- 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; }
```

\[
x = \overline{P}(x)
\]

Discussion

- The translation of an equation \( \overline{x} = t \) 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:

\[
\text{code} \gamma a \rho = \text{uatom } a \\
\text{code} \gamma X \rho = \text{uvar } (\rho X) \\
\text{code} \gamma \_ \rho = \text{pop} \\
\text{code} \gamma X \rho = \text{uref } (\rho X) \\
\ldots \quad // \text{to be continued}
\]

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

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

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

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!

\[
\begin{align*}
\text{code} \gamma (\hat{X} = t) \rho &= \text{put } \hat{X} \rho \\
\text{code} \gamma t \rho &
\end{align*}
\]
Let us first consider the unification code for atoms and variables only:

\[
\begin{align*}
\text{code02 } \texttt{a } & \texttt{ = uatom } \texttt{a} \\
\text{code02 } \bar{x} \rho & \texttt{ = uvar } \( \bar{x} \ X \) \\
\text{code02 } \_ \rho & \texttt{ = pop} \\
\text{code02 } \bar{x} \rho & \texttt{ = uref } \( \bar{x} \ X \) \\
\_ & \texttt{// to be continued}
\end{align*}
\]

The instruction \texttt{utom a} implements the unification with the atom \( a \):

\[
\texttt{utom a}
\]

\[
\texttt{R} \rightarrow \texttt{a}
\]

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

- 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:

\[
\texttt{FP+i} \rightarrow \texttt{uvar i} \rightarrow \texttt{FP+i}
\]

\[
\texttt{S[FP+i]} = \texttt{S[SP]}; \texttt{SP}--; \\
\]

The instruction \texttt{pop} implements the unification with an anonymous variable. It always succeeds.

\[
\texttt{pop} \\
\]

\[
\texttt{SP}--; \\
\]
The instruction `uref i` implements the unification with an initialized variable:

\[ \theta = \text{mg} \alpha_{x, y} \]

\[ \text{unify}(S[SP], \text{deref}(S[FP+1])); \]

\[ \text{SP} = \cdots; \]

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

\[ f(a, x, y) = f(x, z, t) \]

\[ x = a \]
\[ z = a \]
\[ y = a \]

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

\[ \text{code}_{A} f(t_{1}, \ldots, t_{n}) \rho = \text{struct/}n \; A \]

\[ \text{son/}1 \]
\[ \text{code}_{A} t_{1} \rho \]
\[ \ldots \]
\[ \text{son/}n \]
\[ \text{code}_{A} t_{n} \rho \]

\[ \text{up/}B \]
\[ A : \text{check/}cons(f(t_{1}, \ldots, t_{n})) \rho \]
\[ \text{code}_{B} f(t_{1}, \ldots, t_{n}) \rho \]

\[ \text{bind} \]

\[ B : \ldots \]
The unification code performs a pre-order traversal over
the tree. In case, execution hits an unbound variable, we switch from checking to
building.

\[ f(t_1, \ldots, t_n) \rho = \text{struct } f \text{ in } A \]  
\[ \text{son } 1 \]
\[ \text{code } f(t_1) \rho \]
\[ \ldots \]
\[ \text{son } n \]
\[ \text{code } f(t_n) \rho \]
\[ \text{up } B \]
\[ A : \text{check } \text{invars}(f(t_1, \ldots, t_n)) \rho \]  
\[ \text{code}_A \ f(t_1, \ldots, t_n) \rho \]  
\[ \text{bind} \]  
\[ B : \ldots \]

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(SISP, deref(S[FP+i])))
  backtrack();
```
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:

\[
\text{code}_f(t_1, \ldots, t_n) \rho = \begin{cases} 
\text{ustruct } f/n A & \text{// test} \\
\text{son } 1 & \text{// recursive descent} \\
\text{code}_f t_1 \rho & \\
\ldots & \\
\text{son } n & \text{// recursive descent} \\
\text{code}_f t_n \rho & \\
\text{up B} & \text{// ascent to father} \\
\text{check } \text{vars}(f(t_1, \ldots, t_n)) \rho & \\
\text{code}_A f(t_1, \ldots, t_n) \rho & \\
\text{bind} & \\
A : & \\
B : & \ldots 
\end{cases}
\]

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

If so, unification fails and \text{backtracking} is caused:

\[
\text{if } (!\text{check } (S[SP], \text{deref } (S[F+1]))) \text{ backtrack();}
\]

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

\[
\begin{array}{c}
\text{ustruct } f/n A \\
\text{ustruct } f/n A
\end{array}
\]

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

... 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:

![Diagram of pushing a reference](image)

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

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

![Diagram of popping a reference](image)

\[ \text{SP} \leftarrow \text{PC} = B; \]

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:

<table>
<thead>
<tr>
<th>ustruct f/3 A₁</th>
<th>up B₂</th>
<th>B₂: son 2</th>
<th>putvar 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>son 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ustruct g/2 A₂ A₃: check 1</td>
<td>son 3</td>
<td>putatom a</td>
<td></td>
</tr>
<tr>
<td>son 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uref 1</td>
<td>putvar 2</td>
<td>up B₁</td>
<td>putvar 3</td>
</tr>
<tr>
<td>son 2</td>
<td>putstruct g/2 A₁: check 1</td>
<td>bind</td>
<td></td>
</tr>
<tr>
<td>uvar 2</td>
<td>bind</td>
<td>putref 1</td>
<td>B₁: ...</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>ustruct f/3 A₁</th>
<th>up B₂</th>
<th>B₂: son 2</th>
<th>putvar 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>son 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ustruct g/2 A₂ A₃: check 1</td>
<td>son 3</td>
<td>putatom a</td>
<td></td>
</tr>
<tr>
<td>son 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uref 1</td>
<td>putvar 2</td>
<td>up B₁</td>
<td>putvar 3</td>
</tr>
<tr>
<td>son 2</td>
<td>putstruct g/2 A₁: check 1</td>
<td>bind</td>
<td></td>
</tr>
<tr>
<td>uvar 2</td>
<td>bind</td>
<td>putref 1</td>
<td>B₁: ...</td>
</tr>
</tbody>
</table>

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

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

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

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.

Then we translate:

\[
\text{code}_r \begin{array}{c}
\text{pushenv } m \\
\text{code}_r g_1 \rho \\
... \\
\text{code}_r g_e \rho \\
\text{popenv}
\end{array} \quad /\!\!/ \text{allocates space for locals}
\]

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.

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

\[
\begin{array}{c}
\text{SP} = \text{FP} + m_i
\end{array}
\]

Example

Consider the clause \( r \):
\[
a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})
\]

Then \( \text{code}_r \begin{array}{c}
\text{pushenv } 3 \\
\text{mark A:} \\
\text{mark B:} \\
\text{popref 3} \\
\text{putref 2} \\
\text{call f/2} \\
\text{call a/2}
\end{array} \begin{array}{c}
\text{A:} \\
\text{B:}
\end{array}
\]

\[
\begin{array}{c}
\text{A:} \\
\text{B:}
\end{array}
\]

\[
\begin{array}{c}
\text{A:} \\
\text{B:}
\end{array}
\]

\[
\begin{array}{c}
\text{A:} \\
\text{B:}
\end{array}
\]

\[
\begin{array}{c}
\text{A:} \\
\text{B:}
\end{array}
\]
33 Predicates

A predicate \( q/k \) is defined through a sequence of clauses \( r_i \). The translation of \( q/k \) provides the translations of the individual clauses \( r_i \).

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

\[
\text{codep} \quad r_i = q/k : \quad \text{codec} \quad r_i
\]

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

\[ \rightarrow \text{backtracking} \]

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) 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 \( \text{trail} \).

The current stack frame where backtracking should return to is pointed at by the extra register \( \text{BP} \):

\( S \)

0

\( \text{TP} \)

Trail Pointer points to the topmost occupied Trail cell

\( \text{SP} \)

\( \text{FP} \)

\( \text{DY} \)
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
    FPoold -1
    HPold  -2
    TPold  -3
    BPold  -4
    negCont -5
```

For more comprehensible notation, we thus introduce the macros:

```
posCont  = S[FP]
FPoold   = S[FP - 1]
HPold    = S[FP - 2]
TPold    = S[FP - 3]
BPold    = S[FP - 4]
negCont  = S[FP - 5]
```

for the corresponding addresses.

**Remark**

Occurrence on the left    = saving the register
Occurrence on the right   = restoring the register

Calling the run-time function `void backtrack()` yields:

```
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.