CSCI 2113.30 Lab 11

April 24, 2017

Goals

1. Review, higher-order functions and currying by re-implementing sqrt() function.
2. Implement a Set data structure using only functions

Activity

1. In the first portion of the Lab we'll re-implement the sqrt() function using a different technique while utilizing higher-order functions and currying.
2. First, download the Newtons.scala file from blackboard to study the original square root function from the lecture.
3. Now some definitions:

   Fixed point of a function is a location where input value equal to the applied value. In other words, fixed point of a function of $f(x)$ is where $x = f(x)$.

   Example:

   Fixed point of $f(x) = 1 + 2/x$ is 2: $f(2) = 1 + 2/2 = 2$

4. For some functions, we can find the fixed points by starting with an guess then applying the function over and over again until it stops changing, like this:

   $x, f(x), f(f(x)), f(f(f(x))), \ldots$

   Example:

   Given, $f(x) = 1 + 2/x$ and guess of 1...

   $x = 1$
   $f(x) = f(1) = 1 + 2/1 = 3$
   $f(f(x)) = f(3) = 1 + \frac{2}{3} = 1.666\ldots$
   $f(f(f(x))) = f(1.667) = 1 + 2/1.667 = 2.2$
   $f(f(f(f(x)))) = f(2.2) = 1.90909\ldots$

   Notice how the value oscillates around actual fixed point of 2 while approaching the value.

5. Create a file called Lab11.scala and create a single module called Lab11 and a main method:

```
object Lab11 {
  def main(args: Array[String]): Unit = {
  }
}
```

6. Inside the module, let's try to implement this method of finding the fixed point by implementing the following function:

```
def fixedPoint(f: Double => Double)(startingGuess: Double): Double = {
  ???
}
```

You might need a helper function:

```
def abs(x: Double): Double = if (x < 0) -x else x

def isCloseEnough(guess: Double, f: Double => Double): Boolean = {
  ???
}
```

The guess is good if f(guess) is equal to or close to guess. Think back to newton's method for reference. There is possible implementation at the end of this document but try to do it on your own before looking at the reference implementation.

7. Test your implementation by trying the example function from #2:

```
object Lab11 {

  ... your implementation here ...

  def main(args: Array[String]): Unit = {
    def f(x: Double) = 1 + 2/x
    val fixedPointFinder: Double => Double = fixedPoint(f)
    println(fixedPointFinder(1.0))
  }
}
```

8. Now. let's implement square root function using the fixed point function.

   Given the following function:

   $f(x) = a / x$

   Notice what happens if you apply the function to sqrt(a):

   $$\begin{aligned} f(sqrt(a)) &= a / sqrt(a) \\ &= a^{1}a^{-½} \\ &= a^{(1-½)} = a^{-½} = sqrt(a) \end{aligned}$$

   This implies that $sqrt(x)$ is the fixed point of the function $f(x) = a / x$

9. Let's define a new square root function in terms of fixedPoint:

```
def sqrt(a: Double) = fixedPoint(x => a / x)(1.0)
```

10. Test the new square root function by calling `sqrt(2.0)`. Does it work? Try some other values than 2.0. Does it still work? If it doesn't, can you add some debugging `println` statements to figure out what's going on?
11. Unfortunately, our new square root function goes into an infinite loop. The oscillation that we observed before does not damp as we recursively call our function, unlike last time. But we can improve our guess by taking the two oscillating value and averaging them:

Given $f(x) = a / x$, where $a = 2$ (in other words, $sqrt(2)$),

Initial guess: $x = 1$

Second guess: $f(\mathrm{x}) = f(1) = 2$

Third guess: $f(f(x)) = f(2) = 1$

Fourth guess: $f(f(f(x))) = f(1) = 2$

... repeats forever ...

Instead, for our third guess, let's take the average of first and second guess and use that as our guess:

Our new third guess:

$$f(\text{average of } x \text{ and } f(x)) = ½\,(x + a/x) = ½\,(1 + 2) = 1.5$$

Then do the same for every subsequent guesses, taking the average of two previous guesses.

This means our square root function can be rewritten to be:

$$sqrt(\mathrm{a}) = \text{fixed point of } f(x) = ½\,(x + a / x)$$

11. Re-implement the square root function in scala using the new definition above, then test it again.
12. Notice the similarity between how we improved the fixed point guess and and newton's method's improve function. This is a frequently occurring pattern called **Average Damping**. Let's factor this out and turn it into its own function. Our average damping function will take a function and a value then return the average between the value and the function applied to the value.

```
def averageDamp(f: Double => Double)(x: Double): Double = ???
```

13. Let's re-write our square root function once again to use the averageDamp function.
14. One you have finished your solution, compare it to the reference implementation at the end of this document.

Further Investigation

1. Notice how `fixedPoint` and `averageDamp` function was implemented using multiple argument groups, which is a form of curried function is scala. Try re-implementing the two functions without currying (as in functions with two arguments). Then try to write your square root function. Notice how it's awkward it is to combine `fixedPoint` and `averageDamp` functions. (You would have to create a brand new function to pass the result between `fixedPoint` and `averageDamp`.)
2. Biggest gain from currying a function is that it gives us easy way to combine functions together to create new functions. (Study the sum of range examples from the lectures.)

Assignment

You will implement the Set data structure using only functions and for simplicity, we'll only deal with sets of integers.

As a motivation to use only functions instead of classes to model data structure, imagine trying to represent a set of all positive integers. It's impossible to actually represent that using values and memory space using classe. So instead, we will create a function that informs us if a number is included in the set: Is 3 part of the set? No. Is -1 part of the set? Yes.

We will create a function that takes an integer as an argument and returns boolean to indicate if the integer belongs to a set. This function is called **characteristic function** of the set.

The characteristic function for the set of negative numbers would look something like this:

```
(x: Int) => x < 0
```

For this assignment, let's represent a general set by its characteristic function and for easy syntax, let's define a type alias for this representation:

```
type Set = Int => Boolean
```

Type aliases does not add any functionality other than saves us typing. So instead of typing:

```
def contains(s: Int => Boolean, elem: Int): Boolean = ...
```

We can write:

```
def contains(s: Set, elem: Int): Boolean = ...
```

Let's finish the implementation of the contains function:

```
def contains(s: Set, elem: Int): Boolean = s(elem)
```

1. Download the FunctionalSet.scala file and put all your implementation in that file.
2. Define a function **singletonSet** which creates a singleton set from one integer value: the set represents the set of the one given element. Now that we have a way to create singleton sets, we want to define a function that allow us to build bigger sets from smaller ones.

   ```
   def singletonSet(elem: Int): Set = ???
   ```

   Remember, `Set` is a function that takes an `Int` and returns a `Boolean`.
   You can test this set by running the following code:

   ```
   val setOfOne = singletonSet(2)  // Creates a set of single int, 2.
   assert(contains(setOfOne, 2) == true)
   assert(contains(setOfOne, 1) == false)
   ```

3. Define the functions **union**,**intersect**, and **diff**, which takes two sets, and return, respectively, their union, intersection and differences. diff(s, t) returns a set which contains all the elements of the set s that are not in the set t.

```
//Set of all elements that are in either s or t
def union(s: Set, t: Set): Set = ???

//Set of all elements that are in both s and t
def intersect(s: Set, t: Set): Set = ???

//Set of all elements in s but not in t
def diff(s: Set, t: Set): Set = ???
```

Usage:

```
val a = singletonSet(1)
val b = union(a, singletonSet(2))
val c = union(singletonSet(3), singletonSet(4))
assert(contains(a, 1) == true)
assert(contains(union(b, c), 3) == true)
assert(contains(intersect(a, b), 1) == true)
assert(contains(intersect(a, b), 2) == false)
assert(contains(diff(b, a), 2) == false)
assert(contains(diff(b, a), 2) == true)
```

4. Define the function filter which selects only the elements of a set that are accepted by a given predicate p. The filtered elements are returned as a new set.

```
def filter(s: Set, p: Int => Boolean): Set = ???
```

Usage:

```
val set = union(
            singltonSet(1), singltonSet(2),
            singltonSet(3), singltonSet(4))
val evenNumbers = filter(set, x => x % 2 == 0)
assert(contains(even, 1) == false)
assert(contains(even, 2) == true)
assert(contains(even, 3) == false)
assert(contains(even, 4) == true)
```

5. While above functions are useful, we would like to write functions that operates on the individual elements. Write a function that tests whether a given predicate is true for all elements of the set. This forall function has the following signature:

```
def forall(s: Set, p: Int => Boolean): Boolean
```

Note that there is no direct way to find which elements are in a set. *contains* only allows to know whether a given element is included. Thus, if we wish to do something to all elements of a set, then we have to iterate over all integers, testing each time whether it is included in the set, and if so, to do something with it. We know the range of all possible Int (32-bit integer). Here, however, we will say that an integer x has the property -1000 <= x <= 1000 in order to limit the search space. The function will have to be recursive.

Usage:

```
val set = union(
            singltonSet(1), singltonSet(2),
            singltonSet(3), singltonSet(4))
assert(forall(set, x => x < 10) == true)
assert(forall(set, x => x % 2 == 0) == false)
```

6. Using **forall**, implement a function **exists** which tests whether a set contains at least one element for which the given predicate is true.

```
def exists(s: Set, p: Int => Boolean): Boolean = ???
```

7. Finally, write a function **map** which transforms a given set into another one by applying to each of its elements the given function.

```
def map(s: Set, f: Int => Int): Set = ???
```

Usage:

```
val set = union(
            singltonSet(1), singltonSet(2),
            singltonSet(3), singltonSet(4))
val newSet = map(set, x => x * x)
assert(contains(newSet, 9) == true)
assert(contains(newSet, 16) == true)
assert(contains(newSet, 3) == false)
```

8. The FunctionalSet.scala file contains some extra functions to help with testing and debugging.
9. Submit the FunctionalSet.scala file only.

Appendix - reference implementation of square root using fixed point

```scala
object Lab11 {
  def fixedPoint(f: Double => Double)(startingGuess: Double): Double = {
    def inner(guess: Double): Double = {
      if (isCloseEnough(guess, f)) guess
      else inner(f(guess))
    }
    inner(startingGuess)
  }

  def abs(x: Double): Double = if (x < 0) -x else x

  def isCloseEnough(guess: Double, f: Double => Double): Boolean =
    abs(f(guess) - guess) < 0.0001

  def averageDamp(f: Double => Double)(x: Double): Double = (x + f(x)) / 2

  def sqrt(a: Double) = fixedPoint(averageDamp(x => a / x))(1.0)
  //without average damp function, sqrt would have looked something like this:
  //def sqrt(a: Double) = fixedPoint(x => (x + (a / x)) / 2)(1.0)


  def main(args: Array[String]): Unit = {
      println(sqrt(2))
  }
}
```