Elements of Programming Languages
Tutorial 8: References and laziness
Week 10 (November 20–24, 2023)

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

1. **Semantics of references**

   (a) Give explicit small-step rules for evaluating the sequential composition expression $e_1; e_2$. (Remember that it can also be viewed as syntactic sugar for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ provided x is a fresh variable unused in either expression)

   (b) Evaluate the following expression to completion:

   $$\texttt{let}\ r = \texttt{ref}(\texttt{ref}(42))\ \texttt{in}\ !(!(r))$$

   (c) Consider the following expression:

   $$\texttt{let}\ r = \texttt{ref}(\lambda x.\ x)\ \texttt{in}\ r := (\lambda x.\ x + 1); (!r)(\texttt{true})$$

   Apply small-step evaluation to this expression until it reaches either a value or an error state.

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

   Consider the following expression e:

   $$\texttt{let}\ r = \texttt{ref}(42)\ \texttt{in}\ (\lambda x.\texttt{print}(x); \texttt{print}(x))\ (r := !r + 1; !r)$$

   where `print` is a side-effecting operation that fully evaluates its argument to a value and then prints it. For each of the following evaluation strategies, explain informally how e would be evaluated and what the printed output will be.

   (a) call-by-value
   (b) call-by-name
   (c) call-by-need / lazy evaluation

3. **Embedding $\mathsf{L_{While}}$ in Scala**

   Recall the statements of $\mathsf{L_{While}}$:

   $$\mathit{Stmt} \ni s \quad ::= \quad \texttt{skip} \mid s_1; s_2 \mid x := e \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ e\ \texttt{do}\ s$$

   In this exercise, we will show how to embed these statements into Scala, viewing $\mathsf{L_{While}}$'s variables as references using the `Ref[T]` type discussed in class:

```
class Ref[A](val x: A) {
 private var a = x
 def get = a
 def set(y: A) = { a = y }
}
```

Statements in $\mathsf{L}_{\mathsf{While}}$ will correspond to expressions of type `Unit` in Scala, and variables will correspond to instances of the `Ref[T]` type. Consider the following interface:

```
 val skip : ()
 def seq(s1: => Unit,s2: => Unit): Unit
 def assign[T](x: Ref[T], e: => T): Unit
 def Ifthenelse(e: => Boolean, s1: => Unit, s2: => Unit): Unit
 def whiledo(e: => Boolean, s: => Unit): Unit
```

Notice in particular that most arguments are passed *by name* (that is, their types are of the form `=> T`).

(a) Implement the above operations.

(b) Why do the statements and expressions in `assign`, `ifthenelse`, and `whiledo` need to be passed by name? What would happen if they were passed by value?

(c) ($\star$) We have not considered how to map $\mathsf{L}_{\mathsf{While}}$ *expressions* to $\mathsf{L}_{\mathsf{Ref}}$. In $\mathsf{L}_{\mathsf{While}}$, a mutable variable occurring in an expression is evaluated to its value. How should we adjust such expressions in $\mathsf{L}_{\mathsf{Ref}}$?

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

Consider the following `Stream` type:

```
abstract class Stream[+A]
case object Empty extends Stream[Nothing]
case class SCons[+A](h: A, t: () => Stream[A]) extends Stream[A]
```

This defines a type of *streams*, which are similar to lists, but the evaluation of the tail of a stream is delayed.

Define Scala functions on streams as follows:

(a) `const[A]: A => Stream[A]` so that `const(a)` produces an infinite stream of `a`'s.

(b) `take[A]: (Int,Stream[A]) => List[A]` so that `take(n,s)` lists the first `n` elements from `s`.

(c) `repeat[A]: (A => A) => A => Stream[A]` such that

```
  repeat(a)(f) = Stream(a,f(a),f(f(a)),..)
```

For example, `repeat(0) (incr)` should produce the stream `0,1,2,3,...`, if `incr` is the increment function.

(d) `map[A]: Stream[A] => (A => B) => Stream[B]` that applies the function `f: A => B` to each element of the stream `s: Stream[A]` yielding a stream of `B`s.