Otherwise, we could implement the run-time function \( \text{check}() \) 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 \( X = t \) is very simple \( \top \)
- Often the constructed cells immediately become garbage \( \bot \)

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!
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!!

\[ \text{code}_t (\hat{X} = t) \rho = \text{put} \hat{X} \rho \]
\[ \text{code}_t t \rho \]

---

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

\[ \text{code}_t \ a \rho = \text{uatom} \ a \]
\[ \text{code}_t \ X \rho = \text{uvar} (\rho X) \]
\[ \text{code}_t \ - \rho = \text{pop} \]
\[ \text{code}_t \ \bar{X} \rho = \text{uref} (\rho X) \]

... // to be continued :-(

---

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

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

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

---

The instruction \texttt{uvar i} implements the unification with an un-initialized variable:

\[ \text{FP} + i = \text{SP}; \ SP -->; \]

\[ S[FP+i] = S[SP]; \ SP -->; \]
The instruction `pop` implements the unification with an anonymous variable. It always succeeds

\[ \text{SP} \leftarrow ; \]

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

\[ \text{FP} + i \quad x \rightarrow \text{FP} + i \quad \theta \ y \]
\[ \theta = \text{mgu}(x, y) \]

\[ \text{SP} \leftarrow ; \]

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.

\[
\text{code}_I \ f(t_1, \ldots, t_n) \ \rho = \\
\quad \text{ustruct } l/n (A) \quad \text{test} \\
\quad \text{son } 1 \\
\quad \text{code}_I \ t_1 \ \rho \\
\quad \ldots \\
\quad \text{son } n \\
\quad \text{code}_I \ t_n \ \rho \\
\quad \up B \\
\quad A : \text{check invrs}(f(t_1, \ldots, t_n)) \ \rho \quad \text{// occur-check} \\
\quad \text{code}_I \ f(t_1, \ldots, t_n) \ \rho \quad \text{// building} \\
\quad \text{bind} \quad \text{// creation of bindings} \\
\quad B : \ldots
\]

\[
\text{code}_I \ f(t_1, \ldots, t_n) \ \rho = \\
\quad \text{ustruct } l/n A \quad \text{test} \\
\quad \text{son } 1 \\
\quad \text{code}_I \ t_1 \ \rho \\
\quad \ldots \\
\quad \text{son } n \\
\quad \text{code}_I \ t_n \ \rho \\
\quad \up B \\
\quad A : \text{check invrs}(f(t_1, \ldots, t_n)) \ \rho \quad \text{// occur-check} \\
\quad \text{code}_I \ f(t_1, \ldots, t_n) \ \rho \quad \text{// building} \\
\quad \text{bind} \quad \text{// creation of bindings} \\
\quad B : \ldots
\]
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 if the binding of no variable inside \( t' \) contains a reference to \( X' \).

\[ \text{ivars}(t') \Rightarrow \text{set of already initialized variables of } t. \]

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

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

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

\[
\text{code}_{\text{eff}} f(t_1, \ldots, t_n) \rho = \begin{align*}
&\text{\# test} \\
&\text{son } 1 \\
&\text{code}_{\text{eff}}(t_1 \rho) \\
&\ldots \\
&\text{son } n \\
&\text{code}_{\text{eff}}(t_n \rho) \\
&\text{up } B \\
&A: \text{check ivars}(f(t_1, \ldots, t_n)) \rho \quad \text{// occur-check} \\
&B: \text{bind} \quad \text{// building} \\
&\text{bind} \quad \text{// creation of bindings}
\end{align*}
\]

The instruction \( \text{check} i \) 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:

\[
\text{if (check } S[S[\text{SP}]], \text{ deref } S[\text{FP}+i]) \quad \text{backtrack();}
\]

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

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

\[
\begin{align*}
\text{bind} \\
&\text{bind} \\
&\text{bind}
\end{align*}
\]
• The unification code performs a pre-order traversal over $t$.
• In case, execution hits at an unbound variable, we switch from checking to building.

```plaintext
code t \_ (\ldots, t_n) \rho =
  \text{\texttt{ustruct}} f/\!\!/A \quad \text{// test}
  \text{\texttt{son}} \ 1
  code t_1 \rho
  \ldots
  \text{\texttt{son}} \ n
  code t_n \rho
  \text{\texttt{up}} B
  \text{\texttt{A : check}} \ \text{\text{\texttt{ivars}}}(f(t_1, \ldots, t_n)) \rho \quad \text{// occur-check}
  code f(t_1, \ldots, t_n) \rho \quad \text{// building !!}
  bind \quad \text{// creation of bindings}
B : \ldots
```

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 $f/\!\!/$.
• 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:

```plaintext
code t \_ (\ldots, t_n) \rho =
  \text{\texttt{ustruct}} f/\!\!/A \quad \text{// test}
  \text{\texttt{son}} \ 1 \quad \text{// recursive descent}
  code t_1 \rho
  \ldots
  \text{\texttt{son}} \ n \quad \text{// recursive descent}
  code t_n \rho
  \text{\texttt{up}} B \quad \text{// ascent to father}
A : \ \text{\texttt{check}} \ \text{\text{\texttt{ivars}}}(f(t_1, \ldots, t_n)) \rho
  code f(t_1, \ldots, t_n) \rho
  bind
B : \ldots
```

The instruction $\text{\texttt{ustruct}}$ implements the test of the root node of a structure:

```plaintext
\text{\texttt{switch}} (\text{\texttt{H}[S[SP]])\{ \\
  \text{\texttt{case}} (S, b/n) : \text{\text{\texttt{break}}}; \\
  \text{\texttt{case}} (R,_) : \text{\text{\texttt{PC}} = A; \text{\texttt{break}}}; \\
  \text{\texttt{default}} : \text{\texttt{backtrace}}(); \\
\}

... the argument reference is not yet popped. \rangle
```
The instruction `son i` pushes the (reference to the) i-th sub-term from the structure pointed at from the topmost reference:

\[ S[SP+1] = \text{deref} (H[S[SP]+1]); \ SP++; \]

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`.

Example:

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

\[
\begin{array}{lllllll}
\text{ustruct f/3} & A_1 & \text{up} B_2 & B_3: & \text{son} 2 & \text{putvar} 2 \\
\text{son} 1 & \text{uatom} a & \text{putatom} a \\
\text{ustruct g/2} & A_2: & \text{check} 1 & \text{son} 3 & \text{putvar} 3 \\
\text{son} 1 & \text{putref} 1 & \text{uvar} 3 & \text{putvar} 3 \\
\text{uref} 1 & \text{putvar} 2 & \text{up} B_1 & \text{putstruct f/3} \\
\text{son} 2 & \text{putstruct g/2} & A_1: & \text{check} 1 & \text{bind} \\
\text{uvar} 2 & \text{bind} & \text{putref} 1 & B_1: & \ldots \\
\end{array}
\]

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

Example:

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

\[
\begin{array}{lllllll}
\text{ustruct f/3} & A_1 & \text{up} B_2 & B_3: & \text{son} 2 & \text{putvar} 2 \\
\text{son} 1 & \text{uatom} a & \text{putatom} a \\
\text{ustruct g/2} & A_2: & \text{check} 1 & \text{son} 3 & \text{putvar} 3 \\
\text{son} 1 & \text{putref} 1 & \text{uvar} 3 & \text{putvar} 3 \\
\text{uref} 1 & \text{putvar} 2 & \text{up} B_1 & \text{putstruct f/3} \\
\text{son} 2 & \text{putstruct g/2} & A_1: & \text{check} 1 & \text{bind} \\
\text{uvar} 2 & \text{bind} & \text{putref} 1 & B_1: & \ldots \\
\end{array}
\]

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

286

287

288
32 Clauses

Clausal code must

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

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

Let \( \{ X_1, \ldots, X_n \} \) 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.

Then we translate:

\[
\text{code}_C \; r &= \ \text{pushenv} \; m \quad // \text{allocates space for locals} \\
\text{code}_C \; g_1 \; \rho \\
\vdots \\
\text{code}_C \; g_n \; \rho \\
popenv
\]

The instruction \text{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.

Then we translate:

The instruction \text{pushenv} \; m sets the stack pointer:

\[
FP \xrightarrow{\text{pushenv} \; m} FP + m; \quad \text{SP} = \text{FP} + m;
\]
33 Predicates

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

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

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

\[
\text{code}_f \ r_f = \text{code}_c \ r_1
\]

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

On failure, the next alternative must be tried

\[
\implies \text{backtracking} \implies
\]

33.1 Backtracking

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

\[
\begin{array}{c|c|c}
\text{T} & \text{TP} \\
\hline
0 & \text{TP} & \text{Trail Pointer} \\
& & \text{points to the topmost occupied Trail cell}
\end{array}
\]
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:

For more comprehensible notation, we thus introduce the macros:

\[
\begin{align*}
\text{posCont} & \equiv S[\text{FP}] \\
\text{FPold} & \equiv S[\text{FP} - 1] \\
\text{HPold} & \equiv S[\text{FP} - 2] \\
\text{TPold} & \equiv S[\text{FP} - 3] \\
\text{BPold} & \equiv S[\text{FP} - 4] \\
\text{negCont} & \equiv S[\text{FP} - 5]
\end{align*}
\]

for the corresponding addresses.

**Remark:**

Occurrence on the left \(\equiv\) saving the register

Occurrence on the right \(\equiv\) restoring the register

Calling the run-time function \(\text{void backtrack()}\) yields:

\[
\begin{align*}
\text{posCont} & \quad 0 \\
\text{FPold} & \quad -1 \\
\text{HPold} & \quad -2 \\
\text{TPold} & \quad -3 \\
\text{BPold} & \quad -4 \\
\text{negCont} & \quad -5
\end{align*}
\]

\[
\begin{align*}
\text{FP} & \quad \text{backtrack()}; \\
\text{FP} & \quad \text{FP = BP}; \quad \text{HP} = \text{HPold}; \\
& \quad \text{reset}(\text{TPold}, \text{TP}); \\
& \quad \text{TP} = \text{TPold}, \quad \text{PC} = \text{negCont};
\end{align*}
\]

where the run-time function **reset()** undoes the bindings of variables established **since** the backtrack point.
Calling the run-time function

```c
void backtrack()
```

yields:

```c
void backtrack()
{
    FP = BP; HP = HPold;
    reset (TPold, TP);
    TP = TPold; PC = negCont;
}
```

where the run-time function `reset()` undoes the bindings of variables established since the backtrack point.