Elements of Programming Languages
Tutorial 3: Data structures and polymorphism
October 19–23, 2015

Exercises marked $\star$ are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Pairs, variants, and polymorphism in Scala**

   Scala includes built-in pair types `(T1,T2)`, with pairing written `(e1,e2)` and projection written `e._1`, `e._2`. Likewise, Scala's library includes binary sums `Either[T1,T2]` with constructors `Left(_)` and `Right(_)`. Pattern matching can be used to analyze `Either[T1,T2]`. Using these operations, write polymorphic Scala functions having the following types, polymorphic in `A,B,C`:

   (a) `(A,B) => (B,A)`

   (b) `Either[A,B] => Either[B,A]`

   (c) `((A,B) => C) => (A => (B => C))`

   (d) `(A => (B => C)) => ((A,B) => C)`

   (e) `(Either[A,B] => C) => (A => C, B => C)`

   (f) `(A => C, B => C) => (Either[A,B] => C)`

2. **Typing derivations**

   Construct typing derivations for the following expressions, or argue why they are not well-formed:

   (a) $\Lambda A.\lambda x{:}A.x + 1$

   (b) $\lambda x{:}\mathtt{int} + \mathtt{bool}.\mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{left}(y) \Rightarrow y == 0\ ;\ \mathtt{right}(z) \Rightarrow z\}$

   (c) $\lambda x{:}\mathtt{int} \times \mathtt{int}.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{left}(\mathtt{fst}\ x)\ \mathtt{else}\ \mathtt{right}(\mathtt{snd}\ x)$

   (d) ($\star$) $\Lambda A.\lambda x{:}A \times A.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{fst}\ x\ \mathtt{else}\ \mathtt{snd}\ x$

3. **Evaluation derivations**

   Construct evaluation derivations for the following expressions, or explain why they do not evaluate:

   (a) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{int}]\ 42$

   (b) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{bool}]\ \mathtt{true}$

4. **Multiple argument functions**

   So far, our function definitions take only one argument. Consider $\mathsf{L}_{\mathsf{Data}}$ with named functions extended with multi-argument function definitions and applications:

$$e ::= \cdots \mid \mathtt{let\ fun}\ f(x_1 : \tau_1, x_2 : \tau_2) = e_1\ \mathtt{in}\ e_2 \mid f(e_1, e_2)$$

(a) Write appropriate typing rules for these constructs.

(b) Show that these constructs can be defined in $\mathsf{L}_{\mathsf{Data}}$.

(c) What about functions of three or more arguments?

5. ($\star$) **Mutual recursion**

   In Lecture 5, we considered a simple form of recursion that just defines one recursive function with one argument. Part 4 of this tutorial showed how to accommodate multiple arguments. But what about mutual recursion?

   A simple example is

$$\begin{array}{l}\texttt{let rec}\ even(x{:}\texttt{int}) : \texttt{bool} = \texttt{if}\ x == 0\ \texttt{then true else}\ odd(x-1) \\ \texttt{and}\ odd(x{:}\texttt{int}) : \texttt{bool} = \texttt{if}\ x == 0\ \texttt{then false else}\ even(x-1) \\ \texttt{in}\ e\end{array}$$

   Show that we can use pairing and `rec` to define these mutually recursive functions, by filling in the following template with an expression having type $\texttt{unit} \to ((\texttt{int} \to \texttt{bool}) \times (\texttt{int} \to \texttt{bool}))$ with the desired behavior:

$$\begin{array}{l}\texttt{let}\ p = \cdots\ \texttt{in} \\ \texttt{let}\ even = \texttt{fst}\ p()\ \texttt{in} \\ \texttt{let}\ odd = \texttt{snd}\ p()\ \texttt{in} \\ e\end{array}$$