CS558 Programming Languages – Fall 2023 – Study Questions Lecture 6b

These questions are intended for self-study, to help review and deepen your understanding of the lecture. Sample answers are available. There is nothing to hand in.

1. Explain why, in a language with only downwards funargs, if function `f` defines a nested function `g`, it is impossible for `g` to be called after `f` has returned.

2. Scala has first-class functions that can have free variables, and it builds closures for them internally. But, as illustrated on slide 9, we can also use the object-oriented features of Scala to model first-class functions as objects, so that we never need to use functions with free variables. This exercise develops this idea. There is no reason to use this technique in real Scala code, but the fact that it works shows the deep similarity between closures and objects.

The basic idea is to represent a first class function of type `A => B` by a class that implements the following Scala *trait* (which is similar to an abstract class or interface in Java, C++, etc.):

```
trait FCF[A,B] {
  def apply(arg:A) : B
}
```

Here `FCF` is meant to stand for "first-class function" and `A`,`B` are *type parameters*. For example, the case class given on slide 9 implements this trait:

```
case class MultiplesOf(n: Int) extends FCF[List[Int],List[Int]] {
    def apply(xs:List[Int]) : List[Int] = xs match {
      case Nil => Nil
      case (y::ys) => if (y%n == 0) y::apply(ys) else apply(ys)
    }
}
val evens = MultiplesOf(2)
val v = evens.apply(List(1,2,3,4)) // yields List(2,4)
```

Here is the `filter` function in the same style. Note that the argument is now itself an `FCF`:

```
case class Filter[A](p: FCF[A,Boolean]) extends FCF[List[A],List[A]] {
    def apply(xs:List[A]) : List[A] = xs match {
      case Nil => Nil
      case (y::ys) => if (p.apply(y)) y::apply(ys) else apply(ys)
    }
  }
```

An ordinary Scala function can be converted into an `FCF` using this class:

```
case class MkFCF[A,B](f: A => B) extends FCF[A,B] {
  def apply(x:A) : B = f(x)
```

```
}
def evenf(x:Int) : Boolean = x%2 == 0
val even : FCF[Int,Boolean] = MkFCF(evenf)
val evens = Filter(even)
val v = evens.apply(List(1,2,3,4)) // yields List(2,4)
// or, more compactly:
val v1 = Filter(MkFCF((x:Int) => x%2 == 0)).apply(List(1,2,3,4))
```

This will actually work in Scala even if the `mkFCF` argument `f` has free variables, but the idea here is to use `MkFCF` only on functions that do *not* have any free variables, and use specialized clases like `MultiplesOf` for functions that do.

Here are two exercises:

(a) Write the `map` function from lecture 6a slide 12 as a class implementing `FCF[List[A],List[B]]`, and apply it to an FCF version of `pow` to get the cubes of `List(1,2,3)`.

(b) A more general version of the `compose` function from lecture 6a slide 11 can be written:

```
def compose[A,B,C](f: B => C, g : A => B) (x:A) = f(g(x))
```

Write this function as an FCF class, and use it to compute the equivalent of `compose((x:Int) => x > 3,(y:Int) => y * 2)`.

3. Convert the following Scala functions into continuation-passing style (CPS). Note that a necessary condition of CPS is that all function calls must be tail calls, but this is not a *sufficient* condition: CPS is *not* the same as "accumulator style."

(a)

```
  def fac(n:Int) :Int =
    if (n < 2)
      1
    else n * fact(n-1)
```

(b) This one is trickier!

```
  def fib(n:Int) :Int =
    if (n < 2)
      n
    else fib(n-1) + fib(n-2)
```