Elements of Programming Languages
Tutorial 3: Data structures and polymorphism
Solution notes

Exercises marked ⋆ 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(_), 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)

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

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

```scala
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))

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

Equivalent alternative form:

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

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

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

Equivalent alternative form:

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

Notice that τ₁ → τ₂ → τ₃ parses as (τ₁ → (τ₂ → τ₃)), so some of the parentheses in the above two types are unnecessary.

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

```scala
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)

```scala
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:

```scala
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) \( \lambda x: \text{int} + \text{bool}. \text{case } x \text{ of } \{ \text{left}(y) \Rightarrow y == 0; \text{right}(z) \Rightarrow z \} \)

\[
\Gamma \vdash x: \text{int} + \text{bool} \quad \Gamma, y: \text{int} \vdash y == 0: \text{int} \\
\Gamma, y: \text{int} \vdash y == 0: \text{bool} \\
\Gamma, z: \text{bool} \vdash z: \text{bool} \\
\Gamma \vdash \text{case } x \text{ of } \{ \text{left}(y) \Rightarrow y == 0; \text{right}(z) \Rightarrow z \}: \text{int} + \text{bool} \rightarrow \text{bool}
\]

where \( \Gamma = x: \text{int} + \text{bool} \).

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

\[
\Gamma \vdash x: \text{int} \times \text{int} \quad \Gamma \vdash x: \text{int} \times \text{int} \\
\Gamma \vdash \text{fst } x: \text{int} \\
\Gamma \vdash \text{snd } x: \text{int} \\
\Gamma \vdash \text{if } \text{fst } x == \text{snd } x \text{ then left(\text{fst } x) } \text{else right(\text{snd } x)}: \text{int} \times \text{int} \rightarrow \text{int} + \text{int}
\]

where \( \Gamma = x: \text{int} \times \text{int} \).

3. Lists

(a) Typing rules:

\[
\Gamma \vdash \text{nil}: \text{list}[\tau] \\
\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \text{list}[\tau] \\
\Gamma \vdash \text{cons } e_1 e_2 : \text{list}[\tau]
\]

(b) Evaluation rules:

\[
e_0 \Downarrow \text{nil} \quad e \Downarrow v \\
e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \\
\text{case } e_0 \text{ of } \{ \text{nil } \Rightarrow e; \cdots \} \Downarrow v \\
\text{case } e_0 \text{ of } \{ \cdots ; x : y \Rightarrow e \} \Downarrow v
\]

4. (\ast) 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.

\[
\text{let } f(x_1 : \tau_1, x_2 : \tau_2) = e_1 \text{ in } e_2 \\
\iff \text{let } f(p : \tau_1 \times \tau_2) = e_1[fst p/x, snd p/x] \text{ in } e_2
\]

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

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. \( \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e_1 : \tau_3 \quad \Gamma, f : \tau_1 \times \tau_2 \rightarrow \tau_3 \vdash e_2 : \tau \)

\[
\Gamma \vdash \text{let } f(x_1 : \tau_1, x_2 : \tau_2) = e_1 \text{ in } e_2 : \tau \\
\Gamma \vdash f(x_1 : \tau_1, x_2 : \tau_2) = e_1 \text{ in } e_2 : \tau
\]

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 \( \text{fst } z \) for \( x_1 \), \( \text{fst } (\text{snd } z) \) for \( x_2 \) and so on. Likewise for an arbitrary number of arguments using iterated pairing.
let $p = \text{rec } p(z:\text{unit}):\text{(int $\to$ bool)} \times \text{(int $\to$ bool)} =$

$(\lambda x:\text{int}. \text{if } x == 0 \text{ then true else snd } p())(x - 1),$

$(\lambda x:\text{int}. \text{if } x == 0 \text{ then false else fst } p())(x - 1))$

in

let even = fst $p()$ in
let odd = snd $p()$ in

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