Functional Programming in Scala for Mortals

by Sam Halliday

Functional Programming in Scala for Mortals

Sam Halliday

This book is for sale at http://leanpub.com/fp-scala-mortals

This version was published on 2017-04-24

Contents

Preface

This book is for Scala developers with an Object Oriented (OOP) background who wish to learn the **Functional Programming** (FP) paradigm.

Until now, Scala has lacked a practical introduction to FP. We do not believe that learning Haskell should be a prerequisite. We also do not accept that the merits of FP are obvious. Therefore, this book justifies every concept with practical examples, in Scala.

We recommend The Red Book[1] as optional further reading. It is a textbook to learn the fundamentals and write your own FP library in Scala, serving a different purpose than this book.

We also recommend Haskell Programming from First Principles[2] as optional further reading. FP innovation has traditionally been in Haskell because it is the academic standard. But Scala has gained much more traction in industry and brings with it features, stability, interop, powerful frameworks and a commercial ecosystem.

[1]https://www.manning.com/books/functional-programming-in-scala
[2]http://haskellbook.com/

Copyleft Notice

[3]https://www.gnu.org/philosophy/free-sw.en.html
[4]https://creativecommons.org/licenses/by-sa/4.0/legalcode
[5]https://www.apache.org/licenses/LICENSE-2.0

Thanks

Diego Esteban Alonso Blas, Raúl Raja Martínez and Peter Neyens of 47 degrees for their help with understanding the principles of FP, cats and freestyle. Yi Lin Wei and Zainab Ali for their tutorials at Hack The Tower meetups.

Ani Chakraborty and Rory Graves for giving feedback on early drafts of this text.

Juan Manuel Serrano for All Roads Lead to Lambda[6], Pere Villega for On Free Monads[7], Dick Wall and Josh Suereth for For: What is it Good For?[8], John de Goes for A Beginner Friendly Tour[9].

Those who helped explain concepts needed to write the example project drone-dynamic-agents[10]: Paul Snively.

The helpful souls on the cats[11] chat room: Merlin Göttlinger, Edmund Noble, Rob Norris, Adelbert Chang, Kai(luo) Wang.

The helpful souls on the fs2[12] chat room: Michael Pilquist, Adam Chlupacek, Pavel Chlupacek.

[6]https://skillsmatter.com/skillscasts/9904-london-scala-march-meetup#video
[7]http://perevillega.com/understanding-free-monads
[8]https://www.youtube.com/watch?v=WDaw2yXAa50
[9]http://degoes.net/articles/easy-monads
[10]https://github.com/fommil/drone-dynamic-agents/issues?q=is%3Aissue+is%3Aopen+label%3A%22needs+guru%22
[11]https://gitter.im/typelevel/cats
[12]https://gitter.im/functional-streams-for-scala/fs2

Practicalities

If you'd like to set up a project that uses the libraries presented in this book, you will need to use a recent version of Scala with FP-specific features enabled (e.g. in `build.sbt`):

```
scalaVersion in ThisBuild := "2.12.2"
scalacOptions in ThisBuild ++= Seq(
  "-language:_",
  "-Ypartial-unification"
)
```

and add the following dependencies to your project's settings:

```
libraryDependencies ++= Seq(
  "io.circe" %% "circe-core",
  "io.circe" %% "circe-generic",
  "io.circe" %% "circe-parser"
).map(_ % "0.7.0") ++ Seq(
  "org.typelevel" %% "cats"     % "0.9.0",
  "com.spinoco"   %% "fs2-http" % "0.1.6"
)

resolvers += Resolver.sonatypeRepo("snapshots")
addCompilerPlugin("org.scalamacros" %  "paradise"  % "2.1.0" cross CrossVersion.full)
libraryDependencies += "com.47deg"  %% "freestyle" % "0.1.0-SNAPSHOT"
```

In order to keep our snippets short, we will omit the `import` section. Unless told otherwise, assume that all snippets have the following imports:

```
import cats._
import cats.implicits._
import freestyle._
import freestyle.implicits._
import fs2._
```

Introduction

It is human instinct to be sceptical of a new paradigm. To put some perspective on how far we have come, and the shifts we have already accepted on the JVM, let's start with a quick recap of the last 20 years.

Java 1.2 introduced the Collections API, allowing us to write methods that abstracted over mutable collections. It was useful for writing general purpose algorithms and was the bedrock of our codebases.

But there was a problem, we had to perform runtime casting:

```
public String first(Collection collection) {
  return (String)(collection.get(0));
}
```

In response, developers defined domain objects in their business logic that were effectively `CollectionOfThings`, and the Collection API became implementation detail.

In 2005, Java 5 introduced *generics*, allowing us to define `Collection<Thing>`, abstracting over the container **and** its elements. Generics changed how we wrote Java.

The author of the Java generics compiler, Martin Odersky, then created Scala with a stronger type system, immutable data structures and multiple inheritance. This brought about a fusion of object oriented (OOP) and functional programming (FP).

For most developers, FP means using immutable data structures as much as possible, but mutable state is still a necessary evil that must be isolated and managed, e.g. with Akka actors or `synchronized` classes. This style of FP results in simpler programs that are easier to parallelise and distribute, an improvement over Java. But it is only scratching the surface of the benefits of FP, as we'll discover in this book.

Scala also brings `Future`, making it easy to write asynchronous applications. But when a `Future` makes it into a return type, *everything* needs to be rewritten to accomodate it, including the tests, which are now subject to arbitrary timeouts.

We have a problem similar to Java 1.0: there is no way of abstracting over execution, much as we had no way of abstracting over collections.

Abstracting over Execution

Let's say we want to interact with the user over the command line interface. We can `read` what the user types and we can `write` a message to them.

```
trait TerminalSync {
  def read(): String
  def write(t: String): Unit
}

trait TerminalAsync {
  def read(): Future[String]
  def write(t: String): Future[Unit]
}
```

But how do we write generic code that does something as simple as echo the user's input synchronously or asynchronously depending on our runtime implementation?

We could write a synchronous version and wrap it with `Future` but now we have to worry about which thread pool we should be using for the work, or we could `Await.result` on the `Future` and introduce thread blocking. In either case, it's a lot of boilerplate and we are fundamentally dealing with different APIs that are not unified.

Let's try to solve the problem like Java 1.2 by introducing a common parent. To do this, we need to use the *higher kinded types* Scala language feature.

Higher Kinded Types allow us to use a *type constructor* in our type parameters, which looks like `C[_]`. This is a way of saying that whatever `C` is, it must take a type parameter. For example:

```
trait Foo[C[_]] {
  def wrap(i: Int): C[Int]
}
```

A type constructor is syntax for a type that takes a type to construct another type. `List` is a type constructor because it takes a type (e.g. `Int`) and constructs a type (`List -> Int -> List[Int]`). We can implement `Foo` using `List`:

```
object FooList extends Foo[List] {
  def wrap(i: Int): List[Int] = List(i)
}
```

We can also implement `Foo` for anything with a type parameter hole, e.g. `Either[String, _]`. Unfortunately it is a bit clunky and we have to create a type alias:

```
type EitherString[T] = Either[String, T]
object FooEitherString extends Foo[EitherString] {
 def wrap(i: Int): Either[String, Int] = Right(i)
}
```

There is a trick we can use when we want to ignore the type constructor. Recall that type aliases don't define any new types, they just use substitution for convenient names. Let's define a type alias to be equal to its parameter:

```
type Id[T] = T
```

Before proceeding, convince yourself that `Id[Int]` is the same thing as `Int`, by substituting `Int` into `T`. But `Id` is a valid type constructor, so we can use `Id` in an implementation of `Foo`:

```
object FooId extends Foo[Id] {
  def wrap(i: Int): Int = i
}
```

We want to define `Terminal` for a type constructor `C[_]`. By defining `Now` to construct to its type parameter (like `Id`), we can implement a common interface for synchronous and asynchronous terminals:

```
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}

type Now[X] = X

object TerminalSync extends Terminal[Now] {
  def read: String = ???
  def write(t: String): Unit = ???
}

object TerminalAsync extends Terminal[Future] {
  def read: Future[String] = ???
  def write(t: String): Future[Unit] = ???
}
```

You can think of `C` as a *Context* because we say "in the context of executing `Now`" or "in the `Future`".

But we know nothing about `C` and we can't do anything with a `C[String]`. What we need is a kind of execution environment that lets us call a method returning `C[T]` and then be able to do something with the `T`, including calling another method on `Terminal`. We also need a way of wrapping a value as a `C[_]`. This signature works well:

```
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def wrap[B](b: B): C[B]
}
```

letting us write:

```
def echo[C[_]](t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { in: String =>
    e.doAndThen(t.write(in)) { _: Unit =>
      e.wrap(in)
    }
  }
```

We can now share the `echo` implementation between synchronous and asynchronous codepaths!

We only need to write an implementation for `Execution[Now]` and `Execution[Future]` once and we can reuse it forever, for any method like `echo`. We can trivially write a mock implementation of `Terminal[Now]` and use it in our tests.

But the code is horrible! Let's use the `implicit class` Scala language feature (aka "enriching", "ops" or "syntax") to give `C` some nicer methods when there is an implicit `Execution` available. We'll call these methods `flatMap` and `map` for reasons that will become clearer in a moment:

```
object Execution {
  implicit class Ops[A, C[_]](val c: C[A]) extends AnyVal {
    def flatMap[B](f: A => C[B])(implicit e: Execution[C]): C[B] =
          e.doAndThen(c)(f)
    def map[B](f: A => B)(implicit e: Execution[C]): C[B] =
          e.doAndThen(c)(f andThen e.wrap)
  }
}
```

which cleans up `echo` a little bit

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  t.read.flatMap { in: String =>
    t.write(in).map { _: Unit =>
      in
    }
  }
```

we can now reveal why we used `flatMap` as the method name: it lets us use a *for comprehension*, which is just syntax sugar over nested `flatMap` and `map`.

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
    in <- t.read
     _ <- t.write(in)
  } yield in
```

Our `Execution` has the same signature as a trait in the cats library called `Monad` (except `doAndThen` is `flatMap` and `wrap` is `pure`). We say that `C` is *monadic* when there is an implicit `Monad[C]` available. In addition, cats has the `Id` type alias.

The takeaway is: if we write methods that operate on monadic types, then we can write procedural code that abstracts over its execution context. Here, we have shown an abstraction over synchronous and asynchronous execution but it can also be for the purpose of more rigorous error handling (where `C[_]` is `Either[Error, _]`), managing access to volatile state, performing I/O, or auditing of the session.

Pure Functional Programming

FP functions have three key properties:

- **Totality** return a value for every possible input
- **Determinism** return the same value for the same input
- **Purity** the only effect is the computation of a return value.

Together, these properties give us an unprecedented ability to reason about our code. For example, caching is easier to understand with determinism and purity, and input validation is easier to isolate with totality.

The kinds of things that break these properties are *side effects*: accessing or changing mutable state (e.g. generating random numbers, maintaining a `var` in a class), communicating with external resources (e.g. files or network lookup), or throwing exceptions.

But in Scala, we perform side effects all the time. A call to `log.info` will perform I/O and a call to `asString` on a `Http` instance will speak to a web server. It's fair to say that typical Scala is **not** FP.

However, something beautiful happened when we wrote our implementation of `echo`. Anything that depends on state or external resources is provided as an explicit input: our functions are deterministic and pure. We not only get to abstract over execution environment, but we also get to dramatically improve the repeatability - and performance - of our tests. For example, we are free to implement `Terminal` without any interactions with a real console.

Of course we cannot write an application devoid of interaction with the world. In FP we push the code that deals with side effects to the edges. That kind of code can use battle-tested libraries like NIO, Akka and Play, isolated away from the core business logic.

This book expands on the FP style introduced in this chapter. We're going to use the traits and classes defined in the *cats* and *fs2* libraries to implement streaming applications. We'll also use the *freestyle* and *simulacrum* developer tooling to eliminate some of the boilerplate we've already seen in this chapter, allowing you to focus on writing pure business logic.

For Comprehensions

Scala's `for` comprehension is heavily used in FP — it is the ideal abstraction to write pure procedural code. But most Scala developers only use `for` to loop over collections and are not aware of its full potential.

In this chapter, we're going to visit the principles of `for` and how cats can help us to write cleaner code with the standard library. This chapter doesn't try to write pure programs and the techniques can be immediately applied to a non-FP codebase.

Syntax Sugar

Scala's `for` is just a simple rewrite rule that doesn't have any contextual information. The compiler does the rewrite during parsing as *syntax sugar*, designed to reduce verbosity of the language.

The easiest way to see what a `for` comprehension is doing is to use the `show` and `reify` feature in the REPL to print out what code looks like after type inference (alternatively, invoke the compiler with the `-Xprint:typer` flag):

```
scala> import scala.reflect.runtime.universe._
scala> val a, b, c = Option(1)
scala> show { reify {
         for { i <- a ; j <- b ; k <- c } yield (i + j + k)
       } }

$read.a.flatMap(
  ((i) => $read.b.flatMap(
    ((j) => $read.c.map(
      ((k) => i.$plus(j).$plus(k)))))))
```

There's a lot of noise due to additional sugarings that you can ignore (e.g. `+` is rewritten `$plus`). The basic rule of thumb is that every `<-` (generator) is a nested `flatMap` call, with the final generator being a `map`, containing the `yield`.

For the remaining examples, we'll skip the `show` and `reify` for brevity when the REPL line is `reify>`, and also manually clean up the generated code so that it doesn't become a distraction.

We can assign values inline like `val ij = i + j` (the `val` keyword is not needed).

```
reify> for {
         i <- a
         j <- b
         ij = i + j
         k <- c
       } yield (ij + k)

a.flatMap {
  i => b.map { j => (j, i + j) }.flatMap {
    case (j, ij) => c.map {
      k => ij + k }}}
```

A `map` over the `b` introduces the `ij` which is flat-mapped along with the `j`, then the final `map` for the code in the `yield`.

> `val` doesn't have to assign to a single value, it can be anything that works as a `case` in a pattern match. The same is true for assignment in `for` comprehensions.
>
> Be careful that you don't miss any cases or you'll get a runtime exception (a *totality* failure):
>
> ```
> scala> val (first, second) = ("hello", "world")
> first: String = hello
> second: String = world
>
> scala> val list: List[Int] = ...
> scala> val head :: tail = list
> head: Int = 1
> tail: List[Int] = List(2, 3)
>
> // not safe to assume the list is non-empty
> scala> val a :: tail = list
> scala.MatchError: List()
> ```

Unfortunately we cannot assign a value before any generators[13]:

```
scala> for {
         initial = getDefault
         i <- a
       } yield initial + i
<console>:1: error: '<-' expected but '=' found.
```

but we can workaround it by defining a `val` outside the `for` or wrap the initial assignment:

[13] https://github.com/typelevel/scala/issues/143

```
scala> val initial = getDefault
       for { i <- a } yield initial + i

scala> for {
         initial <- Option(getDefault)
         i <- a
       } yield initial + i
```

It's possible to put `if` statements after a generator to call `withFilter`:

```
reify> for {
         i  <- a
         j  <- b
         if i > j
         k  <- c
       } yield (i + j + k)

a.flatMap {
  i => b.withFilter {
    j => i > j }.flatMap {
      j => c.map {
        k => i + j + k }}}
```

Older versions of scala called `filter`, but since `filter` in the collections library creates new collections, and excessive memory churn, `withFilter` was more performant.

Finally, if there is no `yield`, the compiler will use `foreach` instead of `flatMap`, which is only useful for side-effects.

```
reify> for { i <- a ; j <- b } println(s"$i $j")

a.foreach { i => b.foreach { j => println(s"$i $j") } }
```

The full set of methods that can be (optionally) used by a `for` comprehension do not share a common super type; each generated snippet is independently compiled. If there were a trait, it would roughly look like:

```
trait ForComprehendable[C[_]] {
  def map[A, B](f: A => B): C[B]
  def flatMap[A, B](f: A => C[B]): C[B]
  def withFilter[A](p: A => Boolean): C[A]
  def foreach[A](f: A => Unit): Unit
}
```

If an implicit `cats.FlatMap[T]` is available for your type `T`, you automatically get `map` and `flatMap` and can use your `T` in a `for` comprehension. `cats.Monad` implements `cats.FlatMap`, so anything that is monoidic (i.e. has an implicit `Monad[T]`) can be used in a `for`. Please do not make the equivalence between `for` and `Monad`, just because something can be used in a `for` comprehension does not mean it is monoidic (e.g. `Future` is not monoidic). We'll learn the difference when we discuss *laws*.

`withFilter` and `foreach` are not concepts that are useful in functional programming, so we won't discuss them any further.

It surprises developers that inline `Future` calculations in a `for` comprehension do not run in parallel:

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

for {
  i <- Future { expensiveCalc() }
  j <- Future { anotherExpensiveCalc() }
} yield (i + j)
```

This is because the `flatMap` spawning `anotherExpensiveCalc` is strictly **after** `expensiveCalc`. To ensure that two `Future` calculations begin in parallel, start them outside the `for` comprehension.

```
val a = Future { expensiveCalc() }
val b = Future { anotherExpensiveCalc() }
for { i <- a ; j <- b } yield (i + j)
```

`for` comprehensions are fundamentally for defining procedural programs. We will show a far superior way of defining parallel computations in a later chapter.

Unhappy path

So far we've only considered what the rewrite rules are, not what is happening in `map` and `flatMap`. Let's consider what happens when the container decides that it can't proceed any further.

In the `Option` example, the `yield` is only called when `i,j,k` are all defined.

```
for {
  i <- a
  j <- b
  k <- c
} yield (i + j + k)
```

How often have you seen a function that takes `Option` parameters but requires them all to exist? An alternative to throwing a runtime exception is to use a `for` comprehension:

```
def namedThings(
  someName  : Option[String],
  someNumber: Option[Int]
): Option[String] = for {
  name   <- someName
  number <- someNumber
} yield s"$number ${name}s"
```

but this is clunky and bad style. If a function requires every input then it should make this requirement explicit, pushing the responsibility of dealing with optional parameters to its caller — don't use `for` unless you need to.

If any of `a,b,c` are `None`, the comprehension short-circuits with `None` but it doesn't tell us what went wrong. If we use `Either`, then a `Left` will cause the `for` comprehension to short circuit with some extra information, much better than `Option` for error reporting:

```
scala> val a = Right(1)
scala> val b = Right(2)
scala> val c: Either[String, Int] = Left("sorry, no c")
scala> for { i <- a ; j <- b ; k <- c } yield (i + j + k)

Left(sorry, no c)
```

And lastly, let's see what happens with `Future` that fails:

```
scala> import scala.concurrent._
scala> import ExecutionContext.Implicits.global
scala> for {
         i <- Future.failed[Int](new Throwable)
         j <- Future { println("hello") ; 1 }
       } yield (i + j)
scala> Await.result(f, duration.Duration.Inf)

java.lang.Throwable
```

The `Future` which prints to the terminal is never called because, like `Option` and `Either`, the `for` comprehension short circuits.

Short circuiting for the unhappy path is a common and important theme. `for` comprehensions cannot express resource cleanup: there is no way to do `try` / `finally`. Cleanup needs to be a part of the thing that we're flat-mapping over. This is good, in FP it puts a clear ownership of responsibility for dealing with unexpected errors onto the `Monad`, not the business logic.

Gymnastics

Although it's easy to rewrite simple procedural code as a `for` comprehension, sometimes you'll want to do something that appears to require mental summersaults. This section collects some practical examples and how to deal with them.

Let's say we are calling out to a method that returns an `Option` and if it's not successful we want to fallback to another method, like when we're using a cache:

```
def getFromReddis(s: String): Option[String] = ...
def getFromSql(s: String): Option[String] = ...

getFromReddis(key) orElse getFromSql(key)
```

But, if we call `reddis <- getFromReddis(key)` in a `for`, it will short-circuit when it is `None`. What we need to do is to wrap the result in *another* for-comprehendable thing. We'll use a `Future` to make a point

```
for {
  cache <- Future { getFromReddis(key) }
  res   <- cache match {
             case Some(cached) => Future.successful(cached)
             case None         => Future { getFromSql(key) }
           }
} yield res
```

The call to `Future.successful` is like the `Option` constructor because it just wraps a single value. Every `Monad` in cats has a method called `pure` on its companion, adding some consistency to this pattern.

This `for` returns a `Future[Option[String]]` instead of the `Option[String]` that we started with, so we need to get out of the container by blocking the thread. If we had used `Option` instead of `Future`, we could use `flatten`.

In the next chapter we'll write an application and show that it is much easier if we define our methods to wrap everything in a monoidic container (just like in the introduction chapter), and let cats take care of everything

```
def getFromReddis(s: String): M[Option[String]]
def getFromSql(s: String): M[Option[String]]

for {
  cache <- getFromReddis(key)
  res   <- cache match {
             case Some(cached) => cached.pure
             case None         => getFromSql(key)
           }
} yield res
```

We could play code golf and write

```
for {
  cache <- getFromReddis(key)
  res   <- cache.orElseA(getFromSql(key))
} yield res
```

by defining https://github.com/typelevel/cats/issues/1625

If functional programming was like this all the time, it'd be a nightmare. Thankfully these tricky situations are the corner cases.

TODO Monad Transformers