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:

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

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) localen Variablen der Klausel ist.

Beispiel:

Betrachten wir unser Beispiel:

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

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:

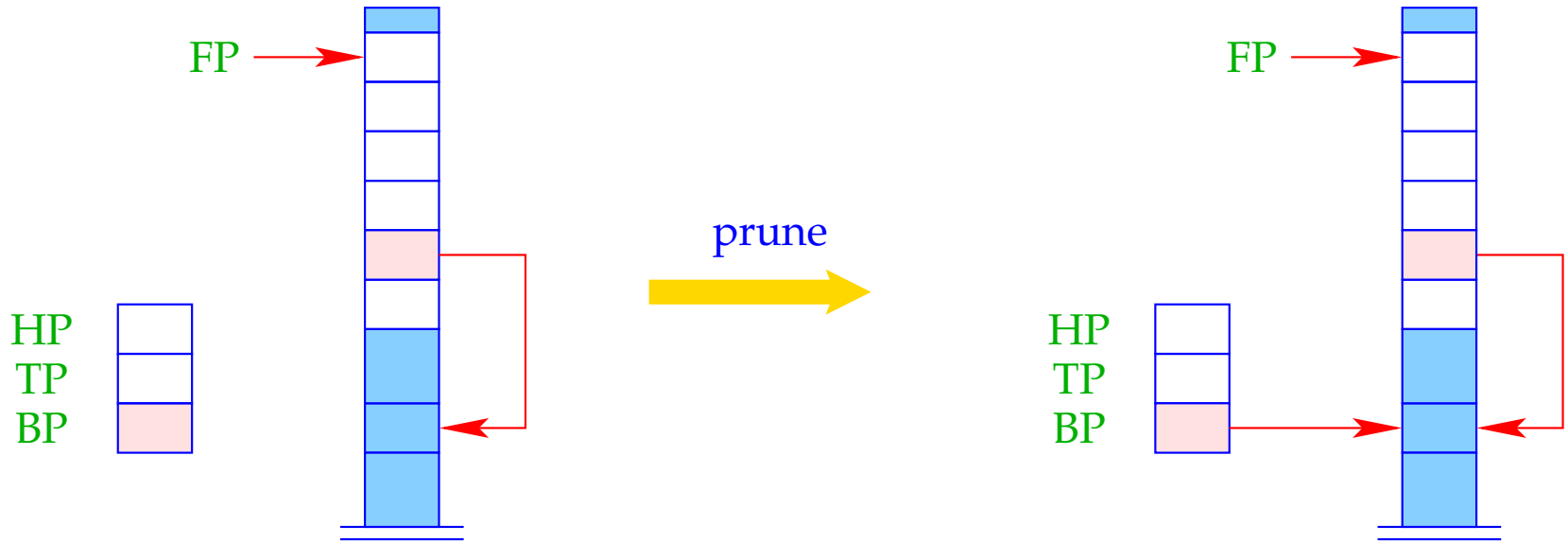
Betrachten wir unser Beispiel:

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

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;

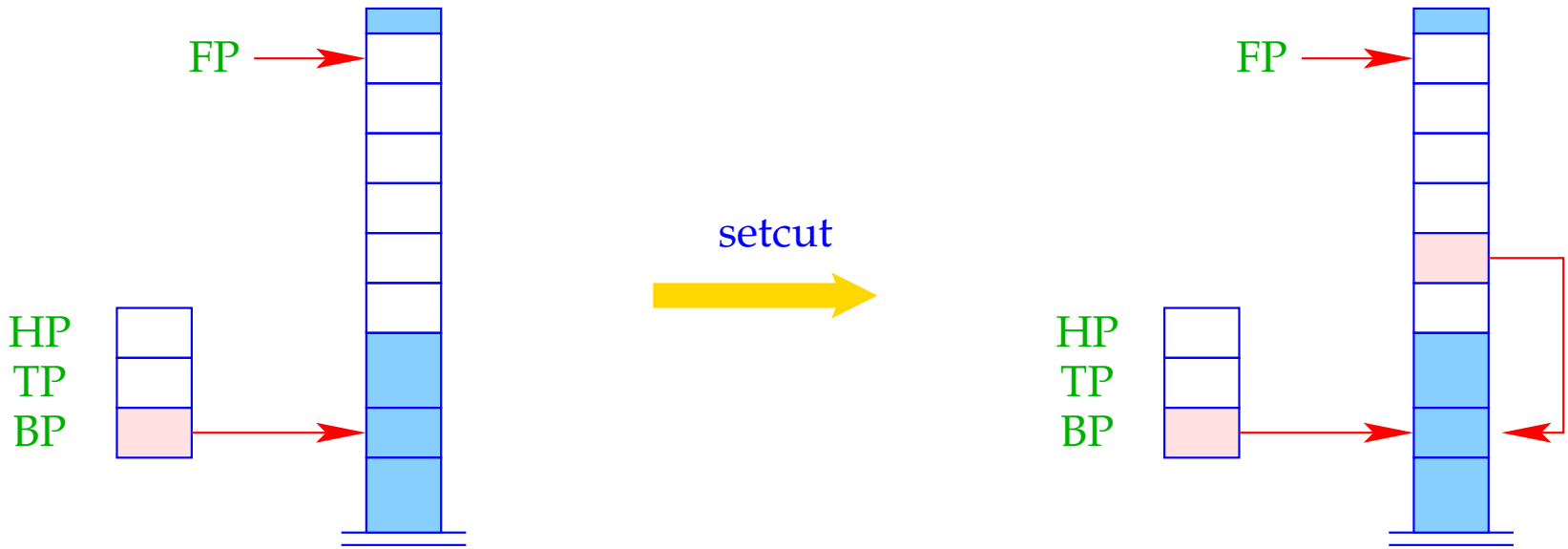
Problem:

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

$\Longrightarrow$

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 :-)

$$\begin{aligned} \mathsf{notP}(X) &\leftarrow \mathsf{p}(X), !, \mathsf{fail} \\ \mathsf{notP}(X) &\leftarrow \end{aligned}$$

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 !!!

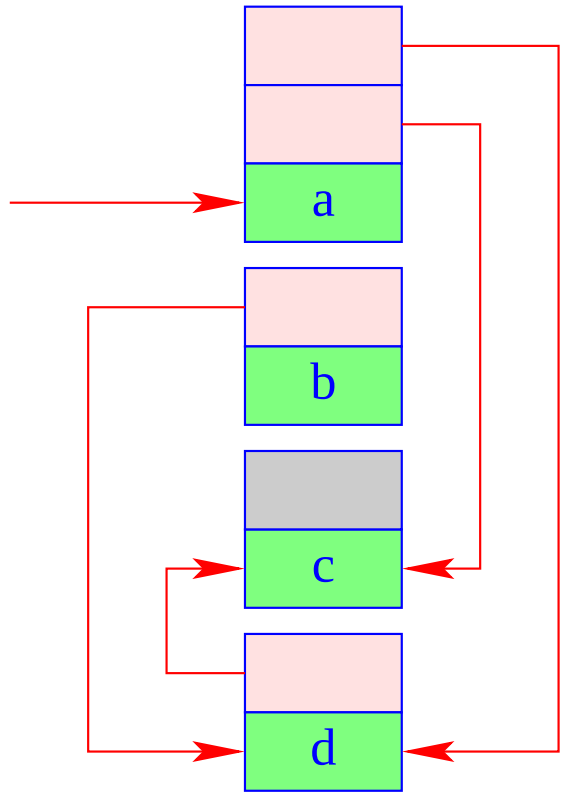
Vorgehen eines kopierenden Kollektors:

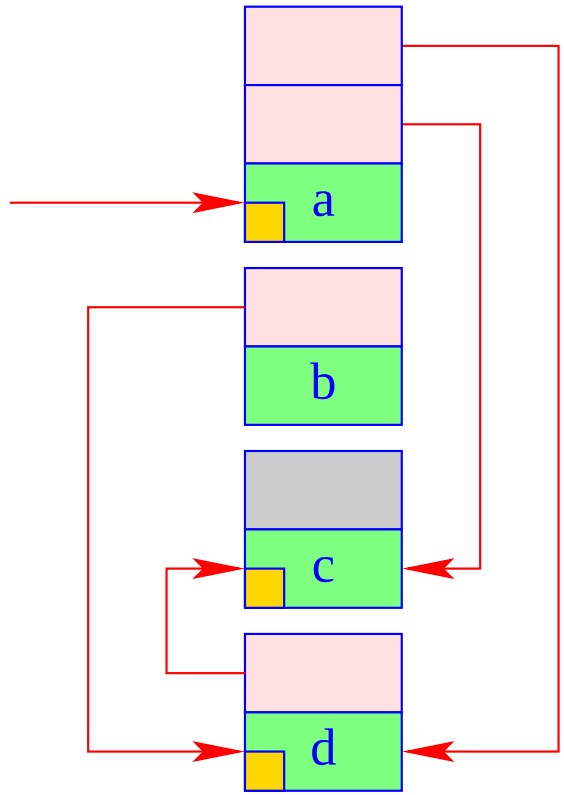
(1) Auffinden der noch lebendigen Objekte:

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

$\Longrightarrow$

Graph-Erreichbarkeit.


a
b
c
d

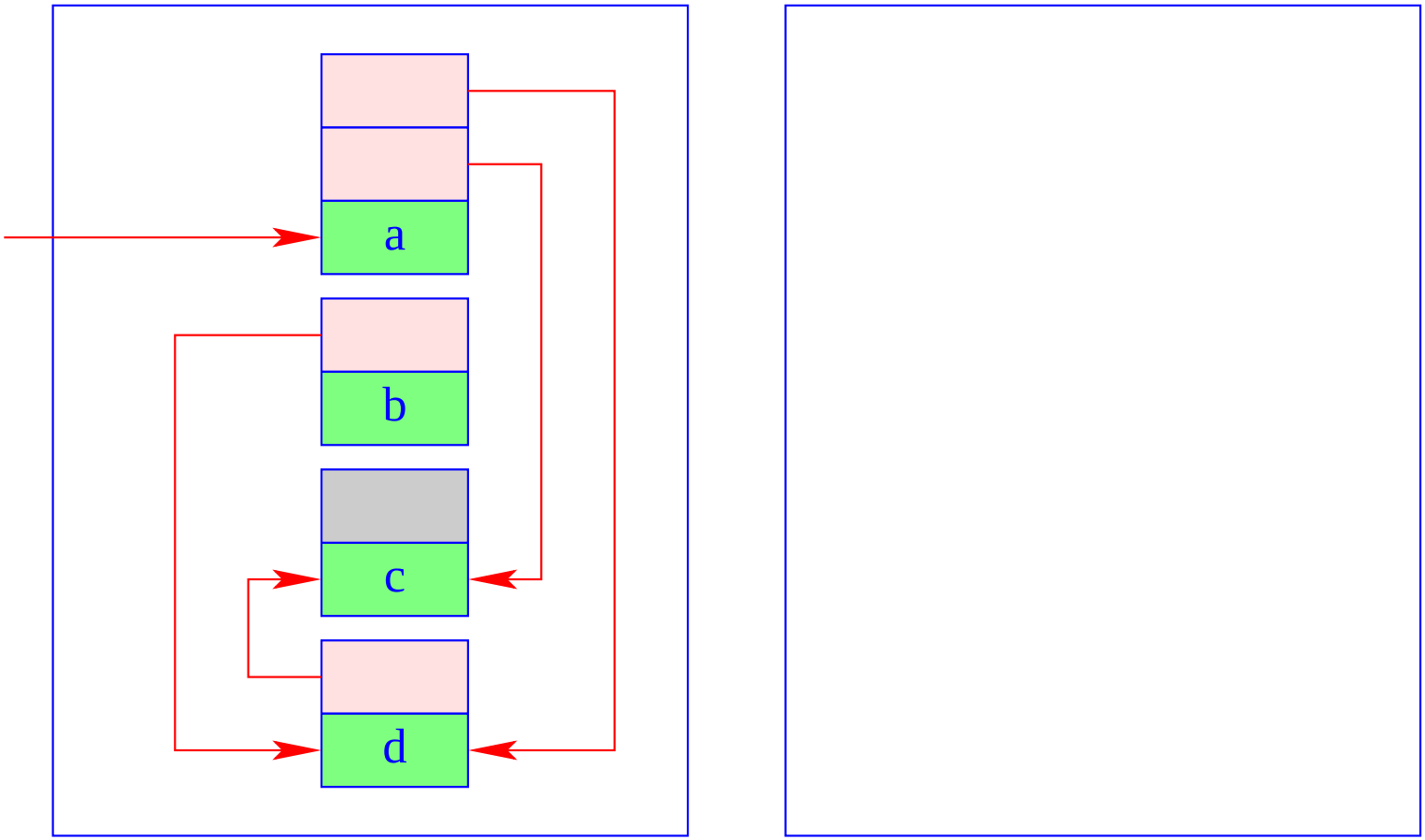

a
b
c
d

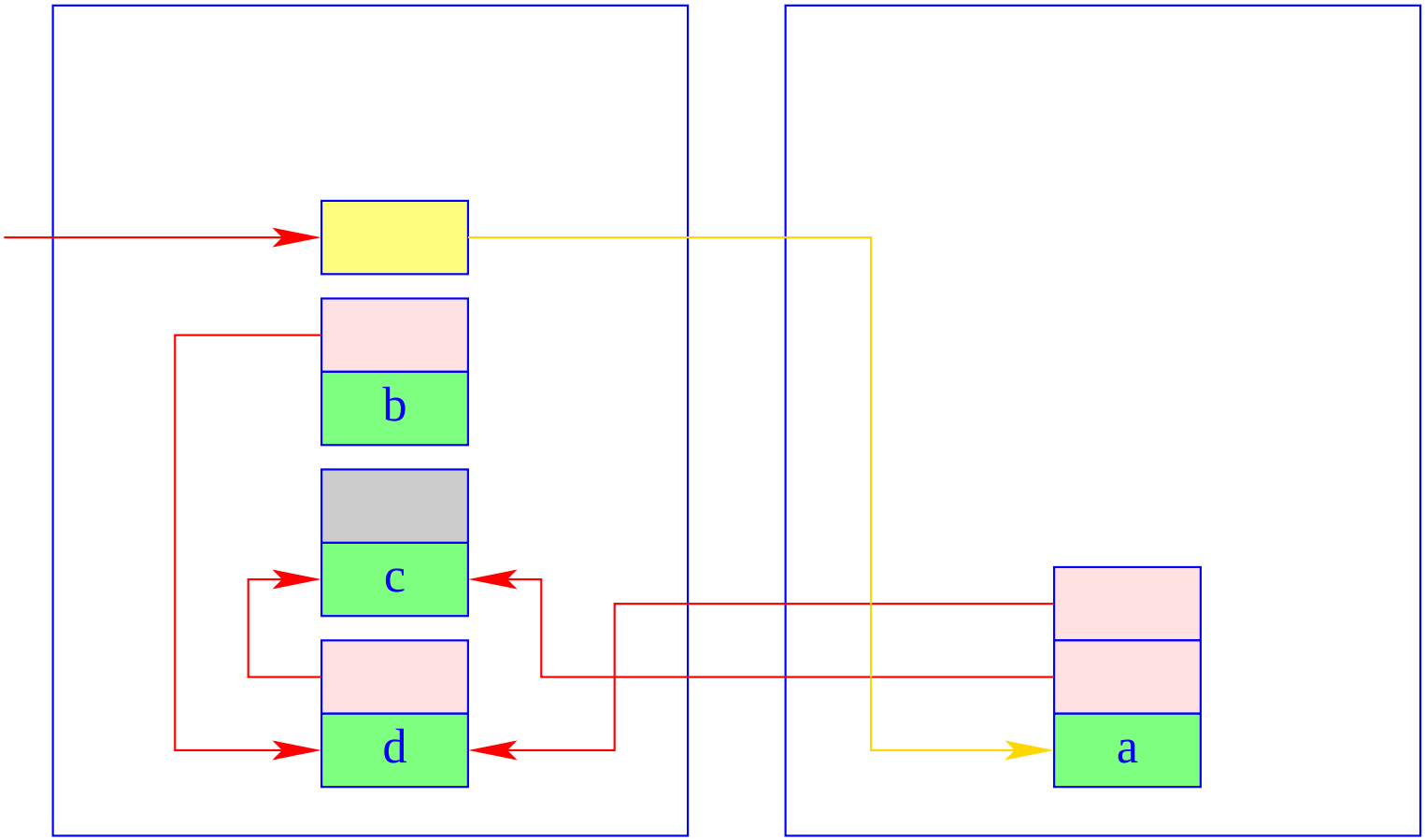
(2) Kopieren der lebendigen Objekte aus dem alten Speicherbereich From-Space in einen neuen BereichTo-Space. Das heißt für jedes aufgefundene Objekt:

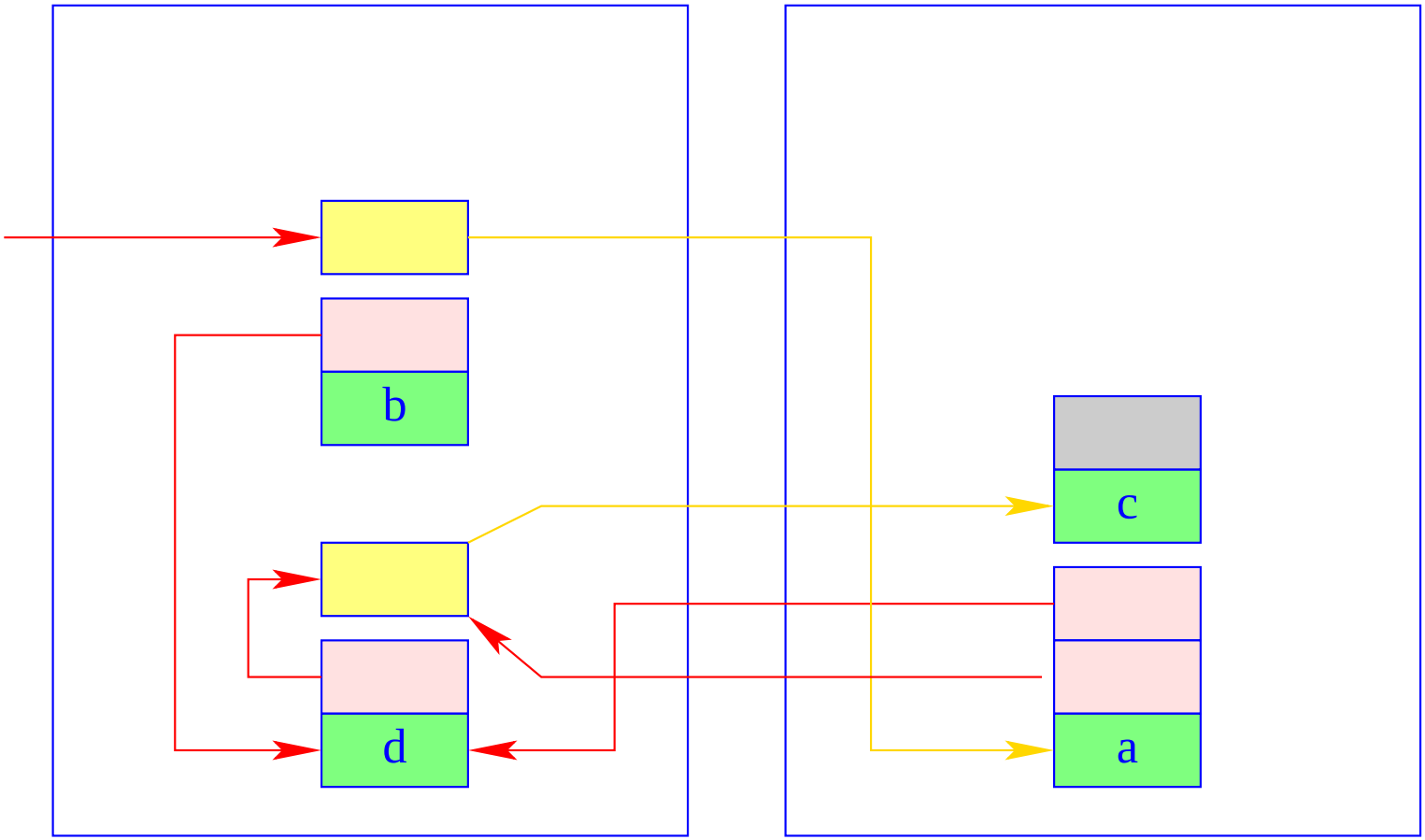
- Kopieren des Objekts;
- Vermerk des neuen Platzes an der alten Stelle (Forwärts-Referenz).

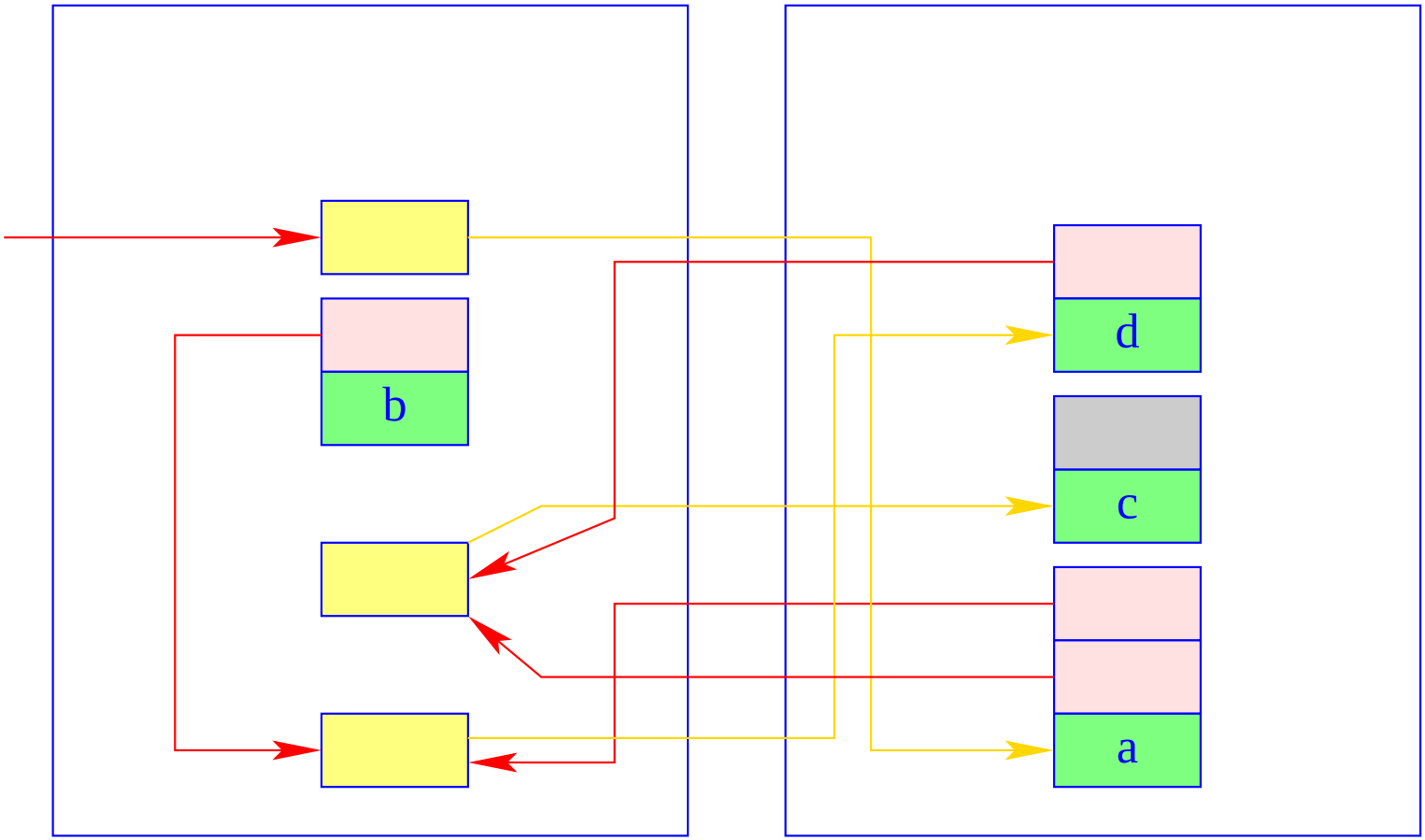
$\Longrightarrow$

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

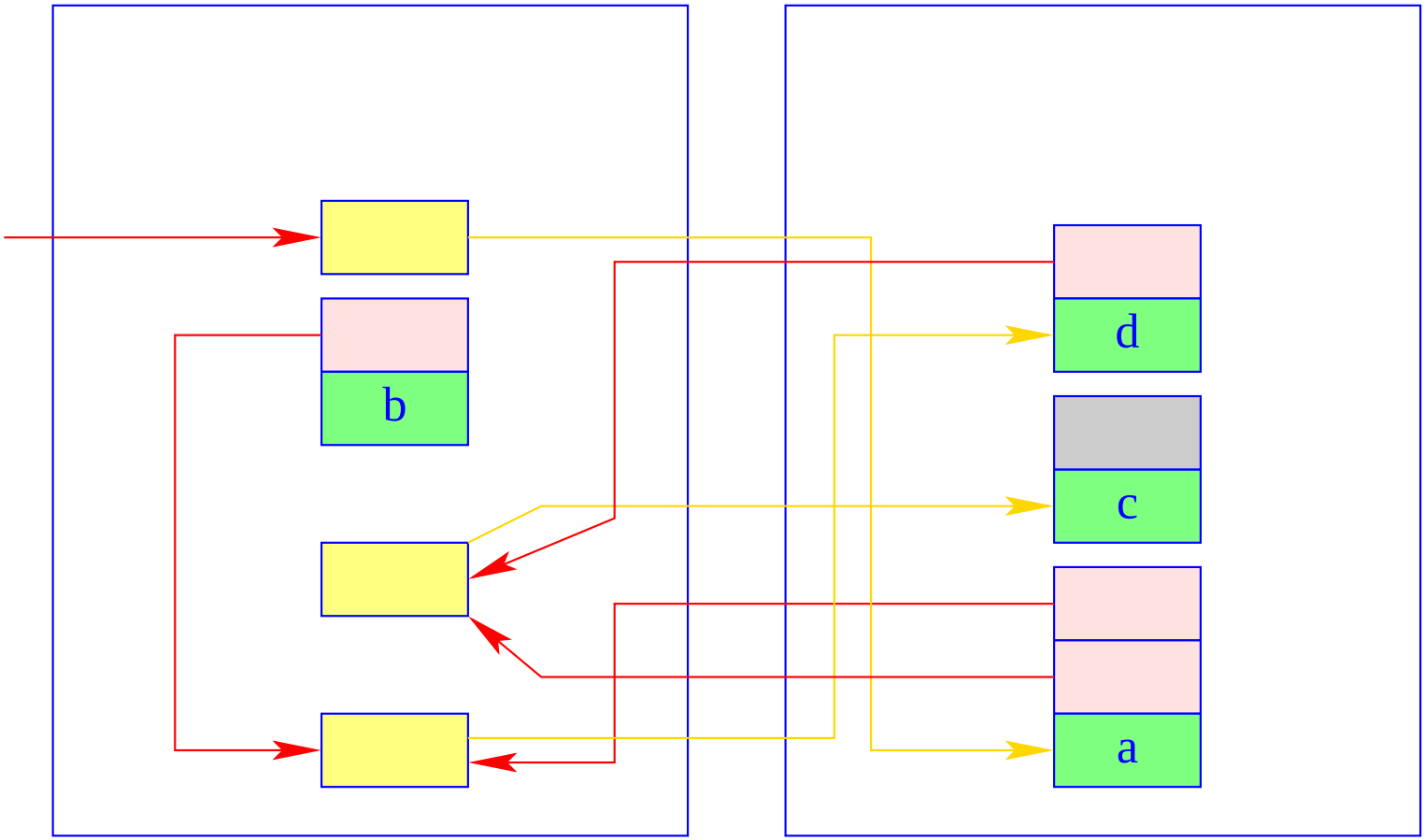

a
b
c
d

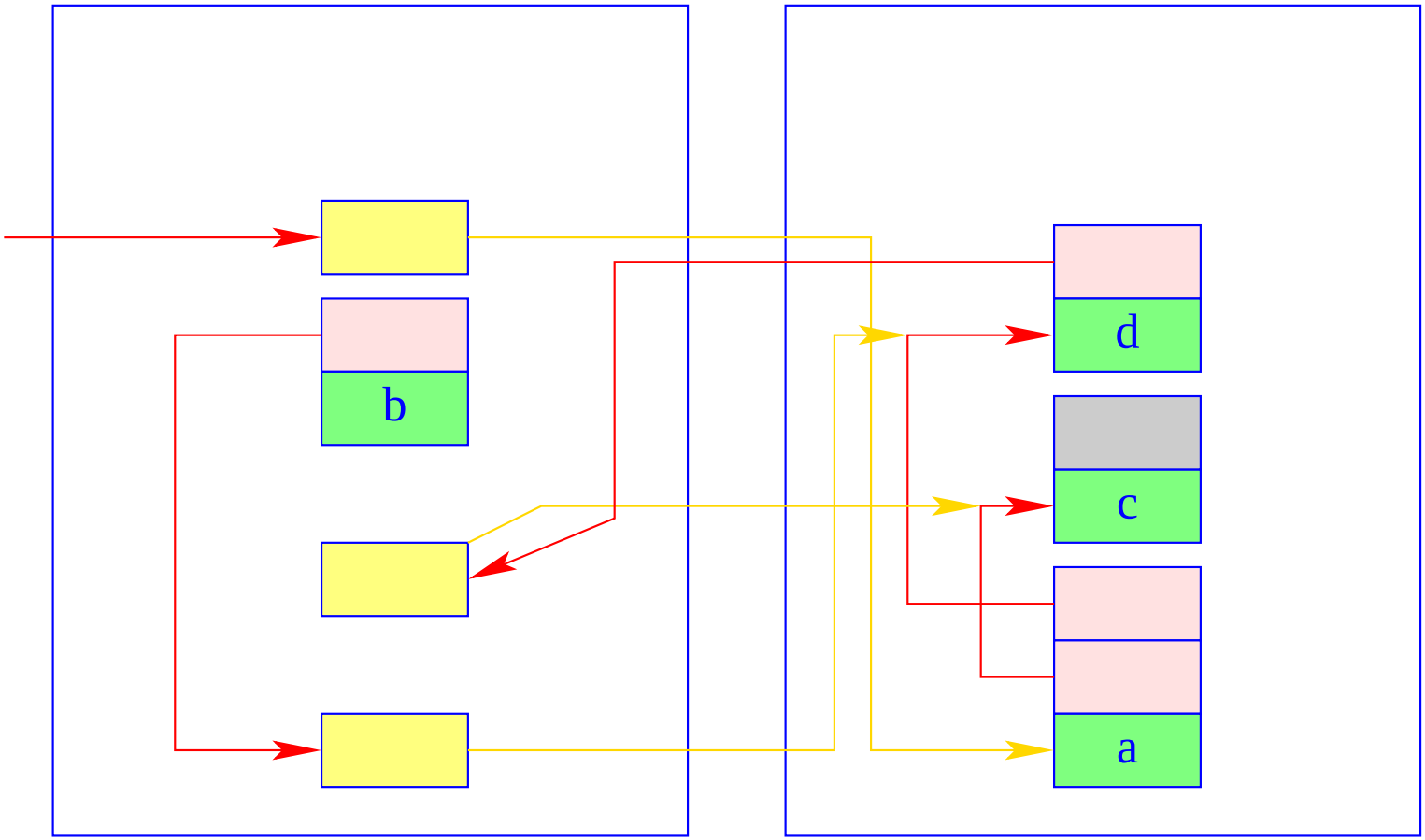

b
c
d
a

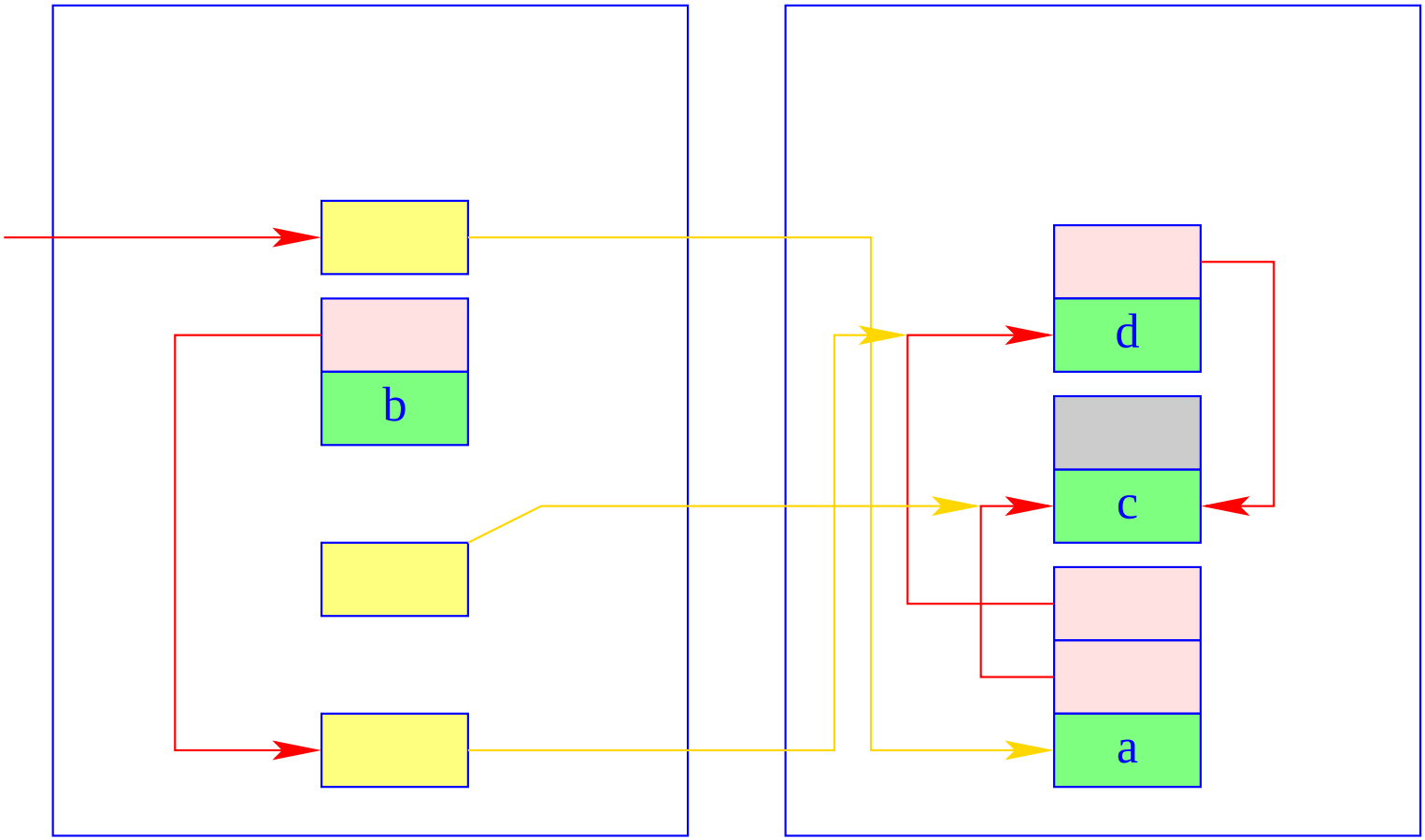

b
c
d
a

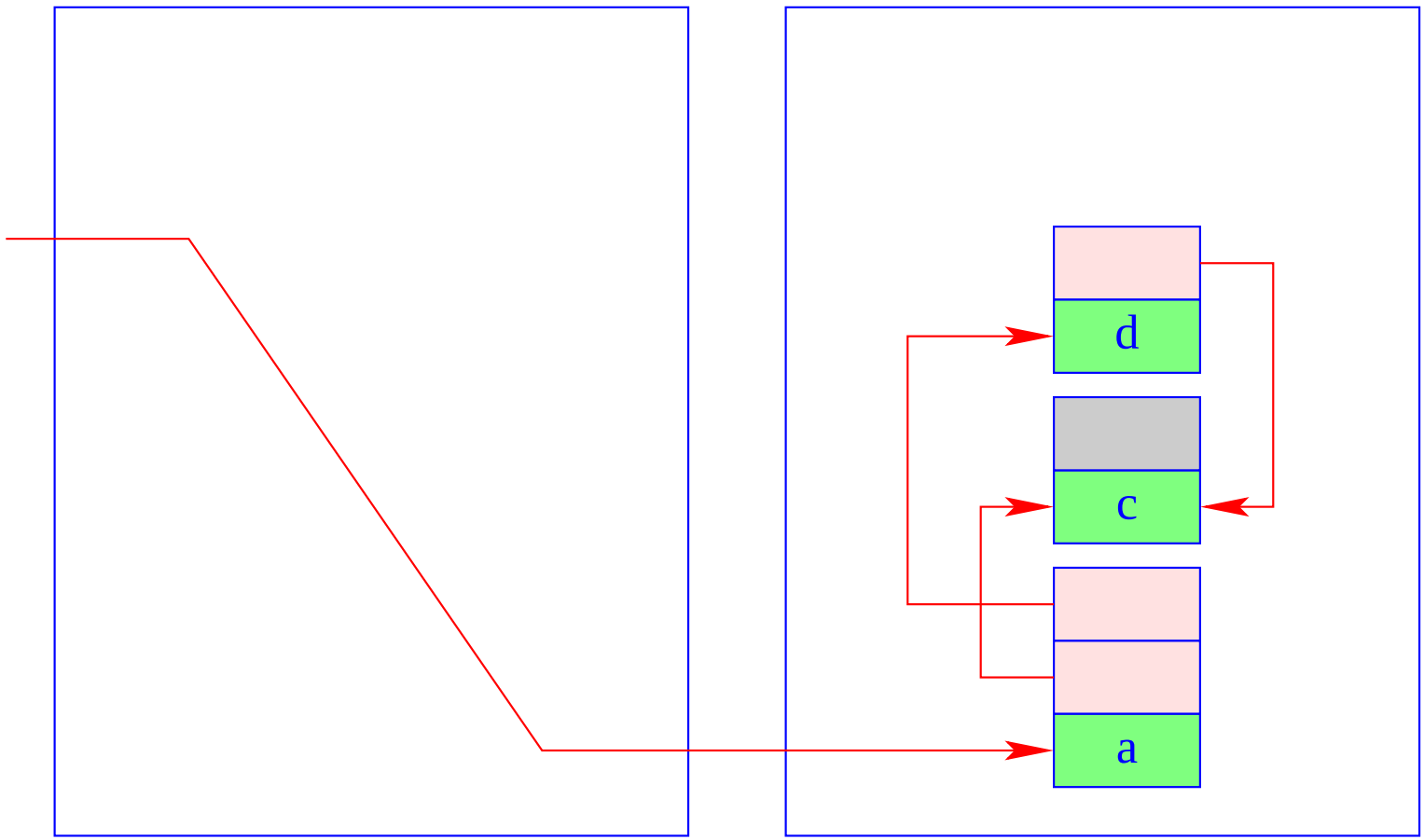

b
d
c
a

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

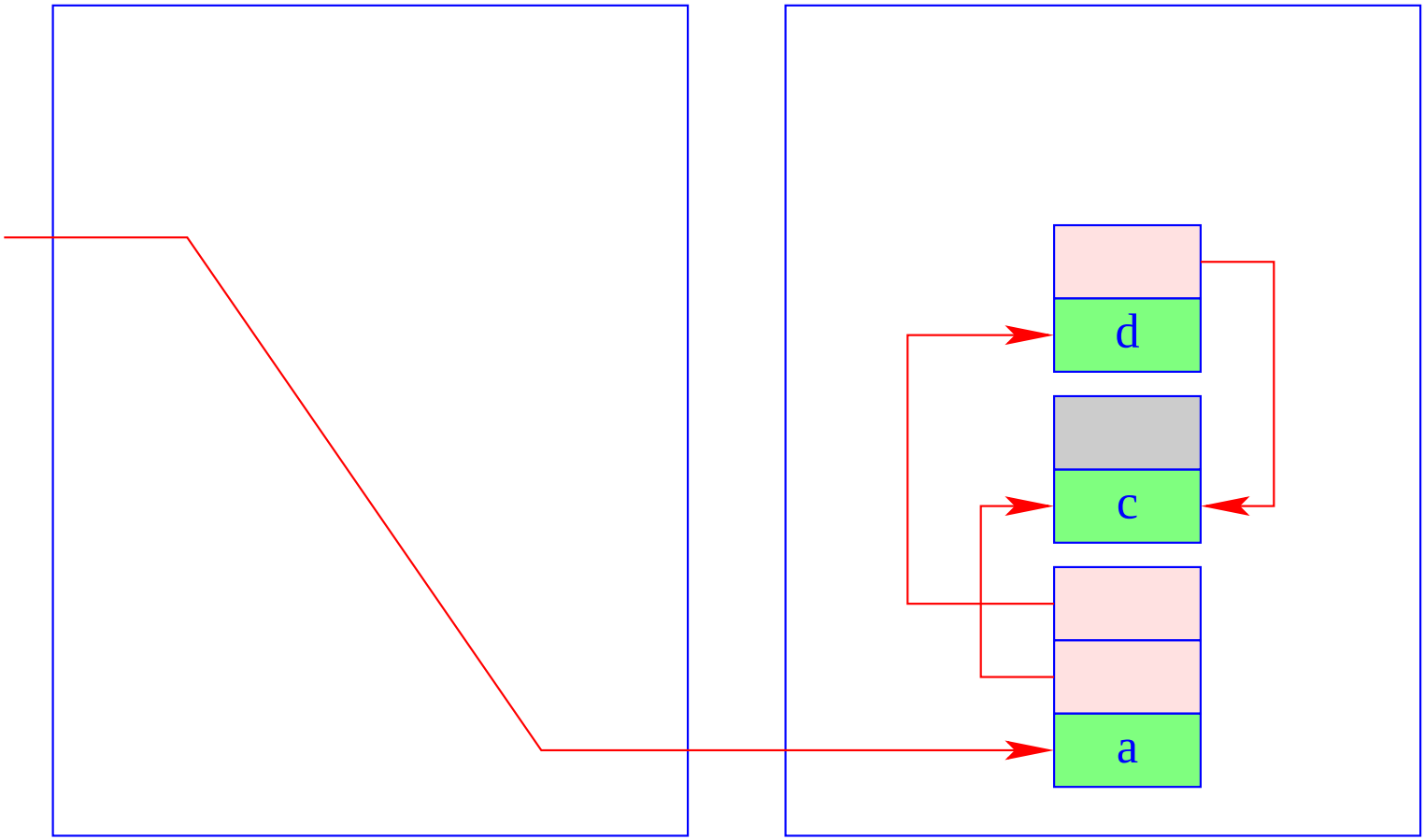


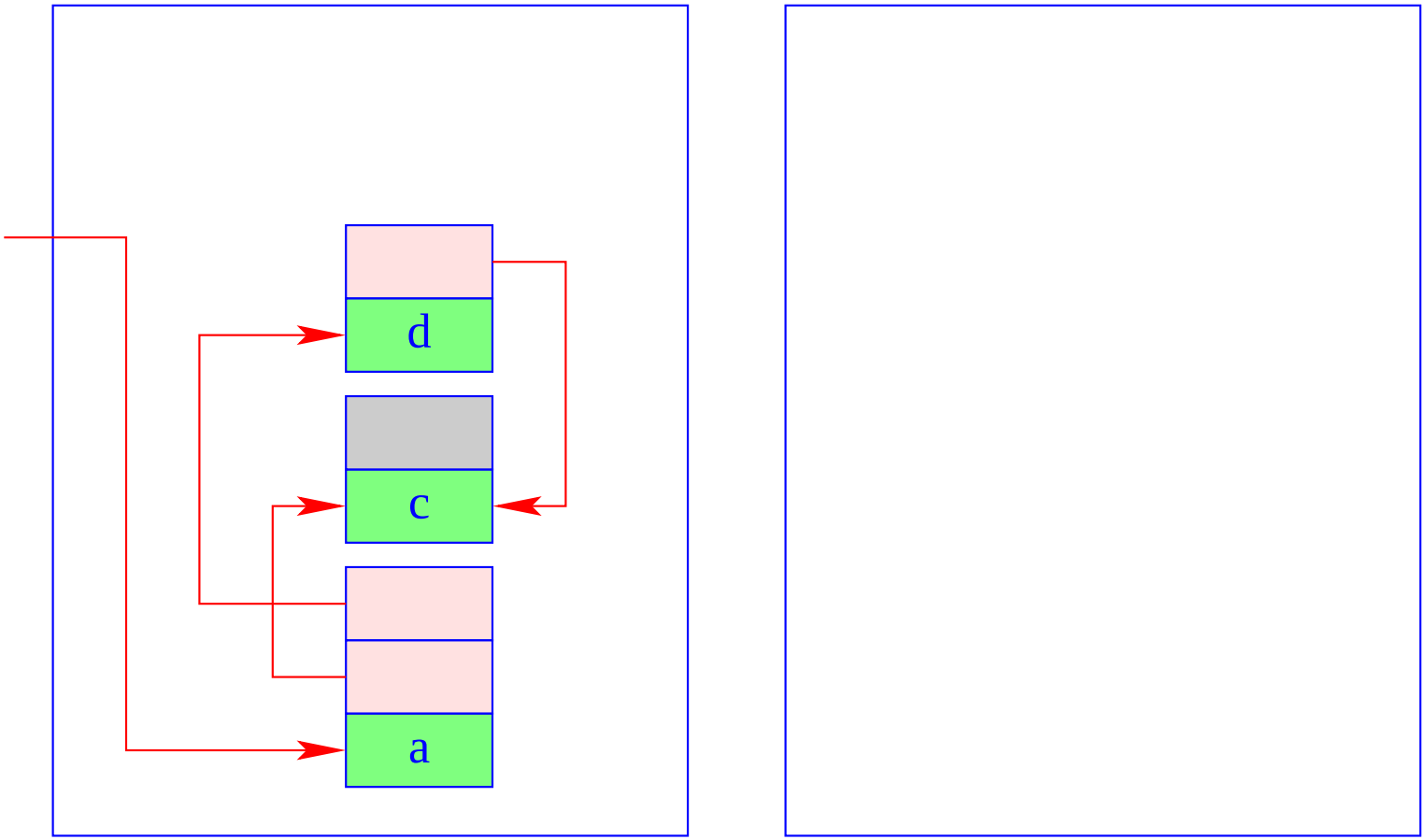

b
d
c
a


b
d
c
a


d
c
a

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



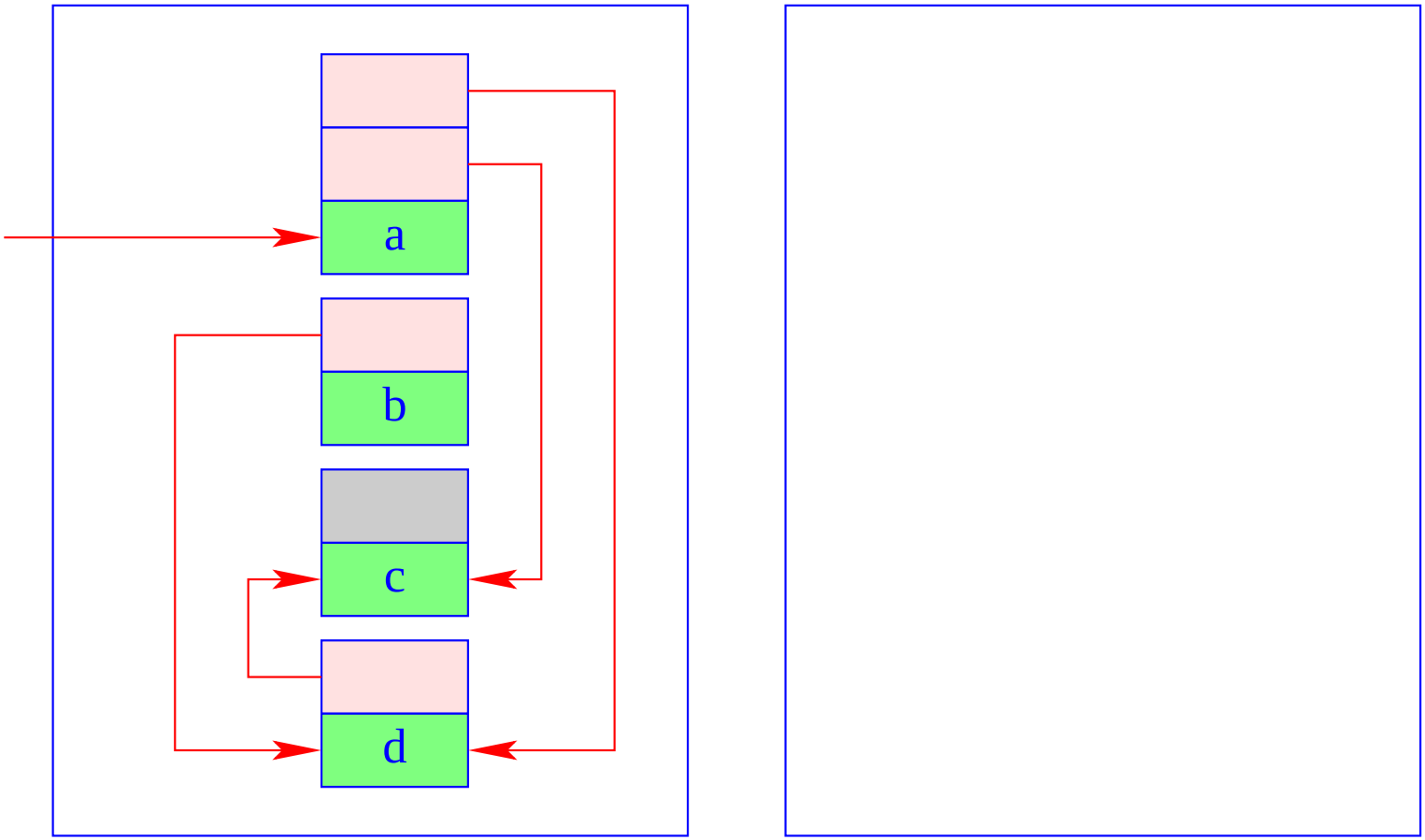

d
c
a

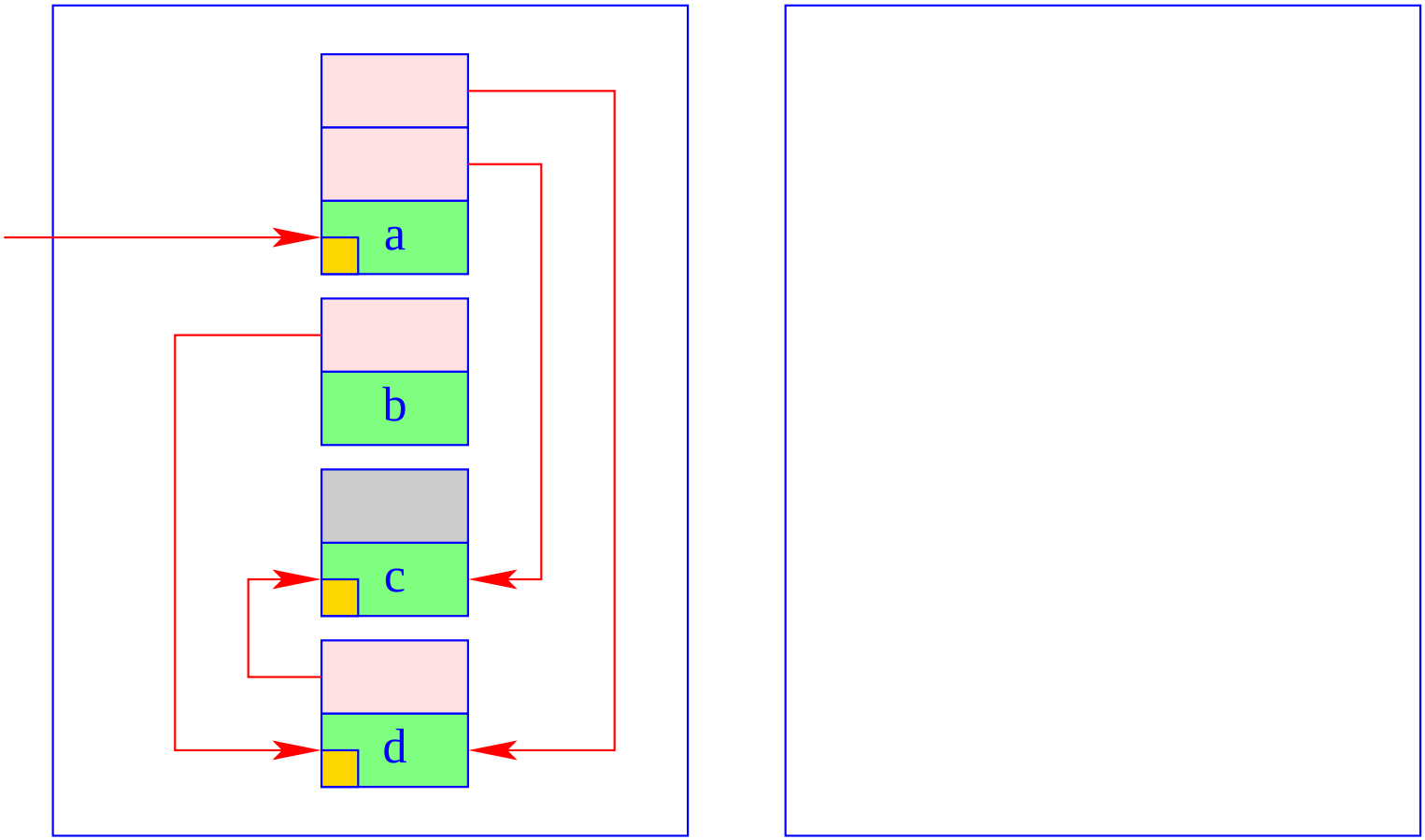
Achtung:

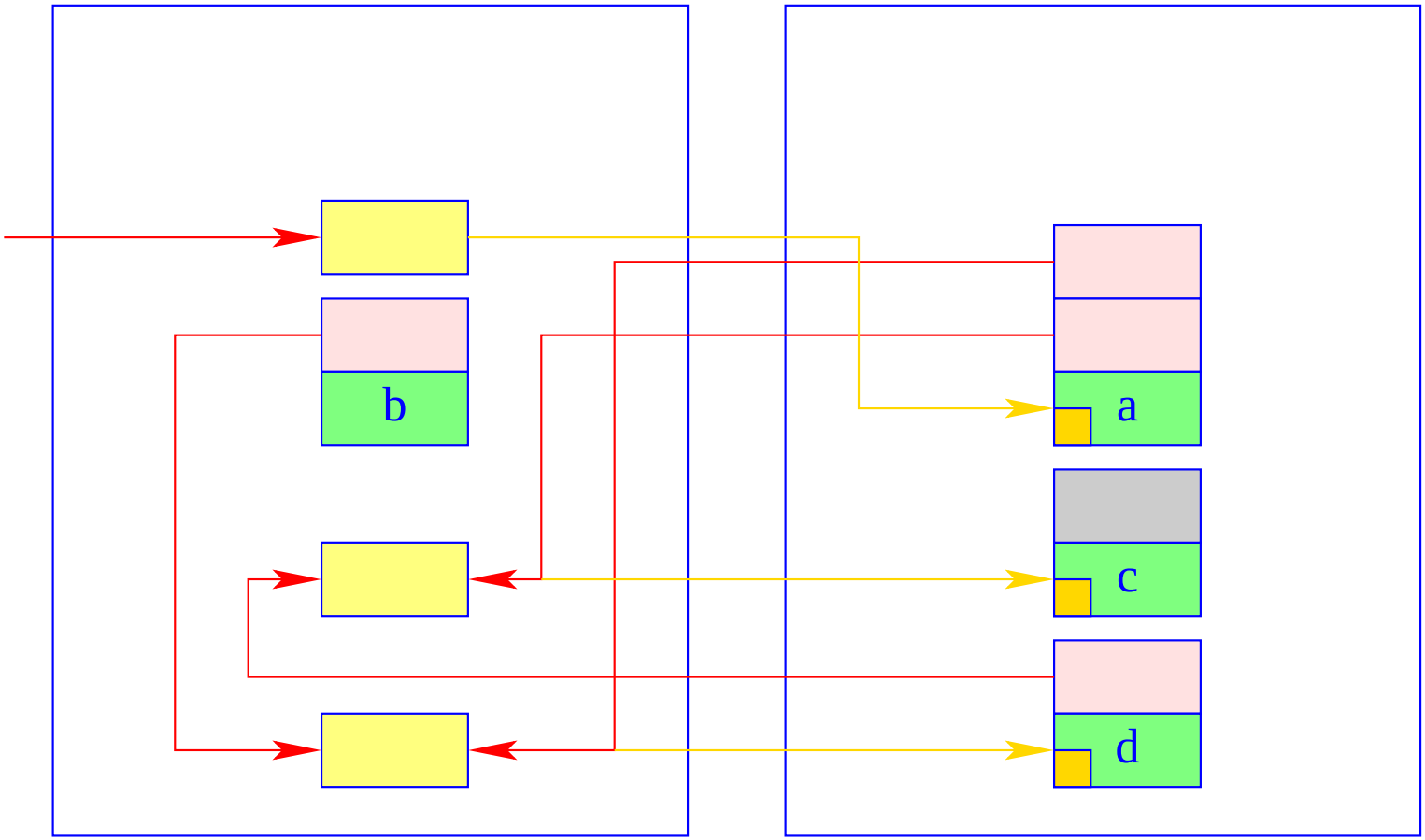
Die Garbage Collection in der WiM muss mit dem Backtracking harmonisieren.

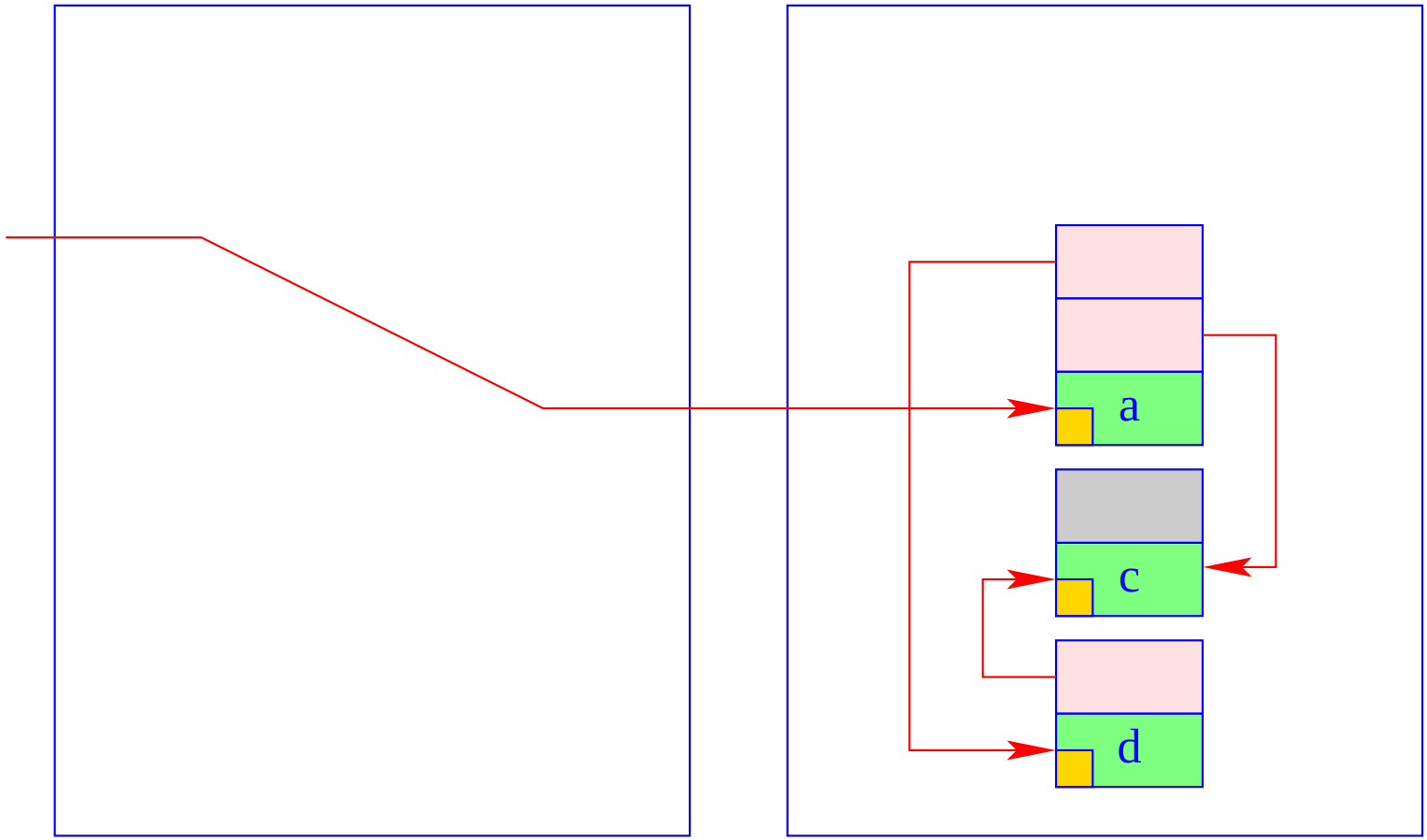
Das heißt:

- 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

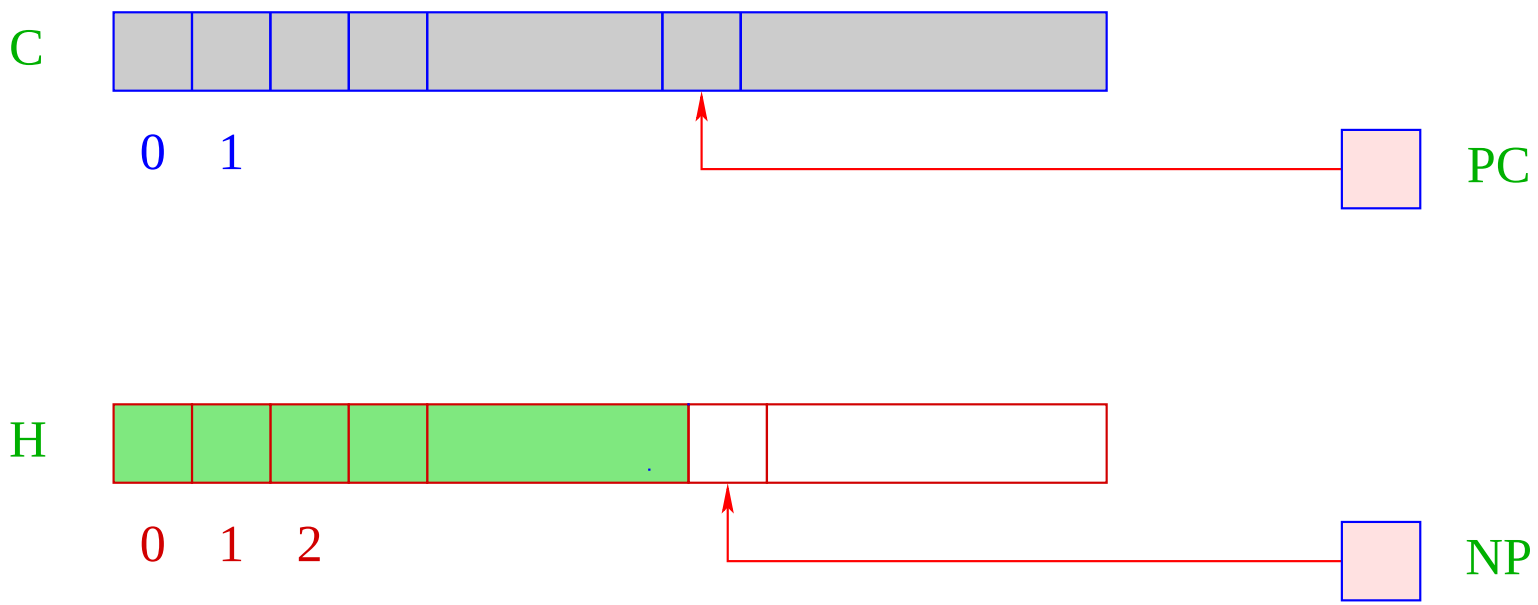
Wir erweitern C um ein einfaches Thread-Konzept. Insbesondere stellen wir Funktionen bereit, um:

- 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:

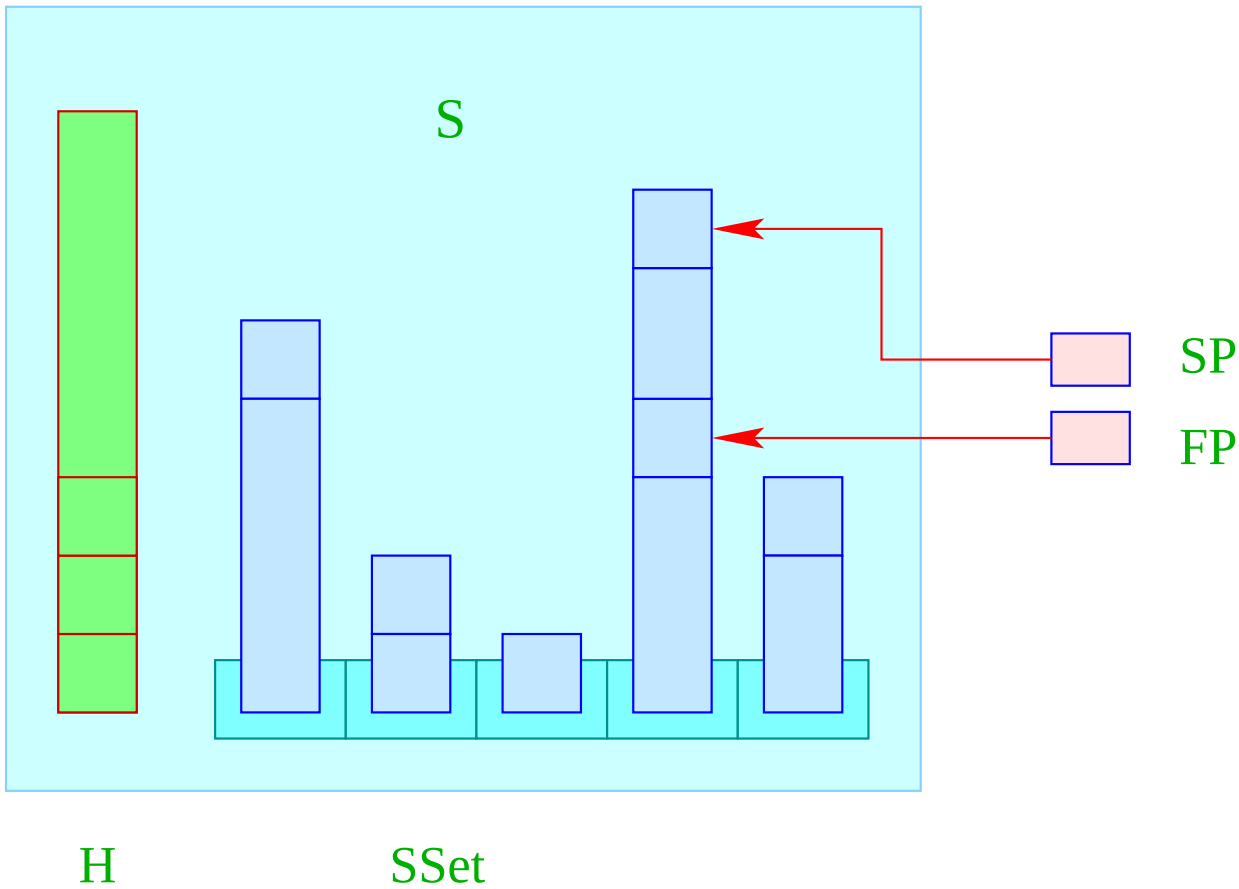


... wie bei der CMa haben wir:

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.

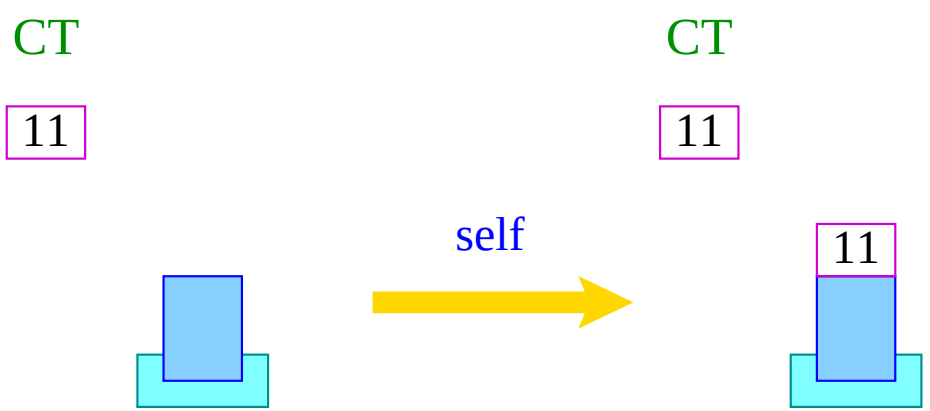
41 Die Ready-Schlange

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:

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

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

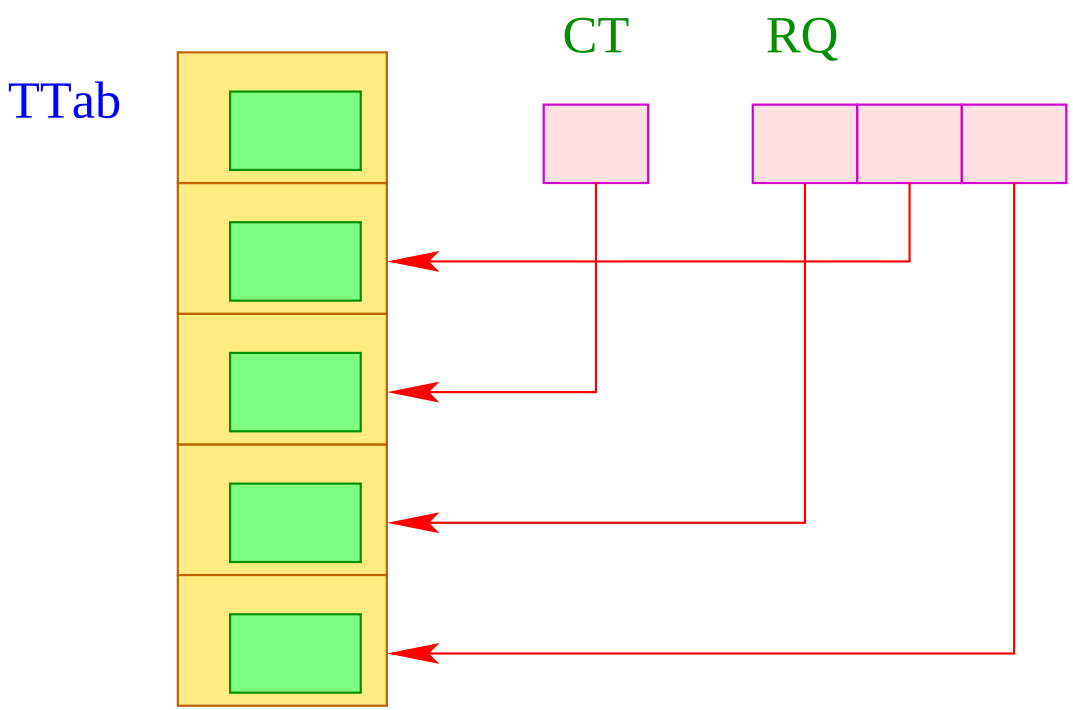


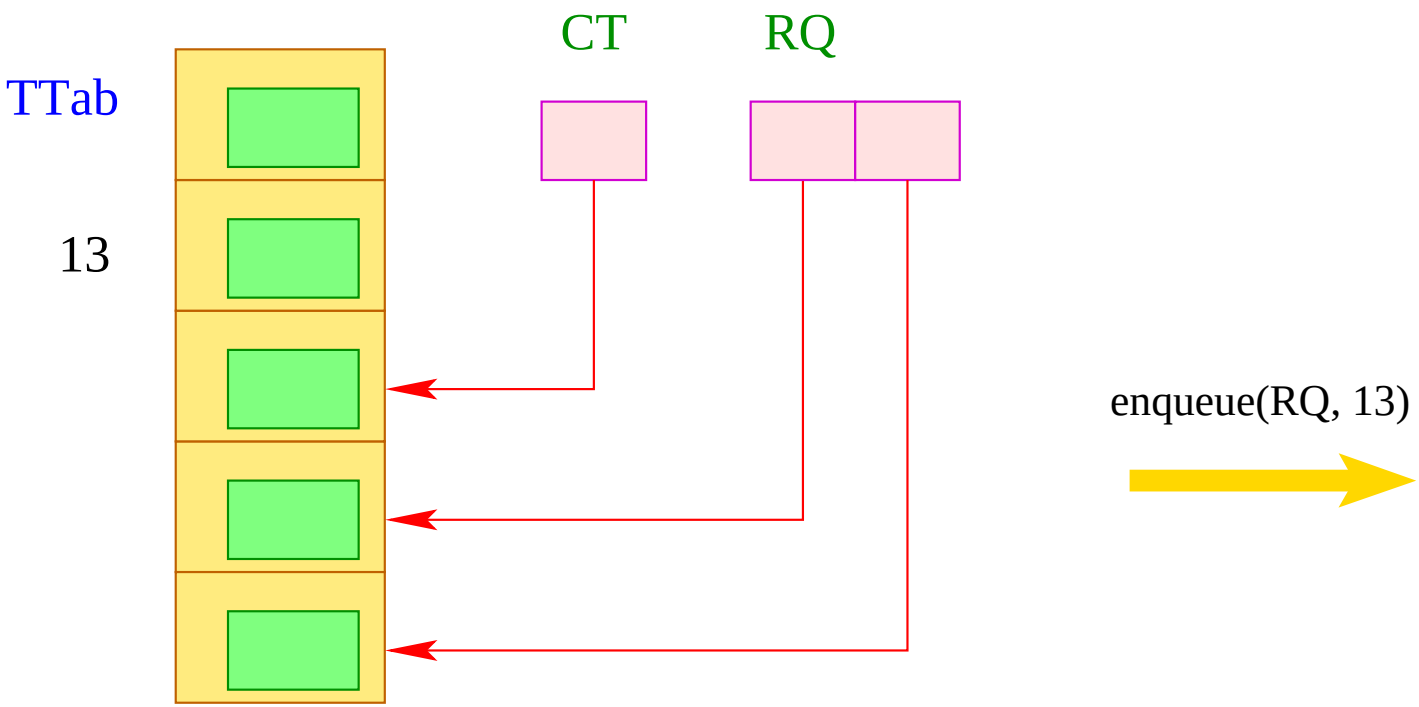
S[SP++] = CT;

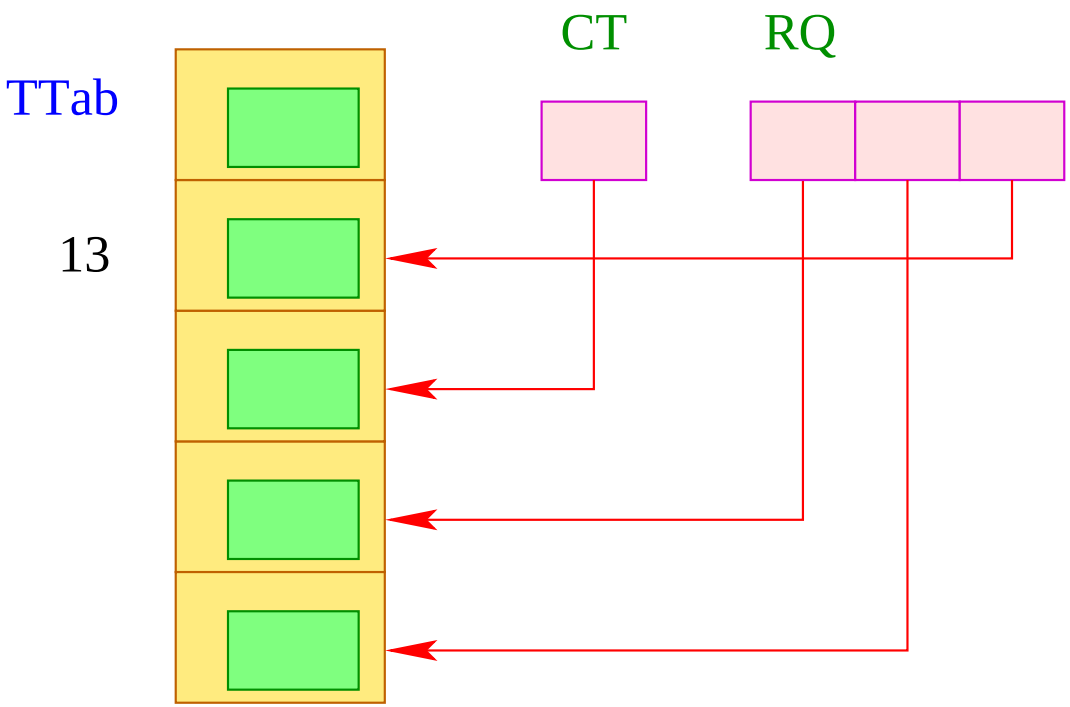
- Die weiteren lauffähigen Threads (bzw. deren tid's) verwalten wir in einer Schlange RQ (Ready-Queue).
- Für Schlangen von Threads benötigen wir die Funktionen:

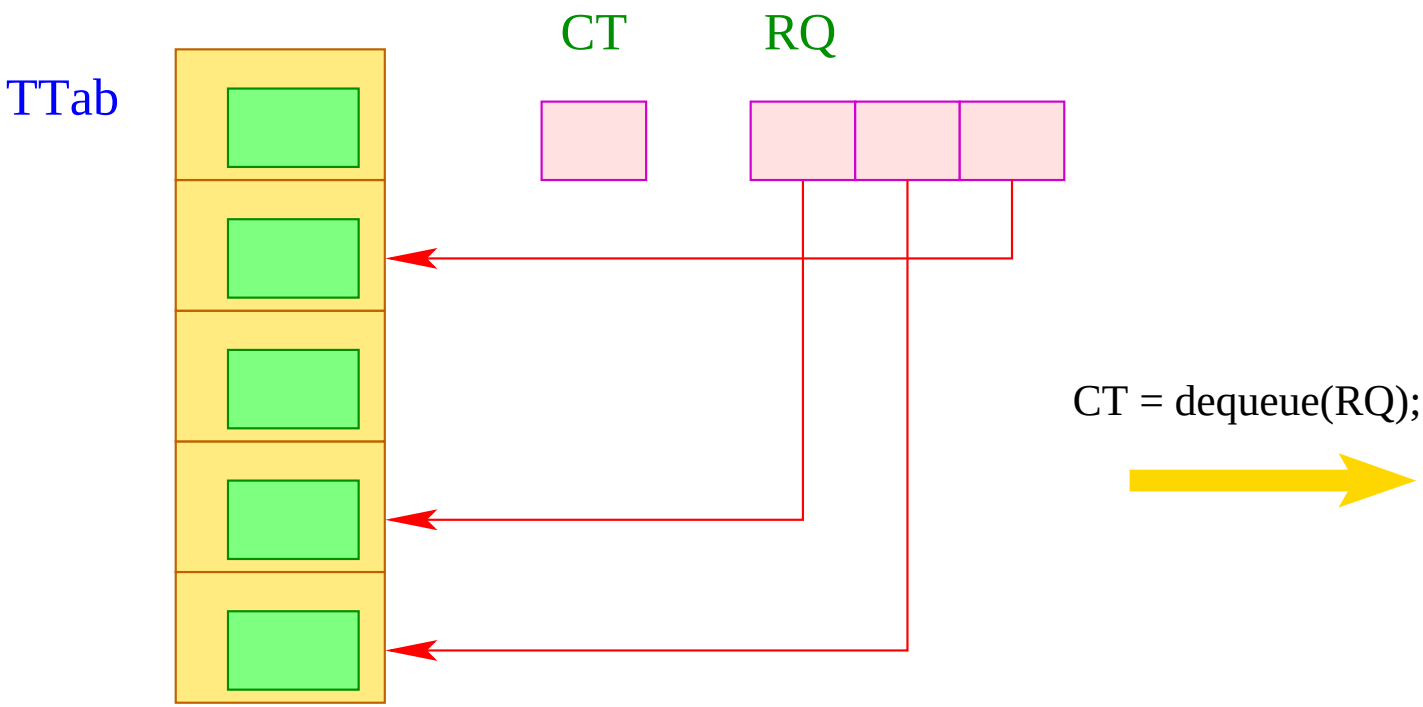
  ```
  void enqueue (queue q, tid t),
  tid dequeue (queue q)
  ```

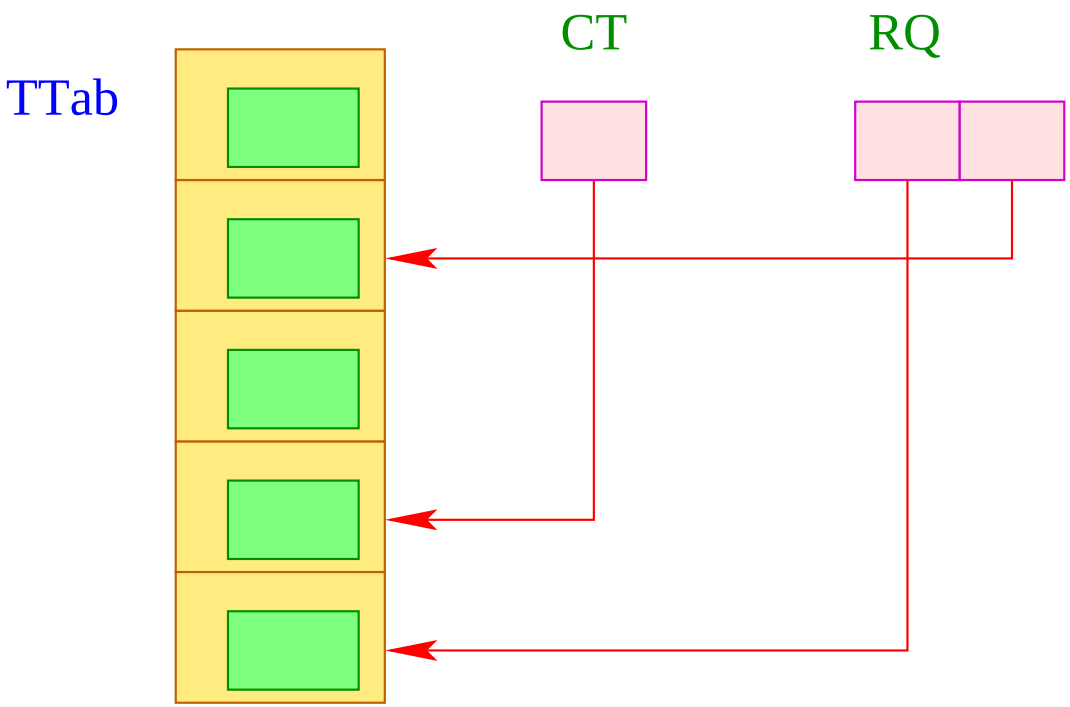
  die eine neue tid in die Schlange einfügen bzw. die erste zurück liefern ...


TTab
CT
RQ


CT
RQ
TTab
13
enqueue(RQ, 13)

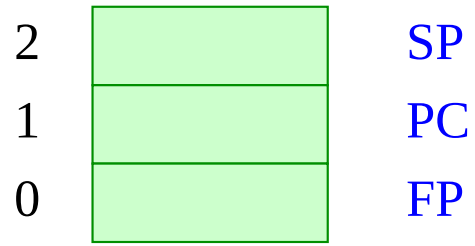

CT
RQ
TTab
13


TTab
CT
RQ
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.

$$\Longrightarrow$$

- 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:

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

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 ...