

Diskussion:

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

Idee 2:

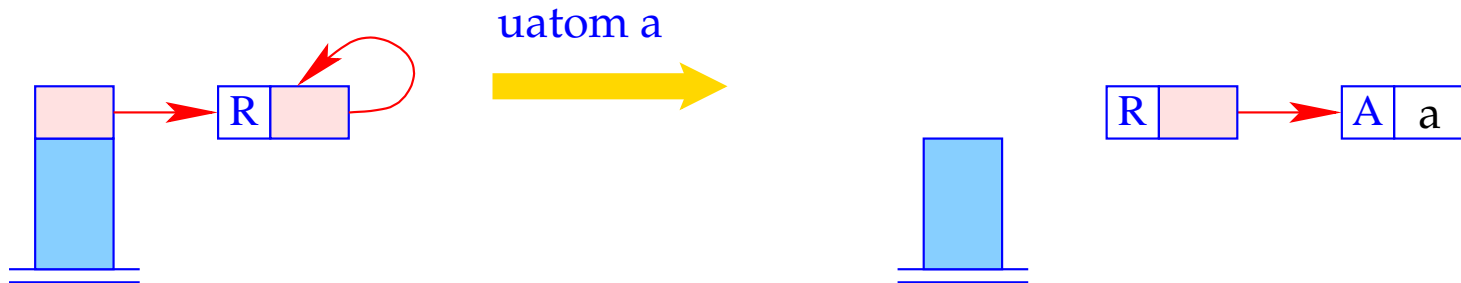
- Lege einen Verweis auf die aktuelle Bindung von X oben auf den Keller.
- Vermeide die Konstruktion von Termen soweit möglich!
- Übersetze t in eine Instruktions-Folge, die die Unifikation mit t implementiert !!!

$$\text{code}_G (\tilde{X} = t) \rho = \text{put } \tilde{X} \rho \\ \text{code}_U t \rho$$

Betrachten wir zuerst Unifikation nur für Atome und Variablen:

```
codeU a ρ = uatom a  
codeU X ρ = uvar (ρ X)  
codeU _ ρ = pop  
codeU X̄ ρ = uref (ρ X)  
... // wird fortgesetzt :-)
```

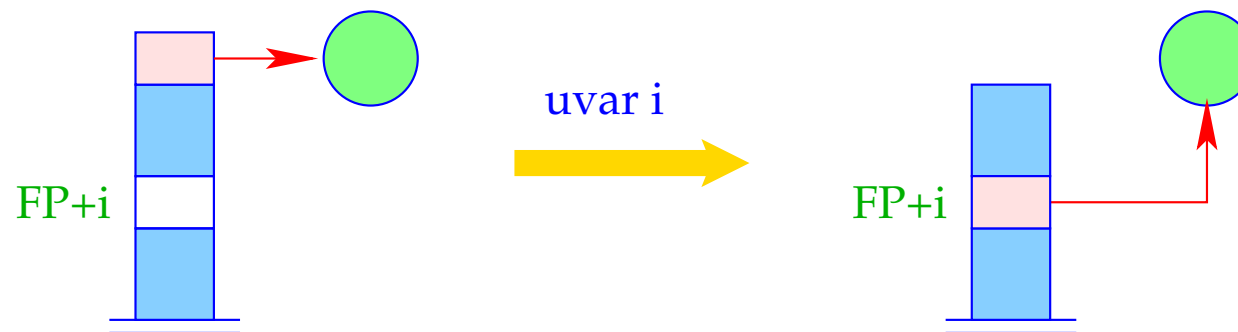
Die Instruktion `uatom a` implementiert die Unifikation mit dem Atom `a`:



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

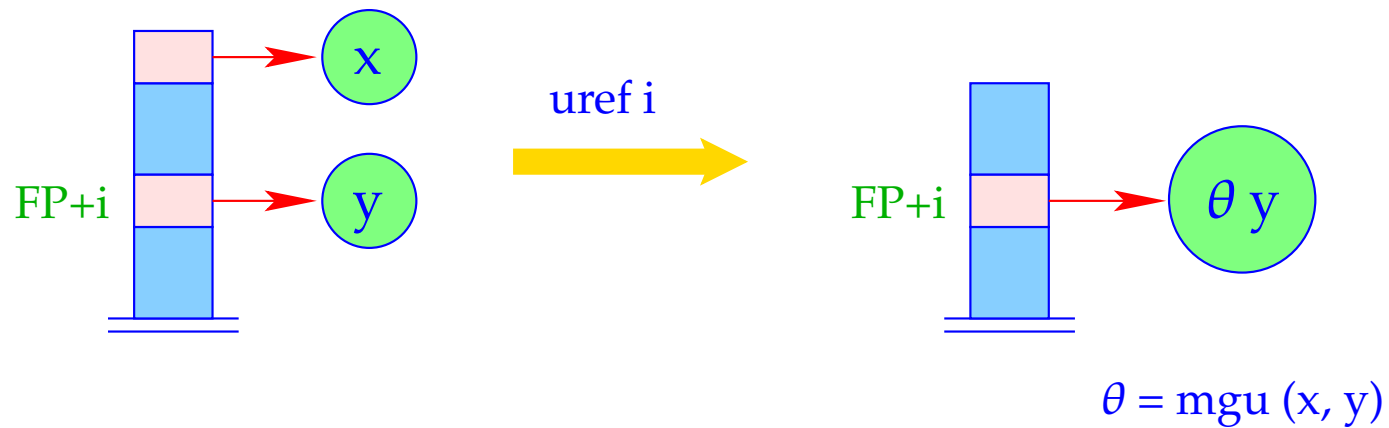
- Der Aufruf von `trail()` **vermerkt** eine potentiell neue Bindung.
- Der Aufruf von `backtrack()` **initiiert backtracking**.

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.

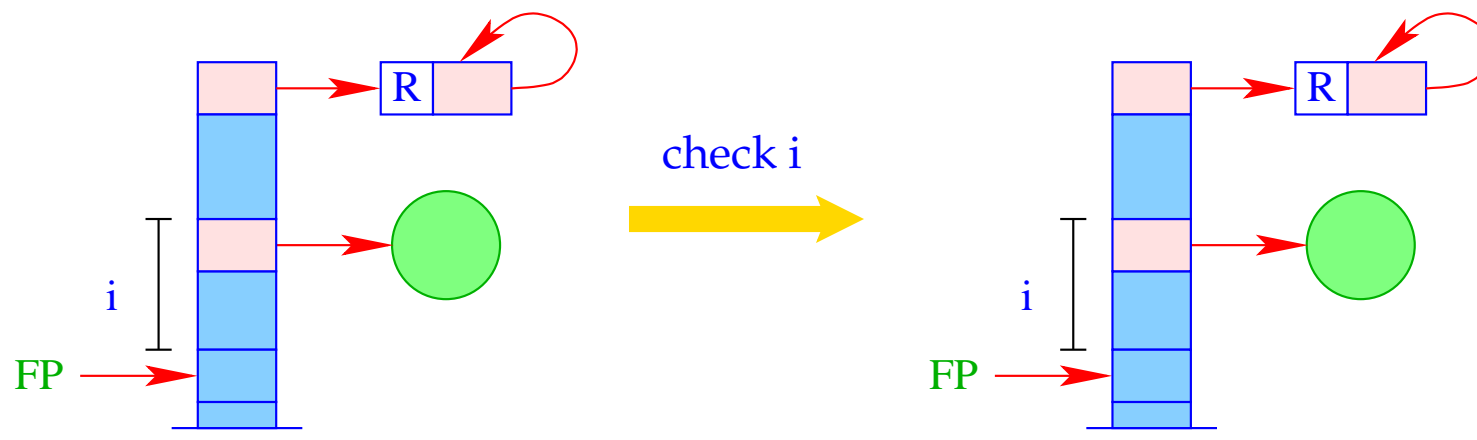
⇒ $ivars(t')$ liefert die Menge der bereits initialisierten Variablen in t .

⇒ Das Macro `check` $\{Y_1, \dots, Y_d\} \rho$ erzeugt die notwendigen Tests der Variablen Y_1, \dots, Y_d :

$$\begin{aligned} \text{check } \{Y_1, \dots, Y_d\} \rho &= \text{check } (\rho Y_1) \\ &\quad \text{check } (\rho Y_2) \\ &\quad \dots \\ &\quad \text{check } (\rho Y_d) \end{aligned}$$

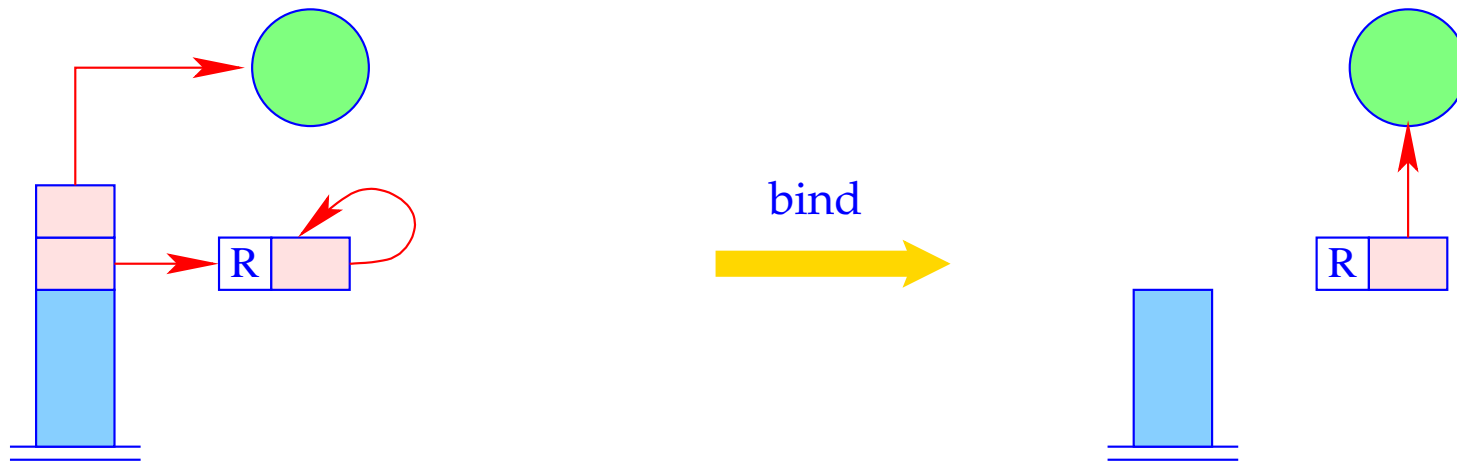
Die Instruktion `check i` überprüft, ob die (ungebundene) Variable oben auf dem Keller innerhalb des Term vorkommt, an den die Variable `i` gebunden ist.

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:

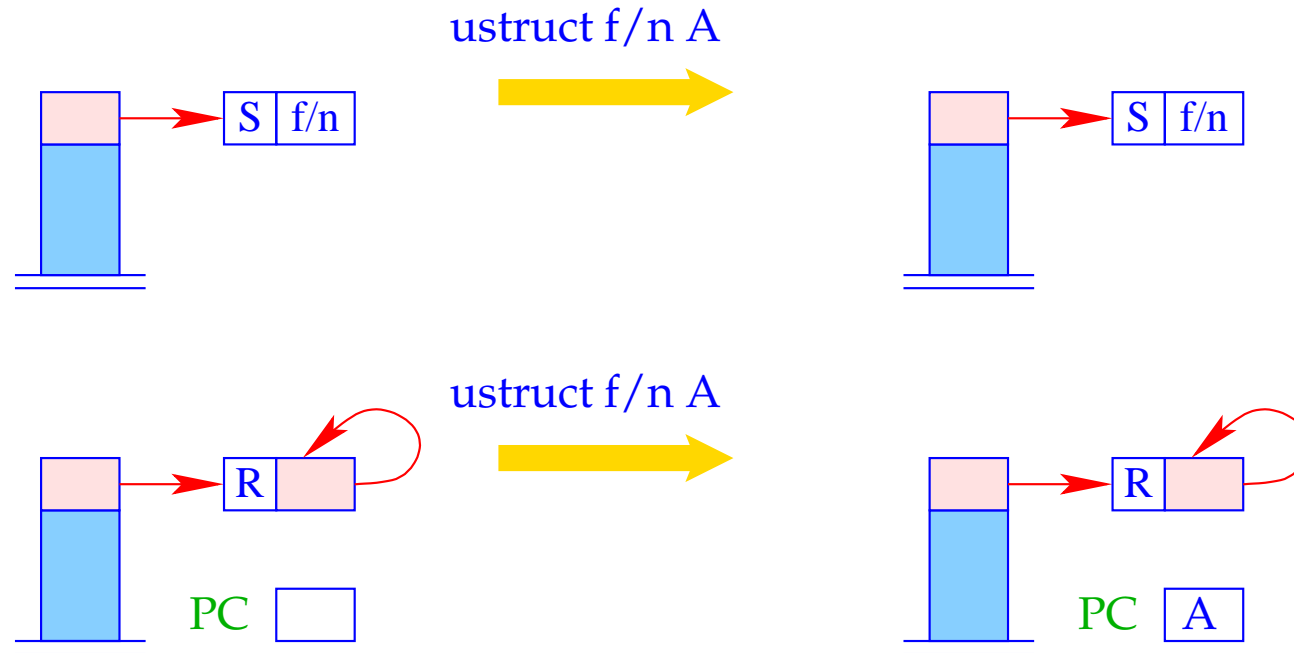


```
H[SP-1] = (R, S[SP]);  
trail (S[SP-1]);  
SP = SP - 2;
```

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:

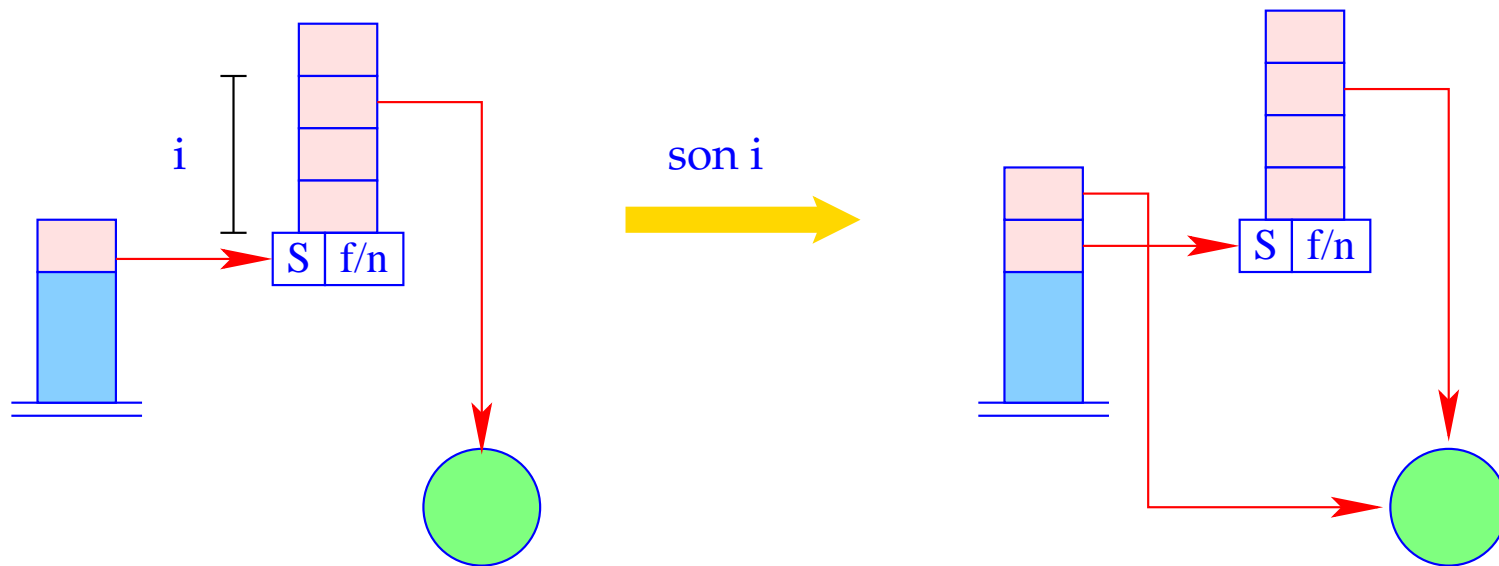
Der Befehl `ustruct i` implementiert den Test des Wurzel-Knotens der Struktur:



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

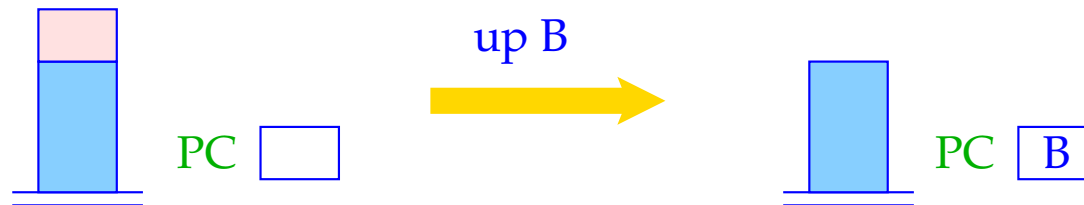
... die Argument-Referenz wurde **noch nicht** gepoppt :-)

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



$S[SP+1] = \text{deref}(H[S[SP]+i]); SP++;$

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 $p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n$.

Sei $\{X_1, \dots, X_m\}$ 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** :-)

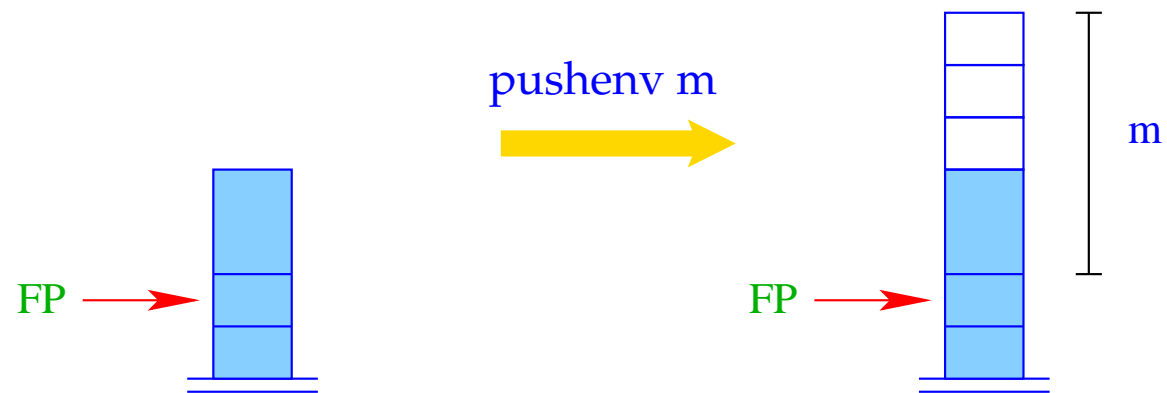
Dann übersetzen wir:

```
codeC r = pushenv m           //reserviert Platz für lokale Vars.  
          codeG g1 ρ  
          ...  
          codeG gn ρ  
          popenv
```

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 :

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

Dann liefert `codeC r` :

pushenv 3

mark A

A: mark B

B: popenv

putref 1

putref 3

putvar 3

putref 2

call f/2

call a/2

32 Prädikate

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

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

Insbesondere haben wir für $f = 1$:

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

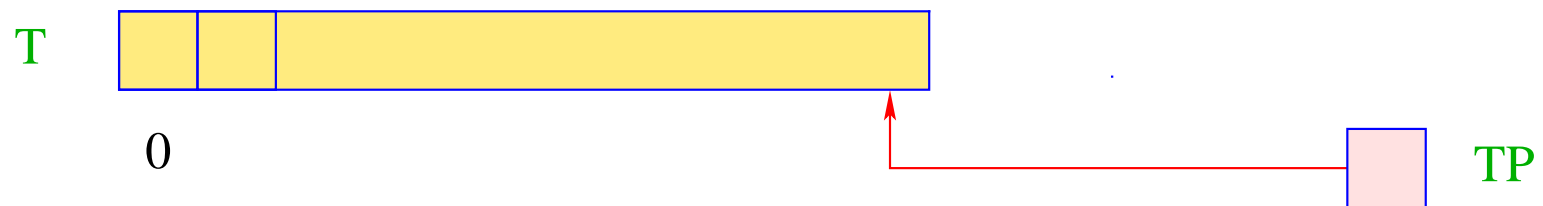
Falls q/k durch mehrere Klauseln definiert ist, muss die erste Alternative ausprobiert werden.

Bei Fehlschlag wird die nächste Alternative probiert ...

\implies backtracking :-)

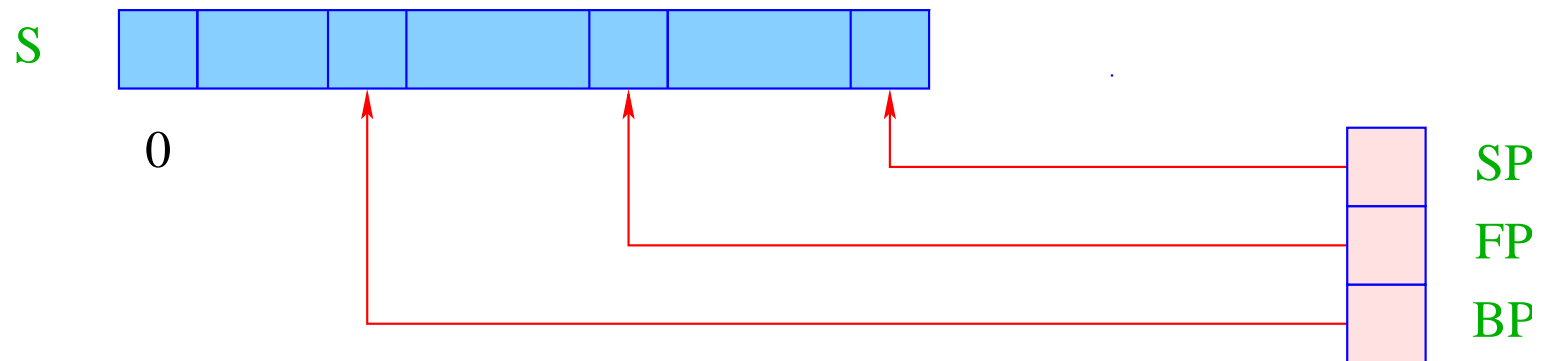
32.1 Backtracking

- 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 \implies 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 == 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:

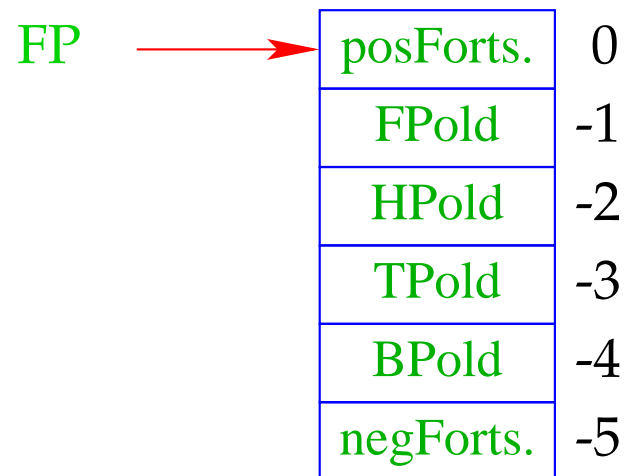


Beachte: Ein neuer **BP** erhält den Wert des aktuellen **FP** :-)

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:



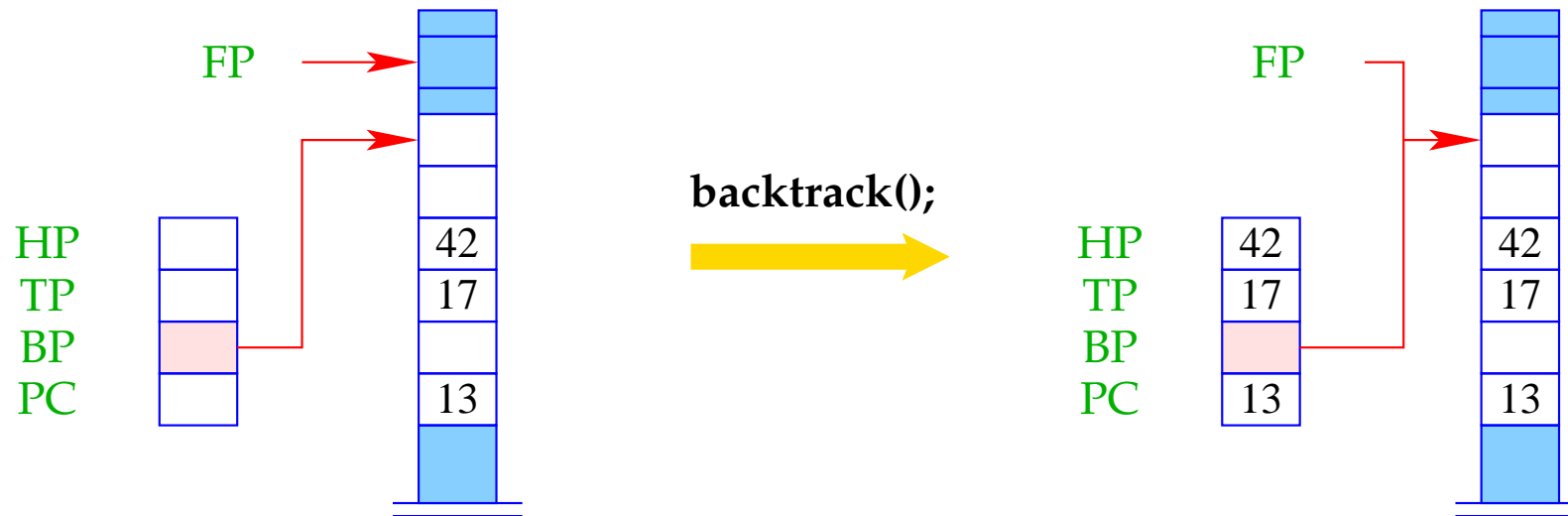
Zur besseren Lesbarkeit führen wir die folgenden nützlichen Makros ein:

`posCont` \equiv $S[\text{FP}]$
`FPold` \equiv $S[\text{FP} - 1]$
`HPold` \equiv $S[\text{FP} - 2]$
`TPold` \equiv $S[\text{FP} - 3]$
`BPold` \equiv $S[\text{FP} - 4]$
`negCont` \equiv $S[\text{FP} - 5]$

Bemerkung:

Vorkommen auf der [linken Seite](#) \equiv retten der Register
Vorkommen auf der [rechten Seite](#) \equiv 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 $rr \equiv r_1, \dots, r_f$ ($f > 1$) definiert.

Wir benötigen code, um

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

Das bedeutet:

```

codeP rr = q/k : setbtp
                try A1
                ...
                try Af-1
                delbtp
                jump Af
A1 : codeC r1
    ...
Af : codeC rf

```

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 **:-))**

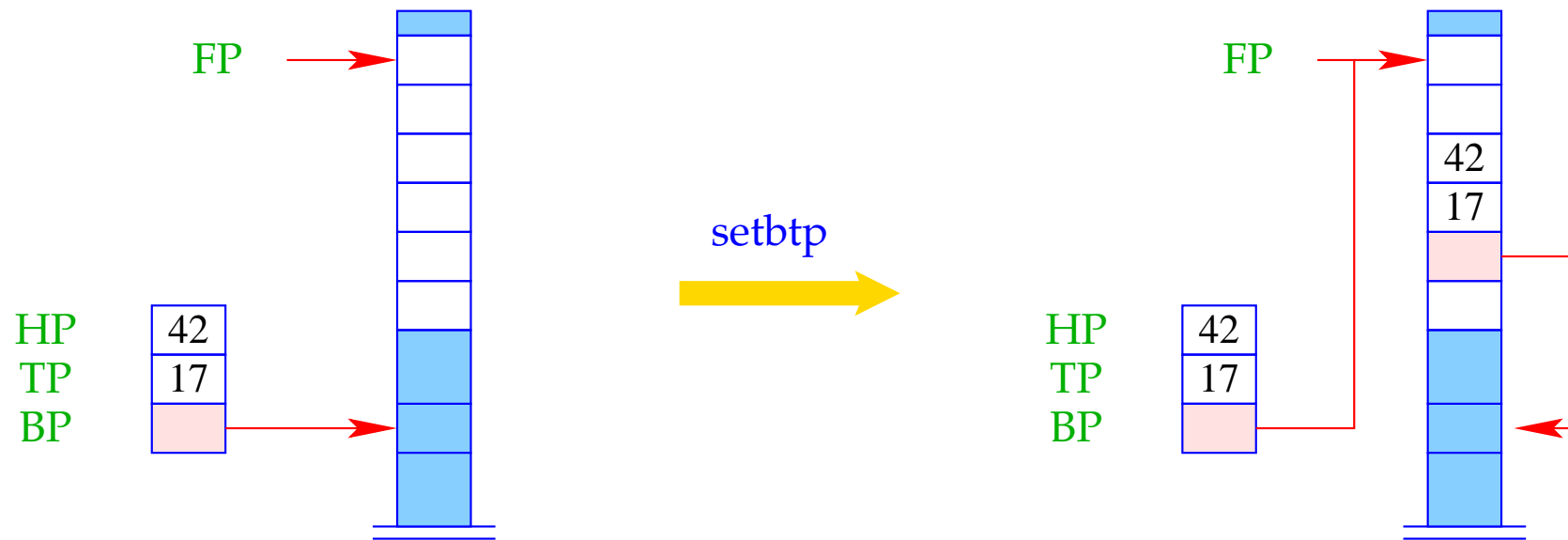
Beispiel:

$$\begin{aligned} s(X) &\leftarrow t(\bar{X}) \\ s(X) &\leftarrow \bar{X} = a \end{aligned}$$

Die Übersetzung des Prädikats s liefert:

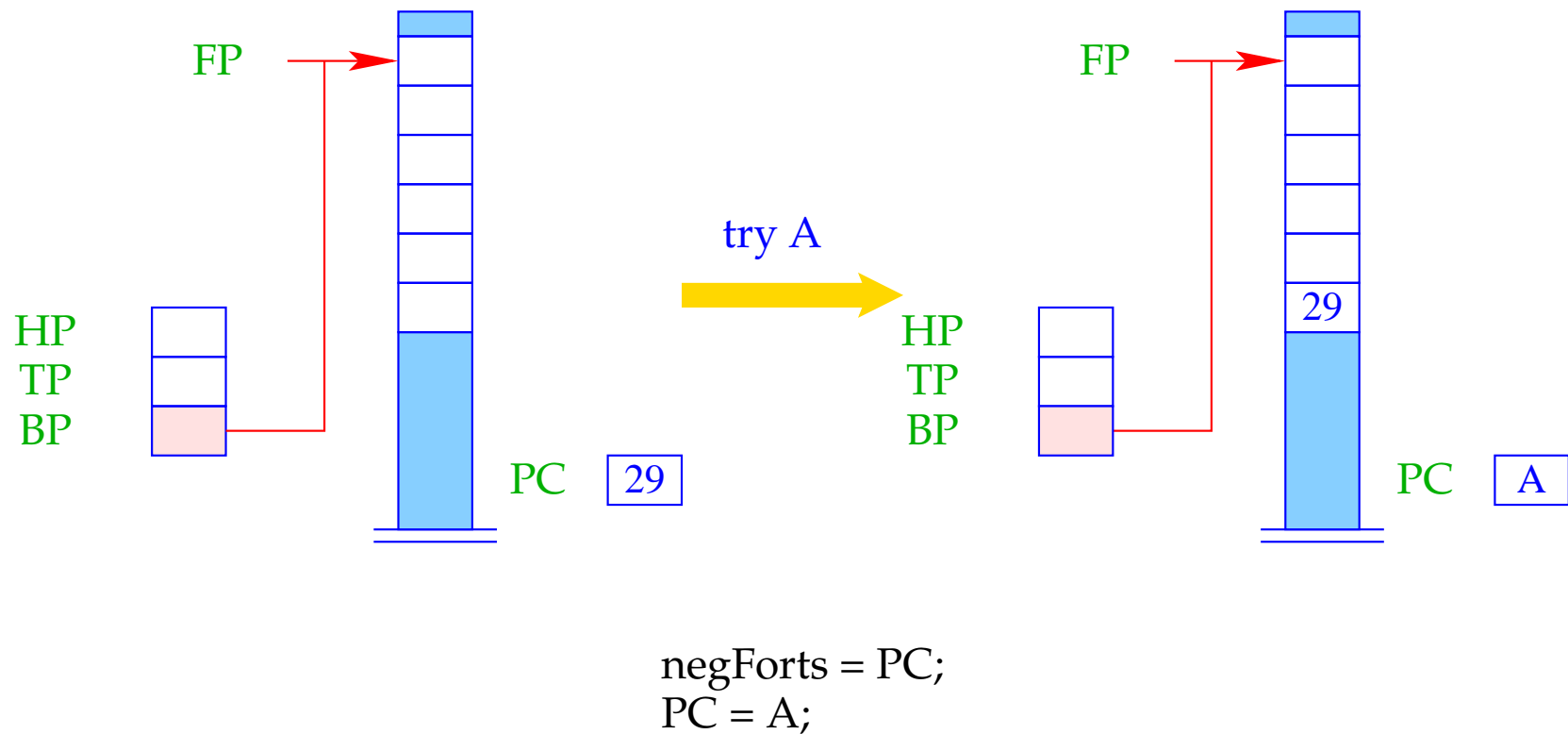
$s/1:$	setbtp	A:	pushenv 1	B:	pushenv 1
	try A		mark C		putref 1
	delbtp		putref 1		uatom a
	jump B		call t/1		popenv
		C:	popenv		

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

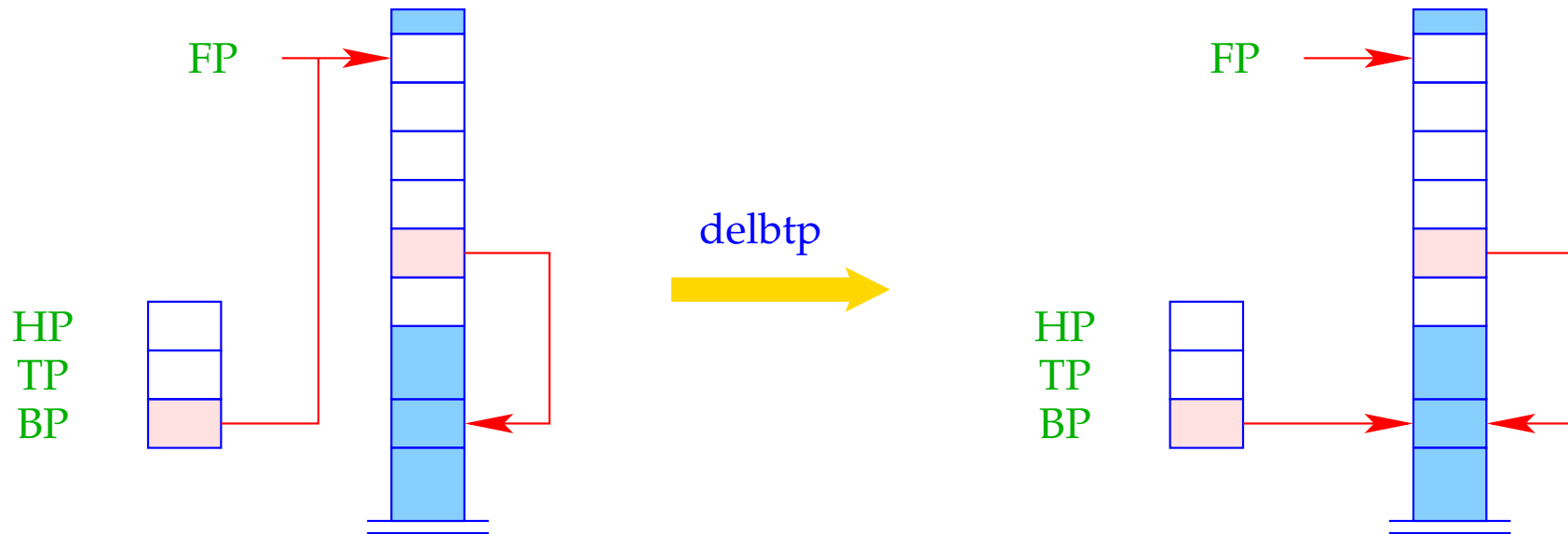


HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;

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



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:

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

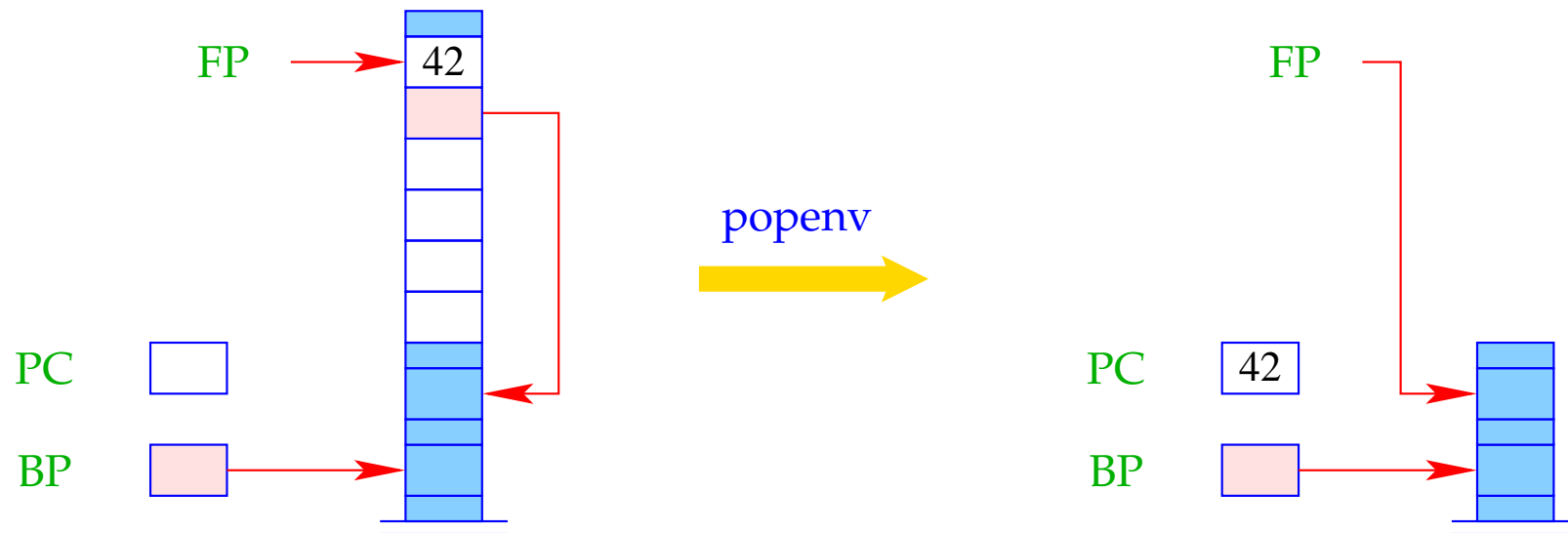
Der aktuelle Kellerrahmen kann aufgegeben werden ...

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

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

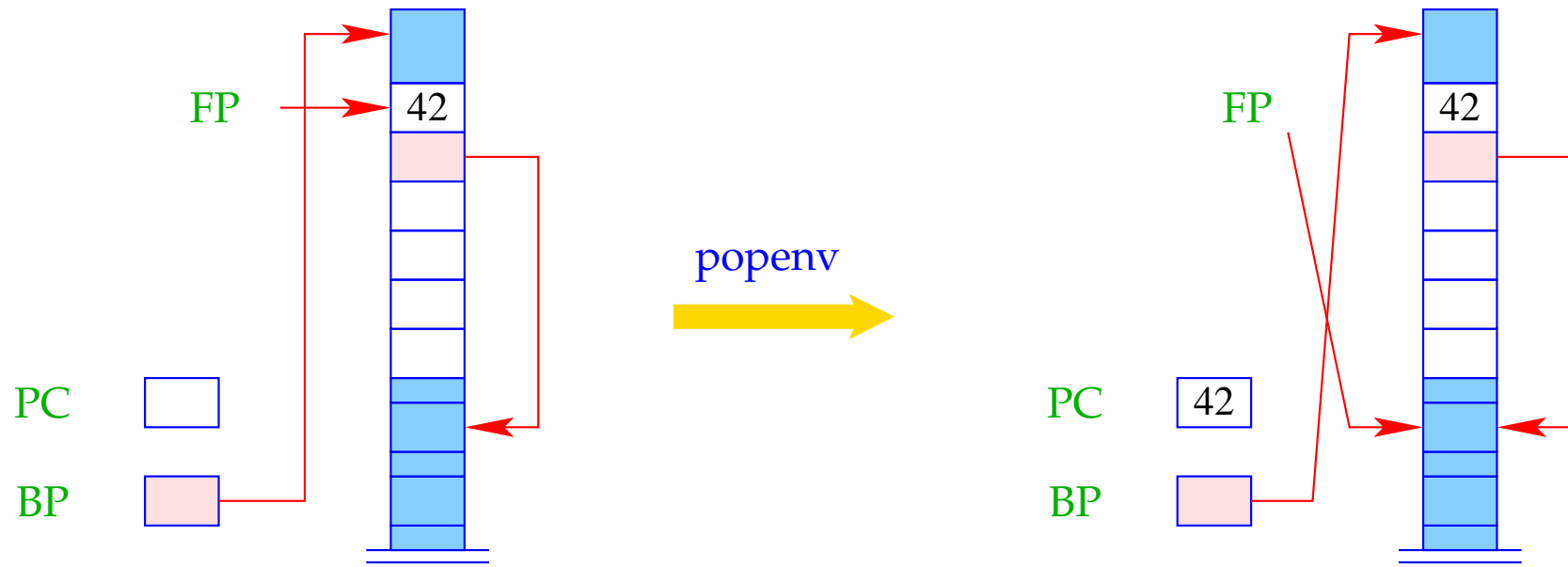
\implies **FP > BP**

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 :-))