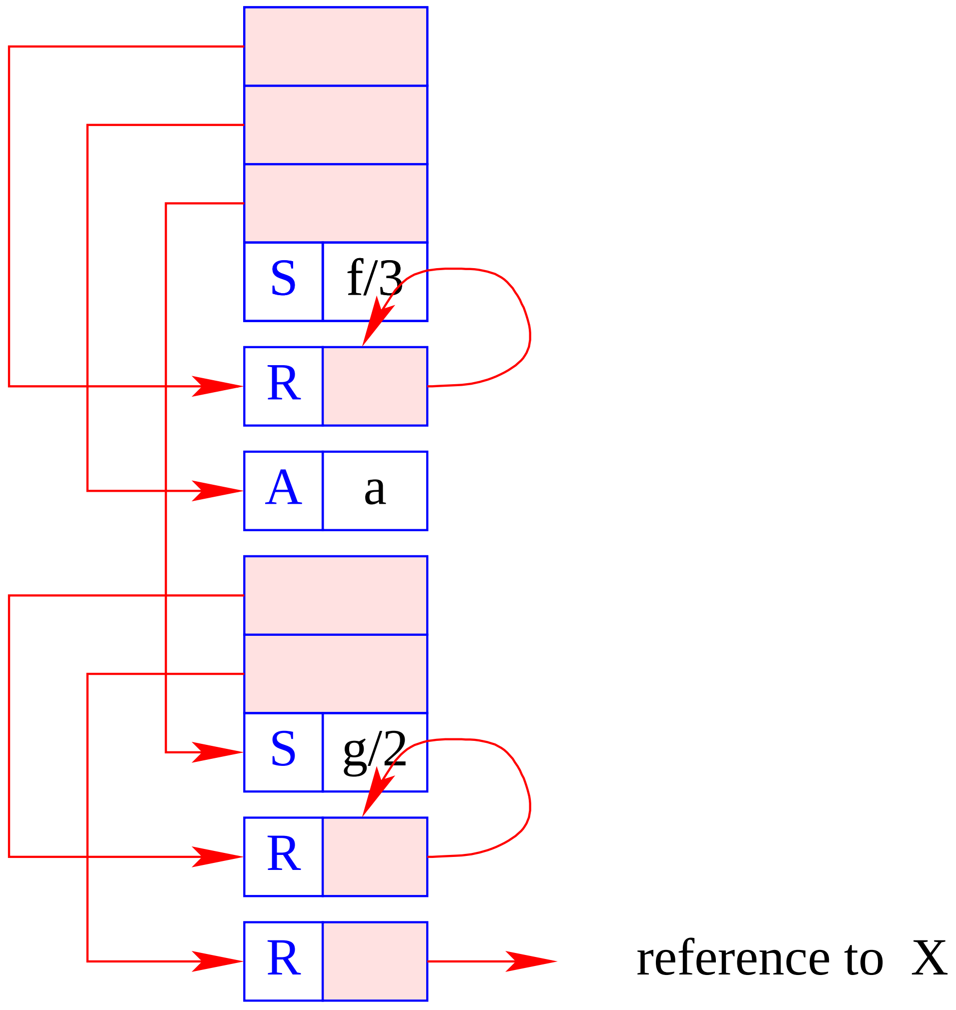
Representing $t \equiv f(g(X, Y), a, Z)$:



For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

Then we define:

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

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

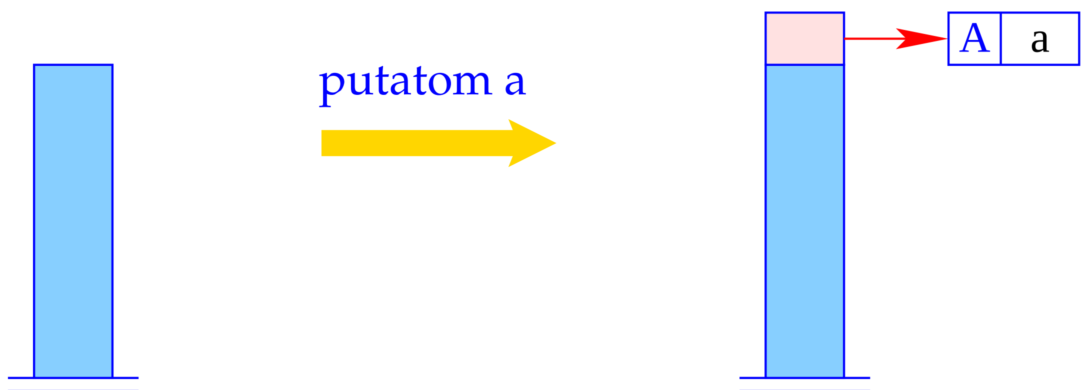
Then we define:

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

For $f(g(\overline{X}, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ this results in the sequence:

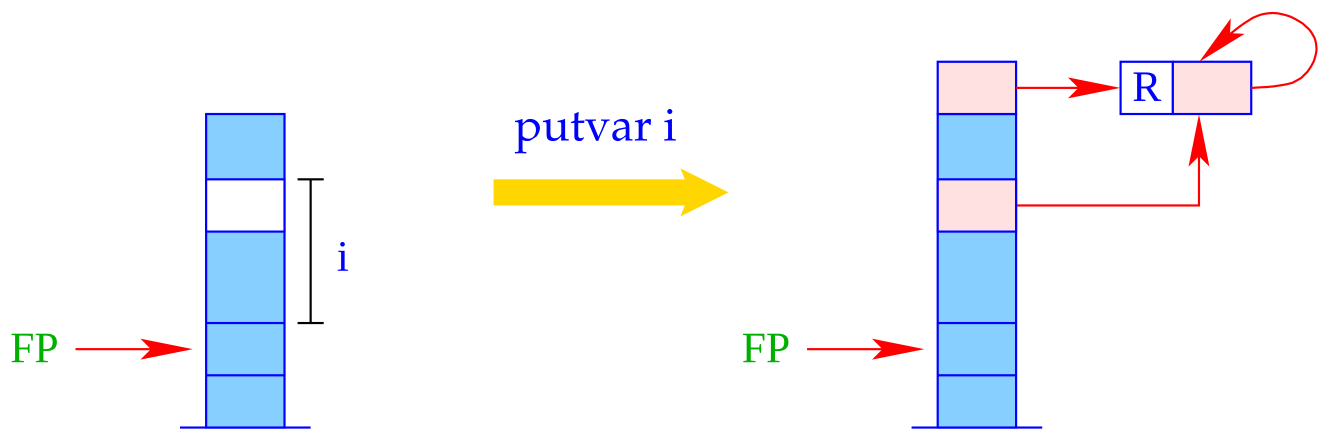
putref 1	putatom a
putvar 2	putvar 3
putstruct g/2	putstruct f/3

The instruction putatom a constructs an atom in the heap:



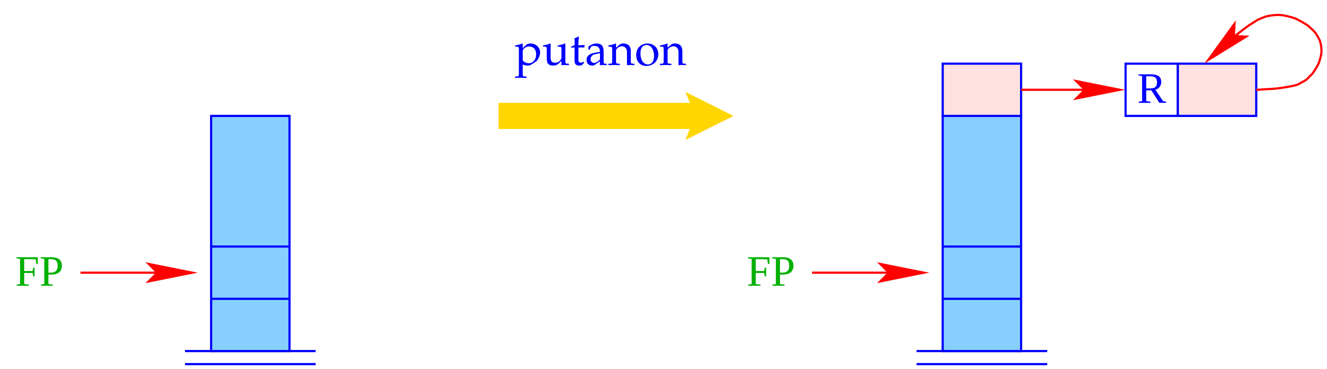
SP++; S[SP] = new (A,a);

The instruction putvar i introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:



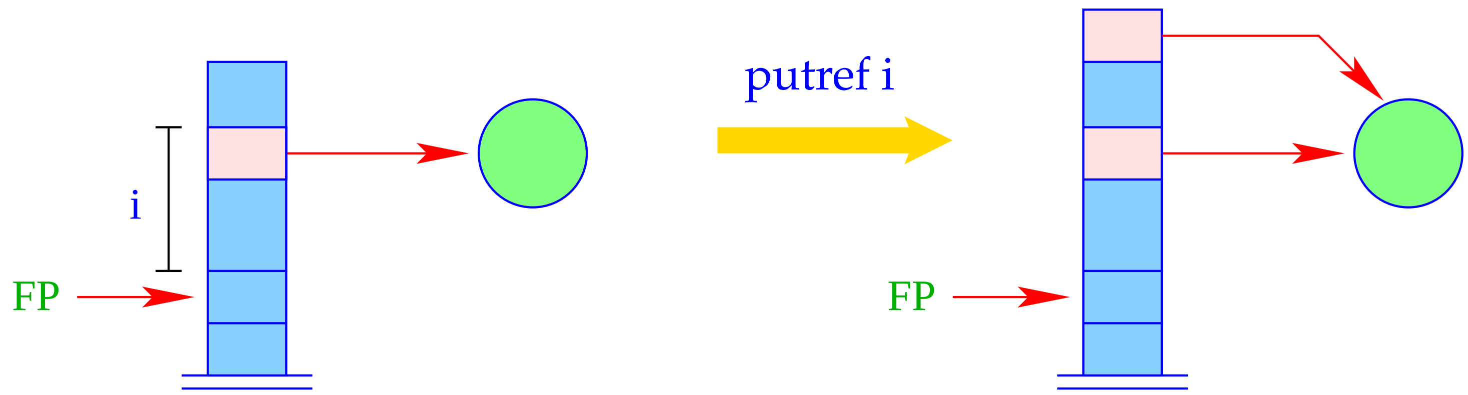
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];

The instruction putanon introduces a new unbound variable but does not store a reference to it in the stack frame:



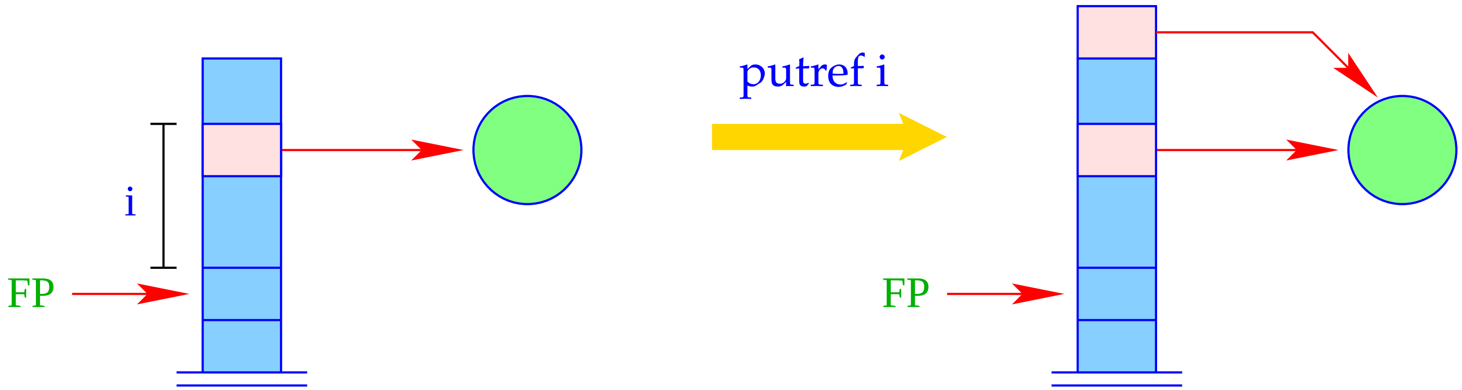
SP = SP + 1;
S[SP] = new (R, HP);

The instruction putref i pushes the value of the variable onto the stack:



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

The instruction putref i pushes the value of the variable onto the stack:

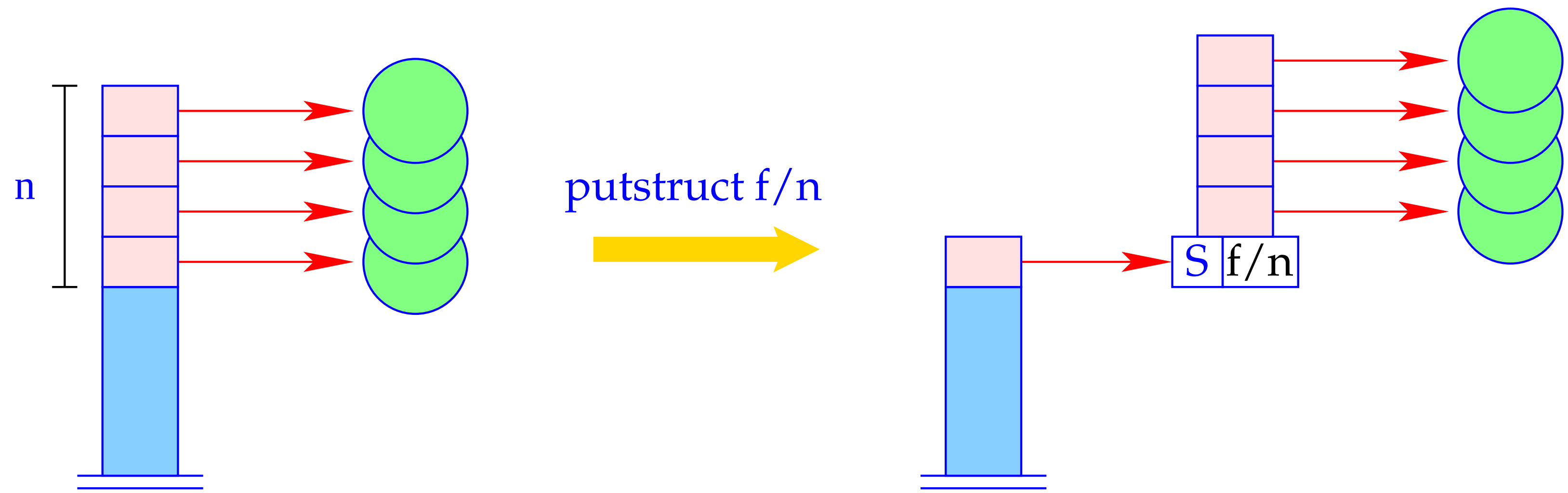


SP = SP + 1;
S[SP] = deref S[FP + i];

The auxiliary function deref contracts chains of references:

```
ref deref (ref v) {
      if (H[v]==(R,w) && v!=w) return deref (w);
      else return v;
}
```

The instruction putstruct f/n builds a constructor application in the heap:



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;

Remarks:

- The instruction putref i does not just push the reference from $S[FP + i]$ onto the stack, but also dereferences it as much as possible

  $\Longrightarrow$ maximal contraction of reference chains.

- In constructed terms, references always point to smaller heap addresses.

  Also otherwise, this will be often the case. Sadly enough, it cannot be guaranteed in general :-(

29 The Translation of Literals (Goals)

Idea:

- Literals are treated as procedure calls.
- We first allocate a stack frame.
- Then we construct the actual parameters (in the heap)
- ... and store references to these into the stack frame.
- Finally, we jump to the code for the procedure/predicate.

$$
\begin{array}{rcll}
\mathrm{code}_G\ p(t_1,\ldots,t_k)\ \rho & = & \mathrm{mark\ B} & \text{// allocates the stack frame} \\
 & & \mathrm{code}_A\ t_1\ \rho & \\
 & & \ldots & \\
 & & \mathrm{code}_A\ t_k\ \rho & \\
 & & \mathrm{call\ p/k} & \text{// calls the procedure p/k} \\
 & \mathrm{B:} & \ldots &
\end{array}
$$

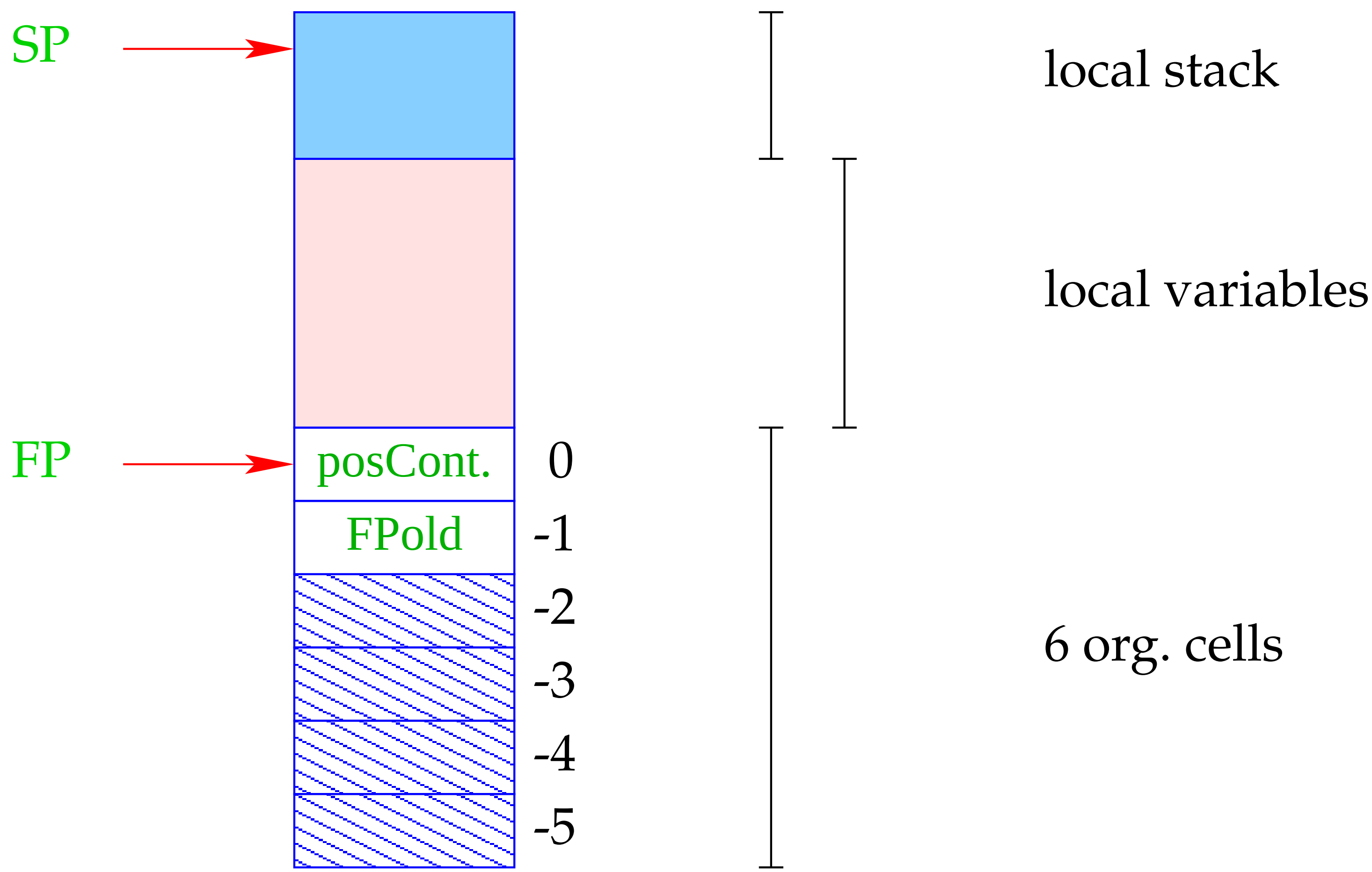
$$
\begin{array}{rrll}
\text{code}_G\ p(t_1,\ldots,t_k)\ \rho & = & \text{mark B} & \text{// allocates the stack frame} \\
 & & \text{code}_A\ t_1\ \rho & \\
 & & \ldots & \\
 & & \text{code}_A\ t_k\ \rho & \\
 & & \text{call p/k} & \text{// calls the procedure p/k} \\
 & \text{B :} & \ldots &
\end{array}
$$

Example: $p(a, X, g(\bar{X}, Y))$ with $\rho = \{X \mapsto 1, Y \mapsto 2\}$

We obtain:

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

Stack Frame of the WiM:

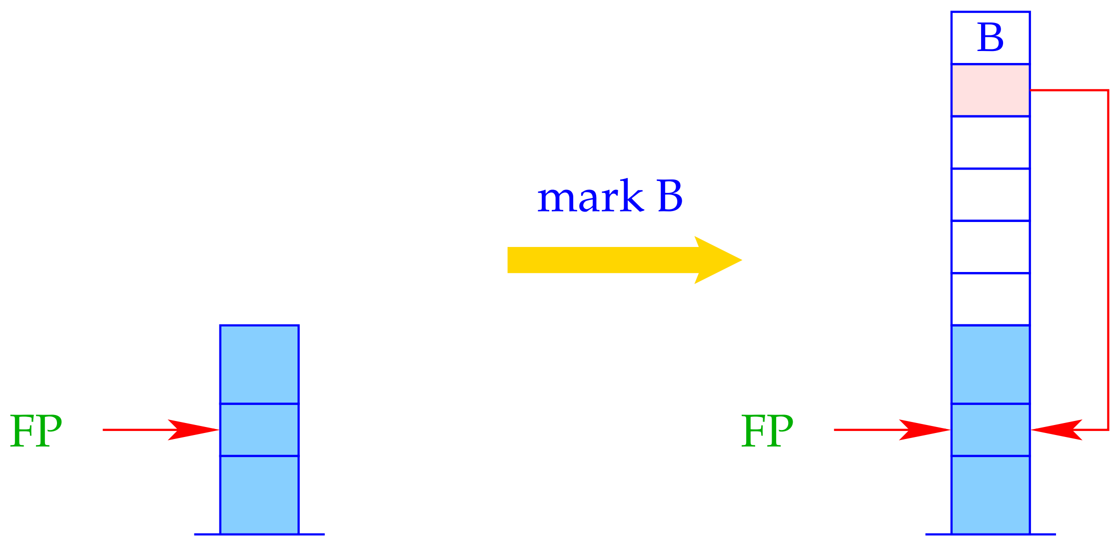


Remarks:

- The positive continuation address records where to continue after successful treatment of the goal.
- Additional organizational cells are needed for the implementation of backtracking

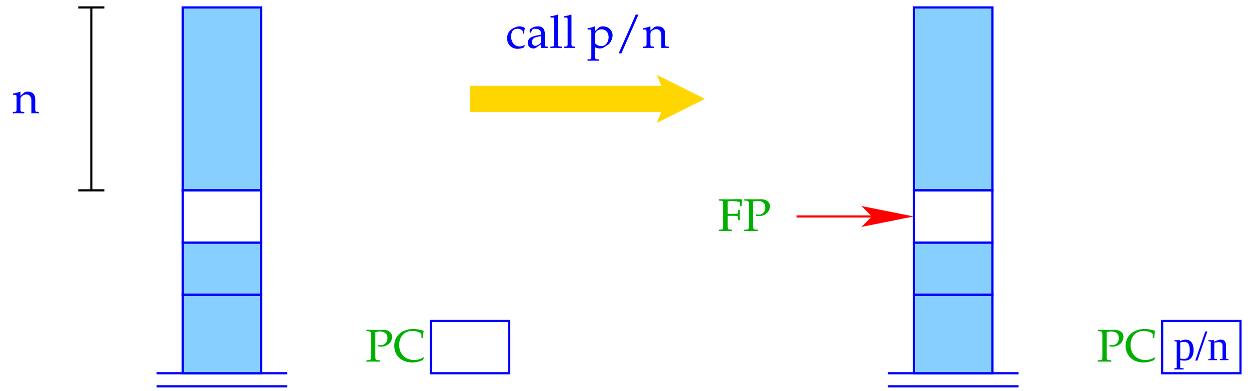
$\Longrightarrow$ will be discussed at the translation of predicates.

The instruction mark B allocates a new stack frame:



SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

The instruction call p/n calls the n-ary predicate p :



FP = SP - n;
PC = p/n;

30 Unification

Convention:

- By $\tilde{X}$, we denote an occurrence of X;
  it will be translated differently depending on whether the variable is initialized or not.
- We introduce the macro $\text{put}\ \tilde{X}\ \rho$:

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

Let us translate the unification $\tilde{X} = t$.

Idea 1:

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

Let us translate the unification $\tilde{X} = t$.

Idea 1:

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

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

Example:

Consider the equation:

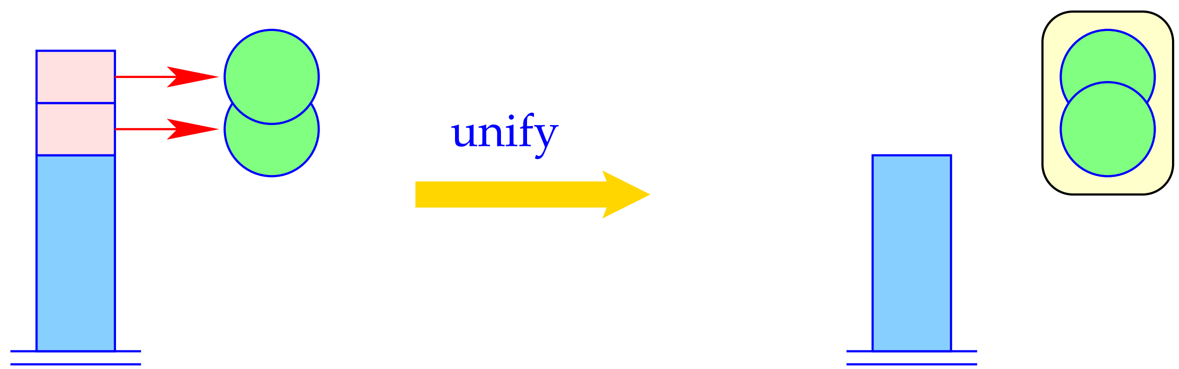
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Then we obtain for an address environment

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

putref 4	putref 1	putatom a	unify
	putvar 2	putvar 3	
	putstruct g/2	putstruct f/3	

The instruction unify calls the run-time function `unify()` for the topmost two references:



unify (S[SP-1], S[SP]);
SP = SP−2;

The Function `unify()`

- ... takes two heap addresses.

  For each call, we guarantee that these are maximally de-referenced.

- ... checks whether the two addresses are already identical.

  If so, does nothing :-)

- ... binds younger variables (larger addresses) to older variables (smaller addresses);

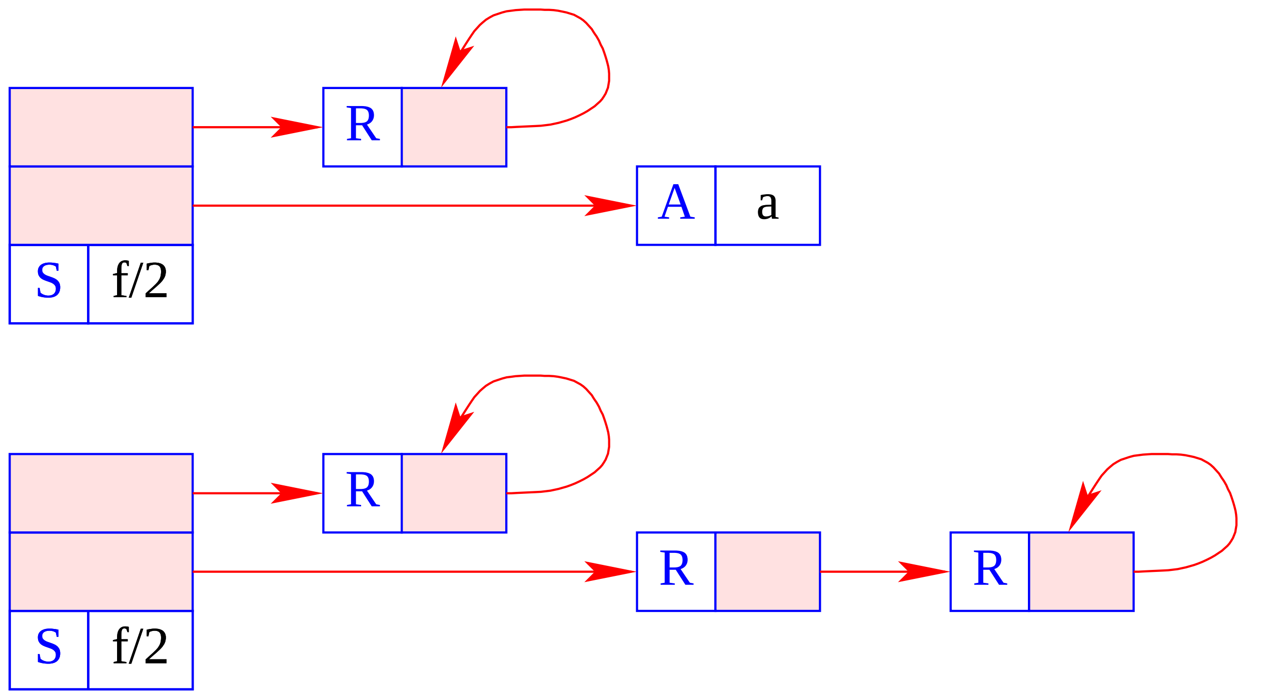
- ... when binding a variable to a term, checks whether the variable occurs inside the term $\Longrightarrow$ occur-check;

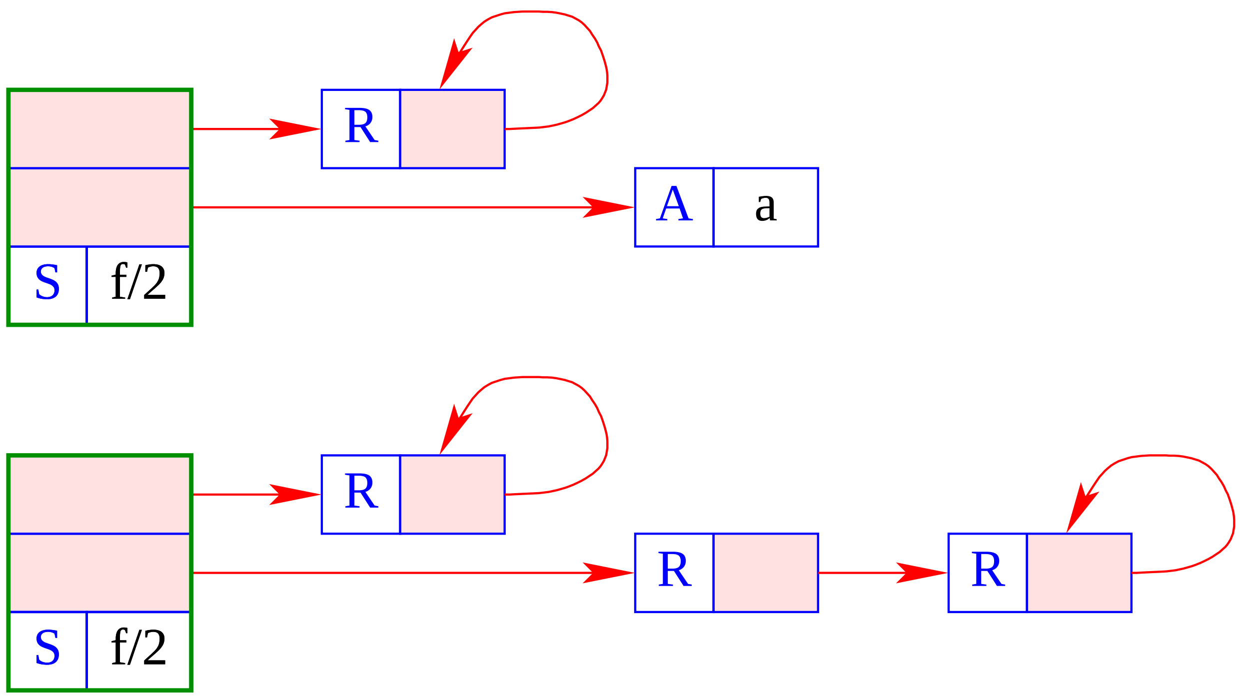
- ... records newly created bindings;

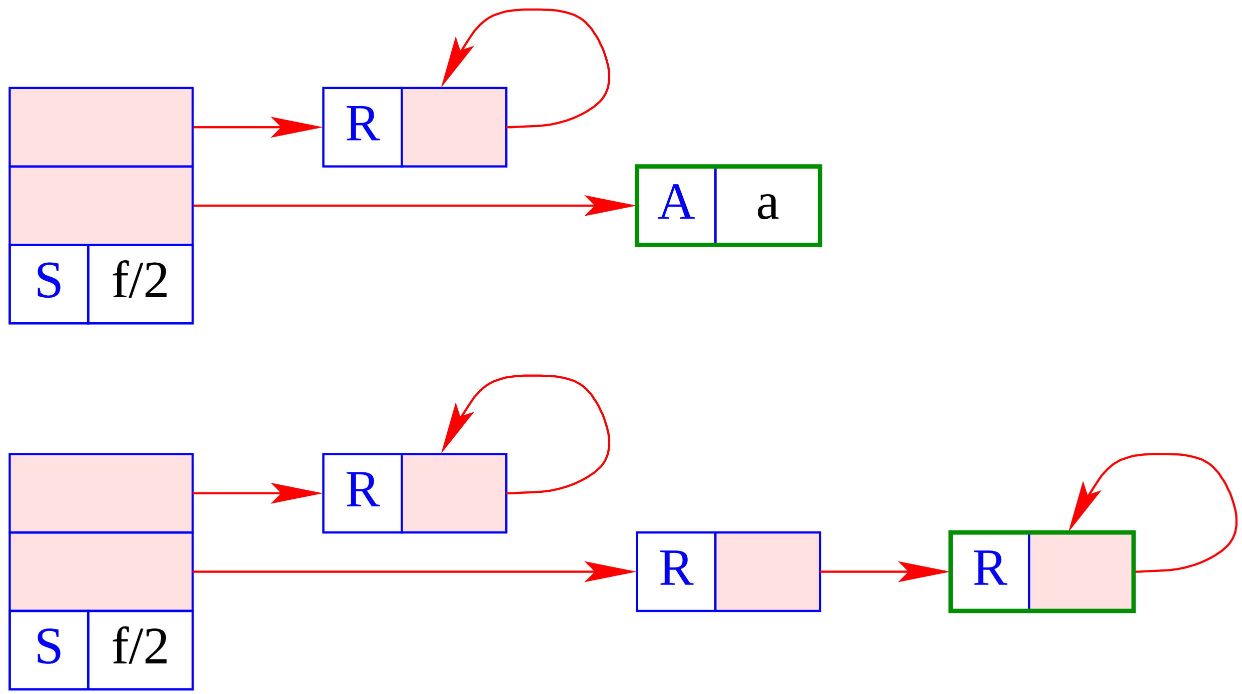
- ... may fail. Then backtracking is initiated.

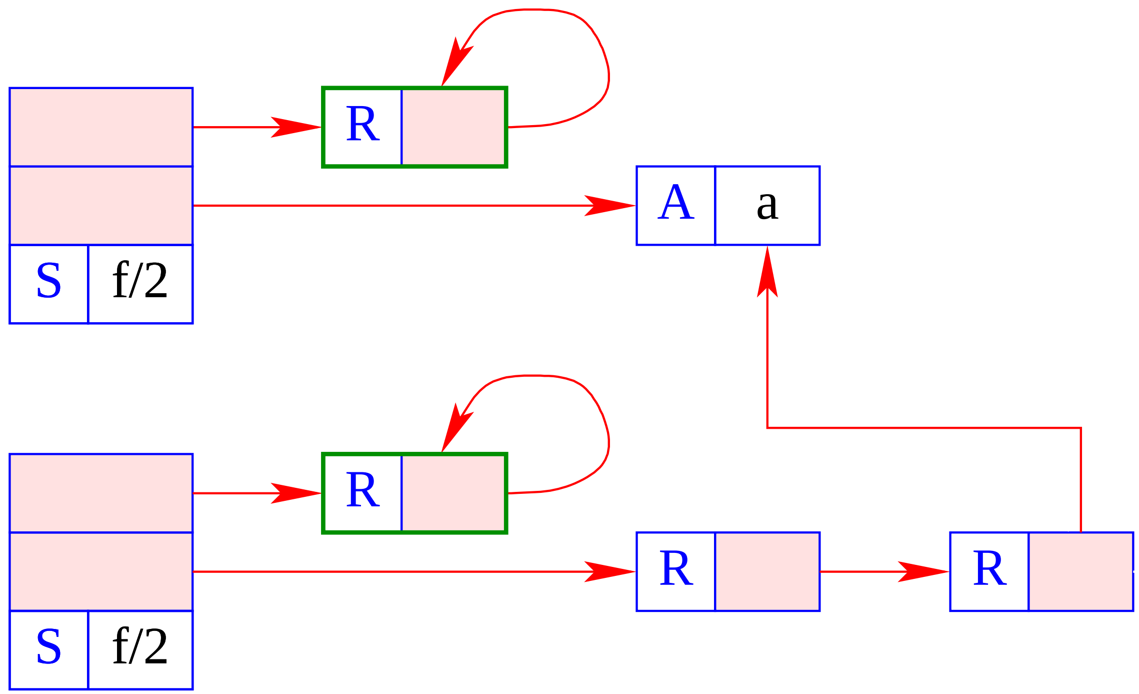
```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u]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

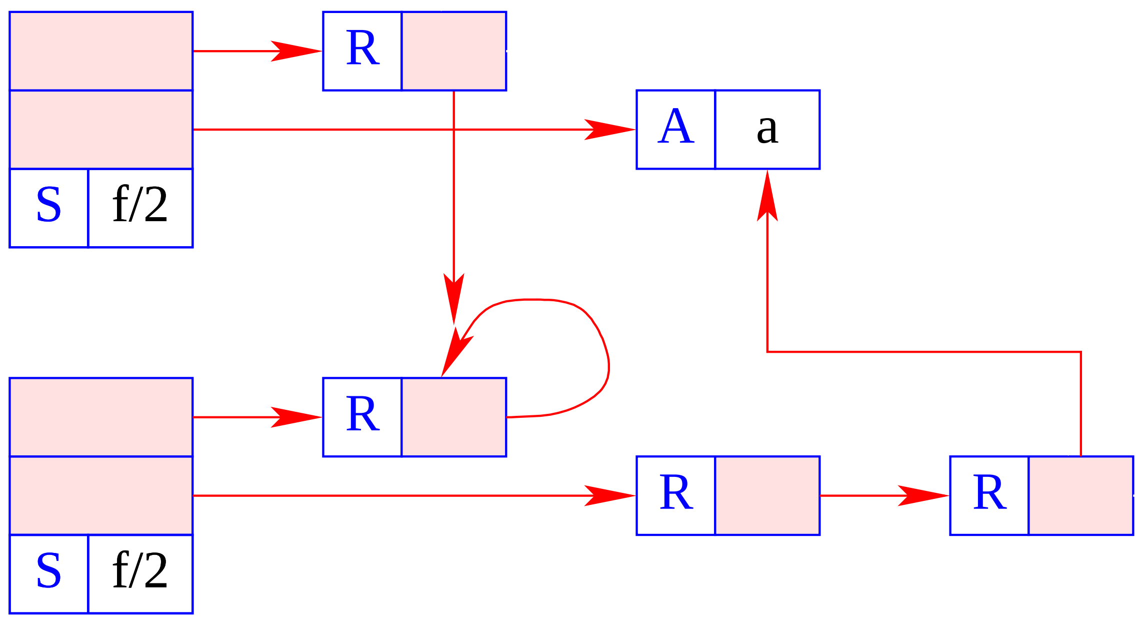
```
...
 if ((H[v] == (R,_)) {
      if (check (v,u)) {
         H[v] = (R,u); trail (v); return true;
      } else {
         backtrack(); return false;
      }
   }
   if (H[u]==(A,a) && H[v]==(A,a))
      return true;
   if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
      for (int i=1; i<=n; i++)
     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

- The run-time function `trail()` records the a potential new binding.
- The run-time function `backtrack()` initiates backtracking.
- The auxiliary function `check()` performs the occur-check: it tests whether a variable (the first argument) occurs inside a term (the second argument).
- Often, this check is skipped, i.e.,

```
bool check (ref u, ref v) { return true;}
```

Otherwise, we could implement the run-time function `check()` as follows:

```
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;
}
```

Discussion:

- The translation of an equation $\tilde{X} = t$ is very simple :-)
- Often the constructed cells immediately become garbage :-(

Idea 2:

- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of t whenever possible !
- Translate each node of t into an instruction which performs the unifcation with this node !!

Discussion:

- The translation of an equation $\tilde{X} = t$ is very simple :-)
- Often the constructed cells immediately become garbage :-(

Idea 2:

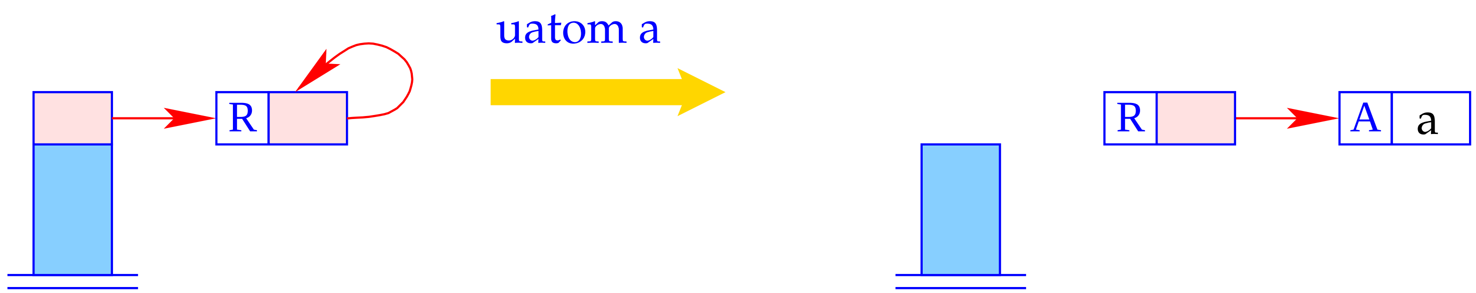
- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of t whenever possible !
- Translate each node of t into an instruction which performs the unifcation with this node !!

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

Let us first consider the unifcation code for atoms and variables only:

$$
\begin{array}{lcl}
\text{code}_U\ a\ \rho & = & \text{uatom a} \\
\text{code}_U\ X\ \rho & = & \text{uvar}\ (\rho\, X) \\
\text{code}_U\ \_\ \rho & = & \text{pop} \\
\text{code}_U\ \bar{X}\ \rho & = & \text{uref}\ (\rho\, X) \\
 & \dots & \qquad\qquad // \text{ to be continued } \quad \text{:-)}
\end{array}
$$

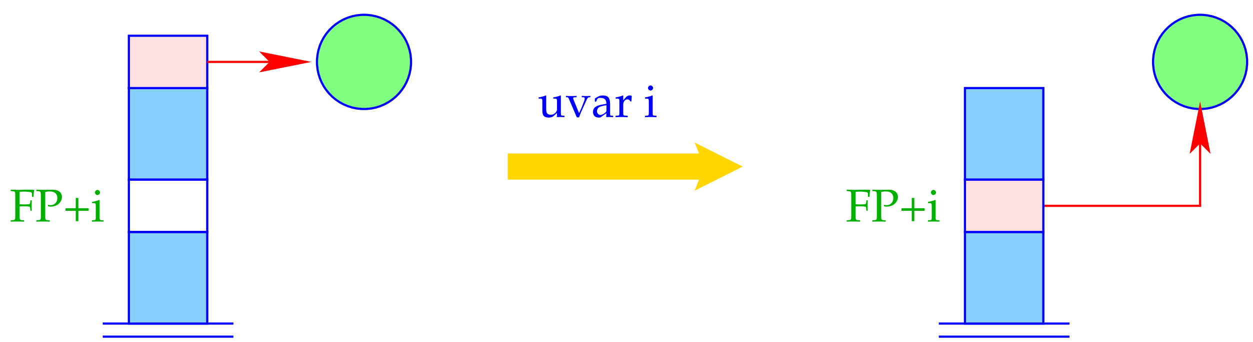
The instruction uatom a implements the unification with the atom a:




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

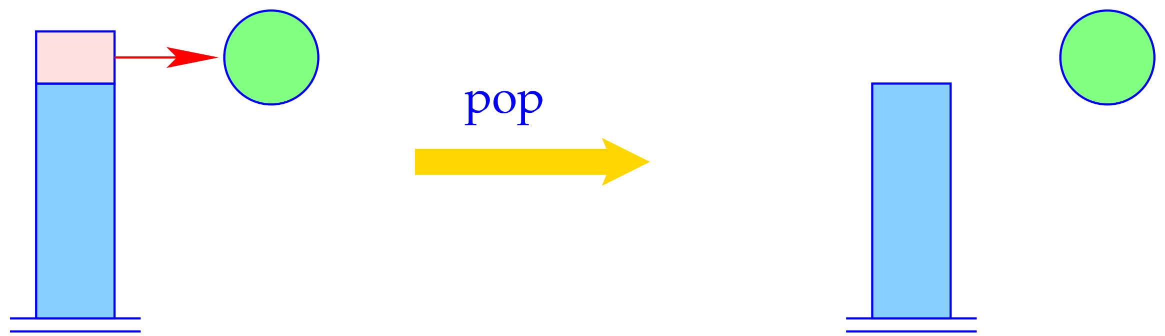
- The run-time function `trail()` records the a potential new binding.
- The run-time function `backtrack()` initiates backtracking.

The instruction uvar i implements the unification with an un-initialized variable:



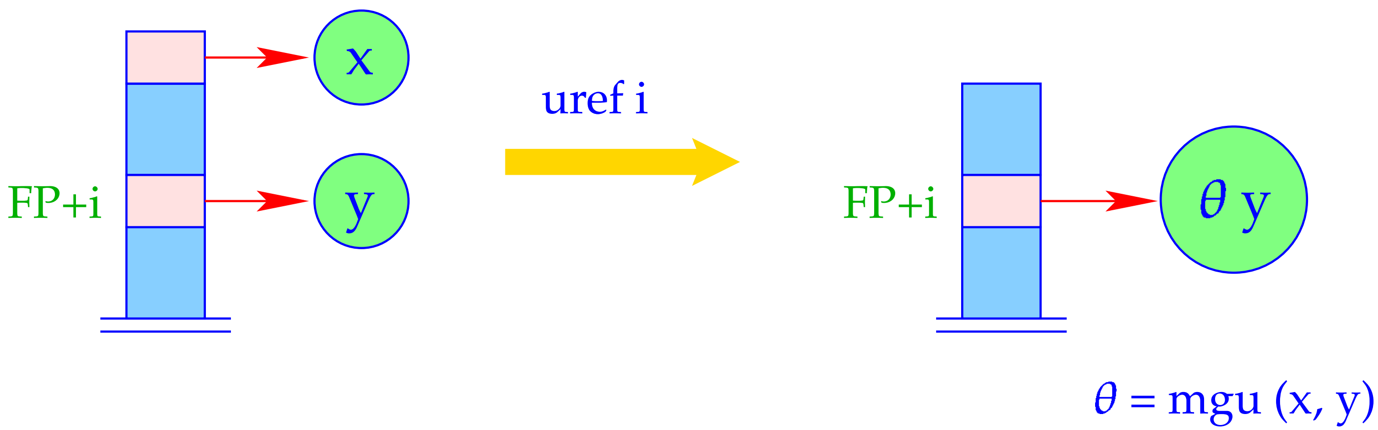
S[FP+i] = S[SP]; SP− −;

The instruction pop implements the unification with an anonymous variable. It always succeeds :-)



SP−−;

The instruction uref i implements the unification with an initialized variable:



unify (S[SP], deref (S[FP+i]));
SP– –;

It is only here that the run-time function `unify()` is called :-)

- The unification code performs a pre-order traversal over t.
- In case, execution hits at an unbound variable, we switch from checking to building :-)

$$
\begin{array}{llll}
\text{code}_U\ f(t_1,\ldots,t_n)\ \rho & = & & \text{ustruct f/n}\ A & // \text{ test} \\
& & & \text{son 1} & \\
& & & \text{code}_U\ t_1\ \rho & \\
& & & \ldots & \\
& & & \text{son n} & \\
& & & \text{code}_U\ t_n\ \rho & \\
& & & \text{up B} & \\
& & A: & \text{check}\ \mathit{ivars}(f(t_1,\ldots,t_n))\ \rho & // \text{ occur-check} \\
& & & \text{code}_A\ f(t_1,\ldots,t_n)\ \rho & // \text{ building !!} \\
& & & \text{bind} & // \text{ creation of bindings} \\
& & B: & \ldots &
\end{array}
$$

The Building Block:

Before constructing the new (sub-) term t' for the binding, we must exclude that it contains the variable X' on top of the stack !!!

This is the case iff the binding of no variable inside t' contains (a reference to) X'.

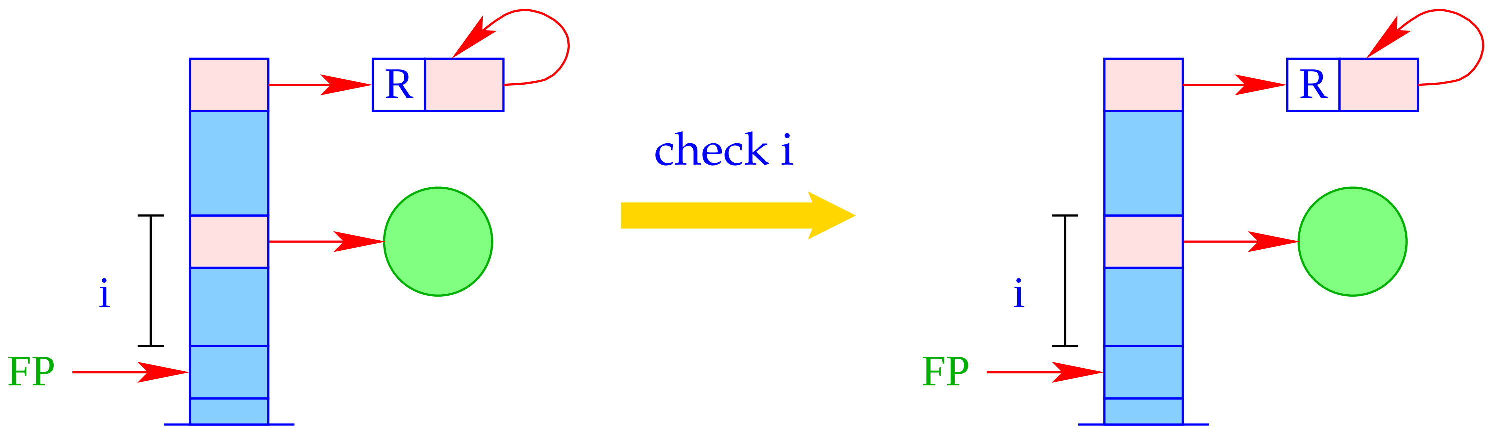
$\Longrightarrow$ $\mathit{ivars}(t')$ returns the set of already initialized variables of t.

$\Longrightarrow$ The macro check $\{Y_1, \ldots, Y_d\}\ \rho$ generates the necessary tests on the variables $Y_1, \ldots, Y_d$:

$$
\begin{array}{rcl}
\text{check } \{Y_1, \ldots, Y_d\}\ \rho & = & \text{check } (\rho\ Y_1) \\
 & & \text{check } (\rho\ Y_2) \\
 & & \ldots \\
 & & \text{check } (\rho\ Y_d)
\end{array}
$$

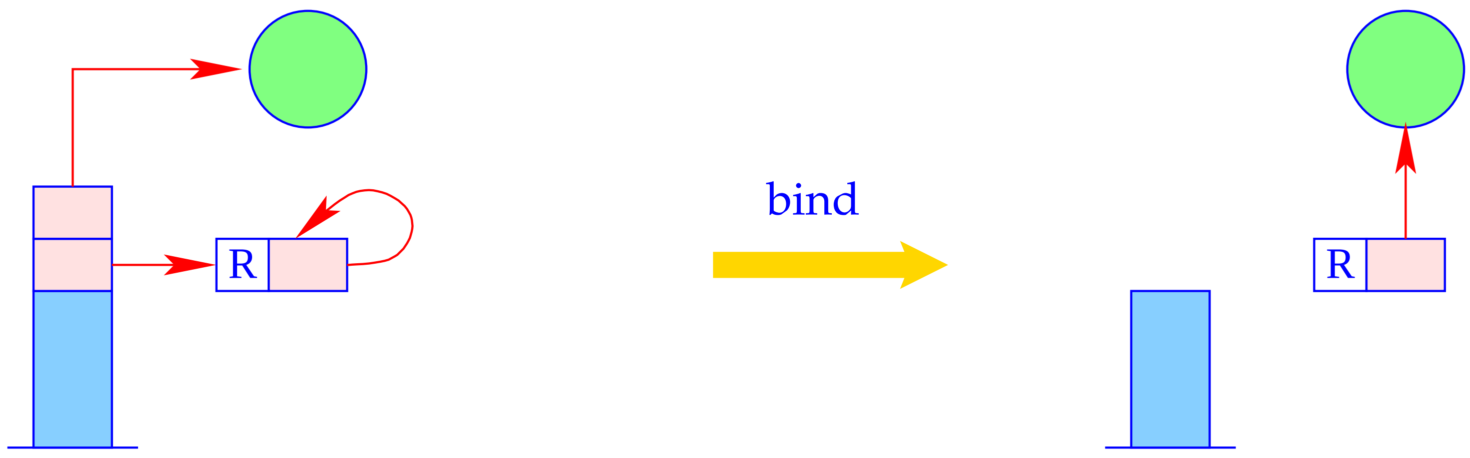
The instruction check i checks whether the (unbound) variable on top of the stack occurs inside the term bound to variable i.

If so, unification fails and backtracking is caused:




```
if (!check (S[SP], deref S[FP+i]))
    backtrack();
```

The instruction bind terminates the building block. It binds the (unbound) variable to the constructed term:



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

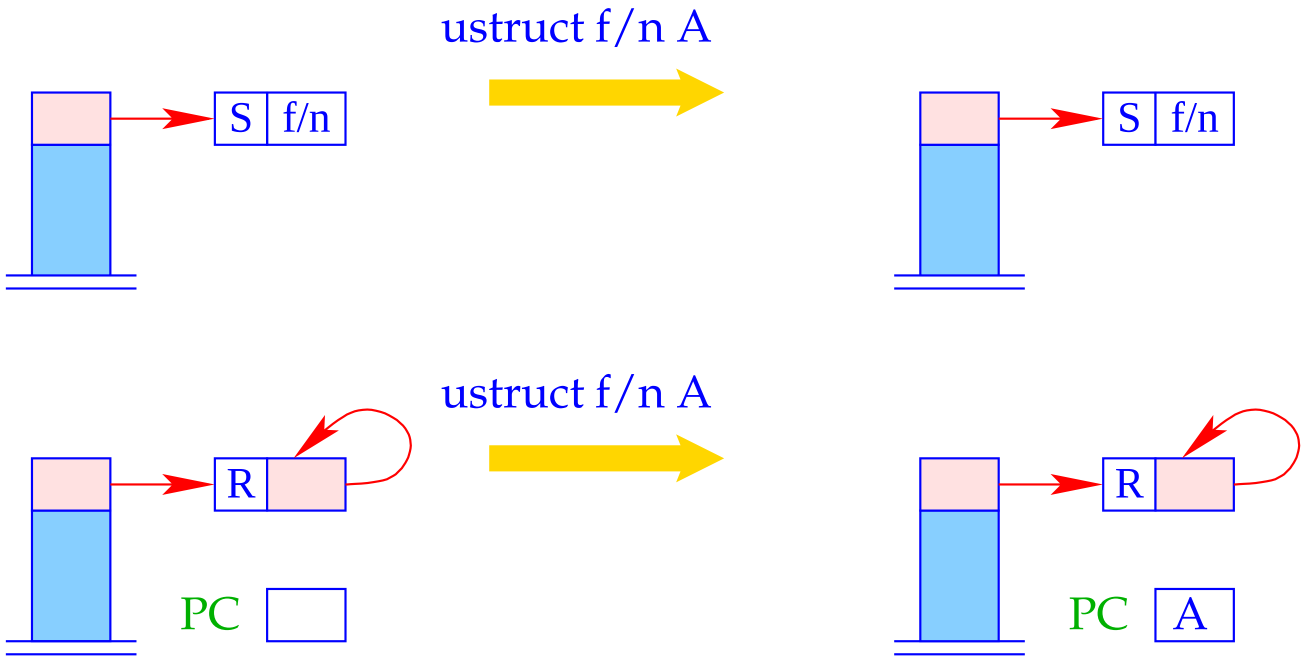
The Pre-Order Traversal:

- First, we test whether the topmost reference is an unbound variable.
  If so, we jump to the building block.
- Then we compare the root node with the constructor f/n.
- Then we recursively descend to the children.
- Then we pop the stack and proceed behind the unification code:

Once again the unification code for constructed terms:

$$
\begin{array}{lllll}
\text{code}_U\ f(t_1,\ldots,t_n)\ \rho & = & & \text{ustruct f/n}\ A & \text{// test} \\
& & & \text{son 1} & \text{// recursive descent} \\
& & & \text{code}_U\ t_1\ \rho & \\
& & & \ldots & \\
& & & \text{son n} & \text{// recursive descent} \\
& & & \text{code}_U\ t_n\ \rho & \\
& & & \text{up B} & \text{// ascent to father} \\
& & A: & \text{check}\ \mathit{ivars}(f(t_1,\ldots,t_n))\ \rho & \\
& & & \text{code}_A\ f(t_1,\ldots,t_n)\ \rho & \\
& & & \text{bind} & \\
& & B: & \ldots &
\end{array}
$$

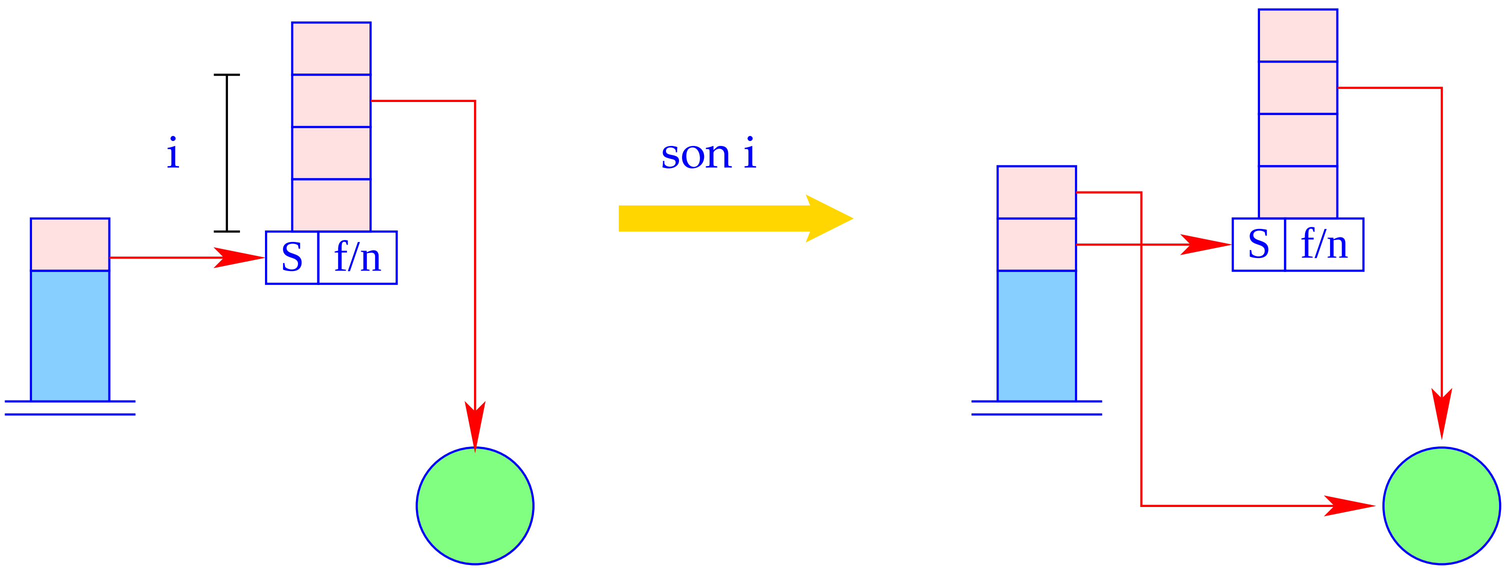
The instruction ustruct i implements the test of the root node of a structure:



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

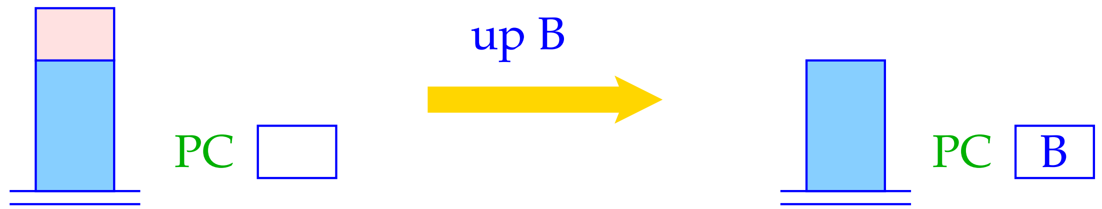
... the argument reference is not yet popped :-)

The instruction son i pushes the (reference to the) *i*-th sub-term from the structure pointed at from the topmost reference:



S[SP+1] = deref (H[S[SP]+i]); SP++;

It is the instruction up B which finally pops the reference to the structure:



SP– –; PC = B;

The continuation address B is the next address after the build-section.

Example:

For our example term $f(g(\bar{X}, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ we obtain:

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:	...

Code size can grow quite considerably — for deep terms. In practice, though, deep terms are “rare” :-)