Lecture 18: Scala

December 4th, 2018

Topics: Scala, introduction to recursion.

1 Language Paradigms

Programming languages can actually be classified into paradigms based on the features they provide. It's possible for languages to fit into multiple paradigms, as we will emphasize below. See: comparison of paradigms.

1.1 Imperative: "an authoritative command"

Statements and Expressions An imperative program executes *statements*, which themselves can be composed of *expressions* to be evaluated. A *simple* statement is something like an assignment, function call, or return, whereas a *compound* statement could be a control-flow structure such as an `if`, or a `for`-loop. Note that in contrast to an expression, a statement doesn't return a result (it's simply executed for its side-effect); on the other hand, an expression *always* returns a result and often doesn't have side-effects.

Computer Architecture lends itself to being directed by an Imperative Language We've mentioned in this class that fundamental operations in a computer may involve transferring data from RAM into Cache and ultimately processor registers so that data may be computed on by the ALU. Low-level assembler languages are imperative in nature: they execute *statements* such as moving data from one location to another or adding data from two registers together.

Development of higher-level languages, and ultimately OOP Although machine-code lends itself well to instructing a computer, its not amenable to abstraction and creating complex programs. In the 50's, FORTRAN was invented at IBM: it featured named variables and sub-routines, which are now considered essential features of an imperative language. In the 80's, there started to emerge a growing interest in object-oriented programming (C++ played a fundamental role in this), wherein instructions are grouped with the data they operate on.

1.2 Declarative: "not imperative; referentially transparent"

Totally distinct from an imperative paradigm, in a declarative language the programmer simply specifies properties of the desired result but *not* how to compute it. SQL and HTML programs are declarative in flavor; other familiar examples include `CVX`, wherein the program specifies a set of (in)-equalities we wish to satisfy, and the program returns a set of variables consistent with the system.

Functional A subset of declarative programming includes the functional paradigm. A famous early example is Lisp.[1] In a functional language, we specify computations via composition of (mathematically pure, i.e. referentially transparent) functions and we forbid mutation of data. I.e. there shall be *no statements* (which can depend on the state of local/global variables) but instead there shall *only* be side-effect free functions: the result of a function shall depend only on its arguments and no other variables; further, variables shall not be mutable, since this introduces a notion of state.

Scala The name is a portmanteau of Scalable Language, and its designed to be extensible to grow and support the changing needs of its users. Its design spawned in 2001 by Martin Odersky out of EPFL. There are other languages which are more purely functional, such as Haskell, however, Scala is a bit more friendly to learn as it also supports object oriented programming in addition to functional programming. It's also used to build Apache Spark, a powerful distributed computing system used widely in industry. In general, functional programming languages lend themselves well to parallel programming since expressions are independent from each other (since they do not depend on external state and have no side-effects) whence they may be evaluated in parallel. Scala is statically typed which makes it safer to build and reason about the correctness of larger programs.

1.3 A Multi-Paradigm Reality

We mentioned in the introduction that languages often support multiple-paradigms. This is certainly true: for example, a C++ program can be written in a procedural style (without embracing OOP), and it can also be written in a functional style. Some languages lend themselves more to certain paradigms than others, and other languages are quite restrictive in prohibiting multiple paradigms (e.g. Haskell will *not* support any notion of procedural programming). It's important to remember that most languages (Python, C++, R) support multiple paradigms, and to not let ourselves get too distracted over the debate of which is better. After all, Fortran and Lisp are distinctly different (imperative vs. declarative), ancient, and still well in use today; so clearly both paradigms contain merit.

[1] Lisp is the second oldest high-level programming language (this title goes to Fortran, one year older); its name derives from "LISt Processor" and linked-lists are the fundamental data structure.

2 Scala

We follow closely from Martin Odersky's text, "Programming in Scala". Within the learning resources listed on `Scala-lang.org`, the first-edition is linked for free (legal) online reading.

2.1 Characteristics of Scala

Scala is Object-Oriented Everything value is an object, and every operation is actually a method call. I.e. `1 + 2` invokes the method named `+` defined in class `Int`.

Scala is Functional There are two main tenants behind functional programming.

1. Functions are *first-class values*: we can pass functions as arguments (like any other type), and we can define functions within functions just like we can define local objects.
2. Operations should map input values to output values, as opposed to mutating data in place; i.e. functions shall have *no side-effects*. Equivalently, for any particular input, an invocation of a *referentially transparent* function can be replaced by its corresponding result without changing the behavior of the program.

In general, functional languages *encourage* immutable data structures and referential transparency. Scala is amenable in that it also allows imperative code. Properties of Scala:

1. Built off the Java Virtual Machine (JVM), which effectively means we can interface Scala with any Java program; types in Scala are "dressed up" analogues from Java.[2]
2. Concise, high-level language that allows us to manage complexity through abstraction.
3. Statically typed, which means verifying properties of a program's abstractions is easier. Concision (above) mitigates usual burdens of a statically typed language.
   (a) *Verifiable Properties:* the compiler can ensure things like booleans aren't added to integers, or that private variables are never accessed from outside their class.
   (b) *Safe Refactoring:* if we for example add an argument to a method and recompile our program, the compiler will instantly alert to us (via errors) of every invocation that must be amended. After this act, all relevant code has been changed.
   (c) *Documentation:* declaring the type of each object can be viewed as a form of documentation, which is in fact checked by the compiler for correctness. In contrast with a comment which cannot be checked by a compiler, a type annotation never "goes stale" and is always enforced.

Installation Instructions can be found here: `https://www.scala-lang.org/download/`.

[2] A JVM exists to mitigate issues of portability. Instead of writing C++ code that gets translated/compiled into machine code directly, thus rendering the program tied to a particular architecture, Java code is first byte-compiled where it can be run on a JVM, wherein the JVM takes care of finally translating the byte-code into instructions for the particular architecture it resides on. The idea is "write once, run anywere".

2.2 First Steps

Although Scala is a compiled language, it supports a REPL interpreter wherein each line of input is compiled separately and automatically via JIT. To launch the interpreter, simply type `scala` from a shell.

```
$ scala
Welcome to Scala version ...

scala>
```

We're now able to, for example evaluate the expression `1 + 2`.

```
scala> 1 + 2
res0: Int = 3
```

The interpreter informs us that we've implicitly defined a name `res0`, itself an `Int` which takes on the value 3. We mentioned before that Scala is object-oriented. To see this explicitly, realize that we can almost always use binary operators either via infix notation or as methods.

```
scala> 1.‘+‘(2)
res1: Int = 3
```

Note that in the code above, the back-ticks are not being displayed correctly, and we emphasize that the `+` operator is being enclosed by back-ticks and *not* single-quotation markers.

2.3 Variables and Values

Scala supports immutable and mutable data types, via `val` and `var` respectively. We must specify the type of variable (`val` or `var`), but it's fun to realize that Scala can infer the type of a variable automatically.

```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```

I.e. the immutable variable `msg` is a `String` object.[3] We could more explicitly declare and initialize an object as follows:

```
scala> val msg1: String = "Another string"
msg1: String = Another string
scala> val x: Int = 42
x: Int = 42
```

Once initialized, a `val` can never be reassigned or modified throughout its lifetime, whereas a `var` can be modified. Attempting to reassign to a `val` yields an error.

[3]Note that from Scala, the `java.lang` namespace is visible, so we can drop this prefix if we'd like.

2.4 Functions

Let's start with a simple example.

```
scala> def max(x: Int, y: Int): Int = {
  if (x > y) x else y
}
max: (Int,Int)Int
```

Note that Scala can also deduce the return type of a non-recursive function. Further, we can omit braces if we have a simple statement as the sole contents of our function definition.

```
scala> def max2(x: Int, y: Int) = if (x>y) x else y
max2(Int,Int)Int
```

2.4.1 Function Literals and Composition

Since functions are first class objects, we're allowed to create what are known as (unnamed) function literals. Just like the value 42 can appear anywhere in our code, unnamed, we can also drop in unnamed functions "on the fly". E.g. a simple addition function literal:

```
(x: Int, y: Int) => x + y
```

We are allowed to drop a function literal in any place where a function is expected to appear.

```
val vec = (1 to 10).toList                    // A list of 10 integers.
vec.reduce( (x: Int, y: Int) => x + y )       // Returns 55, as expected.
```

Is equivalent to:

```
def add(x: Int, y: Int): Int = x + y
vec.reduce(add)
```

In fact, as syntactic sugar Scala even lets us use a *placeholder* syntax via _ (which can be used in many different, complex ways in Scala) so that we can re-write our vector summation:

```
vec.reduce(_+_)
```

Another nice functional in which placeholder syntax appears commonly are `filters`.

```
-5.to(5).filter(_ % 2 == 0)
```

2.4.2 Foray into Basic Functional Utils

This is honestly pretty cool. If we want to perform element-wise addition on two Lists, we may start with a `zip` method to place corresponding elements next to each other in a List:

```
scala> vec zip vec
res5: List[(Int, Int)] = List( (1,1), (2,2), ..., (10,10) )
```

Once each corresponding element has been placed next to each other, we may apply the addition operator across each element (itself a tuple of Ints) in the list, via `map` method:

```
def ListAdd(x: List[Int], y: List[Int]): List[Int] = {
  (x zip y).map{ case(x,y) => x+y }
}
```

And we can try adding `vec` to itself.

```
scala> Add(vec, vec)
res4: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Or, for example we could write a succinct inner-product operator, this time showing off the `reduce` method, which takes a List and a associative binary operator and *reduces* the collection into (in this case) a scalar value. [4]

```
def Dot(x: List[Int], y: List[Int]): Int = {
  (x zip y).map{ case(a,b) => a*b }.reduce(_+_)
}
```

2.5 Classes in Scala

We've mentioned that Scala is object-oriented. Defining a class is intuitive and concise; see classes in Scala.

```
class Point(var x: Int, var y: Int) {    // Constructor.
  def move(dx: Int, dy: Int): Unit = {  // Move method. Eff. returns void.
    x = x + dx
    y = y + dy
  }
  override def toString: String = {      // Already exists a generic toString...
    s"($x, $y)"
  }
}
```

Members are public by default, but of course there is a `private` access specifier to override this behavior.

2.6 Iteration in Scala

Scala does support `while` loops. However, we should hopefully recognize that this is an imperative style of programming. Instead of a traditional `for`-loop *statement*, Scala provides

[4] After working on our final project in C++, we can perhaps appreciate how succinct this one-liner utility is, when compared with e.g. a `for`-loop and a (mutable) variable to track the running sum. C++ also supports functionals like `std::transform` and `std::reduce`.

`for`-expressions (or `for`-comprehensions) which actually *return* a sequence of values. We can use `yield` to return a sequence of values, and we can add (semi-colon separated) filters.

```
val fizzbuzz = for (i <- 1 to 100 if i % 3 == 0 || i % 5 == 0) yield i
```

Nested iteration is as simple as using multiple generators (signified by `<-`).

```
for (i <- 1 to 3; j <- 11 to 13) yield i*j
res5: scala.collection.immutable.IndexedSeq[Int] =
  Vector(11, 12, 13, 22, 24, 26, 33, 36, 39)
```

2.7 Lists and Recursion

Lists in Scala are proper linked lists, and all operations on Lists can be described using the following expressions:

- `head`: returns the first element of a list.
- `tail`: returns a list consisting of all elements but the first.
- `isEmpty`: returns `true` if the list is empty.

Insertion Sort Suppose we have a list of numeric elements, and we wish to sort into ascending order. Realize that if we have a list of elements, it suffices to simply take the first element and insert it appropriately into an already sorted list of the remaining elements. This recursive implementation makes sense because we have broken a larger problem into more manageable and smaller sub-problems: rather than `sort`ing a list of length n, we instead are tasked with inserting a single element into an already sorted list of length $n-1$. Further, this second task of inserting an element into a sorted list can be broken down into two simpler sub-problems: either we may simply prepend the element into the sorted list (since it is smaller than the leading element, and the list is sorted), *or* we can realize that our element belongs somewhere after the `head` of our sorted list. This yields the following algorithm(s):

```
def isort(xs: List[Int]): List[Int] =
    if (xs.isEmpty) Nil
    else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
    if (xs.isEmpty || x <= xs.head) x :: xs
    else xs.head :: insert(x, xs.tail)
```

We've used the `::` method to concatenate lists. Compare the above implementation with an analogue implementation from C; ours is much more concise! For an input sequence of n items, we'll always end up calling `isort(xs.tail)` which means we invoke our recursive function $\Theta(n)$ times. In the best case, our input is already sorted an we may always fall into the first predicate in the sole-expression of `insert`. But in the worst case, `insert` may

require comparing the head element with all $O(n)$ elements in the remaining sub-list. I.e. our best-case work required is $\Omega(n)$, but our worst case is $O(n^2)$.

Divide and Conquer, Pattern Matching An important paradigm in recursive programming is that of divide and conquer. In Scala, we may do this via pattern matching: we first divide an input problem into components (sub-problems) and then recursively solve each in-turn. They key is that the new sub-problems we've created are easier to solve when compared with our original. Consider appending two Lists together.

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

Let's try another simple example, which demonstrates that calling `length` method on `Lists` is actually an expensive operation.

```
def Length[T](xs: List[T]): Int = xs match {
  case List() => 0
  case hd :: tl => 1 + Length(tl)
}
```

I.e. testing `l.length == 0` and `l.isEmpty()` are equivalent, but the latter is a constant time operation whereas the former requires work proportional to the number of items being stored.

2.8 Currying

A functional programming technique, currying, which is the idea of transforming a single function accepting multiple arguments into an equivalent sequence of functions each accepting a single argument.

Curried Sum Consider a vanilla definition for summation.

```
def PlainSum(x: Int, y: Int): Int = x + y
```

We can analogously define a curried function, which instead of accepting a single list of two input arguments, instead accepts two lists of singleton input arguments.

```
def curriedSum(x: Int)(y: Int) = x + y
```

We could then invoke it via `curriedSum(1)(2)` to yield 3. A related idea is that of partial function application, wherein we supply some (but not all) required function arguments, returning a new function which accepts an argument list of smaller arity; e.g. `curriedSum(5)_`.

Writing New Control Structures We can use currying to define our own (arbitrary) control structures. Consider defining a functional which repeats an operation twice and returns the result.

```
def twice(op: Double => Double, x: Double) = op(op(x))
twice(_ + 1, 5)  // Returns Double = 7.0
```

Here, the type of `op` is given by `Double => Double`, indicating that the `op` is a function which accepts a single `Double` as argument and returns a `Double` as output. The second argument to our functional is given an alias `x` and is itself another `Double`.

Revisiting Sorting Routines: Merge Sort Above, we implemented Insertion Sort. It was concise, but in the worst-case it requires work that is proportional to the square of its input sequence. The pitfall lay in the fact that the `Insertion` step required up to $O(n)$ work depending on how the standalone element compared to the remaining sub-list. Another idea is to break our input list into two equally sized sub-pieces, sort each, and merge the sorted-lists together (quickly).

```
def msort[T](less: (T, T) => Boolean)
  (xs: List[T]): List[T] = {
    def merge(xs: List[T], ys: List[T]): List[T] =
      (xs, ys) match {
        case (Nil, _) => ys
        case (_, Nil) => xs
        case (x :: xs1, y :: ys1) =>
          if (less(x, y)) x :: merge(xs1, ys)
          else            y :: merge(xs, ys1)
      }
    val n = xs.length / 2
    if (n == 0) xs
    else {
      val (ys, zs) = xs splitAt n
      merge(msort(less)(ys), msort(less)(zs))
    }
  }
```

You'll notice that we wrote a curried version of `msort` which is agnostic to what comparator we supply as argument. The `splitAt` method simply partitions a List into two-sublists at the location specified by input argument. With some thought, we can realize that our `merge` sub-routine which merges two sorted sub-lists requires $\Theta(n)$ work since in each invocation we make a constant time comparison and reduce our problem size by a single element. Then, realize that if each invocation of `msort` creates two sub-problems each half as large, that in solving for $\left(\frac{n}{2}\right)^k = 1$ (to see how many times we must half our input size before reaching our base case of having only one element left, which is trivially sorted against itself) we see we require $k = \log_2(n)$ recursive calls. Since each recursive call invokes a call to `merge` (itself costing $\Theta(n)$) work, we see the total work required is $\Theta(n \log n)$.

An example invocation of our function is given by

```
msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
```

where we first provide the comparator (in this case a function literal) and then we provide an input list to sort. And of course we could define a reverse-sort function quite easily.

```
val ReverseSort = msort( (x: Int, y: Int) => x > y) _
```

Here, we've used the `_` to represent a missing *argument list*.

2.9 Monoids

What's an introduction to functional programming without a mention of *monoids*. The term comes from Abstract algebra, and this can be a really fun branch of mathematics to learn about! A monoid is an algebraic structure which has a domain (or set of elements), a single associative binary operator, and an identity element.

Formal Definition If S is a set, $\circ$ is an associative binary operator with domain S, i.e. a mapping $S \times S \to S$, and we are given identity element e, then the triplet $(S, \circ, e)$ forms a monoid. By definition, such a structure satisfies the following axioms:

- *Associativity*: I.e. for any $a, b, c \in S$ we have that
$$(a \circ b) \circ c = a \circ (b \circ c).$$
- *Identity Element*: There exists an element $e \in S$ such that for any $a \in S$,
$$e \circ a = a \circ e = a.$$

How can we link this back to something familiar? See abstract algebra's basic concepts.

$$\texttt{Sets} \xrightarrow{\text{binary op}} \texttt{Magmas} \xrightarrow{\text{associativity}} \texttt{Semigroup} \xrightarrow[\text{inverse}]{\text{identity elem,}} \texttt{Group} \xrightarrow{\text{> 1 operation...}} \left\{ \begin{array}{c} \texttt{ring,} \\ \texttt{field,} \\ \texttt{vector space} \end{array} \right\}.$$

Familiar Examples See a listing of (somewhat familiar) examples. Some of the easiest to intuit are:

- The set of naturals $\mathbb{N}$, alongside addition operator `+` and identity element zero.
- The set of naturals $\mathbb{N}$, alongside multiplication operator $\cdot$ and unit-valued identity.
- The set of all finite strings over any fixed alphabet Σ, alongside the `string_concatenate` operation and an empty string as identity.
- A set of `Lists` paired with the concatenate operation (`++`, in Scala) and an empty list (`List.empty`) identity element.

2.9.1 Hands on Monoids

See Hands on Monoids, by Leif Battermann. Suppose we wish to aggregate a `List` of `Maps`.

```
List(  Map("a" -> 1, "b" -> 2),  Map("a" -> 2, "b" -> 0),  Map("c" -> 7) )
```

Desired prototype: `def Aggregate(counts : List[Map[String, Int]]) = ???`.
And for the example input above, we should obtain as output

```
Map(("a" -> 3, "b" -> 2, "c" -> 7)
```

One idea is to first "flatten" our list of `Maps` into a single list of key-value pairs, where the *key* is the `String` and the *value* is the integer (or perhaps count); we can then aggregate (sum) values across like keys, and we're done! We can do this in a declarative way as follows:

```
counts.flatMap(_.toList),                                  // List[(String, Int)].
       groupBy{ case (k,v) => k }.                         // Map[Str, List[Str,Int]].
       map{ case( k, v) => (k, v.map(_._2).sum) }          // Extract integers, sum.
```

The first function call to `flatMap` takes the contents of each `Map` (two different `Maps`) and places them in a list, whereupon the lists are flattened/appended together to form an output. The second function call to `groupBy` allows us to store counts of similar keys adjacently. Finally, we use the `map` function to iterate over each key-value pair (here a `String` is the `key`, and the value is a List of counts), and apply an aggregation function to we sum counts.

Turns out that thanks to Monoids, the above expression is equivalent to the following:

```
import cats.implicits._
counts.combineAll
```

This works because the type `Map[A, B]` is used as part of a default type-class, and the same is true for the type `Int`. I.e. they can be defined to be part of a set of default monoids.

Foray into Parallel Computing For any binary associative operator, and given sufficient compute resources, we can actually apply our resources across an input of size $\Theta(n)$ elements with $\Theta(n)$ computational *work*, and *makespan* bounded by $\Theta(\log n)$.[5] A sequential algorithm might initialize a counter variable and iterate over elements, updating the counter in turn in a way equivalent to the following mathematical expression is evaluated:

$$\Big(((a_1 + a_2) + a_3) + \ldots + a_n \Big)$$

However, we can also realize that we may *pair* elements from our original input sequence and compute their individual sums independently from one another. Each time we do this, we half our input size (since each pairing takes two values and combines them into a singleton). Whence we require $\Theta(\log_2 n)$ recursive invocations of our recursive algorithm before "bottoming out" to a base case where we are left with a single element (the sum of interest). This technique holds for any binary associative operator!

[5] See for example CME 323: Distributed Algorithms, and my Lecture 1 scribe.