Scala Idioms

Dominic Fox

Scala Idioms

Dominic Fox

This book is for sale at http://leanpub.com/scalaidioms

This version was published on 2013-06-16

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©Dominic Fox 2013

Contents

Scala Is Not A Language

If you're new to Scala, the chances are you'll have some complaints. The language seems bewilderingly large and complex; things can and do go wrong in unexpected ways. One moment it's implicit function arguments that are the source of the frustration, the next it's something called the "cake pattern". Those who've been working with the language a little longer are likely to smile benevolently and humour your objections with an infuriating air of enlightened tolerance. "Just stick with it", they may say. "I felt exactly the same six months ago, and then…"

And then what? Not everybody will make their peace with Scala: some people will always dislike the language, and some of them will have good reasons for doing so. But the transition from wary discomfort to actual enjoyment of Scala programming is a common one, and seems to take place for many people after somewhere around six months of exposure to it. What changes for them? Is it just a matter of increased familiarity, or is there some particular realisation that makes all the difference?

A colleague of mine, Ian, described his own discovery of the true nature of Scala in the following terms. On first acquaintance, he had thought of it as a language like Java or Python, only bigger and hairier. His own preference for compact, single-idiom languages that are easy to fit into your brain made him rebel against Scala's seemingly profligate abundance of features and supported programming styles.

According to Larry Wall, famously, in Perl "there's more than one way to do it" - which makes Perl a great tool for hackers who enjoy creatively flexing the language, but a many-tentacled nightmare for those who prefer Python's more prescriptive rubric: "there should be one - and preferably only one - obvious way to do it". Scala seemed to Ian to favour the solo hacker (or freewheeling show-off) over the team-player who wrote software for others to read and understand.

Indeed, there was a general feeling in our team at the time that the only safe way to use Scala was to restrain the show-offs, and stick to a subset of the language that everyone felt comfortable with. The trouble with *that* approach was that whatever features we decided to rope off as unsafe, we'd find ourselves forced to use by 3rd-party libraries for which they appeared to be essential. The complexity seemed unavoidable.

However, my colleague came to realise that Scala was *not a language* in the sense the Java or Python were languages, but rather a toolkit out of which a family of related mini-languages could be put together. Each of these mini-languages was compact and consistent: the art of Scala programming was knowing their different idioms, and deciding which to use for a given task. Once you understood this, the different pieces of the picture started to separate out. Instead of a confusing whirl of novelty, Scala started to seem like a collection of well-designed *parts* - parts that actually hung together surprisingly well.

Of course, it's an exaggeration to say that Scala "isn't a language". What Scala isn't, is a *single-idiom* language. Scala is both "multi-paradigm", notably in the sense that it supports both object-oriented and functional programming styles, and *multi-idiom*, in the sense that it supports the creation of mini-languages tailored for specific purposes. If you’re having trouble finding out what “idiomatic Scala” is supposed to look like, it may be because there are multiple idioms to choose from.

The purpose of this book is to explore some of these idioms, and to show how a DSL (Domain Specific Language) like ScalaQuery, an advanced typeclass library like Scalaz and a distributed programming framework like Akka can coexist within the same language, each defining its own particular patterns of usage. We’ll be looking at code examples, getting to know the techniques and language features which support each idiom, and critically comparing them. My ambition is not to answer the question “when should I use *X*?” for you, but to enable you to answer that question for yourself, on the basis of a widely-informed understanding of the options and possibilities that Scala provides.

Rudiments

Styles of composition

In Scala, we are most often programming with *expressions* rather than with *statements*. What's the difference? A statement prescribes an action, something that the program will *do* while executing. For example, take the following method for deleting a file:

```
 def delete(file: java.io.File): Unit= {
     println("Deleting a file")
     file.delete();
     println("Deleted the file");
 }
```

This method executes a sequence of actions: write to the console, delete the file, write to the console again. The return type of the method as a whole is `Unit`, which indicates that it returns no value that we care about: the meaning of the method lies in the relationship between its inputs and what it *makes happen*. Now consider this method for computing the sum of three integers:

```
def sum3(i1: Int, i2: Int, i3: Int): Int = i1 + i2 + i3
```

The expression `i1 + i2 + i3` describes how to obtain a new value by combining existing values. It has no side-effects (such as deleting a file), but it defines a computation. The meaning of the method lies in the relationship between its inputs and the result of this computation.

In practice, things are often more muddled than this. A method that reads a line from a file and returns it has both side-effects (it moves the file-pointer on) and a non-`Unit` return value, one that we presumably care about. A method that writes a line to a file may return `Unit`, but might also throw an exception if the write-operation fails. Scala isn't a "pure" language, so it doesn't enforce (like Haskell) a strict separation between side-effecting and non-side-effecting code. It also permits a blend of code that deals with failures by throwing exceptions with code that discriminates between success and failure in its return values.

It makes sense, therefore, to think of a "statement" in Scala as a peculiar type of "expression" - one that is side-effecting, and which may have a `Unit` return type. Statements are usually composed into a sequence, as part of the body of a method: in order to do *X*, first do this, then that, then that. But expressions in general can be composed in a variety of ways, three of which we'll quickly consider here.

“Fluent” method chaining

You will often see Scala code that looks like this:

```scala
people.sortBy(_.name)
      .zipWithIndex
      .map { case (person, i) => "%d: %s".format(i + 1, person.name) }
      .foreach { println(_) }
```

Here, the return value of each method call becomes the *target* of the next. Sometimes (but not always) it can be clarifying to pull out and name some of the intermediate values (I’ve also added their types, although it’s usually unnecessary to state these explicitly):

```scala
val peopleByName: List[Person] = people.sortBy(_.name)
val peopleWithRank: List[(Person, Int)] = peopleByName.zipWithIndex
val toString = peopleWithRank.map { case (person, i) => "%d: %s".format(i +\
 1, person.name) }
toString.foreach { println(_) }
```

One advantage of this more explicit style is that it makes it clear where intermediate values, especially collections, are being constructed. (We’ll discuss later the use of `view` to make chained collection-transforming methods more efficient).

Infix combinators

Infix combinators enable functions to be chained together; the most commonplace is `andThen`:

```scala
val sortByName: List[Person] => List[Person] = _.sortBy(_.name)
val addRank: List[Person] => List[(Person, Int)] = _.zipWithIndex
val makeString: List[(Person, Int)] => List[String] = _.map { case (person,\
 i) => "%d: %s".format(i + 1, person.name) }
val printOut: List[String] => Unit = _.foreach{ println(_) }

(sortByName andThen addRank andThen makeString andThen printOut)(people)
```

Scalaz defines quite a few more, to do with such category-theoretic exotica as applicative functors and kleisli arrows. We’ll cover these in due course.

`for`-comprehensions

These deserve a longer treatment (and will get one, later on). For now, let’s just note that

```scala
(for {
  byName   <- persons.sortBy(_.name).point[Identity]
  withRank <- byName.zipWithIndex.point[Identity]
  toString <- withRank.map { case (person, i) => "%d: %s".format(i + 1, per\
son.name) }.point[Identity]
  _        <- toString.foreach { println(_) }.point[Identity]
} yield Unit).value
```

translates the code in the previous section into the *Identity monad*, and that `for`-comprehensions provide an extremely powerful idiom for composing computations of many different kinds.

If you already know about monads, that last sentence won't seem to say very much. If you don't, it will probably seem to say even less. For the moment, I just want to list the `for`-comprehension *syntax* among the styles of composition mentioned here. Monads are important even in "vanilla" Scala (that is, Scala without the Scalaz library); they're also *idiomatic*, to the point where a programmer who neither knows nor cares what a monad is will soon find themselves using a handful of common monads comfortably, and entirely unawares.

Querying and transforming data: `map`, `flatMap`, `filter` and `fold`

In this section, we'll look at one of the simplest and most fundamental sub-languages of Scala, the *data query and transformation language* provided by the collections library. We'll see how the logic of *relational queries* over database tables, expressed in SQL, translates into the logic of *functional queries* over in-memory collections, expressed in native Scala.

By "functional queries" I mean simply queries expressed using the higher-order functions `map`, `flatMap`, `filter` and `fold` (or its variants: `foldLeft`, `foldRight` and `reduce`). In Scala these are implemented as methods on collection types (e.g. `List`) which take a function and return the method's target transformed using that function. Let's start with an example of one such query. Consider the following snippet of code, which takes a list of three numbers, multiplies each of them by 2, throws away any that are not greater than 2, and adds the remaining numbers together:

```scala
List(1, 2, 3).map(_ * 2)
             .filter(_ > 2)
             .foldLeft(0)(_ + _)
//> res1: Int = 10
```

Syntactically, this has the look of a "fluent API" in which method calls are chained and laid out across several lines. Each method call returns a new value - the initial `List(1, 2, 3)` is unmodified - so that a sequence of *intermediate values* is threaded from call to call. If we pull out and name the intermediate values, it looks like this:

```
val initialList  = List(1, 2, 3)                  //> initialList: List[In\
t] = [1, 2, 3]
val doubles      = initialList.map(_ * 2)         //> doubles: List[Int] =\
 [2, 4, 6]
val greaterThan2 = doubles.filter(_ > 2)          //> greaterThan2: List[I\
nt] = [4, 6]
val summed       = greaterThan2.foldLeft(0)(_ + _) //> summed: Int = 10
```

The `map` method applies the function `_ * 2` to each member of `initialList`, returning a list of `doubles`, while the `filter` method creates a copy of `doubles` which contains only the values for which the function `_ > 2` returns `true`. Finally, the `foldLeft` function takes an initial value of `0` and adds each of the values in `greaterThan2` to it in turn, obtaining the sum of the values in the list. (In fact, for lists of numeric values there's a `sum` method that does exactly the same.)

A great many operations which involve selecting some subset of a dataset, transforming it and presenting some summary of the result can be expressed using this language, and it is common in Scala to use this idiom to express algorithms which can be broken down into steps of these kinds. (We'll consider some of the shortcomings of this idiom, and discuss an alternative approach, later on).

Relational queries

A *relational query* is a query which is expressed in terms of some variation of *relational algebra*. The database query language SQL may be considered a somewhat low-fidelity implementation of this idea. Among the operations it supports are *projection*, which identifies a range of columns to return, and *selection* which restricts the returned column-sets to those matching a particular condition.

Confusingly, the keyword SQL uses for projection is `SELECT`:

```
SELECT name, age
FROM   Person
```

Here we have a table, `Person`, and a query which projects the `name` and `age` columns from this table, returning a result set containing the name and age of every person in the table. For selection, we use the `WHERE` clause:

```
SELECT name, age
FROM   Person
WHERE  age > 40
```

In this case, we are only interested in persons over `40` - the condition (or "predicate") `age > 40` is applied as a test to each person in the table, and the projection is only applied to those which meet the condition.

Projection and selection in Scala

Let's see if we can define a similar query in Scala. Firstly, we define a `Person` using a case class:

```
case class Person(name: String, age: Int, favouriteColour: String)
```

Secondly, we create a data structure - in this case, a plain `List` - containing several Persons:

```
val maggie = Person("Maggie", 32, "Red")
val millie = Person("Millie", 24, "Blue")
val molly  = Person("Molly",  45, "Green")
val may    = Person("May",    41, "Yellow")
val persons = List(maggie, millie, molly, may)
```

Finally, we can express our query over this structure using `map` and `filter`:

```
persons.map(person => (person.name, person.age))
       .filter(_._2 > 40)
//> res1: List[(String, Int)] = [("Molly", 45), ("May", 41)]
```

The `map` performs the "projection" part of the SQL query (the `SELECT` clause), projecting a tuple of name and age out of each `Person`. The `filter` then applies the "selection" (the `WHERE` clause).

Joining tables

Now for a more complicated example. Here's a straightforward join in SQL, expressed by taking the *Cartesian product* of two tables and selecting only those rows in the product where the row in the second table is related to the row in the first:

```
SELECT p.name, p.age, a.postcode
FROM   Person p, Address a
WHERE  a.occupant_id = p.id
```

This will give us the name, age and postcode of every person who has an address, for every address that they have (some people might have more than one).

Now here's some Scala that will (rather inefficiently) do the same:

```
case class Address(postcode: String, occupant: Person)

val addresses = List(
  Address("AB1 2CD", maggie),
  Address("VB6 5UX", maggie),
  Address("HR9 5BH", may)
)

addresses.map(address =>
  persons.filter(address.occupant == _)
         .map(person => (person.name, person.age, address.postcode)))
//> res1: List[List[(String, Int, String)]] = List(List((Maggie,32,AB1 2CD)\
), List((Maggie,32,VB6 5UX)), List((May,41,HR9 5BH)))
```

Because we have two nested `maps` here, we end up with a `List` of `Lists`. We can "flatten" this into a single `List` by using `flatMap` in place of the first `map`:

```
addresses.flatMap(address =>
  persons.filter(address.occupant == _)
         .map(person => (person.name, person.age, address.postcode)))
//> res1: List[(String, Int, String)] = List((Maggie,32,AB1 2CD), (Maggie,3\
2,VB6 5UX), (May,41,HR9 5BH))
```

The combined use of `flatMap` and `map` here gives us a clue that this code could be written more concisely and pleasingly in `for`-comprehension syntax (if you don't know why this is, don't worry - we'll come back to it):

```
for {
  person  <- persons
  address <- addresses if address.occupant == person
} yield (person.name, person.age, address.postcode)
```

However, as we said, this code is rather inefficient - it has to do the equivalent of a "table scan" across the `addresses` collection for each `person` in the `persons` collection. If we had many more persons and addresses, then it might be better to construct an index:

```scala
val addressesByOccupant = addresses.groupBy(_.occupant)
for {
  person  <- persons
  address <- addressesByOccupant.getOrElse(person, List.empty)
} yield (person.name, person.age, address.postcode))
```

What does `groupBy` do? It maps over the collection, using the supplied function to extract a *key* for each value, and building up a list of items which have the same key. The resulting set of key/list-of-items pairs is returned as a `Map[Key, List[Item]]`. We can then use this to look up all the addresses having a given occupant (returning an empty `List` if the person has no addresses).

A good SQL query optimiser will determine when it is more efficient to construct an index to compute a join, and try to ensure that the "query plan" into which the SQL statement is compiled implements a suitable strategy for retrieving and combining the required data. In this Scala code we are working at the level of the query execution engine which actually carries out the query plan, rather than defining a query in a *declarative* style which can then be optimised. Later on we'll see how ScalaQuery makes it possible to define declarative queries, which are converted to SQL and passed to the database engine to optimise, using native Scala syntax.

All you need is `fold`

We could write a `groupBy` function explicitly in terms of `map` and `foldRight` as follows:

```scala
  def groupBy[A, B](iterable: Iterable[A], key: A => B): Map[B, List[A]] =
  iterable.map(item => (key(item), item))
          .foldRight(Map.empty[B, List[A]]) { (map, pair) =>
    map + (pair._1 -> (pair._2 :: map.getOrElse(pair._1, List.empty[A])))
  }
```