Elements of Programming Languages
Tutorial 4: Subtyping and polymorphism
Week 6 (October 23–27, 2017)

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

1. **Subtyping and type bounds**

   Consider the following Scala code:

```
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. $Sub1 \times Sub2 <: Super \times Super$
      ii. $Sub1 \to Sub2 <: Super \to Super$
      iii. $Super \to Super <: Sub1 \to Sub2$
      iv. $Super \to Sub1 <: Sub2 \to Super$
      v. ($\star$) $(Sub1 \to Sub1) \to Sub2 <: (Super \to Sub1) \to Super$

   (c) Suppose we have a function

```
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:

```
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`?

(e) Finally, consider this version:

```
def f3[A <: Super](x: A): A = x match {
  case Sub1(n) => x
  case Sub2(b) =>x
}
```

Here, we have used Scala's support for a feature called *type bounds* to constrain `A` to be a subtype of `Super`, with return type `A`. Does this typecheck? Why or why not? If it typechecks, does it solve the problems we encountered with `f1` and `f2`?

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

   (a) $\Lambda A.\lambda x{:}A.x + 1$

   (b) ($\star$) $\Lambda A.\lambda x{:}A \times A.\mathtt{if}\ \mathtt{fst}\ x == \mathtt{snd}\ x\ \mathtt{then}\ \mathtt{fst}\ x\ \mathtt{else}\ \mathtt{snd}\ x$ (and how does its well-formedness depend on the typing rule for equality?)

3. **Evaluation derivations**

   Construct evaluation derivations for the following expressions, or explain why they do not evaluate:

   (a) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{int}]\ 42$

   (b) $(\Lambda A.\lambda x{:}A.x + 1)[\mathtt{bool}]\ \mathtt{true}$

4. ($\star$) **Lists and polymorphism**

   Recall the proposed rules for lists from the previous tutorial.

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

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

   (a) Write a polymorphic function *map* that has this type:

$$\forall A.\forall B.(A \to B) \to (\mathtt{list}[A] \to \mathtt{list}[B])$$

   so that $map(f)(l)$ is the function that traverses a list of A's and, for each element x in l, applies the function f to it.

   (b) Write out a typing derivation tree for the expression

$$map[\mathtt{int}][\mathtt{int}](\lambda x.x + 1)(2 :: \mathtt{nil})$$

   assuming that map has the type given above.

   (c) Are lists and their associated operations definable in $\mathsf{L}_{\mathsf{Poly}}$ already? Why or why not?