Beispiel:

$$
\begin{array}{lcl}
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Y) \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Z), \mathsf{is\_bigger}(Z, Y) \\
? \quad \mathsf{is\_bigger}(\mathit{elephant}, \mathit{dog}) & &
\end{array}
$$

Ein realistischeres Beispiel:

$$
\begin{array}{lcl}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\
? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & &
\end{array}
$$

Ein realistischeres Beispiel:

$$
\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\
? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & &
\end{array}
$$

Bemerkung:

$[\,]$	$==$	das Atom leere Liste
$[H\|Z]$	$==$	binäre Konstructor-Anwendung
$[a, b, Z]$	$==$	Abkürzung für: $[a\|[b\|[Z\|[\,]]]]$

Ein Programm p ist darum wie folgt aufgebaut:

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

- 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 $p(X_1, \ldots, X_k)$ 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 Berechnungschritt
Bindung von Variablen	$=\!=\!=$	Seiteneffekt

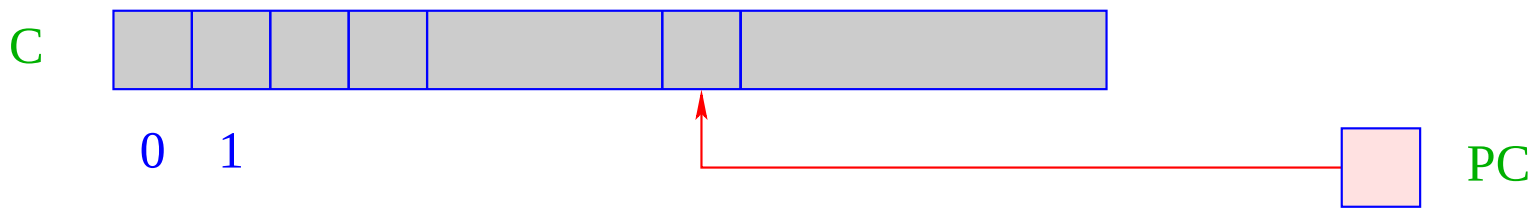
Achtung: Prädikat-Aufrufe ...

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

$\Longrightarrow$ backtracking

27 Architektur der WiM:

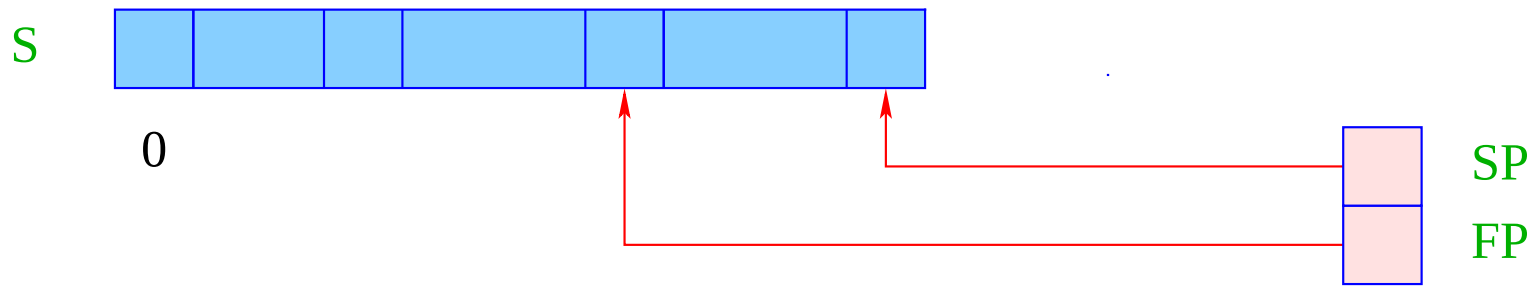
Der Code-Speicher:



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

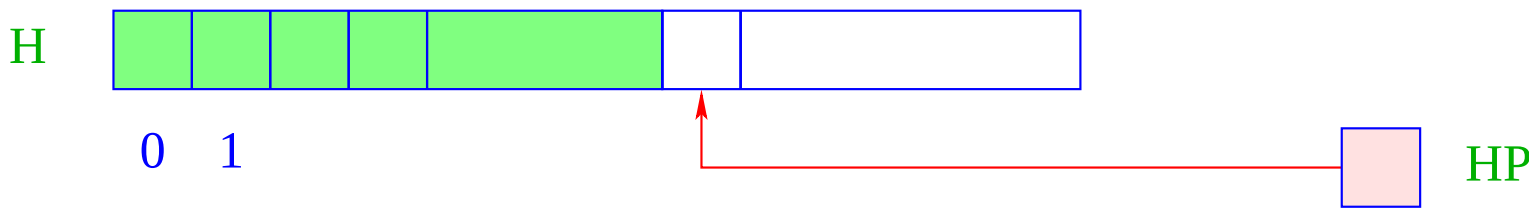
PC = Program Counter – zeigt auf nächsten auszuführenden Befehl.

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.

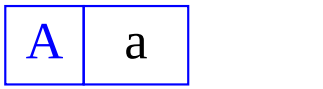
Der Heap:



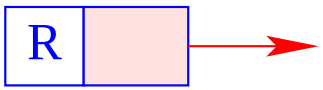
H = Heap für dynamisch erzeugte Terme;

HP = Heap-Pointer – zeigt auf die erste freie Zelle.

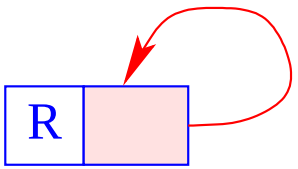
- Der Heap wird ebenfalls wie ein Keller verwaltet :-)
- Ein new-Befehl allokiert ein Objekt in H.
- Objekte sind mit ihrem Typ markiert (wie bei der MaMa) ...



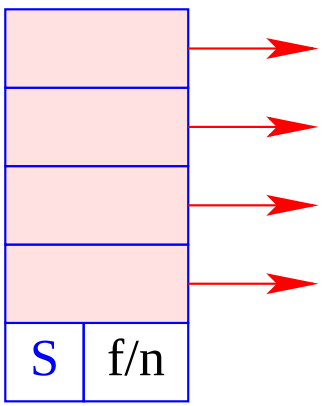
Atom 1 Zelle



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.

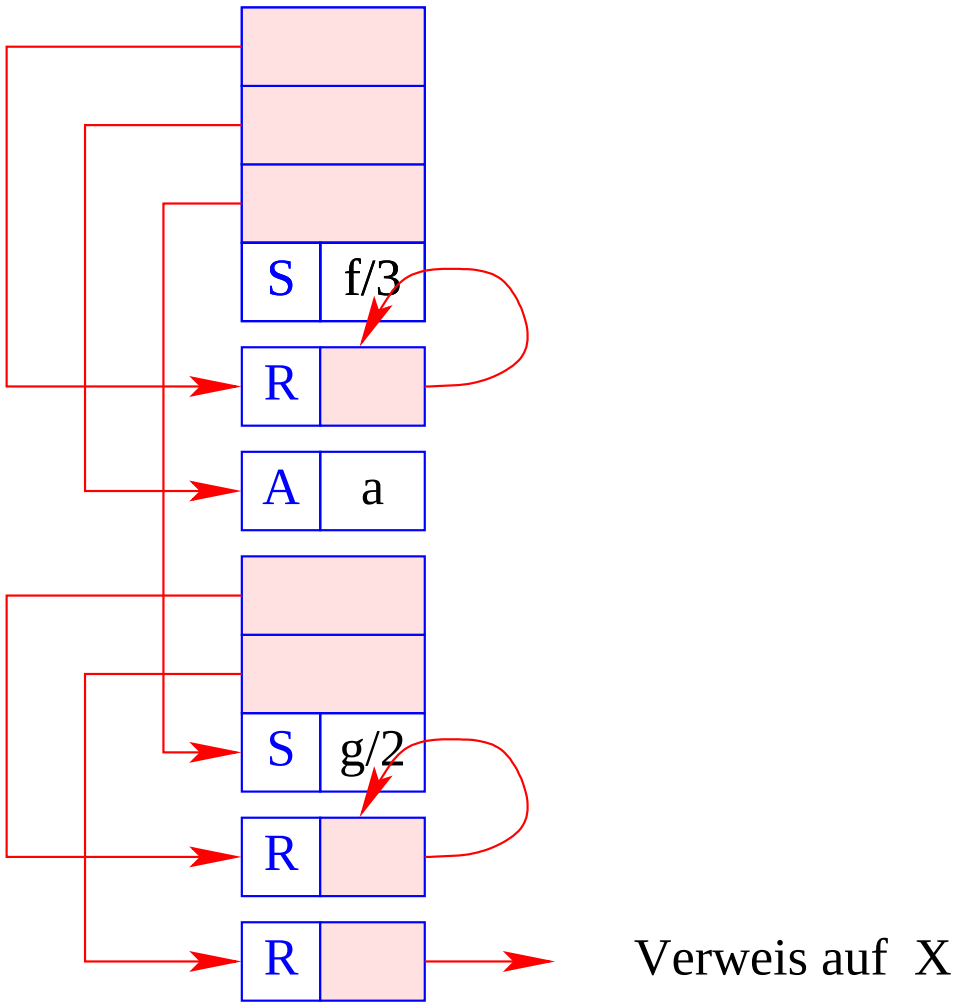
Dann wollen wir für einen Term t eine Folge $\text{code}_A\ t\ \rho$ von Instruktionen erzeugen, die (eine Repräsentation von) t im Heap aufbaut.

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:


S
f/3
R
A
a
S
g/2
R
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:

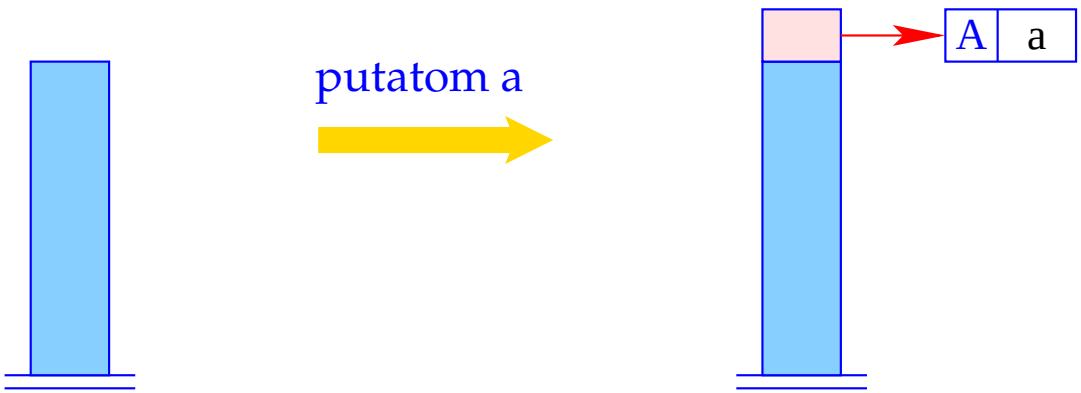
$$
\begin{array}{lcl}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\ X) \\
\text{code}_A\ \bar{X}\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

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

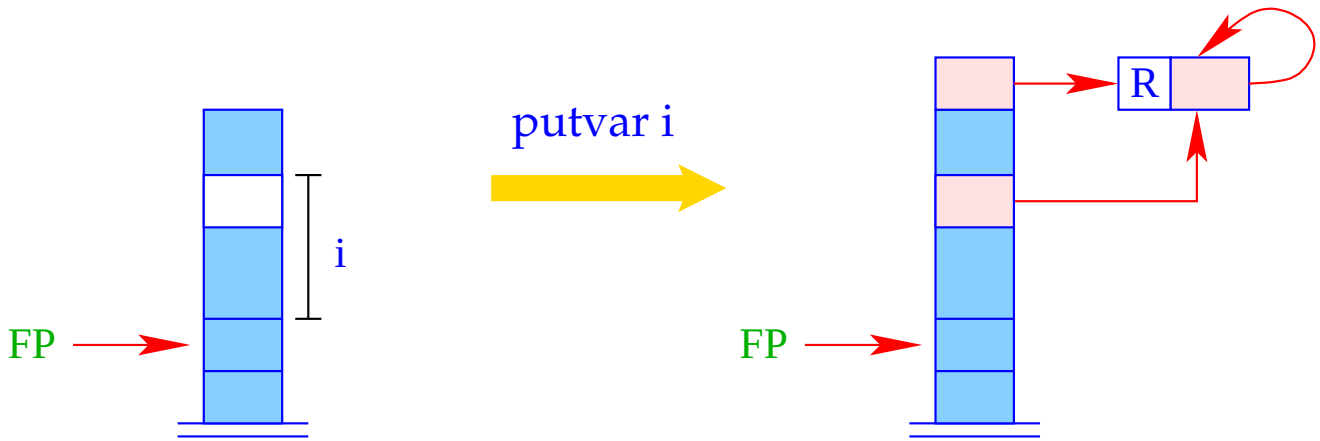
Dabei bedeuten die Befehle:

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



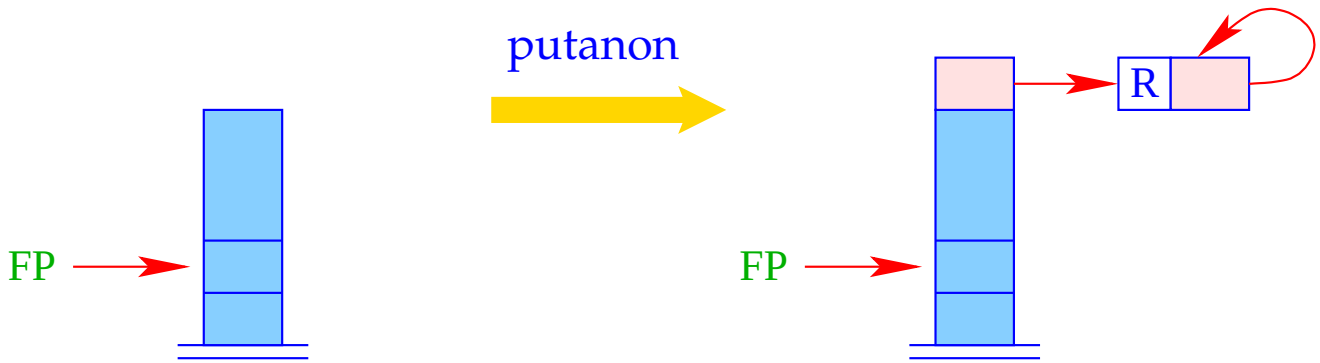
SP++; S[SP] = new (A,a);

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



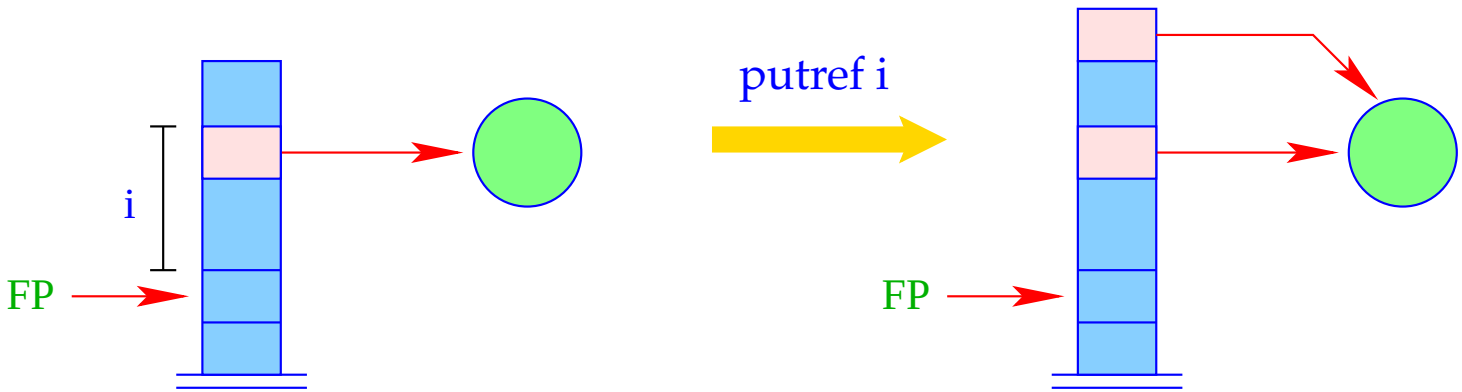
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];

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



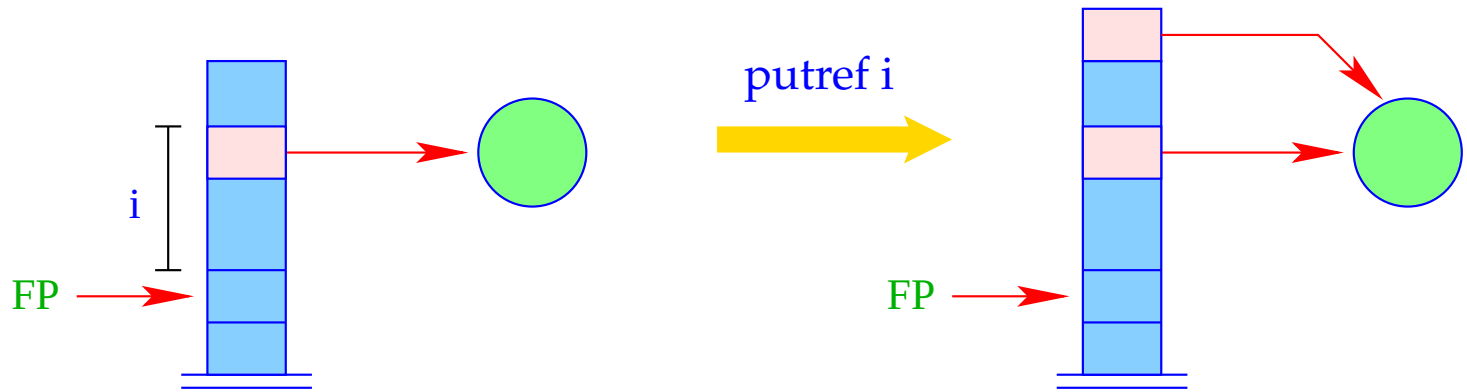
SP = SP + 1;
S[SP] = new (R, HP);

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:

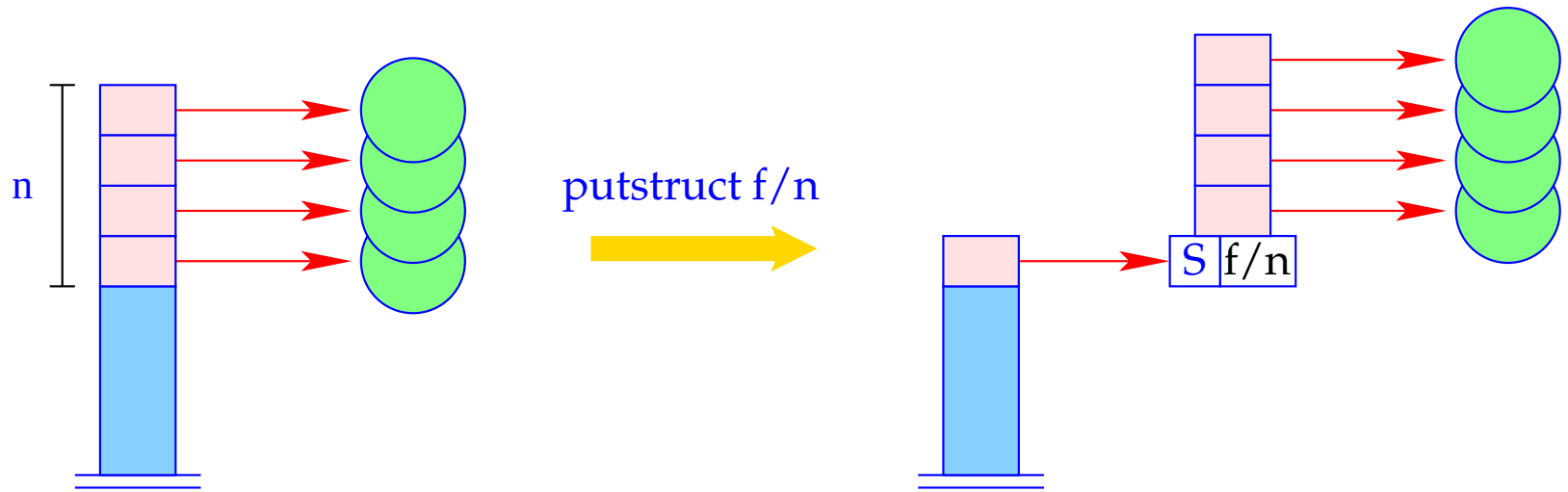


SP = SP + 1;
S[SP] = deref S[FP + i];

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:

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

$\text{code}_G\ p(t_1, \ldots, t_k)\ \rho \quad = \quad$ mark B // legt einen Kellerrahmen an

$\text{code}_A\ t_1\ \rho$

$\ldots$

$\text{code}_A\ t_k\ \rho$

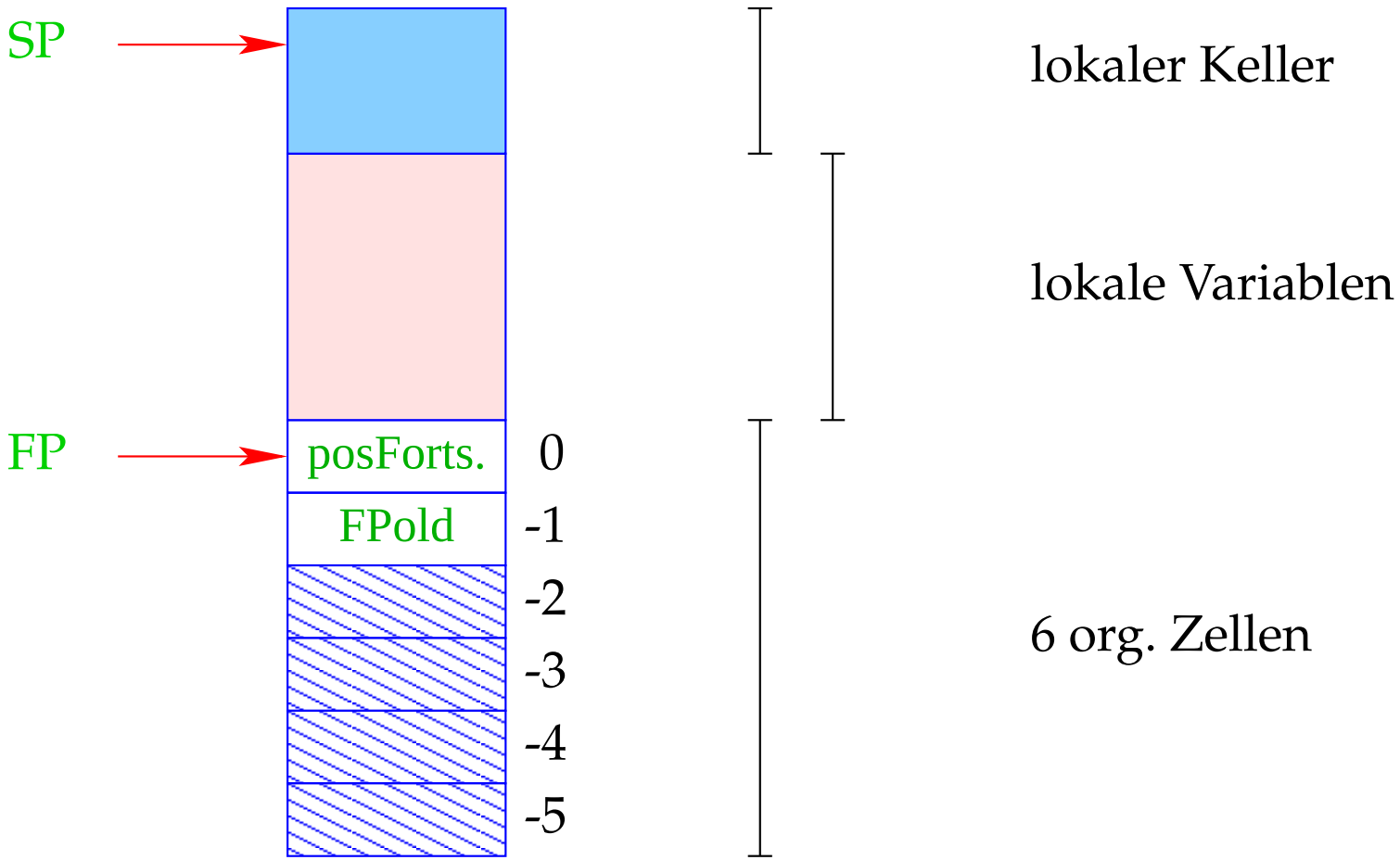
call p/k // ruft Prozedur p/k auf

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			

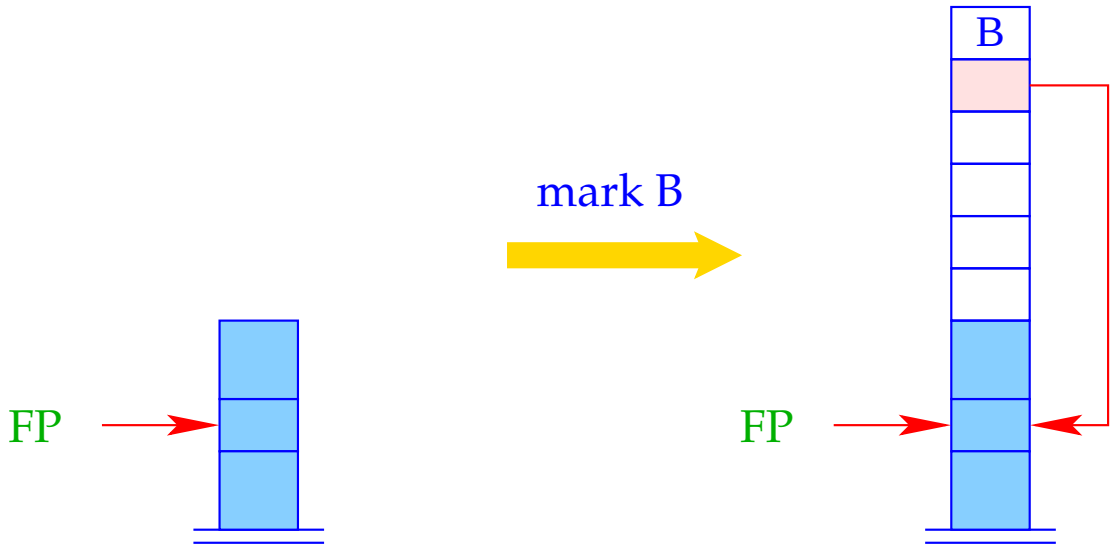
Ein Kellerrahmen der WiM hat den folgenden Aufbau:


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

Bemerkungen:

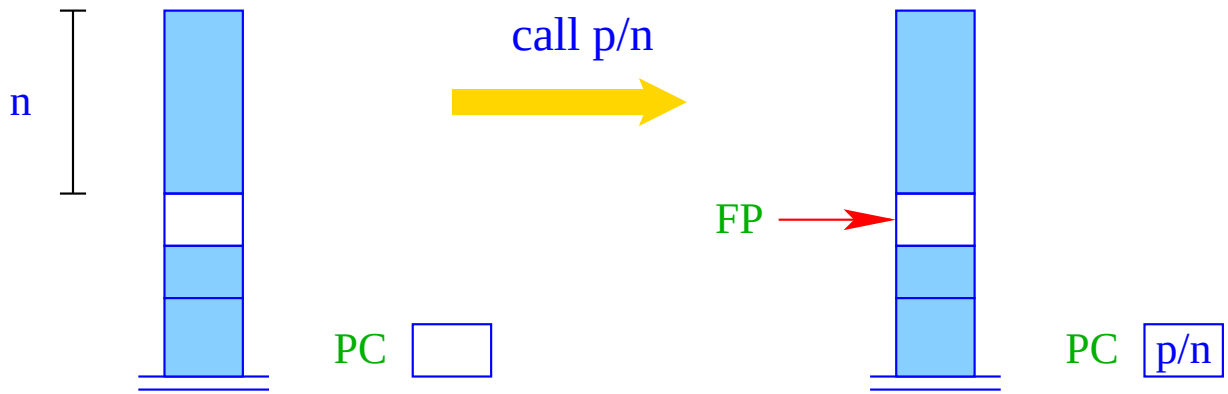
- Die positive Fortsetzungsadresse gibt an, an welcher Stelle im Code fortgefahren werden soll, wenn das Ziel erfolgreich abgearbeitet wurde.
- Die Zelle FPold enthält einen Verweis auf den Rahmen des Aufrufers.
- Die zusätzlichen organisatorischen Zellen benötigen wir zur Implementierung des Backtracking $\Longrightarrow$ 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:

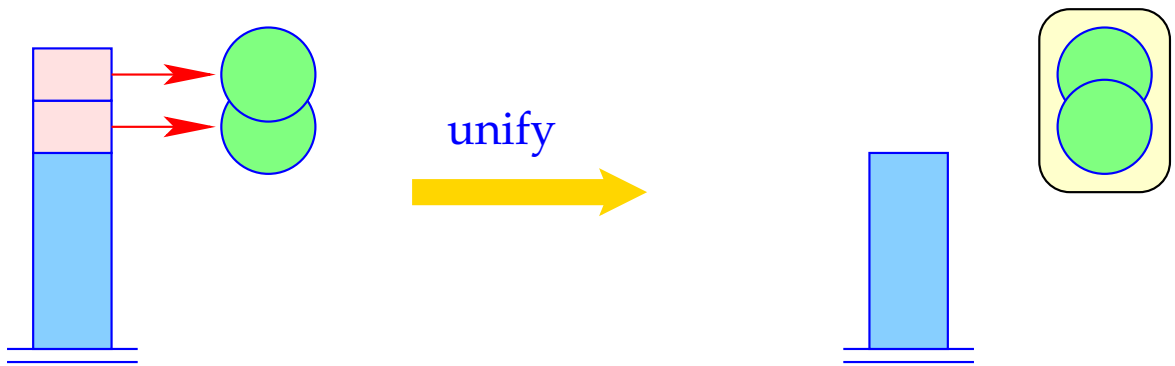
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

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.