28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment \( \rho \) returns, for each clause variable \( X \) its address (relative to \( FT \) on the stack. Then \( \text{code}_A, t \rho \) should ... 

- construct a presentation of \( t \) in the heap; and
- return a reference to it on top of the stack.

Idea:

- Construct the tree during a post-order traversal of \( t \)
- with one instruction for each new node!

Example: \( t \equiv f(g(X,Y),a,Z) \).
Assume that \( X \) is initialized, i.e., \( \text{SIFP} + \rho X \) contains already a reference, \( Y \) and \( Z \) are not yet initialized.

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. \( X \)).

Note: Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_A a \rho &= \text{putatom} a \\
\text{code}_A f(t_1, \ldots, t_n) \rho &= \text{code}_A t_1 \rho \\
\text{code}_A X \rho &= \text{putvar} (\rho X) \\
\text{code}_A _X \rho &= \text{putref} (\rho X) \\
\text{code}_A _- \rho &= \text{putanon} \\
\end{align*}
\]
For a distinction, we mark occurrences of already initialized variables through overlining (e.g. $\bar{X}$).

**Note:** Arguments are always initialized!

Then we define:

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

For $f(g(\bar{X}, Y), u, Z)$ and $\rho = \{ X \rightarrow 1, Y \rightarrow 2, Z \rightarrow 3 \}$ 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:

\[
\text{SP} \rightarrow \text{SP} + 1; \\
S[\text{SP}] = \text{new} (A, a);
\]

The instruction **putvar** $i$ introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:

\[
\text{SP} = \text{SP} + 1; \\
S[\text{SP}] = \text{new} (R, \text{HP}); \\
S[\text{FP} + 1] = S[\text{SP}];
\]

The instruction **putanon** introduces a new unbound variable but does not store a reference to it in the stack frame:

\[
\text{SP} = \text{SP} + 1; \\
S[\text{SP}] = \text{new} (R, \text{HP});
\]
The instruction \texttt{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 auxiliary function \texttt{deref} contracts chains of references:

\[
\begin{align*}
\text{ref} \text{ deref} (\text{ref } v) &\{ \\
&\text{if } (S[v] = (v, u) \land u = v) \text{ return } \text{deref } (u); \\
&\text{else return } v; \\
\} \\
\end{align*}
\]

The instruction \texttt{putstruct f/n} builds a constructor application in the heap:

\[
\begin{align*}
v &= \text{new } (S, f, n); \\
SP &= SP - n + 1; \\
\text{for } (i = 1; i <= n; i++) \\
S[F + i] &= S[SP + i - 1]; \\
S[SP] &= v; \\
\end{align*}
\]

Remarks:

- The instruction \texttt{putref i} does not just push the reference from \(S[FP + i]\) onto the stack, but also dereferences it as much as possible.
- 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. :(
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:

\[
\text{ref deref (ref } v \}\}
\[
\text{if } (H[v] = (k, u) \text{ and } v \equiv u) \text{ return deref } (u);\]
\[
\text{else return } v;\}
\]

For a distinction, we mark occurrences of already initialized variables through delining (e.g., \(X\)).

Note: Arguments are always initialized!

Then we define:

\[
\text{code}_{\lambda} a \rho = \text{putatom } a
\]
\[
\text{code}_{\lambda} f(t_1, \ldots, t_n) \rho = \text{code}_{\lambda} t_1 \rho
\]
\[
\text{code}_{\lambda} X \rho = \text{putvar } (\rho X)
\]
\[
\text{code}_{\lambda} X \rho = \text{putref } (\rho X)
\]
\[
\text{code}_{\lambda} \_ \rho = \text{putanon}
\]

\[
\text{putstruct } t/n
\]

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

putref 1  putatom a
putvar 2  putvar 3
putstruct g/2  putstruct t/3

Example:

\(p(a, X, g(X, Y))\) with \(\rho = \{ X \mapsto 1, Y \mapsto 2 \}\)

We obtain:

mark B  putref 1  call p/k
putatom a  putvar 2  B: ...
putvar 1  putstruct g/2
Stack Frame of the WiM:

- **SP**
- **FP**

```
+---------------+------------------+
|               | local stack      |
|               | local variables  |
+---------------+------------------+
        | 0               |
         | -1              |
         | -2              |
         | -3              |
         | -4              |
         | -5              |
+---------------+------------------+
```

**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. Additional organizational cells will be discussed at the translation of predicates.

The instruction **mark B** allocates a new stack frame:

- **SP** = SP + 6;
- **S[SP]** = B;
- **S[SP-1]** = FP;

The instruction **call p/n** calls the n-ary predicate **p**:

- **FP** = SP - n;
- **PC** = p/n;
30 Unification

Convention:

- By $\hat{X}$, we denote an occurrence of $X$; it will be translated differently depending on whether the variable is initialized or not.
- We introduce the macro \textbf{put} $\hat{X} \rho$ :

  \[
  \begin{align*}
  \text{put} \ X \rho & = \text{putvar} \ (\rho \ X) \\
  \text{put} \_ \rho & = \text{putanon} \\
  \text{put} \hat{X} \rho & = \text{putref} \ (\rho \ X)
  \end{align*}
  \]

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 $\therefore$

\[
\text{code}_a \ (X = t) \rho = \text{put} \ X \rho \\
  \text{code}_a \ t \rho \\
  \text{unify}
\]

Let us translate the unification $\hat{X} = t$.

Example:

Consider the equation:

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

Then we obtain for an address environment

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

\[
\begin{align*}
\text{putref} & 4 \\
\text{putref} & 1 \\
\text{putatom} & a \\
\text{unify} & \\
\text{putvar} & 2 \\
\text{putvar} & 3 \\
\text{putstruct} & g/2 \\
\text{putstruct} & t/3
\end{align*}
\]
The instruction `unify` calls the run-time function `unify()` for the topmost two references:

```
unify (S[SP-1], S[SP]);
SP = SP-2;
```

Let us translate the unification $X = t$.

**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_{\rho} (X = t) \rho = \text{put } X \rho
code_{\rho} t \rho
\text{unify}
```

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.
The Function `unify()`

- ... takes two heap addresses.
  - For each call, we guarantee that these are maximally dereferenced.
- ... 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 `==` ocurrence-check;
- ... records newly created bindings;
- ... may fail. Then `backtrack` is initiated.

```c
bool unify(ref u, ref v) {
    if (u == v) return true;
    if (H[u] == (R,_.)) {
        if (H[v] == (R,_.)) {
            if (uu == v) {
                H[u] = (R,v); trail (u); return true;
            } else {
                H[v] = (R,u); trail (v); return true;
            }
        } else if (check (u,v)) {
            H[u] = (R,v); trail (u); return true;
        } else {
            backtrack(); return false;
        }
    } else if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else if (check (v,u)) {
        H[u] = (R,v); trail (u); return true;
    } else {
        backtrack(); return false;
    }
    ...
}
```

```c
... if (H[v] == (R,_.)) {
    if (check (v,u)) {
        H[v] = (R,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
} else if (check (v,u)) {
    H[v] = (R,u); trail (v); return true;
} else if (check (v,u)) {
    H[u] = (R,v); trail (u); return true;
} else {
    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.,

```c
bool check(ref u, ref v) { return true; }
```

Discussion:

- The translation of an equation $\bar{X} = \bar{t}$ is very simple $\rightarrow$
- Often the constructed cells immediately become garbage $\rightarrow$

Idea 2:

- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of $\bar{t}$ whenever possible!
- Translate each node of $\bar{t}$ into an instruction which performs the unification with this node $!!$

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] == ($, f/n)) {
        for (int i=1; i<=n; i++)
            if (!check(u, deref(H[v+i])))
                return false;
        return true;
    }
}
```

```
Y=f(f(a))
```

```c
... 
    if (H[v] == ($, _)) {
        if (check(v,u)) {
            H[v] = (a,v); trail(v); return true;
        } else {
            backtrack(); return false;
        }
    }
    backtrack(); return false;
}
```

```c
if (H[u] == ($, f/n) && H[v] == ($, 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;
```
Discussion:

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

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 $!!$

\[
\text{code}_C (\bar{X} = t) \rho = \text{put} \bar{X} \rho \\text{code}_U t \rho
\]

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 $\rightarrow$

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

\[\text{uatom a} \quad \text{R[}\quad \rightarrow \quad \text{A[} \quad a\quad ]}\]

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

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

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 $\rightarrow$
The instruction `uvar i` implements the unification with an un-initialized variable:

$$S[FP+1] = S[SP]; SP--;$$

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

- `code\_a \rho \ = \ uatom\ a`
- `code\_X \rho \ = \ uvar\ (\rho X)`
- `code\_\_ \rho \ = \ pop`
- `code\_\_X \rho \ = \ uref\ (\rho X)`

... // to be continued :-)

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

$$\delta = mgu\ (x, y)$$

unify ($S[SP]$, deref ($S[FP+1]$));

$SP--;$

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

The instruction `pop` implements the unification with an anonymous variable. It always succeeds :-(
The unification code performs a pre-order traversal over \( t \).

In case, execution hits at an unbound variable, we switch from checking to building.

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

The instruction `bind` terminates the building block. It binds the (unbound) variable to the constructed term:

\[
\text{bind} \quad \text{// creation of bindings}
\]

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

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

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

\[ \text{check}(Y_1, \ldots, Y_d) \rho = \begin{cases} \text{check}(\rho Y_1) \\ \text{check}(\rho Y_2) \\ \vdots \\ \text{check}(\rho Y_d) \end{cases} \]

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

\[ \text{code}_f(t_1, \ldots, t_n) \rho = \begin{cases} \text{struct } f/n A \\ \text{son } 1 \\ \text{code}_f t_1 \rho \\ \vdots \\ \text{son } n \\ \text{code}_f t_n \rho \\ \text{up } B \\ \text{check } \text{itsars}(f(t_1, \ldots, t_n)) \rho & \text{occurs-check} \\ \text{code}_f f(t_1, \ldots, t_n) \rho & \text{building} \\ \text{bind} & \text{creation of bindings} \end{cases} \]

The Pre-Order Traversal:

- First, we \textbf{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 \textbf{recursively descend} to the children.
- Then we \textbf{pop} the stack and proceed behind the unification code.
Once again the unification code for constructed terms:

\[
\text{code}_2(t_1, \ldots, t_n) \; \rho \; = \; \begin{cases} 
\text{ustruct } f/n \; \rho \; & \text{// test} \\
\text{son } 1 \; & \text{// recursive descent} \\
\text{code}_2 \; t_1 \; \rho \; & \\
\ldots \; & \\
\text{son } n \; & \text{// recursive descent} \\
\text{code}_2 \; t_n \; \rho \; & \\
\text{up } B \; & \text{// ascent to father} \\
\end{cases} \\
A : \; \text{check } \text{vars}(f(t_1, \ldots, t_n)) \; \rho \\
\text{code}_2 \; f(t_1, \ldots, t_n) \; \rho \; & \\
\text{bind} \; & \\
B : \; \ldots \\
\]

The instruction \( \text{ustruct } i \) implements the test of the root node of a structure:

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

... the argument reference is not yet popped \( \_ \_ \_ \_ \)

The instruction \( \text{son } i \) 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 \( \text{up } B \) which finally pops the reference to the structure:

\[
\begin{array}{c}
\text{SP}--; \; \text{PC } = \; B; \\
\end{array}
\]

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:

```
usrtct f/3 A1    up B2    B2: son 2    putvar 2
son 1
usrtct g/2 A2    check 1   son 3    putatom a
son 1
uref 1           putvar 2   up B1    putstruct g/2
son 2           putstruct g/2 A1: check 1    bind
uvar 2           bind       putref 1   B1: ...
```

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