CMSC631 Project Exercises

Austin Bourgerie

7 December 2017

1 Sum Types

11.1 Enumerations may be defined as sums over a finite index set I, where all I-classified values have type unit. In terms of finite sums:

$$\begin{aligned} \texttt{enum}[I] &\triangleq [\texttt{unit}]_{\_\in I} \\ i_{i\in I} &\triangleq i \cdot \langle\rangle \\ \texttt{switch}\ e\ \{i \hookrightarrow e_i\}_{i\in I} &\triangleq \texttt{case}\ e\ \{i \cdot x_i \hookrightarrow e_i\}_{i\in I, x_i \notin e_i} \end{aligned}$$

This representation reflects that enums are a special case of finite sums wherein all introduction and elimination forms imply that contents are simply $\langle\rangle$.

11.2

$$\begin{aligned} \texttt{null} &\triangleq \langle \texttt{false}, \texttt{null} \rangle \\ \texttt{just}(e) &\triangleq \langle \texttt{true}, e \rangle \\ \texttt{which}\ e\ \{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x) \hookrightarrow e_2\} &\triangleq \texttt{if}\ e \cdot \texttt{l}\ \texttt{then}\ [e \cdot \texttt{r}/x]e_1\ \texttt{else}\ e_2 \end{aligned}$$

Even if we allow the use of a `null` value for every type, the `just` case in the elimination form would still theoretically allow `null`s to propagate into the branch by first using the introduction form `just(null)`. This kind of clumsy handling is what earned this problem the name "the billion dollar mistake".

11.3 If limited to a traditional product type-based representation of a database, a similar system of flags to the Hoare `null` may be be used to encode heterogeneous types. For example, the concept of 5-digit vs 9-digit postal codes that may be missing can be encoded by extending the schema as $\langle ..., \times\ postal_{flag} \times postal_5 \times postal_9\rangle$, where $postal_{flag}$ encodes which code to look at, or if the code is missing. For example, one possible encoding would be:

$$\begin{aligned} 0 &\hookrightarrow \texttt{null} \\ 1 &\hookrightarrow postal_5 \\ 2 &\hookrightarrow postal_9 \end{aligned}$$

However, having sum types would enable a more self-evident type-safe encoding. If instead the schema could be extended to $\langle ... \times (\texttt{unit} + postal_5 + postal_9)\rangle$, a single column in the schema could safely maintain a consistent relationship between the data and an ability to identify it.

11.4

$$NOR = \texttt{if}\ x\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{if}\ y\ \texttt{false}\ \texttt{else}\ \texttt{true}$$

$$NAND = \texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ y\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{true})\ \texttt{else}\ \texttt{true}$$

$$HALF\text{-}ADDER = \lambda x.\lambda y.\langle XOR\ x\ y, AND\ x\ y\rangle$$

$$\begin{aligned}FULL\text{-}ADDER = \lambda x.\lambda y.\lambda c.\langle & XOR\ c\ (XOR\ x\ y),\\ & OR\ (AND\ x\ y)\ (AND\ c\ (XOR\ x\ y))\rangle\end{aligned}$$

$$\begin{aligned}NYBBLE\text{-}ADDER = & \\ \lambda x.\lambda y.&let\ \langle z_0, c_1\rangle = HALF\text{-}ADDER\ x\cdot 0\ y\cdot 0\ in\\ &let\ \langle z_1, c_2\rangle = HALF\text{-}ADDER\ x\cdot 1\ y\cdot 1\ in\\ &let\ \langle z_2, c_3\rangle = HALF\text{-}ADDER\ x\cdot 2\ y\cdot 2\ in\\ &let\ \langle z_3, c_4\rangle = HALF\text{-}ADDER\ x\cdot 3\ y\cdot 3\ in\\ &\langle\langle z_0, z_1, z_2, z_3\rangle, c_4\rangle\end{aligned}$$

11.5 Omitted

2 Generic Programming

14.1 For all type operators $t.\tau$ poly and $t'.\tau'$ poly, prove $t.[\tau/t']$ poly.

Proof by induction on τ'

– Base Cases:

$$
\begin{aligned}
t.[\tau/t']t' \text{ poly} &= t.\tau \text{ poly} && \text{(By assumption)} \\
t.[\tau/t']\texttt{unit} \text{ poly} &= t.\texttt{unit} \text{ poly} && \text{(By definition)} \\
t.[\tau/t']\texttt{void} \text{ poly} &= t.\texttt{void} \text{ poly} && \text{(By definition)}
\end{aligned}
$$

– Inductive Step:

$$
\begin{aligned}
t.[\tau/t'](\tau_1 \times \tau_2) \text{ poly} &= t.[\tau/t']\tau_1 \times [\tau/t']\tau_2 \text{ poly.} && \text{(By IH)} \\
t.[\tau/t'](\tau_1 + \tau_2) \text{ poly} &= t.[\tau/t']\tau_1 + [\tau/t']\tau_2 \text{ poly.} && \text{(By IH)}
\end{aligned}
$$

14.2 For all values e of type τ, prove that $\texttt{map}\{\_.\tau\}(x.e')(e) \mapsto e$

Proof by induction on τ.

– Base Cases:

$$
\begin{aligned}
\texttt{map}\{\_.\texttt{unit}\}(x.e')(e) &\longmapsto e && \text{(By definition)} \\
\texttt{map}\{\_.\texttt{void}\}(x.e')(e) &\longmapsto \texttt{abort}(e) && \text{(By definition)}
\end{aligned}
$$

The case for $\texttt{map}\{t.t\}$... need not be considered, as we are reasoning about cases where no instances of the type variable occur in the type operator.

– Inductive Step:

* Case Product

$$
\begin{aligned}
&\texttt{map}\{\_.\tau_1 \times \tau_2\}(x.e')(e) \\
\longmapsto &\langle \texttt{map}\{\_.\tau_1\}(x.e')(e \cdot l), \texttt{map}\{\_.\tau_2\}(x.e')(e \cdot r)\rangle \\
\longmapsto &\langle e \cdot l, e \cdot r\rangle = e && \text{(By IH)}
\end{aligned}
$$

* Case Sum: $\texttt{map}\{\_.\tau_1 + \tau_2\}(x.e')(e)$
  Proceed by case analysis on e.
  · Case $l \cdot e_l$

$$... \longmapsto l \cdot \texttt{map}\{\_.\tau_1\}(x.e')(e_l) \longmapsto l \cdot e_l = e$$

  · Case $r \cdot e_r$

$$... \longmapsto r \cdot \texttt{map}\{\_.\tau_1\}(x.e')(e_r) \longmapsto r \cdot e_r = e$$

14.3 1. $t.\langle i_1 \hookrightarrow \tau_1, ..., \text{first} \hookrightarrow t, ..., \text{last} \hookrightarrow t, i_n \hookrightarrow \tau_n \rangle$

2. c (the assumed capitalization function)

3. $\mathtt{map}\{t.\langle i_1 \hookrightarrow \tau_1, ..., \text{first} \hookrightarrow t, ..., \text{last} \hookrightarrow t, i_n \hookrightarrow \tau_n \rangle\}(c)(\text{row})$

14.4 The following is a definition of non-negative type operators.

$$\text{trans}\,\frac{t.\tau \text{ pos}}{t.\tau \text{ non-neg}}$$

$$\text{neg}\,\frac{t.\tau_1 \text{ non-neg}}{t.\tau_1 \to \tau_2 \text{ neg}} \qquad \text{non-neg}\,\frac{t.\tau_1 \text{ neg} \quad t.\tau_2 \text{ non-neg}}{t.\tau_1 \to \tau_2 \text{ non-neg}}$$

The following is a proof of $t.(t \to \mathtt{bool}) \to \mathtt{bool}$ non-neg.

$$\text{non-neg}\,\frac{\text{neg}\,\dfrac{\text{trans}\,\dfrac{\text{14.4a}\,\dfrac{}{t.t \text{ pos}}}{t.t \text{ non-neg}}}{t.t \to \mathtt{bool} \text{ neg}} \qquad \text{trans}\,\dfrac{\dfrac{}{\mathtt{bool} \text{ pos}}}{\mathtt{bool} \text{ non-neg}}}{t.(t \to \mathtt{bool})) \to \mathtt{bool} \text{ non-neg}}$$

14.5 The semantics for the $\mathtt{map}^{--}$ and $\mathtt{map}^{-}$ are identical to those for the provided $\mathtt{map}^{+}$, except in the arrow case. The following are the semantics for negative and non-negative type operators in that case.

$$\frac{}{\begin{array}{c}\mathtt{map}^{--}\{t.\tau_1 \to \tau_2\}(x.e')(e) \\ \longmapsto \\ \lambda(x_1 : [\rho'/t]\tau_1)\mathtt{let}\ x_1'\ \mathtt{be}\ \mathtt{map}^{-}\{t.\tau_1\}(x.e')(x_1)\ \mathtt{in} \\ \mathtt{map}^{--}\{t.\tau_2\}(x.e')(e(x_1'))\end{array}}$$

$$\frac{}{\begin{array}{c}\mathtt{map}^{-}\{t.\tau_1 \to \tau_2\}(x.e')(e) \\ \longmapsto \\ \lambda(x_1 : [\rho/t]\tau_1)\mathtt{let}\ x_1'\ \mathtt{be}\ \mathtt{map}^{--}\{t.\tau_1\}(x.e')(x_1)\ \mathtt{in} \\ \mathtt{map}^{--}\{t.\tau_2\}(x.e')(e(x_1'))\end{array}}$$

The idea in applying either case is to wrap the transformation in a function expecting the new type of t (ρ' for non-negative, simply ρ for negative, in accordance with the statics). When the function is called with a concrete input, that input is recursively transformed by the functor.

3 Inductive and Coinductive Types

15.1 The following is a definition for a function which sends a natural number to its identity as a conatural number.

$$\begin{aligned}&\lambda n.\texttt{let } \mathit{zero} \texttt{ be } \lambda n.\texttt{rec}(x.\texttt{case } x\{\texttt{l}\cdot \_ \hookrightarrow \texttt{true} \mid \texttt{r}\cdot \_ \hookrightarrow \texttt{false}\}; n) \texttt{ in}\\ &\quad \texttt{let } \mathit{pred} \texttt{ be } \lambda n.\texttt{rec}(x.\texttt{case } x\{\\ &\qquad \texttt{l}\cdot\langle\rangle \hookrightarrow \langle \texttt{false}, \texttt{fold}(\texttt{l}\cdot\langle\rangle)\rangle\\ &\qquad \mid \texttt{r}\cdot v \hookrightarrow \langle \texttt{true}, \texttt{if } v\cdot\texttt{l} \texttt{ then } \texttt{fold}(\texttt{r}\cdot(v\cdot\texttt{r})) \texttt{ else } v\cdot\texttt{r}\}; n) \texttt{ in}\\ &\quad \texttt{gen}(x.\texttt{if } \mathit{zero}\ x \texttt{ then } \texttt{l}\cdot\langle\rangle \texttt{ else } \texttt{r}\cdot(\mathit{pred}\ x); n)\end{aligned}$$

15.2 The following is a derivation of the primitive iterator for natural numbers in terms of the inductive nat type (not the full inductive type).

$$\texttt{rec}_{\texttt{nat}}(x.\texttt{case } x\{\texttt{l}.\langle\rangle \hookrightarrow e_0 \mid \texttt{r}.v \hookrightarrow [v/x]e_1\}, e)$$

15.3 The following is a derivation of the stream generator in terms of the coinductive stream type (not the full coinductive type).

$$\texttt{strgen } x \texttt{ is } e \texttt{ in } < \texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2 > \triangleq \texttt{gen}_{\texttt{stream}}(x.\langle e_1, e_2\rangle, e)$$

15.4 Just as every stream can be turned into a sequence using the provided code, so can every sequence be turned into a stream whose nth element is the nth element of the given sequence via the following function.

$$\lambda \mathit{seq}.\texttt{strgen } x \texttt{ is } 0 \texttt{ in} < \texttt{hd} \hookrightarrow \mathit{seq}(x), \texttt{tl} \hookrightarrow \texttt{fold}_{\texttt{nat}}(\texttt{r}\cdot x) >$$

15.5 The following is a definition of lists of natural numbers in terms of full-fledged inductive types.

$$\begin{aligned}\texttt{natlist} &\triangleq \texttt{ind}(t.\texttt{unit} + (\texttt{nat} \times t))\\ \texttt{nil} &\triangleq \texttt{fold}(\texttt{l}\cdot\langle\rangle)\\ \texttt{cons}(e_1, e_2) &\triangleq \texttt{fold}(\texttt{r}\cdot\langle e_1, e_2\rangle)\\ \texttt{rec } e\{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x; y) \hookrightarrow e_1\} &\triangleq \begin{array}{l}\texttt{rec}(x.\texttt{case } x\{\\ \quad \texttt{l}\cdot\langle\rangle \hookrightarrow e_0\\ \quad \mid \texttt{r}\cdot v \hookrightarrow [v\cdot\texttt{l}, v\cdot\texttt{r}/x, y]e_1\}; e)\end{array}\end{aligned}$$

15.6 The following is a definition of the dynamics for possibly empty binary trees as a full-fledged coinductive type.

$$\frac{}{\texttt{itgen } x \texttt{ is } e \texttt{ in } e' \text{ val}} \qquad \frac{e \longmapsto e'}{\texttt{view}(e) \longmapsto \texttt{view}(e')}$$

$$\frac{}{\texttt{view}(\texttt{itgen } x \texttt{ is } e \texttt{ in } e') \longmapsto \texttt{map}(y.\texttt{itgen } x \texttt{ is } y \texttt{ in } e')([e/x]e')}$$

15.7 Omitted

4 System F of Polymorphic Types

16.1 Omitted

16.2 The following is a definition of polymorphic Church booleans.

$$\begin{aligned}
\texttt{bool} &\triangleq \forall(t.t \to t \to t) \\
\texttt{true} &\triangleq \Lambda(t)\lambda(a:t)\lambda(b:t)a \\
\texttt{false} &\triangleq \Lambda(t)\lambda(a:t)\lambda(b:t)b \\
\texttt{if}\ e\ \texttt{then}\ e_0\ \texttt{else}\ e_1 &\triangleq e[\rho](e_0)(e_1)
\end{aligned}$$

16.3 The following is a definition of lists of natural numbers.

$$\begin{aligned}
\texttt{natlist} &\triangleq \forall(t.t \to (\texttt{nat} \to t \to t) \to t) \\
\texttt{null} &\triangleq \Lambda(t)\lambda(n:t)\lambda(c:\texttt{nat} \to t \to t)t \\
\texttt{cons}(e_1; e_2) &\triangleq \Lambda(t)\lambda(n:t)\lambda(c:\texttt{nat} \to t \to t)c(e_1)(e_2[t](n)(c))
\end{aligned}$$

Just as with the natural numbers, inductive processing is accomplished by applying an instance of `natlist` to a concrete result type t, as well as a value to represent the base case and a function to join the head of the list and an inductive result into a stepped result. This pattern can be seen in the semantics for the body of the `cons` case, which applies the given step function c to the head and the result of processing the tail.

16.4 The following is a definition of inductive types.

$$\mu(t.\tau) \triangleq \forall r(([r/t]\tau \to r) \to r)$$

Whereas the inductive step in the previous exercise is represented in terms of separate base and recursive cases, the generalization joins the two by using a function from the inductive type τ to the result type r, where the inductive points of τ are concretized by r. This closely mirrors the typing judgement for the inductive elimination form (the recursor) presented in chapter 15.

16.5 Define the type t `list` to be as follows.

$$t\ \texttt{list} \triangleq \forall(t.(\forall u.u \to (t \to u \to u) \to u))$$

For all t `lists` l of type τ, define the property $\mathcal{Q}_{member}$ to hold on lists which only contain elements drawn from l. It follows directly from the parametricity theorem of System F that, since $\mathcal{Q}_{member}$ holds on l (a list's elements must reflexively be members of itself), it must also hold on the list resulting from $f[\tau](l)$.