Updated: 2021-09-12

2.3 Functional Programming in Scala

Northeastern University

Functional Programming in Scala

- This lesson is long but it contains most of the features which (in my opinion) make functional programming (and Scala in particular) work so well.
- You should take good notes on all this.
- There will be some repetition of a couple of features I've already mentioned briefly.

What exactly is functional programming? (repeat)

- Functional programs are *expressions*;
- Higher-order functions and functional composition;
- Parametric expressions (function “literals”) encoded as “lambdas”;
- Lazy evaluation;
- Recursion as the primary re-use mechanism;
- Immutability, pure functions, referential transparency, zero side-effects;
- Pattern-matching;
- Type inference.

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 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.

Lambda calculus

- In the 1930s, Alonzo Church developed a notation and a set of simple rules that governed the way computable expressions could be defined.
- In particular, the way he defined the abstraction of a function (see last two slides) was <u>anonymous</u>.
- Thus, we use the terms *lambda* and *anonymous function* as synonyms (mean the same).

Parametric expressions encoded as lambdas

- Church’s lambda calculus does not take into account the types (or domains) of the variables and terms. But Scala is strictly typed so we must specify the types. And, if we need to refer to a function, we can give it a name.
- 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) and 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. And, you can see from the type of the **val** part, that *x* is of type *Double* and the result is of type *String*.

* I added extra space around the = of f1 just to clearly separate the two sides.

Composing functions

- 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 each function on all the partitions of a dataset over all the remote executors can 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;
- In Scala, a lazy list is called a “LazyList” (from 2.13*), and it is possible to define a *LazyList* 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*.
- Java 11 has some lazy features (via Streams) but not as much flexibility as Scala.

* previously, a lazy list was called a *Stream*.

Lazy evaluation (2)

- But laziness is much more important than just for building lazy lists.
- 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!

* We will explain *Try, apply, Failure, collect* later

Recursion

- In an "imperative" language such as Java, the primary mechanism for doing things many times (i.e. re-using code) is *iteration*.
- In FP, the primary mechanism is *recursion*.
- Does it matter? Well, it often doesn’t matter much. 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)
```

* *Head* and *tail* refer to the first element of the list, and the rest.

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 ☹️

Recursion (3)

- There’s another advantage of recursion: when you *reduce* a problem into a set of easier problems, one of the ways to do that is simply to arrange for each sub-problem to operate on a subset. This is known as “divide and conquer.”
- There are two obvious ways to do this for a set (collection) of size *N*:
    A. Make the sets of size 1 and *N*-1;
    B. Make the sets of size *N/k* where *k* is some integer, usually 2.
- The first strategy (*A*) naturally gives rise to iteration; the second (*B*) naturally gives rise to recursion.
- Algorithms which use *A* tend to be ***O****(N)*; algorithms that use *B* tend to be ***O****(log N)*.

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.

* By imperative, we mean the paradigm where the programmer instructs the compiler *how* to perform a task.

Immutability (2)

- So, what exactly do we mean my immutability?
  - When is a variable or a collection mutable and when is it immutable?
- For variables, it’s easy: the variable must be declared as **var** and we will see two or more *assignments*:
  - `var x = 1`
  - `x = 2`
- For collections, it’s a bit harder to distinguish:
  - First, a mutable collection is instantiated from the *scala.collection.mutable* package, for instance:
  - `val xs = mutable.Map(1 -> "A", 2 -> "B", 3 -> "C")`
  - But, notice that *xs* is still typically a *val*, not a *var*, because it’s the collection itself that can be mutated, not the reference to the collection.

Immutability (3)

- What are the mutating operators on, say, *Map*?
  - `xs += 4 -> "D"`
- So, how would we add an element to an immutable *Map*?
  - `val map3 = Map(1 -> "A", 2 -> "B", 3 -> "C")`
  - `map3 += 4 -> "D”` ❌
  - This results in a compiler error. Instead, we write something like this:
  - `val map4 = map3 + (4 -> "D")`
  - Now, we have a new identifier *map4* which has four key-value pairs, while *map3* still has only three.

Immutability (4)

- What about a *LazyList*? Are the head and tail mutable? Because they do appear to change value at some later time than their construction.
- No, lazy values are not mutable, just lazy (deferred evaluation). It’s just that, when first declared, a lazy value has no value at all. Once a lazy value gets evaluated, it never changes.
- What about things like recursive methods?
  - `def factorial(x: Int):Int = if (x<=1) 1 else x * factorial(x-1)`
- There’s nothing here that’s mutable. Each time *factorial* is invoked, it has a new, immutable, value of *x*.

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, you can think of pattern-matching like a powerful, generalized, and glorified *switch* (or *if*) statement.
- But, another way to think about it is that Pattern-matching is the opposite of assignment (i.e. variable declaration).
  - 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 (Scala has a special type for this-- Java has it now, also):

```
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 here.

Type inference

- This might just seem like a convenience to the programmer so you don’t have to explicitly mention the type of a variable or method.
- But, actually, it’s much more than that.
- If the type of the variable or method is known, type inference can be used by the compiler to determine which of several methods or variables can form part of the expression on the right of the = sign.
- And, it actually goes further than that (we will eventually cover that).

Lambda calculus

- In the 1930s, Alonzo Church developed a notation and a set of simple rules that governed the way computable expressions could be defined. He called it the lambda calculus because he used the Greek symbol ƛ. See https://en.wikipedia.org/wiki/Lambda_calculus.
- The three rules:
  - If *x* is a variable then *x* is a valid lambda term;
  - If *t* is a lambda term, and *x* is a variable, then *ƛt.x* is a valid lambda term (an abstraction);
  - If *t* and *s* are lambda terms, then *(t s)* is a valid lambda term (an application).

* I added extra space around the = of f1 just to clearly separate the two sides.

Lambda calculus continued

- Abstractions:
  - $\lambda x.x^2$ represents a function which takes an input (here denoted by *x*, but we could use any symbol we like, except λ) and which yields the square of the input. We consider *x* to be <u>bound</u> in the expression x^2. In other words, we have no choice about *x*, it is provided for us by the input to the function.
  - $\lambda x.xy$ represents a function which takes an input *x* and which yields the product of *x* and *y*. Since *y* could take on any value as far as our function is concerned, we say that *y* is <u>free</u>.
- Applications:
  - *t s* represents the application of function *t* to input *s*, that's to say we pass *s* into *t*.

Summary

- Functions are just objects, like 1, “abc,” math.Pi, etc.
- They can be found as parameters (or results) of another function (or method) and they can be composed into expressions using “higher-order” functions.
- You do not have to remember anything of the previous two slides—those are just for a little more background.