

Ein Programm p ist darum wie folgt aufgebaut:

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

- 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** $p(X_1, \dots, X_k)$ 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 Berechnungsschritt
Bindung von Variablen	==	Seiteneffekt

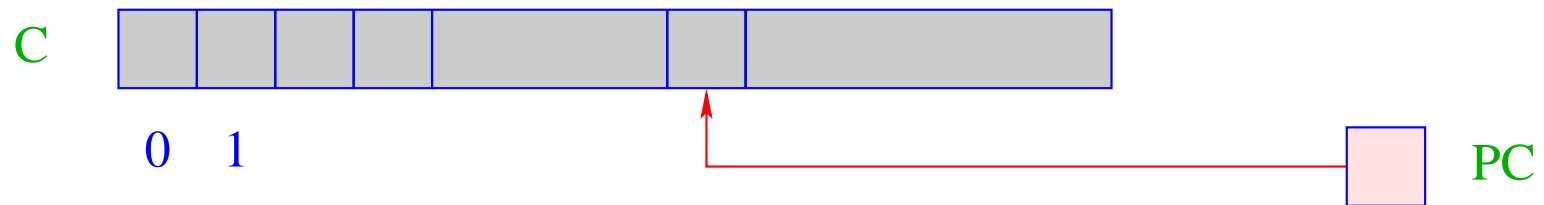
Achtung: Prädikat-Aufrufe ...

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

⇒ backtracking

27 Architektur der WiM:

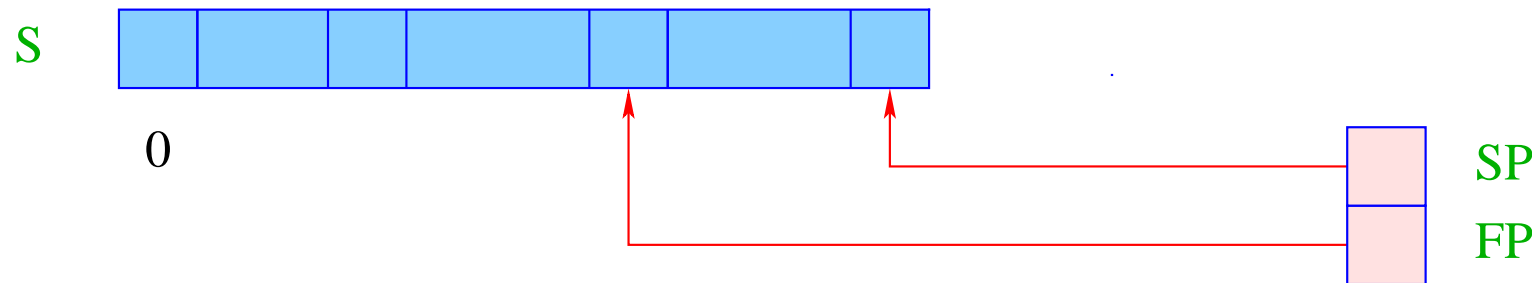
Der Code-Speicher:



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

PC = Program Counter – zeigt auf nächsten auszuführenden Befehl.

Der Laufzeit-Keller:



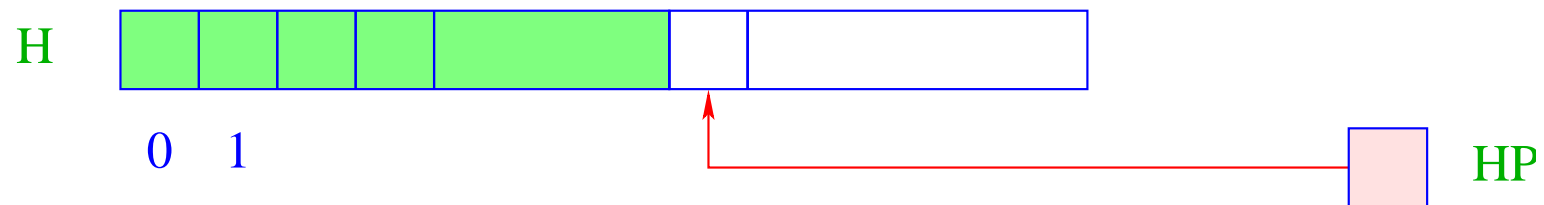
S = Laufzeit-Stack – jede Zelle enthält einen Wert oder eine Adresse;

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:



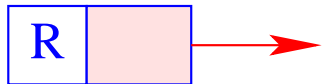
H = Heap für dynamisch erzeugte Terme;
HP = Heap-Pointer – zeigt auf die erste freie Zelle.

- 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**) ...



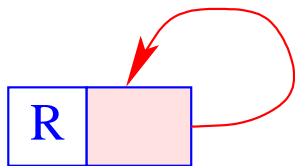
Atom

1 Zelle



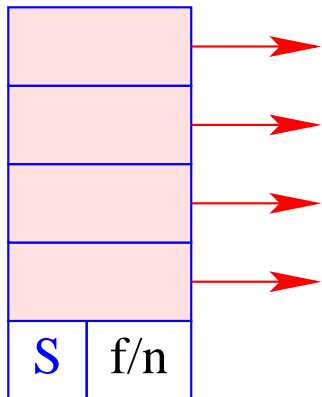
Variable

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.

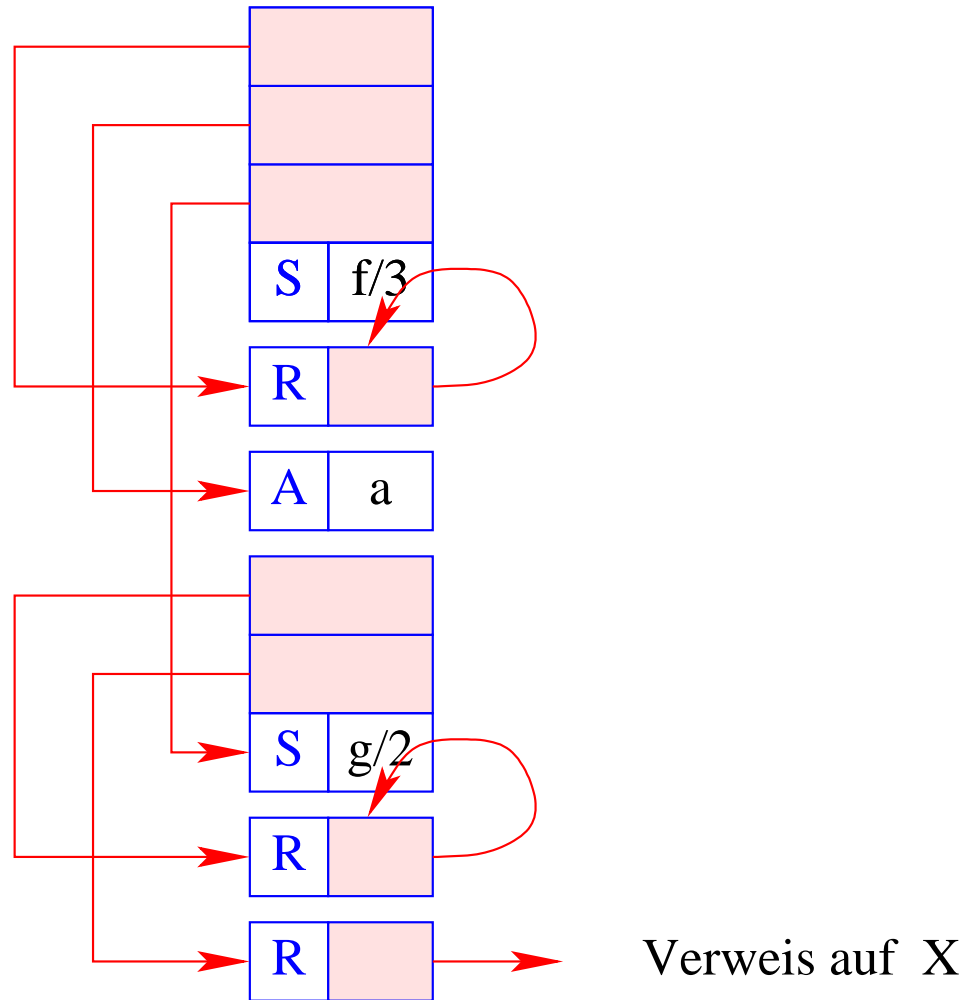
Dann wollen wir für einen Term t eine Folge $\text{code}_A t \rho$ von Instruktionen erzeugen, die (eine Repräsentation von) t im Heap aufbaut.

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:



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:

<code>code_A a ρ</code>	=	<code>putatom a</code>
<code>code_A f(t₁, ..., t_n) ρ</code>	=	<code>code_A t₁ ρ</code>
		...
		<code>code_A t_n ρ</code>
		<code>putstruct f/n</code>
<code>code_A X ρ</code>	=	<code>putvar (ρ X)</code>
<code>code_A \bar{X} ρ</code>	=	<code>putref (ρ X)</code>
<code>code_A _ ρ</code>	=	<code>putanon</code>

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

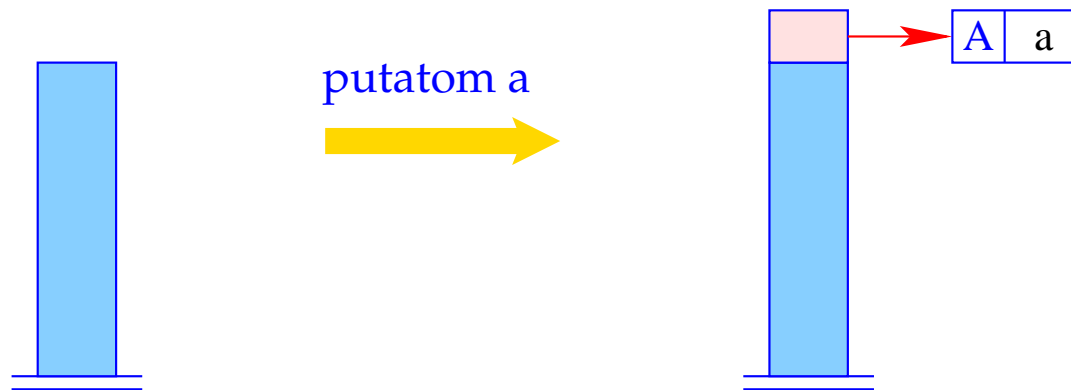
putvar 3

putstruct g/2

putstruct f/3

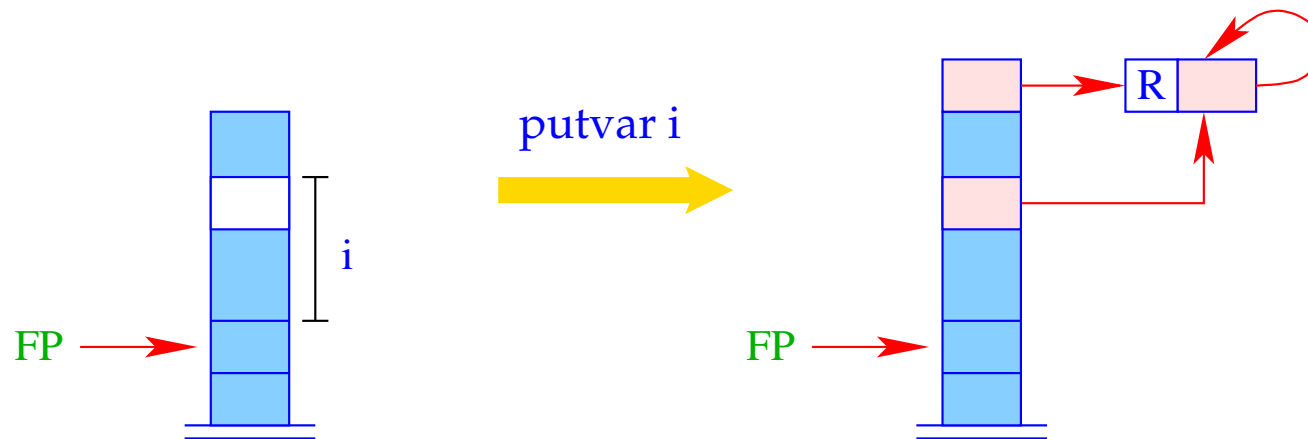
Dabei bedeuten die Befehle:

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



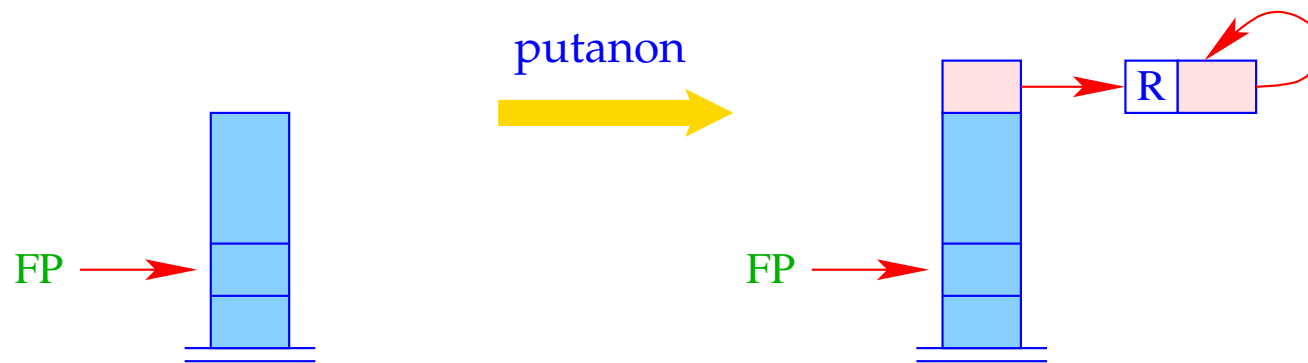
`SP++; S[SP] = new (A,a);`

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



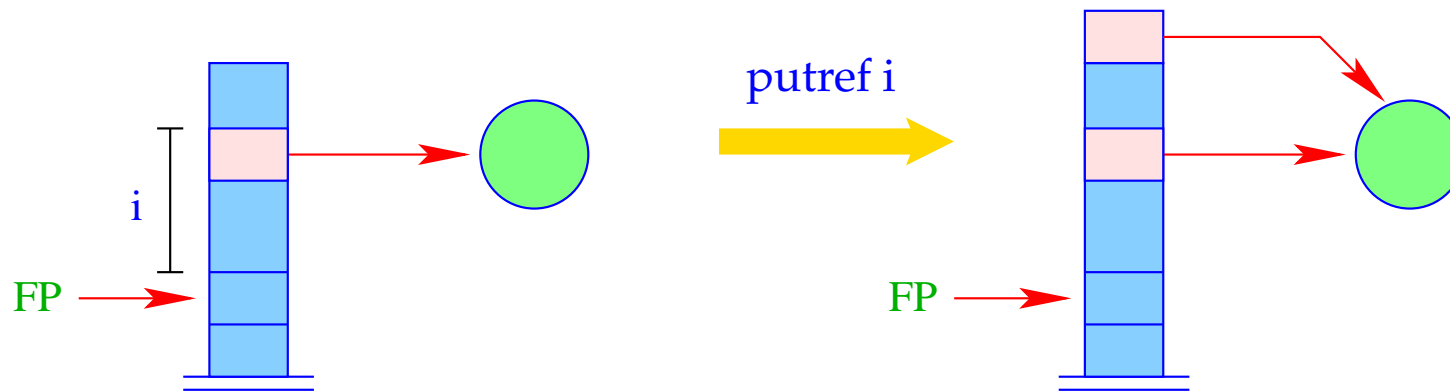
```
SP = SP + 1;  
S[SP] = new (R, HP);  
S[FP + i] = S[SP];
```

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



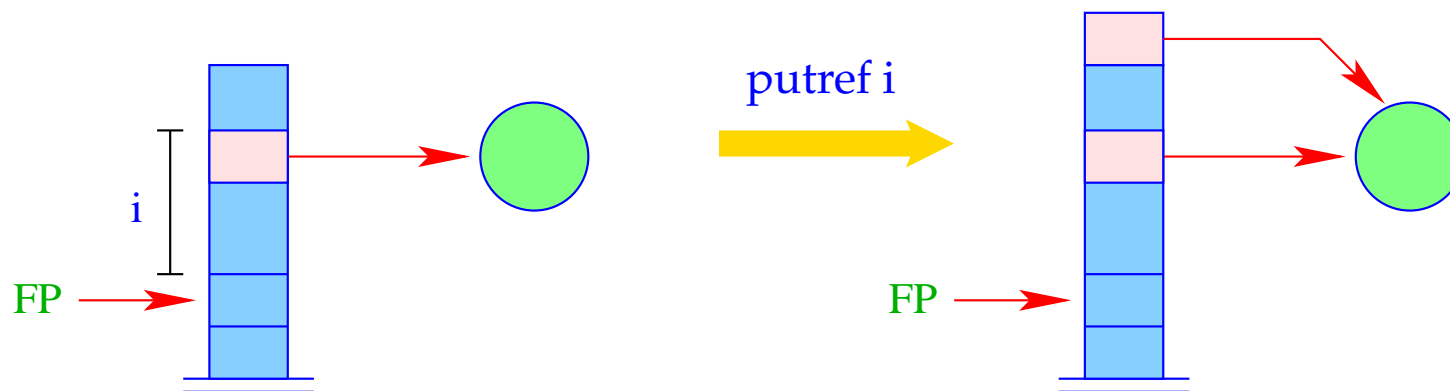
$SP = SP + 1;$
 $S[SP] = \text{new } (R, HP);$

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



$SP = SP + 1;$
 $S[SP] = \text{deref } S[FP + i];$

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

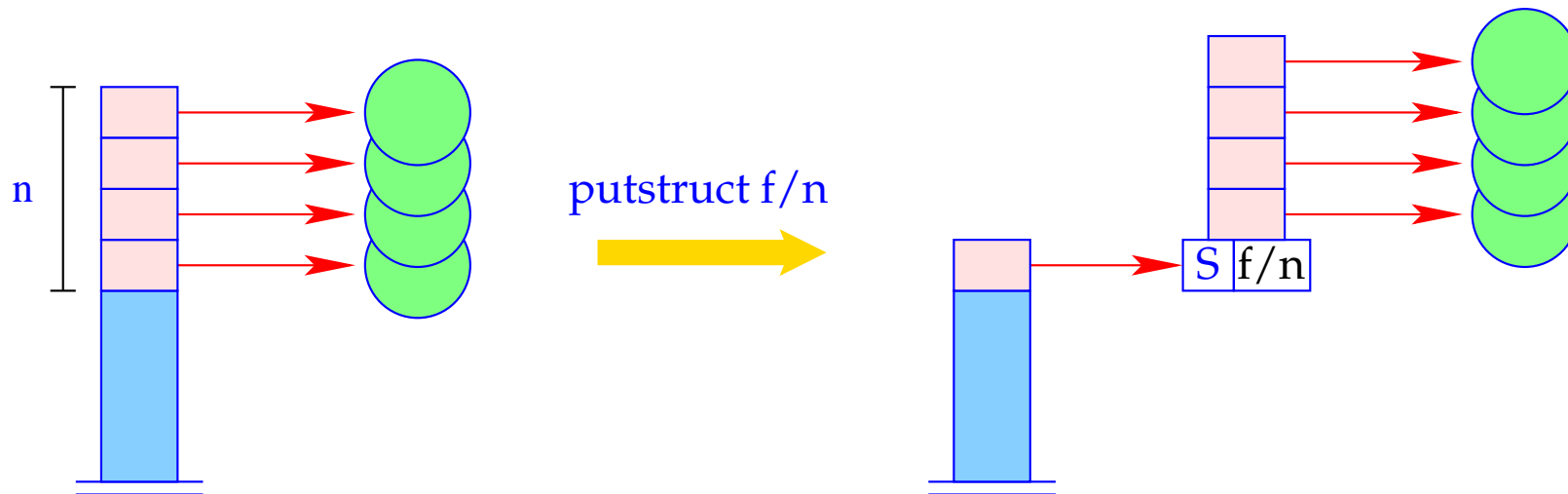


```
SP = SP + 1;  
S[SP] = deref S[FP + i];
```

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 \implies 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:

```

codeG p(t1, ..., tk) ρ =   mark B           // legt einen Kellerrahmen an
                             codeA t1 ρ
                             ...
                             codeA tk ρ
                             call p/k         // ruft Prozedur p/k auf
B : ...

```

```

codeG p(t1, ..., tk) ρ =   mark B           // legt einen Kellerrahmen an
                             codeA t1 ρ
                             ...
                             codeA tk ρ
                             call p/k         // ruft Prozedur p/k auf
B : ...

```

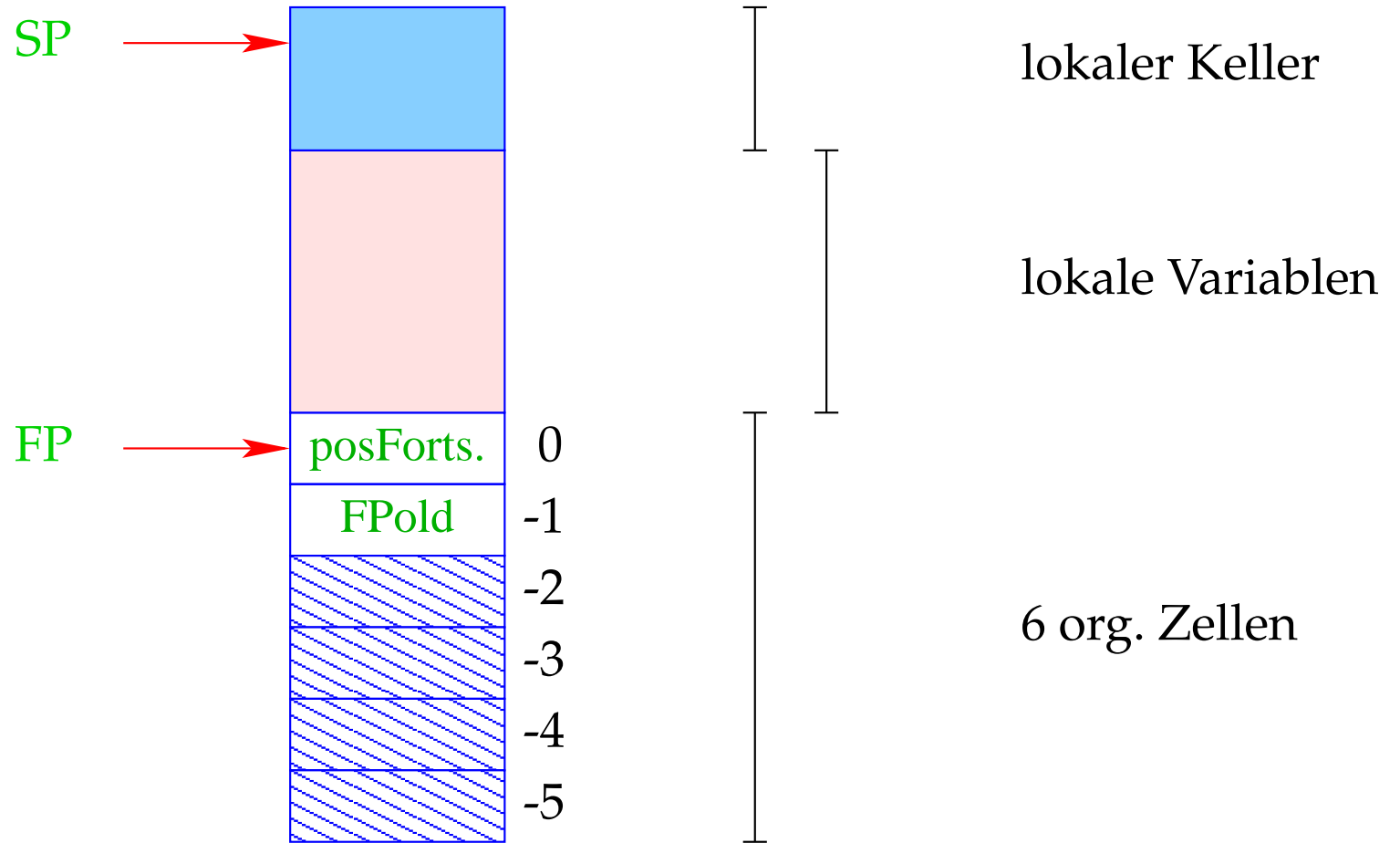
Beispiel: $g \equiv p(a, X, g(\bar{X}, Y))$. Mit $\rho = \{X \mapsto 1, Y \mapsto 2\}$ gibt das:

```

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

```

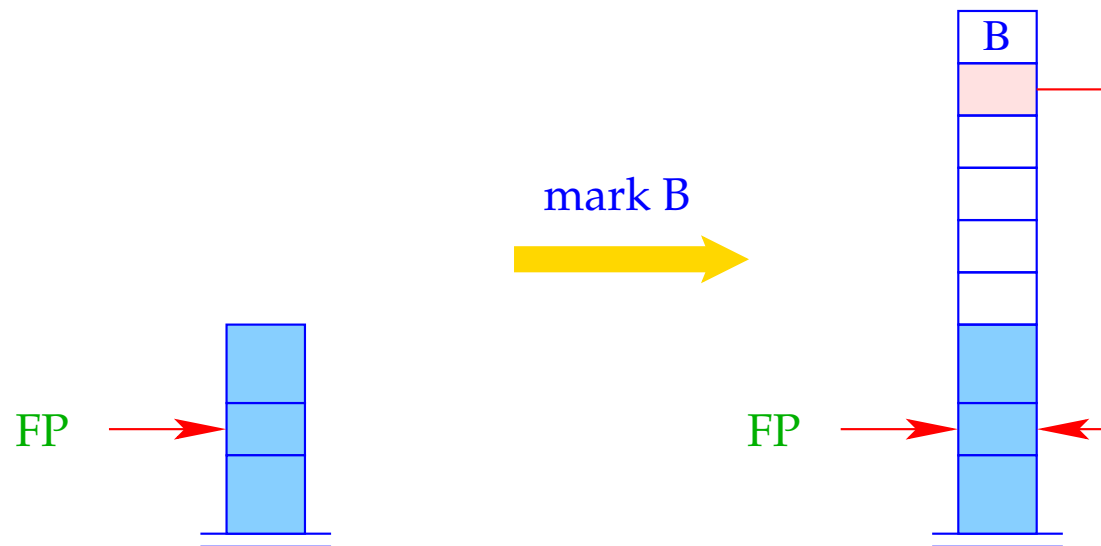
Ein Kellerrahmen der WiM hat den folgenden Aufbau:



Bemerkungen:

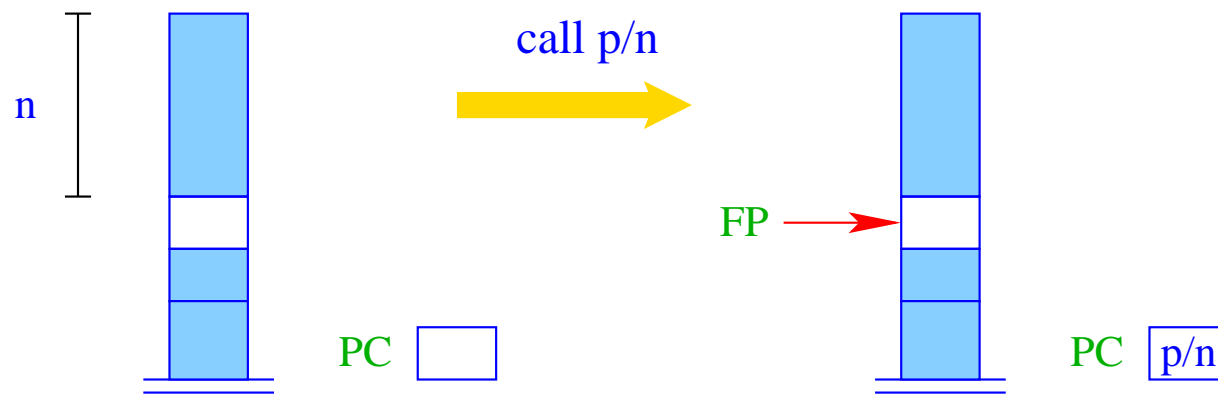
- Die positive Fortsetzungsadresse gibt an, an welcher Stelle im Code fortgefahren werden soll, wenn das Ziel erfolgreich abgearbeitet wurde.
- Die Zelle **FPold** enthält einen Verweis auf den Rahmen des Aufrufers.
- Die zusätzlichen organisatorischen Zellen benötigen wir zur Implementierung des Backtracking \implies 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 $\text{put } \tilde{X} \rho$ ein:

$$\text{put } X \rho = \text{putvar } (\rho X)$$

$$\text{put } _ \rho = \text{putanon}$$

$$\text{put } \bar{X} \rho = \text{putref } (\rho X)$$

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 :-)

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

Beispiel:

Betrachte die Gleichung:

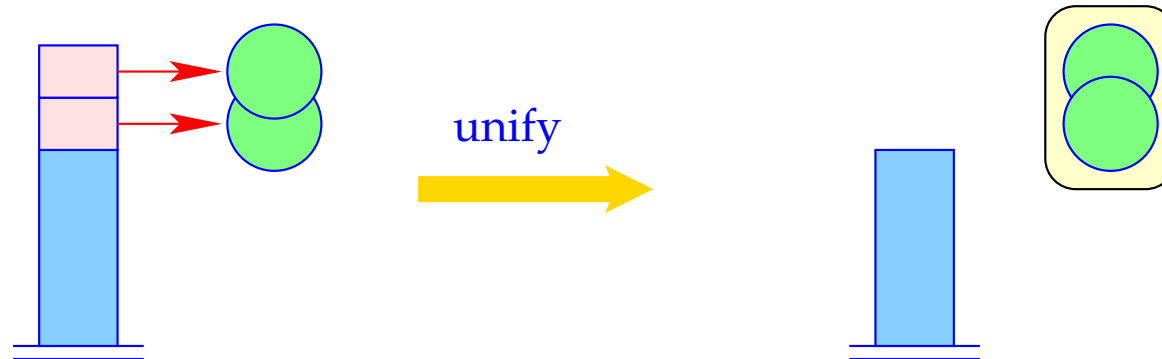
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

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.

```

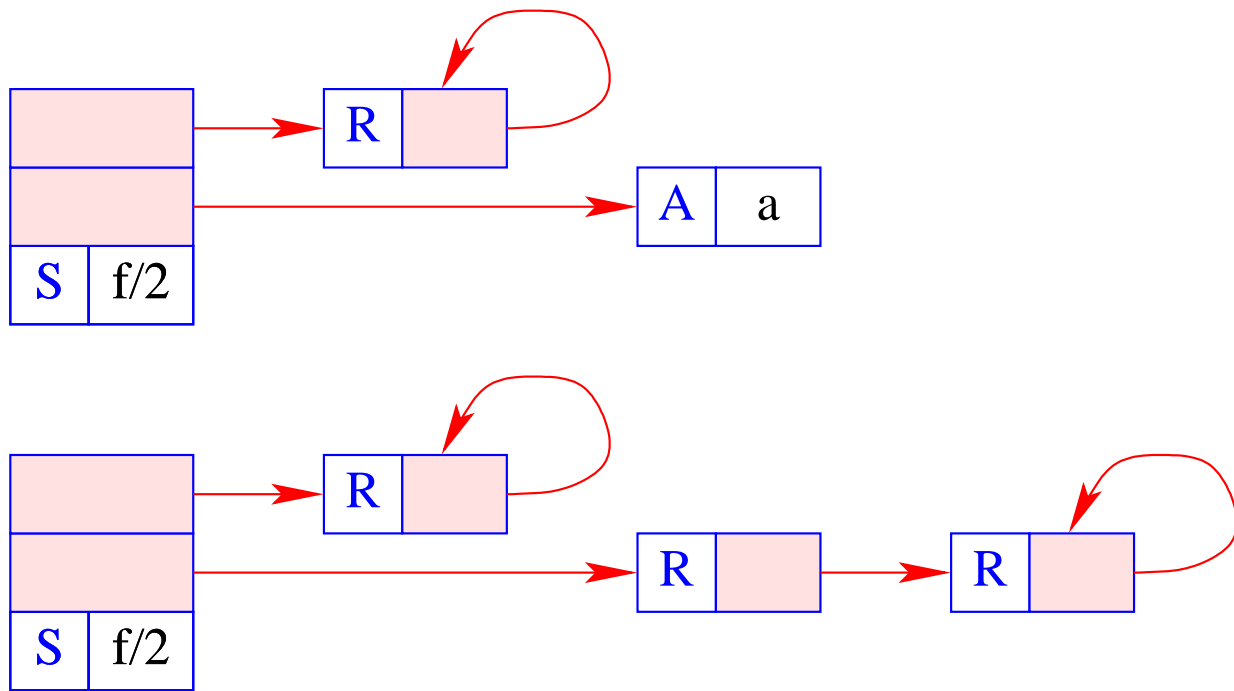
bool unify (ref u, ref v) {
    if (u == v) return true;
    if (H[u] == (R,_)) {
        if (H[v] == (R,_)) {
            if (u>v) {
                H[u] = (R,v); trail (u); return true;
            } else {
                H[v] = (R,u); trail (v); return true;
            }
        }
        } elseif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

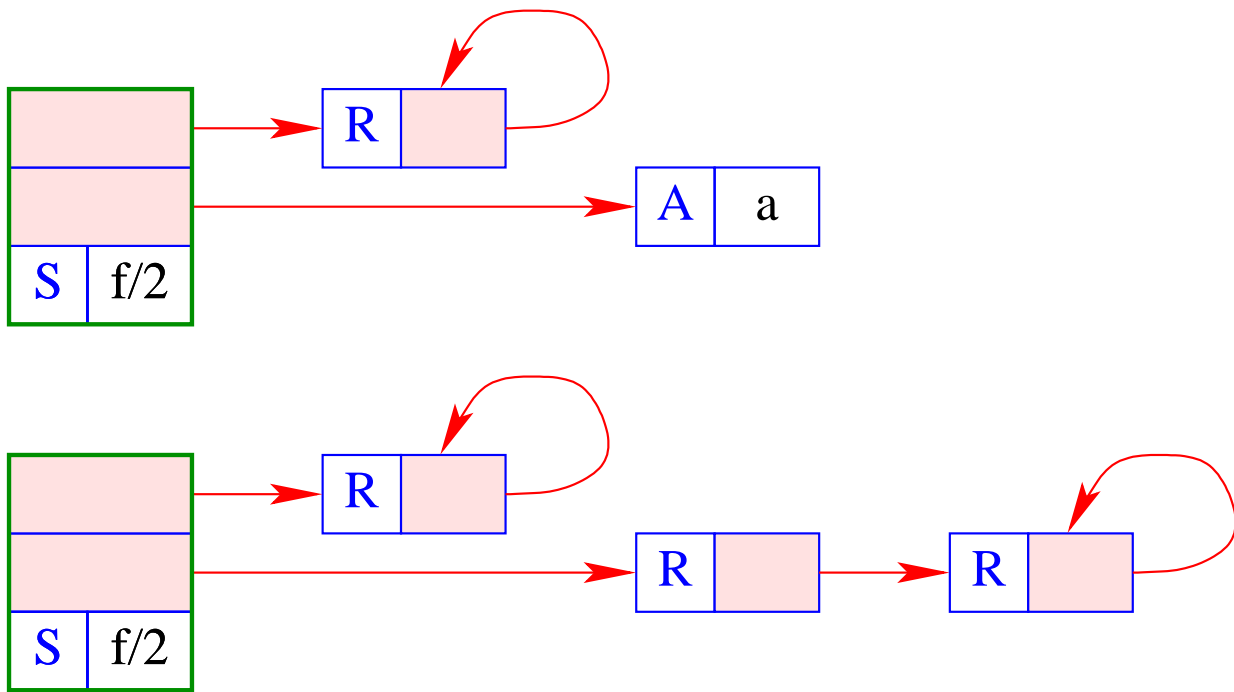
```

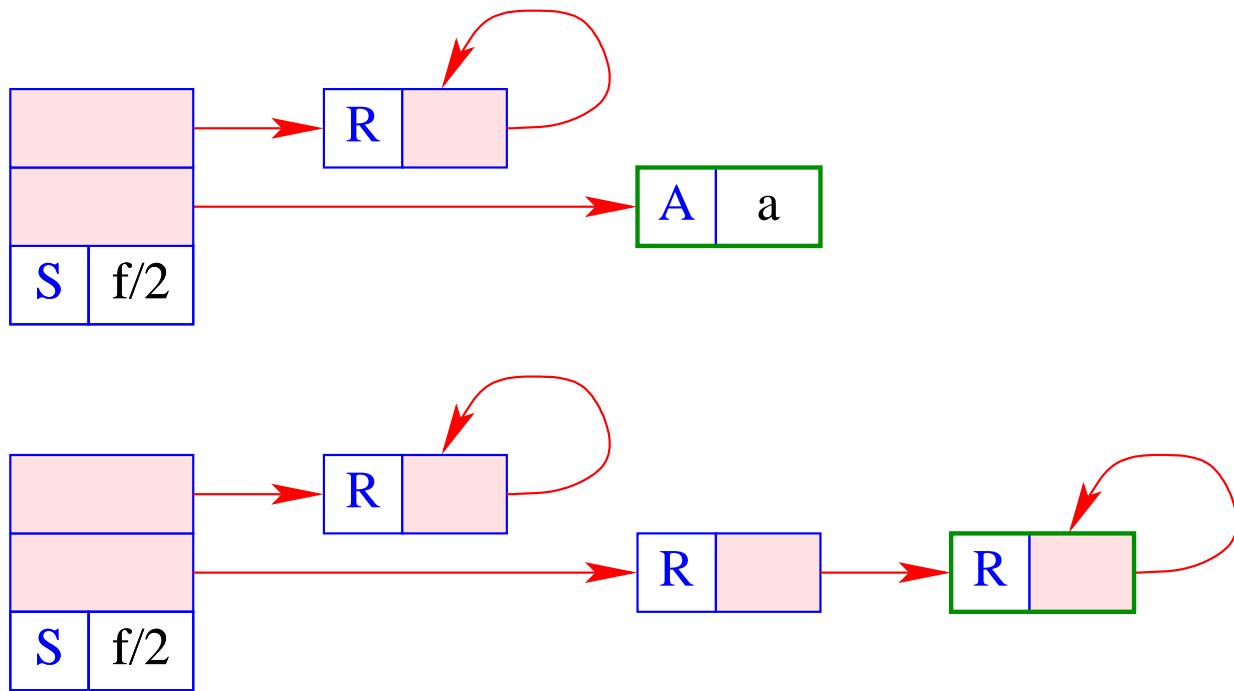
```

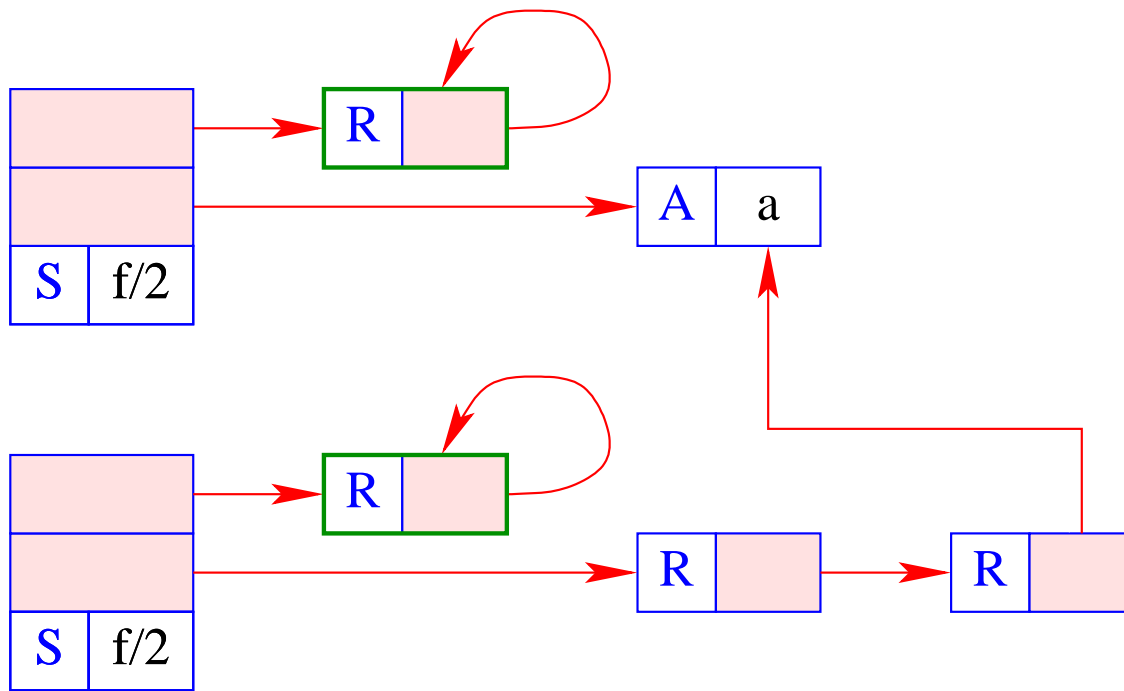
...
if ((H[v] == (R,_)) {
    if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
}
if (H[u]==(A,a) && H[v]==(A,a))
    return true;
if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
    for (int i=1; i<=n; i++)
        if(!unify (deref (H[u+i]),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

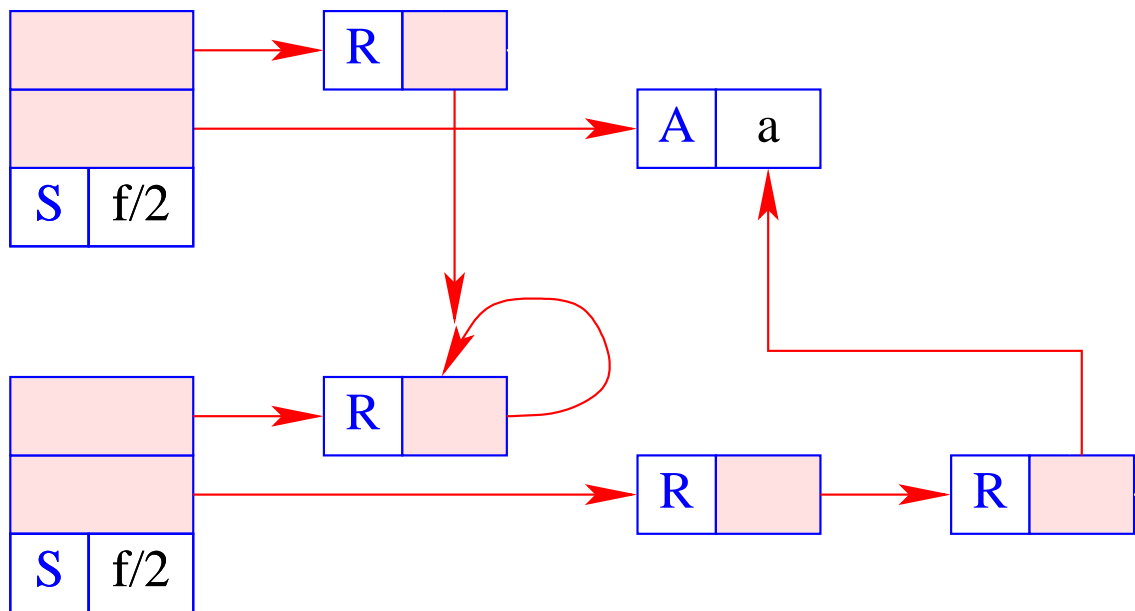
```









- 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 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 !!!