Elements of Programming Languages
Tutorial 3: Recursion and data structures
Solution notes

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)`

   ```
   def swap[A,B](p: (A,B)) = (p._2,p._1)
   ```

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

   ```
   def flip(x: Either[A,B]) = x match { case Left(y) => Right(y)
     case Right(z) => Left(z) }
   ```

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

   ```
   def curry[A,B,C](f: (A,B) => C) = {a: A => {b: B => f(a,b)}}
   ```

   Equivalent alternative form:

   ```
   def curry[A,B,C](f: (A,B) => C)(a: A)(b: B) = f(a,b)
   ```

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

   ```
   def uncurry[A,B,C](f: A => (B => C)) = { (p: (A,B)) => f(p._1)(p._2) }
   ```

   Equivalent alternative form:

   ```
   def uncurry[A,B,C](f: A => (B => C))(p: (A,B)) = f(p._1,p._2)
   ```

   Notice that $\tau_1 \to \tau_2 \to \tau_3$ parses as $\tau_1 \to (\tau_2 \to \tau_3)$, so some of the parentheses in the above two types are unnecessary.

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

```
def split[A,B,C](f: Either[A,B] => C) = ({a: A => f(Left(a))},
{b: B => f(Right(b))})
```

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

```
def merge[A,B,C](f: A => C, g: A => C) = {x: Either[A,B] => x
 match { case Left(a) => f(a) case Right(b) => g(b) }}
```

Alternative form:

```
def merge[A,B,C](f: A => C, g: A => C)(x: Either[A,B]) = x match
{ case Left(a) => f(a) case Right(b) => g(b) }
```

2. **Typing derivations**

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

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

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int} + \texttt{bool}} \quad \dfrac{\dfrac{}{\Gamma, y{:}\texttt{int} \vdash y : \texttt{int}} \quad \dfrac{}{\Gamma, y{:}\texttt{int} \vdash 0 : \texttt{int}}}{\Gamma, y{:}\texttt{int} \vdash y == 0 : \texttt{bool}} \quad \dfrac{}{\Gamma, z{:}\texttt{bool} \vdash z : \texttt{bool}}}{\Gamma \vdash \texttt{case}\ x\ \texttt{of}\ \{\texttt{left}(y) \Rightarrow y == 0\ ;\ \texttt{right}(z) \Rightarrow z\} : \texttt{bool}}}{\vdash \lambda x{:}\texttt{int} + \texttt{bool}.\texttt{case}\ x\ \texttt{of}\ \{\texttt{left}(y) \Rightarrow y == 0\ ;\ \texttt{right}(z) \Rightarrow z\} : \texttt{int} + \texttt{bool} \to \texttt{bool}}$$

where $\Gamma = x{:}\texttt{int} + \texttt{bool}$.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x{:}\texttt{int} \times \texttt{int}}}{\Gamma \vdash \texttt{fst}\ x : \texttt{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}\texttt{int} \times \texttt{int}}}{\Gamma \vdash \texttt{snd}\ x : \texttt{int}}}{\Gamma \vdash \texttt{fst}\ x == \texttt{snd}\ x : \texttt{bool}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash x{:}\texttt{int} \times \texttt{int}}}{\Gamma \vdash \texttt{fst}\ x : \texttt{int}}}{\Gamma \vdash \texttt{left}(\texttt{fst}\ x) : \texttt{int} + \texttt{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash x{:}\texttt{int} \times \texttt{int}}}{\Gamma \vdash \texttt{snd}\ x : \texttt{int}}}{\Gamma \vdash \texttt{right}(\texttt{snd}\ x) : \texttt{int} + \texttt{int}}}{\Gamma \vdash \texttt{if}\ \texttt{fst}\ x == \texttt{snd}\ x\ \texttt{then}\ \texttt{left}(\texttt{fst}\ x)\ \texttt{else}\ \texttt{right}(\texttt{snd}\ x) : \texttt{int} + \texttt{int}}}{\vdash \lambda x{:}\texttt{int} \times \texttt{int}.\texttt{if}\ \texttt{fst}\ x == \texttt{snd}\ x\ \texttt{then}\ \texttt{left}(\texttt{fst}\ x)\ \texttt{else}\ \texttt{right}(\texttt{snd}\ x) : \texttt{int} \times \texttt{int} \to \texttt{int} + \texttt{int}}$$

where $\Gamma = x{:}\texttt{int} \times \texttt{int}$.

3. **Lists**

   (a) Typing rules:

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{list}[\tau]} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{list}[\tau]}{\Gamma \vdash e_1 :: e_2 : \texttt{list}[\tau]}$$

   (b) Evaluation rules:

$$\frac{}{\texttt{nil} \Downarrow \texttt{nil}} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}$$

$$\frac{e_0 \Downarrow \texttt{nil} \quad e \Downarrow v}{\texttt{case}_{\texttt{list}}\ e_0\ \texttt{of}\ \{\texttt{nil} \Rightarrow e\ ;\ \cdots\} \Downarrow v} \qquad \frac{e_0 \Downarrow v_1 :: v_2 \quad e_1[v_1/x, v_2/y] \Downarrow v}{\texttt{case}_{\texttt{list}}\ e_0\ \texttt{of}\ \{\cdots\ ;\ x :: y \Rightarrow e\} \Downarrow v}$$

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

   (a) i. **The following approach uses pairs. Another valid approach is to use currying and uncurrying, but this is a little more complicated.**

$$\begin{aligned} &\texttt{let fun}\ f(x_1 : \tau_1, x_2 : \tau_2) = e_1\ \texttt{in}\ e_2 \\ &\iff \texttt{let fun}\ f(p : \tau_1 \times \tau_2) = e_1[\texttt{fst}\ p/x_1, \texttt{snd}\ p/x_2]\ \texttt{in}\ e_2 \end{aligned}$$

$$f(e_1, e_2) \iff f((e_1, e_2))$$

   Notice that the left hand side $f(e_1, e_2)$ is a two-argument function call and $f((e_1, e_2))$ is a one-argument function call where the argument is a pair.

   ii.

$$\frac{\Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_1 : \tau_3 \quad \Gamma, f{:}\tau_1 \times \tau_2 \to \tau_3 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let fun}\ f(x_1 : \tau_1, x_2 : \tau_2) = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\Gamma(f) = \tau_1 \times \tau_2 \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash f(e_1, e_2) : \tau}$$

   These rules only consider named function definitions/ calls with multiple arguments

   iii. **For functions with 3 arguments, we could use a similar idea with triples represented as $(e_1, (e_2, e_3))$ and substituting $\texttt{fst}\ z$ for x_1, $\texttt{fst}\ (\texttt{snd}\ z)$ for x_2 and so on. Likewise for an arbitrary number of arguments using iterated pairing.**

   (b)

$$\begin{aligned} &\texttt{let}\ p = \texttt{rec}\ p(z{:}\texttt{unit}) : (\texttt{int} \to \texttt{bool}) \times (\texttt{int} \to \texttt{bool}) = \\ &\texttt{let pair}\ (even, odd) = p()\ \texttt{in} \\ &\quad (\lambda x{:}\texttt{int}.\ \texttt{if}\ x == 0\ \texttt{then true else}\ even\ (x-1), \\ &\quad \lambda x{:}\texttt{int}.\ \texttt{if}\ x == 0\ \texttt{then false else}\ odd\ (x-1)) \\ &\texttt{in} \\ &\texttt{let pair}\ (even, odd) = p()\ \texttt{in} \\ &e \end{aligned}$$

   Notice that we need to add a (unused) argument $z : \texttt{unit}$, because $\texttt{rec}$ requires a function argument.