Elements of Programming Languages
Tutorial 3: Recursion and data structures
Week 5 (October 16–20, 2017)

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 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) $\lambda x{:}\mathtt{int} + \mathtt{bool}.\mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{left}(y) \Rightarrow y == 0\ ;\ \mathtt{right}(z) \Rightarrow z\}$
   (b) ($\star$) $\lambda x{:}\mathtt{int} \times \mathtt{int}.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{left}(\mathtt{fst}\ x)\ \mathtt{else}\ \mathtt{right}(\mathtt{snd}\ x)$

3. **Lists**

   We could add built-in lists to $\mathsf{L}_{\mathsf{Data}}$ as follows:

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \mathtt{nil} \mid e_1 :: e_2 \mid \mathtt{case}_{\mathtt{list}}\ e\ \mathtt{of}\ \{\mathtt{nil} \Rightarrow e_1\ ;\ x :: y \Rightarrow e_2\} \\
v & ::= & \cdots \mid \mathtt{nil} \mid v_1 :: v_2 \\
\tau & ::= & \cdots \mid \mathtt{list}[\tau]
\end{array}
$$

   Define $\mathsf{L}_{\mathsf{List}}$ to be $\mathsf{L}_{\mathsf{Data}}$ extended with the above constructs.

   The typing rule for $\mathtt{case}_{\mathtt{list}}$ is:

$$
\frac{\Gamma \vdash e : \mathtt{list}[\tau] \quad \Gamma \vdash e_1 : \tau' \quad \Gamma, x{:}\tau, y{:}\mathtt{list}[\tau] \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{case}_{\mathtt{list}}\ e\ \mathtt{of}\ \{\mathtt{nil} \Rightarrow e_1\ ;\ x :: y \Rightarrow e_2\} : \tau'}
$$

   The basic idea here is: Given a list e, a $\mathtt{case}_{\mathtt{list}}$ expression does a case analysis. If e evaluates to `nil`, then we evaluate e_1. Otherwise, e must evaluate to a non-empty list of the form $v :: v'$, and we bind x to the head element v and y to the tail v', and evaluate e_2.

(a) Write appropriate typing rules for `nil` and ::.

(b) ($\star$) Write appropriate evaluation rules for the above constructs.

4. ($\star$) **Multiple argument functions and mutual recursion**

(a) 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:

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

i. Write appropriate typing rules for these constructs.

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

iii. What about functions of three or more arguments?

(b) 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:

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