Elements of Programming Languages  
Tutorial 4: Subtyping and polymorphism  
Week 6 (October 23–27, 2023)

Exercises marked ⋆ are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Subtyping and type bounds**

   Consider the following Scala code:

   ```scala
   abstract class Super
   case class Sub1(n: Int) extends Super
   case class Sub2(b: Boolean) extends Super
   ``)

   This defines an abstract superclass `Super`, and subclasses with integer and boolean parameters.

   (a) What subtyping relationships hold as a result of the above declarations?

   (b) For each of the following subtyping judgments, write a derivation showing the judgment holds or argue that it doesn’t hold.

   i. `Sub1 × Sub2 <: Super × Super`
   ii. `Sub1 → Sub2 <: Super → Super`
   iii. `Super → Super <: Sub1 → Sub2`
   iv. `Super → Sub1 <: Sub2 → Super`
   v. `(⋆)(Sub1 → Sub1) → Sub2 <: (Super → Sub1) → Super`

   (c) Suppose we have a function

   ```scala
   def f1(x: Super): Super = x match {
     case Sub1(n) => x
     case Sub2(b) => x
   }
   ``)

   that simply inspects the type of the argument but preserves the value. Try running `f1` on `Sub2(true)`. What type does it have? What happens if you try to access the `b` field of the result?

   (d) Now consider a different version of this function:

   ```scala
   def f2[A](x: A): A = x match {
     case Sub1(n) => x
     case Sub2(b) => x
   }
   ``)

   where we have abstracted over the argument type. Does this typecheck? Why or why not? If it typechecks, what happens if we apply it to values of type `Sub1, Sub2, Int`?
2. Subtyping and Contravariance

Consider the following Scala declarations:

```scala
abstract class Shape
class Rectangle(...) extends Shape
class Circle(...) extends Shape
```

Thus, Rectangle <: Shape and Circle <: Shape.

(a) Suppose we have a function `f: (Shape => Int) => Int`. What could `f` potentially do with its argument? Does the type system allow us to pass a function of type `Rectangle => Int` to `f`?

(b) Suppose we have a function `g: (Circle => Int) => Int`. What could `g` potentially do with its argument? Does the type system allow us to pass a function of type `Shape => Int` to `g`?

3. Type parameters

Some types, such as lists, are naturally thought of as parameterized. For example, in Scala, the type `List[A]` takes a parameter `A`, the type of elements of the lists.

Consider the following Scala code:

```scala
abstract class List[A]
case class Nil[A]{} extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

This defines a recursive data structure, consisting of lists. (Notice however that `Nil` is a case class and so it carries a type annotation and empty parameter list.)

(a) Using the same approach as above, define a type `Tree[A]` for binary trees whose leaves are labeled by values of type `A`, but nodes do not contain `A`-values. There should be two constructors for such trees: `Leaf(a)` constructing a leaf with data `a`, and `Node(t1, t2)` taking two trees and constructing a tree.

(b) Define a recursive function `sum` that adds up all of the integers in a `Tree[Int].`

(c) Define a recursive function `map: Tree[A] => (A => B) => Tree[B]` that applies a given function `f: A => B` to all of the `A` values on the leaves of a `Tree[A].`

(d) (⋆) Define a function `flatten: Tree[Tree[A]] => Tree[A].`

(e) (⋆) Define a function `flatMap : (Tree[A]) => (A => Tree[B]) => Tree[B]`