Elements of Programming Languages
Tutorial 3: Recursion and data structures
Week 5 (October 14–18, 2024)

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

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

   Scala provides a form of abstraction over types called *polymorphism* (aka *generics* in Java). We will discuss polymorphism in more detail in a later lecture, but for the purpose of this tutorial we will just explain that a Scala function definitoin can take *type parameters* such as `A,B,C` which give names to types that can be used in the function definition, and which can be instantiated to different actual types whenthe function is called. So for example in the following function

   ```
   def f[A,B,C](x: T1, ..., xn: Tn): T = {...}
   ```

   the parameters `A,B,C` can be used in the type annotations `T1,...,Tn,T` as well as possibly inside the function definition.

   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:

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

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

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

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

   The basic idea here is: Given a list e, a $\texttt{case}_{\texttt{list}}$ 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:

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

   Why do we need to define p as a function taking a unit argument, instead of as a pair?