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`

(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`?
(e) Finally, consider this version:

```scala
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 type-check? 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) $(\ast) \Lambda A.\lambda x:A \times A.\text{if~}\text{fst~}x =\text{snd~}x\text{~then~}\text{fst~}x\text{~else~}\text{snd~}x$

3. **Evaluation derivations**

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

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

(b) $(\Lambda A.\lambda x:A \times A.\text{if~}\text{fst~}x =\text{snd~}x\text{~then~}\text{fst~}x\text{~else~}\text{snd~}x)[\text{bool}]\,\text{true}$

4. **Lists and polymorphism**

Recall the proposed rules for lists from the previous tutorial.

$$
e ::= \cdots | \text{nil} | e_1 :: e_2 | \text{case}_\text{list} e \text{ of } \{ \text{nil } \Rightarrow e_1 ; \ x :: y \Rightarrow e_2 \}
$$

$$
u ::= \cdots | \text{nil} | v_1 :: v_2
$$

$$
\tau ::= \cdots | \text{list}[\tau]
$$

Define $L_{\text{List}}$ to be $L_{\text{Poly}}$ extended with the above constructs.

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

$$\forall A.\forall B.(A \rightarrow B) \rightarrow (\text{list}[A] \rightarrow \text{list}[B])$$

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

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

assuming that `map` has the type given above.

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