Combining Imperative and Functional Programming

(for building modern concurrent software)

Kevin Mooney

University of Illinois Urbana-Champaign / CS 421 / Summer 2024

Introduction

Where we've been

It can be argued that years past there was a significant chasm between fully imperative languages and purely functional languages. While functional programming languages maintained their significance in academia their use in practical application was often somewhat niche. For example, while Erlang has been around since 1986, it's use outside of telecom was few and far between. Procedural imperative languages such as Fortran and C became the dominant player for some time, until eventually ceding much market share to object-oriented languages such as C++ and Java.

One could argue the merits of various families of languages until they are blue in the face, but very likely the trends and switches in industry are driven by the needs of the day. During the reign of procedural languages, resources were scarce and these languages provide a blade in dealing with concerns such as direct memory management. Later, in software development as the PC became more and more prevalent, and software became more user driven and less process driven, possibly it made more sense to model the world via relationships and graphs, hence the rise of OOP.

Where we are

Of course in recent years, a new set of innovations has arisen, and with it a new set of concerns. Systems have become increasingly interconnected. In addition, while Moore's law has not cooled as much as many have anticipated, the curve for clock speeds has begun to flatten due to increasing difficulty of cooling such tightly coupled transistors at higher frequencies. This have given rise to multiple CPU sockets on a system board, countless cores on a single CPU die or pipelining techniques such simultaneous multi-threading to provide more logical cores to the operating system. This in turn has again brought new engineering challenges to software design, and as such it is only logical that we adapt the languages and techniques we use to implement said software.

Imperative languages can at times have difficulty navigating the concurrent and asynchronous nature of the aforementioned systems. Often much care must be taken when writing concurrent (especially parallel) applications. Just as memory management can be the bane of languages like C, concurrency primitives such as locks, mutexes and semaphores, can lead to the same level of frustrations for software developers. Further is the issue of asynchronous development, which procedural code by nature does not mesh well with. In the past this was dealt with using polling or callback based APIs. Recently, languages such as Rust, JavaScript and Python have added the async/await concept, to make non-blocking asynchronous code appear to function in a blocking manner, but this is typically just syntactic sugar on top of existing polling or callback based APIs.

Where (I think) we're headed

Given that shifts in the interconnectivity, hardware and general interfacing patterns in the past have driven how we develop software, it is only logical to think the same would arise today, and it has. While functional languages have recently seen a large increase in industry, there has also been a drive to combine declarative programming paradigms (both functional and reactive) with existing imperative programming languages. Furthermore, you are seeing languages designed this way from the very beginning with but this mixed or fluent [Gifford86] paradigm. Typically, these elements include higher order functions, lambda expressions and functors. More recently, either natively or via third party libraries you are increasingly seeing these languages introduce pattern matching, algebraic data types and monad or monad like containers.

One thing that may hold back industry from a greater adoption of functional programming is the steeper learning curve when first breaking. It is doubtful many would argue that it is more difficult for a novice programmer to write a meaningful application in Java vs a language such as OCaml. Eventually this learning curve is surpassed, and then at that point the programmer is limited or at the very least slowed in what they can achieve due to the constraints of the language with a lower entry barrier. Fluent, languages provide two benefits in these cases. The first is that it lets the programmer ease into a more familiar procedural or OOP paradigm, and slowly graduate to the more complex functional aspects of the language. The second is that it now puts programmers in a position to exploit the strengths of each paradigm or mask the weakness of one over the other within the same program. The remainder of the paper will use one such fluent language, Scala, as well as supporting open source libraries to show the effectiveness of mixing imperative and functional concepts to build software that is performant, easier to reason about and easier to test.

Scala Fundamentals

To demonstrate the combination of imperative and functional programming concepts, the paper will use the Scala programming language. While a native compiler on top of LLVM as well as a javascript transpiler exist for Scala, it's first and still most prevalent compiler generates Java bytecode to be run on a Java virtual machine. The paper will also explore multiple open source libraries that are part of the Scala ecosystem. These libraries are largely independent of the base platform (native, js or jvm). Scala has facilities to support both imperative and functional programming. For the imperative bits, it very much resembles Java in syntax and object-oriented semantics. It was of course first built to run on the Java virtual machine, and with that also comes compatibility with existing Java libraries (when running the jvm based version).

When taking into account the functional aspects of Scala, you will see that it offers much of what you would expect from such a language - higher order functions, currying and immutability. Facilities you'd come to expect out of a language like Haskell such as algebraic data types and pattern matching are also present. With scala being both functional and object-oriented polymorphism is supported both via type classes as well inheritance. It so happens these are both supported by what is called a `trait` in Scala. Trait's can be used akin to an interface in Java or be used to define a type class. Syntactically they are the same, it is the context in which a trait is used which determines it's semantics.

Scala does not explicitly have type classes for a functor or a monad, however there are implicit language characteristics for supporting them (there are libraries that add official type classes for them, but those are out of scope for this paper). Functors are more subtle, in just that many types have a map function of type `F[A] => (A => B) => F[B]` similar to the signature of `fmap` in the Haskell Functor type class. The arguments seem reversed compared to Haskell, alas that is a consequence of how we must evaluate in Scala, where the map function is a member function of the functor where the functor is an object, and thus the function is evaluated as `obj.map(...)`.

Monad's in Scala are implemented by the existence of a single argument constructor to act as the unit (`return` in Haskell) definition and `flatMap` function to act as the bind definition. Hence, similar to the map function, flatMap carries the signature `F[A] => (A => F[B]) => F[B]`. To demonstrate what a simple trait may look like enforcing flatMap, we present the following (omitting the unit definition here as this is implemented via constructor)

```
trait Monad[A] {
  def flatMap[B](f: A => Monad[B]): Monad[B]
}
```

Note the above is only enforcing via the interface like usage of a trait. If we wanted to define a type class using a trait we would have defined it something like

```
trait Monad[F[A]] {
  def flatMap[B](m: F[A], f: A => F[B]): F[B]
}
```

While official type classes do not exist for functor or monad, implementing the aforementioned map and flatMap do

give way to using the functor or monad in a for comprehension. When sequencing a single functor, which is of the form

```
for (element <- functor) yield ???
```

the map function will be used, and the above can be thought more of as a list comprehension in Haskell. However, when sequencing over multiple monads, taking on the form

```
for {
  element1 <- monad1
  element2 <- monad2
} yield ???
```

the flatMap function will be used, and the above now functions more like the `do` notation. It is worth noting, functor only objects do not support multi-sequencing due to the lack of a flatMap function. Scala has many monads out of the box. Some of the most common functors you see in Haskell have Scala counterparts - List, Either and Maybe (called Option in Scala). The main concern of this paper is to cover things in the context of concurrent programming, so let's review two important monads scala provides to help facilitate this.

Try Monad

While the main concern is to focus on concurrent development, we would be remiss not to discuss error handling. Scala contains monad call `Try` that allows us to wrap a block of code.
The unit function (remember just a single argument constructor in Scala), defines it's parameter as call-by-name, which in Scala allows us to pass in any arbitrary expression. If the code succeeds, the resultant value is lifted into a `Success` which implements the Try trait. In Haskell parlance we can think of this as one of Try's type constructors. If the code throws, then the Throwable is caught and lifted into an instance of `Failure`. This provides us with a convenient means to wrap and compose impure side effecting code. Following are examples to demonstrate Try -

Lift execution into a Try. As we map over the Try, we are passed the values of successful computation. If at any point a Throwable is thrown, the execution is short-circuited, and the Throwable is now lifted into the Try

```
val simple: Try[String] = {
  val longComputation = Try((1L until 100000L).foldRight(0L)((a, b) => a + b ))
  longComputation
    .map(_ - 4999949958L)
    .map(answer => s"The meaning of life is $answer")
}
```

Using a for-comprehension to extract values to construct further Trys

```
val composed: Try[Int] = {
  val wrapped1: Try[Int] = Try(MockApi.blockingCall)
  val wrapped2: Try[Int] = Try(MockApi.blockingCall)
  for {
    inputA <- wrapped1
    inputB <- wrapped2
    sum  <- Try(inputA + inputB)
  } yield sum
}
```

Success or failure can be gleaned and the successful value or Throwable can be extracted using pattern matching

```
val patternMatching: String =
  simple match {
```

```
    case Success(result) => result
    case Failure(exception) => exception.getLocalizedMessage
  }
```

Future Monad

While the Try monad gave us a way to compose simple imperative and/or side-effecting code, the execution still happens linearly within the same context of the Try. That is to say it happens directly in the thread you created the Try in and the constructor of Try blocks while the code executes. As with Try the Future monad's unit parameter is defined as call-by-name. This of course allows for passing in any arbitrarily large expression, but also due to the lazy nature of call-by-name allows us to transfer the evaluation of that expression to a different execution context, and this is precisely what Future does. One nuance of the constructor definition for Future is that it is meant to be curried.

Let us step back for one second, and clarify the constructors for monads in Scala. While they have been described as being single argument constructors, these are not normal constructors in the sense of direct instantiation of the object. Scala allows defining a companion `object` alongside of a `class` definition. To skip over some of the gory details, these companion objects are often used to define static members of the class, among other things. One special function that can be defined on companion objects is the `apply` method. This then provides syntactic sugar that allows you to just specify the object/class name followed by parens, which will then invoke the apply method matching the argument signature you have passed if one exists. For example Future defines a companion object with an apply as follows

```
object Future {
  //Other code truncated for brevity

  final def apply[T](body: => T)(implicit executor: ExecutionContext): Future[T] =
    unit.map(_ => body)
}
```

This then allows you to invoke the apply method (or pseudo-constructor if you will), as follows

```
val future = Future {
  //arbitrary expression here
}
```

Now this may leave you wondering about the second curried argument `executor` in the apply method. This requires one further clarification. As can be seen, the parameter is annotated with the `implicit` qualifier. In Scala, an implicit parameter allows you to define an implicit value or function of that data type within certain designated scopes (I will not cover this scoping here as it can be quite complex). At runtime this value will be used (or if a function will be invoked, and it's return value taken) and fed to the implicit argument. Thus, you will typically have an implicit `ExecutionContext` in scope to feed to your Future.

```
/*
This would be defined somewhere within implicit scope with a real value.
Scala would automatically pass this to your curried Future constructor at runtime.
 */
implicit val executionContext: ExecutionContext = ???
```

With all of that out of the way, it needs to be noted what that ExecutionContext really is. It is simply a trait, which defines an execute method, which takes a java.lang.Runnable as it's only argument. As you can guess, Future will wrap the expression you passed into it's constructor in a Runnable and then leverage this ExecutionContext to execute your code. This has a major benefit in that you can control exactly how the Future executes your code

concurrently. Your ExecutionContext can be backed by a fixed thread, a thread pool or just run inline on the thread you are currently executing in. Recall that Scala is not restricted to only the jvm. The implementation that transpiles to javascript is then able to use an ExecutionContext that instead can make use of a javascript Promise to provide concurrency.

Similar to how Try allows us to compose synchronous code, and encapsulate failure, Future allows us to do the same. The major difference here is that the code can be concurrent and more importantly parallel and/or asynchronous. The composed Futures need not share the same ExecutionContext, and can thus be managed in different ways. Whichever way that may be, we are now able to reason about the concurrent execution in a more natural way. When composing futures via methods like for comprehensions it begins to look much like imperative code that is more familiar to the masses. Following are some examples to demonstrate Future -

Lift execution into a Future. As we flatMap over the Future, we are passed the values of successful computation. If at any point a Throwable is thrown, the execution is short-circuited, and the Throwable is now lifted into the Future. It is worth noting that the execution of the closures inside the flatMap and the Futures that are generated within will each be executed in a different context.

```
val bind: Future[String] = {
  val longComputation = Future((1L until 100000L).foldRight(0L)((a, b) => a + b ))
  longComputation
    .flatMap(n => Future(n - 4999949958L))
    .flatMap(answer => Future(s"The meaning of life is $answer"))
}
```

Using a for-comprehension to extract values to construct further Futures as with the above, the bits happening inside the for comprehension and in the yield itself are executed in a separate context from the Futures themselves

```
val composed: Future[Int] = {
  val wrappedLongExec: Future[Int] = Future(MockApi.blockingCall)
  val nonBlocking: Future[Int] = MockApi.nonBlockingCall
  for {
    inputA <- wrappedLongExec
    inputB <- nonBlocking
    sum  <- MockApi.nonBlockingCallWithInput(inputA, inputB)
  } yield sum
}
```

Lift a value directly into a Future, without invoking any concurrent execution. Useful when you want to avoid context switching or already know the value to return.

```
val simpleLift: Future[String] = {
  Future
    .successful(42)
    .map(answer => s"The meaning of life is $answer")
}
```

Use a Promise to lift a callback's execution into a Future. A Promise contains a .success method that we can call when we have a value to complete it with. This then completes an internal Future on the Promise that can be accessed via it's .future member. This allows us to pass a callback to a callback API that invokes the .success method. Externally the user interfaces with and collects the result via the promises .future member.

```
val callbackLift: Future[Int] = {
  val promise = Promise[Int]()
  MockApi.nonBlockingCallWithCallback(promise.success)
  val callbackFuture: Future[Int] = promise.future
  val nonBlocking: Future[Int] = MockApi.nonBlockingCall
```

```
  for {
    inputA <- callbackFuture
    inputB <- nonBlocking
    sum  <- MockApi.nonBlockingCallWithInput(inputA, inputB)
  } yield sum
}
```

Effectful Programming

While Try and Future are powerful monads for composing imperative code in a more functional manner, they are both simply wrappers around the results of an execution. To begin with, they are not lazy in the normal sense in their evaluation of the expression that they wrap. Try will execute immediately Future will execute as soon as it's ExecutionContext parameter will allow it. Due to this, they are stateful and can only be evaluated once. What we really want to do is wrap the potential of an expression rather than its evaluation. This is where effects enter the picture, as they allow us to wrap the description of an action as opposed to the result of an action.

Scala does not have an effect abstraction built-in, however the ecosystem contains multiple libraries which provide effectual monads. Arguably the most popular library, and the one that will be used as an example is Cats Effect. The core entity in Cats Effect is the IO monad. The IO monad functions in exactly as it's identically named counterpart from Haskell, allows you to wrap pure values as well as side-effecting actions. This allows us to wrap all of our imperative and side-effecting code to some degree into an effect, and then compose these bit's as if they were all pure.

IO has several methods for lifting pure values or actions into it. In addition to pure values or arbitrary expressions there are conveniences to lift Scala Future, Try, Either and Option monads directly into an IO, traverse sequence types or lift asynchronous/callback based actions directly into an IO. Facilities are also provided evaluate IOs in parallel. While the purpose of this paper is to focus on programming paradigms, the way Cats Effects manages IO execution and concurrency is worth mentioning. Under the hood it uses the concept of fibers (often also referred to as green threads) which are lightweight structures on top of native and/or bytecode virtual machine threads. The runtime manages execution of these fibers on top of normal threads, reducing OS/VM context switching and providing ways to cancel/interrupt threads not typically possible in something like a typical JVM thread. Programs are then a composition of IOs which are then feed through the Cat Effect managed runtime. Let's take a look at a couple examples of creating IO effect, and a sample program -
Simulate long-running synchronous API call. The » operator in analogous to a bind operation which doesn't actually pass the contained result of the first IO to the second.

```
def simple: IO[String] =
  IO.sleep(1.second) >> IO.pure(42).map(answer => s"The meaning of life is $answer")
```

Lift an async callback based API into an IO

```
def asyncCallback: IO[Int] =
  IO.async_(cb => MockApi.nonBlockingCallWithCallback(num => cb(Right(num))))
```

Execute multiple IO in parallel

```
def parallel: IO[Seq[Int]] =
  IO.parSequenceN(2)(Seq(simple, asyncCallback))
```

Small example of a program written using Cats Effect. This program reads fully qualified domain names from a text file, then resolves the IP address of those domain names through DNS.

```
import cats.effect.std.Console
import cats.effect.{ExitCode, IO, IOApp}

object IOSampleApp extends IOApp {
```

```
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      fqdns <- IOExamples.readFqdnList
      fqdnsWithDummy <- IO("dummy.bad" :: fqdns)
      _ <- IO.parTraverseN(fqdnsWithDummy.length)(fqdnsWithDummy)(hostname =>
        IOExamples
          .resolveIP(hostname)
          .map(address => s"Successfully resolved host hostname with ip $address")
          .handleError(error => s"Unable to resolve host ${error.getMessage}")
          .flatMap(Console[IO].println)
      )
    } yield ExitCode.Success
  }
}
```

Here we are extending the IOApp trait, which provides an abstract run function. Internally this trait contains the runtime dependent entry point setup (i.e. `static void main` on the JVM), which will in turn execute your run function definition. One simply need return an `IO[ExitCode]`, so the general idea is to compose several IO which in the end through various functor, monadic and similar transformations end with an ExitCode as the contained value. The referenced function IOExamples.readFqdnList has return type `IO[List[String]]` and returns a list of fully qualified domain names read from some arbitrary location. IOExamples.resolveIP has return type `IO[String]` and which represents a DNS resolved IP address. The IO.parTraverseN allows us to traverse over a sequence, here being a sequence FQDNs (with the first parameter controlling the number of fibers to execute on concurrently). The last parameter of parTraverseN expects a HOF which takes the sequence's contained type as a parameter and returns an IO to be evaluated in parallel. You can see we further transform the IO from IOExamples.resolveIP using map (functor fmap) and flatMap (monad bind).

This is all composed within a for comprehension which ultimately binds out to the ExitCode. This is a very small example for demonstration, but you can imagine for larger programs you simply compose smaller portions into larger logically composed IOs. This allows you to construct parallel and asynchronous code in a way that looks very similar to imperative code, while in the end being much easier to reason about than representing these concurrency constructs in actual imperative code. The also us to easily decompose our application down to its atomic structure if you will (if one can imagine effects as being the atoms). These discrete bits are then much easier to unit test and verify, especially when encapsulating aforementioned concurrent actions. They are also easier to integration test as you can compose them with other mock IOs and mini programs which simulate interactions of a real world program.

Reactive Programming

The final subject to touch on is a newer but very closely related programming style to utilizing effects called reactive programming. The concept itself however, is not directly tied to functional programming. Reactive programming itself is a declarative paradigm which is meant as a way to deal with streams of data. Often times, the syntax and semantics will employ constructs of both imperative and functional programming. Programs are often modelled as pipelines or graphs, through which data is transformed. These transformations often resemble behavior seen in that of functors and monads, or common functional patterns performed on sequences such as fold aggregations. In almost all languages reactive programming comes in the form of a library. Scala has various implementations some which take an actor model approach (which is out of scope for this discussion), but most which do lean almost exclusively in the functional camp.

One such library is FS2, which itself is based on Cats Effect that we have already reviewed. Being based on Cats Effect, it makes use of said libraries IO monad quite heavily. FS2 itself is built around Stream, a higher kinded type that takes two type parameters. The first type parameter is to provide a monad type constructor, and the second is to provide an output type. While you can provide your own monad evaluator to FS2, out of the box it is designed to work with the IO monad as mentioned. A Stream has several methods on it that allow you to transform elements as they pass through what can be thought of as a directed graph where each node represents some transformation. Some of these transforms such as map transform from one output type to another. Others operate more closely to a bind, and expect a function that takes one output type and returns the monad type of the Stream (typically IO). Streams are seeded from a source of data that can be viewed as a logic stream, such as a sequence type or some

source of real time events. The following are examples of creating a Stream, and a sample program -

Simple stream created from an apply (which takes a varargs set of elements). The stream adds 100 to each element

```
def simpleStream: Stream[IO, Int] =
  Stream(1, 2, 3, 4, 5).map(100.+)
```

Create a stream that just continuously evaluates an IO. Here the IO is lifted from an API call that returns a Future. It filters the data for even numbers and takes the first 5 it sees.

```
def fromFutureWithFilter: Stream[IO, Int] =
  Stream
    .repeatEval(IO.fromFuture(IO(MockApi.nonBlockingCall)))
    .filter(n => (n % 2) == 0)
    .take(5)
```

Build a stream using Queue. In this example, we use this queue in a closure that is passed to a callback based API. An internal stream is created from the evaluation of an IO, and thus we flatten out the surrounding stream to lift the internal one out.

```
def queueStream: Stream[IO, Int] =
  Stream.eval {
    for {
      queue <- Queue.unbounded[IO, Option[Int]]
      events <- IO.fromFuture(IO(MockApi.nonBlockingCallWithMultiCallback[IO[Unit]](n => queue.offer(Some(n)), 5)))
      _ <- IO.parSequenceN(5)(events.toList)
      _ <- queue.offer(None)
    } yield Stream.fromQueueNoneTerminated(queue).map(n => n * 2 + 1000)
  }.flatten
```

Small example of a program written using FS2. This program reads fully qualified domain names from a text file, then resolves the IP address of those domain names through DNS.

```
import cats.effect.std.Console
import cats.effect.{ExitCode, IO, IOApp}
import fs2.Stream

import java.net.InetAddress

object FS2SampleApp extends IOApp {

  override def run(args: List[String]): IO[ExitCode] =
    Stream
      .eval(IOExamples.readFqdnList)
      .flatMap(Stream.emits)
      .append(Stream("dummy.dummy"))
      .parEvalMapUnordered(4) { fqdn =>
        IO(InetAddress.getByName(fqdn))
          .map(Right.apply[Throwable, InetAddress])
          .handleError(Left.apply[Throwable, InetAddress])
      }
      .parEvalMapUnordered(4) {
        case Right(host) =>
          Console[IO].println(s"Successfully resolved host ${host.getHostName} with ip ${host.getHostAddress}")
        case Left(error) =>
```

```
            Console[IO].println(s"Unable to resolve host ${error.getMessage}")
        }
        .compile
        .drain
        .as(ExitCode.Success)
}
```

The example achieves the same result as the Cats Effect sample we showed earlier, however using a Stream abstraction. We first create a stream from IOExamples.readFqdnList which as before returns an `IO[List[String]]`. We then use flatMap to perform a bind like operation over another Stream, which is created using Stream.emits. Stream.emits simply creates a Stream from a sequence type. Next we append another Stream created from a single element. These parts compose the elements we will send downstream to other operators for transformation.

Moving to the first parEvalMapUnordered this process upstream elements in parallel. It takes a closure which creates an IO monad for evaluation. In this step which try to resolve the IP address. Note that we are mapping the result of this in an Either. The second parEvalMapUnordered, pattern matches on the Either emitted from the prior transform evaluating an IO which merely prints out success or not.

The compile function generates a projection that allows the Stream to be converted to it's monadic type parameter in a number of ways. In our instance we use the drain method which merely runs the Stream to completion and returns `M[Unit]` where M is our monad type constructor (IO in the example). There are other rich methods on the projection such as asList which will return the results as `M[List[O]]` where M is the monad type of the stream, and O is the type of the output of the final transformation. There are similar methods that perform operations such as a count on the elements or performing a right fold over the final elements.

Conclusion

We have seen by example how one can combine imperative and functional program to write heavily concurrent and side-effecting code in a more easily discernible and validated way. Scala was a logical choice here as the language was designed for this from the beginning. However, you can see many traditional imperative languages adding functional aspects as time goes on. Newer languages like Rust and Dart are like Scala being designed from the start to be multi-paradigm. In addition, it seems almost every language has at least one popular library which provides reactive extensions. As systems grow in number of cores, be it CPU, GPU or TPU, I feel that language constructs and libraries such as these will increase in popularity. To take it one step further, I believe these constructs will further bleed into distributed systems with libraries increasingly taking the same approach to providing location transparency.

Appendix

The accompanying repository for this document is located at https://github.com/moonkev/cs421-project

Examples can be found @ src/main/scala/com/github/moonkev/*Examples.scala

Samples apps can be found @ src/main/scala/com/github/moonkev/*SampleApp.scala

Tests cases can be found @ src/test/scala/com/github/moonkev/*Spec.scala

To run the test cases, run the following command (replace `./gradlew` with `gradlew.bat` on Windows)

```
$ ./gradlew test
```

To run the sample Cats Effect app

```
$ ./gradlew runcats
```

To run the sample FS2 app

```
$ ./gradlew runfs2
```

References

[Alexander17] Alvin Alexander *Functional Programming, Simplified (Scala Edition)*, CreateSpace Independent Publishing Platform

[Gifford86] David K. Gifford and John M Lucassen *Integrating Functional and Imperative Programming*, MIT Laboratory for Computer Science (1986)

[Lipovaca11] Miran Lipovaca *Learn you a Haskell for Great Good!*, No Starch Press (2011)

[Cats24] Typelevel (Various) *Cats Effect*, https://typelevel.org/cats-effect/ (2017-2022)

[FS24] Typelevel (Various) *FS2*, https://fs2.io/#/ (2024)