Representing $t \equiv f(g(X, Y), a, Z)$ : 

Reference to $X$
For a distinction, we mark occurrences of already initialized variables through **over-lining** (e.g. $\bar{X}$).

**Note:** Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_A a \rho & = \text{putatom } a \\
\text{code}_A X \rho & = \text{putvar } (\rho X) \\
\text{code}_A \bar{X} \rho & = \text{putref } (\rho X) \\
\text{code}_A _\ldots \rho & = \text{putanon}
\end{align*}
\]

\[
\begin{align*}
\text{code}_A f(t_1, \ldots, t_n) \rho & = \text{code}_A t_1 \rho \\
& \quad \ldots \\
& \quad \text{code}_A t_n \rho \\
& \quad \text{putstruct } f/n
\end{align*}
\]
For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\overline{X}$).

Note: Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_A a \rho &= \text{putatom } a \\
\text{code}_A X \rho &= \text{putvar } (\rho X) \\
\text{code}_A \overline{X} \rho &= \text{putref } (\rho X) \\
\text{code}_A _- \rho &= \text{putanon} \\
\text{code}_A f(t_1, \ldots, t_n) \rho &= \text{code}_A t_1 \rho \\
\end{align*}
\]

For $f(g(\overline{X}, Y), a, Z)$ and $\rho = \{ X \mapsto 1, Y \mapsto 2, Z \mapsto 3 \}$ this results in the sequence:

\[
\begin{align*}
\text{putref } 1 & \quad \text{putatom } a \\
\text{putvar } 2 & \quad \text{putvar } 3 \\
\text{putstruct } g/2 & \quad \text{putstruct } f/3 \\
\end{align*}
\]
The instruction `putatom a` constructs an atom in the heap:

\[ SP++; S[SP] = \text{new} (A,a); \]
The instruction **putvar i** introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:

\[
\begin{align*}
SP &= SP + 1; \\
S[SP] &= \text{new (R, HP);} \\
S[FP + i] &= S[SP];
\end{align*}
\]
The instruction `putanon` introduces a new unbound variable but does not store a reference to it in the stack frame:

\[
\text{SP} = \text{SP} + 1; \\
\text{S}[\text{SP}] = \text{new} (\text{R}, \text{HP});
\]
The instruction `putref i` pushes the value of the variable onto the stack:

\[
\begin{align*}
SP &= SP + 1; \\
S[SP] &= \text{deref } S[FP + i];
\end{align*}
\]
The instruction `putref i` pushes the value of the variable onto the stack:

```
SP = SP + 1;
S[SP] = deref S[FP + i];
```

The auxiliary function `deref` contracts chains of references:

```cpp
ref deref (ref v) {
    if (H[v]==(R,w) && v!=w) return deref (w);
    else return v;
}
```
The instruction \texttt{putstruct f/n} builds a constructor application in the heap:

\begin{Verbatim}
\begin{verbatim}
v = new (S, f, n);
SP = SP - n + 1;
for (i=1; i<=n; i++)
    H[v + i] = S[SP + i -1];
S[SP] = v;
\end{verbatim}
\end{Verbatim}
Remarks:

- The instruction `putref i` does not just push the reference from `S[FP + i]` onto the stack, but also dereferences it as much as possible
  \[ \rightarrow \text{maximal contraction of reference chains.} \]

- In constructed terms, references always point to smaller heap addresses. Also otherwise, this will be often the case. Sadly enough, it cannot be guaranteed in general. :-(

29 The Translation of Literals (Goals)

Idea:

- Literals are treated as procedure calls.
- We first allocate a stack frame.
- Then we construct the actual parameters (in the heap)
- ... and store references to these into the stack frame.
- Finally, we jump to the code for the procedure/predicate.
$$\text{code}_G p(t_1, \ldots, t_k) \rho = \begin{align*}
\text{mark } B & \quad \text{// allocates the stack frame} \\
\text{code}_A t_1 \rho & \\
\ldots & \\
\text{code}_A t_k \rho & \\
\text{call } p/k & \quad \text{// calls the procedure } p/k \\
B : & \quad \ldots
\end{align*}$$
\[ \text{code}_C \ p(t_1, \ldots, t_k) \ \rho \ =\ \text{mark } B \quad \text{// allocates the stack frame} \]
\[ \text{code}_A \ t_1 \ \rho \]
\[ \ldots \]
\[ \text{code}_A \ t_k \ \rho \]
\[ \text{call } p/k \quad \text{// calls the procedure } p/k \]

**Example:** \[ p(a, X, g(\bar{X}, Y)) \] with \[ \rho = \{ X \mapsto 1, Y \mapsto 2 \} \]

We obtain:

\[ \begin{align*}
\text{mark } B & \quad \text{putref } 1 & \quad \text{call } p/3 \\
\text{putatom } a & \quad \text{putvar } 2 & \quad \text{B: } \ldots \\
\text{putvar } 1 & \quad \text{putstruct } g/2
\end{align*} \]
Stack Frame of the WiM:

- SP
- FP
- posCont.
- FPold
- local stack
- local variables
- 6 org. cells

241
Remarks:

• The positive continuation address records where to continue after successful treatment of the goal.

• Additional organizational cells are needed for the implementation of backtracking

⇒ will be discussed at the translation of predicates.
The instruction \textbf{mark B} allocates a new stack frame:

\begin{align*}
&\text{mark B} \\
&\text{SP} = \text{SP} + 6; \\
&S[\text{SP}] = B; S[\text{SP}-1] = \text{FP};
\end{align*}
The instruction \texttt{call p/n} calls the \texttt{n}-ary predicate \texttt{p}:

\[
\begin{align*}
\text{FP} &= \text{SP} - n; \\
\text{PC} &= \text{p/n};
\end{align*}
\]
30 Unification

Convention:

• By $\tilde{X}$, we denote an occurrence of $X$; it will be translated differently depending on whether the variable is initialized or not.

• We introduce the macro $\text{put } \tilde{X} \rho$:

$$
\begin{align*}
\text{put } X \rho &= \text{putvar } (\rho X) \\
\text{put } _- \rho &= \text{putanon} \\
\text{put } \tilde{X} \rho &= \text{putref } (\rho X)
\end{align*}
$$
Let us translate the unification $\tilde{X} = t$.

**Idea 1:**

- Push a reference to (the binding of) $X$ onto the stack;
- Construct the term $t$ in the heap;
- Invent a new instruction implementing the unification $\Code{G(\tilde{X} = t) /AQ}$ put $\tilde{X}/AQ$ $\Code{A(t)/AQ}$ $\Code{\text{unify}}$ $\Code{246}$
Let us translate the unification $\bar{X} = t$.

**Idea 1:**
- Push a reference to (the binding of) $X$ onto the stack;
- Construct the term $t$ in the heap;
- Invent a new instruction implementing the unification $:-)$

$$\text{code}_G (\bar{X} = t) \rho = \text{put} \; \bar{X} \; \rho$$
$$\text{code}_A t \; \rho$$
$$\text{unify}$$
Example:

Consider the equation:

\[ \bar{U} = f(g(\bar{X}, Y), a, Z) \]

Then we obtain for an address environment

\[ \rho = \{ X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4 \} \]
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  

\textbf{unify()}  

- ... takes two heap addresses.  
  For each call, we guarantee that these are \textbf{maximally de-referenced}.  
- ... checks whether the two addresses are already \textbf{identical}.  
  If so, does nothing  :-(  
- ... binds \textbf{younger variables} (larger addresses) to \textbf{older variables} (smaller addresses);  
- ... when binding a variable to a term, checks whether the variable occurs inside the term \textbf{occur-check};  
- ... \textbf{records} newly created bindings;  
- ... may \textbf{fail}. Then \textbf{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 \texttt{trail()} records the a potential new binding.
• The run-time function \texttt{backtrack()} initiates \texttt{backtracking}.
• The auxiliary function \texttt{check()} performs the \texttt{occur-check}: it tests whether a variable (the first argument) \texttt{occurs inside} a term (the second argument).
• Often, this check is skipped, i.e.,

\begin{verbatim}
bool check (ref u, ref v) { return true;}
\end{verbatim}
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 $\tilde{X} = t$ 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 unification 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 unification with this node !!

\[
\text{code}_G (\tilde{X} = t) \rho = \text{put} \tilde{X} \rho \\
\text{code}_U t \rho
\]
Let us first consider the unification code for atoms and variables only:

\[
\begin{align*}
\text{code}_U a \rho & = \text{uatom } a \\
\text{code}_U X \rho & = \text{uvar } (\rho X) \\
\text{code}_U _\rho & = \text{pop} \\
\text{code}_U \bar{X} \rho & = \text{uref } (\rho X)
\end{align*}
\]

... // to be continued  :-)}
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 \texttt{uvar i} implements the unification with an un-initialized variable:

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

```
SP--;
```

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

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

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

It is only here that the run-time function \texttt{unify()} is called  \\(\text{:-})\)
• The unification code performs a pre-order traversal over $t$.

• In case, execution hits at an unbound variable, we switch from checking to building :-)

\[
\text{code}_U \ f(t_1, \ldots, t_n) \ \rho = \begin{cases} 
\text{ustruct } f/n \ A \\
\text{son } 1 \\
\text{code}_U \ t_1 \ \rho \\
\ldots \\
\text{son } n \\
\text{code}_U \ t_n \ \rho \\
\text{up } B \\
A: \ \text{check } \text{ivars}(f(t_1, \ldots, t_n)) \ \rho \quad \text{// occur-check} \\
\text{code}_A \ f(t_1, \ldots, t_n) \ \rho \quad \text{// building !!} \\
\text{bind} \\
B: \ \ldots
\end{cases}
\]
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 iff the binding of no variable inside $t'$ contains (a reference to) $X'$.

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

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

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

```c
if (!check (S[SP], deref S[FP+i]))
    backtrack();
```
The instruction `bind` terminates the building block. It binds the (unbound) variable to the constructed term:

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

\[
\text{code}_{U} f(t_1, \ldots, t_n) \rho = \begin{cases} 
\text{struct } f/n A & \quad \text{// test} \\
\text{son 1} & \quad \text{// recursive descent} \\
\text{code}_{U} t_1 \rho \\
\ldots \\
\text{son } n & \quad \text{// recursive descent} \\
\text{code}_{U} t_n \rho \\
\text{up } B & \quad \text{// ascent to father} \\
A : \quad \text{check } ivars(f(t_1, \ldots, t_n)) \rho \\
\text{code}_{A} f(t_1, \ldots, t_n) \rho \\
\text{bind} \\
B : \quad \ldots
\end{cases}
\]
The instruction \texttt{ustruct i} implements the test of the root node of a structure:

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

... the argument reference is \textbf{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:

$$\text{S}[\text{SP}+1] = \text{deref}(\text{H}[\text{S}[\text{SP}+i]]); \text{SP}++;$$
It is the instruction \textit{up B} which finally pops the reference to the structure:

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

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

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

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

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

276