Discussion:

- The translation of an equation \( \bar{X} = t \) is very simple \( \Rightarrow \)
- Often the constructed cells immediately become garbage \( \Leftarrow \)

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!!

\[
\text{code}_G (\bar{X} = t) \rho = \begin{array}{l}
\text{put } \bar{X} \rho \\
\text{code}_{\bar{X}} t \rho
\end{array}
\]

---

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

\[
\begin{align*}
\text{code}_{\bar{a}} \rho & = \text{uatom } a \\
\text{code}_{X} \rho & = \text{var } (\rho X) \\
\text{code}_{\_} \rho & = \text{pop} \\
\text{code}_{\bar{X}} \rho & = \text{ref } (\rho X) \\
\ldots & \quad // \text{ to be continued } \Rightarrow
\end{align*}
\]

---

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

Let the runtime function \( \text{trail}() \) records the a potential new binding.
- The run-time function \( \text{backtrack}() \) initiates backtracking.
The instruction `uvar i` implements the unification with an un-initialized variable:

\[ [FP+i] = [SP], SP--; \]

The instruction `pop` implements the unification with an anonymous variable. It always succeeds.

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

\[ \beta = \text{mgu}(x, y) \]

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.

\[
\begin{align*}
\text{code}_{s_1} f(t_1, \ldots, t_n) \rho &= \text{ustruct} f/A(\text{ustruct} f/A(A)) \\
&\quad + \text{code}_{s_1} t_1 \rho \\
&\quad + \ldots \\
&\quad + \text{code}_{s_n} t_n \rho \\
&\quad + \text{codes} f(t_1, \ldots, t_n) \rho \\
&\quad + \text{check-magic}(f(t_1, \ldots, t_n)) \rho \\
&\quad + \text{check-magic}(f(t_1, \ldots, t_n)) \rho \\
&\quad + \text{codes} f(t_1, \ldots, t_n) \rho \\
&\quad + \text{check-magic}(f(t_1, \ldots, t_n)) \rho
\end{align*}
\]
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) \( X' \).

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

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

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

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

```
code_{f}(t_1, \ldots, t_n) \rho =
  \begin{align*}
    &\text{\texttt{struct f/n A}} \quad \text{// test} \\
    &\text{son 1} \\
    &\text{code}_{t_1} \rho \\
    &\ldots \\
    &\text{son n} \\
    &\text{code}_{t_n} \rho \\
    &\text{up B} \\
  \end{align*}

A : \begin{align*}
  &\text{check \texttt{itars}(f(t_1, \ldots, t_n)) \rho} \quad \text{// occur-check} \\
  &\text{code}_{f}(t_1, \ldots, t_n) \rho \quad \text{// building} \\
  &\text{bind} \\
  \end{align*}

B : \ldots
```

The instruction \texttt{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;}
```

### The Pre-Order Traversal:

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

```
code_{f}(t_1, \ldots, t_n) \rho =
  \begin{align*}
    &\text{\texttt{struct f/n A}} \quad \text{// test} \\
    &\text{son 1} \\
    &\text{code}_{t_1} \rho \\
    &\ldots \\
    &\text{son n} \\
    &\text{code}_{t_n} \rho \\
    &\text{up B} \\
  \end{align*}

A : \begin{align*}
  &\text{check \texttt{itars}(f(t_1, \ldots, t_n)) \rho} \\
  &\text{code}_{f}(t_1, \ldots, t_n) \rho \\
  &\text{bind} \\
  \end{align*}

B : \ldots
```

Once again the unification code for constructed terms:

```
code_{f}(t_1, \ldots, t_n) \rho =
  \begin{align*}
    &\text{\texttt{struct f/n A}} \quad \text{// test} \\
    &\text{son 1} \\
    &\text{code}_{t_1} \rho \\
    &\ldots \\
    &\text{son n} \\
    &\text{code}_{t_n} \rho \\
    &\text{up B} \\
  \end{align*}

A : \begin{align*}
  &\text{check \texttt{itars}(f(t_1, \ldots, t_n)) \rho} \\
  &\text{code}_{f}(t_1, \ldots, t_n) \rho \\
  &\text{bind} \\
  \end{align*}

B : \ldots
```
The instruction \texttt{ustruct}\ i\ implements the test of the root node of a structure:

\begin{verbatim}
switch (H[S][SP]) {
  case (S.f/n):  break;
  case (R.n):  PC = A; break;
  default:    backtrack();
}
\end{verbatim}

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

The instruction \texttt{son}\ i\ pushes the (reference to the) \(i\)-th sub-term from the structure pointed at from the topmost reference:

\begin{verbatim}
S[SP+1] = deref (H[S][SP+1]); SP++;
\end{verbatim}

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

\begin{verbatim}
SP-- ; PC = B;
\end{verbatim}

The continuation address \texttt{B} is the next address after the build-section.
Example:

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

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

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

---

31 Clauses

Clausal code must

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

Let \( \tau \) denote the clause: \( p(X_1, \ldots, X_n) \leftarrow X_1, \ldots, X_n \).

Let \( \{ X_1, \ldots, X_n \} \) denote the set of locals of \( \tau \) and \( \rho \) the address environment:

\( \rho \ X_i = i \)

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

---

Then we translate:

\[
\begin{align*}
\text{code; } \tau & = \text{pushenv } m \quad // \text{allocates space for locals} \\
\text{code; } g_1 \rho & \quad \text{...} \\
\text{code; } g_n \rho & \quad \text{popenv}
\end{align*}
\]

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:

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

Example:

Consider the clause \( r \):

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

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

```
pushenv 3
mark(A) → A:
putref 1
putvar 3
putref 2
call f/2
call a/2
```

32 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 \( f = 1 \):

\[
\text{code}_P rr = \text{code}_C r_1
\]

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

```
backtracking :-(
```

32.1 Backtracking

- Whenever unification fails, we call the run-time function \( \text{backtrack}() \).
- The goal is to roll back the whole computation to the (dynamically last) goal where another clause can be chosen.
- 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 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:

```
FP → posCont. 0
    FFold -1
    HPold -2
    TFold -3
    BPold -4
    negCont. -5
```
For more comprehensible notation, we thus introduce the macros:

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

for the corresponding addresses.

**Remark:**

- Occurrence on the left: saving the register
- Occurrence on the right: restoring the register

Functions \texttt{void trail(ref u)} and \texttt{void reset(ref y, ref x)} can thus be implemented as:

\[
\begin{align*}
\text{void trail}(\text{ref } u) \{ \\
&\quad \text{if } (u < S[BP - 2]) \{ \\
&\quad\quad \text{TP} = TP + 1; \\
&\quad\quad H(T[u]) = (R, T[u]); \\
&\quad\quad T[TP] = u;
\}
\}
\end{align*}
\]

\[
\begin{align*}
\text{void reset}(\text{ref } z, \text{ref } y) \{ \\
&\quad \text{for } (\text{ref } w; x; u--) \{ \\
&\quad\quad \text{TP} = \text{TP} + 1; \\
&\quad\quad H(T[u]) = (R, T[u]); \\
&\quad\quad T[TP] = u;
\}
\}
\end{align*}
\]

Here, \( S[BP - 2] \) represents the heap pointer when creating the last backtrack point.
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.

Functions `void trail(ref u)` and `void reset (ref y, ref x)` can thus be implemented as:

```
void trail (ref u) {
    if (u < S[BP-2]) {
        for (ref w;y; x; u--)
        
        T[TP] = u;
        
    }
}
```

Here, \( S[BP-2] \) represents the heap pointer when creating the last backtrack point.

### 32.3 Wrapping it Up

Assume that the predicate \( q/k \) is defined by the clauses \( r_1, \ldots, r_f \) (\( f > 1 \)). We provide code for:

- setting up the backtrack point;
- successively trying the alternatives;
- deleting the backtrack point.

This means:

```
code p \( rr = q/k: \)
  `setbp`
  `try A_1`
  ...
  `try A_{f-1}`
  `delbp`
  `jump A_f`

A_1 : `code c r_1`
      ...
A_f : `code c r_f`
```

Note:
- We delete the backtrack point before the last alternative \( \rightarrow \)
- We jump to the last alternative — never to return to the present frame \( \rightarrow \)
Example:

\[ s(X) \leftarrow t(X) \]
\[ s(X) \leftarrow \text{\textit{X}} = a \]

The translation of the predicate \( s \) yields:

s/1:  setbtp  try A  delbtp  jump B
      A:  pushenv 1  mark C  putref 1  call t/1
      \hspace{1cm} B:  pushenv 1  \hspace{1cm} C:  popenv

The instruction \( \text{setbtp} \) saves the registers HP, TP, BP:

HPold = HP;  
TPold = TP;  
BPold = BP;  
BP = FP;

The instruction \( \text{try A} \) tries the alternative at address A and updates the negative continuation address to the current PC:

negForts = PC;  
PC = A;

The instruction \( \text{delbtp} \) restores the old backtrack pointer:

BP = BPold;
32.4 Popping of Stack Frames

Recall the translation scheme for clauses:

\[
\text{code}_c \cdot r = \text{pushenv } m \\
\text{code}_c \cdot x; p \\
\ldots \\
\text{code}_c \cdot y; p \\
\text{popenv}
\]

The present stack frame can be popped ...
- if the applied clause was the last (or only); and
- if all goals in the body are definitely finished.

\[\Rightarrow \text{ the backtrack point is older } \}
\[\Rightarrow \text{ FP > BP}
\]

The instruction \text{popenv} restores the registers FP and PC and possibly pops the stack frame:

\[
\begin{align*}
\text{FP} & \rightarrow 42 \\
\text{PC} & \rightarrow 42 \\
\text{BP} & \rightarrow \text{FPold}
\end{align*}
\]

If (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;

Warning: \text{popenv} may fail to de-allocate the frame !!!