The Function \texttt{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 \(\Rightarrow\) occur-check;
- ... records newly created bindings;
- ... may fail. Then backtracking is initiated.
... if ((H[v] == (R,..)) {
    if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
} else {
    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<=s; i++)
            if(!unify (deref (H[v+s]), 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.,

```cpp
bool check (ref u, ref v) { return true; }
```
Otherwise, we could implement the run-time function `check()` as follows:

```c
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 $\hat{X} = t$ 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 unification with this node $!!$

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

```
code_2 s ρ = uatom a
code_2 X ρ = uvar (ρ X)
code_2 ⊥ ρ = pop
code_2 (X ρ = uref (ρ X))
.....  // to be continued $:-)$
```

```c
code_2 (X = t) ρ = put X ρ
code_2 t ρ
```
Let us first consider the unification code for atoms and variables only:

\[
\begin{align*}
\text{code}_U \ a \ \rho &= \ \text{uatom} \ a \\
\text{code}_U \ X \ \rho &= \ \text{uvar} \ (\rho \ X) \\
\text{code}_U \ _\ \rho &= \ \text{pop} \\
\text{code}_U \ X \ \rho &= \ \text{uref} \ (\rho \ X) \\
... &\quad \text{// to be continued} \\
\end{align*}
\]

The instruction \text{uatom} \ a \ implements the unification with the atom \ a:

\[
\begin{align*}
u &= \text{S}[\text{SP}; \text{SP}--]; \\
\text{switch} \ (\text{H}[v]) \ {\{} \\
\text{case} \ (A, a): \ &\quad \text{break}; \\
\text{case} \ (R, \_): \ &\quad \text{H}[v] = (R, \text{new} \ (A, a)); \\
\text{default}: \ &\quad \text{backtrack}(); \\
{\}}
\end{align*}
\]

- The run-time function \text{trail} () \ records the a potential new binding.
- The run-time function \text{backtrack} () \ initiates backtracking.

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

The instruction \text{uref} i \ implements the unification with an initialized variable:

\[
\begin{align*}
\text{FP}_i \ x \\
\text{FP}_i \ y \\
\text{FP}_i \\
\theta \ y \\
\theta = \text{mgu} \ (x, y)
\end{align*}
\]

It is only here that the run-time function \text{unify} () \ is called \ :-)

\[
\begin{align*}
\text{unify} \ (\text{S}[\text{S}I\ dref \ (\text{S}[\text{FP}_i])); \\
\text{SP}--;
\end{align*}
\]
The instruction \texttt{uref}\,\,\texttt{i} implements the unification with an initialized variable:

\[
\begin{align*}
\text{FP}\,\text{+i} & \quad \xrightarrow{\text{uref}\,\,\texttt{i}} \quad \Theta \\
\theta = \text{mu}_\theta(x, y)
\end{align*}
\]

\text{unify}((\text{SP}', \text{deref}((\text{SP}\,\text{+i}))))

\text{SP}' = \cdots

It is only here that the run-time function \texttt{unify()} is called :-)

---

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

\[
\begin{align*}
\text{code}_\text{u} f(t_1, \ldots, t_n) \, \rho & = \quad \text{unsafe} t/n A \\
& \quad \text{son 1} \\
& \quad \text{code}_\text{u} t_1 \, \rho \\
& \quad \ldots \\
& \quad \text{son n} \\
& \quad \text{code}_\text{u} t_n \, \rho \\
& \quad \text{upB B} \\
A : & \quad \text{check} \ \text{trans}(f(t_1, \ldots, t_n)) \, \rho \\
& \quad \text{code}_\text{u} f(t_1, \ldots, t_n) \, \rho \\
& \quad \text{bind} \\
B : & \quad \ldots
\end{align*}
\]

// test

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 !!!

\( \implies \text{trans}(t') \) returns the set of already initialized variables of \( t \).

\( \implies \) The macro \texttt{check}\{\( Y_1, \ldots, Y_k \)\} \( \rho \) generates the necessary tests on the variables \( Y_1, \ldots, Y_k \):

\[
\begin{align*}
\text{check}\{\( Y_1, \ldots, Y_k \)\} \, \rho & = \quad \text{check} (\rho Y_1) \\
& \quad \text{check} (\rho Y_2) \\
& \quad \ldots \\
& \quad \text{check} (\rho Y_k)
\end{align*}
\]
• The unification code performs a pre-order traversal over \( t \).
• In case, execution hits an unbound variable, we switch from checking to building \( \Rightarrow \)

\[
\text{code} (f(t_1, \ldots, t_n)) \rho = \begin{cases} 
\text{struct } f/n \ A \quad \text{// test} \\
\quad \text{son } 1 \\
\quad \text{code} \ t_1 \ \rho \\
\quad \ldots
\quad \text{son } n \\
\quad \text{code} \ t_n \ \rho \\
\quad \text{up } B
\end{cases}
\]

\[ A : \quad \text{check i} = \text{trans}(f(t_1, \ldots, t_n)) \rho \quad \text{// occur-check} \\
\quad \text{code} \ f(t_1, \ldots, t_n) \ \rho \quad \text{// building !!} \\
\quad \text{bind} \quad \text{// creation of bindings}
\]

\[ B : \quad \ldots \]

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 if the binding of no variable inside \( t' \) contains (a reference to) \( X' \).

\[ \text{trans}(t') \quad \text{returns the set of already initialized variables of } t. \]

\[ \text{check } \{ Y_1, \ldots, Y_d \} \ \rho \quad \text{generates the necessary tests on the variables } Y_1, \ldots, Y_d : \]

\[ \text{check } \{ Y_1, \ldots, Y_d \} \ \rho = \begin{cases} 
\text{check } (\rho \ Y_1) \\
\text{check } (\rho \ Y_2) \\
\ldots
\end{cases} \]

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 if the binding of no variable inside \( t' \) contains (a reference to) \( X' \).

\[ \text{trans}(t') \quad \text{returns the set of already initialized variables of } t. \]

\[ \text{check } \{ Y_1, \ldots, Y_d \} \ \rho \quad \text{generates the necessary tests on the variables } Y_1, \ldots, Y_d : \]

\[ \text{check } \{ Y_1, \ldots, Y_d \} \ \rho = \begin{cases} 
\text{check } (\rho \ Y_1) \\
\text{check } (\rho \ Y_2) \\
\ldots
\end{cases} \]
The unification code performs a pre-order traversal over \( t \).

In case, execution hits at an unbound variable, we switch from checking to building. 

\[
\text{code}_\text{U} f(t_1, \ldots, t_n) \rho = \begin{cases}
\text{struct } f / n \ A \\
\text{son } 1 \\
\text{code}_\text{U} t_1 \rho \\
\ldots \\
\text{son } n \\
\text{code}_\text{U} t_n \rho \\
\text{up } B \\
A : \text{check } \text{trans}(f(t_1, \ldots, t_n)) \rho & \text{ occurs-check} \\
\text{code}_\text{U} f(t_1, \ldots, t_n) \rho & \text{building} \\
\text{bind} & \text{creation of bindings} \\
B : \ldots 
\end{cases}
\]

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

\[
H[S[SP \cdot 1]] = (R, S[SP]); \\
\text{trail } S[SP \cdot 1] = \cdot; \\
SP = SP - 2;
\]

The Pre-Order Traversal

- First, we \text{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 \text{recursively descend} to the children.
- Then we \text{pop} the stack and proceed behind the unification code:

\[
\text{code}_\text{U} f(t_1, \ldots, t_n) \rho = \begin{cases}
\text{struct } f / n \ A \\
\text{son } 1 \\
\text{code}_\text{U} t_1 \rho \\
\ldots \\
\text{son } n \\
\text{code}_\text{U} t_n \rho \\
\text{up } B \\
A : \text{check } \text{trans}(f(t_1, \ldots, t_n)) \rho \\
\text{code}_\text{U} f(t_1, \ldots, t_n) \rho \\
\text{bind} & \text{bind} \\
B : \ldots 
\end{cases}
\]

Once again the unification code for constructed terms:
The instruction \texttt{ustruct f/n A} implements the test of the root node of a structure:

\begin{itemize}
  \item \texttt{switch (H[S][SP+i]) { case (S, fn)): break; case (R, _): PC = A; break; default: backtrack();}}
\end{itemize}

\texttt{... the argument reference is not yet popped :-)}

285

It is the instruction \texttt{up B} which finally pops the reference to the structure:

\begin{itemize}
  \item \texttt{SP--}; PC = B;
\end{itemize}

The continuation address B is the next address after the build-section.
It is the instruction `up B` which finally pops the reference to the structure.

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

---

### 32 Clauses

Clausal code must:
- **allocate** stack space for locals;
- **evaluate** the body;
- **free** the stack frame (whenever possible) ☺

Let `r` denote the clause:

\[ p(X_1, \ldots, X_k) \leftarrow \varphi_1, \ldots, \varphi_n \]

Let \( \{X_1, \ldots, X_m\} \) denote the set of locals of `r` and \( \rho \) the address environment:

\[ \rho \ X_i = i \]

**Remark** The first `k` locals are always the **formals** ☺

---

### Example

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

\[
\begin{align*}
\text{unistuct } & f/3 \ A_1: \quad \text{up } B_2: \quad \text{bind} \\
\text{son } & 1: \quad \text{putvar } 2: \quad \text{putvar } 2 \\
\text{unistuct } & g/2 \ A_2: \quad \text{substruct g/2} \\
\text{son } & 1: \quad \text{putatom a} \\
\text{upref } & 1: \quad \text{putvar } 2: \quad \text{up } B_1: \\
\text{uvar } & 2: \quad \text{bind} \\
\text{substruct g/2} & A_1: \quad \text{check 1} \\
\text{putref } & 1: \quad \text{putatom a} \\
\text{bind } & 2: \quad \text{up } B_1: \\
\end{align*}
\]

Code size can grow quite considerably — for deep terms. In practice, though, deep terms are "rare" ☺

---

Then we translate:

\[
\text{code: } r = \ \text{pushenv } m \quad \text{// allocates space for locals} \\
\text{code: } g_1: \rho \quad \text{...} \quad \text{code: } g_k: \rho \\
\text{popenv} \\
\]

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:

\[ \text{SP} = \text{FP} + m; \]

33 Predicates

A predicate \( q/k \) is defined through a sequence of clauses \( r_1 \equiv r_2 \ldots r_f \).

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

In particular, we have for \( f = 1 \):

\[
\text{code}_r \ r = q/k : \text{code}_c \ r_1
\]

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

On failure, the next alternative must be tried

\[ \Longrightarrow \text{backtracking \ :) \} \]

Example

Consider the clause \( r \):

\[ a(X, Y) \leftarrow f(X, X_1), a(X_1, Y_1) \]

Then \( \text{code}_c \ r \) yields:

\[ \text{pushenv} \ 3 \quad \text{mark} \ A \quad \text{putref} \ 1 \quad \text{putvar} \ 3 \quad \text{call} \ f/2 \quad \text{A:} \quad \text{mark} \ B \quad \text{putref} \ 2 \quad \text{putref} \ 3 \quad \text{call} \ a/2 \]

33.1 Backtracking

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