Script generated by TTT

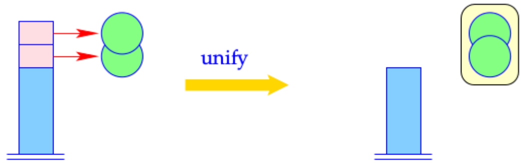
Title: Seidl: Virtual_Machines (16.06.2015)

Date: Tue Jun 16 10:16:14 CEST 2015

Duration: 90:43 min

Pages: 46

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



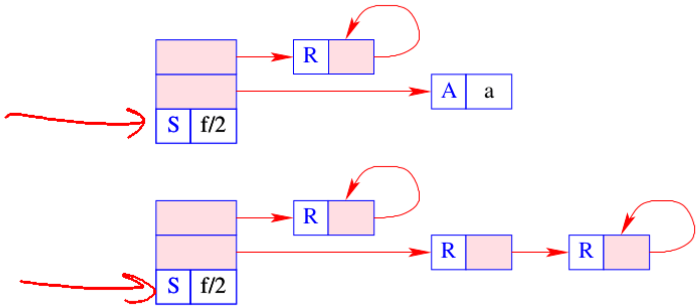

```
unify (S[SP-1], S[SP]);
SP = SP–2;
```

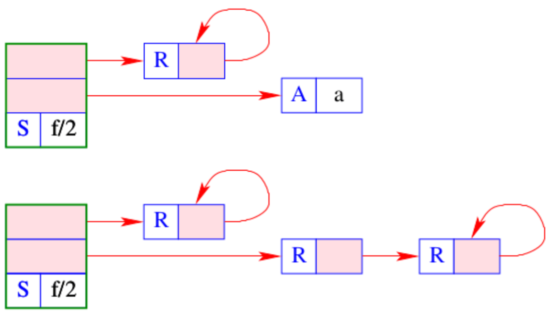

The Function `unify()`

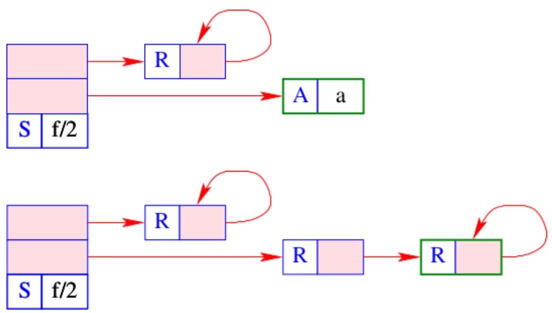
- ... takes two heap addresses.
  For each call, we guarantee that these are maximally de-referenced.
- ... checks whether the two addresses are already identical.
  If so, does nothing :-)
- ... binds younger variables (larger addresses) to older variables (smaller addresses);
- ... when binding a variable to a term, checks whether the variable occurs inside the term ⟹ occur-check;
- ... records newly created bindings;
- ... may fail. Then backtracking is initiated.

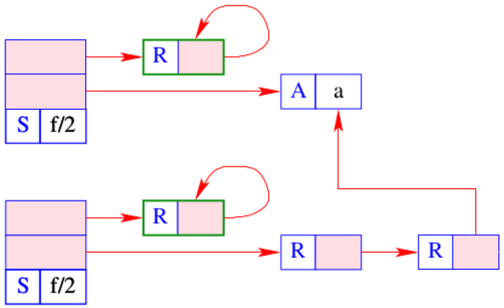

```
bool unify (ref u, ref v) {
   if (u == v) return true;
   if (H[u] == (R,_)) {
      if (H[v] == (R,_)) {
         if (u>v) {
            H[u] = (R,v); trail (u); return true;
         } else {
            H[v] = (R,u); trail (v); return true;
         }
      } elseif (check (u,v)) {
         H[u]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

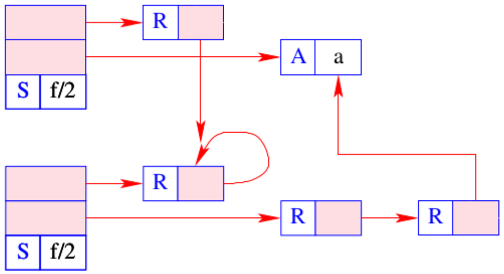
```
...
 if ((H[v] == (R,_)) {
      if (check (v,u)) {
         H[v] = (R,u); trail (v); return true;
      } else {
         backtrack(); return false;
      }
   }
   if (H[u]==(A,a) && H[v]==(A,a))
      return true;
   if (H[u]==(S, f/n) && H[v]==(S, f/n)) {
      for (int i=1; i<=n; i++)
     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









- 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;}
```


- 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;}
```

$X = f(X)$

$X = f(f(\ldots))$

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 !!

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

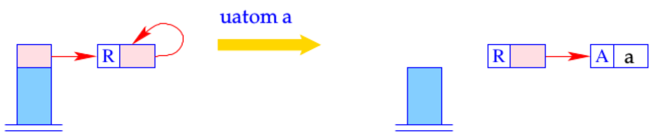
Let us first consider the unifcation code for atoms and variables only:

$$\begin{array}{lcll} \text{code}_U\ a\ \rho & = & \text{uatom a} & \\ \text{code}_U\ X\ \rho & = & \text{uvar}\ (\rho\, X) & \\ \text{code}_U\ \_\ \rho & = & \text{pop} & \\ \text{code}_U\ \tilde{X}\ \rho & = & \text{uref}\ (\rho\, X) & \\ & \ldots & & \text{// to be continued :-)} \end{array}$$

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

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

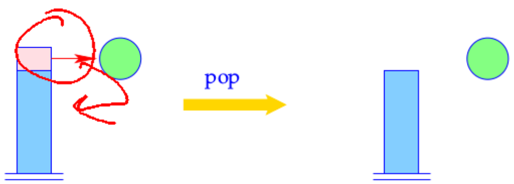
The instruction uatom a implements the unification with the atom a:




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

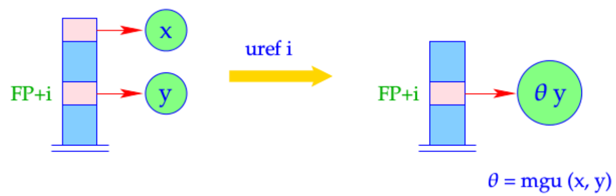
- The run-time function `trail()` records the a potential new binding.
- The run-time function `backtrack()` initiates backtracking.

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



SP−−;

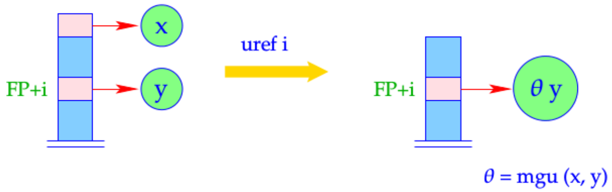
The instruction uref i implements the unification with an initialized variable:




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

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

The 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 :-)

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

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

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

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'.

$\Longrightarrow$ $ivars(t')$ 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 $Y_1, \ldots, Y_d$:

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

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

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

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'.

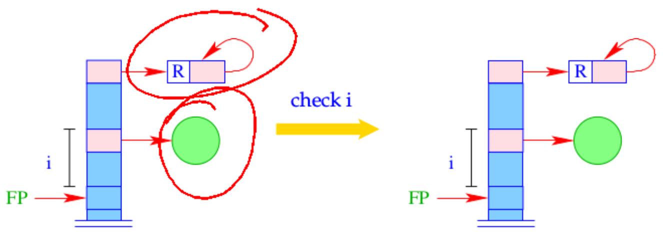
$\Longrightarrow$ $ivars(t')$ returns the set of already initialized variables of t.

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

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

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 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'.

$\Longrightarrow$ $ivars(t')$ 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 $Y_1, \ldots, Y_d$:

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

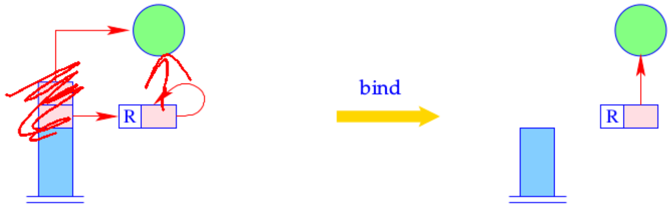
280

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

$\text{code}_U\ f(t_1,\ldots,t_n)\ \rho \quad = \quad$

```
        ustruct f/n A                      // test
        son 1
        code_U t_1 ρ
        ...
        son n
        code_U t_n ρ
        up B
A :     check ivars(f(t_1,...,t_n)) ρ      // occur-check
        code_A f(t_1,...,t_n) ρ            // building !!
        bind                               // creation of bindings
B :     ...
```


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

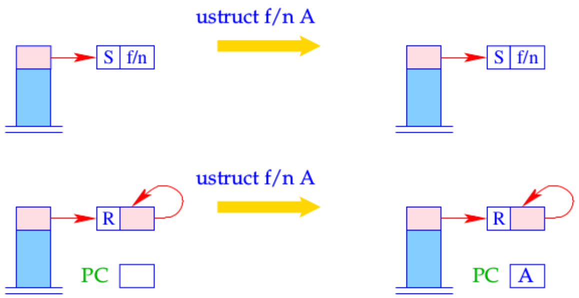
- First, we test whether the topmost reference is an unbound variable.
  If so, we jump to the building block.
- Then we compare the root node with the constructor f/n.
- Then we recursively descend to the children.
- Then we pop the stack and proceed behind the unification code:

Once again the unification code for constructed terms:

$\text{code}_U\ f(t_1,\ldots,t_n)\ \rho \quad = \quad$

```
        ustruct f/n A                      // test
        son 1                              // recursive descent
        code_U t_1 ρ
        ...
        son n                              // recursive descent
        code_U t_n ρ
        up B                               // ascent to father
A :     check ivars(f(t_1,...,t_n)) ρ
        code_A f(t_1,...,t_n) ρ
        bind
B :     ...
```

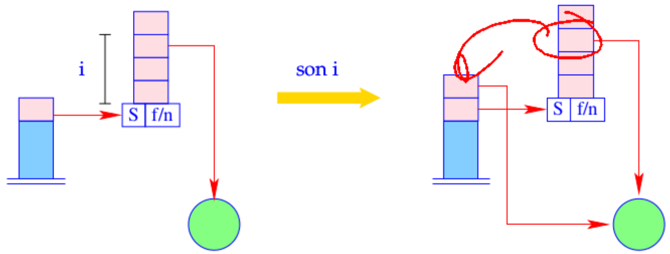
The instruction ustruct f/n A implements the test of the root node of a structure:




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

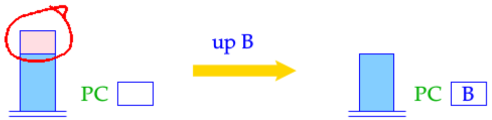
... the argument reference is not yet popped :-)

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++;

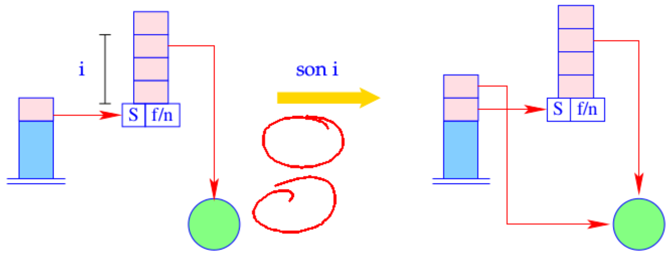
It is the instruction up B which finally pops the reference to the structure:



SP– –; PC = B;

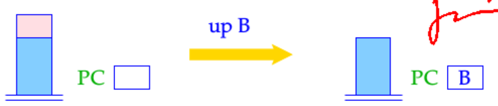
The continuation address B is the next address after the build-section.

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 $f(g(X, Y), a, Z)$ 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" :-)

32 Clauses

Clausal code must

- allocate stack space for locals;
- evaluate the body;
- free the stack frame (whenever possible :-)

Let r denote the clause: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$.

Let $\{X_1, \ldots, X_m\}$ denote the set of locals of r and ρ the address environment:

$$\rho\ X_i = i$$

Remark The first k locals are always the formals :-)

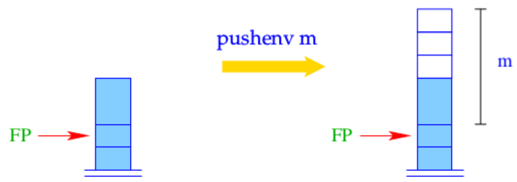
Then we translate:

$$\begin{array}{lcll} \text{code}_C\ r & = & \text{pushenv m} & \text{// allocates space for locals} \\ & & \text{code}_G\ g_1\ \rho & \\ & & \ldots & \\ & & \text{code}_G\ g_n\ \rho & \\ & & \text{popenv} & \end{array}$$

The instruction popenv restores FP and PC and tries to pop the current stack frame.

It should succeed whenever program execution will never return to this stack frame :-)

The instruction pushenv m sets the stack pointer:



$$SP = FP + m;$$

Example

Consider the clause r:

$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Then $\text{code}_C\ r$ yields:

pushenv 3	mark A	A:	mark B	B:	popenv
	putref 1		putref 3		
	putvar 3		putref 2		
	call f/2		call a/2		

33 Predicates

A predicate q/k is defined through a sequence of clauses $rr \equiv r_1 \ldots r_f$.

The translation of q/k provides the translations of the individual clauses r_i.

In particular, we have for $f = 1$:

$$\text{code}_P\ rr = q/k : \text{code}_C\ r_1$$

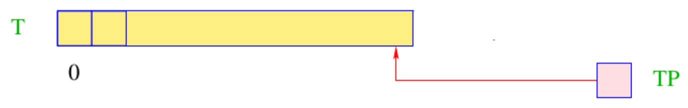
If q/k is defined through several clauses, the first alternative must be tried.

On failure, the next alternative must be tried

$\Longrightarrow$ backtracking :-)

33.1 Backtracking

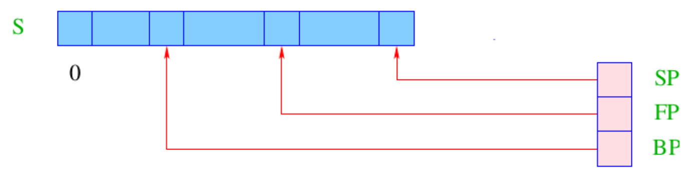
- Whenever unifcation fails, we call the run-time function `backtrack()`.
- The goal is to roll back the whole computation to the (dynamically :-) latest goal where another clause can be chosen $\Longrightarrow$ the last backtrack point.
- In order to undo intermediate variable bindings, we always have recorded new bindings with the run-time function `trail()`.
- The run-time function `trail()` stores variables in the data-structure trail:



TP == Trail Pointer

points to the topmost occupied Trail cell

The current stack frame where backtracking should return to is pointed at by the extra register BP:



A backtrack point is stack frame to which program execution possibly returns.

- We need the code address for trying the next alternative (negative continuation address);
- We save the old values of the registers HP, TP and BP.
- Note: The new BP will receive the value of the current FP :-)

For this purpose, we use the corresponding four organizational cells:

FP →	posCont.	0
	FPold	-1
	HPold	-2
	TPold	-3
	BPold	-4
	negCont.	-5