24 Structured Data

In the following, we extend our functional programming language by some datatypes.

24.1 Tuples

**Constructors:** \((\ldots, x, \ldots)\), \(k\)-ary with \(k \geq 0\);

**Destructors:** \(#j\) for \(j \in \mathbb{N}_0\) (Projections)

We extend the syntax of expressions correspondingly:

\[
e ::= \ldots \mid (e_0, \ldots, e_{k-1}) \mid #j \ e
\]

<table>
<thead>
<tr>
<th>(e)</th>
<th>(\alpha)</th>
<th>(\eta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(e_0)</td>
<td>(e_{k-1})</td>
<td>(e_k)</td>
</tr>
</tbody>
</table>
In order to construct a tuple, we collect sequence of references on the stack. Then we construct a vector of these references in the heap using `mkvec`.

For returning components we use an indexed access into the tuple.

\[
\text{code}_\rho (e_0, \ldots, e_{k-1}) \rho \, \text{sd} = \begin{array}{ll}
\text{code}_\rho e_0 \, \rho \, \text{sd} & \\
\text{code}_\rho e_1 \, \rho \, (\text{sd} + 1) & \\
\vdots & \\
\text{code}_\rho e_{k-2} \, \rho \, (\text{sd} + k - 2) & \\
\text{mkvec} \, k
\end{array}
\]

\[
\text{code}(\#j \, e) \rho \, \text{sd} = \begin{array}{ll}
\text{code}_\rho e \, \rho \, \text{sd} & \\
\text{get}_j & \\
\text{eval}
\end{array}
\]

In the case of CBV, we directly compute the values of the \( e_i \).

**Inversion:** Accessing all components of a tuple simultaneously:

\[
e \equiv \text{let} \, (y_0, \ldots, y_{k-1}) = e_1 \, \text{in} \, e_0
\]

This is translated as follows:

\[
\text{code}_\rho e \, \rho \, \text{sd} = \begin{array}{ll}
\text{code}_\rho e_1 \, \rho \, \text{sd} & \\
\text{getvec} \, k & \\
\text{code}_\rho e_2 \, \rho \, (\text{sd} + k) & \\
\text{slide} \, k
\end{array}
\]

where \( \rho' = \rho \oplus \{ y_i \mapsto (L, \text{sd} + i + 1), \, i = 0, \ldots, k - 1 \} \).

The instruction `getvec k` pushes the components of a vector of length \( k \) onto the stack.
### 24.2 Lists

Lists are constructed by the constructors:

- `[]` "Nil", the empty list;
- `::` "Cons", right-associative, takes an element and a list.

Access to list components is possible by match-expressions...

**Example:** The append function `app`:

```
app = fun | y -> match _ with
       | [] -> y
       | h :: t -> (app t y)
```

---

**Inversion:** Accessing all components of a tuple simultaneously:

\[
e \equiv \text{let } (y_0, \ldots, y_{l-1}) = c_1 \text{ in } c_0
\]

This is translated as follows:

\[
\text{code } e \rho \text{ sd} = \text{code } c_1 \rho \text{ sd}
\text{getvec } k
\text{code } c_0 \rho' \text{ (sd + k)}
\text{slide } k
\]

where \( \rho' = \rho \odot \{ y_i \mapsto (L, sd + i + 1) | i = 0, \ldots, k - 1 \} \).

The instruction `getvec k` pushes the components of a vector of length `k` onto the stack:

---

Accordingly, we extend the syntax of expressions:

\[
e ::= \ldots | [] | (e_1 :: e_2) | (\text{match } e_0 \text{ with } \_ \rightarrow e_1 | h :: t \rightarrow e_2)
\]

Additionally, we need new heap objects:

```
L | Nil                   empty list
L | Cons                  non-empty list
```

s[0]  s[1]
24.2 Lists

Lists are constructed by the constructors:

- `[]` "Nil", the empty list;
- `::` "Cons", right-associative, takes an element and a list.

Access to list components is possible by match-expressions ...

Example: The append function \( \text{app} \):

\[
\text{app} = \text{fun } y \rightarrow \text{match } l \text{ with }\\
\quad l \rightarrow y\\
\quad h :: t \rightarrow h :: (\text{app } t \ y)
\]

```
\[
\text{app} = \text{fun } y \rightarrow \text{match } l \text{ with }\\
\quad l \rightarrow y\\
\quad h :: t \rightarrow h :: (\text{app } t \ y)
\]
```

201

Accordingly, we extend the syntax of expressions:

\[
e ::= \ldots \mid [] \mid (e_1 :: e_2) \\
\quad \mid (\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2)
\]

Additionally, we need new heap objects:

```
\[
\text{empty list}
\]
```

```
\[
\text{non-empty list}
\]
```

202

24.3 Building Lists

The new instructions \( \text{nil} \) and \( \text{cons} \) are introduced for building list nodes.

We translate for CBN:

\[
\text{code}_Y \enspace [] \rho \sigma_d = \text{nil} \\
\text{code}_Y \enspace (e_1 :: e_2) \rho \sigma_d = \text{code}_Y \enspace e_1 \rho \sigma_d :: \text{code}_Y \enspace e_2 \rho (\sigma_d + 1)
\]

Note:
- With CBN: Closures are constructed for the arguments of "::";
- With CBV: Arguments of "::" are evaluated \( \Rightarrow \)
24.4 Pattern Matching

Consider the expression \( e \equiv \text{match } a \text{ with } [] \to e_1, [h : t] \to e_2 \).

Evaluation of \( e \) requires:
- evaluation of \( e_1 \);
- check, whether resulting value \( v \) is an L-object;
- if \( v \) is the empty list, evaluation of \( e_1 \), ...
- otherwise storing the two references of \( v \) on the stack and evaluation of \( e_2 \).
This corresponds to binding \( h \) and \( t \) to the two components of \( v \).

In consequence, we obtain (for CBN as for CBV):

\[
\begin{align*}
\text{code}_v \ e & \ p \ \text{sd} = \text{code}_v \ e_0 \ p \ \text{sd} \\
\text{tlist} & A \\
\text{code}_v \ e_1 & \ p \ \text{sd} \\
\text{jump} & B \\
A : \ & \text{code}_v \ e_2 \ p' \ (\text{sd} + 2) \\
\text{slide} & 2 \\
B : & \ldots
\end{align*}
\]

where \( p' = p \oplus \{ h \mapsto (L, sd + 1), t \mapsto (L, sd + 2) \} \).

The new instruction \( \text{tlist} \ A \) does the necessary checks and (in the case of \( \text{Cons} \)) allocates two new local variables:
Example: The (disentangled) body of the function \texttt{app} with \texttt{app} \to \langle \texttt{G}, 0 \rangle:

0 \quad \text{tang 2} \quad 3 \quad \text{pushglob 0} \quad 0 \quad \text{C: mark D} \\
0 \quad \text{pushloc 0} \quad 4 \quad \text{pushloc 2} \quad 3 \quad \text{pushglob 2} \\
1 \quad \text{eval} \quad 5 \quad \text{pushloc 6} \quad 4 \quad \text{pushglob 1} \\
1 \quad \text{tlist A} \quad 6 \quad \text{mkvec 3} \quad 5 \quad \text{pushglob 0} \\
0 \quad \text{pushloc 1} \quad 4 \quad \text{mkvec C} \quad 6 \quad \text{eval} \\
1 \quad \text{eval} \quad 4 \quad \text{cons} \quad 5 \quad \text{apply} \\
1 \quad \text{jump B} \quad 3 \quad \text{slide 2} \quad 1 \quad \text{D: update} \\
2 \quad \text{A: pushloc 1} \quad 1 \quad \text{B: return 2}

Note:

Datatypes with more than two constructors need a generalization of the \texttt{tlist} instruction, corresponding to a \texttt{switch}-instruction  \to}

24.5 Closures of Tuples and Lists

The general schema for \texttt{codeC} can be optimized for tuples and lists:

\[
\begin{align*}
\texttt{codeC} \langle \varepsilon_0, \ldots, \varepsilon_{k-1} \rangle \rho \texttt{sd} & = \texttt{codeV} \langle \varepsilon_0, \ldots, \varepsilon_{k-1} \rangle \rho \texttt{sd} \\
& = \texttt{codeC} \varepsilon_0 \rho \texttt{sd} \\
& = \texttt{codeC} \varepsilon_1 \rho \texttt{sd + 1} \\
& \ldots \\
& = \texttt{codeC} \varepsilon_{k-1} \rho \texttt{sd + k - 1} \\
& \texttt{mkvec k} \\
\texttt{codeC} \varepsilon \rho \texttt{sd} & = \texttt{codeV} \varepsilon \rho \texttt{sd} = \varepsilon_0 \\
\texttt{codeC} (\varepsilon_1 \sqcup \varepsilon_2) \rho \texttt{sd} & = \texttt{codeV} (\varepsilon_1 \sqcup \varepsilon_2) \rho \texttt{sd} \\
& = \texttt{codeC} \varepsilon_1 \rho \texttt{sd} \\
& = \texttt{codeC} \varepsilon_2 \rho \texttt{sd + 1} \\
& \texttt{cons}
\end{align*}
\]
24.5 Closures of Tuples and Lists

The general schema for codec can be optimized for tuples and lists:

\[
\begin{align*}
\text{codec}_c \ (e_0, \ldots, e_{k-1}) \ \rho \ \text{sd} & = \ \text{codec}_v \ (e_0, \ldots, e_{k-1}) \ \rho \ \text{sd} \\
\text{codec}_c \ e_0 \ \rho \ \text{sd} & = \ \text{codec}_c e_1 \ \rho \ (\text{sd} + 1) \\
\vdots & \\
\text{codec}_c \ e_{k-1} \ \rho \ (\text{sd} + k - 1) & = \ \text{mkvec} \ k \\
\text{codec}_c \ [] \ \rho \ \text{sd} & = \ \text{codec}_v [] \ \rho \ \text{sd} \\
\text{codec}_c \ (e_1 :: e_2) \ \rho \ \text{sd} & = \ \text{codec}_v \ (e_1 :: e_2) \ \rho \ \text{sd} \\
\text{codec}_c \ e_1 \ \rho \ \text{sd} & = \ \text{codec}_c e_2 \ \rho \ (\text{sd} + 1) \\
\text{cons} & \\
\end{align*}
\]

25 Last Calls

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

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

Examples:

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

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 \(l \equiv (\rho' e_0 \ldots e_k)\) inside a function \(f\) with \(k\) arguments must

1. allocate the arguments \(e_i\) and evaluate \(\rho'\) 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.

\[
\begin{align*}
\text{codec}_v \ l \ \rho \ \text{sd} & = \ \text{codec}_c \ e_{m-1} \ \rho \ \text{sd} \\
\text{codec}_c \ e_{m-2} \ \rho \ (\text{sd} + 1) & \\
\vdots & \\
\text{codec}_c \ e_0 \ \rho \ (\text{sd} + m - 1) & \text{ Evaluation of the function} \\
\text{codec}_v \ \rho' \ (\text{sd} + m) & \text{ Deallocation of } r \text{ cells} \\
\text{move} \ r \ (m + 1) & \text{ Apply} \\
\end{align*}
\]

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