Elements of Programming Languages
Tutorial 4: Subtyping and imperative programming
Week 6 (October 26–30, 2015)

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

1. Imperative programming
Write evaluation derivations for the following imperative programs, starting with the environment \( \sigma = [x = 3, y = 4] \).

(a) \( y := x + x \)
(b) \( \text{if } x == y \text{ then } x := x + 1 \text{ else } y := y + 2 \)
(c) \( \text{while } x < y \text{ do } x := x + 1 \)

2. 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. \( \text{Sub1} \times \text{Sub2} <: \text{Super} \times \text{Super} \)
   ii. \( \text{Sub1} \rightarrow \text{Sub2} <: \text{Super} \rightarrow \text{Super} \)
   iii. \( \text{Super} \rightarrow \text{Super} <: \text{Sub1} \rightarrow \text{Sub2} \)
   iv. \( \text{Super} \rightarrow \text{Sub1} <: \text{Sub2} \rightarrow \text{Super} \)
   v. \( (\ast) (\text{Sub1} \rightarrow \text{Sub1}) \rightarrow \text{Sub2} <: (\text{Super} \rightarrow \text{Sub1}) \rightarrow \text{Super} \)

(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 \( \text{Sub2}(true) \). 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 typecheck? Why or why not? If it typechecks, does it solve the problems we encountered with f1 and f2?

3. Lists

We could add built-in lists to LPoly as follows:

```
e ::= ··· | nil | e1 :: e2 | case_list e of {nil ⇒ e1 ; x :: y ⇒ e2}
v ::= ··· | nil | v1 :: v2
t ::= ··· | list[t]
```

Define LList to be LPoly extended with the above constructs.

The typing rule for case_list is:

```
Γ ⊢ e : list[t]  Γ ⊢ e1 : τ'  Γ, x : τ, y : list[τ] ⊢ e2 : τ'
Γ ⊢ case_list e of {nil ⇒ e1 ; x :: y ⇒ e2} : τ'
```

The basic idea here is: Given a list e, a case_list expression does a case analysis. If e evaluates to nil, then we evaluate e1. Otherwise, e must evaluate to a non-empty list of the form v :: v', and we bind x to the head element v and y to the tail v', and evaluate e2.

(a) Write appropriate typing rules for nil and ::.
(b) Write a polymorphic function map that has this type:

```
∀A. ∀B. (A → B) → (list[A] → 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.

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

```
map[int][int][λx.x + 1](2 :: nil)
```

assuming that map has the type given above.

(d) Write appropriate evaluation rules for the above constructs.

(e) Are lists and their associated operations definable in LPoly already? Why or why not?