Functional Programming in Scala

A quick introduction

Part 1

What is Functional Programming?

What exactly is functional programming?

- Functional programs are *expressions*;
- Parametric expressions encoded as “lambdas”;
- Lazy evaluation;
- Recursion;
- Immutability, pure functions, referential transparency, zero side-effects;
- Pattern-matching.

What are examples of FP languages?

- Lisp, Scheme, Miranda, Clojure, Erlang, OCaml, Haskell, F#
- What about Scala?
  - Scala is a *hybrid* language: it has features of FP, but it also has features of O-O;
  - Some people think this is a good thing (me);
  - Others think it's a terrible thing (FP purists);
  - More on this later.

Functional Programs are expressions

- A Java (or other “imperative” language) program is a series of statements that accomplish some task such as printing out a value or updating a database.
- A functional program is an expression, such as *f(x)*.
- What good is that? What use is a program that just provides some value that has no physical presence?
- Not much good as a main program perhaps, but providing a value to something that can display it/store it/print it, whatever is just fine!
- And, in any case, FP languages allow for I/O, but with stricter rules.

Parametric expressions encoded as lambdas

- “Lambdas” are parametric expressions which ”close” on free variables.
- For example:

  val *y* = 2
  val *f1*: Double=>String = x => (x/*y*).toString

  - In this example, the lambda is *x => (x/y).toString* and this is treated as a function of *x* (the “bound” variable) which can be passed around a program as an object. Because the lambda is a “closure,” it has the value of 2 for *y* (the “free” variable), wherever it is used.
- Lambdas were introduced by Alonzo Church in 1936 in the ”Lambda Calculus.”

Lambdas (2)

- Once you can define and reference lambdas as functions, you can *compose* them!
- Just like you can compose values (with operators such as +, /, etc.), you can also compose functions (with operators such as *compose* or *andThen**).
- Example of composition (Scala):

  val h: Int=>Double = (x => x/2.) andThen (x => x+1)

  - Where *h* is a function such that *h(x) = 1+x/2*.
- **And why is this useful**? Because, using something like Spark, invoking a function on all the partitions of a dataset over all the remote executors could be very expensive—but if we can compose two functions into one, we've halved the problem!

* Those are the Scala names—they have different names in other languages

Lazy evaluation

- ”Lazy” is good (in a program, not in a programmer);
- Lazy is also known as “non-strict” or delayed/deferred evaluation:
  - A strict parameter or a strict variable is one which is always evaluated (when it is passed in, or when it is declared);
  - A non-strict (lazy) parameter/variable is one which is only evaluated if and when required.
- This may not seem like a big deal—but it is. It has the potential for huge improvements in performance;
- A lazy list is called a “stream” and it is possible to define a stream of *all* the positive integers, for example. We can use that to filter out all non-primes, for instance, and yield the list of all primes. Of course, we can’t evaluate the whole list but we can evaluate as many as we actually need.

Lazy evaluation (2)

- But laziness is much more important than just for building streams.
- Examples:
  - when logging to debug, for example, the parameter which builds the string to be logged can be passed in lazily (“call by name”): it’s never evaluated if it’s not needed (i.e. debug logging is off);
  - when passing an expression *x* to *Try.apply(x)*, the *x* is evaluated <u>inside</u> the *apply* method and so any exception can be caught there and turned into a *Failure*.
  - In Spark, for example, an *RDD* (and therefore a *Dataframe*/*Dataset*) is lazy:

    ```
    rdd.map(f).map(g).collect()
    ```
  - is equivalent to:

    ```
    rdd.map(f andThen g).collect()
    ```
  - In other words, Spark can compose the functions *f* and *g* and visit all of the elements of the *RDD* only once!

Recursion

- In an imperative language such as Java, the primary mechanism for doing things many times is *iteration*.
- In FP, the primary mechanism is *recursion*.
- Does it matter? Well, it often doesn't matter a lot. Other times it can make a big difference. You can't do iteration without mutable variables (think about how you might sum the elements of a list).
- Recursion is typically a more mathematical way of expressing some sort of aggregate function:

```
    def sum(xs: Seq[Int]): Int = if (xs.isEmpty) 0 else xs.head +
sum(xs.tail)
```

Recursion (2)

- But isn't recursion a BAD thing?
- Recursion is like iteration but, since you are not mutating a variable each time around, you have *history*. This history is the breadcrumbs you need to find your way out of the recursion.
- Trouble is that this history takes up space: on the very limited system stack*.
- But, for many applications (most, actually), you don't really need the history and in this case you can make your recursion *tail*-recursive:

```
def factorial(n: Int) = {
    def inner(r: Long, n: Int): Long =
            if (n <= 1) r
            else inner(n * r, n - 1)
    inner(1L, n)
}
```

- So, in practice, FP uses tail-recursion wherever it can and the problems of stack overflow do not occur.

* And when you exhaust the stack, you suffer a StackOverflow ☹️

Immutability, pure functions, RT, zero side-effects

- A cornerstone of functional programming is that when you write *f(x)*, you expect the result always to be the same, assuming *x* is the same.
    - But if *f* involved some other variable *y* that was free to mutate, then the result wouldn't always be the same.
- Functions which behave this way (pure) functions are so predictable that you can *prove* functional programs to be correct. Yes, you read that right! Imperative programs cannot be *proven*. They can only be *tested*.
- So, pure functional programs do not allow variables to mutate or cause side-effects.
- And, if you have `def f(x) = x + 1` (for example), the program is the *same* whether you write *f(x)* or *x + 1* in some expression ("substitution principle").
- This concept is also known as referential transparency.

Pattern-matching

- Pattern-matching is another big difference between functional and non-functional code:
  - You can match on constant- or variable- values;
  - You can match on types (so you don't need to dynamically cast anything);
  - You can match on the de-struction of objects (the opposite of a constructor—called an extractor).
- In functional programming, pattern-matching is like a powerful, generalized, and glorified *switch* (or *if*) statement.
- Pattern-matching is like the opposite of assignment:
  - Instead of taking an expression and assigning it to an identifier…
  - Pattern-matching allows you to take an identifier and match it as an expression.

Pattern-matching: Scala examples

- Getting the contents of an optional value:

```
Option(maybeX) match {
    case Some(x) => println(x)
    case None =>
}
```

- Remember that *sum* method from before? How about this?

```
def sum(xs: Seq[Int]): Int = xs match {
    case Nil => 0
    case h :: t => h + sum(t)
}
```

  - *Nil* is simply the name of the empty list;
  - *h :: t* is a pattern* that says match *xs* as the head element *h* followed by the tail *t* (i.e. the rest of the list).

* In fact, :: is actually a case class (yeah, really) and its *unapply* method is used

Part 2

Scala and Functional Programming

Scala and functional programming

- There are five key features of Scala:
  - Functional Programming;
  - Object-oriented;
  - Static types;
  - Java virtual machine;
  - Implicits.

- We already discussed Functional Programming.

Object-oriented

- Type hierarchy—basically the same as for Java;
- Polymorphism, encapsulation, all that good stuff;
- Classes, traits, objects.
  - traits instead of Interface (traits can have default implementations, similar to what's now in Java8);
  - "objects" in Scala are singleton classes (similar to marking stuff static in Java);
  - Also "case" classes:
    - Provide all of the standard boiler-plate code, including *unapply* for pattern-matching;
- There are no "primitives" in Scala (at least *you* don't have to be aware of them);
- The Class Loader (doesn't distinguish between Scala and Java code);
- The ubiquitous "dot" operator.

Static Types

- Parametric (as opposed to generic) types
  - Liskov substitution principle (defines what sub- and super-types mean);
  - Full treatment of variance (where *S* is a sub-type of *T*):
    - Invariant: *Array[T]* is neither subtype nor supertype of *Array[S]*
    - Covariant: *List[S]* is a subtype of *List[T]*
    - Contravariant: *T* => *U* is a subtype of *S* => *U*
  - This allows us always to be very strict about types (unlike Java or Python, for example)
- Type classes and other type constructors
  - T[X]
- **Once you have succeeded in compiling a Scala program, it usually does what you want (run-time errors are unusual).**

Java Virtual Machine

- Scala wouldn’t be a serious contender in the language space if it didn’t run on the JVM.
- Running on the JVM gives access to thousands of Java libraries, especially important in the early days of Scala.
- The JVM, lazy evaluation, functions, class loader, etc. are what powers Spark. There’s nothing magic in Spark.

Implicits

- One of the big problem areas in any language is dealing with (configuring, searching) “global” variables;
- Implicits not only provide a solution to these global variables, but they give us many other benefits (e.g. “extension” methods);
- Type classes;
- Implicit classes and conversions (instead of just “widening” as in Java).

What’s different from other FP languages?

- Scala does allow *var* (mutable variable) and mutable collections:
    - but you are discouraged from using them! And you really don’t need them!
- Scala has similar ways of defining new types (*Option*, *Either*, etc.) as well as “type constructors” and lots of arcane stuff that you will never actually come across–however,
    - Scala also allows you to define new types by inheritance (just like O-O)
- Scala doesn’t actually have a monadic type—
    - if you really want that, you have to use one of the libraries such as ScalaZ

What’s different from Java?

- Scala avoids *nulls* and exceptions via built-in types:
  - *Option[X]* which has two cases: *Some(x)* and *None*;
  - *Try[X]* which has two cases: *Success(x)* and *Failure(e)*;
- Other “union” types like *Either[L,R];*
- *Future[X]* is used to handle asynchronous results
  - different from Java’s *Future*.
- Scala doesn’t have primitives like *int*, *double;*
- Collections are different;
- Function types are much more general than Java8’s function types:
  - multiple parameters easy to define;
  - “curried” functions for example;
- Other types, e.g. *String*, *Array*, *MyClass*, are the <u>same</u>;
- Traits can have default methods (Java8 introduced this).

What’s different from Java (continued)?

- You can define related types (trait, classes, etc.) in one module
  - i.e. without having to make private inner classes;
- There are no static (“class”) methods: all singletons are ”objects”;
- In Java, you can’t return more than one value from a method unless you go to the trouble of defining a class—in Scala you just return a tuple.
- In Java, generics are kind of a mess (they were an after-thought).
  - You can cheat, or you can just make everything a “?”.
  - In Scala, parametric types are very strictly enforced. So, you can’t be surprised at run-time by a value that doesn’t conform to the proper type.

Part 3

How to write good Scala code

Making the transition from Java to Scala (1)

- I would estimate that 99.9% of Scala programmers were (or still are) Java programmers;
- There are some things that need getting used to:
  - Perhaps the number one thing is the use of *var*. Many new Scala programmers don't know how to do things without mutable variables and collections. That's not surprising. It takes training and a knowledge of FP to be able to avoid *var*s.
  - Immutable collections: Scala collections are immutable by default (you have to import *mutable.xxx* in order to get the mutating version). This takes a little getting used to and causes confusion about the operators such as ++, +=, etc.
  - Equality: Java has primitives (int, double, etc.) while Scala doesn't. Therefore, Java provides two methods for testing equality: "==" for primitives and "equals" for objects. In Scala, the "==" method simply delegates to Java's *equals* method. If you want to test that two objects are the same object in Scala (rare), you use *eq*.

Making the transition from Java to Scala (2)

- More things that need getting used to:
  - Java loves to wrap everything in {} and separate statements with ";" —in Scala, you can eliminate most of this clutter.
  - In Java, you're allowed multiple returns from a method—in FP, this is very much frowned on. You don't need the *return* keyword in Scala—the final expression in a method or block is what is returned: if you ever find yourself using *return*, ask yourself why.
  - In Java8, functions are all defined as lambdas—but in Scala you can take advantage of the so-called *eta* expansion—that's to say that wherever Scala expects a function, you can provide a method (it's much clearer what's going on when you define your function as a method—for one thing a method has a name).
  - Defining classes in Scala: it's rare to define a straightforward class in Scala: a class is either an abstract class or a case class. You can do it, but there's no good reason to.

Making the transition from Java to Scala (3)

- Yet more things that need getting used to:
  - Class constructors: in Java it's common practice to provide several different constructors for a class. In Scala, you can do this too (syntax is different), but it's better practice to code these as additional "apply" methods in the companion object.
  - Case classes: Get used to using these for just about everything. If you just need a temporary way to combine values (say for the return from a method), you can use a tuple (you don't have to name the fields or given them types). But be aware that case classes are also tuples.
  - Modules: a single file in Scala can have any number of related classes (don't abuse this, though). Typically, you will have one (sealed) trait and a number of case classes which extend that trait. Each of these case classes may also have a companion object.

Making the transition from Java to Scala (4)

- Even more things that need getting used to:
  - In Java, it can be a pain to initialize a list. Not so in Scala. You can just write List(1,2,3)
  - How about adding an element to a list? In Java, you typically do that with *add* (a mutating method). In Scala, you do it immutably:

```
val xs1 = List(1)
val xs2 = xs1 :+ 2
```

    - Note that the : is always on the side of the collection.

```
val xs2 = 2 +: xs1
```

How to write good Scala code (1)

- Perhaps the first difference you notice from Java is the syntax:
  - No semi-colons (unless you need them to fit stuff on one line);
  - Fewer braces and parentheses (no-arg methods usually don't need parentheses);
- and built-in semantics:
  - Case classes, for instance, provide all the getters (possibly setters, too), *equals*, *hashCode*, *toString*, all that jazz—and the definition is really the constructor—and you don't need "new".
- In short, Scala code is a lot more compact than the equivalent Java code.
- So, try to use these improvements to the look-and-feel of the code to write really elegant, clear code.
- Have each class know about one domain and put all the methods that use that knowledge in the class—and have each method do only one simple thing.

The principle of Simple, Obvious, Elegant

- Most of the time when writing, say, a method in FP, you have a very limited set of variables which are in scope and, because each has some particular type, they can only be combined together in a small number of ways:
  - Don’t be afraid to follow where your instinct is leading you.
  - Let the IDE guide you by using ctrl-space (or whatever): it will show you the possible expansions, with the types that they result in.

Getting the help you need

- Your first place to go for anything is: https://www.scala-lang.org/
- The definitive book is *Programming in Scala* (3rd edition) by Odersky et al.
- But the book that will really teach you what's going on is *Functional Programming in Scala* by Rúnar Bjarnesson and Paul Chiusano (the "red book"). It's fun to read but not for the faint of heart.
- The Lightbend site is also good (more application-oriented).
- Of course, StackOverflow is great for all Scala questions.
- There are some great blogs out there too:
  - The Scala Times
  - The Neophyte's guide to Scala
  - (my) Scalaprof blog