Script generated by TTT

Title: Seidl: Virtual_Machines (03.07.2012)

Date: Tue Jul 03 14:08:00 CEST 2012

Duration: 87:05 min

Pages: 52

Example:

$$
\begin{array}{lll}
\text{bigger}(X,Y) & \leftarrow & X = \textit{elephant}, Y = \textit{horse} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{horse}, Y = \textit{donkey} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{donkey}, Y = \textit{dog} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{donkey}, Y = \textit{monkey} \\
\text{is\_bigger}(X,Y) & \leftarrow & \text{bigger}(X,Y) \\
\text{is\_bigger}(X,Y) & \leftarrow & \text{bigger}(X,Z), \text{is\_bigger}(Z,Y) \\
? & \text{is\_bigger}(\textit{elephant}, \textit{dog}) &
\end{array}
$$

A More Realistic Example:

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

A More Realistic Example:

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

Remark:

$[\,]$	==	the atom empty list
$[H\|Z]$	==	binary constructor application
$[a,b,Z]$	==	shortcut for: $[a\|[b\|[Z\|[\,]]]]$

A program p is constructed as follows:

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

- A term t either is an atom, a variable, an anonymous variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program consists of a sequence of clauses together with a single goal as query.

Procedural View of Proll programs:

~~goal~~	$==$	procedure call
predicate	$==$	procedure
clause	$==$	definition
term	$==$	value
unification	$==$	basic computation step
binding of variables	$==$	side effect

Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only :-)
- ... may fail. Then the next definition is tried :-))

$\Longrightarrow$ backtracking

Procedural View of Proll programs:

goal	$==$	procedure call
predicate	$==$	procedure
clause	$==$	definition
term	$==$	value
unification	$==$	basic computation step
binding of variables	$==$	side effect

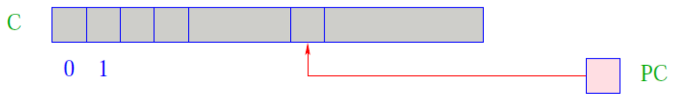
Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only :-)
- ... may fail. Then the next definition is tried :-))

$\Longrightarrow$ backtracking

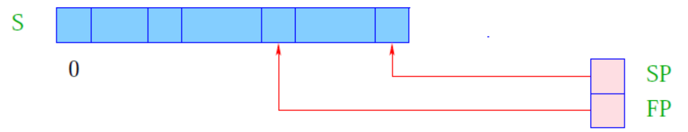
27 Architecture of the WiM:

The Code Store:



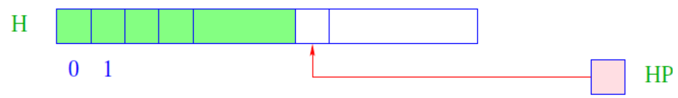
C	=	Code store – contains WiM program; every cell contains one instruction;
PC	=	Program Counter – points to the next instruction to executed;

The Runtime Stack:



S	=	Runtime Stack – every cell may contain a value or an address;
SP	=	Stack Pointer – points to the topmost occupied cell;
FP	=	Frame Pointer – points to the current stack frame.
		Frames are created for predicate calls,
		contain cells for each variable of the current clause

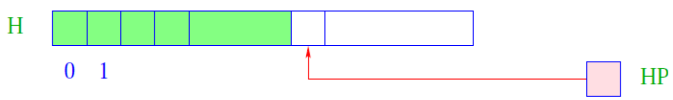
The Heap:



H	=	Heap for dynamicly constructed terms;
HP	=	Heap-Pointer – points to the first free cell;

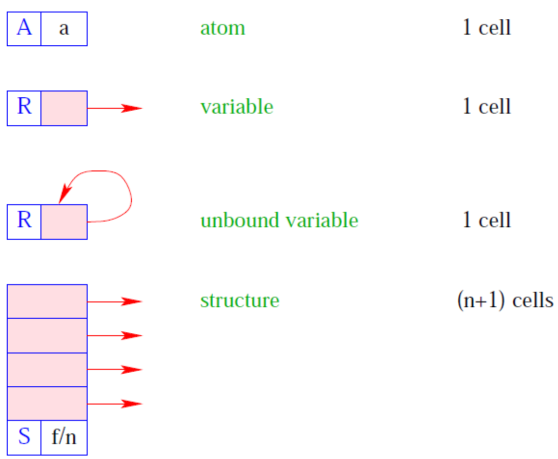
- The heap in maintained like a stack as well :-)
- A new-instruction allocates a object in H.
- Objects are tagged with their types (as in the MaMa) ...

The Heap:



H	=	Heap for dynamicly constructed terms;
HP	=	Heap-Pointer – points to the first free cell;

- The heap in maintained like a stack as well :-)
- A new-instruction allocates a object in H.
- Objects are tagged with their types (as in the MaMa) ...



28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment ρ returns, for each clause variable X its address (relative to FP) on the stack. Then $\text{code}_A\ t\ \rho$ should ...

- construct (a presentation of) t in the heap; and
- return a reference to it on top of the stack.

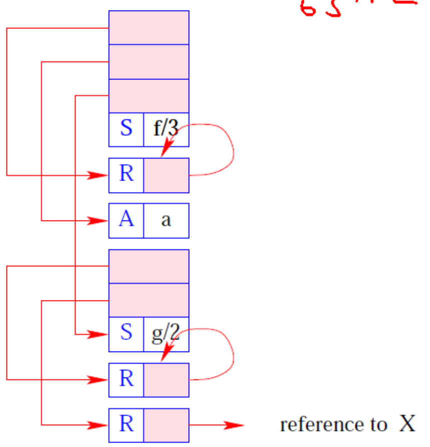
Idea:

- Construct the tree during a post-order traversal of t
- with one instruction for each new node!

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

Assume that X is initialized, i.e., $S[FP + \rho\, X]$ contains already a reference, Y and Z are not yet initialized.

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:

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

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:

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

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

$p(X) \leftarrow q(\bar{X}, Y).$

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

Note: Arguments are always initialized!

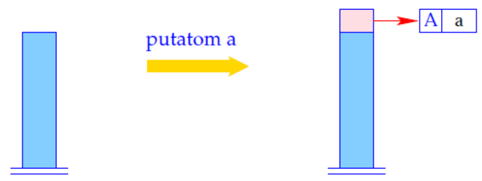
Then we define:

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

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

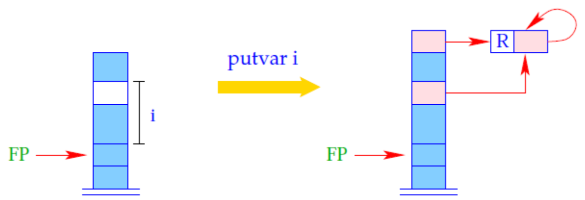
putref 1	putatom a
putvar 2	putvar 3
putstruct g/2	putstruct f/3

The instruction putatom a constructs an atom in the heap:



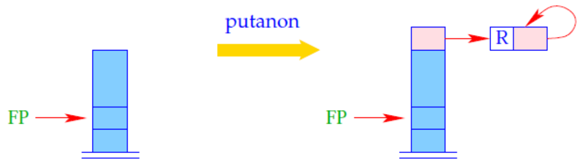
SP++; S[SP] = new (A,a);

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



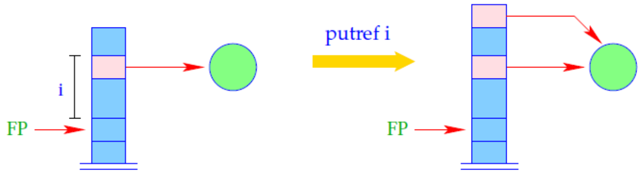
SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];

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

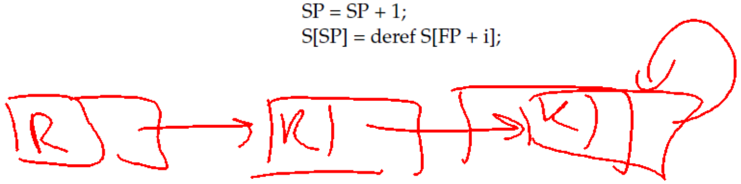


SP = SP + 1;
S[SP] = new (R, HP);

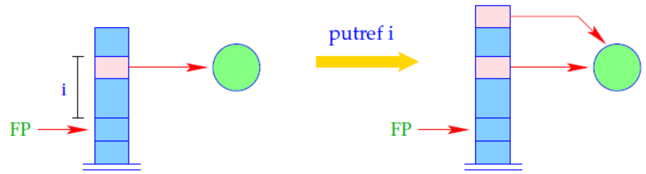
The instruction putref i pushes the value of the variable onto the stack:



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



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

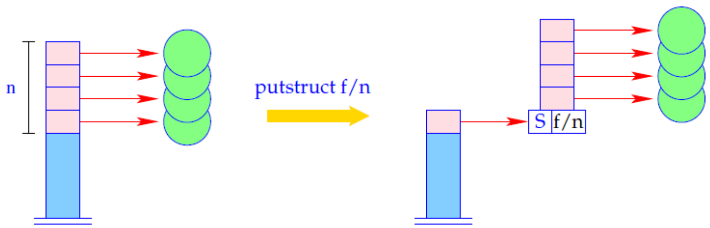


SP = SP + 1;
S[SP] = deref S[FP + i];

The auxiliary function deref contracts chains of references:

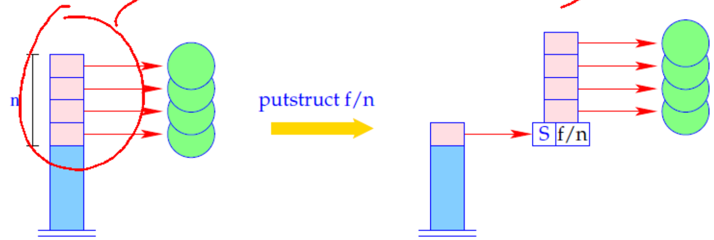
```
ref deref (ref v) {
    if (H[v]==(R,w) && v!=w) return deref (w);
    else return v;
}
```


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;

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;

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.

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

```
        mark B          // allocates the stack frame
        codeA t1 ρ
        ...
        codeA tk ρ
        call p/k        // calls the procedure p/k
B :     ...
```


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

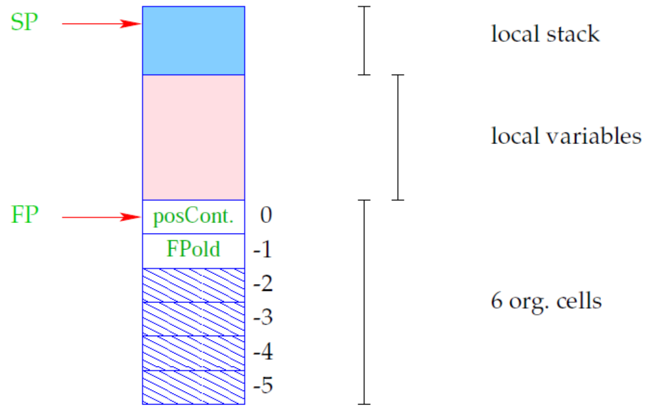
```
        mark B          // allocates the stack frame
        codeA t1 ρ
        ...
        codeA tk ρ
        call p/k        // calls the procedure p/k
B :     ...
```

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:

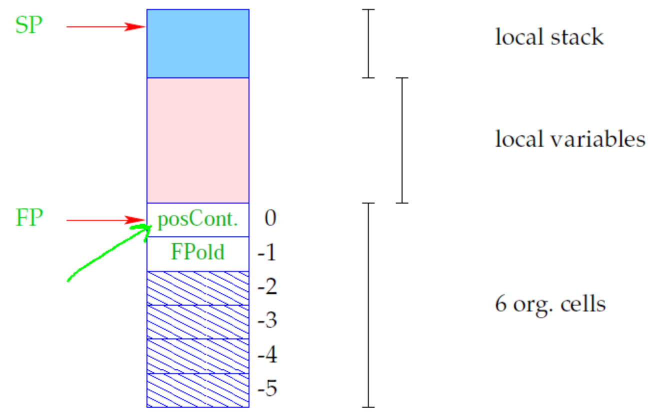


Remarks:

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

⟹ will be discussed at the translation of predicates.

Stack Frame of the WiM:

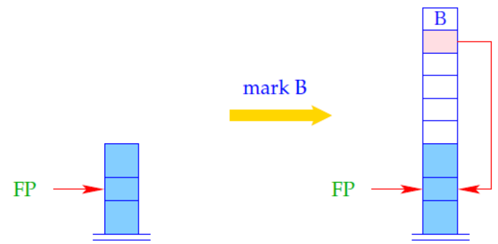


Remarks:

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

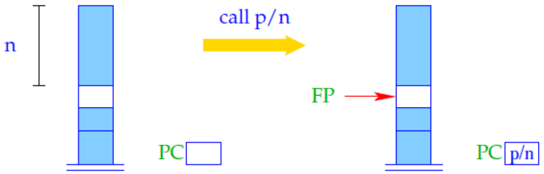
⟹ will be discussed at the translation of predicates.

The instruction mark B allocates a new stack frame:



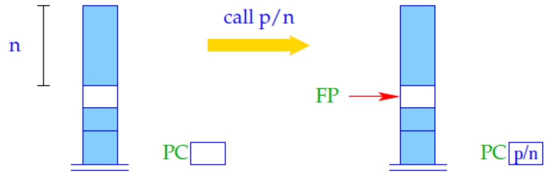
SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

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



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

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



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

30 Unification

Convention:

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

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

X = a

30 Unification

Convention:

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

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

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 :-)

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

_ = t

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 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 :-)

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

Example:

Consider the equation:

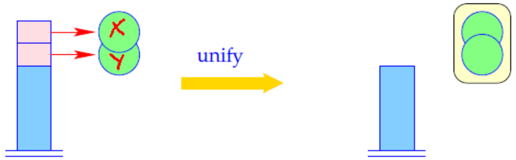
$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Then we obtain for an address environment

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

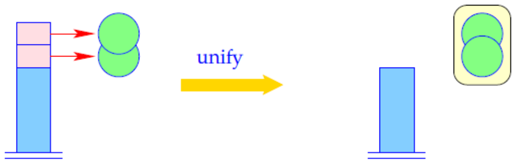
```
putref 4    putref 1        putatom a       unify
            putvar 2        putvar 3
            putstruct g/2   putstruct f/3
```


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




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


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



unify (S[SP-1], S[SP]);
SP = SP–2;

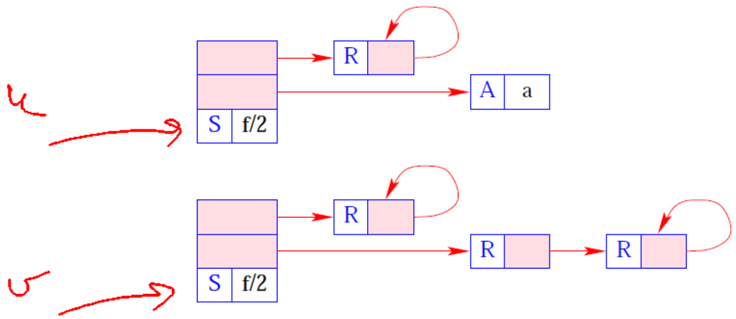
X = f(X)

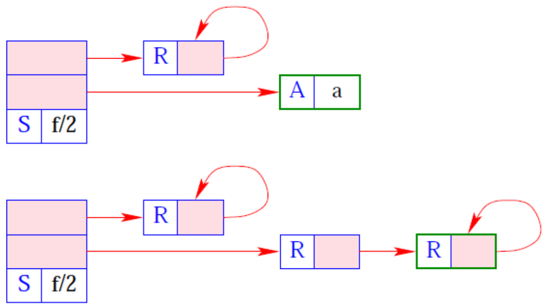
```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u]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

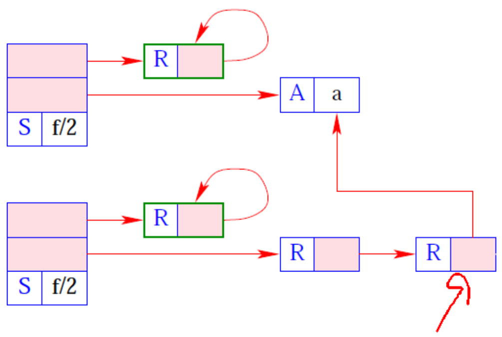
```
...
if ((H[v] == (R,_)) {
    if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
  }
  if (H[u]==(A,a) && H[v]==(A,a))
    return true;
  if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
    for (int i=1; i<=n; i++)
      if(!unify (deref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

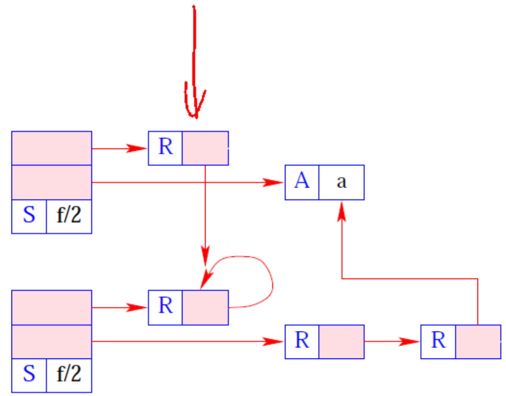


```
...
if ((H[v] == (R,_)) {
    if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
  }
  if (H[u]==(A,a) && H[v]==(A,a))
    return true;
  if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
    for (int i=1; i<=n; i++)
      if(!unify (deref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







- The run-time function `trail()` records the ~~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 !!