

## 32.4 Endbehandlung von Klauseln

Erinnern wir uns an die Übersetzung von Klauseln:

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

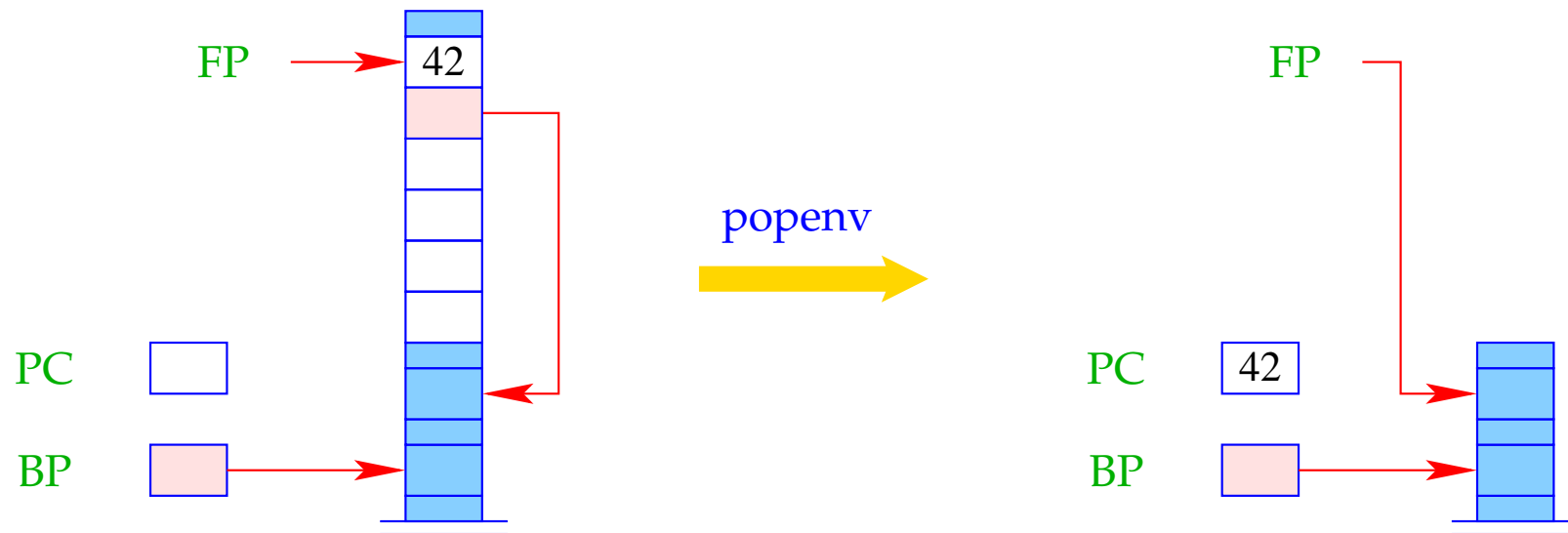
Der aktuelle Kellerrahmen kann aufgegeben werden ...

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

$\implies$  der aktuelle Rücksetz-Punkt ist **älter** :-)

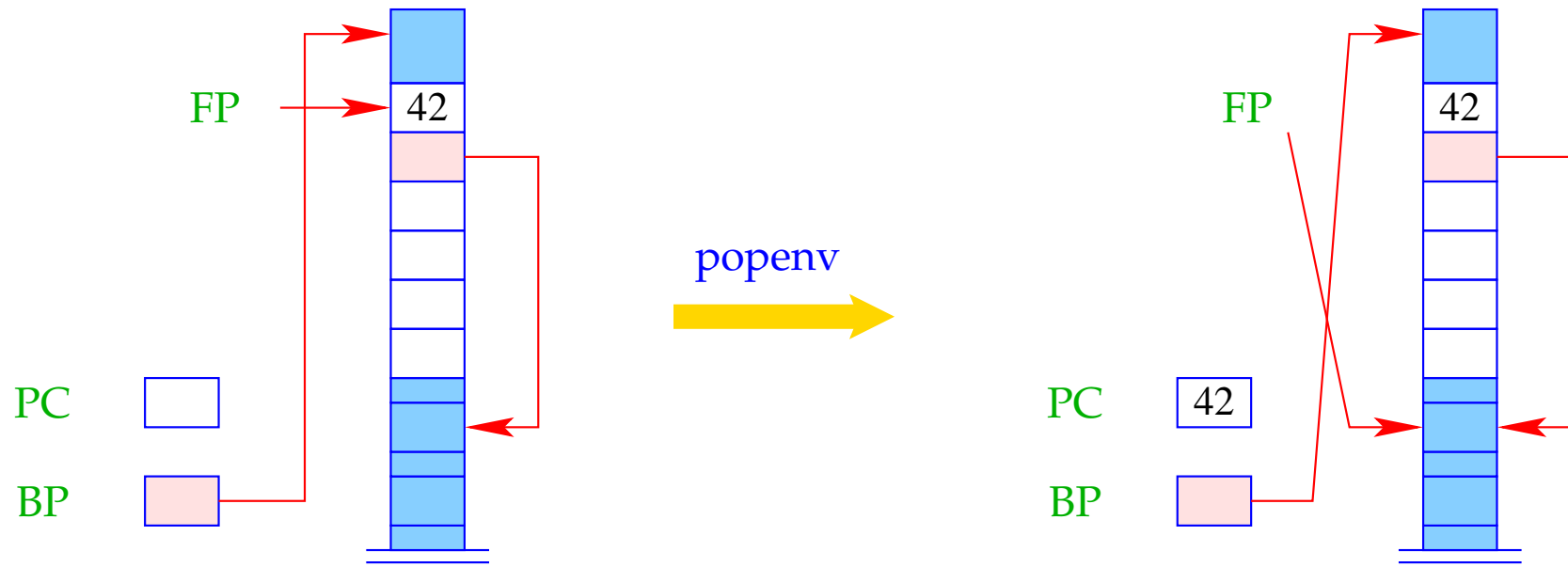
$\implies$  **FP** > **BP**

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  $p \equiv rr_1 \dots rr_h ?g$  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 angelegt werden.

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

```

code p =      init A
              pushenv d
              code_G g ρ
              halt d
A: no
              code_p rr_1
              ...
              code_p rr_h

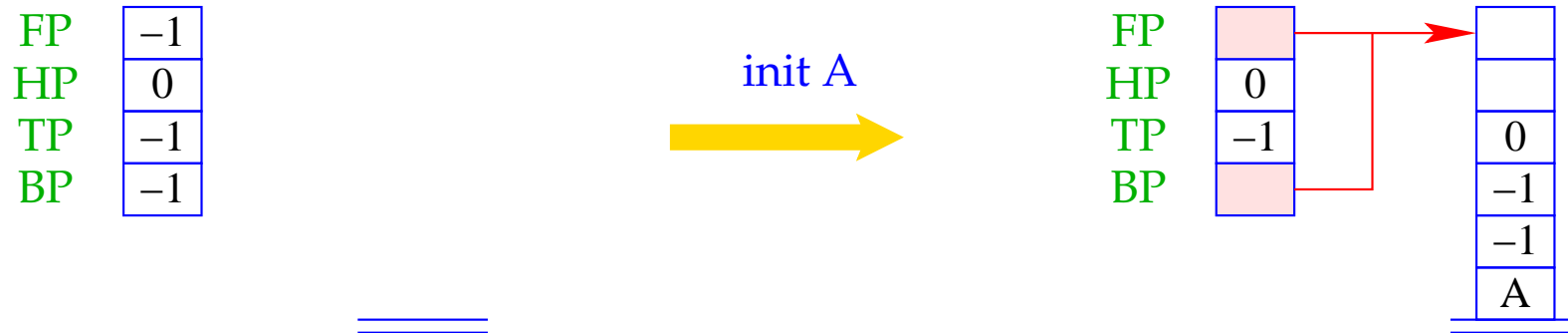
```

wobei  $free(g) = \{X_1, \dots, X_d\}$  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:

$$\begin{array}{lll}
 t(X) \leftarrow \bar{X} = b & q(X) \leftarrow s(\bar{X}) & s(X) \leftarrow \bar{X} = a \\
 p \leftarrow q(X), t(\bar{X}) & s(X) \leftarrow t(\bar{X}) & ? \quad p
 \end{array}$$

Die Übersetzung liefert:

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

## 34 Letzte Ziele

Betrachte das Prädikat `app` aus der Einleitung:

$$\text{app}(X, Y, Z) \leftarrow X = [], Y = Z$$

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

Wir beobachten:

- Der rekursive Aufruf ist das **letzte** Ziel der Klausel.
- Ein solches Ziel ist ein **letzter Aufruf** :-)
  - ⇒ wir versuchen, es im **aktuellen** Kellerrahmen auszuwerten !!!
  - ⇒ nach (erfolgreicher) Beendigung werden wir nicht mehr zum gegenwärtigen Aufrufer zurückkehren !!!



Betrachten wir die Klausel  $r$ :  $p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n$   
 mit  $m$  lokalen Variablen, wobei  $g_n \equiv q(t_1, \dots, t_h)$ . Das Zusammenspiel  
 zwischen  $\text{code}_C$  und  $\text{code}_G$ :

```

code_C r =      pushenv m
                code_G g_1 ρ
                ...
                code_G g_{n-1} ρ
                mark B
                code_A t_1 ρ
                ...
                code_A t_h ρ
                call q/h
                B : popenv
  
```

Ersetzung:     $\text{mark B} \implies \text{lastmark}$   
                   $\text{call q/h; popenv} \implies \text{lastcall q/h m}$

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

```

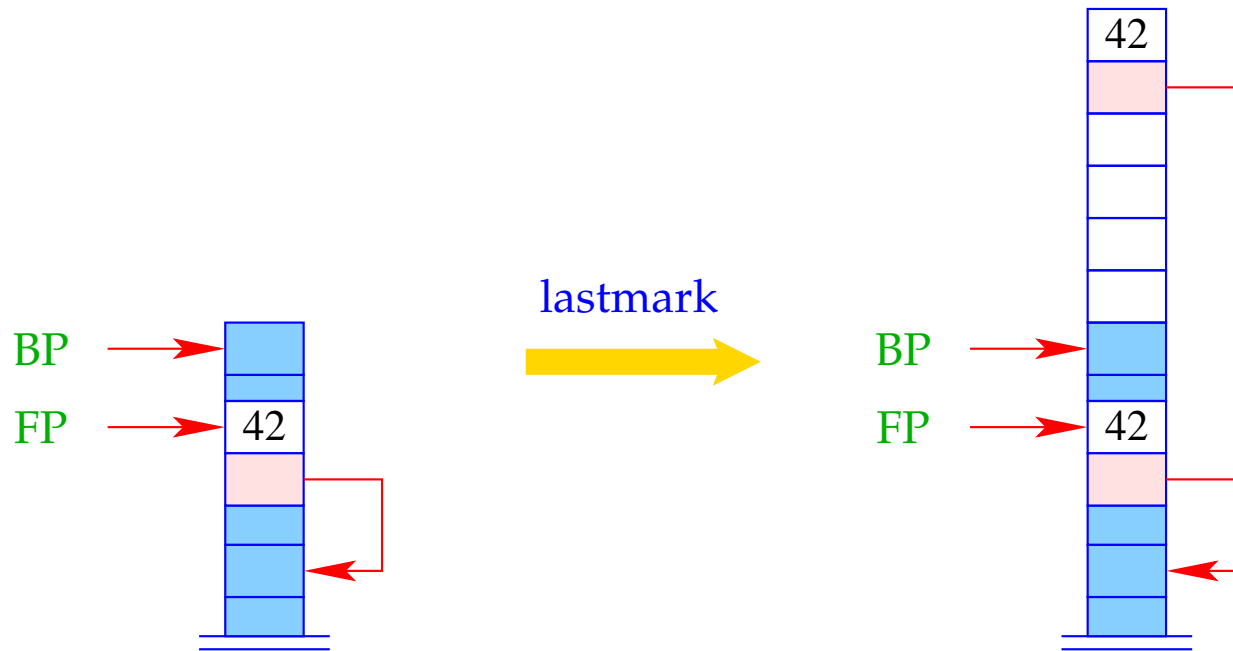
codeC r =      pushenv m
                codeG g1 ρ
                ...
                codeG gn-1 ρ
                lastmark
                codeA t1 ρ
                ...
                codeA th ρ
                lastcall q/h m

```

Ersetzung:	mark B	⇒	lastmark
	call q/h; popenv	⇒	lastcall q/h m

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

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



```

if (FP  $\leq$  BP) {
    SP = SP + 6;
    S[SP] = posCont; S[SP-1] = FPold;
}

```

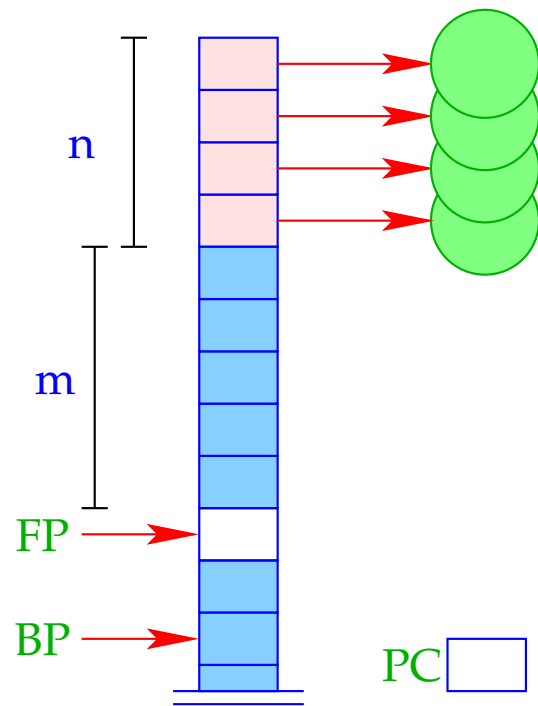
Falls **FP**  $>$  **BP** dann tut **lastmark** nichts :-)

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

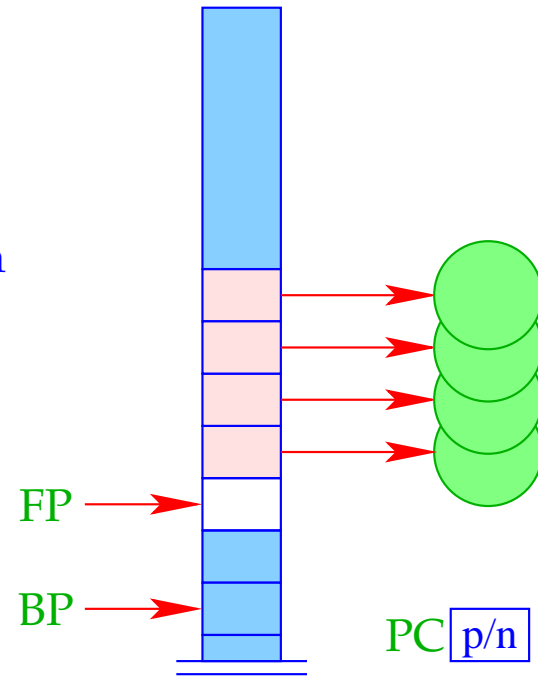
Andernfalls werden die aktuellen Parameter in den Zellen  $S[FP+1]$ ,  $S[FP+2]$ , ...,  $S[FP+h]$  ausgetauscht und `p/h` angesprungen.

```
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  $m = SP - h - FP$  ist gerade gleich der Anzahl der lokalen Variablen im Kellerrahmen.



lastcall p/n m



## Beispiel:

Betrachten wir die Klausel  $r$ , von oben:

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

Die Optimierung letzter Ziele liefert für `codeC 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 **einzig**e in dieser Klausel, können wir auf **lastmark** verzichten und **lastcall p/h m** durch die Folge **move m h; jump p/h** ersetzen.

## Beispiel:

Betrachte die **letzte** Klausel des Prädikats `app`:

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

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

A: pushenv 6			uref 4	bind
putref 1	B: putvar 4		son 2	E: putref 5
ustruct [[]]/2 B	putvar 5		uvar 6	putref 2
son 1	putstruct [[]]/2		up E	putref 6
uvar 4	bind	D: check 4		move 6 3
son 2	C: putref 3	putref 4		jump app/3
uvar 5	ustruct [[]]/2 D	putvar 6		
up C	son 1	putstruct [[]]/2		

## 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:

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

## 35 Verkleinerung von Kellerrahmen

### Idee:

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

### Beispiel:

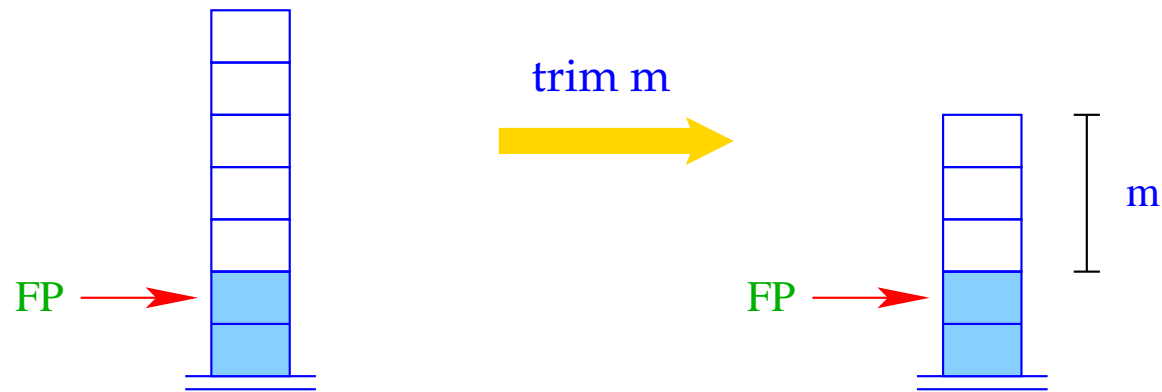
Betrachte die Klausel:

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

Nach der Anfrage  $p_2(\bar{X}_1, X_2)$  ist die Variable  $X_1$  tot.

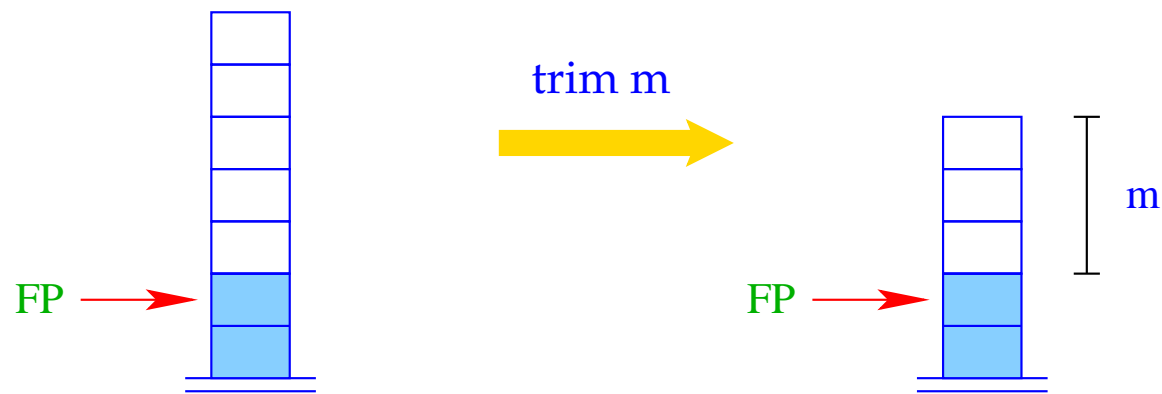
Nach der Anfrage  $p_3(\bar{X}_2, X_3)$  ist die Variable  $X_2$  tot **:-)**

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



```
if (FP ≥ BP)
    SP = FP + m;
```

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



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

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

## Beispiel (Forts.):

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

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 p <sub>2</sub> /2	call p <sub>3</sub> /2	lastcall p <sub>4</sub> /2 3
call p <sub>1</sub> /2	B:	trim 4	C:	trim 3

## 36 Klausel-Indizierung

### Beobachtung:

Oft werden Prädikate durch Fallunterscheidung nach dem ersten Argument definiert.

⇒ Berücksichtigung des ersten Arguments kann viele Alternativen ausschließen :-)

⇒ Fehlschlag wird früher entdeckt :-)

⇒ Rücksetz-Punkte werden früher beseitigt :-))

⇒ Kellerrahmen werden früher gepoppt :-)))

**Beispiel:** Das app-Prädikat:

$$\text{app}(X, Y, Z) \leftarrow X = [], Y = Z$$

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

- 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  $r_1 \dots r_m$  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
      jump A2
      CONS:      jump A2 // Konstruktor [[]]
      DEFAULT:  fail      // Default
```

## 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
      jump A2
      CONS:      jump A2 // Konstruktor [[]]
      DEFAULT:  fail      // Default
```

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

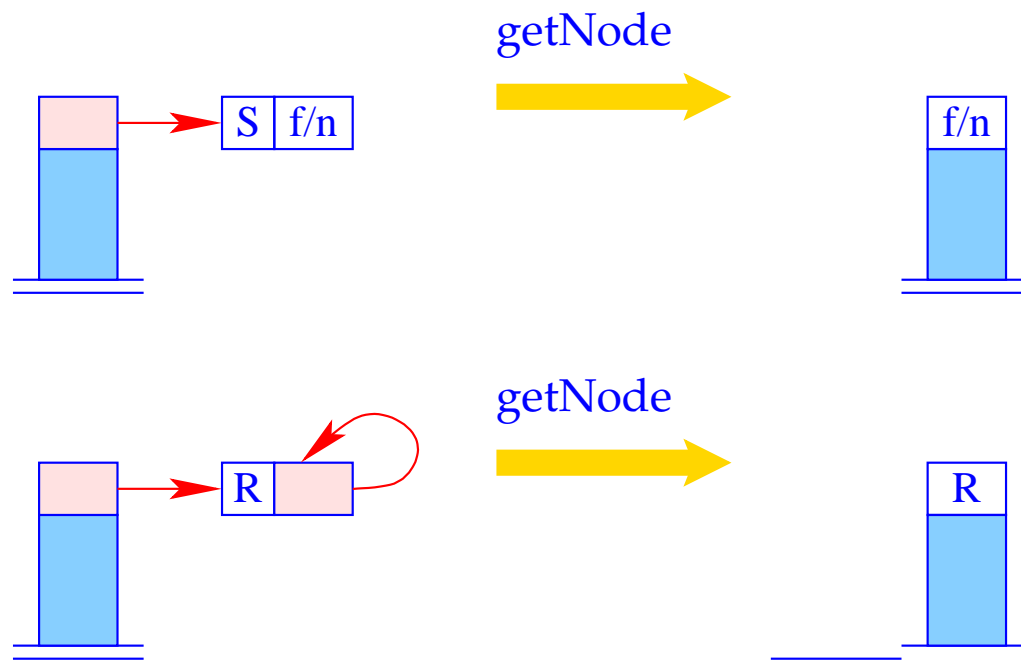
```
fail = backtrack()
```

... löst direkt `backtracking` aus :-)

Dann erzeugen wir für das Prädikat  $p/k$ :

```
codep rr = putref 1
           getNode // extrahiert die Wurzel-Beschriftung
           index p/k // springe zum Try-Block
           tchains rr
A1 : codeC r1
      ...
Am : codeC rm
```

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:

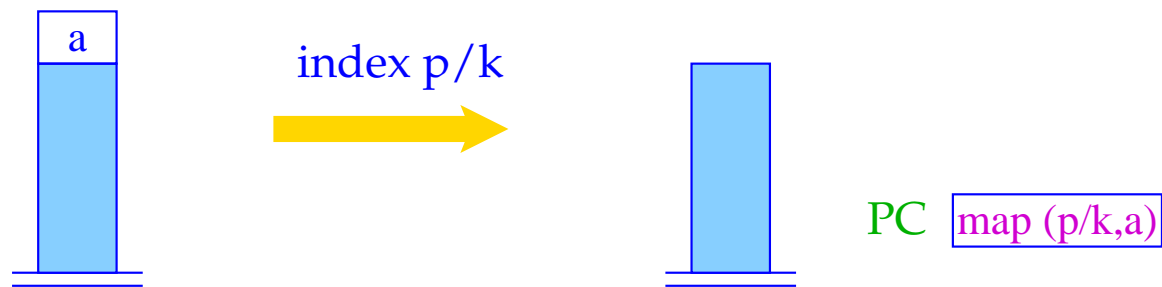


```

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;
}

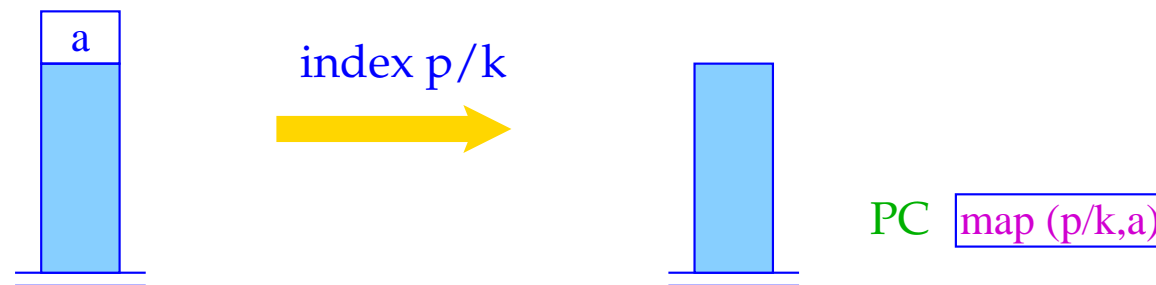
```

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:

$$\begin{aligned} \text{branch}(X, Y) &\leftarrow p(X), !, q_1(X, Y) \\ \text{branch}(X, Y) &\leftarrow q_2(X, Y) \end{aligned}$$

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) lokalen Variablen der Klausel ist.

## Beispiel:

Betrachten wir unser Beispiel:

$\text{branch}(X, Y) \leftarrow p(X), !, q_1(X, Y)$

$\text{branch}(X, Y) \leftarrow q_2(X, Y)$

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 q <sub>1</sub> /2 2		move 2 2
							jump q <sub>2</sub> /2

## Beispiel:

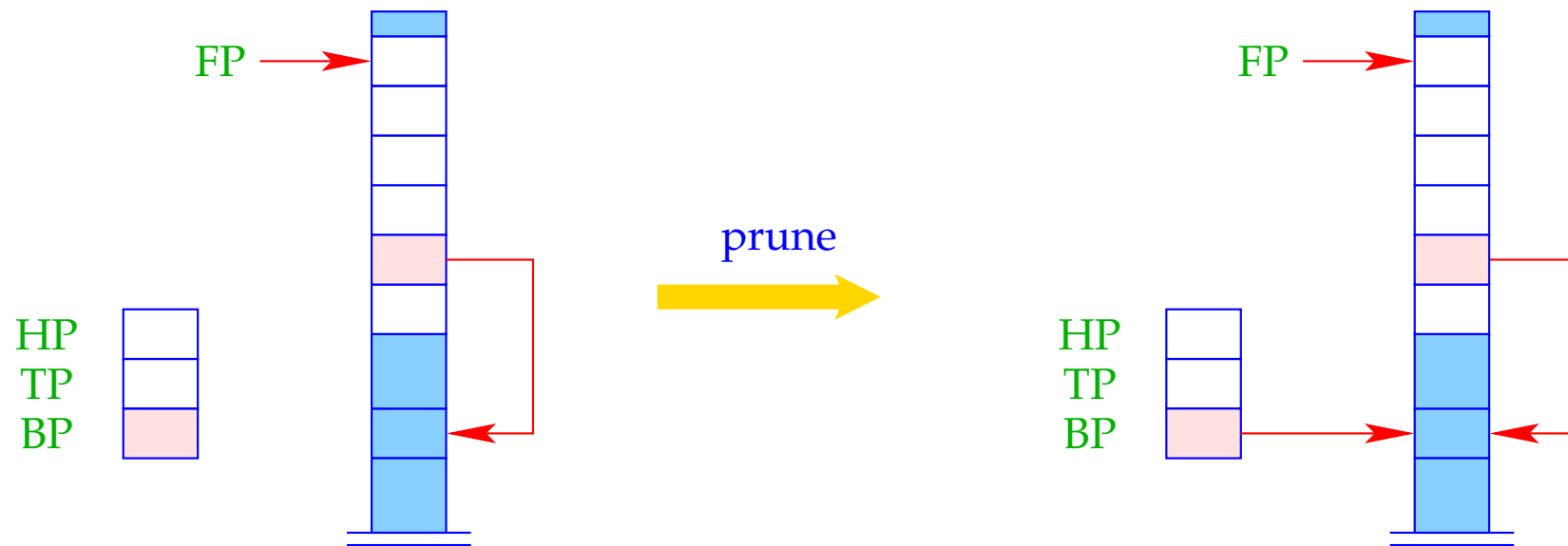
Betrachten wir unser Beispiel:

$$\text{branch}(X, Y) \leftarrow p(X), !, q_1(X, Y)$$
$$\text{branch}(X, Y) \leftarrow q_2(X, Y)$$

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 q <sub>1</sub> /2		move 2 2
							jump q <sub>2</sub> /2

Die neue Instruktion `prune` restauriert einfach den `BP`:



$BP = BP_{old};$

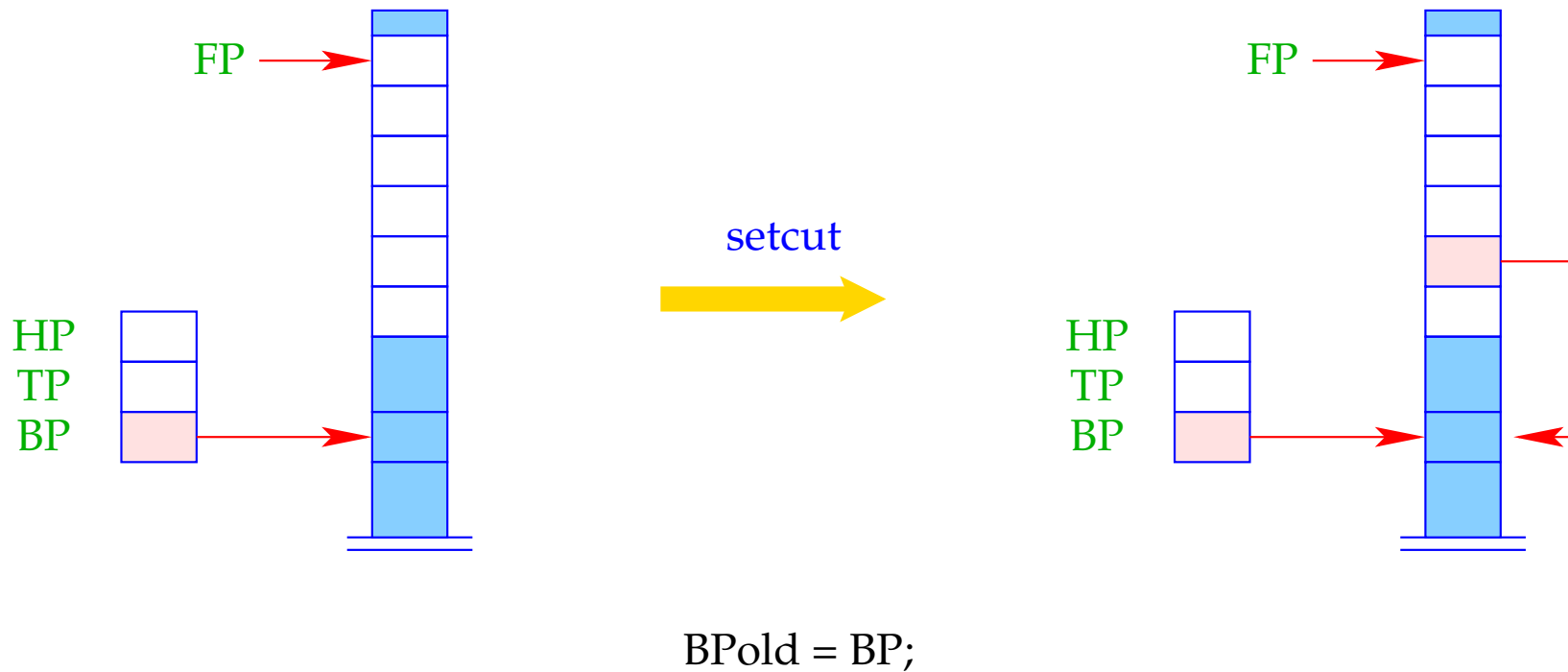
## Problem:

Ist eine Klausel **einzel**, 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**:



## Das allerletzte Beispiel:

## Negierung durch Fehlschlag

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

```
notP(X) ← p(X), !, fail
notP(X) ←
```

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
```