An evening of Scala

An evening of Scala

- A first look at Scala from a functional programming perspective
- Darren Wilkinson - darrenjw.github.io

Classes, objects and values

Mutable and immutable values

Scala has mutable variables and immutable values, and optional type inference.

```
var z = 5
// z: Int = 5
z = z + 1
z
// res1: Int = 6

val x = 5
// x: Int = 5
x
// res2: Int = 5
val y: Double = 2.5
// y: Double = 2.5
y
// res3: Double = 2.5
```

Case classes

Scala is often described as an object-functional hybrid language. Case classes are a nice lightweight way of representing *product data types*.

```
case class Person(name: String, age: Int)
val p1 = Person("Fred", 25)
// p1: Person = Person("Fred", 25)
p1
// res4: Person = Person("Fred", 25)
p1.name
// res5: String = "Fred"
```

Sum types

Sealed traits can model *sum types*.

```
sealed trait Pet {
  val name: String
  val age: Int
}
case class Cat(name: String, age: Int) extends Pet
case class Dog(name: String, age: Int) extends Pet

val c = Cat("Garfield", 20)
// c: Cat = Cat("Garfield", 20)
Dog("Scooby", 10)
// res6: Dog = Dog("Scooby", 10)
```

Note that by default, case class attributes are immutable.

Pattern matching

Scala has reasonably sophisticated support for *pattern-matching*, and case classes are often used in conjunction with pattern-matching.

```
def petString(p: Pet): String = p match {
    case Cat(n, _) => "A cat named " + n
    case Dog(n, _) => "A dog named " + n
}

petString(c)
// res7: String = "A cat named Garfield"
```

If a trait is declared `sealed`, then all cases must be defined in the same source file, and incomplete pattern matches will trigger a compiler warning.

Methods and functions

Function types

Scala makes a distinction between *methods*, declared using `def`, which is code associated with a particular class or object, and *functions* which are values having a function type. Methods can be converted to functions (sometimes transparently, but sometimes using an _). Methods can be type-polymorphic, whereas functions must have an explicit function type.

```
def pairs(n: Int): Int = n*(n-1)/2
pairs(4)
// res8: Int = 6

val pairsF: Int => Int = n => n*(n-1)/2
// pairsF: Int => Int = <function1>
pairsF(4)
// res9: Int = 6

pairs _
// res10: Int => Int = <function1>
```

Type polymorphism and currying

```scala
def pair[A](a1: A, a2: A): (A, A) = (a1, a2)

pair(1, 2)
// res11: (Int, Int) = (1, 2)
pair[Int](2, 3)
// res12: (Int, Int) = (2, 3)
pair("foo", "bar")
// res13: (String, String) = ("foo", "bar")

pair[Double] _
// res14: (Double, Double) => (Double, Double) = <function2
(pair[Double] _).curried
// res15: Double => Double => (Double, Double) = scala.Fun
```

The return type of a method or function can usually be inferred, but it is usually considered good practice to declare it.

HoFs

Since functions are just values, they can be passed around like any other value. Scala therefore supports *higher-order functions* - functions which either accept a function as an argument or return a function as a result (or both). HoFs are very often used in conjunction with collections.

Collections

Lists and vectors

Scala has a wide array of collection types, both mutable and immutable, but immutable by default.

```
val l = List(1, 2, 3)
// l: List[Int] = List(1, 2, 3)
l.head
// res17: Int = 1
l.tail
// res18: List[Int] = List(2, 3)
l(1)
// res19: Int = 2
0 :: l
// res20: List[Int] = List(0, 1, 2, 3)
l ++ l
// res21: List[Int] = List(1, 2, 3, 1, 2, 3)
```

Ranges and indexing

```
(1 to 10).toList
// res22: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
(0 until 10).toList
// res23: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
val v = Vector(1, 2, 3, 4)
// v: Vector[Int] = Vector(1, 2, 3, 4)
val v2 = v.updated(2, 5)
// v2: Vector[Int] = Vector(1, 2, 5, 4)
v2
// res24: Vector[Int] = Vector(1, 2, 5, 4)
v
// res25: Vector[Int] = Vector(1, 2, 3, 4)
```

Monadic collections

`List` has good *push* and *pop* performance, but `Vector` is an immutable data structure with better random access and update performance. The collections are *monadic*, supporting HoFs such as `map` and `flatMap` (called `bind` or >>= in Haskell).

```
v map (x => x*2)
// res26: Vector[Int] = Vector(2, 4, 6, 8)
v.map(x => x.toDouble)
// res27: Vector[Double] = Vector(1.0, 2.0, 3.0, 4.0)
v flatMap {x => Vector(x, x+1)}
// res28: Vector[Int] = Vector(1, 2, 2, 3, 3, 4, 4, 5)
```

`for` expressions

Scala has `for`-expressions (similar to Haskell's `do`-notation) for writing pure functional monadic expressions in an imperative style.

```
for {
  vi <- v
  vj <- v
} yield (vi, vj)
// res29: Vector[(Int, Int)] = Vector(
//   (1, 1),
//   (1, 2),
//   (1, 3),
//   (1, 4),
//   (2, 1),
//   (2, 2),
//   (2, 3),
//   (2, 4),
//   (3, 1),
//   (3, 2),
//   (3, 3)
```

for desugaring

Note that

```
for {
  vi <- v
  vj <- v
} yield (vi, vj)
```

is just syntactic sugar for

```
v flatMap {vi => (v map {vj => (vi, vj)})}
// res31: Vector[(Int, Int)] = Vector(
//   (1, 1),
//   (1, 2),
//   (1, 3),
//   (1, 4),
//   (2, 1),
//   (2, 2),
//   (2, 3),
//   (2, 4),
```

Scans and folds

The collections also support *scans*, *folds* and *reductions*.

```
v reduce (_ + _)
// res32: Int = 10
v reduce (_ * _)
// res33: Int = 24
v.foldLeft(0)(_ + _)
// res34: Int = 10
v.scanLeft(0)(_ + _)
// res35: Vector[Int] = Vector(0, 1, 3, 6, 10)
v.foldLeft(0.0)(_ + _)
// res36: Double = 10.0
```

Recursion

Recursion

Scala methods and functions can be recursive, though in this case, the return type must be provided.

```
def logFactorial(n: Int): Double =
    if (n <= 1) 0.0 else math.log(n)+logFactorial(n-1)

logFactorial(4)
// res37: Double = 3.1780538303479453
logFactorial(100)
// res38: Double = 363.7393755555636
logFactorial(1000)
// res39: Double = 5912.128178488171
logFactorial(10000)
// res40: Double = 82108.92783681415
```

Eventually this will blow the stack, since it is not tail-recursive (the function modifies the result of the recursive call).

Tail recursion

The compiler recognises tail-recursions, and performs tail call elimination.

```
@annotation.tailrec final
def logFact(n: Int, acc: Double = 0.0): Double =
    if (n <= 1) acc else logFact(n-1, math.log(n) + acc)

logFact(10000)
// res41: Double = 82108.92783681446
logFact(10000000)
// res42: Double = 1.5118096548755759E8
```

Note that the `tailrec` annotation is optional - the compiler will eliminate the tail call regardless of whether it is specified. The annotation ensures that an error will be thrown at compile time in the event that for some reason the compiler can not eliminate the tail call. This is typically better than discovering this fact at run time.

Streams

Streams

The Scala standard library includes a `Stream` type that is just a lazy list. There are quite a few limitations of such a stream, and the type is actually deprecated in the latest versions of Scala. There are many better stream types provided by other libraries (such as **monix** and **fs2**), but the simple built-in type is good enough to illustrate some of the basic concepts.

```
val naturals = Stream.iterate(1)(_ + 1)

naturals.take(8).toList
// res43: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)

val triangular = naturals.scanLeft(0)(_ + _).drop(1)

triangular.take(8).toList
// res44: List[Int] = List(1, 3, 6, 10, 15, 21, 28, 36)
```

Fibonacci numbers

```
def fib(a: Int = 1, b: Int = 1): Stream[Int] =
    a #:: fib(b, a+b)

fib().take(8).toList
// res45: List[Int] = List(1, 1, 2, 3, 5, 8, 13, 21)
```

This is actually stack-safe, though it might not be immediately obvious why. Again, you might prefer to use a proper streaming data library.

Type classes

Type classes

Scala supports a very powerful (but dangerous) programming feature allowing values and classes to be passed into functions and otherwise "summoned" *implicitly*. There are many potential applications of *implicits*, but in Scala 2 they are often used to support the *type class* programming pattern, popularised by Haskell. Note that Scala 3 ("dotty") provides more direct support for the type class pattern. The Cats library provides the standard type classes from category theory, together with instance definitions for types in the standard library, and convenient syntax. eg. The *monoid* type class is very useful and commonly used, and comes with the syntax |+| for the associative combine operation.

Monoid

```scala
import cats._
import cats.implicits._
import cats.syntax._

3 |+| 4
// res46: Int = 7
"foo" |+| "bar"
// res47: String = "foobar"
List(1, 2) |+| List(3, 4, 5)
// res48: List[Int] = List(1, 2, 3, 4, 5)
Map("a" -> 1, "b" -> 2) |+| Map("b" -> 3, "c" -> 4)
// res49: Map[String, Int] = Map("b" -> 5, "c" -> 4, "a" ->
```

Monoid type class constraint

```
def combineAll[A: Monoid](la: List[A]): A = la match {
    case Nil => implicitly[Monoid[A]].empty
    case x :: xs => x |+| combineAll(xs)
}

combineAll(List(2, 3, 4))
// res50: Int = 9
combineAll(List("foo", "bar"))
// res51: String = "foobar"
```

Functor

Defining a type class like `Monoid` in a language like Scala requires parameterised types (generics). Defining parameterised type classes requires parameterised types that are themselves parameterised - higher kinded types (HKTs). Very few mainstream programming languages support HKTs, but Scala is one of them. These can be used to define type classes like `Functor` and `Monad`.

Functor type class constraint

```
def doubleAll[F[_]: Functor](fi: F[Int]): F[Int] =
    fi map (_ * 2)

doubleAll(List(1, 2, 3))
// res52: List[Int] = List(2, 4, 6)
doubleAll(Vector(2, 3, 4))
// res53: Vector[Int] = Vector(4, 6, 8)
doubleAll(Option(3))
// res54: Option[Int] = Some(6)
```

Parameterising functions this way ensures that you don't use any idiosyncratic feature of the particular container type that is being used, and will render trivial the switching of the container type in some later refactor of the code.

Wrap-up

Conclusions

- Scala is a very powerful strongly typed language, with type inference, higher order functions and higher-kinded types, monadic for-expressions, and excellent support for immutable values and data structures, type classes, and functional programming more generally
- It also has a very mature tooling ecosystem, with excellent build tools, IDEs, testing frameworks, etc.
- It benefits from the JVM ecosystem, including the Sonatype central repository, for the sharing of code-signed libraries
- Scala is not a pure functional language, but an increasing number of Scala developers are embracing a pure functional approach to program development in Scala, and are creating a library ecosystem to support this