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.
\[
\text{code}_G \ p(t_1, \ldots, t_k)_{\alpha} = \text{mark } B \quad \quad \quad // \text{allocates the stack frame}
\]
\[
\text{code}_A \ t_1_{\alpha}
\]
\[
\ldots
\]
\[
\text{code}_A \ t_k_{\alpha}
\]
\[
call \ p/k \quad \quad \quad // \text{calls the procedure } p/k
\]
\[
B: \quad \ldots
\]
\[ \text{code}_G p(t_1, \ldots, t_k) \equiv \begin{array}{l}
\text{mark B} \quad //\text{allocates the stack frame} \\
\text{code}_A t_1 \equiv \\
\ldots \\
\text{code}_A t_k \equiv \\
\text{call p/k} \quad //\text{calls the procedure p/k} \\
B : \ldots
\end{array} \]

Example: \( p(a, X, g(\bar{X}, Y)) \) with \( \alpha = \{ X \mapsto 1, Y \mapsto 2 \} \)

We obtain:

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

- SP
- FP
- posCont.
- FPold
- local stack
- local variables
- 0
- -1
- -2
- -3
- -4
- -5
- 6 org. cells
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 `mark B` allocates a new stack frame:

\[
\text{FP} \quad \text{FP} \\
\text{B} \quad \\
\text{SP} = \text{SP} + 6; \\
\text{S}[\text{SP}] = \text{B}; \text{S}[\text{SP}-1] = \text{FP};
\]
The instruction \texttt{call p/n} calls the \textit{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 $\text{put } \tilde{X}$ $\sigma$:

\[
\begin{align*}
\text{put } X \sigma &= \text{putvar } (\sigma X) \\
\text{put } \_ \sigma &= \text{putanon} \\
\text{put } \bar{X} \sigma &= \text{putref } (\sigma X)
\end{align*}
\]
Let us translate the unification $\hat{X} = t$.

**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 $\hat{X} = t$.

:-)
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 $\vdash$)

$$\text{code}_G (\tilde{X} = t) \xrightarrow{x} = \text{put} \tilde{X} \xrightarrow{x}$$
$$\text{code}_A t \xrightarrow{x}$$
$$\text{unify}$$
Example:

Consider the equation:

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

Then we obtain for an address environment

$$\alpha = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

\begin{align*}
\text{putref } 4 & \quad \text{putref } 1 & \quad \text{putatom } a & \quad \text{unify} \\
\text{putvar } 2 & \quad \text{putvar } 3 \\
\text{putstruct } g/2 & \quad \text{putstruct } f/3
\end{align*}
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 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.
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 `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:

```cpp
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 !!

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

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

... // to be continued  :-)

257
The instruction \texttt{uatom a} implements the unification with the atom \texttt{a}:

\begin{verbatim}
  uatom 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();
  }
\end{verbatim}

- 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:

\begin{equation}
S[FP+i] = S[SP]; \ SP- -;
\end{equation}
The instruction \texttt{pop} implements the unification with an anonymous variable. It always succeeds \texttt{:-)}
The instruction `uref i` implements the unification with an initialized variable:

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

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

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

\[
\text{It is only here that the run-time function } \text{unify()} \text{ is called :)}
\]
• The unification code performs a pre-order traversal over $t$.
• In case, execution hits at an unbound variable, we switch from checking to building 

$$\text{code}_U f(t_1, \ldots, t_n) = \text{ustruct } f/n A$$

```
son 1
\text{code}_U t_1
\ldots
son n
\text{code}_U t_n
\text{up } B
A: \text{check } ivars(f(t_1, \ldots, t_n))$$

```

$$= \text{ustruct } f/n A$$

```
A: \text{check } ivars(f(t_1, \ldots, t_n))$$

```
\text{code}_A f(t_1, \ldots, t_n)$$

```
\text{bind}$$

```
B: \ldots
```

// test

// occur-check

// building !!

// creation of bindings
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) $X'$.

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

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

\[\text{check } \{Y_1, \ldots, Y_d\} = \text{check } (\alpha \ Y_1) \]
\[\text{check } (\alpha \ Y_2) \]
\[\text{...} \]
\[\text{check } (\alpha \ Y_d) \]
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:

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

\[
\text{trail (S[SP-1]);}
\]

\[
\text{SP = SP - 2;}
\]
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:

\[
\text{code}_U f(t_1, \ldots, t_n) = \text{struct } f / n \ A \quad /\ \text{// test}
\]

son 1 \quad /\ \text{// recursive descent}

\text{code}_U t_1 \quad /

\ldots

son n \quad /\ \text{// recursive descent}

\text{code}_U t_n \quad /

\text{up } B \quad /\ \text{// ascent to father}

\text{A} : \ \text{check } \text{ivars}(f(t_1, \ldots, t_n)) \quad /

\text{bind}

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

```
switch (H[S[SP]]) {
    case (S, f/n): break;
    case (R, _): PC = A; break;
    default: backtrack();
}

... 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:

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

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

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

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

\[
\begin{array}{ll}
\text{ustruct f/3 A}_1 & \text{up } B_2 & B_2: \text{ son 2} & \text{putvar 2} \\
\text{son 1} & & & \\
\text{ustruct g/2 A}_2 & A_2: \text{ check 1} & \text{son 3} & \text{putatom a} \\
\text{son 1} & \text{putref 1} & \text{uvar 3} & \text{putvar 3} \\
\text{uref 1} & \text{putvar 2} & \text{up } B_1 & \text{putstruct f/3} \\
\text{son 2} & \text{putstruct g/2} & A_1: \text{ check 1} & \text{bind} \\
\text{uvar 2} & \text{bind} & \text{putref 1} & B_1: \text{ ...} \\
\end{array}
\]

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

Let $\{X_1, \ldots, X_m\}$ denote the set of locals of $r$ and $\alpha$ the address environment:

$$\alpha X_i = i$$

Remark: The first $k$ locals are always the formals :-}
Then we translate:

\[
\text{code}_C \quad r = \begin{array}{l}
\text{pushenv m} \quad // \text{allocates space for locals} \\
\text{code}_C \quad g_1 \quad x \\
\text{...} \\
\text{code}_C \quad g_n \quad x \\
popenv
\end{array}
\]

The instruction \texttt{popenv} restores FP and PC and \texttt{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 $r$:

$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Then $\text{code}_C r$ yields:

```
pushenv 3  mark A  A:  mark B  B:  popenv
putref 1
putvar 3
putref 2
```

call $f/2$

call $a/2$