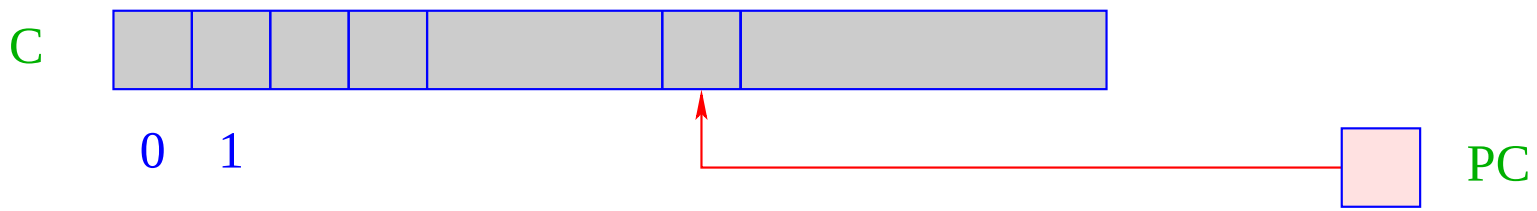
27 Architektur der WiM:

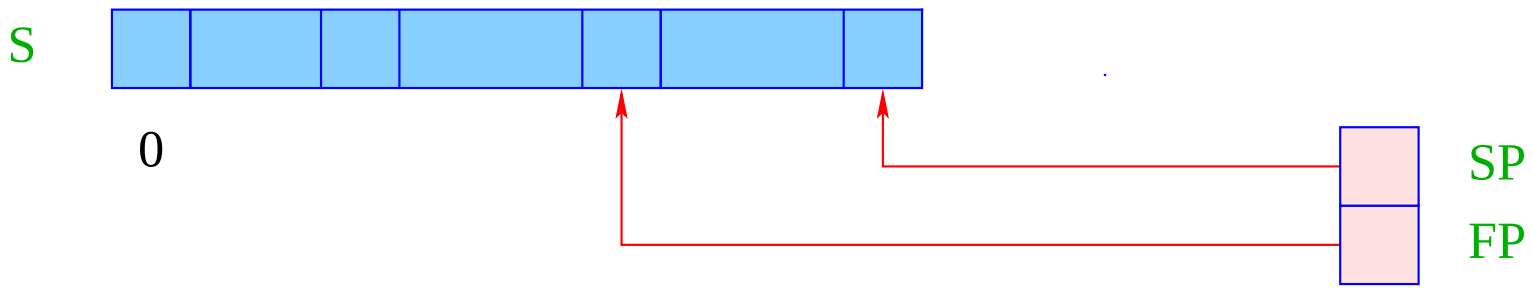
Der Code-Speicher:



C = Code Speicher – enthält WiM-Programm;
jede Zelle enthält einen Befehl;

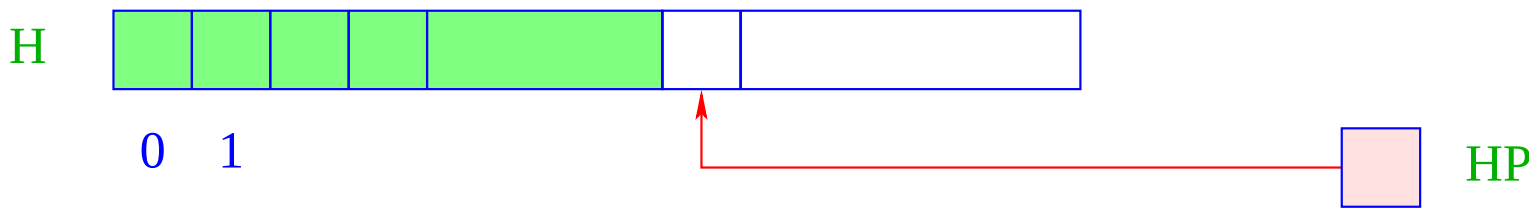
PC = Program Counter – zeigt auf nächsten auszuführenden Befehl.

Der Laufzeit-Keller:



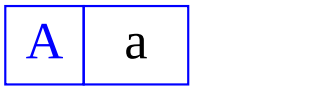
S	=	Laufzeit-Stack – jede Zelle enthält einen Wert oder eine Adresse;
SP	=	Stack Pointer – zeigt auf die oberste belegte Zelle;
FP	=	Frame Pointer – zeigt auf den aktuellen Kellerrahmen.
		Rahmen werden für jeden Prädikat-Aufruf erzeugt,
		enthalten Platz für die Variablen der aktuellen Klausel.

Der Heap:

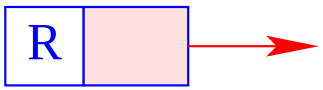


H	=	Heap für dynamisch erzeugte Terme;
HP	=	Heap-Pointer – zeigt auf die erste freie Zelle.

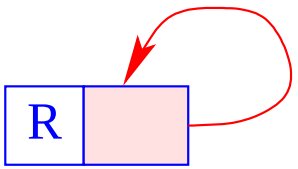
- Der Heap wird ebenfalls wie ein Keller verwaltet :-)
- Ein new-Befehl allokiert ein Objekt in H.
- Objekte sind mit ihrem Typ markiert (wie bei der MaMa) ...



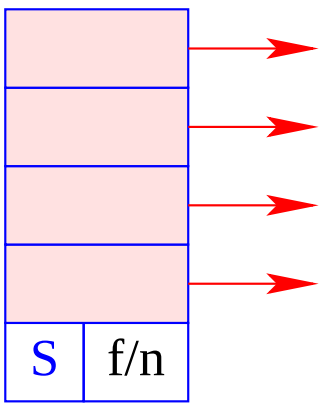
Atom 1 Zelle



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.

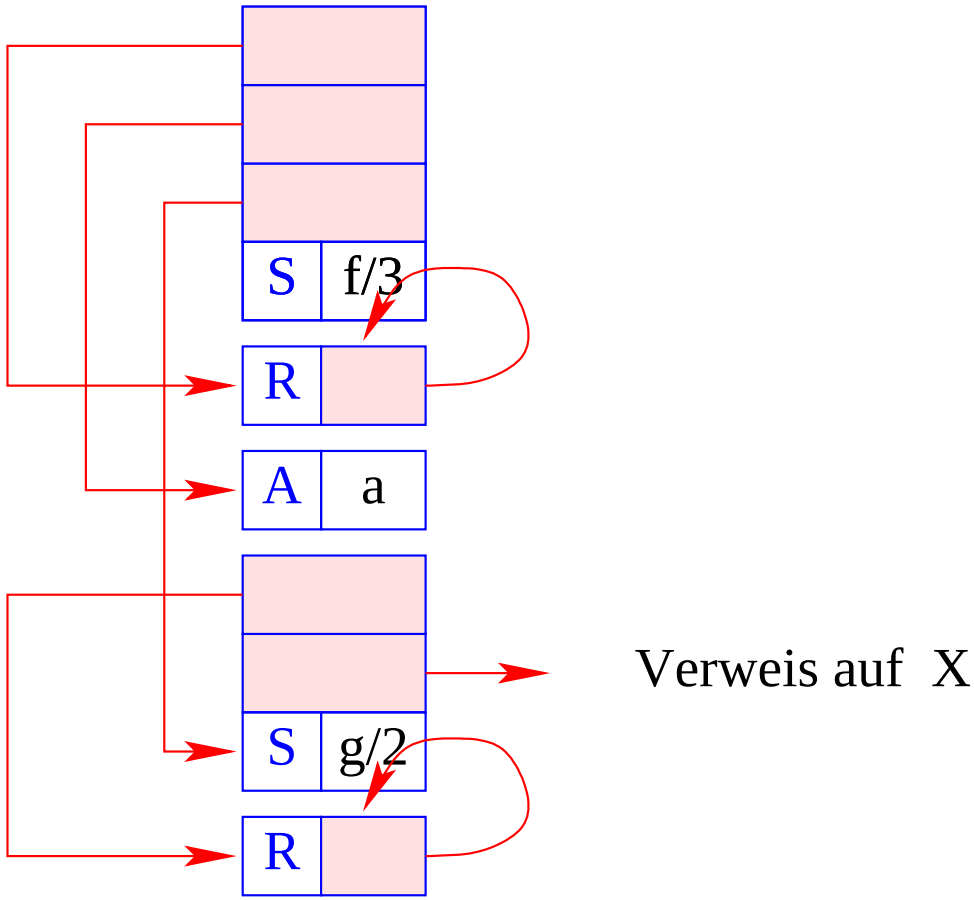
Dann wollen wir für einen Term t eine Folge $\text{code}_A\ t\ \rho$ von Instruktionen erzeugen, die (eine Repräsentation von) t im Heap aufbaut.

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
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:

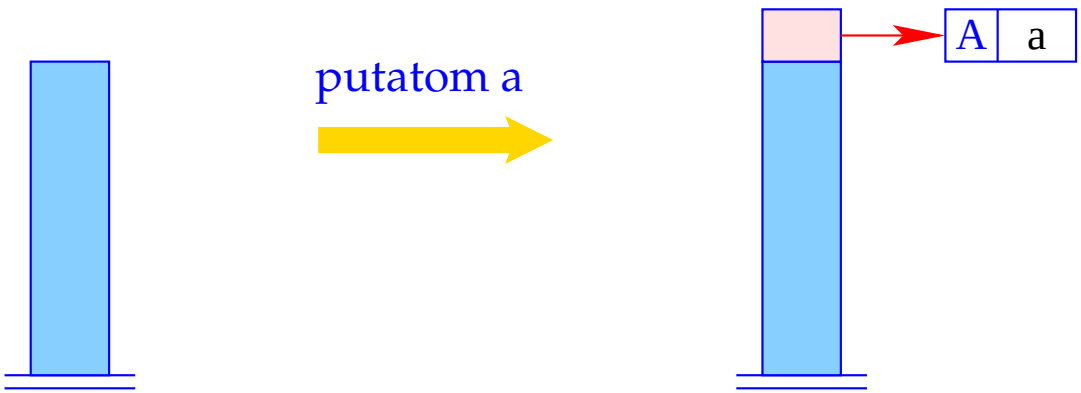
$$
\begin{array}{lcl}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\ X) \\
\text{code}_A\ \bar{X}\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

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

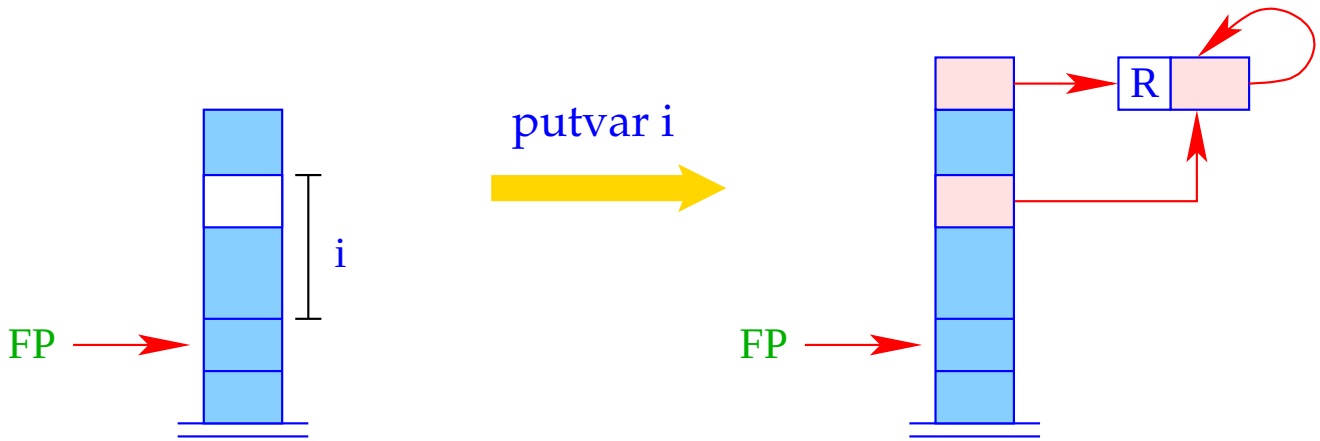
Dabei bedeuten die Befehle:

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



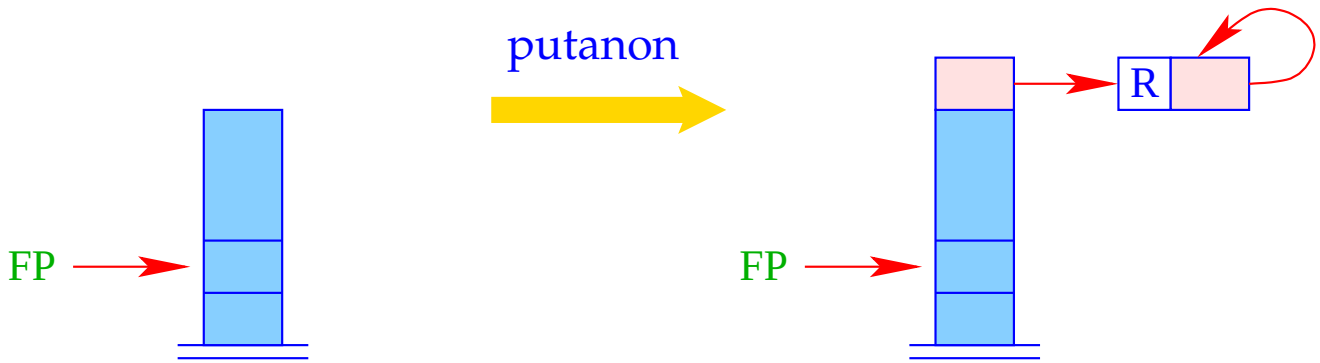
SP++; S[SP] = new (A,a);

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



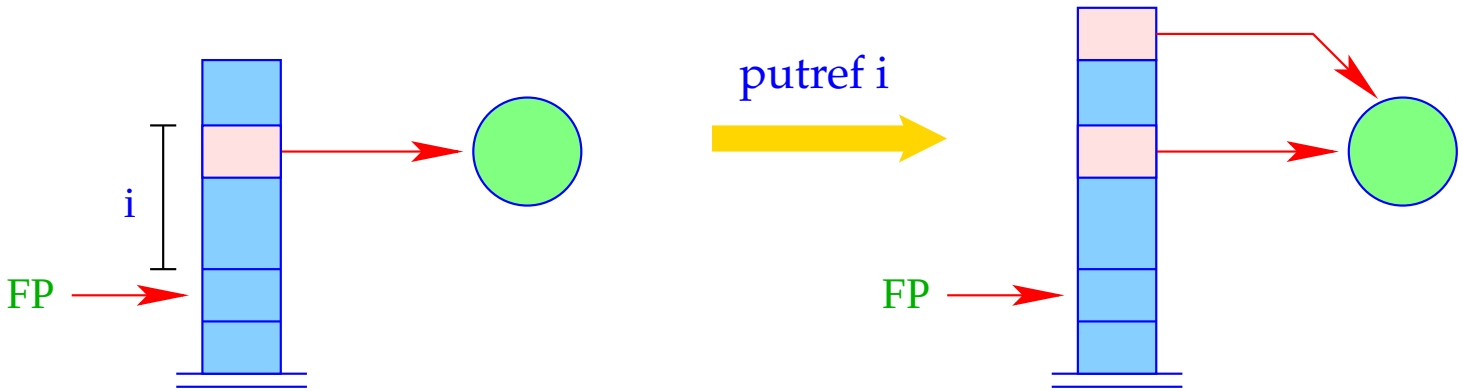

```
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];
```

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



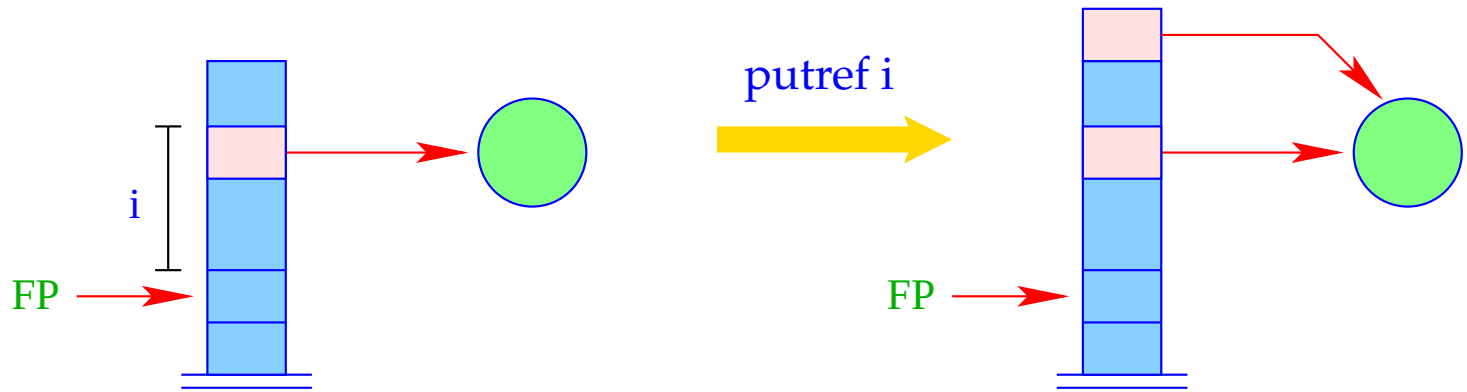
SP = SP + 1;
S[SP] = new (R, HP);

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:

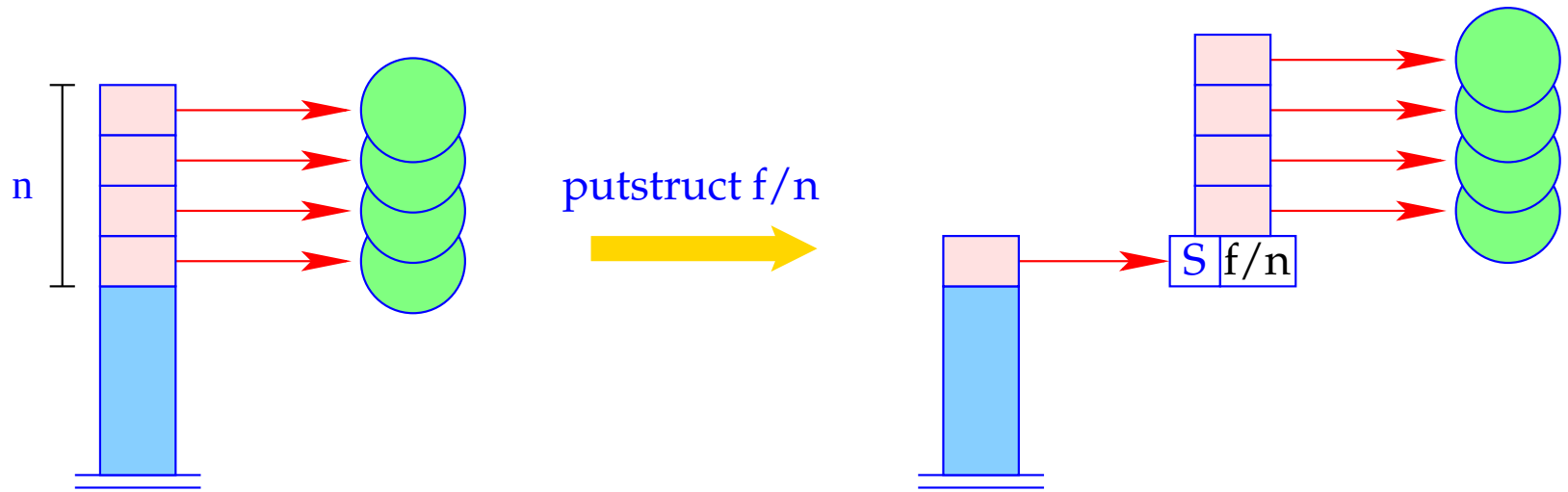


SP = SP + 1;
S[SP] = deref S[FP + i];

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:

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

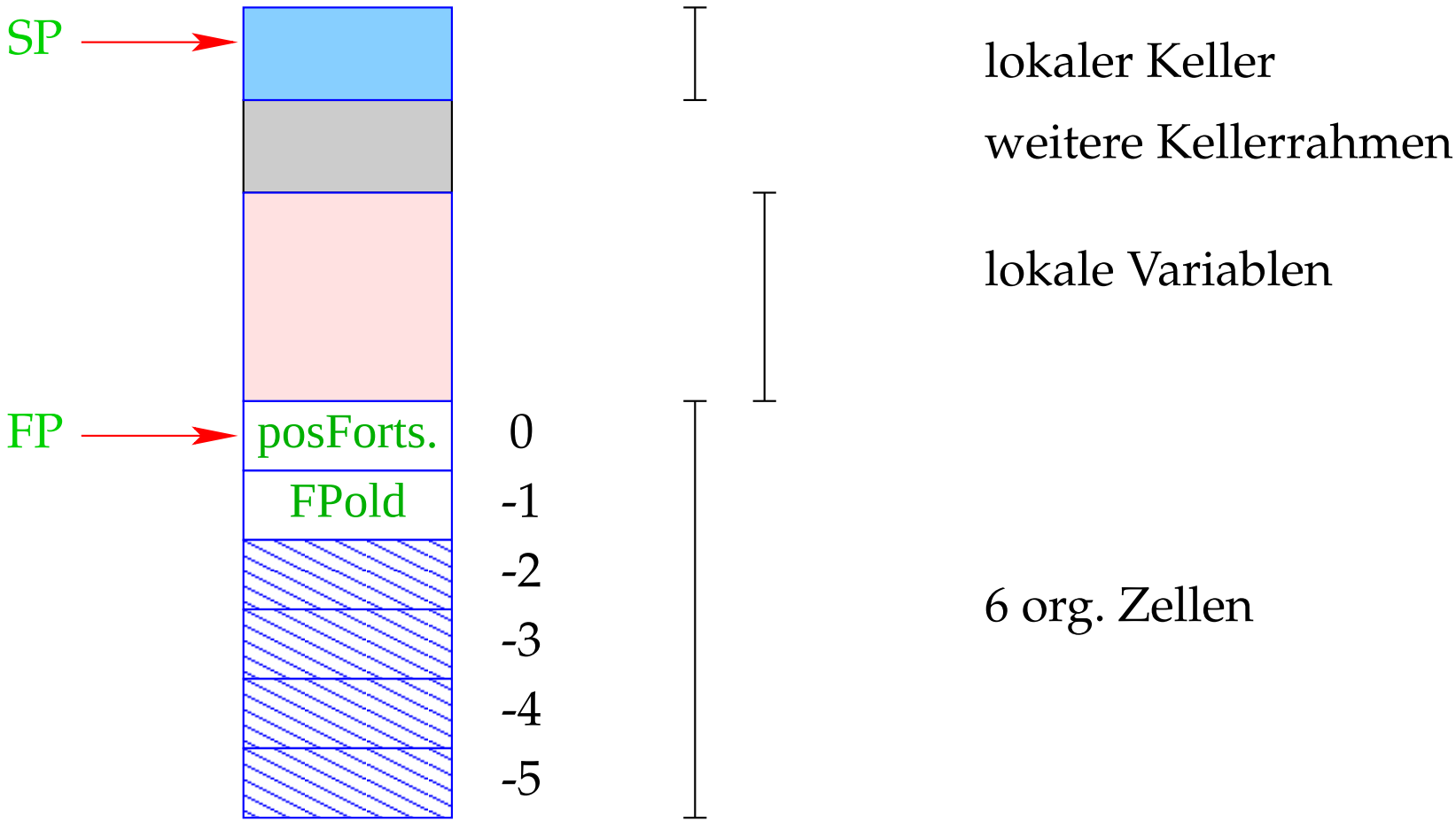
$\text{code}_G\ p(t_1, \ldots, t_k)\ \rho$ =

```
        mark B              // legt einen Kellerrahmen an
        code_A t_1 ρ
        ...
        code_A t_k ρ
        call p/k            // ruft Prozedur p/k auf
B :     ...
```

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
```

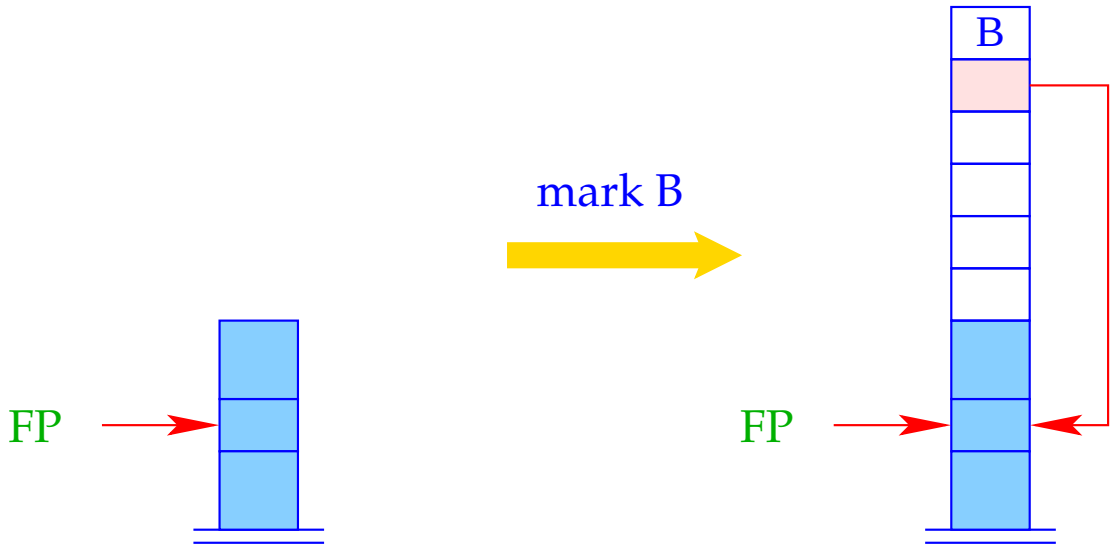
Ein Kellerrahmen der WiM hat den folgenden Aufbau:


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

Bemerkungen:

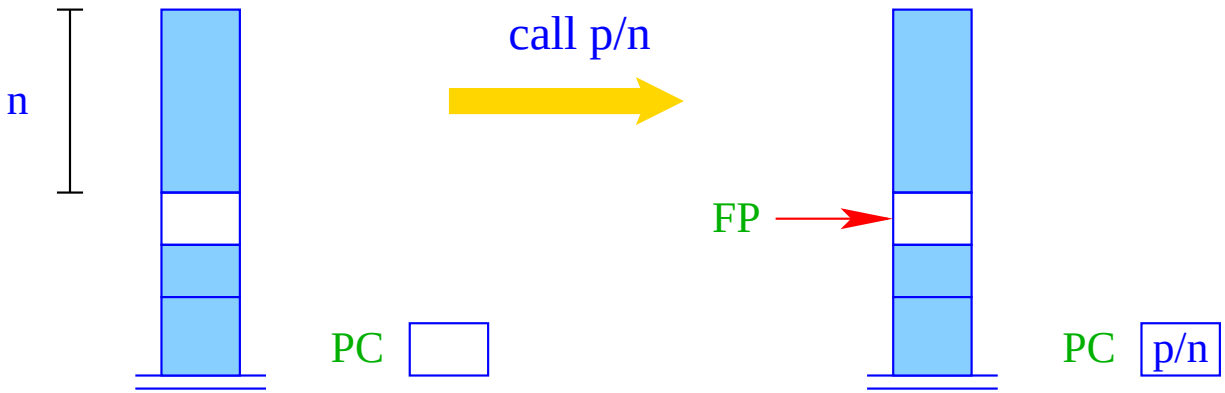
- Zwischen lokalem Keller und den lokalen Variablen können weitere Kellerrahmen eingeschoben sein :-)
- Die positive Fortsetzungsadresse gibt an, an welcher Stelle im Code fortgefahren werden soll, wenn das Ziel erfolgreich abgearbeitet wurde.
- Die Zelle FPold enthält einen Verweis auf den Rahmen des 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:

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

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 :-)

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

Beispiel:

Betrachte die Gleichung:

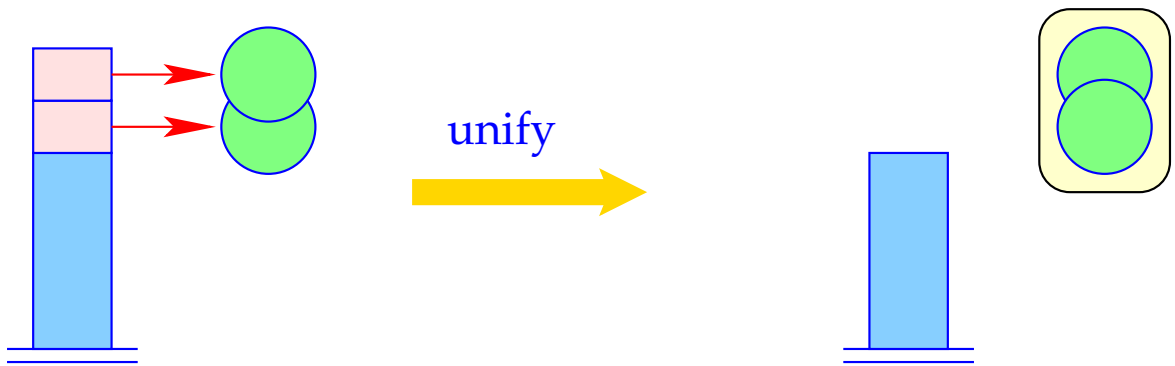
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

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:



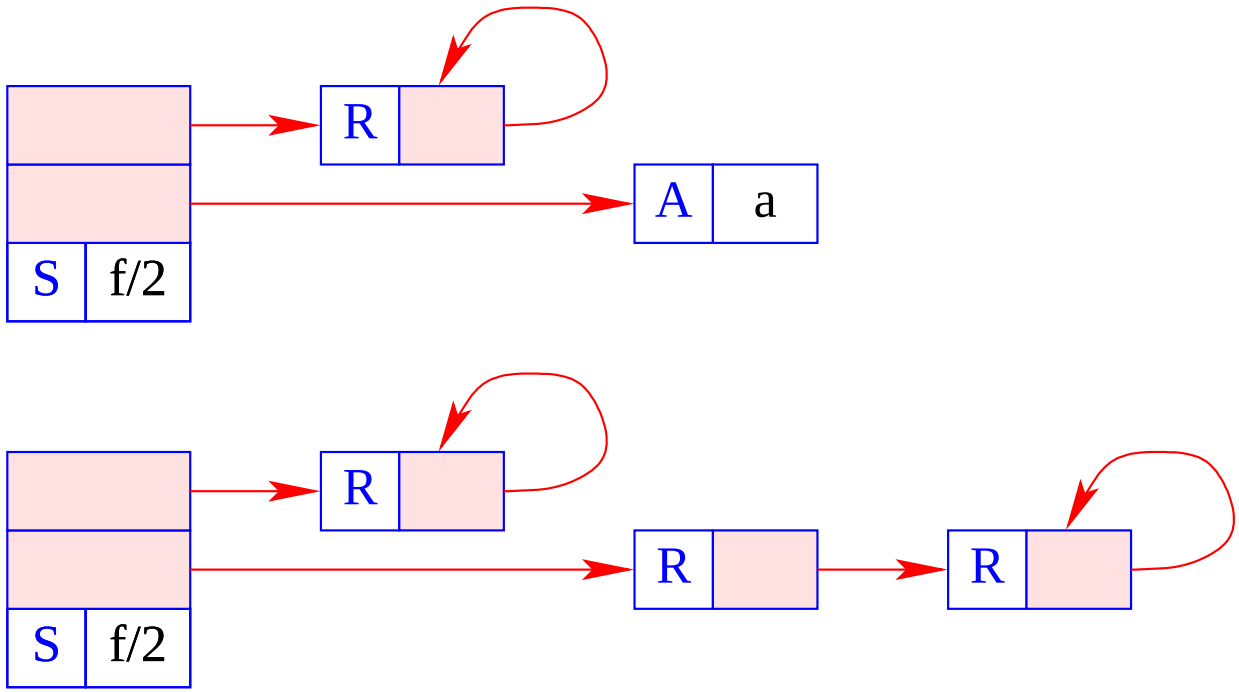

```
unify (S[SP-1], S[SP]);
SP = SP–2;
```

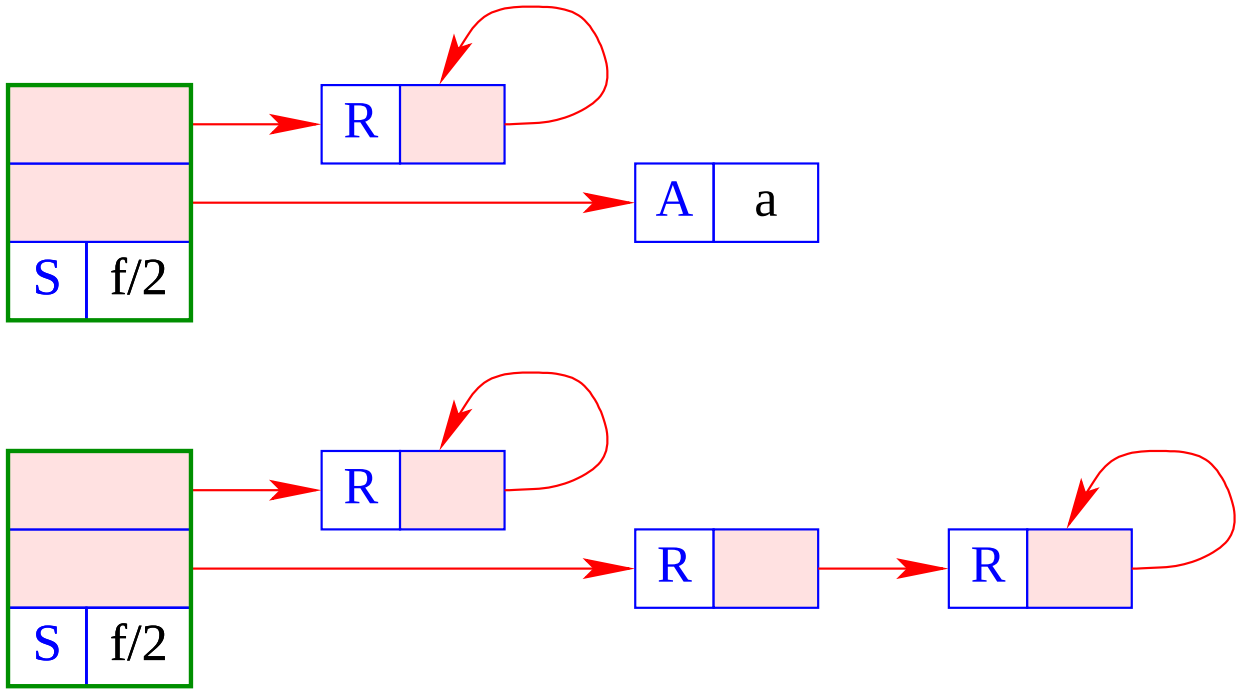
Die Funktion `unify()` des Laufzeit-Systems erhält als Input zwei Heap-Adressen und führt die Unifikation durch. Dabei beachtet sie, dass

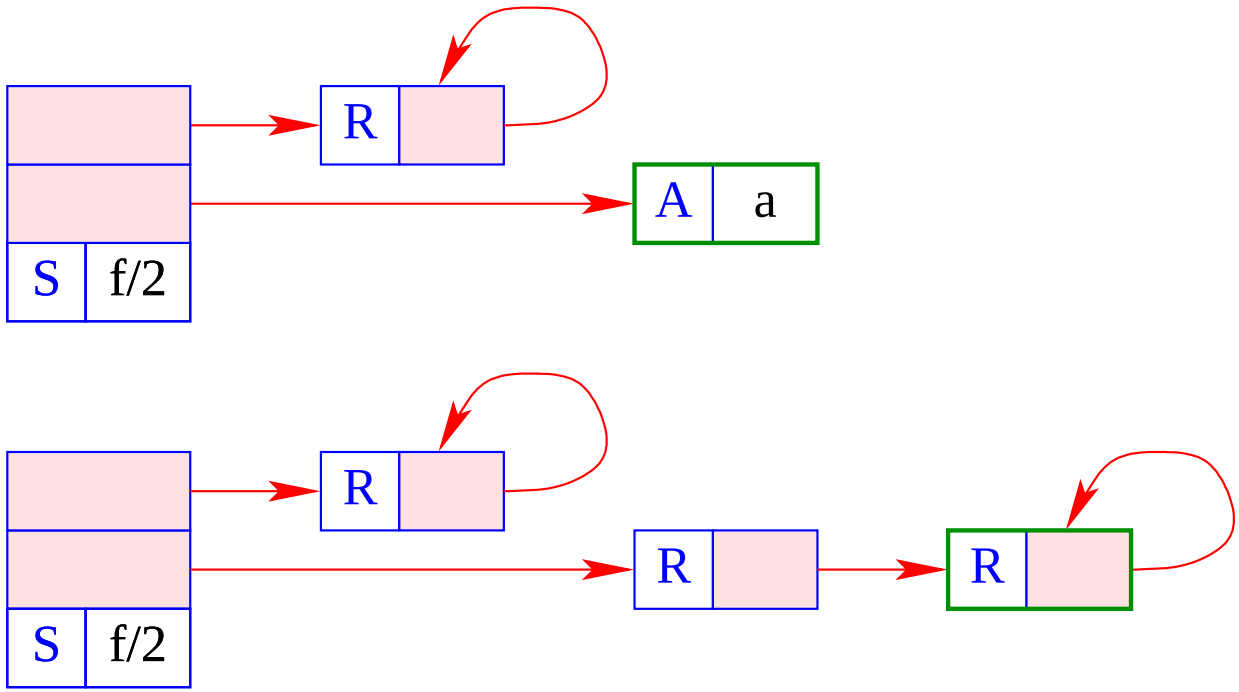
- gleiche Heapadressen bereits unifiziert sind :-)
- beim Binden zweier Variablen aneinander die jüngere (größere Adresse) an die ältere (kleinere Adresse) gebunden wird;
- beim Binden einer Variablen an einen Term diese Variable nicht auch noch im Term vorkommt (Occur Check);
- eingegangene Bindungen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

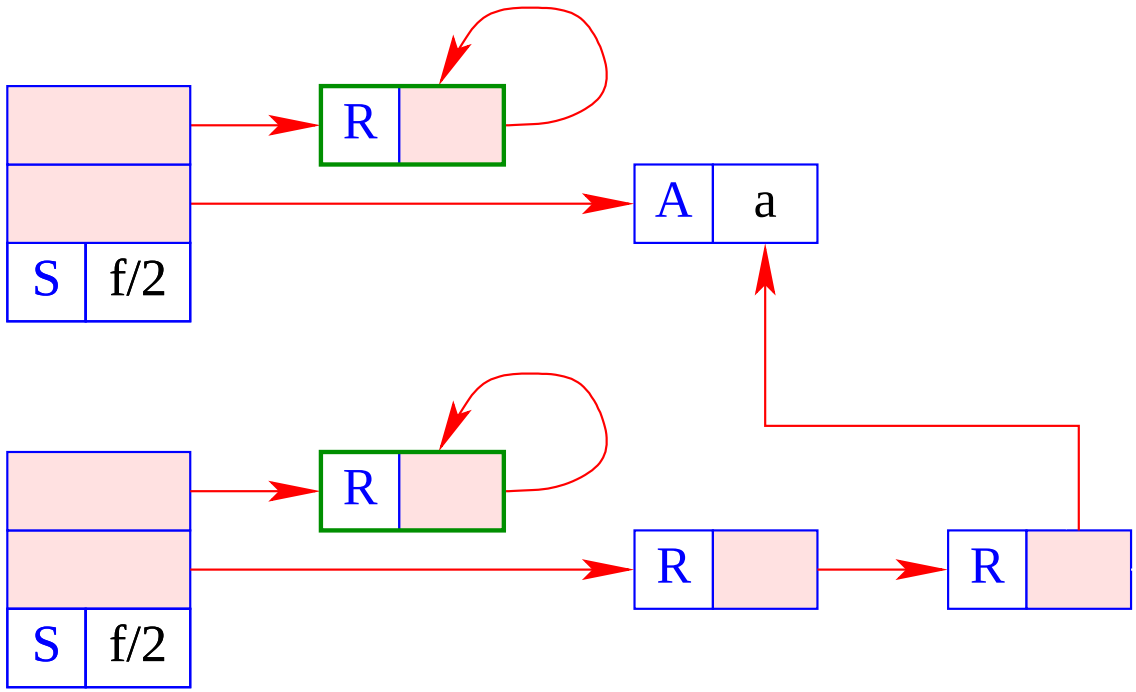
```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u]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

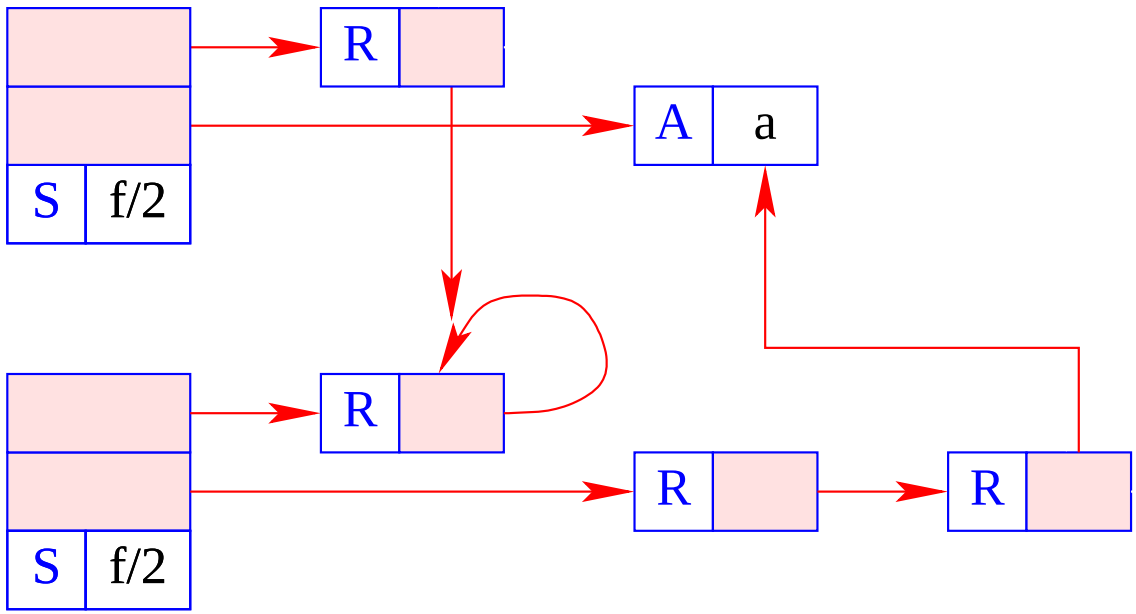
```
...
 if ((H[v] == (R,_)) {
      if (check (v,u)) {
         H[v] = (R,u); trail (v); return true;
      } else {
         backtrack(); return false;
      }
   }
   if (H[u]==(A,a) && H[v]==(A,a))
      return true;
   if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
      for (int i=1; i<=n; i++)
     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

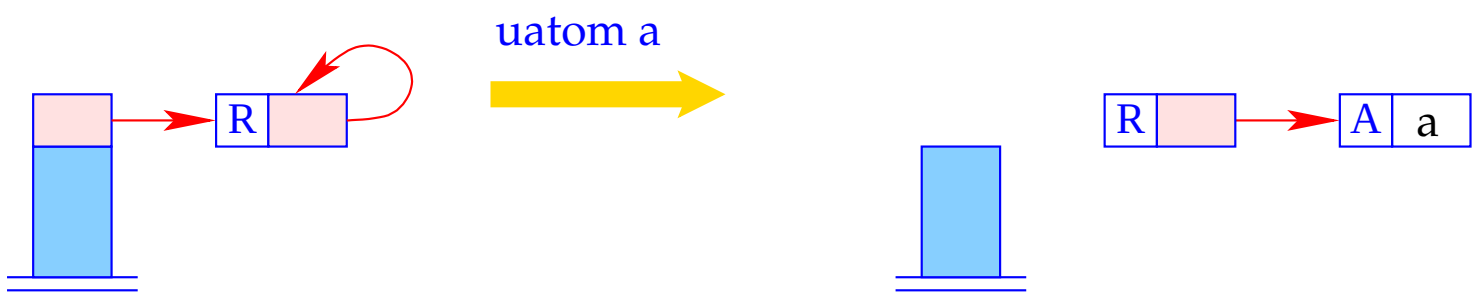
- Lege einen Verweis auf die aktuelle Bindung von X oben auf den Keller.
- Vermeide die Konstruktion von Termen soweit möglich!
- Übersetze t in eine Instruktions-Folge, die die Unifikation mit t implementiert !!!

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

Betrachten wir zuerst Unifikation nur für Atome und Variablen:

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

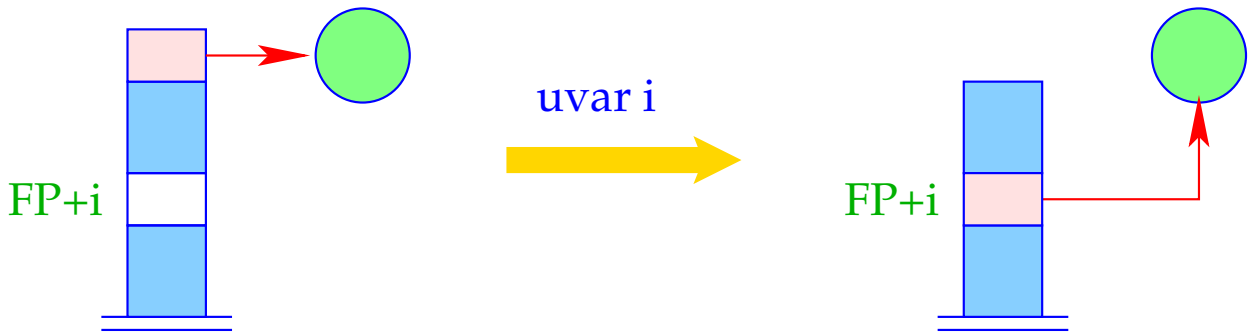
Die Instruktion uatom a implementiert die Unifikation mit dem Atom a:




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

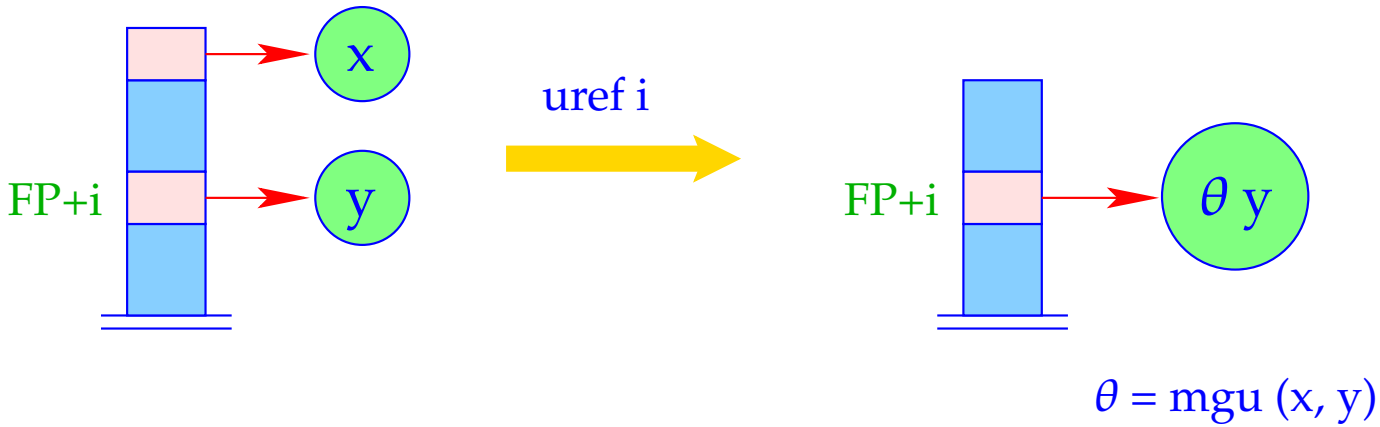
- Der Aufruf von `trail()` vermerkt eine potentiell neue Bindung.
- Der Aufruf von `backtrack()` initiiert backtracking.

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 :-)

$$
\begin{array}{lll}
\text{code}_U\ f(t_1,\ldots,t_n)\ \rho = & \text{ustruct f/n } A & \\
& \text{son 1} & \text{// rekursiver Abstieg} \\
& \text{code}_U\ t_1\ \rho & \\
& \ldots & \\
& \text{son n} & \text{// rekursiver Abstieg} \\
& \text{code}_U\ t_n\ \rho & \\
& \text{up B} & \text{// Rückkehr zum Vater} \\
A: & \text{check } \mathit{ivars}(f(t_1,\ldots,t_n))\ \rho & \text{// Occur-Check} \\
& \text{code}_A\ f(t_1,\ldots,t_n)\ \rho & \\
& \text{bind} & \text{// stellt die Bindung her} \\
B: & \ldots &
\end{array}
$$