24.5 Closures of Tuples and Lists

The general schema for $\text{code}_C$ can be optimized for tuples and lists:

\[
\begin{align*}
\text{code}_C (e_0, \ldots, e_{k-1}) \rho \text{ sd} &= \text{code}_V (e_0, \ldots, e_{k-1}) \rho \text{ sd} = \text{code}_C e_0 \rho \text{ sd} \\
&\phantom{=} \text{code}_C e_1 \rho (\text{sd + 1}) \\
&\cdots \\
&\text{code}_C e_{k-1} \rho (\text{sd + k - 1}) \\
&\text{mkvec k}
\end{align*}
\]

\[
\begin{align*}
\text{code}_C [] \rho \text{ sd} &= \text{code}_V [] \rho \text{ sd} = \text{nil} \\
\text{code}_C (e_1 : e_2) \rho \text{ sd} &= \text{code}_V (e_1 : e_2) \rho \text{ sd} = \text{code}_C e_1 \rho \text{ sd} \\
&\text{code}_C e_2 \rho (\text{sd + 1}) \\
&\text{cons}
\end{align*}
\]
25  Last Calls

A function application is called last call in an expression $e$ if this application could deliver the value for $e$.

A last call usually is the outermost application of a defining expression.

A function definition is called tail recursive if all recursive calls are last calls.

Examples:

\[
\begin{align*}
   r \; t \; (h : y) & \text{ is a last call in } \text{case } x \text{ of } [] \rightarrow y; \; h : t \rightarrow r \; t \; (h : y) \\
   f \; (x - 1) & \text{ is not a last call in } \text{if } x \leq 1 \text{ then } 1 \text{ else } x \ast f \; (x - 1)
\end{align*}
\]

Observation: Last calls in a function body need no new stack frame!

⇒⇒⇒

Automatic transformation of tail recursion into loops!!!
The code for a last call \( l \equiv (e' \ e_0 \ldots e_m) \) inside a function \( f \) with \( k \) arguments must

1. allocate the arguments \( e_i \) and evaluate \( e' \) to a function (note: all this inside \( f \)'s frame!);
2. deallocate the local variables and the \( k \) consumed arguments of \( f \);
3. execute an apply.

\[
\begin{align*}
\text{code}_V l \ \rho \ \text{sd} & = \text{code}_C e_{m-1} \ \rho \ \text{sd} \\
& \quad \text{code}_C e_{m-2} \ \rho \ (\text{sd} + 1) \\
& \ldots \\
& \quad \text{code}_C e_0 \ \rho \ (\text{sd} + m - 1) \\
\text{code}_V e' \ \rho \ (\text{sd} + m) & \quad \text{// Evaluation of the function} \\
\text{move} \ r \ (m + 1) & \quad \text{// Deallocation of } r \text{ cells} \\
\text{apply} & \\
\end{align*}
\]

where \( r = \text{sd} + k \) is the number of stack cells to deallocate.
Example:

The body of the function

\[ r = \text{fn } x, y \Rightarrow \text{case } x \text{ of } [] \rightarrow y; h : t \rightarrow r \, t \, (h : y) \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>targ 2</td>
<td>1</td>
<td>jump B</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>pushloc 0</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>eval</td>
<td>2</td>
<td>A: pushloc 1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>tlist A</td>
<td>3</td>
<td>pushloc 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>pushloc 1</td>
<td>4</td>
<td>cons</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>eval</td>
<td>3</td>
<td>pushloc 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since the old stack frame is kept, return 2 will only be reached by the direct jump at the end of the []-alternative.
SP = SP – k – r;
for (i=1; i ≤ k; i++)
    S[SP+i] = S[SP+i+r];
SP = SP + k;
The Translation of Logic Languages
Here, we just consider the core language Proll ("Prolog-light" :-). In particular, we omit:

- arithmetic;
- the cut operator;
- self-modification of programs through `assert` and `retract`. 
Example:

\[
\begin{align*}
\text{bigger}(X, Y) & \quad \leftarrow \quad X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \quad \leftarrow \quad X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \quad \leftarrow \quad X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) & \quad \leftarrow \quad X = \text{donkey}, Y = \text{monkey} \\
\text{is} _ \text{bigger}(X, Y) & \quad \leftarrow \quad \text{bigger}(X, Y) \\
\text{is} _ \text{bigger}(X, Y) & \quad \leftarrow \quad \text{bigger}(X, Z), \text{is} _ \text{bigger}(Z, Y) \\
? \quad \text{is} _ \text{bigger}(\text{elephant}, \text{dog})
\end{align*}
\]
A More Realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \quad \leftarrow \quad X = [], \ Y = Z \\
\text{app}(X, Y, Z) & \quad \leftarrow \quad X = [H|X'], \ Z = [H|Z'], \ \text{app}(X', Y, Z') \\
? \quad \text{app}(X, [Y, c], [a, b, Z])
\end{align*}
\]
A More Realistic Example:

\[
\text{app}(X, Y, Z) \leftarrow X = [], \ Y = Z
\]

\[
\text{app}(X, Y, Z) \leftarrow X = [H|X'], \ Z = [H|Z'], \ \text{app}(X', Y, Z')
\]

? \text{app}(X, [Y, c], [a, b, Z])

Remark:

[ ] \quad \equiv \quad \text{the atom empty list}

[H|Z] \quad \equiv \quad \text{binary constructor application}

[a, b, Z] \quad \equiv \quad \text{shortcut for: } \ [a|[b|[Z|[ ]]]]

A program $p$ is constructed as follows:

$$
t := a | X | _ | f(t_1, \ldots, t_n)
g := p(t_1, \ldots, t_k) | X = t
c := p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r
p ::= c_1 \ldots c_m?g
$$

- A **term** $t$ either is an atom, a variable, an anonymous variable or a constructor application.
- A **goal** $g$ either is a literal, i.e., a predicate call, or a unification.
- A **clause** $c$ consists of a **head** $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a **body**, i.e., a sequence of goals.
- A **program** consists of a sequence of clauses together with a single goal as query.
Procedural View of Proll programs:

- goal: procedure call
- predicate: procedure
- clause: definition
- term: value
- unification: basic computation step
- binding of variables: side effect

Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only  :-)  
- ... may fail. Then the next definition is tried  :-))  
  ⇒ backtracking
27 Architecture of the WiM:

The Code Store:

\[ C \]

\[ \text{Code store – contains WiM program; every cell contains one instruction;} \]

\[ \text{Program Counter – points to the next instruction to executed;} \]
The Runtime Stack:

- **S** = Runtime Stack – every cell may contain a value or an address;
- **SP** = Stack Pointer – points to the topmost occupied cell;
- **FP** = Frame Pointer – points to the current stack frame.

Frames are created for predicate calls, contain cells for each variable of the current clause.
The Heap:

- $H$ = Heap for dynamically constructed terms;
- $HP$ = Heap-Pointer – points to the first free cell;

- The heap is maintained like a stack as well. :-)
- A new-instruction allocates an object in $H$.
- Objects are tagged with their types (as in the MaMa) ...
A a atom 1 cell
R variable 1 cell
R unbound variable 1 cell
S f/n structure (n+1) cells
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 FP) 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., $S[\text{FP} + \rho X]$ contains already a reference, $Y$ and $Z$ are not yet initialized.
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:

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

\[
\text{code}_A f(t_1, \ldots, t_n) \, \rho = \text{code}_A t_1 \, \rho
\]

\[
\text{putstruct} \, f/n
\]

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

\[
\text{putref} \, 1 \, \text{putatom} \, a \, \text{putvar} \, 2 \, \text{putvar} \, 3 \, \text{putstruct} \, g/2 \, \text{putstruct} \, f/3
\]
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:

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

\[
\text{code}_A f(t_1, \ldots, t_n) \rho = \text{code}_A t_1 \rho \quad \ldots \\
\text{code}_A t_n \rho
\]

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

- putref 1
- putvar 2
- putvar 3
- putstruct g/2
- putstruct f/3
The instruction `putatom a` constructs an atom in the heap:

```
SP++; S[SP] = new (A,a);
```

![Diagram showing the process of putting an atom on the heap](image)
The instruction \texttt{putvar i} introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:

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

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

```
SP = SP + 1;
S[SP] = deref S[FP + i];
```
The instruction `putref i` pushes the value of the variable onto the stack:

\[
\begin{align*}
\text{SP} &= \text{SP} + 1; \\
\text{S}[\text{SP}] &= \text{deref}\text{ S}[\text{FP} + i];
\end{align*}
\]

The auxiliary function `deref` contracts chains of references:

```c
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{align*}
v &= \text{new} \ (S, f, n); \\
SP &= SP - n + 1; \\
\text{for} \ (i=1; \ i \leq n; \ i++) \\
\ H[v+i] &= S[SP+i-1]; \\
S[SP] &= v;
\end{align*}
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
  \[\Rightarrow\] 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. :-(

---

232