40
Functional Composition and For Comprehensions

What exactly is functional composition?

- We've already seen "higher-order functions/methods". These are methods like *map* for *List* which takes a function as its parameter.
  - But what if we apply a function to a function/method? I think we can call that "functional composition."
  - Here are a couple of simple examples:
    - `f.andThen(g)`
    - `f.compose(g)`
  - These are functional composition because we start with a function, apply it to a parameter which is a function and the result is yet another function!

Example of functional composition

- Let's suppose we have a function *f* which takes two parameters, *x* and *y*. But what we really want is a function *g* that takes the same two parameters, but in the order *y* then *x*?
- Let's write a method/function which will convert one form to the other:

```
def swapParams[T1, T2, R](f: (T1, T2) => R): (T2, T1) => R = ???
```

Example of functional composition: swapParams

```
def swapParams[T1, T2, R](f: (T1, T2) => R): (T2, T1) => R =
      { (t2,t1) => f(t1,t2) }

val g = swapParams(f)
```

The pattern pertains to the resulting function; the expression pertains to the original function.

Review: Option/Try
Introduce: Either

- We use *Option[T]*
  - to make it explicit when we may or may not have a *T* value;
  - thus we avoid the use of null;
  - to “wrap” an object returned from a java method which is *Nullable*.
- We use *Try[T]*
  - to make it explicit when we may have a *T* value or instead have an exceptional condition;
  - thus we generally avoid throwing exceptions;
  - to “wrap” an expression that might throw an exception.
- We use *Either[P,Q]*
  - when we might have *either* a *P* or a *Q*.
  - as usual, we have two cases:
    - *Left[P](p: P) extends Either[P,Nothing];*
    - *Right[Q](q: Q) extends Either[Nothing,Q].*
  - the *Right* case is (unsymmetrically) treated as the *right* case.

Either

- For example, a numeric String can be parsed as a *Double* or an *Int* (or neither).

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
  val rDouble = """(-?)([0-9]*)\.([0-9]+)""".r
  val rInt = """(-?)([0-9]+)""".r
  def parse(s: String): Option[Either[Int,Double]] = s match {
    case rDouble(_, _, _) => Some(Right(s.toDouble))
    case rInt(_, _) => Some(Left(s.toInt))
    case _ => None
  }
// Exiting paste mode, now interpreting.
scala> parse("3.1415927")
res0: Option[Either[Int,Double]] = Some(Right(3.1415927))
scala> parse("3")
res1: Option[Either[Int,Double]] = Some(Left(3))
scala> parse("X")
res2: Option[Either[Int,Double]] = None
```

Option Review (1)

- Avoiding exceptions/nulls using *Option*
  - First, what's wrong with nulls (and exception)?
    - nulls (in Java) are for lazy programmers who don't mind running into a null-pointer-exception every now and then. The problem is that they don't *force* the caller to check the result.
    - exceptions are side-effects!
  - We've briefly seen this before, for example, in the *List* method *find*:

    def find(p: (A) ⇒ Boolean): Option[A]

    Finds the first element of the list satisfying a predicate, if any.
    p the predicate used to test elements.
    returns an option value containing the first element in the list that satisfies p, or *None* if none exists.

  - *Option*, therefore, is a **container** whose value is either a *Some* (a wrapper) of a valid value, or *None*.

Option (2)

- Creating *Option* values:

```
scala> Some("hello")
res1: Some[String] = Some(hello)
scala> None
res2: None.type = None
scala> Option(null)
res3: Option[Null] = None
```

Useful if using a Java library that might return a null value

- Using *Option* values — simple ways:

```
scala> val l = List(1,2,3)
l: List[Int] = List(1, 2, 3)
scala> val y = 3
y: Int = 3
scala> val x = l.find{_==y}
x: Option[Int] = Some(3)
scala> x.isDefined
res11: Boolean = true
scala> x.get
res10: Int = 3
scala> x match {case Some(n) => println(s"found $n"); case None => println("not found")}
found 3
scala> x.getOrElse("not found")
res12: Any = 3
scala> val y = 5
y: Int = 5
scala> val x = l.find{_==y}
x: Option[Int] = None
scala> x match {case Some(n) => println(s"found $n"); case None => println("not found")}
not found
scala> x.getOrElse("not found")
res13: Any = not found
```

It's possible to use Option values this way but definitely not recommended!

Try

- Similar to *Option[T]*, *Try[T]* is a container that has one of two possible values: a *T* <u>or</u> an exception
  - The successful form is *Success(t)* where t: T
  - The unsuccessful form is *Failure(x)* where x: Throwable
- As we discussed before, *Try(expression)* is a factory method which evaluates *expression* by name thus being able to catch any exceptions.

Lift, map2, flatMap,
“for comprehensions”

Fasten your seat belts!

Option and Try—in greater depth

- For example, let's look at *Rating* from the *Movie* assignment.

```scala
case class Rating(code: String, age: Option[Int]) {
  override def toString = code + (age match {
    case Some(x) => "-" + x
    case _ => ""
  })
}

object Rating {
  val rRating = """^(\w*)(-(\d\d))?$""".r

  def parse(s: String): Try[Rating] =
    s match {
      case rRating(code, _, age) => Success(apply(code, Try(age.toInt).toOption))
      case _ => Failure(new Exception(s"parse error in Rating: $s"))
    }

}
```

Option and Try (2)

What are these names all about?

- So, we have a method called *parse* which will take a *String* and yield a *Try[Rating]*.
  - Now, we want to add that rating, along with other element(s) to something called *Review:* (simplified)

```
case class Reviews(imdbScore: Double, contentRating: Rating)
val xy = Try("97.5".toDouble)
val ry = Rating.parse("PG-13")
Reviews(xy,ry)
```

  - Oops! we don't have a *Double* and a *Rating*. We have a *Try[Double]* and a *Try[Rating]* instead.
    - So, why not write?

```
val r = Reviews(xy.get,ry.get)
```

Bad idea! Remember, we never want to invoke *get* in these containers

    - In any case, if we do that we essentially lose all the advantage of *Try*. We just simply throw exceptions now if there were failures.
- Wouldn't it be nice if we had a method that took the parameters we actually have and returned a *Try[Reviews]*? Let's write it…

Sidebar: naming identifiers

- Isn't it better if there's a consistent naming convention for the variables which don't have an obvious identifier to use?
  - See http://scalaprof.blogspot.com/2015/12/naming-of-identifiers.html
  - Very briefly, the scheme is that we go in reverse order of the types in the type of the variable.
  - So, a sequence of *X*, such as *List[X]* would be called *xs*. This much is totally standard in Scala. The rest is non-standard: my own scheme:
    - So, *xy* represents a *Try[X]* (we use "t" for a *Tuple*);
    - *xo*: Option[X]
    - *kvm* (or *kVm* or *k_vm* or even *`k,vm`*) is used for a *Map[K,V]* (here, the type parameters of *Map* are not reversed since they're at the same level.
    - etc. You get the idea.

Option and Try (3)

- So, let's try to write the method we need (it's simple stuff)…

```scala
def makeTryReview(xy: Try[Double], ry: Try[Rating]): Try[Reviews] =
  xy match {
    case Success(x) =>
      ry match {
        case Success(r) => Success(Reviews(x,r))
        case Failure(e) => Failure(e)
      }
    case Failure(e) => Failure(e)
  }

val vy = makeTryReview(xy,ry)
```

- That's just what we need! Great…
- Wait a moment! Do we have to write something like this method every time we want to create a *Try[Z]* from a *Try[X]* and a *Try[Y]*??? Aaaaaargh!
- Of course not! Help is on the way.

A better way of dealing with *Options** (1)

- Lift
  - First, wouldn't it be nice if, whenever we had a function *f: A=>B*, we could derive a function *g: Option[A]=>Option[B]* ?
  - That would mean that, whenever we had a function *f* and a variable *ao* of type *Option[A]*, we could do something with it which retained the optional aspect.

    ```
    def lift[A,B](f: A => B): Option[A] => Option[B] = ???
    ```

  - What can we put on the right-hand-side that could possibly make sense? Remember our mantra: *simple, obvious, elegant*

* and other container types

A better way of dealing with *Options** (2)

- So, our lift method should look something like this:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

- Huh? Surely it can't be that simple?? It is that simple!!
- But wait a moment! We've never used "_" *before* a function!
  - We've only used one *after* the function. But the thing that comes *before* the function and the thing that comes *after* are just parameters of the function.
  - So, here, the "_" just means whatever is going to be passed in to the function that we return: in this case, we know it's an *Option[A]*.
- What about lifting a function to a function on *List*, *Try*, or *Seq*?

```
def lift[A,B](f: A => B): List[A] => List[B] = _ map f
def lift[A,B](f: A => B): Try[A] => Try[B] = _ map f
def lift[A,B](f: A => B): Seq[A] => Seq[B] = _ map f
```

- Whoa! Is it really that simple? Yes.

* and other container types

A better way…(2a)

- Incidentally, we could also write *lift* as follows:

```
def lift[A,B](f: A => B): List[A] => List[B] = {a => a map f}
```

  - We will have to use this less elegant form in the following functions…

- Similarly, it would be very convenient if we had a way of combining, say, two *Option* values into one single *Option* value, given a function that can combine the two underlying values. What we need is something like this:

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
    ao match {
      case Some(a) => bo match {
        case Some(b) => Some(f(a,b))
        case _ => None
      }
      case _ => None
    }
```

  - OK, this is nice and general. But for the review, we need *map2* that works with *Try* instead of *Option*.

A better way…(2b)

- Here, we do the exact same thing for Try:

```
def map2[A,B,C](ay: Try[A], by: Try[B])(f: (A, B) => C): Try[C] =
    ay match {
      case Success(a) => by match {
        case Success(b) => Success(f(a,b))
        case Failure(e) => Failure(e)
      }
      case Failure(e) => Failure(e)
    }
```

- Now, we can rewrite *makeTryReview*:

```
def makeTryReview(xy: Try[Double], ry: Try[Rating]): Try[Reviews] =
 map2(xy, ry)(Reviews.apply)
```

- OK, this is nice and general. But can we do better? What do you think *map* and *flatMap* do on an *Option[A]*?

```
def map[B](f: (A) => B): Option[B] = ???
def flatMap[B](f: (A) => Option[B]): Option[B] = ???
```

Please work on this in your peer groups (pairs)

A better way…(2c)

- Continuing with our *map2* on *Option*…

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
    ao match {
      case Some(a) => bo match {
        case Some(b) => Some(f(a,b))
        case _ => None
      }
      case _ => None
    }
```

- Let's write out *map* and *flatMap* as object (non-instance) methods (and with a minor rename in the *map* signature):

```
def map[B,C](bo: Option[B])(f: (B) => C): Option[C] =
  bo match {
    case Some(b) => Some(f(b))
    case _ => None
  }
def flatMap[A,B](ao: Option[A])(f: (A) => Option[B]): Option[B] =
  ao match {
    case Some(a) => f(a)
    case _ => None
  }
```

- Are these looking a little bit similar to *map2*? Kind of…

A better way… (3)

- Let's try the following idea for a method we call "*map2a*":

```
def map2a[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
  ao flatMap (a => bo map (b => f(a, b)))
```

Just made this up!

- Now, let's evaluate *map2a* using our own object-methods and then substituting…

```
def map2a[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
  flatMap(ao)(a => map(bo)(bb => f(a, bb))) => (i.e. can be substituted by)
    ao match {
      case Some(a) => map(bo)(bb => f(a, bb))
      case _ => None
    } => (i.e. can be substituted by)
    ao match {
      case Some(a) => bo match {
        case Some(b) => Some(f(a,b))
        case _ => None
      }
      case _ => None
    }
```

Look familiar???

- Thus we have shown that our *map2* function can be re-written more simply as…

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
  ao flatMap (a => bo map (b => f(a, b)))
```

Whoa!! That's neat.

A better way… (4)

- And an even better way:

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
   for (a <- ao
        b <- bo
    ) yield f(a,b)
```

  - This is called a “for-comprehension” and works for any container type where the container is a monad! It is syntactic sugar for `ao flatMap (a => bo map (b => f(a, b)))`
  - Phew! That was hard. But so important.

Quick summary

- We created a method called *lift* that takes an *A=>B* function and returns a *M[A]=>M[B]* function where *M* is some container type like *Option*, *Try*, *List*, etc.
    - The body of this method is always the same:

```
_ map f
```

- Then we created a method called *map2* that takes, an M[A], an M[B], a function (A,B)=>C and returns an M[C], where *M* is a container as above.
    - The body of this method is always the same:

```
_ flatMap (a => _ map (b => f(a, b)))
```

    - Which we can re-write very nicely as:

```
for (t1: A <- _; t2: B <- _) yield f(t1,t2)
```

“for comprehensions”

- There are two forms of “for comprehension” [we already covered this]:
  - Without *yield* (i.e. relying on side-effect):

    for (*seq* **)** *body*

  - With *yield* (returns value—no side effects):

    for (*seq* **) yield** *expr*

  - In each case, *seq* represents a sequence of *generators*, *definitions* and *filters*, separated by semi-colon (or newline)
    - A generator is of form:

      pattern **<-** *container*

      - where pattern is matched against each item generated from the container (most of the time, the pattern is simply an identifier which matches everything)
    - A definition is of form (exactly like a variable declaration, but without “val”):

      identifier **=** *expr*

    - A filter is of form (just like the guard clause on a match/case pattern):

      if *expr*

Putting it all together

```
object ReadURL {
    import scala.util._
    import scala.io.Source
    import java.net.URL
    def getURLContent(url: String): Try[Iterator[String]] =
      for {
        url <- Try(new URL(url))
        connection <- Try(url.openConnection())
        is <- Try(connection.getInputStream)
        source = Source.fromInputStream(is)
      } yield source.getLines()
    def wget(args: Array[String]): Unit = {
        val maybePages = for {
          arg <- args
           x = getURLContent(arg)
        } yield x
        for {
          Success(p) <- maybePages
          l <- p
        } println(l)
    }
    def main(args: Array[String]): Unit = {
          println(s"web reader: ${args.toList}")
          wget(args)
    }
}
```

Here we are using the real *Try* class in *scala.util*

Instead of *Try[Try[Try[Iterator[String]]]]*, the *Try* classes are collapsed into one—because of the way *flatMap* operates.

From *The Neophyte's Guide to Scala*—this can be improved: we don't close the source for instance.

Note that we can even create the equivalent of a "val" inside a for-comprehension. We can also do filtering, for instance.

This for-comprehension has no yield therefore relies on side-effect

Here's an example of a pattern match

For now, we throw away any error messages.

I ran this with arguments:

- http://htmldog.com/examples/lists0.html
- http://htmldog.com/examples/lists1.html

Some other handy methods:

- What if you had a *Seq[Option[X]]* and you wanted an *Option[Seq[X]]?*
    - *sequence*:

```
def sequence[X](xos: Seq[Option[X]]): Option[Seq[X]] = ???
```

    - this method should iterate through x*os* and, if all elements are *Some(x)*, collect them into a sequence x*s* then return *Some(xs)*. If any of the elements are *None*, return *None*.
    - We’re not quite ready to implement this one.
- What if you had a *Seq[X]* and a function *f: X=>Option[Y]* and you wanted an *Option[Seq[Y]]?*
    - *traverse:*

```
def traverse[X,Y](xs: Seq[X])(f: X=>Option[Y]): Option[Seq[Y]] = ???
```

In general, lots of these functional compositions

- You will be working with some of these in Assignment 4.