Elements of Programming Languages
Tutorial 8: References and laziness
Solution notes

1. **Semantics of references**

   (a) The obvious rules are as follows:

   $\boxed{\sigma, e \mapsto \sigma', e'}$

   $$\frac{\sigma, e_1 \mapsto \sigma', e_2}{\sigma, e_1; e_2 \mapsto \sigma', e_1'; e_2} \qquad \frac{}{\sigma, v; e_2 \mapsto \sigma, e_2}$$

   (b) The derivation should look like this:

   $$\begin{array}{rl} & \texttt{let}\ r = \texttt{ref}(\texttt{ref}(42))\ \texttt{in}\ !(!(r)) \\ \mapsto & [\ell_1 = 42], \texttt{let}\ r = \texttt{ref}(\ell_1)\ \texttt{in}\ !(!(r)) \\ \mapsto & [\ell_1 = 42, \ell_2 = \ell_1], \texttt{let}\ r = \ell_2\ \texttt{in}\ !(!(r)) \\ \mapsto & [\ell_1 = 42, \ell_2 = \ell_1], !(!(\ell_2)) \\ \mapsto & [\ell_1 = 42, \ell_2 = \ell_1], !(\ell_1) \\ \mapsto & [\ell_1 = 42, \ell_2 = \ell_1], 42 \end{array}$$

   It may also be helpful to draw a "box and arrow" picture showing the reference contents as cells with arrows linking them.

   (c) The derivation should look like this:

   $$\begin{array}{rl} & \texttt{let}\ r = \texttt{ref}(\lambda x.\ x)\ \texttt{in}\ r := (\lambda x.\ x + 1); (!r)(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x], \texttt{let}\ r = \ell\ \texttt{in}\ r := (\lambda x.\ x + 1); !r(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x], \ell := (\lambda x.\ x + 1); !\ell(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x + 1], (); !\ell(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x + 1], !\ell(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x + 1], !\ell(\lambda x.x + 1)(\texttt{true}) \\ \mapsto & [\ell = \lambda x.\ x + 1], \texttt{true} + 1 \end{array}$$

2. **Interaction of references and evaluation order**

   (a) call-by-value

   In call-by-value, the argument will be evaluated before the function is called, so x will be incremented (with new value 43) and then dereferenced, and 43 will be passed into the function. 43 will be printed twice.

   (b) call-by-name

   In call-by-name, the argument will be passed in as an unevaluated expression. During the first $print$ operation, x will be incremented to 43 and 43 will be printed. Likewise in the second $print$ operation, x will be incremented to 44 and 44 will be printed.

   (c) call-by-need / lazy evaluation

   In call-by-need, the argument is replaced by a reference to the unevaluated expression. The first time it is needed, during the first $print$ operation, it is evaluated, causing x to be incremented to 43 and then dereferenced, and the value 43 replaces the expression in the reference, and finally 43 is printed. The second time it is needed, the value 43 is looked up and printed again.

3. **Interaction of references and evaluation order**

   (a) call-by-value
   In call-by-value, the argument will be evaluated before the function is called, so x will be incremented (with new value 43) and then dereferenced, and 43 will be passed into the function. 43 will be printed twice.

   (b) call-by-name
   In call-by-name, the argument will be passed in as an unevaluated expression. During the first $print$ operation, x will be incremented to 43 and 43 will be printed. Likewise in the second $print$ operation, x will be incremented to 44 and 44 will be printed.

   (c) call-by-need / lazy evaluation
   In call-by-need, the argument is replaced by a reference to the unevaluated expression. The first time it is needed, during the first $print$ operation, it is evaluated, causing x to be incremented to 43 and then dereferenced, and the value 43 replaces the expression in the reference, and finally 43 is printed. The second time it is needed, the value 43 is looked up and printed again.

4. **Embedding $\mathsf{L}_{\mathsf{While}}$ in Scala**

   (a) The intended solution is as follows:

```
val skip = ()
def seq(s1: => Unit,s2: => Unit) = {
  s1
  s2
}
def assign[T](x: Ref[T], e: => T) = x.set(e)
def ifthenelse(e: => Boolean, s1: => Unit, s2: => Unit) = {
  if (e) {s1} else {s2}
}
def whiledo(e: => Boolean, s: => Unit): Unit = {
  if (e) {
    s
    whiledo(e,s)
  }
}
```

   (b) The statements may contain code that does not (and should not) be run. For example, in `ifthenelse` the non-taken branch should not be executed – doing so could result in incorrect side-effects. The expressions and statements may depend on (or update) references, and we may need to evaluate them more than once, or in a different state than the one present when the statement operation was called. For example, with `whiledo`, the `e` argument may need to be evaluated many times. If we passed this argument by value, then it would be evaluated only once to a fixed Boolean value and not re-evaluated after the first iteration, so the loop would either run forever or halt immediately.

   (c) ($\star$) Variable occurrences in expressions correspond to dereferences $!x$. So, we need to replace such occurrences with $x.get$.

5. **Interaction of references and evaluation order**

   (a) call-by-value
   In call-by-value, the argument will be evaluated before the function is called, so x will be incremented (with new value 43) and then dereferenced, and 43 will be passed into the function. 43 will be printed twice.

   (b) call-by-name
   In call-by-name, the argument will be passed in as an unevaluated expression. During the first $print$ operation, x will be incremented to 43 and 43 will be printed. Likewise in the second $print$ operation, x will be incremented to 44 and 44 will be printed.

   (c) call-by-need / lazy evaluation
   In call-by-need, the argument is replaced by a reference to the unevaluated expression. The first time it is needed, during the first $print$ operation, it is evaluated, causing x to be incremented to 43 and then dereferenced, and the value 43 replaces the expression in the reference, and finally 43 is printed. The second time it is needed, the value 43 is looked up and printed again.

6. **Embedding $\mathsf{L}_{\mathsf{While}}$ in Scala**

   (a) The intended solution is as follows:

```
val skip = ()
def seq(s1: => Unit,s2: => Unit) = {
  s1
  s2
}
def assign[T](x: Ref[T], e: => T) = x.set(e)
def ifthenelse(e: => Boolean, s1: => Unit, s2: => Unit) = {
  if (e) {s1} else {s2}
}
def whiledo(e: => Boolean, s: => Unit): Unit = {
  if (e) {
    s
    whiledo(e,s)
  }
}
```

   (b) The statements may contain code that does not (and should not) be run. For example, in `ifthenelse` the non-taken branch should not be executed – doing so could result in incorrect side-effects. The expressions and statements may depend on (or update) references, and we may need to evaluate them more than once, or in a different state than the one present when the statement operation was called. For example, with `whiledo`, the `e` argument may need to be evaluated many times. If we passed this argument by value, then it would be evaluated only once to a fixed Boolean value and not re-evaluated after the first iteration, so the loop would either run forever or halt immediately.

   (c) ($\star$) Variable occurrences in expressions correspond to dereferences $!x$. So, we need to replace such occurrences with $x.get$.

7. ($\star$) **Stream programming**

   (a) `const`:

```
  def const[A](a: A): Stream[A] = SCons(a,() => const[A](a))
```

   (b) `take`:

```
def take[A](n: Int,s: Stream[A]): List[A] = (n,s) match {
  case (0,_) => List()
  case (n,SCons(a,f)) => a :: take(n-1,f())
  case (_,Empty) => List()
}
```

   (c) `repeat`:

```
def repeat[A](a: A)(f: A => A): Stream[A] =
  SCons(a,() => repeat(f(a))(f))
```

   (d) `map`:

```
def map[A,B](s: Stream[A])(f: A => B): Stream[B] = s match {
  case SCons(a,s) => SCons(f(a),() => map(s())(f))
  case Empty => Empty
}
```