Elements of Programming Languages
Tutorial 3: Data structures and polymorphism
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 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 (that is, case classes) `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(a,b)}
   ```

   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 $\tau_1 \to (\tau_1 \to \tau_3)$, 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**

(a) $\Lambda A.\lambda x{:}A.x + 1$ **does not typecheck because** A **is not** `int`**.**

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}???\\ x{:}A \vdash x : \mathtt{int}\end{array} \quad \dfrac{}{x{:}A \vdash 1 : \mathtt{int}}}{x{:}A \vdash x + 1 : \mathtt{int}}}{\vdash \lambda x{:}A.x + 1 : \mathtt{int}}}{\vdash \Lambda A.\lambda x{:}A.x + 1 : \forall A.\mathtt{int}}$$

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

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

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

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

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

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \mathtt{fst}\ x : A} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \mathtt{snd}\ x : A}}{\Gamma \vdash \mathtt{fst}\ x == \mathtt{snd}\ x : \mathtt{bool}} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \mathtt{fst}\ x : A} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \mathtt{snd}\ x : A}}{\Gamma \vdash \mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{fst}\ x\ \mathtt{else}\ \mathtt{snd}\ x : A}}{\vdash \lambda x{:}A \times A.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{fst}\ x\ \mathtt{else}\ \mathtt{snd}\ x : A \times A \to A}}{\vdash \Lambda A.\lambda x{:}A \times A.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{fst}\ x\ \mathtt{else}\ \mathtt{snd}\ x : \forall A.A \times A \to A}$$

where $\Gamma = x{:}A \times A$. **this only works because we have defined $==$'s typing rule so that any two values of the same type can be compared for equality, including two values of an unknown type A. However, if $==$ is restricted to base types (as in Coursework 1) then we cannot do this.**

3. **Evaluation derivations**

(a) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{int}]\ 42$ **Notice that this does not typecheck, but still evaluates OK.**

$$\dfrac{\dfrac{\dfrac{}{(\Lambda A.\lambda x{:}A.x + 1) \Downarrow (\Lambda A.\lambda x{:}A.x + 1)}}{(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{int}] \Downarrow \lambda x.\ x + 1} \quad \dfrac{}{42 \Downarrow 42} \quad \dfrac{\vdots}{42 + 1 \Downarrow 43}}{(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{int}]\ 42 \Downarrow 43}$$

(b) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{bool}]\ \mathtt{true}$ **This does not typecheck, and does not evaluate either, because when we try to add** `true` **to 1 we get stuck.**

$$\dfrac{\dfrac{\dfrac{}{(\Lambda A.\lambda x{:}A.x + 1) \Downarrow (\Lambda A.\lambda x{:}A.x + 1)}}{(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{bool}] \Downarrow \lambda x.x + 1} \quad \dfrac{}{\mathtt{true} \Downarrow \mathtt{true}} \quad \begin{array}{c}???\\ \mathtt{true} + 1 \Downarrow ???\end{array}}{(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{bool}]\ \mathtt{true} \Downarrow ??}$$

4. **Multiple-argument functions**

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

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

   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.

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

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

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

5. **Mutual recursion**

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

   Notice that we need to add a (unused) argument z : `unit`, because `rec` requires a function argument.