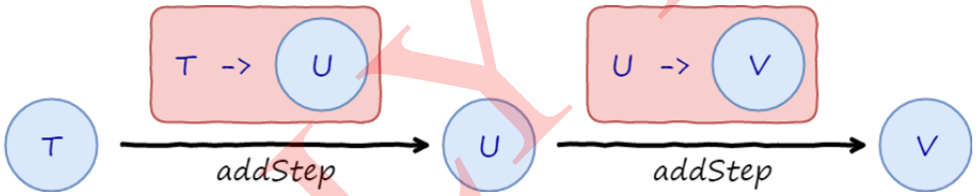
Functional Composition with Monads in Kotlin

Jaideep Ganguly, Sc.D.



Contents

Preface

This book has been written to help developers migrate to the functional composition style of coding and become at least 2x more productive. Functional Programming is based on sound mathematical concepts such immutability, referential transparency, functional composition and monads but can appear daunting.

The distinguished mathematicin, Prof. G.H. Hardy says, "A 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 ideas" – G H Hardy.

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

A Java developer should be able to migrate to Kotlin and be reasonably proficient in 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 prog, leverage type classes and write function compositions that may be executed asyncronously all within a week.

CHAPTER 1

The problem

1.1 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. For a piece of code with business logic of just 10 lines, it is not unusual to see developers write 100 lines in form of Interface, Abstract Classes, Concrete Classes, Factories, Abstract Factories, Builders and Managers. Making changes has become tedious, time consuming and error prone. Most of the assumptions around the need for extensibility is false and much of the code written for the sake of extensibility is never used.

In a typical Object Oriented Paradigm (OOP) Development methodology, developers write Interfaces, Classes, Managers and Factories to achieve what functions achieve in one file in a Functional Paradigm (FP) code. 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 are not lost in a typical FP code if composition is used with extensibility in mind. Moreover, a functional code would be devoid of complexities like Dependency Injection (DI), Mocking, and Mutations, making the code and its testing simpler to reason about, read and write.

1.2 Dependencies

Dependency Injection and Design Patterns that are widely prevalent in OOP, are not evil in themselves, in-fact when they came out, they were hailed as next-gen evolution in development. Quickly though, they became abused, with developers using them whenever they could, mostly to showcase their abilities. Such is the dire status 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!

Lots of dependencies 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. 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.

CHAPTER 2

Documentation

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.

2.1 Language of Abstraction

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

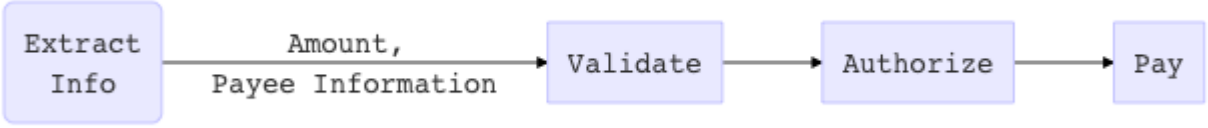
It is worth noting that final machine code is composed of a sequence of instructions depending on the processor and a higher level language such as C had just 32 key words in its first version. Whatever be the functionality from data base kernel to gaming, it had to be composed out of these limited set 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 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 documentas are written, mathematically, there can be never a unique one to one mapping between the document and the code. This is precisely why engineering displines have adopted engineering drawings along with a well defined set of symbols that convey the 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 good quality house.

2.2 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.

2.3 Data Flow diagrams with Typora & Mermaid

Data flows are best capured in a markdown editor that supports Mermaid. It is simple to model entities and activities using a markdown editor such as Typora. For example, consider the following data flow.



CHAPTER 3

Functional Programming

3.1 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 "lines of code".

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

3.2 OOP versus FP

```
Typical Java/OOP 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.

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.
```

3.3 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.

CHAPTER 4

Monad

4.1 Typeclass

Monad is a Typeclass

Listing 4.1 – Typeclass.

```
sealed class Monad<out A> {

    object None : Monad<Nothing>()
    data class Value<out A>(val value: A) : Monad<A>()

    // Monad - Apply a function to a wrapped value and return a wrapped
    // value using flatMap (liftM or >>= in Haskell)
    inline infix fun <B> flatMap(f: (A) -> Monad<B>) : Monad<B> =
            when (this) {
                is None  -> this
                is Value -> f(value)
            }
 }
```

4.2 Functions

Listing 4.2 – Functions.

```
import Monad.None
import Monad.Value


fun mysqrt(a: Double) = when {
    a >= 0 -> {
        Monad.Value(kotlin.math.sqrt(a))
    }
    else   -> {
        Monad.None
    }
}

fun mylog(a: Double) = when {
    a > 0  -> {
        Value(kotlin.math.ln(a))
    }
    else   -> {
        None
    }
}

fun myinv(a: Double) = when {
    a >= 0 -> {
        Value(1/a)
    }
    a <= 0 -> {
        Value(1/a)
    }
    else   -> {
        None
    }
}
```

4.3 Composition

Listing 4.3 – Class & Function.

```
import Monad.Value

fun testMonad() {
    var vin : Monad<Double>
    var vout: Monad<Double>

    var listOfFun: List<(Double) -> Monad<Double>> =
            mutableListOf<(Double)->Monad<Double>>(::mysqrt,::mylog)
    listOfFun += ::myinv


    vin = Value(100.0)

    // This is not composition
    var iter = listOfFun.iterator()
    while (iter.hasNext()) {
        println(vin.flatMap(iter.next()))
    }
    println()


    iter = listOfFun.iterator()
    while (iter.hasNext()) {
        vin = vin.flatMap(iter.next())      // Mutating
        println(vin)
    }
    println()

    // composition
    vin = Value(100.0)
    vout = vin.flatMap(::mysqrt).flatMap(::mylog).flatMap(::myinv)
    println(vout)

    // composition with infix
    vout = vin flatMap ::mysqrt flatMap ::mylog flatMap ::myinv
    println(vout)

    vin = Monad.Value(-100.0)
    vout = vin.flatMap(::mysqrt)
    println(vout)


    println(Value(100.00).flatMap(::mysqrt))
    println(Value(-100.00).flatMap(::mysqrt))

    println(Value(1000.0)
            .flatMap(::mysqrt)
            .flatMap(::mysqrt)
```