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

Exercises marked $\star$ are more advanced. Please try all unstarred 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) $\texttt{if}\ x == y\ \texttt{then}\ x := x + 1\ \texttt{else}\ y := y + 2$

   (c) $\texttt{while}\ x < y\ \texttt{do}\ x := x + 1$

2. **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`?

3. **Lists**

We could add built-in lists to $\mathsf{L}_{\mathsf{Poly}}$ as follows:

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

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

The typing rule for $\texttt{case}_{\texttt{list}}$ is:

$$\frac{\Gamma \vdash e : \texttt{list}[\tau] \quad \Gamma \vdash e_1 : \tau' \quad \Gamma, x{:}\tau, y{:}\texttt{list}[\tau] \vdash e_2 : \tau'}{\Gamma \vdash \texttt{case}_{\texttt{list}}\ e\ \texttt{of}\ \{\texttt{nil} \Rightarrow e_1\ ;\ x :: y \Rightarrow e_2\} : \tau'}$$

The basic idea here is: Given a list e, a $\texttt{case}_{\texttt{list}}$ expression does a case analysis. If e evaluates to `nil`, then we evaluate e_1. 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 e_2.

(a) Write appropriate typing rules for `nil` and $::$.

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

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

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

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

assuming that map has the type given above.

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

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