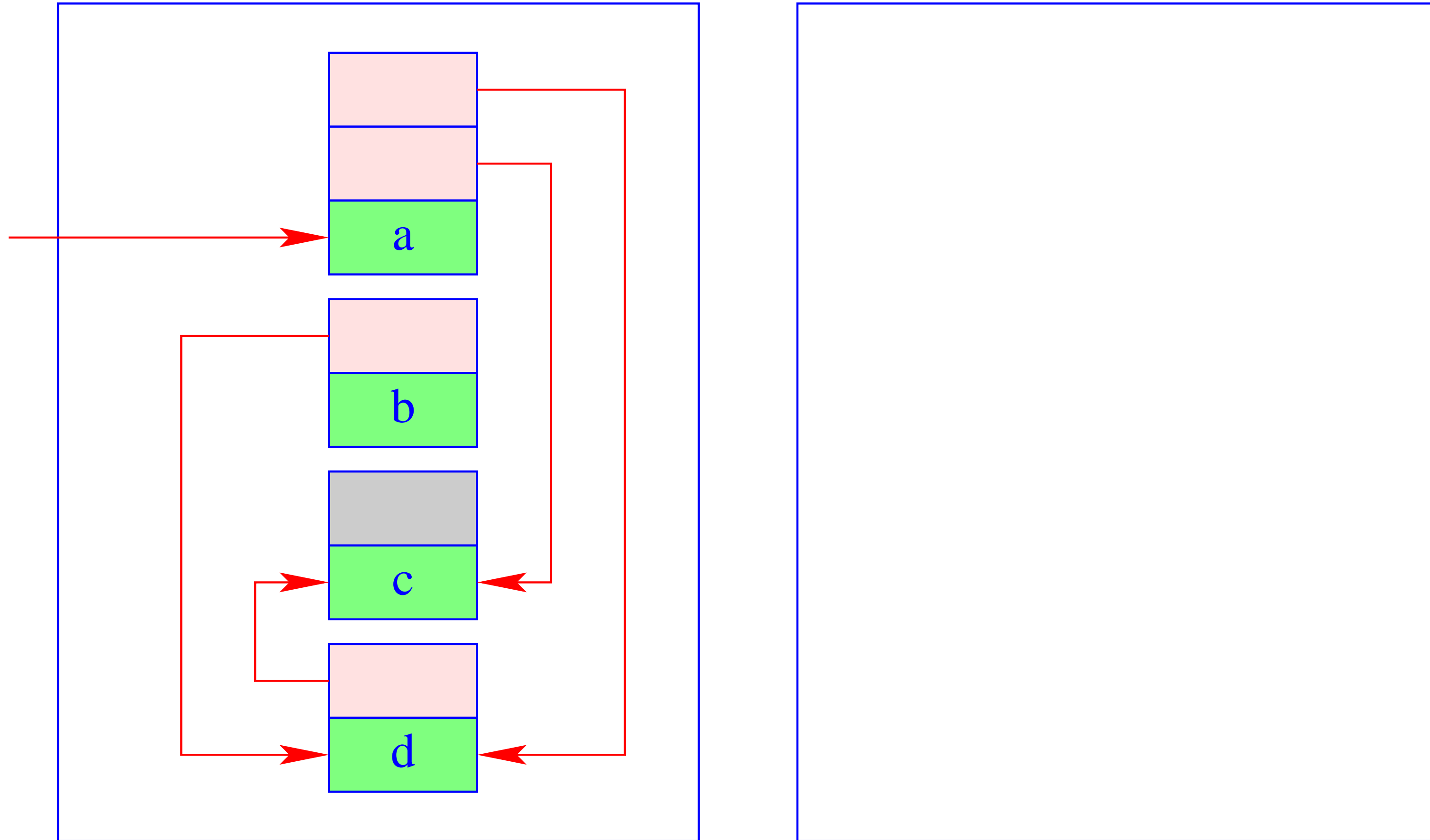

d
c
a

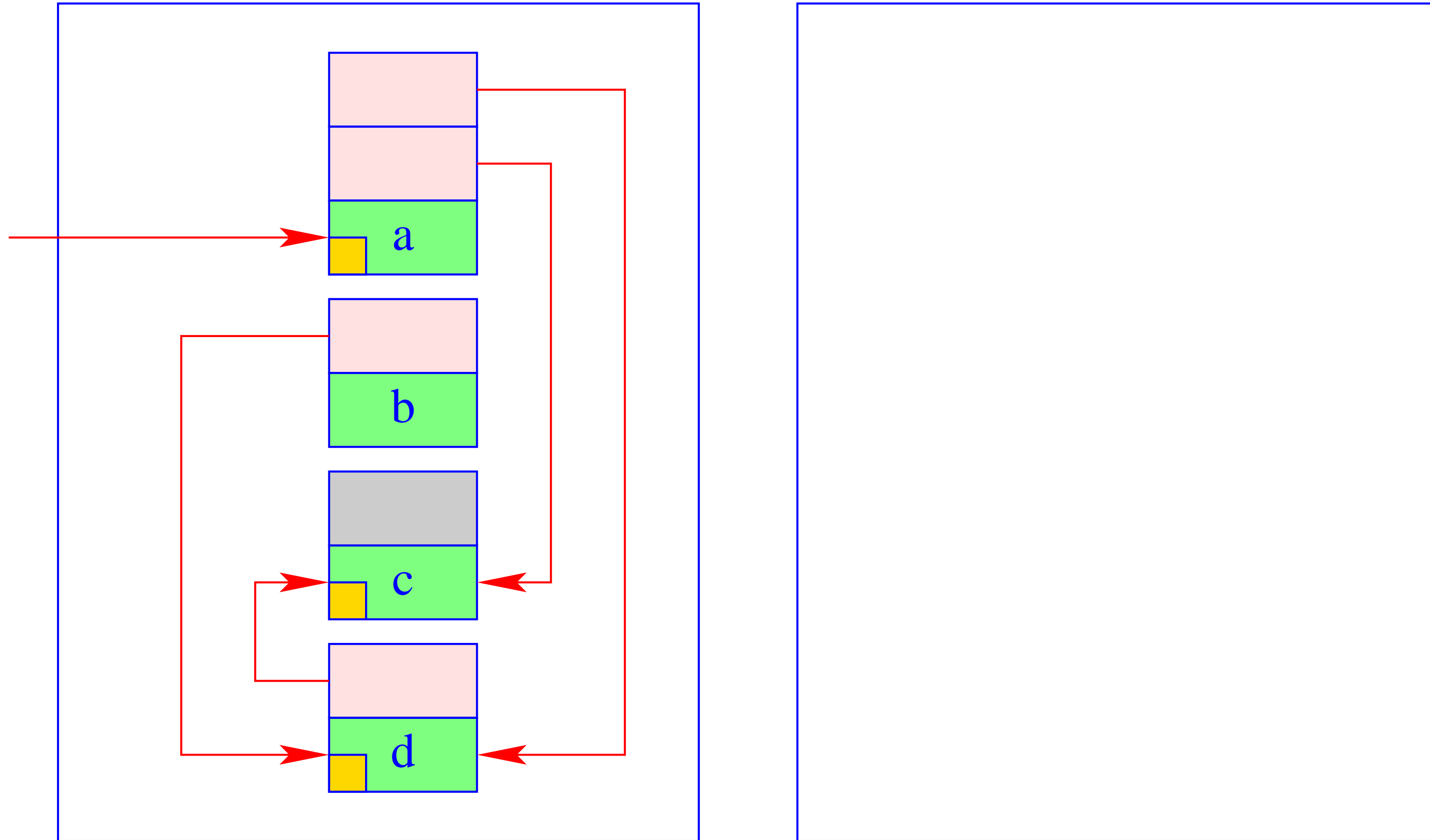
Achtung:

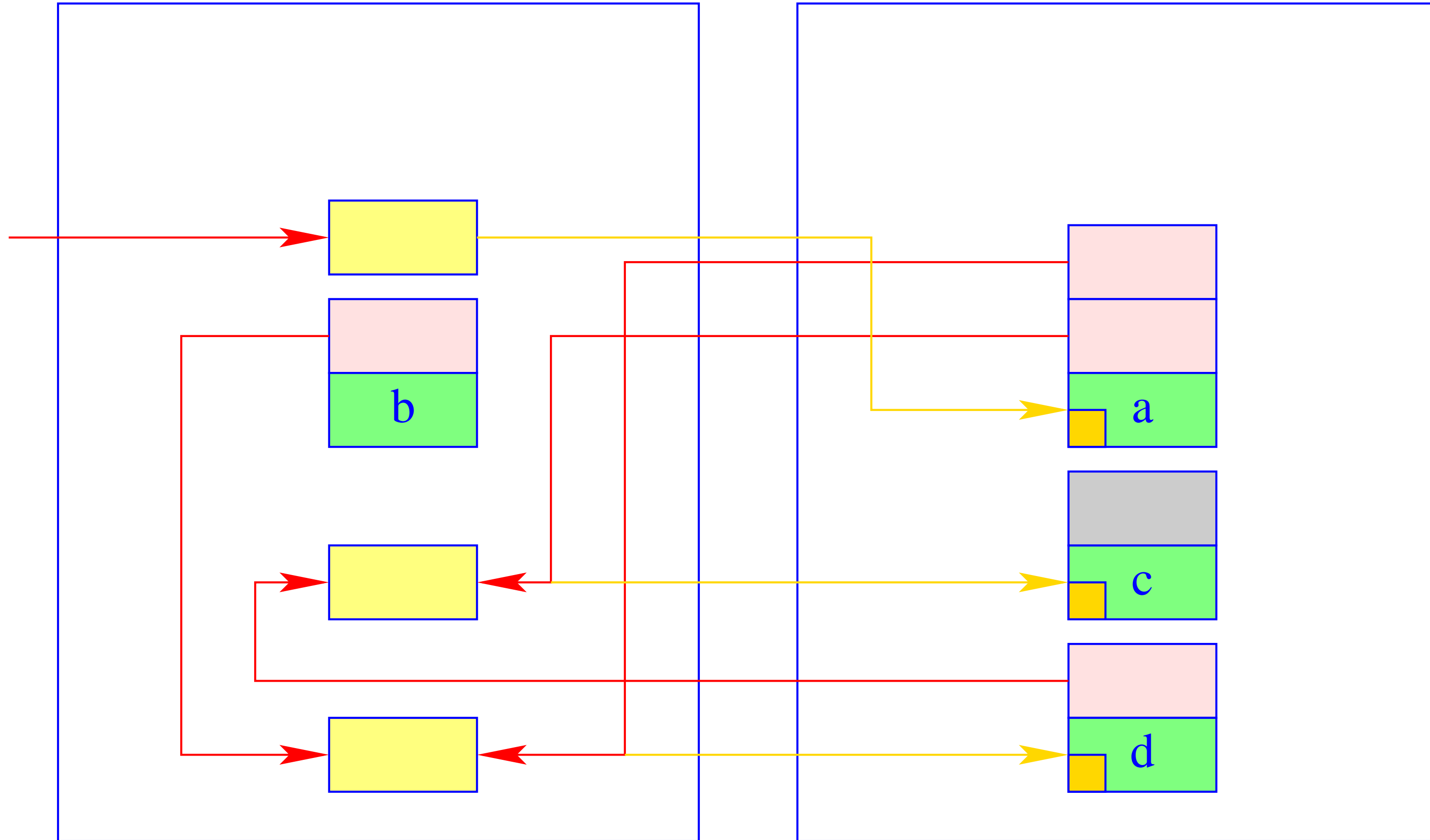
Die Garbage Collection in der WiM muss mit dem Backtracking harmonisieren.

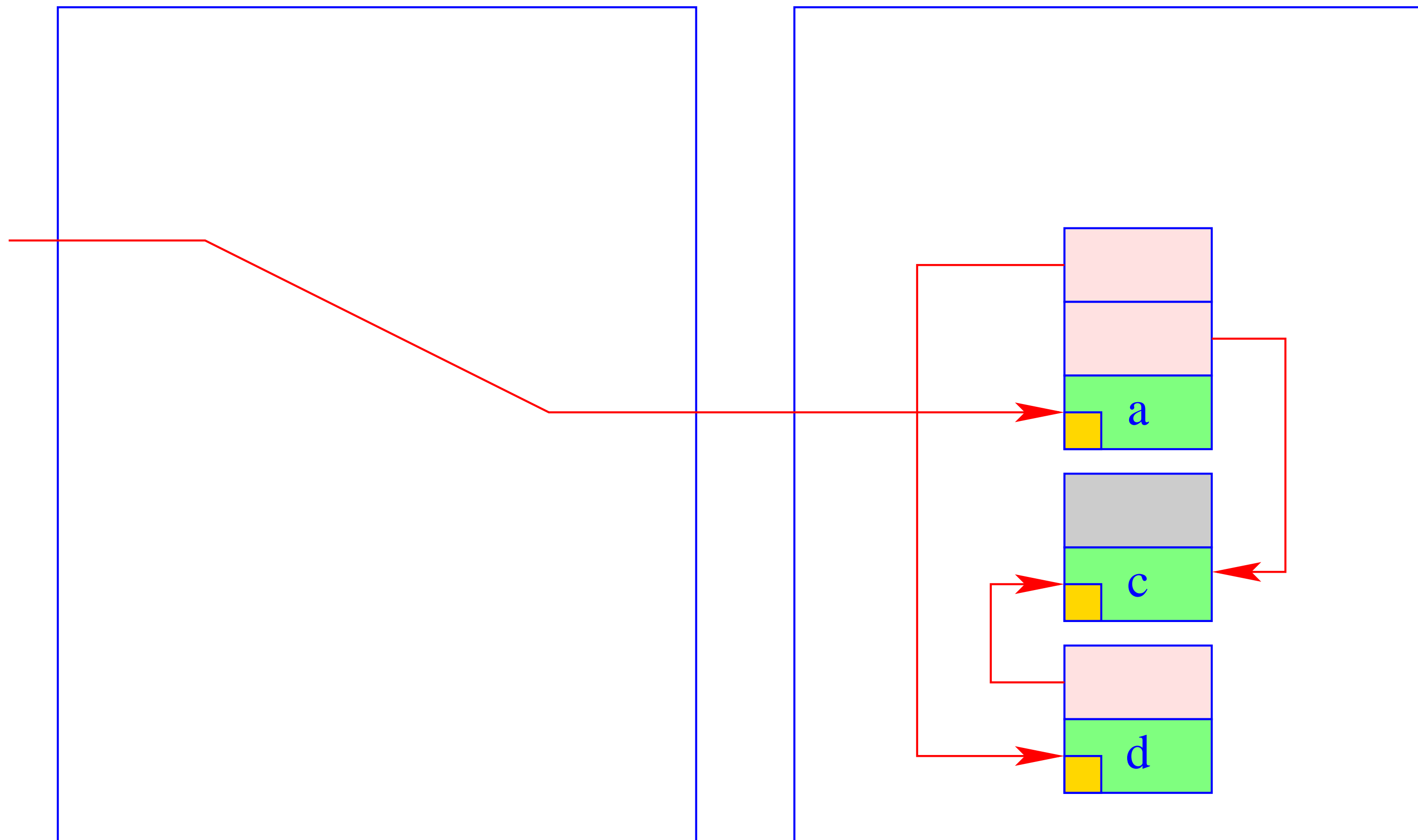
Das heißt:

- Die relative Lage der Halden-Objekte darf sich beim Kopieren nicht verändern :-!!
- Die Halden-Verweise im Trail müssen auf die neuen Positionen der Objekte umgesetzt werden.
- Werden auch Heap-Objekte eingesammelt, die vor dem letzten Rücksetz-Punkt angelegt wurden, müssen auch die Heap-Pointer im Keller umgesetzt werden.


a
b
c
d


a
b
c
d


b
a
c
d


a
c
d

Threads

39 Die Sprache ThreadedC

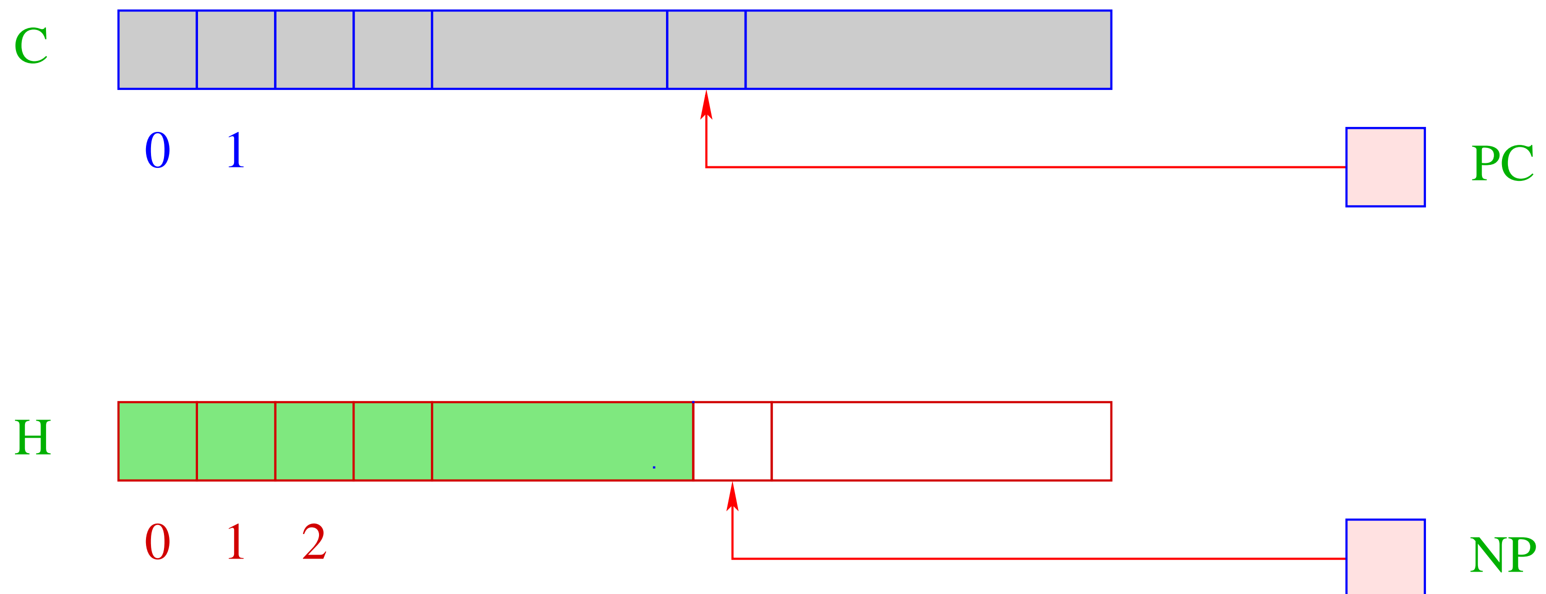
Wir erweitern C um ein einfaches Thread-Konzept. Insbesondere stellen wir Funktionen bereit, um:

- neue Threads zu erzeugen: create();
- einen Thread zu beenden: exit();
- auf die Terminierung eines Threads zu warten: join();
- wechselseitigen Ausschluss zu ermöglichen: lock(), unlock(); ...

Um eine parallele Programm-Ausführung zu ermöglichen, benötigen wir natürlich :-) eine Erweiterung der abstrakten Maschine ...

40 Speicher-Organisation

Allen Threads gemeinsam ist Code-Speicher und Halde:

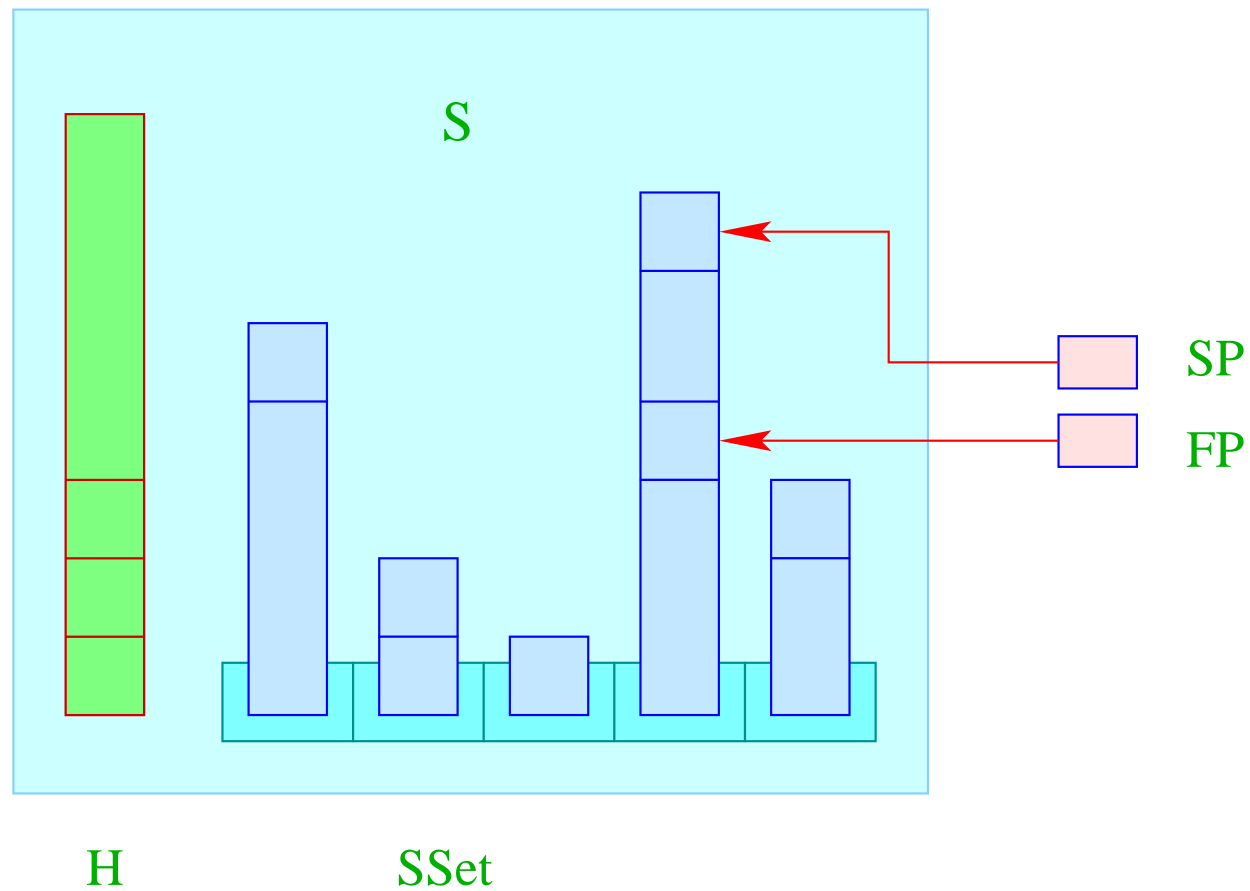


... wie bei der CMa haben wir:

C	=	Code-Speicher – enthält CMa-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter – zeigt auf nächsten auszuführenden Befehl;
H	=	Halde –
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
		am unteren Ende legen wir die globalen Variablen ab;
NP	=	New-Pointer – zeigt auf erste freie Zelle.

Nur nehmen wir zur Vereinfachung an, dass die Halde in einem eigenen Segment abgelegt ist. Die Funktion malloc() scheitert jetzt dann, wenn NP das obere Ende erreichte.

Jeder Thread benötigt andererseits seinen eigenen Stack:



Im Unterschied zur CMa haben wir:

SSet	=	Set of Stacks – enthält die Stacks der Threads;
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
S	=	gemeinsamer Adress-Raum für Halde und Stacks;
SP	=	Stack-Pointer – zeigt auf oberste belegte Zelle;
FP	=	Frame-Pointer – zeigt auf aktuellen Kellerrahmen.

Achtung:

- Zeigten alle Referenzen stets in die Halde, könnten wir für jeden Stack einen eigenen Adressraum verwenden.

  Dann müssten wir uns außer SP und FP auch merken, in welchem Stack wir uns befinden.

- Für C müssen wir allerdings annehmen, dass sämtliche Speicherbereiche im selben Adressraum liegen – jeweils an unterschiedlichen Stellen :-)

  SP und FP identifizieren damit eindeutige Stellen im Speicher.

- Der Einfachkeit halber verzichten wir auf Extreme-Pointer EP.

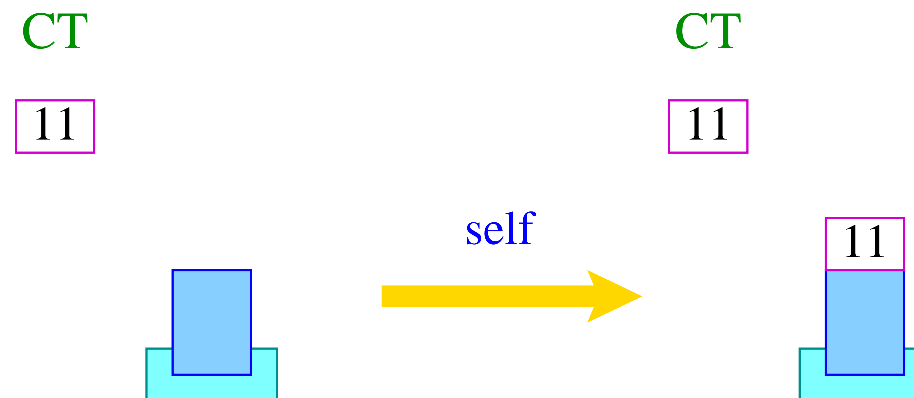
41 Die Ready-Schlange

Idee:

- Jeder Thread hat eine eindeutige Nummer tid.
- Eine Tabelle TTab ermöglicht es, zu einer tid den zugehörigen Thread zu ermitteln.
- Zu jedem Zeitpunkt kann es mehrere ausführbare Threads geben, aber nur einen laufenden (pro Prozessor) geben.
- Die tid des laufenden Threads steht im Register CT (Current Thread).
- Die Funktion: **tid self ()** liefert die tid des laufenden Threads. Folglich:

$$\text{code}_{\text{R}}\ \textbf{self}\ ()\ \rho \quad = \quad \text{self}$$

... wobei die Instruktion self den Inhalt des Registers CT auf den Stack lädt:

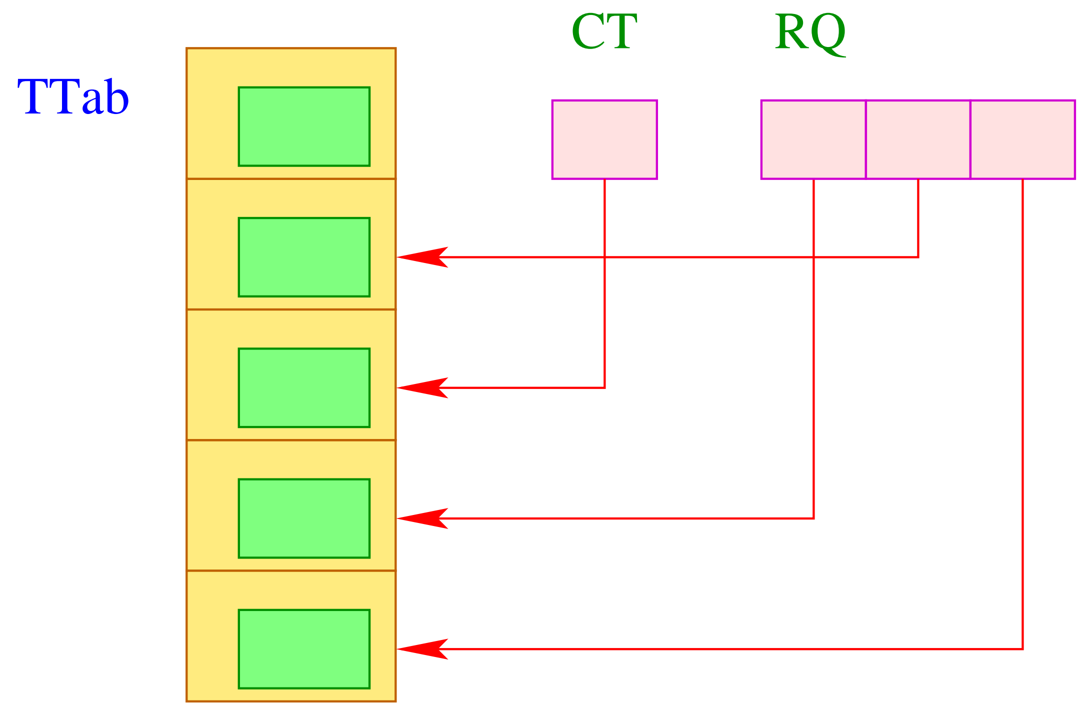


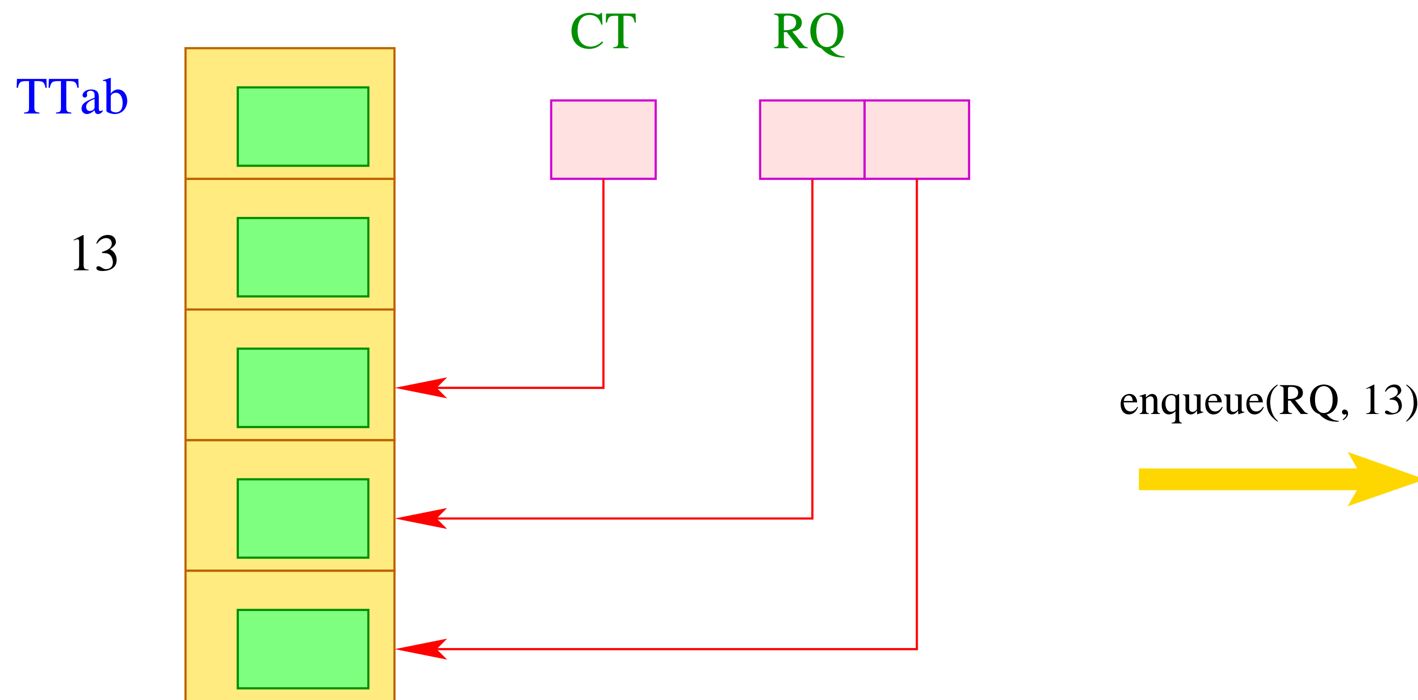
S[SP++] = CT;

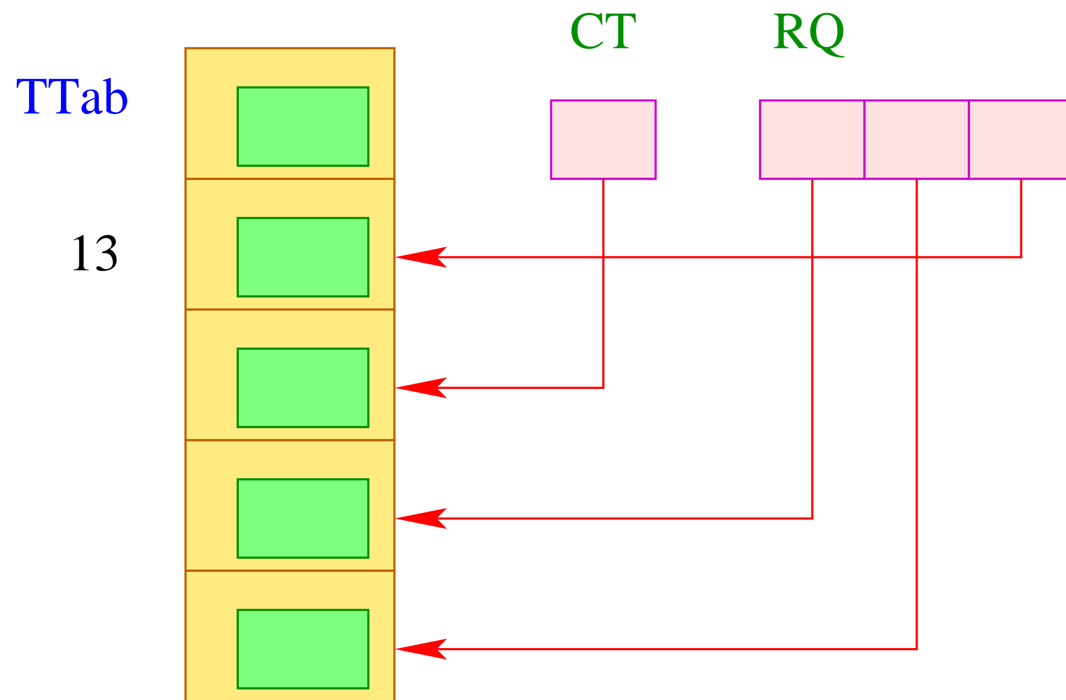
- Die weiteren lauffähigen Threads (bzw. deren tid's) verwalten wir in einer Schlange RQ (Ready-Queue).
- Für Schlangen von Threads benötigen wir die Funktionen:

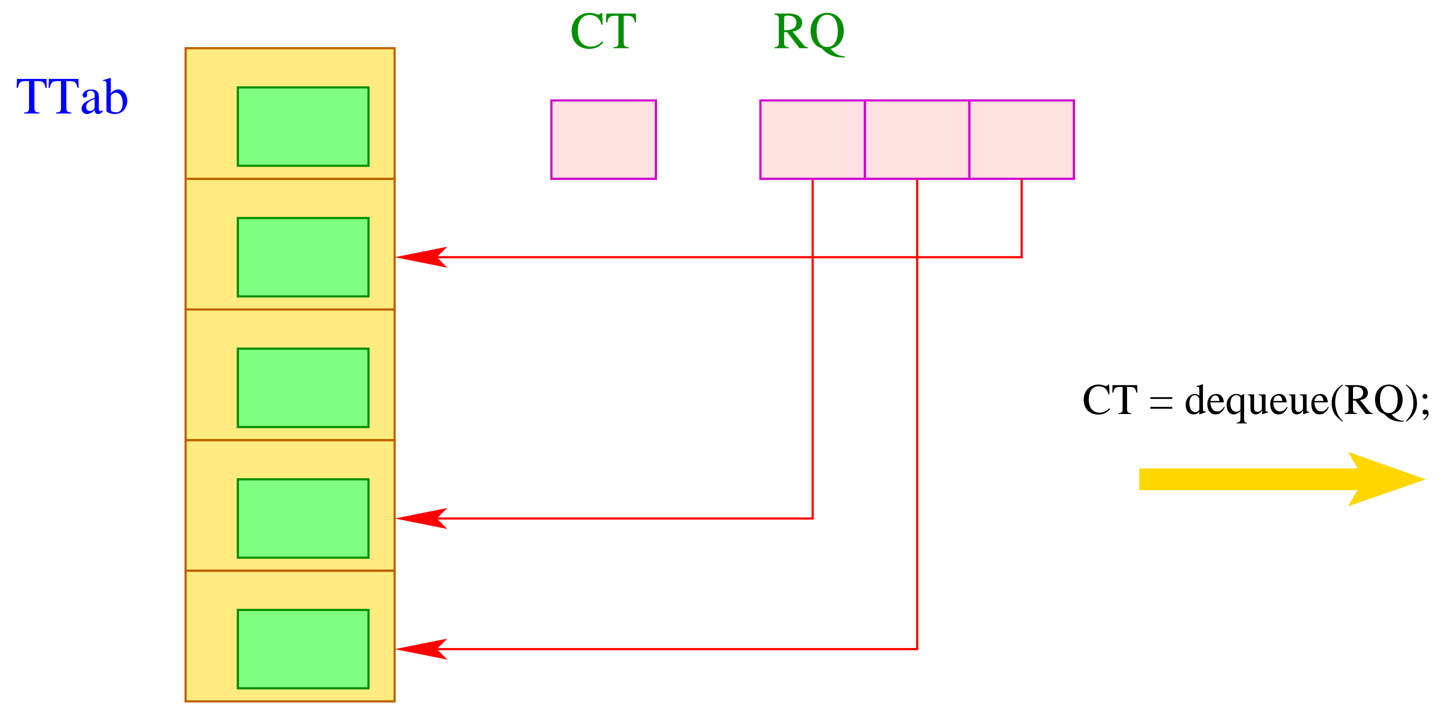
```
void enqueue (queue q, tid t),
tid dequeue (queue q)
```

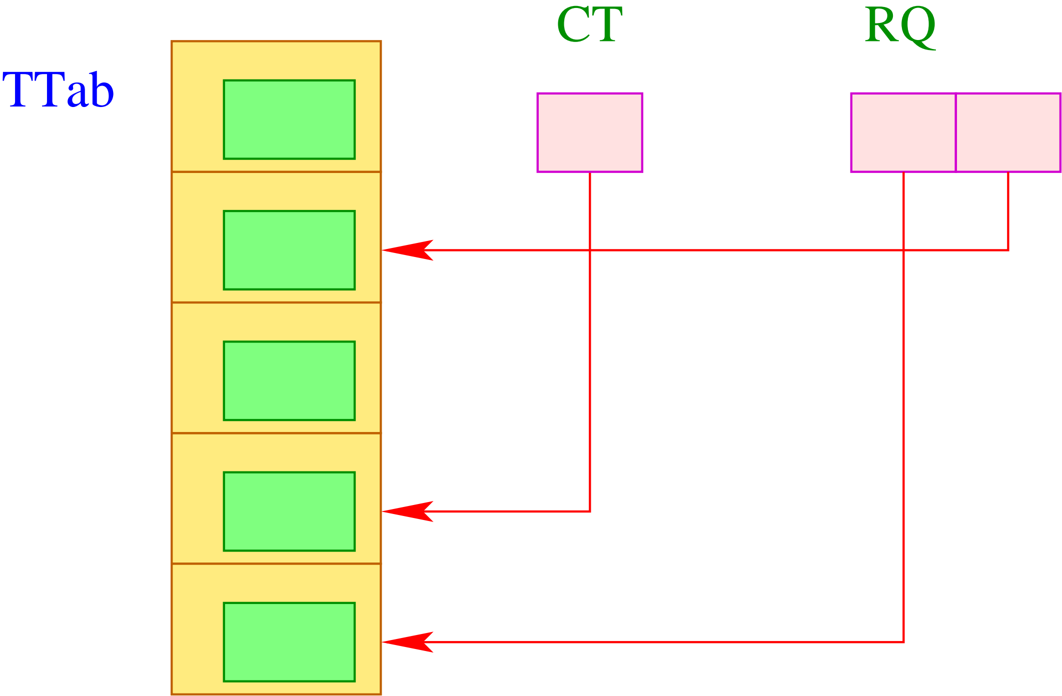
die eine neue tid in die Schlange einfügen bzw. die erste zurück liefern ...


TTab
CT
RQ


CT
RQ
TTab
13
enqueue(RQ, 13)

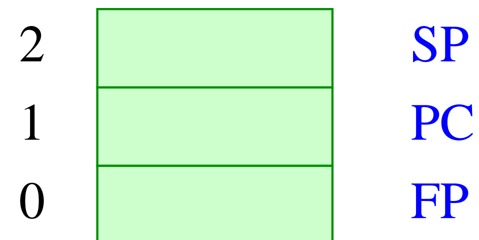

CT
RQ
TTab
13


CT
RQ
TTab
CT = dequeue(RQ);


TTab
CT
RQ

War der Aufruf von dequeue () nicht erfolgreich, liefert die Funktion einen Wert < 0 :-)

Die Thread-Tabelle muss für jeden Thread alle Informationen enthalten, die wir zu seiner Ausführung benötigen. Insbesondere sind das die aktuellen Register PC, SP und FP:



Unterbrechen des gegenwärtigen Threads erfordert darum das Retten dieser Register:

```
void save () {
     TTab[CT][0] = FP;
     TTab[CT][1] = PC;
     TTab[CT][2] = SP;
     }
```

Analog restauriert die Funktion `restore()` diese Register:

```
void restore () {
     FP = TTab[CT][0];
     PC = TTab[CT][1];
     SP = TTab[CT][2];
     }
```

Damit können wir nun eine Instruktion yield realisieren, die einen Thread-Wechsel herbei führt:

```
tid ct = dequeue ( RQ );
if (ct ≥ 0) {
      save (); enqueue ( RQ, CT );
      CT = ct;
      restore ();
      }
```

Nur wenn die Ready-Schlange nicht-leer ist, wird der aktuelle Thread ersetzt :-)

42 Thread-Wechsel

Problem:

Wir wollen alle lauffähigen Threads nach Möglichkeit fair abarbeiten.

- Jeder Thread muss früher oder später dran kommen.
- Jeder Thread muss früher oder später unterbrochen werden.

Mögliche Strategien:

- Thread-Wechsel nur bei explizitem Aufruf einer Funktion yield() :-(
- Thread-Wechsel nach jeder Instruktion $\Longrightarrow$ zu teuer :-(
- Thread-Wechsel nach einer festen Anzahl von Schritten $\Longrightarrow$ wir müssten einen Zähler mitführen und yield dynamisch einfügen :-(

Thread-Wechsel an ausgewählten Programm-Punkten ...

- am Anfang von Funktions-Rümpfen;
- vor jedem Sprung, dessen Ziel nicht größer ist als der aktuelle PC...

$\Longrightarrow$ selten :-))

Das modifizierte Schema für Schleifen $s \equiv \textbf{while}\ (e)\ s$ liefert dann etwa:

$$
\begin{array}{llll}
\text{code}\ s\ \rho & = & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & & \text{jumpz B} \\
 & & & \text{code}\ s\ \rho \\
 & & & \text{yield} \\
 & & & \text{jump A} \\
 & & \text{B}: & \ldots
\end{array}
$$

Beachte:

- **If-then-else**-Statements enthalten nicht notwendigerweise Thread-Wechsel.
- **do-while**-Schleifen erfordern einen Thread-Wechsel am Ende der Bedingung.
- Jede Schleife sollte (mindestens) einen Thread-Wechsel enthalten :-)
- Loop-Unroling verringert die Anzahl dieser Wechsel.
- Bei der Übersetzung von **switch**-Statements legten wir die Sprungtabelle hinter die Alternativen. Hier können wir trotzdem auf Thread-Wechsel verzichten.
- Bei frei programmierter Benutzung von jumpi wie auch jumpz sollte sicherheitshalber auch ein Thread-Wechsel vor dem Sprung (oder am Sprung-Ziel) eingefügt werden.
- Will man die Anzahl der Thead-Wechsel weiter reduzieren, kann man z.B. nur bei jedem 100. Aufruf von yield den Kontext wechseln ...

43 Die Erzeugung neuer Threads

Wir nehmen an, der Ausdruck: $s \equiv \mathbf{create}\ (e_0, e_1)$ wertet erst die Ausdrücke e_i zu Werten f, a aus und erzeugt einen neuen Thread, der $f\ (a)$ abarbeitet.

Scheitert die Thread-Erzeugung, liefert s den Wert -1 zurück, andernfalls liefert s die tid des neuen Prozesses.

Aufgaben des erzeugten Codes:

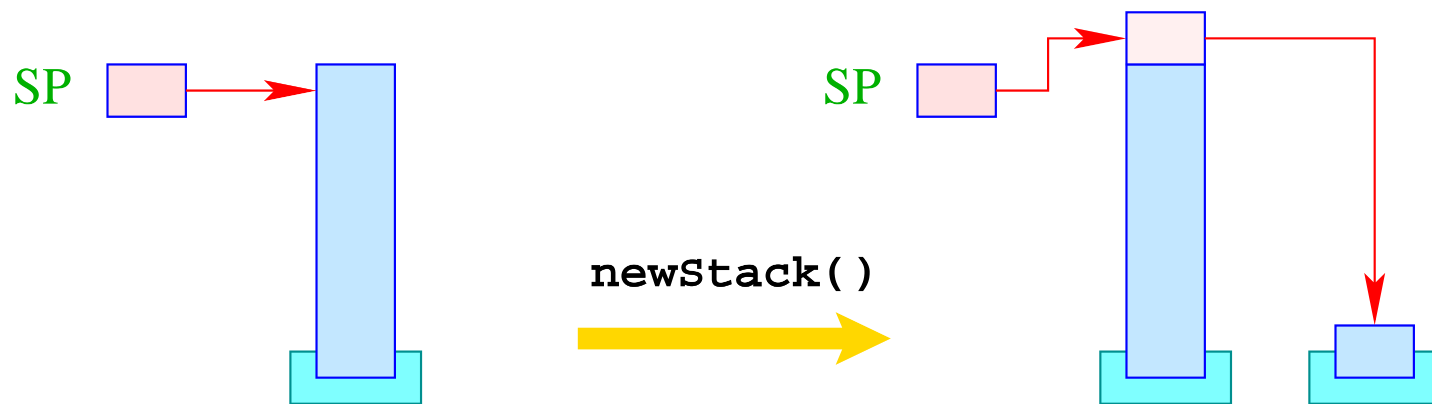
- Auswerten der e_i;
- Anlegen eines neuen Laufzeit-Stacks mit Keller-Rahmen zum Auswerten von $f\ (a)$;
- Erzeugen einer neuen tid;
- Anlegen eines neuen Eintrags in die TTab;
- Einfügen der neuen tid in die Ready-Schlange.

Die Übersetzung von s ist dann ganz einfach:

$$\begin{aligned} \text{code}_\text{R}\; s\; \rho \quad = \quad & \text{code}_\text{R}\; e_0\; \rho \\ & \text{code}_\text{R}\; e_1\; \rho \\ & \text{initStack} \\ & \text{initThread} \end{aligned}$$

wobei wir Platzbedarf 1 für den Wert des Arguments annehmen :-)

Zur Implementierung von initStack benötigen wir eine Laufzeit-Funktion newStack(), welche einen Pointer auf ein erstes Element eines neuen Stacks liefert:



Falls das Anlegen eines neuen Stacks scheitert, soll der Wert 0 zurück geliedert werden.

```
newStack();
if (S[SP]) {
        S[S[SP]+1] = -1;
        S[S[SP]+2] = f;
        S[S[SP]+3] = S[SP-1];
        S[SP-1] = S[SP]; SP--
    }
else S[SP = SP - 2] = -1;
```

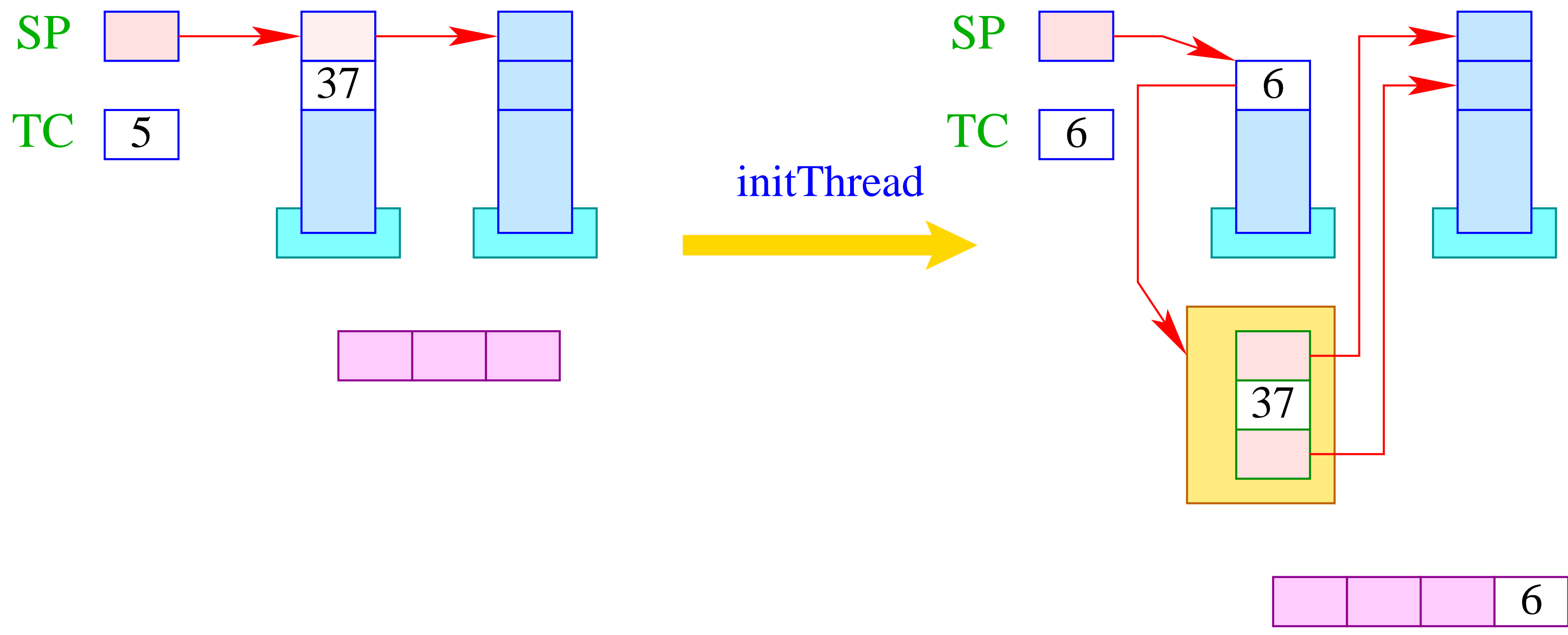
Beachte:

- Die Fortsetzungs-Adresse f zeigt auf den (festen) Code zur Beendigung eines Thread.
- Im Kellerrahmen haben wir keinen Platz mehr für den EP allokiert $\Longrightarrow$ Der Rückgabe-Wert hat darum jetzt Relativ-Adresse -2.
- Den untersten Kellerrahmen erkennen wir daran, dass dort $FPold = -1$ ist.

Um neue Thread-Ids erzeugen zu können, spendieren wir uns ein neues Register TC (Thread Count).

Anfangs hat TC den Wert 0 (entspricht der tid des Start-Threads).

Vor Erzeugen eines neuen Threads, wird TC um eins erhöht.


SP
TC
5
37
initThread
SP
TC
6
6
37
6

```
if (S[SP] ≥ 0) {
    tid = ++TCount;
    TTab[tid][0] = S[SP]-1;
    TTab[tid][1] = S[SP-1];
    TTab[tid][2] = S[SP];
    S[--SP] = tid;
    enqueue( RQ, tid );
}
```

44 Die Beendigung von Threads

Die Beendigung eines Threads liefert (normalerweise :-) einen Wert zurück. Es gibt zwei (reguläre) Verfahren, um einen Thread zu beenden:

1. Der anfängliche Funktions-Aufruf terminiert. Der Rückgabe-Wert des Threads ist gleich des Aufrufs.
2. Der Thread führt das Statement **exit** (e); aus. Der Rückgabe-Wert des Threads ist gleich dem Wert von e.

Achtung:

- Den Rückgabe-Wert wollen wir in der untersten Stack-Zelle übergeben.
- **exit** kann tief geschachtelt in einer Rekursion vorkommen. Dann geben wir sämtliche Kellerrahmen des Threads frei.
- Anschließend springen wir die End-Behandlung von Threads an der Adresse f am Ende des Programms an.

Damit übersetzen wir:

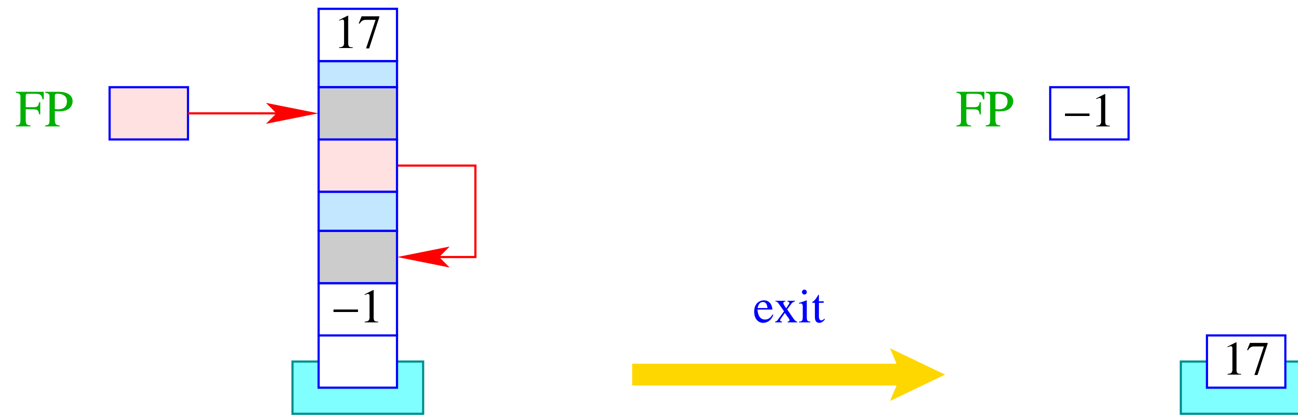
$$\text{code } \textbf{exit } (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

```
exit
term
next
```

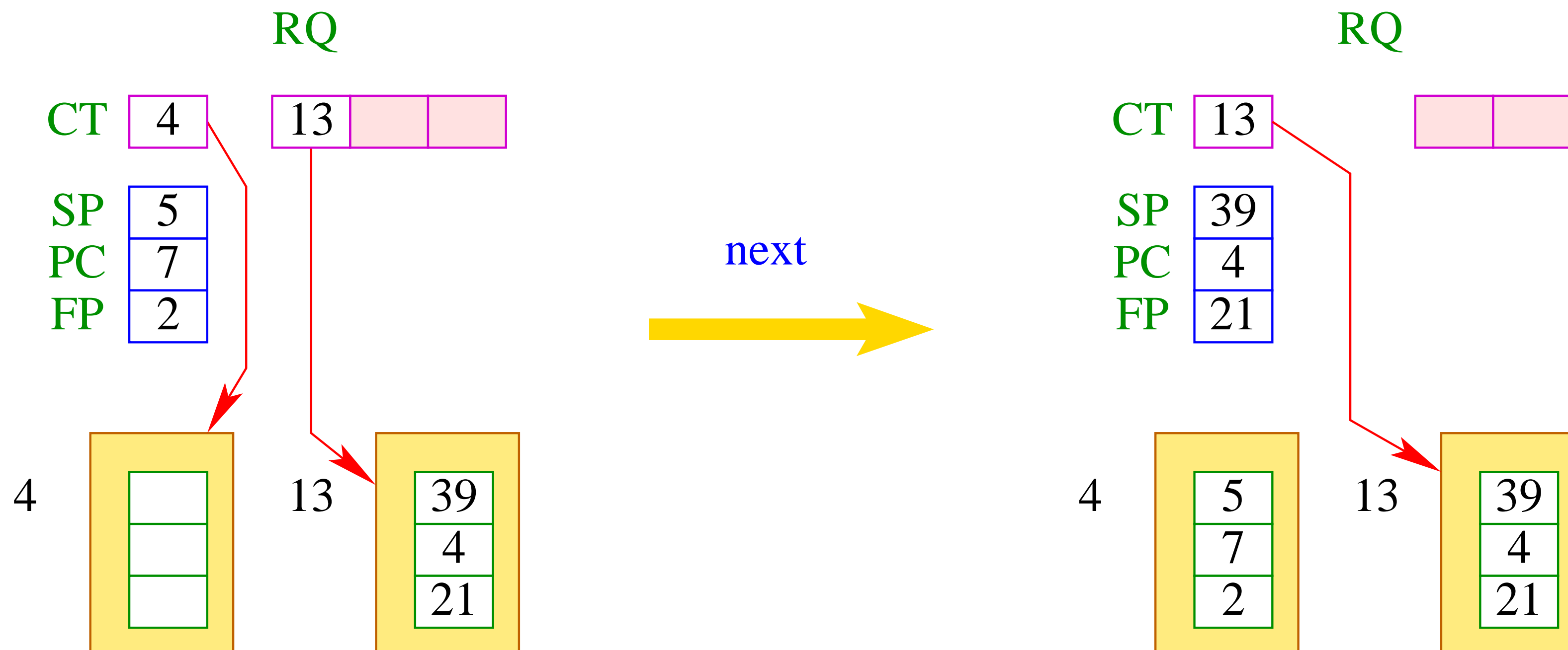
Die Instruktion term behandeln wir später :-)

Die Instruktion exit muss sukzessive sämtliche Keller-Rahmen des Threads aufgeben:

```
result = S[SP];
while (FP ≠ –1) {
        SP = FP–2;
        FP = S[FP–1];
        }
S[SP] = result;
```


17
FP
−1
exit
FP
−1
17

Die Instruktion next aktiviert den nächsten lauffähigen Thread: im Gegensatz zu yield wird jedoch der aktuelle Thread nicht wieder in RQ eingefügt.



Ist die Schlange RQ leer, wird zusätzlich das Programm beendet:

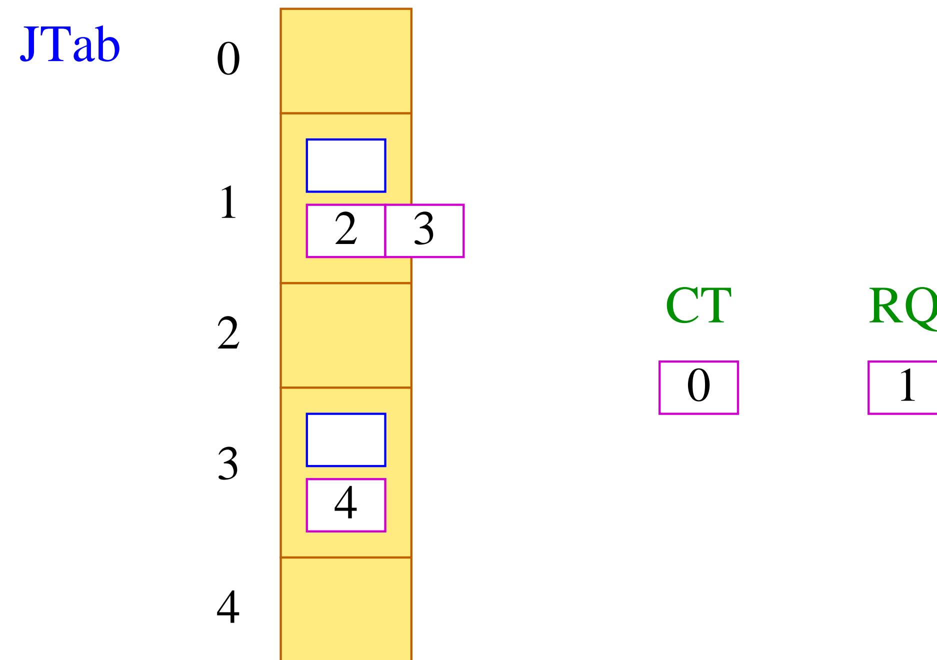
```
if (0 > ct = dequeue( RQ ))   halt;
else {
        save ();
        CT = ct;
        restore ();
    }
```

45 Warten auf Terminierung

Manchmal darf ein Thread erst mit seiner Ausführung fortfahren, wenn ein anderer Thread terminierte. Dafür gibt es den Ausdruck **join** (e). Dabei erwarten wir, dass sich e zu einer Thread-Id tid auswerten lässt.

- Ist der Thread mit dieser Kennung bereits beendet, soll dessen Rückgabe-Wert geliefert werden.
- Ist er noch nicht beendet, müssen wir die aktuelle Programm-Ausführung unterbrechen.
- Wir fügen den aktuellen Thread in die Schlange der anderen bereits auf Terminierung wartenden Threads ein, retten die aktuellen Register und schalten auf den nächsten ausführbaren Thread um.
- Auf Terminierung wartende Threads verwalten wir in der Tabelle JTab.
- Dort legen wir auch den Rückgabe-Wert der Threads ab :-)

Beispiel:



Thread 0 ist am Laufen, Thread 1 könnte laufen, Threads 2 und 3 warten auf Terminierung von 1, und Thread 4 wartet auf Terminierung von 3.

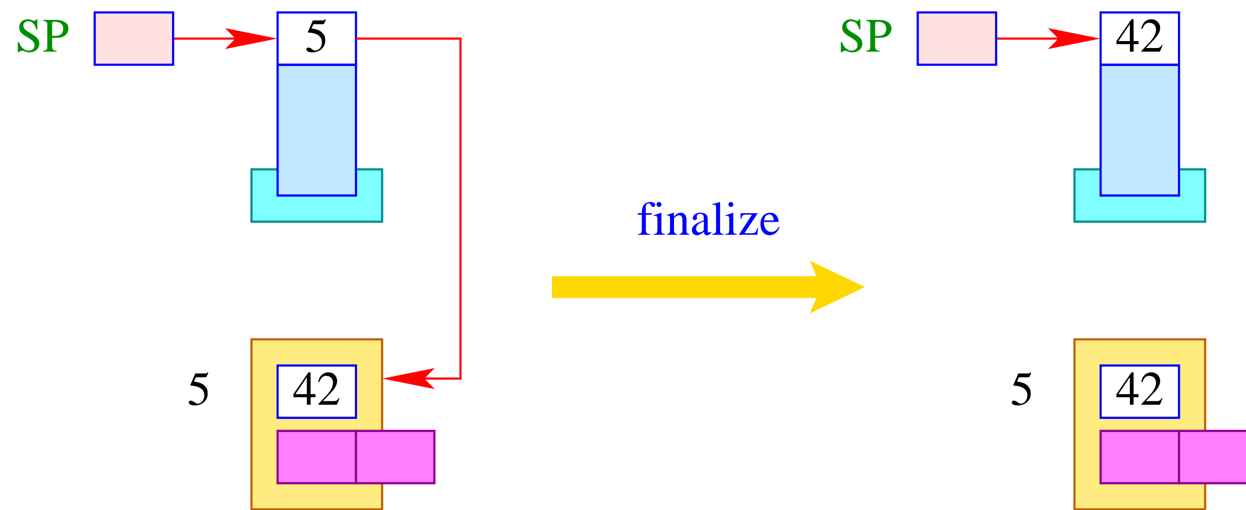
Damit übersetzen wir:

$$\text{code}_R\ \textbf{join}\ (e)\ \rho \quad = \quad \begin{array}{l} \text{code}_R\ e\ \rho \\ \text{join} \\ \text{finalize} \end{array}$$

... wobei die Instruktion join definiert ist als:

```
tid = S[SP];
if (TTab[tid][1] ≥ 0) {
        enqueue ( JTab[tid], CT );
        next
    }
```

... sowie:



S[SP] = JTab[tid][1];

Die Instruktions-Folge:

term
next

soll zuletzt ausgeführt werden, bevor ein Thread terminiert.
Deshalb schreiben wir sie auch an die Stelle f.

Die Instruktion next schaltet zum nächsten lauffähigen Thread weiter.
Vorher muss allerdings noch:

- ... der letzte Kellerrahmen aufgegeben und das Resultat in der Tabelle JTab abgelegt werden;
- ... kenntlich gemacht werden, dass der Thread terminiert ist, z.B. indem der PC auf -1 gesetzt wird;
- ... sämtliche Threads aufgeweckt werden, die auf Beendigung des Threads gewartet haben.

Für die Instruktion term heißt das:

```
PC = –1;
JTab[CT][1] = S[SP];
freeStack(SP);
while (0 ≤ tid = dequeue ( JTab[CT][0] ))
        enqueue ( RQ, tid );
```

Die Laufzeit-Funktion freeStack (int adr) beseitigt den (ein-elementigen) Stack an der Stelle adr :

46 Wechselseitiger Ausschluss

Ein Mutex ist ein (abstrakter) Datentyp (in der Halde), die es der Programmiererin gestatten soll, gemeinsame Resourcen für einen Thread exklusiv zu reservieren (wechselseitiger Ausschluss / mutual exclusion).

Der Datentyp unterstützt folgende Operationen:

Mutex * newMutex ();	—	legt neuen Mutex an;
void lock (**Mutex** *me);	—	versucht, den Mutex zu erwerben;
void unlock (**Mutex** *me);	—	versucht, den Mutex frei zu geben.

Achtung:

Ein Thread darf einen Mutex nur frei geben, wenn es über diesen verfügt :-)

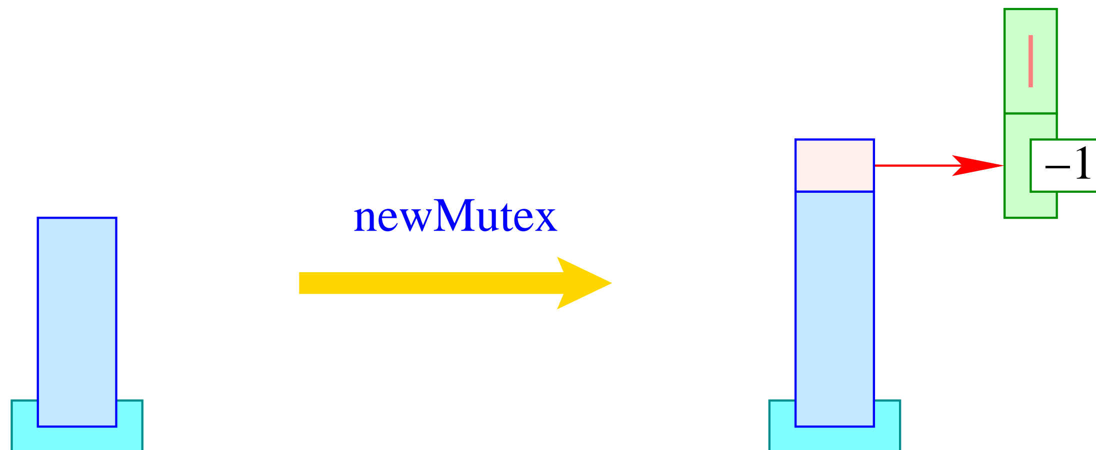
Ein Mutex me besteht aus:

- der tid des gegenwärtigen Besitzers (bzw. -1 falls es keinen gibt);
- der Schlange BQ der blockierten Threads, die den Mutex erwerben wollen.

Dann übersetzen wir:

$$\text{code}_\text{R}\ \textbf{newMutex}\ ()\ \rho \quad = \quad \text{newMutex}$$

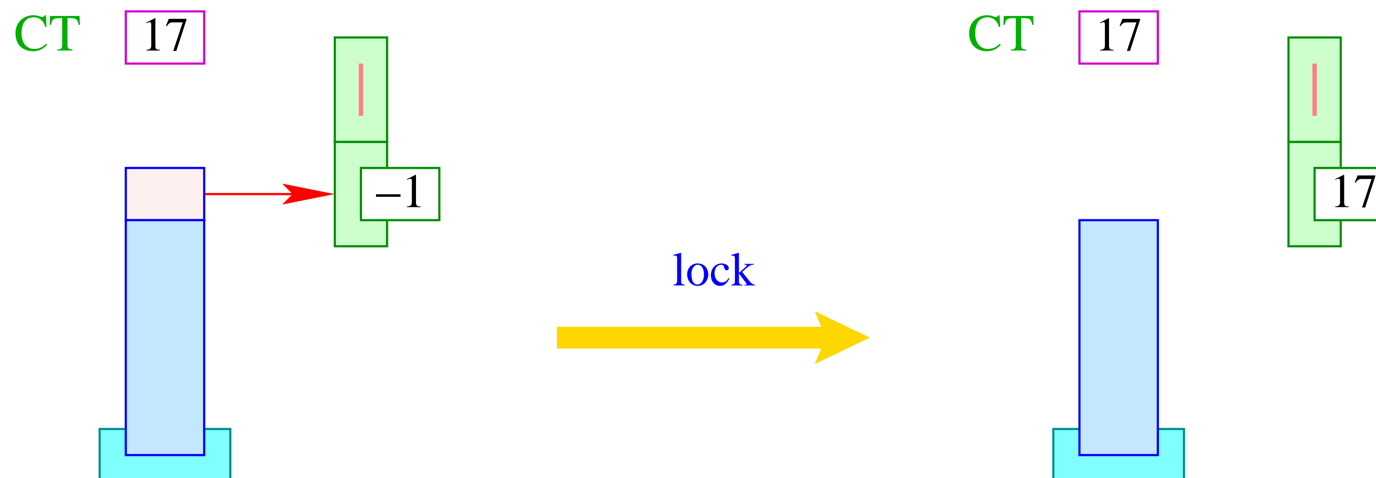
wobei:



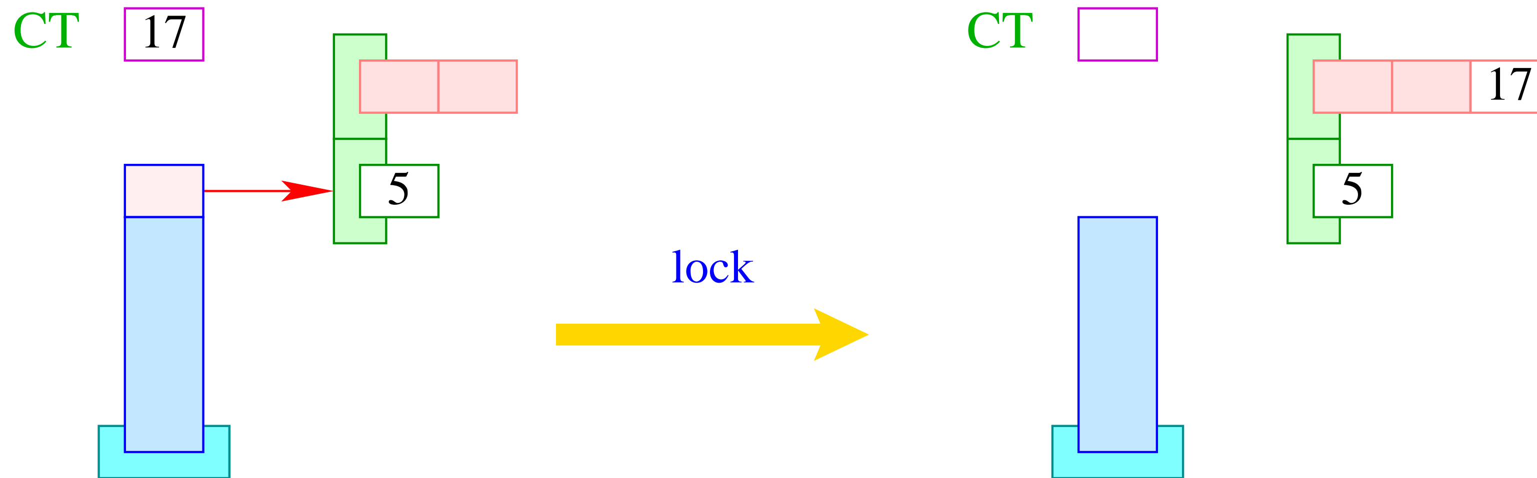
Dann übersetzen wir:

$$\text{code } \mathbf{lock}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{lock}$$

wobei:



Ist der Mutex bereits vergeben, wird der aktuelle Thread unterbrochen:

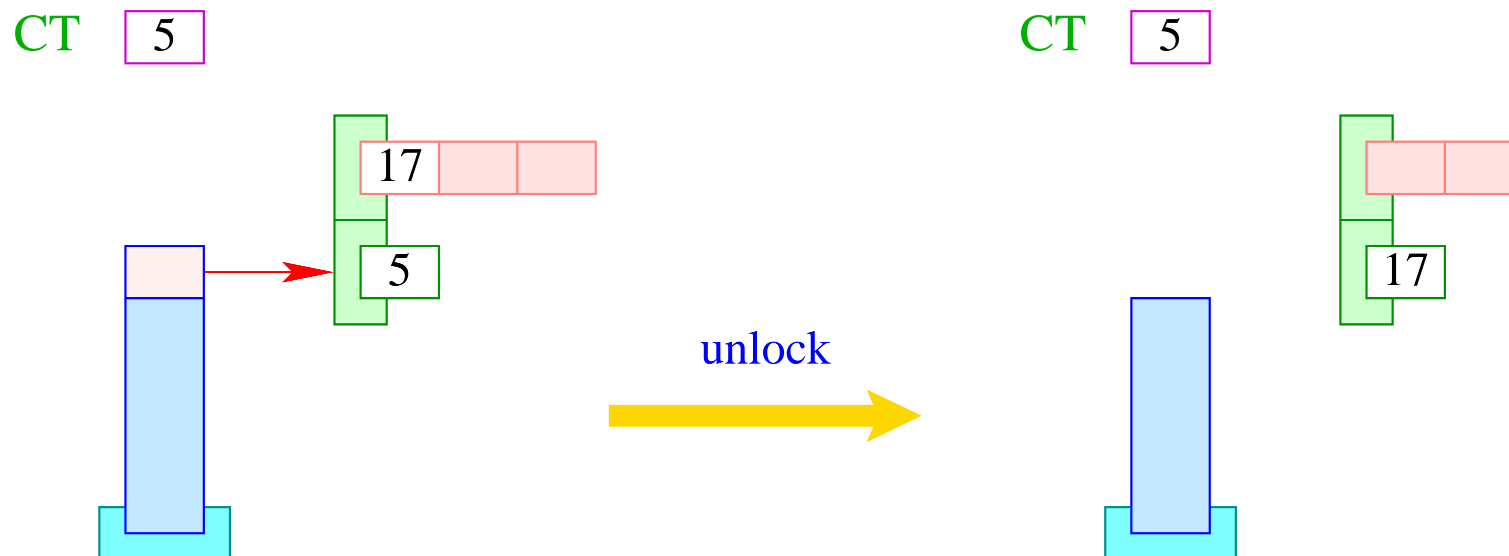


if $(S[S[SP]] < 0)$ $\quad S[S[SP--]] = CT;$
else {
enqueue (S[SP−−]+1, CT);
next;
}

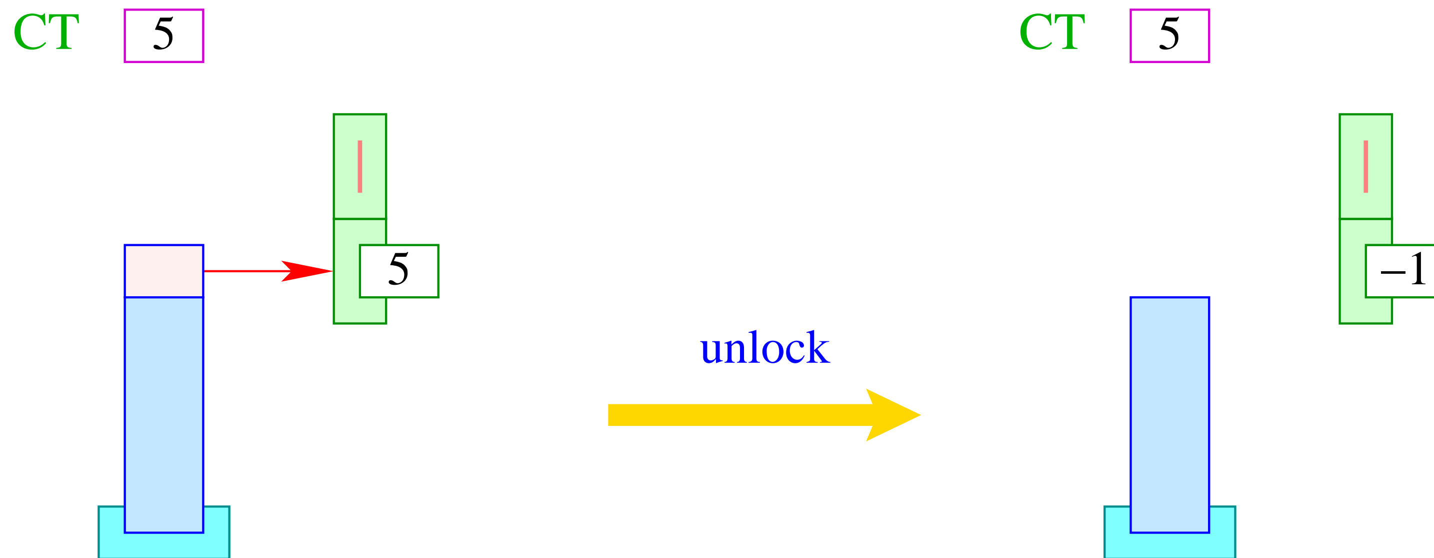
Entsprechend übersetzen wir:

$$\text{code } \mathbf{unlock}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{unlock}$$

wobei:



Ist die Schlange BQ leer, geben wir den Mutex ganz frei:

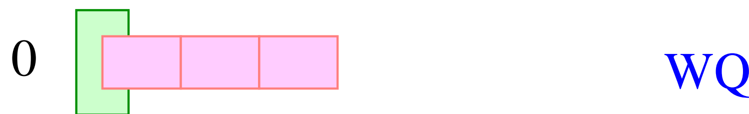


if (S[S[SP]] $\neq$ CT) Error ("Illegal unlock!");
if (0 $>$ tid $=$ dequeue (S[SP]+1)) S[S[SP– –]] = –1;
else {
S[S[SP--]] = tid;
enqueue (RQ, tid);
}

47 Warten auf den Frühling

Es kann vorkommen, dass ein Thread zwar über einen Mutex verfügt, nun aber warten muss, bis eine Bedingung eingetreten ist.

Dann soll der Thread sich selbst blockieren, um später reaktiviert zu werden. Dazu dienen Bedingungsvariablen. Eine Bedingungsvariable besteht aus einer einer Schlange WQ wartender Threads :-)



WQ

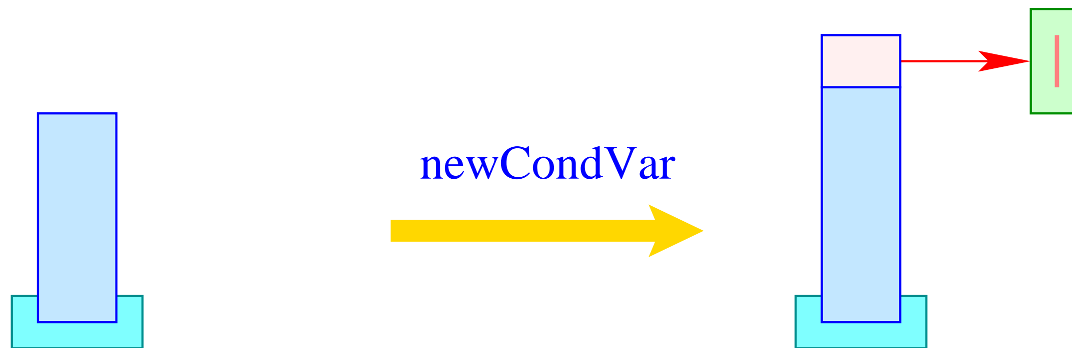
Für Bedingungsvariablen gibt es die Funktionen:

CondVar $*$ newCondVar ();	—	legt eine Bedingungsvariable an;
void wait (**CondVar** $*$ cv), **Mutex** $*$ me);	—	legt aktuellen Thread schlafen;
void signal (**CondVar** $*$ cv);	—	weckt einen wartenden Thread;
void broadcast (**CondVar** $*$ cv);	—	weckt alle wartenden Threads.

Dann übersetzen wir:

$$\text{code}_\text{R}\ \textbf{newCondVar}\ ()\ \rho \quad = \quad \text{newCondVar}$$

wobei:

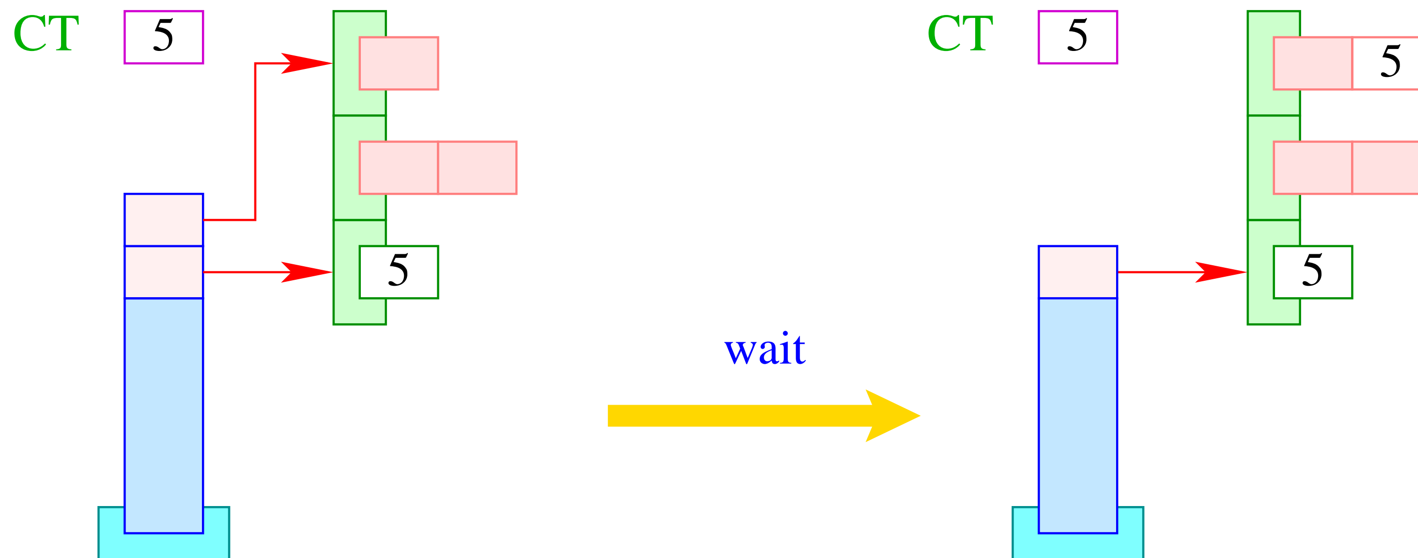


Nach Einreihen in die Warteschlange wird der Mutex wieder frei gegeben. Nach dem Aufwecken muss dieser allerdings neu erworben werden.

Darum übersetzen wir:

$$
\begin{array}{lcl}
\text{code } \textbf{wait } (e_0, e_1);\ \rho & = & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{code}_\text{R}\ e_0\ \rho \\
 & & \text{wait} \\
 & & \text{dup} \\
 & & \text{unlock} \\
 & & \text{next} \\
 & & \text{lock}
\end{array}
$$

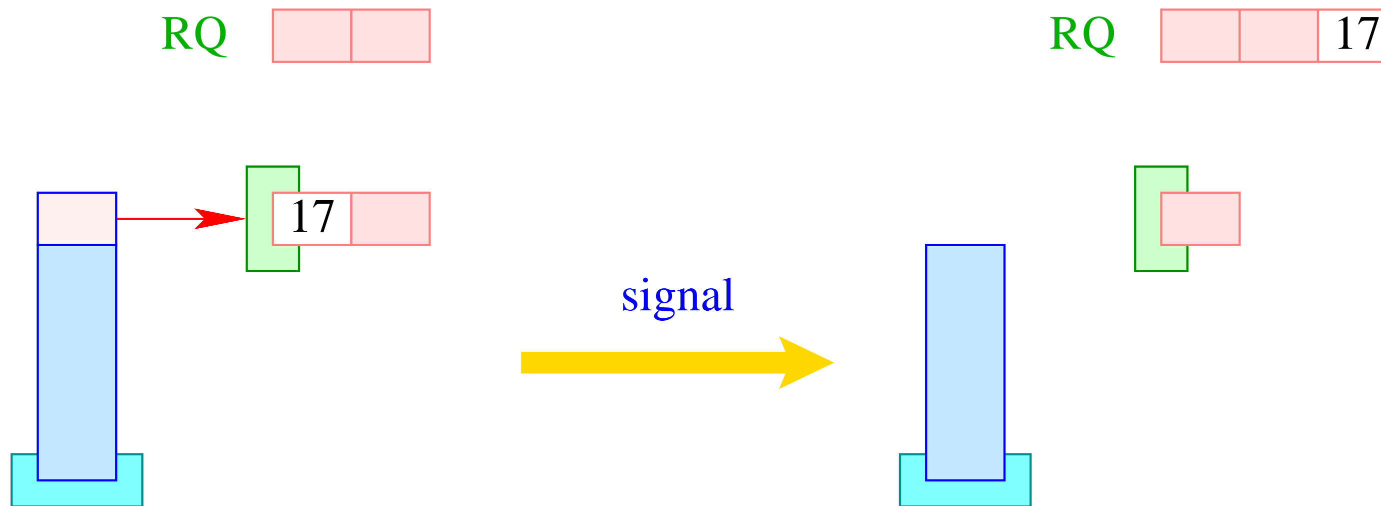
wobei ...



if (S[S[SP-1]] $\neq$ CT) Error ("Illegal wait!");
enqueue (S[SP], CT); SP--;

Entsprechend übersetzen wir:

$$\text{code } \mathbf{signal}\ (e);\ \rho \quad = \quad \text{code}_\text{R}\ e\ \rho$$
$$\text{signal}$$




```
if (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

Analog:

$$\text{code } \textbf{broadcast } (e);\ \rho \quad = \quad \text{code}_{\text{R}}\ e\ \rho$$
$$\text{broadcast}$$

wobei die Instruktion broadcast sämtliche Threads der Schlange WQ in die Schlange RQ einfügt:

```
while (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

Achtung:

Die aufgeweckten Threads sind nicht blockiert !!!

Wenn sie aktiv werden, benötigen sie jedoch als erstes das Lock ihres Mutex :-)

48 Beispiel: Semaphore

Ein Semaphor ist ein abstrakter Datentyp, der den Zugang zu einer festen Anzahl (identischer) Resourcen regeln soll.

Operationen:

Sema ∗ newSema (**int** n)	—	liefert einen Semaphor;
void Up (Sema ∗ s)	—	gibt eine Resource frei;
void Down (Sema ∗ s)	—	allokiert eine Resource.

Ein Semaphor besteht darum aus:

- einem Zähler vom Typ **int**;
- einem Mutex zur Synchronisation der Semaphor-Operationen;
- einer Bedingungsvariablen.

```
typedef struct {
        Mutex * me;
        CondVar * cv;
        int count;
    } Sema;
```

```
Sema * newSema (int n) {
        Sema * s;
        s = (Sema *) malloc (sizeof (Sema));
        s→me = newMutex ();
        s→cv = newCondVar ();
        s→count = n;
        return (s);
    }
```

Die Übersetzung liefert für den Rumpf:

```
alloc 1      newMutex    newCondVar    loadr 1     loadr 2
loadc 3      loadr 2     loadr 2       loadr 2     storer -2
new          store       loadc 1       loadc 2     return
storer 2     pop         add           add
pop                      store         store
                         pop           pop
```

Die Funktion Down() dekrementiert den Zähler.

Rutscht dieser dadurch ins Negative, wird **wait** aufgerufen:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count– –;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```

Die Übersetzung liefert für den Rumpf:

```
alloc 1      loadc 2      add               loadc 1
loadr 1      add          store             add
load         load         loadc 0           load
storer 2     loadc 1      le                wait
lock         sub          jumpz A      A:   loadr 2
             loadr 1      loadr 2           unlock
loadr 1      loadc 2      loadr 1           return
```

Die Funktion `Up()` inkrementiert den Zähler wieder.

Ist dieser danach noch nicht positiv, gibt es wartende Threads, von denen einer ein Signal erhält:

```
void Up (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```

Die Übersetzung liefert für den Rumpf:

```
alloc 1      loadc 2      add               loadc 1
loadr 1      add          store             add
load         load         loadc 0           load
storer 2     loadc 1      le                signal
lock         add          jumpz A      A:   loadr 2
             loadr 1                        unlock
loadr 1      loadc 2      loadr 1           return
```

49 Stack-Management

Problem:

- Alle Threads leben in einem gemeinsamen Speicher.
- Jeder Thread benötigt (konzeptuell) einen eigenen Stack.

1. Idee:

Allokiere für jeden neuen Thread einen festen Speicherbereich auf der Halde!

Dann implementieren wir:

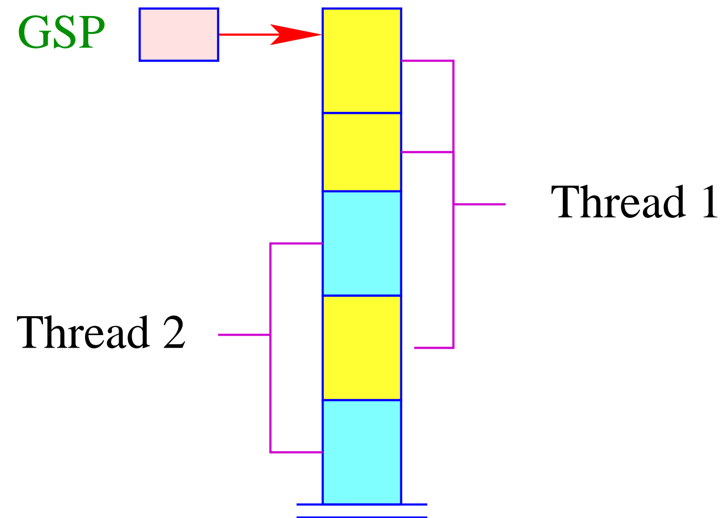
```
void *newStack() { return malloc(M); }
void freeStack(void *adr) { free(adr); }
```

Problem:

- Manche Threads brauchen viel, manche weniger Stack-Space.
- Evt. ist der nötige Platz statisch gar nicht bekannt :-(

2. Idee:

- Verwalte sämtliche Keller zusammen in einem Frame-Heap FH :-)
- Sorge dafür, dass der Platz im Rahmen zumindest ausreicht zum Abarbeiten des aktuellen Funktionsaufrufs.
- Ein globaler Stack-Pointer GSP gibt an, wieviel Platz bereits vergeben ist...



Allokation und De-Allokation eines Frames erfolgt mittels Laufzeitfunktionen:

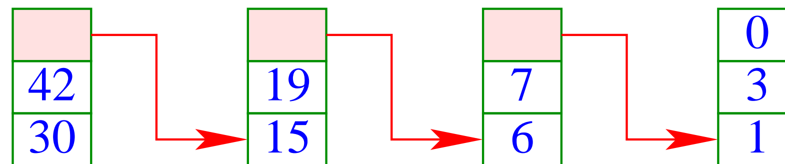
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

void freeFrame(int sp, int size);
```

Achtung:

Der frei zu gebende Block kann im Innern des Stacks liegen :-(

Wir verwalten eine Liste der freigegebenen Abschnitte des Stacks :-)

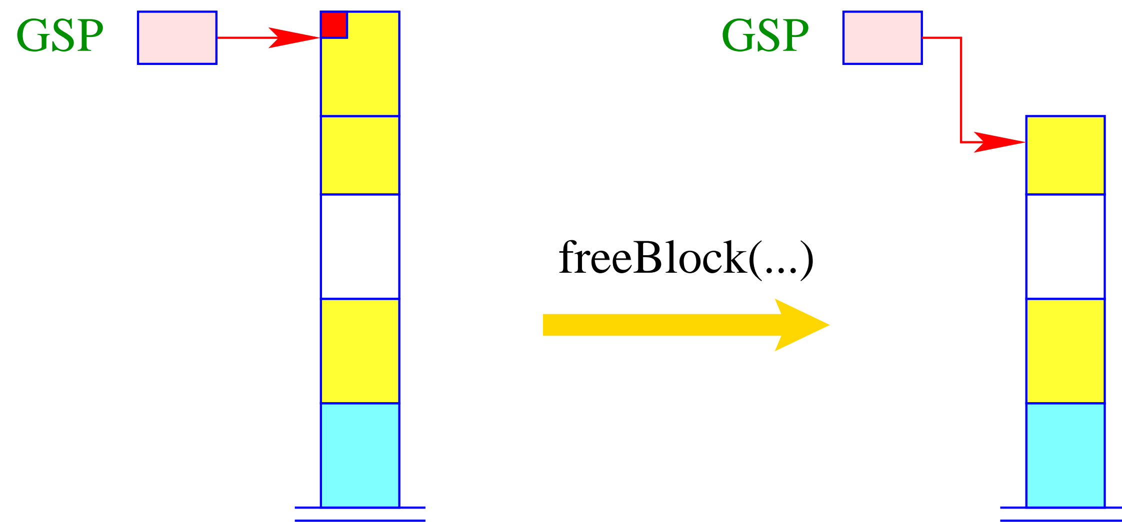


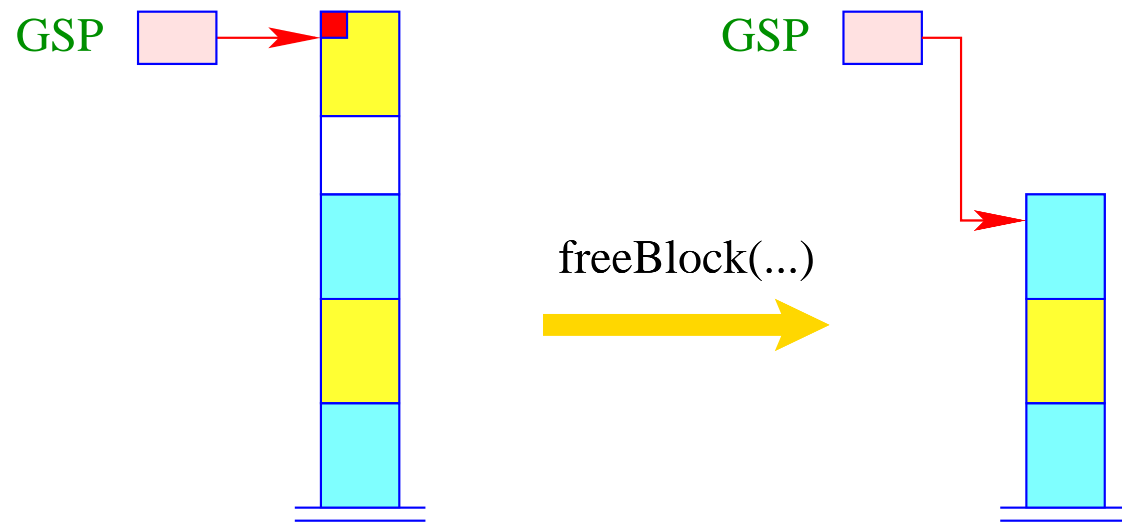
Diese Liste unterstützt eine Funktion

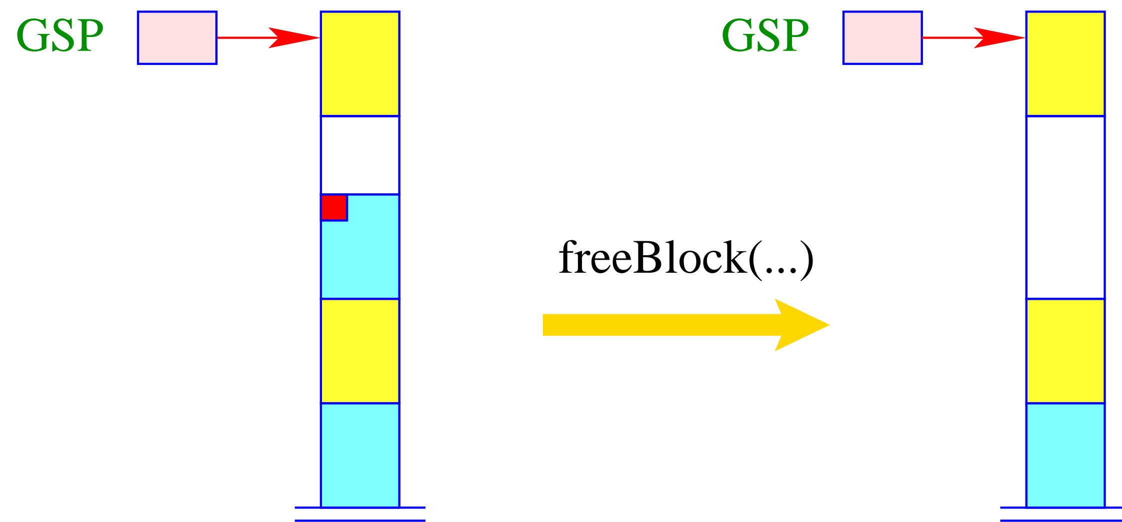
```
void insertBlock(int max, int min)
```

die es gestattet, einzelne Blocks frei zugeben.

- Liegt der Block am oberen Ende des Stacks geben wir ihn sofort frei;
- ... Wie den darunter liegenden Abschnitt – falls dieser bereits de-allokiert ist.
- Liegt er im Innern, verschmelzen wir ihn mit angrenzenden freien Blöcken:


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)

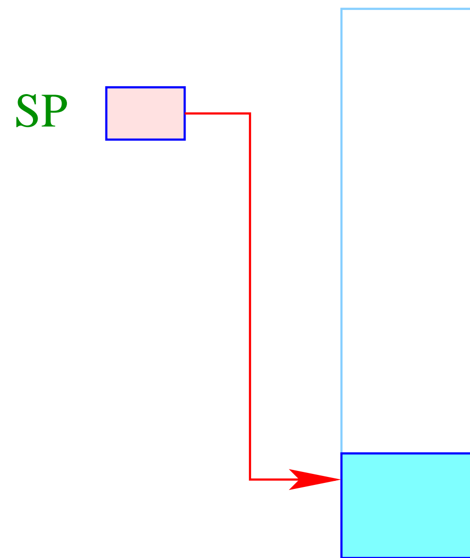
Ansatz:

Wir allokieren einen neuen Block für jeden Funktions-Aufruf ...

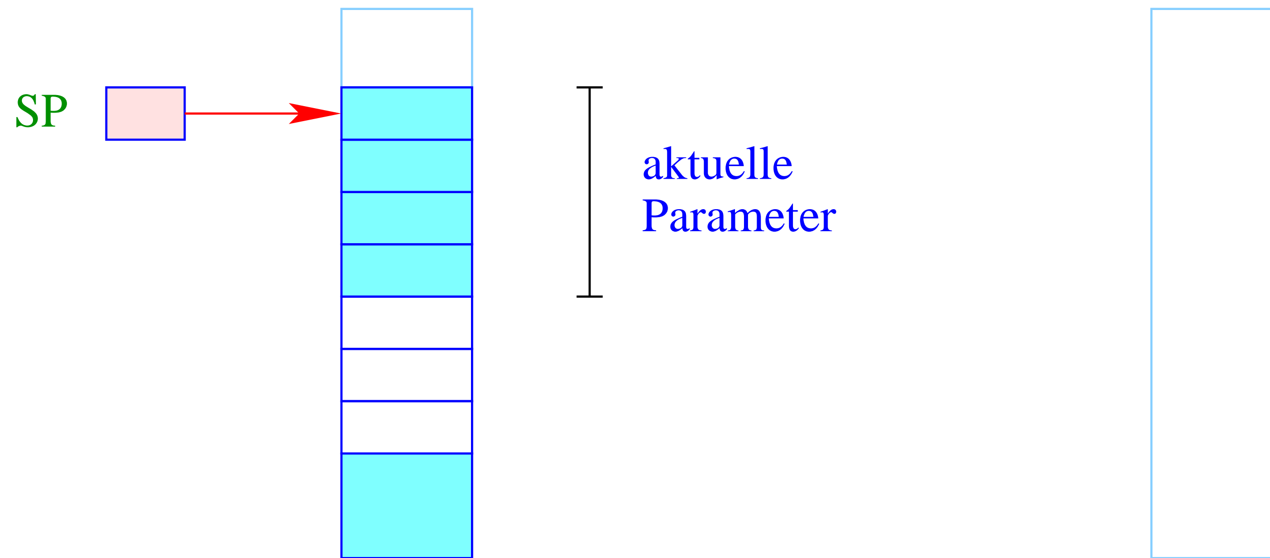
Problem:

Bei Anforderung des neuen Blocks vor dem Aufruf ist der Speicherbedarf der aufgerufenen Funktion noch gar nicht bekannt :-(

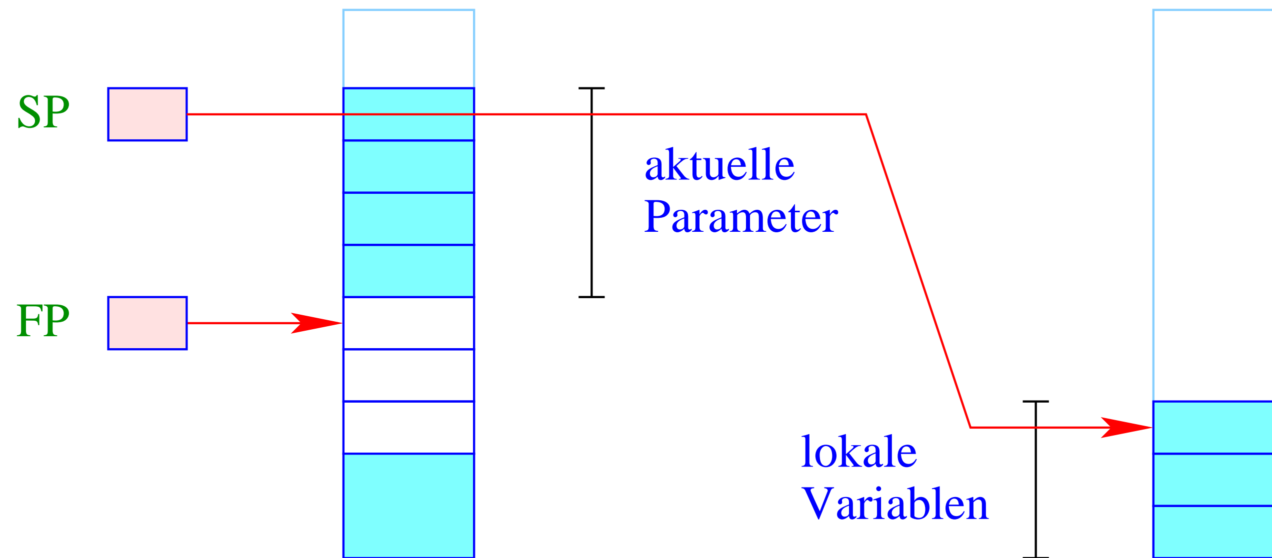
$\Longrightarrow$ Wir können den neuen Block erst bei Betreten des Funktions-Rumpfs anfordern!



Organisatorische Zellen wie aktuelle Parameter müssen noch im alten Block angelegt werden ...



Bei Betreten der neuen Funktion allokieren wir auch den neuen Block ...



Insbesondere liegen jetzt die lokalen Variablen im neuen Block ...

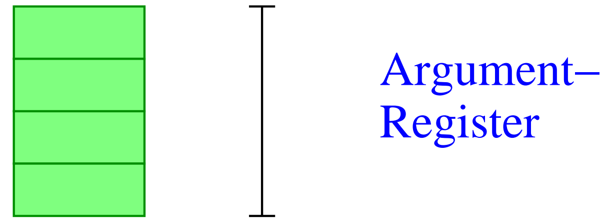
Wir adressieren ...

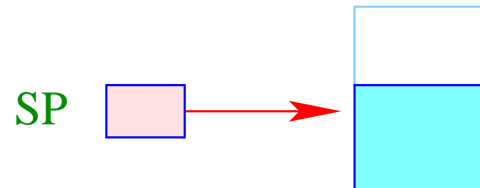
- die formalen Parameter relativ zum Frame-Pointer;
- die lokalen Variablen relativ zum Stack-Pointer :-)

Wir müssen die gesamte Code-Erzeugung umstellen ... :-(

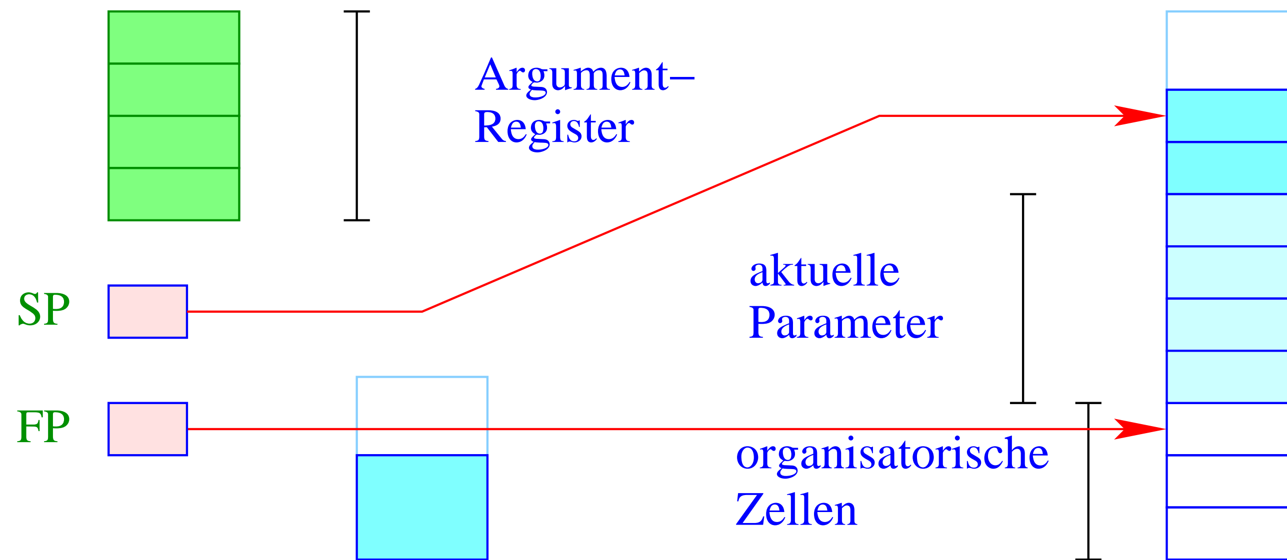
Ausweg:

Übergabe von Parametern in Registern ... :-)

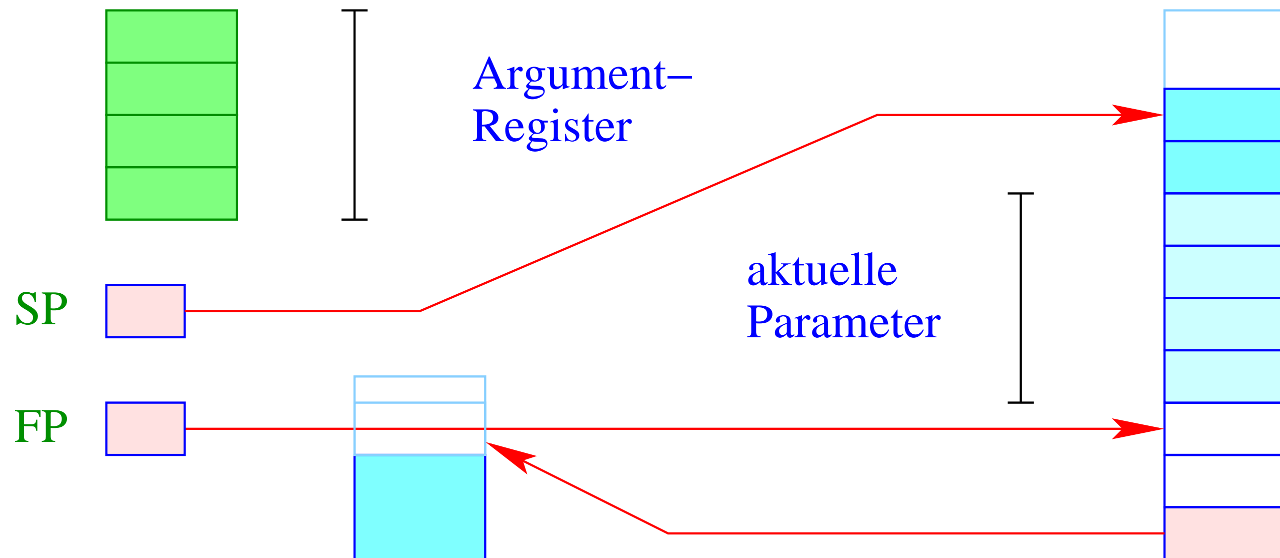




Die Werte der aktuellen Parameter werden vor Anlegen des neuen Keller-Rahmens ermittelt.



Der gesamte Rahmen wird im neuen Block angelegt – inclusive Platz für die aktuellen Parameter.



Im neuen Block müssen wir allerdings uns auch den alten SP (evt. +1) merken, damit das Ergebnis korrekt zurück geliefert werden kann ...

3. Idee: Hybrid-Lösung

- Für die ersten k Threads lege jeweils einen eigenen Speicherbereich an!
- Für alle weiteren benutze reihum einen der bereits vorhandenen ...

- Für wenige Threads extrem einfach und effizient;
- Für viele Threads amortisierte Speicher-Ausnutzung ...

Classes and Objects

Example:

```
int count = 0;
class list {
        int info;
        class list * next;
        list (int x) {
                info = x; count++; next = null;
            }
        virtual int last () {
                if (next == null) return info;
                else return next → last ();
            }
    }
```

Discussion:

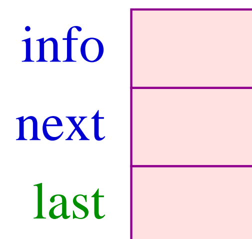
- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular ...
- Classes are considered as extensions of structs. They may comprise:
  - ⇒ attributes, i.e., data fields;
  - ⇒ constructors;
  - ⇒ member functions which either are virtual, i.e., are called depending on the run-time type or non-virtual, i.e., called according to the static type of an object :-)
  - ⇒ static member functions which are like ordinary functions :-))
- We ignore visibility restrictions such as **public**, **protected** or **private** but simply assume general visibility.
- We ignore multiple inheritance :-)

50 Object Layout

Idea:

- Only attributes and virtual member functions are stored inside the class !!
- The addresses of non-virtual or static member functions as well as of constructors can be resolved at compile-time :-)
- The fields of a sub-class are appended to the corresponding fields of the super-class ...

... in our Example:



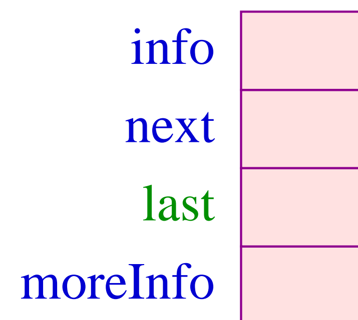
Idea (cont.):

- The fields of a sub-class are appended to the corresponding fields of the super-class :-)

Example:

```
class mylist : list {
        int moreInfo;
    }
```

... results in:



For every class C we assume that we are given an adress environment ρ_C .

ρ_C maps every identifier x visible inside C to its decorated relative address a . We distingish:

global variable	(G, a)
local variable	(L, a)
attribute	(A, a)
virtual function	(V, b)
non-virtual function	(N, a)
static function	(S, a)

For virtual functions x , we do not store the starting address of the code — but the relative address b of the field of x inside the object :-)

For the various of variables, we obtain for the L-values:

$$\text{code}_L\ x\ \rho \ = \begin{cases} \text{loadr } 1 & \text{if} \quad x = \textbf{this} \\[1em] \text{loadc } a & \text{if} \quad \rho\ x = (G, a) \\[1em] \text{loadr } a & \text{if} \quad \rho\ x = (L, a) \\[1em] \text{loadr } 1 & \\ \text{loadc } a & \\ \text{add} & \text{if} \quad \rho\ x = (A, a) \end{cases}$$

In particular, the pointer to the current object has relative address 1 :-)

Accordingly, we introduce the abbreviated operations:

```
loadm q          =   loadr 1
                     loadc q
                     add
                     load

bla ; storem q   =   loadr 1
                     loadc q
                     add
                     bla
                     store
```

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register COP :-)
- This register must updated before calls to non-static member functions and restored after the call.
- We have refrained from doing so since
  - → Only some functions are member functions :-)
  - → We want to reuse as much of the C-machine as possible :-))

51 Calling Member Functions

Static member functions are considered as ordinary functions :-)

For non-static member functions, we distinguish two forms of calls:

(1) directly: $f\ (e_2, \ldots, e_n)$

(2) relative to an object: $e_1.f\ (e_2, \ldots, e_n)$

Idea:

- The case (1) is considered as an abbreviation of **this**.$f\ (e_2, \ldots, e_n)$:-)
- The object is passed to f as an implicit first argument :-)
- If f is non-virtual, proceed as with an ordinary call of a function :-)
- If f is virtual, insert an indirect call :-)

A non-virtual function:

$$
\begin{array}{lll}
\text{code}_R \;\; e_1.f\;(e_2,\ldots,e_n)\;\;\rho & = & \text{mark} \\
 & & \text{code}_L \;\; e_1 \;\; \rho \\
 & & \text{code}_R \; e_2 \; \rho \\
 & & \ldots \\
 & & \text{code}_R \; e_n \; \rho \\
 & & \text{loadc} \; _f \\
 & & \text{call m} + 1
\end{array}
$$

where $(F, _f) \;=\; \rho_C(f)$

C = class of e_1

m = space for the actual parameters

Note:

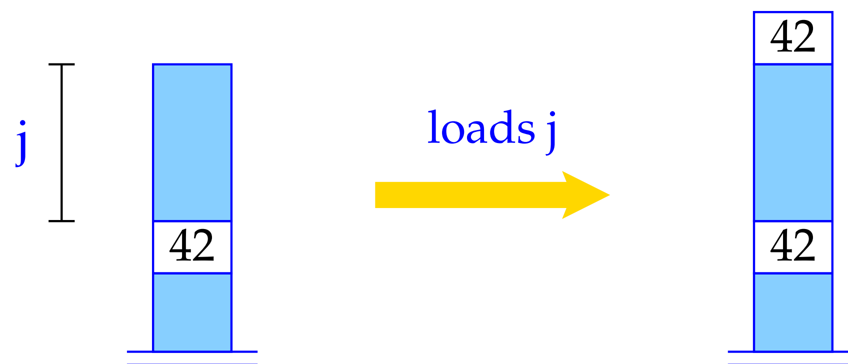
The pointer to the object is obtained by computing the L-value of e_1 :-)

A virtual function:

$$
\begin{array}{lll}
\text{code}_\text{R}\ \ e_1.f\ (e_2, \ldots, e_n)\ \ \rho & = & \text{mark} \\
 & & \text{code}_\text{L}\ \ e_1\ \ \rho \\
 & & \text{code}_\text{R}\ e_2\ \rho \\
 & & \ldots \\
 & & \text{code}_\text{R}\ e_n\ \rho \\
 & & \text{loads m} \\
 & & \text{loadc}\ b \\
 & & \text{add}\ ;\ \ \text{load} \\
 & & \text{call m} + 1
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & (V, b) \ = \ \rho_\text{C}(f) \\
 & \text{C} = \text{class of} \quad e_1 \\
 & \text{m} = \text{space for the actual parameters}
\end{array}
$$

The instruction loads j loads relative to the stack pointer:



S[SP+1] = S[SP−j];

SP++;

... in the Example:

The recursive call

$$\text{next} \rightarrow \text{last}\ ()$$

in the body of the virtual method last is translated into:

```
mark
loadm 1
loads 0
loadc 2
add
load
call 1
```

52 Defining Member Functions

In general, a definition of a member function for class C looks as follows:

$$d \equiv t\ f\ (t_2\ x_2, \ldots, t_n\ x_n)\ \{\ ss\ \}$$

Idea:

- f is treated like an ordinary function with one extra implicit argument
- Inside f a pointer **this** to the current object has relative address 1 :-)
- Object-local data must be addressed relative to **this** ...

$$
\begin{array}{llll}
\text{code}_D\; d\; \rho \;=\; _f: & \text{enter q} & // & \text{Setting the EP} \\
& \text{alloc m} & // & \text{Allocating the local variables} \\
& \text{code}\; ss\; \rho_1 & & \\
& \text{return} & // & \text{Leaving the function}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & \text{q} & = & maxS + \text{m} \quad \text{where} \\
& maxS & = & \text{maximal depth of the local stack} \\
& \text{m} & = & \text{space for the local variables} \\
& \text{k} & = & \text{space for the formal parameters (including } \textbf{this}) \\
& \rho_1 & = & \text{local address environment}
\end{array}
$$

... in the Example:

```
_last:  enter 6
        alloc 0
        loadm 1
        loadc 0
        eq
        jumpz A
        loadm 0
        storer -3
        return

A:      mark
        loadm 1
        loads 0
        loadc 2
        add
        load
        call 1
        storer -3
        return
```

53 Calling Constructors

Every new object should be initialized by (perhaps implicitly) calling a constructor. We distinguish two forms of object creations:

(1) directly: $x = C\,(e_2, \ldots, e_n);$

(2) indirectly: $\textbf{new}\ C\,(e_2, \ldots, e_n)$

Idea for (2):

- Allocate space for the object and return a pointer to it on the stack;
- Initialize the fields for virtual functions;
- Pass the object pointer as first parameter to a call to the constructor;
- Proceed as with an ordinary call of a (non-virtual) member function :-)
- Unboxed objects are considered later ...

$$\text{code}_R \ \textbf{new}\ C\ (e_2, \ldots, e_n);\ \rho \quad = \quad \begin{array}{l} \text{malloc}\ |C| \\ \text{initVirtual}\ \ C \\ \text{mark} \\ \text{loads}\ 4 \qquad // \quad \text{loads relative to SP} \quad \text{:-)} \\ \text{code}_R\ e_2\ \rho \\ \ldots \\ \text{code}_R\ e_n\ \rho \\ \text{loadc}\ _C \\ \text{call}\ m+1 \\ \text{pop} \end{array}$$

where m = space for the actual parameters.

Note:

Before calling the constructor, we initialize all fields of virtual functions.

The pointer to the object is copied into the frame by a new instruction :-)

Assume that the class C lists the virtual functions $f_1, \ldots, f_r$ for C with the offsets and initial addresses: b_i and a_i, respectively:

Then:

$$
\begin{array}{lll}
\text{initVirtual } C & = & \text{dup} \\
 & & \text{loadc } b_1 \text{ ; add} \\
 & & \text{loadc } a_1 \text{ ; store} \\
 & & \text{pop} \\
 & & \ldots \\
 & & \text{dup} \\
 & & \text{loadc } b_r \text{ ; add} \\
 & & \text{loadc } a_r \text{ ; store} \\
 & & \text{pop}
\end{array}
$$

54 Defining Constructors

In general, a definition of a constructor for class C looks as follows:

$$d \equiv C\ (t_2\ x_2, \ldots, t_n\ x_n)\ \{\ ss\ \}$$

Idea:

- Treat the constructor as a definition of an ordinary member function :-)

... in the Example:

```
_list:   enter 3      loada 1     loadc 0
         alloc 0      dup         storem 1
         loadr 2      loadc 1     pop
         storem 0     add         return
         pop          storea 1
                      pop
                      pop
```

Discussion:

The constructor may issue further constructors for attributes if desired :-)

The constructor may call a constructor of the super class B as first action:

$$\text{code } B\,(e_2, \ldots, e_n);\ \rho \quad = \quad \begin{array}{l} \text{mark} \\ \text{loadr 1} \\ \text{code}_R\ e_2\ \rho \\ \quad \ldots \\ \text{code}_R\ e_n\ \rho \\ \text{loadc _}B \\ \text{call m} \end{array}$$

where m = space for the actual parameters.

Thus, the constructor is applied to the current object of the calling constructor :-)

55 Initializing Unboxed Objects

Problem:

The same constructor application can be used for initializing several variables:

$$x = x_1 = C\ (e_2, \ldots, e_n)$$

Idea:

- Allocate sufficient space for a temporary copy of a new C object.
- Initialize the temporary copy.
- Assign this value to the variables to be intialized :-)

$$
\begin{array}{lcl}
\text{code}_R\ C\ (e_2, \ldots, e_n)\ \rho & = & \text{stalloc}\ |C| \\
& & \text{initVirtual}\ C \\
& & \text{mark} \\
& & \text{loads}\ 4 \\
& & \text{code}_R\ e_2\ \rho \\
& & \ldots \\
& & \text{code}_R\ e_n\ \rho \\
& & \text{loadc}\ _C \\
& & \text{call}\ \text{m} + 1 \\
& & \text{pop} \\
& & \text{pop}
\end{array}
$$

where m = space for the actual parameters.

Note:

The instruction stalloc m is like malloc m but allocates on the stack :-)

We assume that we have assignments between complex types :-)

SP = SP+m+1;

S[SP] = SP−m;