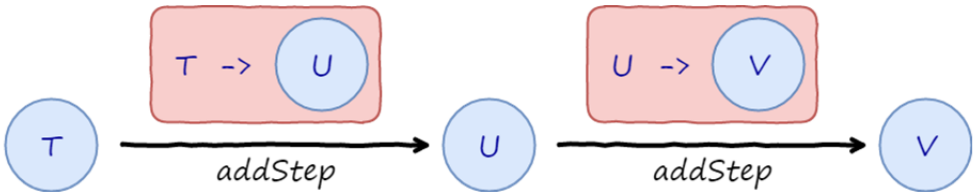
Functional Composition with Monads in Kotlin

Jaideep Ganguly, Sc.D.



Contents

CHAPTER 1

The problem

1.1 The Trouble with Objected Oriented Paradigm

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.

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!

1.2 Dependencies

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

Functional Programming

2.1 Functional Programming

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 after 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 functional programming, extensibility is acheived through the use of higher order functions.

2.2 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.
```

CHAPTER 3

Knowledge Graph

3.1 Data Flow

A knowledge graph depicts the data flow in a process. Data is encapsulated in a data class and flows in and out of an activity. One way to view it is a set of nouns flowing into and out of verbs. Activities are interfaces or a collection of functions.

3.2 Pure Functions

The functions are segregated into pure and impure functions. The functional core consists of pure functions and the outer shell is composed of impure functions.

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)
```

CHAPTER 5

Documentation

5.1 Typora & Mermaid

Data flows are best capured in a markdown editor that supports Mermaid. It is trival to model entities and activities using a markdown editor such as Typora.