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 defer the execution of that expression, 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
implicit val executionContext: ExecutionContext = ???

val future = Future {
  //arbitrary expression here
} // The implicit curried executor parameter is automatically provided at runtime by the above executionCo
```

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

Reactive Programming

Conclusion

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)