Elements of Programming Languages
Tutorial 5: Modules and Objects
Solution notes

1. **Typing derivations**

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

   (a) $\Lambda A.\lambda x{:}A.x + 1$ **does not typecheck because** A **is not** `int`**.**

$$\dfrac{\dfrac{\overset{???}{x{:}A \vdash x : \texttt{int}} \quad \overline{x{:}A \vdash 1 : \texttt{int}}}{x{:}A \vdash x + 1 : \texttt{int}}}{\dfrac{\vdash \lambda x{:}A.x + 1 : A \to \texttt{int}}{\vdash \Lambda A.\lambda x{:}A.x + 1 : \forall A.A \to \texttt{int}}}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{fst } x : A} \quad \dfrac{\overline{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{snd } x : A}}{\Gamma \vdash \texttt{fst } x == \texttt{snd } x : \texttt{bool}} \quad \dfrac{\overline{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{fst } x : A} \quad \dfrac{\overline{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{snd } x : A}}{\Gamma \vdash \texttt{if fst } x == \texttt{snd } x \texttt{ then fst } x \texttt{ else snd } x : A}}{\dfrac{\vdash \lambda x{:}A \times A.\texttt{if fst } x == \texttt{snd } x \texttt{ then fst } x \texttt{ else snd } x : A \times A \to A}{\vdash \Lambda A.\lambda x{:}A \times A.\texttt{if fst } x == \texttt{snd } x \texttt{ then fst } x \texttt{ else snd } x : \forall A.A \times A \to A}}$$

   where $\Gamma = x{:}A \times A$. **this only works because we have defined** $==$**'s typing rule so that any two values of the same type can be compared for equality, including two values of an unknown type** A**. However, if** $==$ **is restricted to base types (as in Coursework 1) then we cannot do this.**

2. **Evaluation derivations**

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

   (a) $(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}]\ 42$ **Notice that this does not typecheck, but still evaluates OK.**

$$\dfrac{\dfrac{\overline{\Lambda A.\lambda x{:}A.x + 1 \Downarrow \Lambda A.\lambda x{:}A.x + 1} \quad \overline{\lambda x{:}\texttt{int}.x + 1 \Downarrow \lambda x{:}\texttt{int}.x + 1}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}] \Downarrow \lambda x{:}\texttt{int}.\ x + 1} \quad \overline{42 \Downarrow 42} \quad \dfrac{\vdots}{42 + 1 \Downarrow 43}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}]\ 42 \Downarrow 43}$$

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

   This does not typecheck, and does not evaluate either, because when we try to add `true` **to 1 we get stuck.**

$$\dfrac{\dfrac{\overline{(\Lambda A.\lambda x{:}A.x + 1) \Downarrow (\Lambda A.\lambda x{:}A.x + 1)} \quad \overline{\lambda x{:}\texttt{bool}.x + 1 \Downarrow \lambda x{:}\texttt{bool}.x + 1)}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{bool}] \Downarrow \lambda x{:}\texttt{bool}.x + 1} \quad \overline{\texttt{true} \Downarrow \texttt{true}} \quad \overset{???}{\texttt{true} + 1 \Downarrow ???}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{bool}]\ \texttt{true} \Downarrow ??}$$

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

   (a)

$$\begin{array}{l}\Lambda A.\Lambda B.\lambda f{:}A \to B.\texttt{rec } map(x{:}\texttt{list}[A]).\\ \quad \texttt{case}_{\texttt{list}}\ x\ \texttt{of}\ \{\texttt{nil} \Rightarrow \texttt{nil}\ ;\ x :: xs \Rightarrow (fx) :: map(xs)\}\end{array}$$

   Notice that the `rec` **only handles the inner function call.**

(b)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash map : \forall A.\forall B.(A \to B) \to (\texttt{list}[A] \to \texttt{list}[B])}}{\vdash map[\texttt{int}] : \forall B.(\texttt{int} \to B) \to (\texttt{list}[\texttt{int}] \to \texttt{list}[B])}}{\vdash map[\texttt{int}][\texttt{int}] : (\texttt{int} \to \texttt{int}) \to (\texttt{list}[\texttt{int}] \to \texttt{list}[\texttt{int}])} \quad \dfrac{\dfrac{}{x{:}\texttt{int} \vdash x{:}\texttt{int}} \quad \dfrac{}{x{:}\texttt{int} \vdash 1 : \texttt{int}}}{x{:}\texttt{int} \vdash x + 1 : \texttt{int}}}{\vdash \lambda x{:}\texttt{int}.x + 1 : \texttt{int} \to \texttt{int}} \quad \dfrac{\vdash 2 : \texttt{int} \quad \vdash \texttt{nil} : \texttt{list}[\texttt{int}]}{\vdash (2 :: \texttt{nil}) : \texttt{list}[\texttt{int}]}}{\vdash map[\texttt{int}][\texttt{int}](\lambda x.x + 1) : \texttt{list}[\texttt{int}] \to \texttt{list}[\texttt{int}]}}{\vdash map[\texttt{int}][\texttt{int}](\lambda x.x + 1)(2 :: \texttt{nil}) : \texttt{list}[\texttt{int}]}$$

(c) This question is intended to provoke discussion; the answer to this question depends on what "definable" means, which is not a concept we have carefully defined.

In one sense, lists and the list operations are not definable, because there is no way to create a data structure of infinite "size" using just pairs and sums (e.g. for any finite program, we can bound the maximum size of a data structure the program constructs.)

In another reasonable sense, lists could be defined (in principle) by encoding pairs, sums, and lists into natural numbers (assuming infinite precision arithmetic). However, this too might be unsatisfactory, since we would not easily be able to do this uniformly in the type of list elements τ, and it would be very difficult to translate a polymorphic program operating over lists.

4. **Modules and Interfaces in Scala**

(a) The components are accessed as follows:

```
A.c A.d A.f B.c B.d B.f
```

(b) After the two import statements, `d` refers to the string value `B.d = "1234"` since this was the most recent import. If we import in the opposite order it refers to `A.d = 2`.

(c) The trait should be something like:

```
trait ABlike {
  type T
  val c: T
  val d: T
  def f(x: T, y: T): T
}
```

(d)

```
def g(x: ABlike) = x.f(x.c,x.d)
```

According to the Scala interpreter the return type is `x.T`.

(e)

```
g(new ABlike{
  type T = Boolean
  val c = true
  val d = false
  def f(x: T, y: T) = x && y
})
```

5. ($\star$) **Ad hoc polymorphism**

(a)

```
abstract class List[A] extends HasSize
case class Nil[A]() extends List[A] {
  def size() = 0
}
case class Cons[A](head: A, tail: List[A]) extends List[A] {
  def size() = tail.size() + 1
}
```

(b)

```
abstract class Tree[A] extends HasSize
case class Leaf[A](a: A) extends Tree[A] {
  def size() = 1
}
case class Node[A](t1: Tree[A], t2: Tree[A]) extends Tree[A] {
  def size() = t1.size() + t2.size()
}
```

(c)

```
def sameSize(x: HasSize, y: HasSize) = x.size() == y.size()
```

```
scala> sameSize(Cons(1,Nil()), Leaf("abc"))
res2: Boolean = true
```