Functional Composition & the Monad

Jaideep Ganguly

August 15, 2019

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[u, "\mu T"] & T \arrow[l, "\mu"]
\end{tikzcd}$$

Contents

1 Introduction

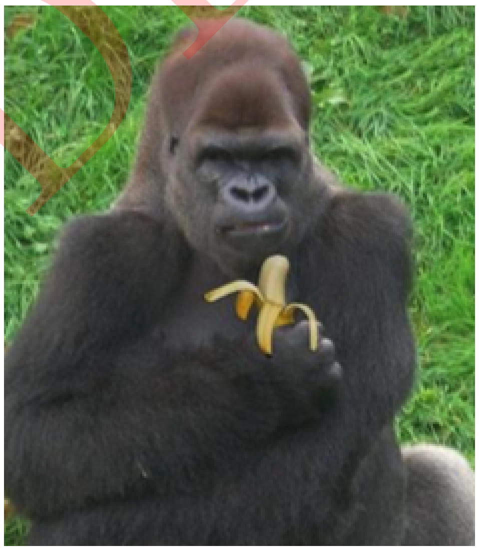
This book has been written to help developers migrate to the functional style of programming and be at least 2x more productive. Functional Programming is based on sound mathematical principles such immutability, referential transparency, functional composition and monads that have persisteed the test of time. The topic may appear daunting but is worth understanding given the benefits. The distinguished mathematician, Prof. G.H. Hardy once said, âĂIJA mathematician, like a painter or poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with ideasâĂİ âĂŞ G H Hardy.

The world of Category Theory in mathematics is full of obscure concepts. However, we will not go through a psychedelic ping pong of abstract ideas. Rather, we will review those concepts that have practical use in software engineering and demonstrate practical solutions through complete code examples.

Kotlin is the language for developing the ideas. A Java developer should be able to migrate to Kotlin and be reasonably proficient in just a couple of days. The IntelliJ IDE from JetBrains, creators of Kotlin, is the ideal IDE for development in Kotlin. Thereafter, one should be able to migrate to the functional style of programming, leverage type classes and write function compositions that can be executed asyncronously in just about a week.

2 The Trouble with Objected Oriented Paradigm

The Object Oriented Programming (OOP) paradigm mostly often results in obfuscating code using a plethora of classes, sub-classes and interfaces that add levels of abstraction but no value. The code follows a verbose, iterative, if-then style to encode business logic. It is not unusual to see developers write layers of abstractions in the form of Interfaces, Abstract Classes, Concrete Classes, Factories, Abstract Factories, Builders and Managers. Making changes become tedious, time consuming and error prone. Most of the assumptions around the need for extensibility is anyway false and much of the code written for the sake of extensibility is never used.



The problem with object-oriented languages is theyâĂŹve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle. - Joe Armstrong, the creator of Erlang.

A typical OOP code would be 50% larger than a functional code. Reusability and extensibility concept promoted by OOP through use of Interfaces and Factories is easily achievable through the use of higher order functions and function compositions. Moreover, a functional code would be devoid of complexities such as Dependency Injection (DI), Mocking, and Mutations, making the code and its testing much easier.

2.1 Dependencies

Dependency Injection and Design Patterns that are widely prevalent in OOP, are not evil in themselves. In fact they were hailed as an evolution in software development. Quickly though, they became abused, with developers using them whenever they could, mostly to showcase their abilities. Such is the dire state in OOP is that for a piece of code with business logic of 10 lines, developers write 100 lines in form of Interface, Abstract Classes, Concrete Classes, Factories, Abstract Factories, Builders and Managers!

2.2 Testing

This style of development introduces a plethora of dependencies and is a code smell. Lots of dependencies require lots of testing and testing in OOP is not easy. Furthermore, the reality is that most software engineers look at unit and integration testing as a chore and do not quite look forward to it. Tests are often perfunctory resulting in illusory productivity with a mess of alpha and beta quality software that require a magnitude more of effort towards fixes.

Furthermore, mocking is fairly ineffective with OOP because of multiple reasons. Firstly, test setup is slow because of the need to mock or stub the inputs along with any dependencies necessary for the code to execute a scenario. Secondly, multiple happy path scenarios and error scenarios need to be considered. And finally, test suites need to be written efficiently so that it does not take hours to run in continuous integration.

3 Language of Abstraction

Architects communicate with building drawings, electrical engineers communicate with circuit diagrams, chemical engineers communicate with process diagrams and so on. Unfortunately, much of software requirements are communicated in verbose word documents and exchanged via emails making the entire process riddled with ambiguity and uncertainty.

A machine code is composed of a sequence of instructions that processor dependent and a higher level language such as C had just 32 key words in its first version. Whatever be the functionality,from data base kernel to gaming, it has to be composed out of these limited set of reserved words. On the contrary, as per Noam Chomsky, humans communicate with a vocabulary of about 8,000 to 10,000 words, the combinatorial expressive power of which is immense. More importantly, the phrases can be ambiguous because semantic parsing can often be tricky. Consider the well known phrase *Mary had a lamb*. Did Mary have a lamb as a pet or did she have lamb with her dinner?

Clearly, no matter how well the requirement documents are written, mathematically, there can be never a unique one to one mapping between a verbose word document and the final code. This is precisely why engineering disciplines have adopted engineering drawings with a well defined set of symbols as a language of communication that conveys the the desired functionality. No much how much you try, an architect cannot build a house from a word document. Building drawings, structural drawings, electrical and HVAC drawings are essential to build a proper house.

Documentaton of business knowledge is critical to business success. We often encouner a plethora of documents or artifacts such as *Business Requiremnet Document*, *Product Requirement Document*, *High Level Design*, *Low Level Design* and so on. These artifacts are verbose, qualitative in nature and have varying degree of rigor depending on who created them.

3.1 Business Processes and Data Flow

Business processes are essentially a flow of information or data between various activities. Typical activities are validation, transformation, computation and so on. Information is encapsulated in entities or data structures and are nouns while activities are verbs. In Kotlin parlance, an entity is a data class and an activity is a function or an interface. Multiple entities flow into an activity and result in an output entity.

Data flows are best captured in a Markdown editor that supports Mermaid. It is simple to model entities and activities using a markdown editor such as Typora. Following is an example of a typical data flow.

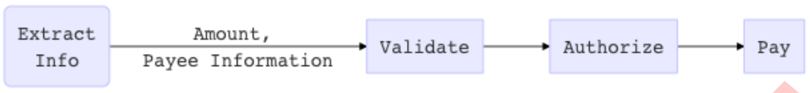


Figure 1: Data Flow diagram

Here the words inside the boxes are verbs and represent activities while entities which are nouns flow between the boxes.

4 Functional Programming

It is evident that there is a need to move from object dependencies to functional dependencies and ultimately evolve APIs towards behavior instead of objects. Functional Programming models behavior through a declarative paradigm, i.e., through expressions or declarations instead of statements. Functional programming brings in simplicity, are easier to test and eliminates the need for mocking.

The functional programming (FP) paradigm is rapidly getting adopted since it promotes the declarative style of coding that makes reading and writing code becomes far easier than before. It replaces multiple design patterns and dependency injection which is overused and abused in OOPs with the concepts of composition and higher order functions. FP does away with all such layers. One can argue that DI supports high extensibility. But most of the assumptions around the need for extensibility is false. Besides, business assumptions for future extensibility are rarely correct, since business moves much faster than code. Most of the code written for the sake of extensibility is never used, and just adds to the âĂIJlines of codeâĂİ.

In any case, in functional programming, extensibility is acheived through the use of higher order functions.

4.1 OOP versus FP

Listing 1: Typical Java OO code

```
/**
Interface
    ->
    Abstract Class
        -> Implementation 1
        -> Implementation 2

1. Interface defines the contract
2. Abstract class has base implementation
3. Implementations provide the actual implementation
4. Builds an inheritance tree, and favors aggregation over inheritance.
*/
```

Listing 2: Typical Kotlin/FP Code

```
fun function1( documentProvider: (Int) -> document,
    argument1: Int, argument2: Int )
/**
1. Higher order functions define the contract.
2. As long as the function documentProvider being passed to function1
follows the contract, the location, of that function is immaterial.
3. In essence, we get rid of the entire inheritance tree.
4. Finally, function style code avoids complexities like dependency
injection, mocking while testing and so on.
5. End result is a far simpler, smaller and extensible code base.
*/
```

4.2 Kotlin

Kotlin, a next generation language, helps eliminate verbose iterative and *if-then* style of coding and replace it with simple, terse and easily readable code using a **functional** style through the use of *collections, filters, lambdas and maps*. One can rely on *static objects* and use *extension functions* whenever required. However, Kotlin also allows you to write in the OOP style.

Kotlin inter-operates well with the Java based ecosystem in which we have heavy investments. It is easy to write high performant non-blocking, asynchronous, event driven code in half the size or even lesser than comparable Java code. It is easy to use its language constructs and write smaller code which removes null-checks, loops, branches and other boiler-plate code, leaving developers to focus on business logic than on repeated boilerplate code. Using Kotlin as a development language is a shift towards the right direction. An added benefit is that it will reduce our hardware costs. It learns from modern languages like C#, GO, Python and Ruby, and plugs in concepts in JVM ecosystem never seen before, and that too, in a highly intuitive and a smooth learning curve. Kotlin is the language of choice for Android development and is officially endorsed by Google.

5 Higher Order Functions

In languages that support the functional programming paradigm, a function can be passed as a parameter or can be returned from a function, the function which does the same is known as a higher-order function. In other words, a higher-order function is a function that takes functions as parameters or returns a function. Consider the following example where the function *pay* invokes a particular payment method depending on where the amount needs to be transferred. If it is within the same bank, no routing number is required but for inter bank transfer, the routing number is required.

Listing 3: Higher Order Functions

```
fun pay(accNo: Int, amount: Float, payMethod: (Int, Float) ->
    Unit, routingNo: Int = 0) : Unit {

    when (routingNo) {
        0 -> payMethod1(accNo, amount)
        else -> payMethod2(accNo,amount,routingNo)
    }
}


fun payMethod1(accNo: Int, amount: Float) {
    ...

}

```

```kotlin
fun payMethod2(accNo: Int, amount: Float, routingNo: Int) {
    ...
}
```

5.1 Pure and Impure Functions

A function is called a pure function if it depends only on the input to produce the result, not on any hidden information or external state in the body of the function. It should not cause any observable side effects, such as modifying a parameter passed by reference or a global variable/object. Pure functions are easy to test and lends themselves for parallel processing.

To minimize side effects in our code, it is important to maintain the discipline of separating pure and impure functions. While the goal is to write pure functions as much as possible, there will be impure functions in the code base due to IO, network operations, database calls and so on. During API design, the pure functions form the core while the outer shell consists of impure functions which call the pure functions. This way, any lateral injection of dependency is eliminated as shown in the figure.

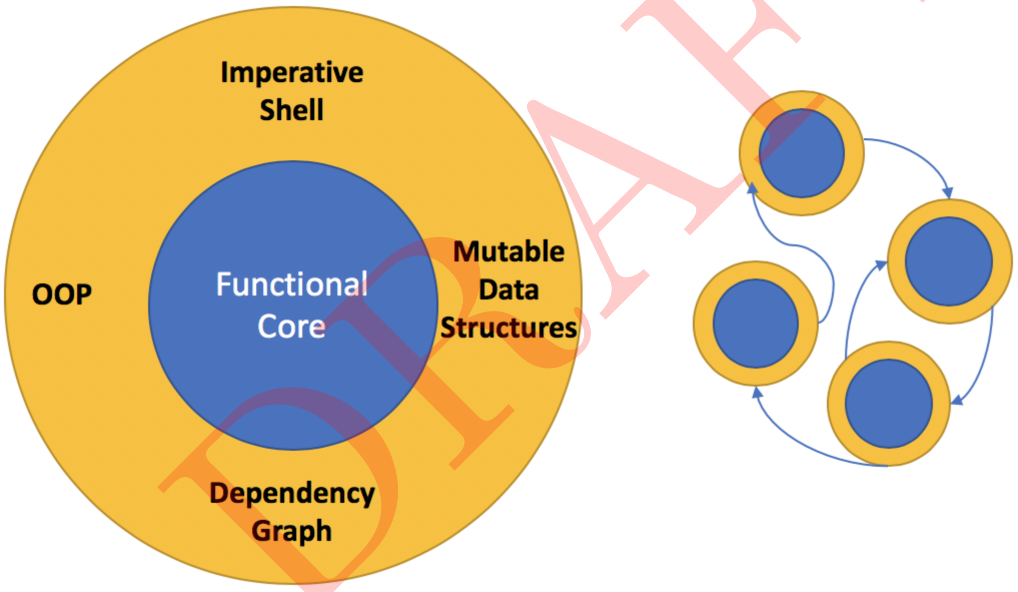


Figure 2: Core of Pure Functions and Outer Shell of Impure Functions

5.2 Call by Reference

In the early days of C and LISP programming, emphasis was laid towards writing programs where arguments were passed by value to ensure program correctness based on experiences with early languages such as fortran. However, in current software development practice, variables are mostly passed by reference as they are typically instances of complex objects. Furthermore, while designing the calling sequence, one has to be careful to ensure that the return type of a function matches the argument of the calling function, i.e., there is no impedance mismatch in the call sequence.

5.3 Exception Handling

The availability of *Try* - *Catch* mechanisms in programming languages have resulted in a drop in engineering rigor and it is common to see exceptions resulting in raw stack trace messages which suggests that the developer did not sufficiently think through the edge cases.

6 Functional Composition

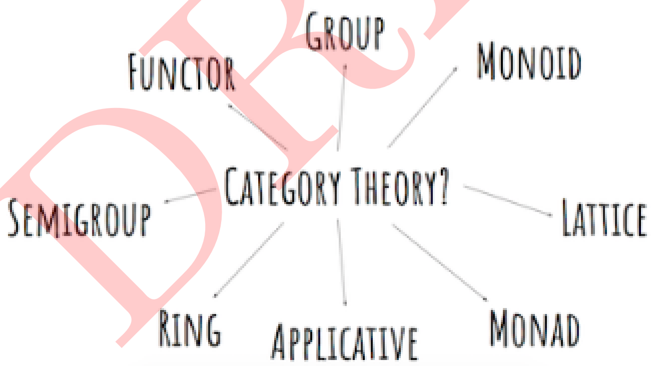
Since programming is a sequence of function calls, our goal is to introduce a design paradigm with the following objectives.

1. Eliminate the need to pass references by designing a data structure that is common to the set of functions that are called in a sequence.
2. The data structure should be able to handle failure modes, i.e., when no valid data is computed.
3. The pipeline, i.e., the composition of functions should be inherently parallelized as the context or the data structure and its associated functions are isolated from the rest of the computation.

In the next chapter we will formalize these concepts and use mathematically valid structures and operations that guarantee correct result at all times. We will develop associate code so that future tasks become almost trivial.

6.1 Category Theory

A category is an algebraic structure that comprises of objects that are linked by arrows. A category has two basic properties: the ability to compose the arrows associatively and the existence of an identity arrow for each object. A simple example is the category of sets, whose objects are sets and whose arrows are functions. A functor is simply a map between categories. An endofunctor is defined as a functor from one category back to the same category. In a statically typed language, we can translate the notion of set into the notion of type. A Monoid is a set that obeys certain rules. A functor is said to be Applicative when it preserves the monoidal structure. And a Monad is just a monoid in the category of endofunctors.



6.2 Type Class

The Applicative typeclass is a superclass of Monad, and the Functor typeclass is a superclass of Applicative. This means that all monads are applicatives, all applicatives are functors, and, therefore, all monads are also functors.

It is both mathematically correct and totally useless to anybody learning functional programming. So, if you haven't understood the above statements, there is not much to worry about.

6.3 Haskell

Haskell's combination of purity, higher order functions, parameterized algebraic data types, and type classes allows us to implement polymorphism on a much higher level than possible in other languages. We don't have to think about types belonging to a big hierarchy of types. Instead, we think about what the types can act like and then connect them with the appropriate type classes.

An Int can act like a lot of things. It can act like an equatable thing, like an ordered thing, like an enumerable thing, etc.

Type classes are open, which means that we can define our own data type, think about what it can act like and connect it with the type classes that define its behaviors. Because of that and because of Haskell's great type system that allows us to know a lot about a function just by knowing its type declaration, we can define type classes that define behavior that's very general and abstract. We've met type classes that define operations for seeing if two things are equal or comparing two things by some ordering. Those are very abstract and elegant behaviors, but we just don't think of them as anything very special because we've been dealing with them for most of our lives.

However, Haskell is not a JVM compatible language. And with so much historical investment in the JVM, we need to implement such capabilities in Kotlin.

Listing 4: Type Class

```
 1 /** Type Class class is a type system construct that supports ad hoc
 2  *  polymorphism. This is achieved by adding constraints to type
 3  *  variables in parametrically polymorphic types. Such a constraint
 4  *  typically involves a type class T and a type variable a, and means
 5  *  that a can only be instantiated to a type whose members support the
 6  *  overloaded operations associated with T.
 7  *  The keyword out is necessary as otherwise line 22 will
 8  *  throw a type mismatch error at compile time
 9  */
10 sealed class TC<out A> {
11 
12     object None: TC<Nothing>()
13     /** Value is actually a data class DC and
14      *  value is dc i.e., an instance of the data class DC and
15         extends the type class TC as below
16      */
17     data class Value<out A>(val value: A): TC<A>()
18 
19     /** Apply a function to a wrapped data, i.e. a type class
20         and return a wrapped data using flatMap
21         (liftM or >>= in Haskell)
22      */
23     inline infix fun <B> flatMap(f: (A) -> TC<B>): TC<B> = when (this) {
24         is None  -> this
25         is Value -> f(value)
26     }
27 }
```

6.4 Functions

Listing 5: Functions

```
data class DC (var data: Double, var rem: String)

fun mysqrt(a: DC) = when {
    a.data >= 0 -> {
        var y = kotlin.math.sqrt(a.data)
        var ds = DC(y,"ok")
        TC.Value(ds)
    }
    else -> {
        TC.Value(DC(0.0,"Error: Number is negative"))
```

```kotlin
    }
}

fun mylog(a: DC) = when {
    a.data > 0  -> {
        var y: Double = kotlin.math.ln(a.data)
        var ds = DC(y,"ok")
        TC.Value(ds)
    }
    else -> {
        TC.Value(DC(0.0,"Error: Denominator is 0"))
    }
}

fun myinv(a: DC) = when {
    a.data > 0 -> {
        var y: Double = 1/a.data
        var ds = DC(y,"ok")
        TC.Value(ds)
    }
    else -> {
        TC.Value(DC(0.0,"Error: Number is 0"))
    }
}
```

6.5 Composition

Listing 6: Class & Function

```kotlin
import TC.Value
import kotlinx.coroutines.*

fun testMonad() {
    /** Input type DC is wrapped
     * in type class TC
     */
    var inp: TC<DC>

    /** Functions take a value dc of type DC and
     *  and return a wrapped value of type class TC
     */
    var listOfFunctions: List<(DC) -> TC<DC>> = mutableListOf()
    listOfFunctions += ::mysqrt
    listOfFunctions += ::mylog
    listOfFunctions += ::myinv

    var dc = DC(100.0,"Start")
    inp = TC.Value(dc)
    println(inp.value.data)
    println(inp.value.rem)

    inp = execute(inp, listOfFunctions) as TC.Value<DC>

    println(inp.value.data)
    println(inp.value.rem)
    println()

    dc = DC(100.0,"Start")
    inp = TC.Value(dc)
    println(inp.value.data)
    println(inp.value.rem)

    var out = inp.flatMap(::mysqrt).flatMap(::myinv) as TC.Value
//    var out = inp flatMap ::mysqrt flatMap ::myinv
    println(out.value.data)
    println(out.value.rem)
    println()
}

fun <T> execute(input: TC<T>, fns: List<(T) -> TC<T>>): TC<T> =
        fns.fold(input) { inp, fn -> inp.flatMap(fn) }

fun main(args: Array<String>) {
    testMonad()

    println("Async")

    var listOfFunctions: List<(DC) -> TC<DC>> = mutableListOf()
    listOfFunctions += ::mysqrt
    listOfFunctions += ::mylog
    listOfFunctions += ::myinv

    var inp1: TC<DC> = TC.Value(DC(100.0,""))
    var inp2: TC<DC> = TC.Value(DC(200.0,""))

    runBlocking {
        val startTime = System.currentTimeMillis()
        val deferred1 = async { execute(inp1, listOfFunctions) }
        val value1 = deferred1.await() as TC.Value<DC>

        val deferred2 = async { execute(inp2, listOfFunctions) }
        val value2 = deferred2.await() as TC.Value<DC>

        println("${Thread.currentThread().name} : ${value1.value}")
        println("${Thread.currentThread().name} : ${value2.value}")

        val endTime = System.currentTimeMillis()

        println("Time taken: ${endTime - startTime}")
    }
}
```

6.6 Parallelization