Denis Kalinin

Mastering Advanced Scala

Mastering Advanced Scala

Exploring the deep end of functional programming

Denis Kalinin

This book is for sale at http://leanpub.com/mastering-advanced-scala

This version was published on 2020-01-06

Contents

Preface

Over the past decade, Scala has evolved into a huge ecosystem that consists of thousands of projects. In this respect, it is similar to other popular languages and technologies - they all tend to grow bigger over time. However, by appealing to different kinds of people, Scala developed an unusually high degree of diversity inside a single and seemingly monolithic world.

There are libraries and frameworks that can be mastered in a couple of weeks. Implementation details behind these libraries may be relatively advanced, but for an end user who simply wants to get their work done they are extremely approachable.

One example of a very approachable framework written in Scala is Play[1]. It was highly inspired by Rails, so most Rails/Django/Grails developers will feel at home there.

If you are new to Scala, I usually recommend starting with Play and learn the language while using the framework. This approach lied the foundation for my first book "Modern Web Development with Scala"[2], which is also available on Leanpub.

In addition to Play, there are many other Scala tools that follow similar ideas. For example, take a look at the following fragment of code:

```scala
DB.readOnly { implicit session =>
  val maybeUser = sql"select * from users where user_code = $userCode".
    map(User.fromRS).single().apply()
  // ...
}
```

This code uses ScalikeJDBC[3], which provides DSL-like API via implicits, currying and several other techniques, but even people unfamiliar with Scala can correctly guess what's happening here.

There is also the standard library that provides many useful abstractions such as `Option`, `Future`, `Try` out of the box. Some people say that some standard classes are not as good as they should be, but I would argue that on the whole, the standard library gives a very pleasant and coherent impression.

Sometimes, however, all of this is just not enough.

My observations show that after developers get familiar with the language and tools, they start looking for ways to improve their code even further. Usually it is possible to achieve but requires mastering some advanced techniques and libraries.

[1] https://playframework.com/
[2] https://leanpub.com/modern-web-development-with-scala
[3] http://scalikejdbc.org/

These techniques and libraries are the main focus of this book.

Just as with my first book, I tried to make the explanation as practical as possible. Instead of concentrating on implementations of abstract concepts, we will be using existing libraries and tools such as ScalaZ and Cats. In particular, Cats will illustrate important category theory abstractions while ScalaZ will be used more or less as an improvement over the standard library. I also included several other libraries sometimes based on Cats and ScalaZ to show how purely functional concepts can be used in real projects.

Please note that although book doesn't expect any familiarity with category theory, it does expect that you already know Scala. If this is not the case, I would recommend leaving this book aside and picking "Modern Web Development with Scala" instead.

If, however, you already have some experience with Scala and want to know more, turn the page and let's get started!

Advanced language features

We will start with discussing some less known features available in the Scala programming language. Some of them are useful on their own and some serve as a basis for implementing other abstractions.

Implicit parameters

Let's recall that there are two types of implicits in Scala:

- implicit parameters
- implicit conversions

Implicit parameters are simple. If you declare a function parameter as `implicit`, the callers of your function will be able to avoid passing this parameter provided that they have an eligible value of a compatible type marked as `implicit`. For example, the following method:

```scala
def sayHello(implicit name: String): String = s"Hello $name"
```

can be called as parameterless as long as the scope contains an `implicit` value of type `String`:

```scala
implicit val name = "Joe"
println(sayHello)
```

When declaring a method taking a parameter of a user defined type, you can define an `implicit` value inside the object companion of this type and this value will be used by default. For example if you define your class and method as follows:

```scala
class Person(val name: String)
object Person {
  implicit val person: Person = new Person("User")
}

def sayHello(implicit person: Person): String = s"Hello ${person.name}"
```

the caller will always be able to call `sayHello` without declaring anything. This is a neat trick, but implicit resolution rules are slightly more involved than that. It turns out that in addition to defining an *implicit default* for a parameter of type `Person`, you can also define an *implicit default* for a parameter of type `F[Person]` (for example, `Option[Person]`, `List[Person]` and so on). And if the companion object contains corresponding implicit values, they could be used as well:

```scala
object Person {
  implicit val maybePerson: Option[Person] = Some(Person("User"))
}
def sayHello(implicit person: Option[Person]): String = /* ... */
```

As a result, users can define or import implicit values in the scope, but the compiler also checks object companions of associated types. We will see why this is convenient when we get to *type classes*.

Implicit conversions

Sometimes you need to change or add new methods to third-party classes. In dynamic languages this is achieved by "monkey patching", in C# or Kotlin by writing extension functions, in Scala by using *implicit conversions*.

For example, we can write an implicit conversion from a `String` to `Person`

```scala
case class Person(name: String) {
  def greet: String = s"Hello! I'm $name"
}
object Person {
  implicit def stringToPerson(str: String): Person = Person(str)
}
```

After importing the conversion method into scope, we will be able to treat `String`s as `Person`s - the compiler will convert types automatically:

```scala
import Person.stringToPerson
"Joe".greet
// Hello! I'm Joe
```

Since conversions like these are commonly used for adding new methods, Scala also provides a shortcut:

```scala
implicit class StringToPerson(str: String) {
  def greet: String = s"Hello! I'm $str"
}
```

By using implicit classes we can get rid of most boilerplate.

Type erasure and type tags

Everyone working with JVM knows that type information is erased after compilation and therefore, it is not available at runtime. This process is known as *type erasure*, and sometimes it leads to unexpected error messages.

For example, if we wanted to create an array of a certain type, which is unknown until runtime, we could write a naive implementation such as this:

```scala
def createArray[T](length: Int, element: T) = new Array[T](length)
```

However, because of type erasure, the above code doesn't compile:

```
error: cannot find class tag for element type T
   def createArray[T](length: Int, element: T) = new Array[T](length)
                                                 ^
```

The error message actually suggests a possible solution. We can introduce an additional implicit parameter of the type `ClassTag` to pass type information to runtime:

```scala
import scala.reflect.ClassTag

def createArray[T](length: Int, element: T)(implicit tag: ClassTag[T]) =
  new Array[T](length)
```

With this little adjustment, the above code compiles and works exactly as we expect:

```
scala> createArray(5, 1.0)
res1: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0)
```

In addition to the syntax shown above, there is also a shortcut that does the same thing:

```scala
def createArray[T: ClassTag](length: Int, element: T) = new Array[T](length)
```

Note that prior to Scala 2.10, you could achieve the same thing with `scala.reflect.Manifest`. Now this approach is deprecated and type tags are the way to go.

Existential types

In Java, type parameters were introduced only in version 1.5, and before that generic types literally didn't exist. In order to remain backwards compatible with existing code, Java still allows the use of raw types, so the following code generates a warning but compiles:

```
static void print(List list) {
  list.forEach(el -> System.out.println(el));
}
public static void main(String[] args) {
  List<Integer> ints = Arrays.asList(1, 2, 3);
  print(ints);
}
```

Even though the `el` parameter from the lambda expression is inferred as `Object`, declaring the function argument as `List<Object>` wouldn't accept a list of `Integers`.

As you probably know, Scala disallows raw types, so the following will result in a compilation error:

```
scala> def printContents(list: List): Unit = list.foreach(println(_))
<console>:12: error: type List takes type parameters
       def printContents(list: List): Unit = list.foreach(println(_))
                               ^
```

So, what can we do here if we need a `List` but don't care about the element type?

One possibility is to parametrize method:

```
scala> def printContents[T](list: List[T]): Unit = list.foreach(println(_))
printContents: [T](list: List[T])Unit
```

This certainly works, but here it seems an overkill. Why define a type parameter if we don't use it? A more logical approach is to use an *existential type*:

```
scala> def printContents(list: List[_]): Unit = list.foreach(println(_))
printContents: (list: List[_])Unit
```

Existential types allow us to specify only the part of the type signature we care about and omit everything else. More formally, `List[_]` can also be written as `List[T forSome { type T}]`, but the latter is significantly more verbose.

Initially, existential types were introduced to Scala for Java interoperability, so the `list` parameter of the Java `print` will translate into `java.util.List[_]` in Scala. Now they are used by many functional libraries and could be used in your code as well.

Type classes

The type class pattern is a very powerful technique that allows users to add new behaviour to existing classes. Instead of relying on inheritance, this approach is based on implicits and doesn't require classes to be tied together. Type classes can be found everywhere in third-party libraries and you should consider utilizing this technique in your code as well.

Typical examples of using type classes include serialization, pretty printing and so on. As an illustration, let's enhance our types with functionality of printing some customized object information (similar to what `toString()` does).

Essentially, what we want to do is be able to call our `printInfo()` method on both built-in types and user defined types:

```
scala> val user = User("Joe", 42)
user: User = User(Joe,42)

scala> user.printInfo()
[User] (Joe, 42)
```

In order to achieve this goal we will need three things:

- a type class interface with one or several methods
- several concrete type class instances (all marked as `implicit`)
- an implicit conversion containing the `printInfo()` method

Let's start with defining a type class interface, which is easy:

```scala
trait InfoPrinter[T] {
  def toInfo(value: T): String
}
```

We know that we will be printing text information, so we're defining a method returning a `String`.

Then we can define several default `InfoPrinter` implementations for built-in or library types:

```scala
object DefaultInfoPrinters {
  implicit val stringPrinter = new InfoPrinter[String] {
    override def toInfo(value: String): String = s"[String] $value"
  }
  implicit val intPrinter = new InfoPrinter[Int] {
    override def toInfo(value: Int): String = s"[Int] $value"
  }
}
```

We're putting everything inside the object for convenience. Also note that we're marking instances as `implicit`. If we didn't do that, automatic implicit resolution would not work and we would have to pass these objects manually.

Finally, we can define an implicit conversion that prints object information using the `toInfo` method:

```scala
object PrintInfoSyntax {
  implicit class PrintInfoOps[T](value: T) {
    def printInfo()(implicit printer: InfoPrinter[T]): Unit = {
      println(printer.toInfo(value))
    }
  }
}
```

This conversion will work as long as for type `T` there exists a type class instance `InfoPrinter[T]` marked as `implicit` and this instance is available in the scope. Since we already defined instances for `String` and `Int`, we can try them out:

```scala
import DefaultInfoPrinters._
import PrintInfoSyntax._

val number = 42
number.printInfo()    // prints "[Int] 42"
```

When users define a custom type `A`, all they need to do is write an `InfoPrinter[A]` implementation and mark it `implicit`. If they put this implicit value inside the companion object, it will be automatically available due to implicit resolution rules:

```scala
case class User(name: String, age: Int)
object User {
  implicit val userPrinter = new InfoPrinter[User] {
    override def toInfo(value: User): String =
      s"[User] (${value.name}, ${value.age})"
  }
}
```

Now, we can call `printInfo()` on `User` objects and it will work as expected.

When working with Play framework, users need to provide `Writes[A]` implementations for their classes if they want to enable JSON serialization:

```scala
case class UserView(userId: UUID, userCode: String, isAdmin: Boolean)
object UserView {
  implicit val writes: Writes[UserView] = Json.writes[UserView]
}
```

The `writes` helper uses a macro to generate necessary code at compile-time, but in any other respect this is a typical type class example.

Using Simulacrum

Michael Pilquist wrote an interesting tool called Simulacrum[4] that can reduce the amount of boilerplate you need to write to create type classes. It is based on Macro Paradise[5] and generates necessary code at compile time.

In order to use it, you need to enable macros and add the library itself in your `build.sbt`:

```
addCompilerPlugin("org.scalamacros" % "paradise" % "2.1.0"
  cross CrossVersion.full)

libraryDependencies ++= Seq(
  "com.github.mpilquist" %% "simulacrum" % "0.14.0"
)
```

With Simulacrum, the above example can be rewritten as follows:

```
import simulacrum._

@typeclass trait InfoPrinter[T] {
  def toInfo(value: T): String
}
```

The `typeclass` annotation creates necessary conversions. Provided that we still have the `User` definition from the previous example, we can invoke `toInfo` on `User` instances:

```
import InfoPrinter.ops._

val user = User("Joe", 42)
println(user.toInfo)    // prints "[User] (Joe, 42)"
```

The advantage of using Simulacrum may not be obvious in simple use cases like our `InfoPrinter` type class. However, once you start using type classes more and more, it becomes an invaluable tool.

[4]https://github.com/mpilquist/simulacrum
[5]http://docs.scala-lang.org/overviews/macros/paradise.html

Exploring ScalaZ

In this section we will look at one popular functional library called ScalaZ[6] and explore how its abstractions are often better than the standard library or more mainstream counterparts. Let's start with looking at ScalaZ disjunctions, which are often used as a replacement for `scala.util.Either`.

ScalaZ disjunctions

The `Either` type allows us to store the exception for later inspection if something goes wrong. If we have a method that works correctly only 60% of the time, we can define its return type as `Either` like this:

```scala
def queryNextNumber: Either[Exception, Long] = {
  val source = Math.round(Math.random * 100)
  if (source <= 60) Right(source)
  else Left(new Exception("The generated number is too big!"))
}
```

Later, we can pattern match the value of type `Either` to determine whether we have a successfully calculated value or an error. The problem here is that `Either` before Scala 2.12 didn't really have any bias. In the example above, we used `Right` as a value storage and `Left` as an exception storage, but it is only a convention. The `Either` itself doesn't have `map/flatMap` methods, so in order to use it in for comprehensions, we would need to switch to `Either` projections and it is not as convenient as it should be. For details, check out Daniel Westheide's excellent post about The Either Type[7].

Note that most code examples can be found in the book repository on GitHub[8]. To see more information about the project organization, please refer to Appendix A.

The `Try` type, which was added in Scala 2.10, solves the problems mentioned above, but also introduces one serious limitation. Unlike `Either`, its left type is fixed as `Throwable`. Therefore, you cannot create your own error type and use it as a method result in `Try`.

Interestingly, ScalaZ offers an alternative to `scala.util.Either` which is right-biased, works great in for comprehensions and comes with some additional utilities.

The usual way to start working with ScalaZ is to import all its definitions with the following:

[6]https://github.com/scalaz/scalaz
[7]http://danielwestheide.com/blog/2013/01/02/the-neophytes-guide-to-scala-part-7-the-either-type.html
[8]https://github.com/denisftw/advanced-scala-code

```
import scalaz._, Scalaz._
```

Then, you can use ScalaZ disjunctions in a way similar to `scala.util.Either`:

```
def queryNextNumber: Exception \/ Long = {
  val source = Math.round(Math.random * 100)
  if (source <= 60) \/.right(source)
  else \/.left(new Exception("The generated number is too big!"))
}
```

Alternatively, you can use `\/[Exception, Long]` instead of `Exception \/ Long`. Also, `\/.right` is the same as `\/-` and `\/.left` is the same as `-\/`.

Unlike `scala.util.Either`, ScalaZ disjunctions are right biased, so you can use them easily in for comprehensions.

Replacing Try

The `Try` type has a convenient feature of safely absorbing thrown exceptions. Not surprisingly, a similar functionality is also supported by disjunctions:

```
def queryNextNumber: Throwable \/ Long = \/.fromTryCatchNonFatal {
  val source = Math.round(Math.random * 100)
  if (source <= 60) source
  else throw new Exception("The generated number is too big!")
}
```

The `fromTryCatchNonFatal` method will happily catch all non-fatal exceptions and put them into an instance of `\/`. Note that here we changed our signature from `Exception \/ Long` to `Throwable \/ Long` and basically ended up with a more verbose version of `Try`. In reality, however, disjunctions are more flexible than that.

Let's create our own `Exception` subclass that will be able to store a generated number in addition to an error message:

```
class GenerationException(number: Long, message: String)
  extends Exception(message)
```

Instead of `fromTryCatchNonFatal`, we need to use the `fromTryCatchThrowable` method. It works in a similar way, but in order to infer the return type correctly, it also requires that a `NonNothing` implicit value is defined in the scope:

```scala
implicit val geNotNothing = NotNothing.isNotNothing[GenerationException]

def queryNextNumber: GenerationException \/ Long = \/.fromTryCatchThrowable {
  val source = Math.round(Math.random * 100)
  if (source <= 60) source
  else throw new GenerationException(source, "The generated number is too big!")
}
```

We don't have to define a `NotNothing` value, but in this case we will have to specify type parameters explicitly, like this:

```scala
def queryNextNumber: GenerationException \/ Long =
  \/.fromTryCatchThrowable[Long, GenerationException] {
  // ...
}
```

If you try invoking the `queryNextNumber` method several times, you will see that it actually works as expected:

```
scala> DangerousService.queryNextNumber
res2: scalaz.\/[services.GenerationException,Long] = \/-(9)

scala> DangerousService.queryNextNumber
res3: scalaz.\/[services.GenerationException,Long] = \
  -\/(services.GenerationException: The generated number is too big!)
```

Sequencing disjunctions

Sometimes you end up with a collection of disjunctions:

```scala
val lst = List(queryNextNumber, queryNextNumber, queryNextNumber)
```

If this is the case, you may want to get the disjunction of a collection. In order to do that, simply use the `sequence` method, which was added to `List` via the first import:

```scala
import scalaz._, Scalaz._
val lstD = lst.sequence
// lstD: \/[GenerationException, List[Long]]
```

If all numbers are generated successfully, you will get a `\/-` containing a `List[Long]`. If there is an exception, you will get a `-\/` with a `GenerationException`.

ScalaZ Task

The Scala standard library provides `scala.concurrent.Future` as a very convenient way to deal with asynchronous code. The `Future`, however, has one feature that often confuses people. In particular, when you wrap your code inside `Future.apply`, it starts executing immediately.

Let's define a simple method that we will use for emulating a sample computation:

```scala
def performAction(num: Int): Unit =
  println(s"Task #$num is executing in ${Thread.currentThread().getName}")
```

Now, let's wrap the call of this method inside of `scala.concurrent.Future` and see what happens:

```scala
import scala.concurrent.ExecutionContext.Implicits.global

val resultF = Future {
  performAction(0)
}
// Task #0 is executing in ForkJoinPool-1-worker-5
```

Our task started executing in a worker pool immediately. Alternatively, we can execute our code in the current thread using `Future.successful`, but this merely lifts a resulting value to `Future` without making the code asynchronous:

```scala
val result2F = Future.successful {
  performAction(1)
}
// Task #1 is executing in main
```

Quite often, this is exactly what you want. However, sometimes you need more control over when the task starts running. And if this is the case, the `scalaz.concurrent.Task` should be used instead:

```scala
val result2T = Task.now {
  performAction(2)
}
// Task #2 is executing in main

val result3T = Task {
  performAction(3)
}
// * nothing happened *
```

The `Task.now` method lifts a value to a `Task` by executing logic in the current thread. The `Task.apply` method schedules execution in a thread pool. You can pass the `ExecutorService` as the second argument or use ScalaZ's default thread pool. Either way, the task will not run until it's manually started. In other words, the sequence of commands inside `Task.apply` is a pure computation without side effects.

It is also possible to lift a computation into a `Task` lazily by means of the `delay` method:

```
val result4T = Task.delay {
  performAction(4)
}
```

This method is guaranteed to be stack-safe and therefore, it's often used in recursive algorithms. We will get to the topic of stack-safety in later chapters, but for now, let's concentrate on the `Task` itself.

Obviously, we can use `map/flatMap` combinations or put `Task`s into for comprehensions and do everything that we usually do with monads.

As for running `Task`s, there are many methods covering every imaginable use case. Here are only some of them:

method	**description**
`unsafePerformSync`	executes the task synchronously
`unsafePerformAsync`	executes the task asynchronously
`unsafePerformSyncAttempt`	executes the task synchronously wrapping results or exceptions in ScalaZ disjunction

Note that unlike `scala.concurrent.Future`, neither `unsafePerformSync`, nor `unsafePerformSync` swallow exceptions. If you want to prevent them, you need to either use `attempt`-methods or wrap your code in something like `\/.fromTryCatchNonFatal`.

Converting callbacks into Tasks

Sometimes you have to work with Java code that relies on callbacks for doing anything asynchronous. For example, AsyncHttpClient can be used for performing non-blocking requests:

```
val asyncHttpClient = new DefaultAsyncHttpClient()
asyncHttpClient.prepareGet("https://httpbin.org/get").execute().
  toCompletableFuture.whenComplete { (response, exc) => {
    // ...
  }
}
```

The problem here is that it returns `ListenableFuture`, which can be converted into Java's `CompletableFuture`. At the same time, everything else in your code probably uses promise-like structures such as ScalaZ

`Tasks` or standard `Futures`. In this case, you can convert a callback handler into a `Task` using the `Task.async` method. This method has the following signature:

```
def async[A](register: ((Throwable \/ A) => Unit) => Unit): Task[A]
```

The important part here is `(Throwable \/ A) => Unit`. Once we pass this function literal to `Task.async`, we will be able to dump the results into this function literal. Where do we get these results? Obviously, from the `whenComplete` callback:

```
val result6T = Task.async[String](handler => {
  asyncHttpClient.prepareGet("https://httpbin.org/get").execute().
    toCompletableFuture.whenComplete { (response, exc) => {
    if (exc == null) {
      handler(\/.right(response.getResponseBody(Charset.forName("UTF-8"))))
    } else handler(\/.left(exc))
  }}
})
// result6T: Task[\/[Throwable, String]]
```

If an exception occurs, the `whenComplete` method will receive a non-null instance of `Throwable`, which we can use to initialize the left side of a resulting disjunction. Otherwise, we need to pass the response body to initialize the right side of `\/`.

ScalaZ Actor

In essence, *actors* are units that interact with each other by sending and receiving immutable objects called *messages*. Received messages are added to the mailbox and processed one by one.

The most popular Scala implementation of *the actor model* is Akka. It's a full-featured, extremely high-performant toolkit that is used for building application infrastructure. Not surprisingly, many popular frameworks such as Spark and Play use Akka underneath.

The central abstraction in Akka is the `Actor` trait. This trait has one abstract method called `receive` which is a `PartialFunction[Any, Unit]`.

```scala
case class Message(text: String)

class MyActor extends Actor {
  override def receive = {
    case message: Message =>
      println(s"Received message: ${message.text}")
  }
}
```

Here, we're defining an actor that is able to work with messages of the `Message` type. However, what will happen if the actor receives something else? Nothing really: unsupported messages are silently swallowed and might potentially cause subtle logical bugs that remain unnoticed for weeks or months. We could somehow improve this situation by adding a "catch-all" case printing a warning, but wouldn't it be better to have an actor parametrized with the message type?

It turns out that with ScalaZ actors you can have that. ScalaZ actors are parametrized with the message type. They are also very minimalistic and lightweight: their entire implementation almost fits one screen. Since they lack many of Akka features (supervision, network transparency etc), they are not meant to be a substitution for Akka. Rather, they should be considered when a full-blown actor system based on Akka seems an overkill.

Creating ScalaZ actors is easy:

```scala
val actor = Actor.actor( (message: Message) => {
  println(s"Received message: ${message.text}")
})
```

The only thing that is required is a handler function. Optionally, you can add an error handler and a *strategy*. The strategy specifies how the actor will handle messages and ScalaZ provides several strategies out of the box:

method	description
`Strategy.DefaultStrategy`	executes evaluations using a default thread pool
`Strategy.Sequential`	executes evaluations in the current thread
`Strategy.Naive`	spawns a new thread for every evaluation
`Strategy.Id`	doesn't execute evaluations
`Strategy.SwingWorker`	executes evaluations using the pool of Swing worker threads

It is important that the `actor` instance defined above has a type of `Actor[Message]`, so any attempt to send a message of a wrong type will result in a compile error.

Isolating the mutable state

Many experts argue that actors shouldn't be used for concurrency tasks. Indeed, for use cases like the one shown above, `Futures` and `Tasks` would probably be a better fit. Actors, however, are great at managing internal (mutable) state.

If you are interested in going deeper into this topic, I recommend starting with a blog post called "Don't use Actors for concurrency"[9] written by Chris Stucchio. Also, check out the comment section below the post.

Since we always define our Akka actors in a class, using mutable state there is a no-brainer:

```scala
class MyActor extends Actor {
  private var counter = 0
  override def receive = {
    case message: Message =>
      counter += 1
      println(s"#$counter: ${message.text}")
  }
}
```

However, we can do pretty much the same thing with ScalaZ `Actors` using classes and closures:

```scala
class MyActor {
  private var counter = 0
  def handler(message: Message): Unit = {
    counter += 1
    println(s"#$counter: ${message.text}")
  }
}
object MyActor {
  def create: Actor[Message] = Actor.actor(new MyActor().handler)
}
```

In this case, each newly created instance of an `Actor` will have its own copy of `counter`.

[9] https://www.chrisstucchio.com/blog/2013/actors_vs_futures.html