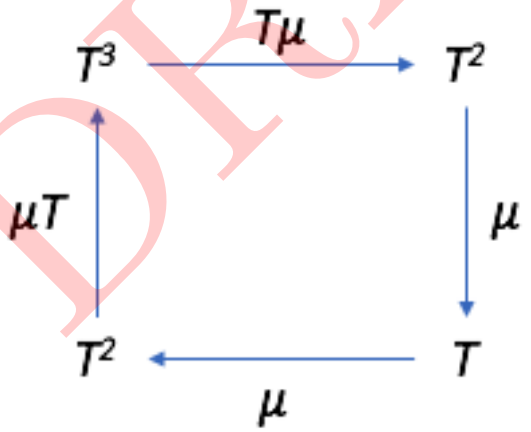
API design through Function Compositions and Type Classes

Jaideep Ganguly

August 22, 2019

DRAFT



Contents

1 Introduction **4**

2 The Trouble with Objected Oriented Paradigm **4**

2.1 Resusability & Extensibilty 4

2.2 Dependencies 5

2.3 Testing 5

3 Language for Functional Descriptions **5**

3.1 Business Processes and Data Flow 6

3.2 Kotlin 6

4 Functional Programming **7**

4.1 First Class Functions 7

4.2 Side Effects 7

4.3 Mutation 7

4.4 Referential Transparency 8

4.5 Closure 8

4.6 Pure Function 8

4.7 Tail Recursion 9

4.8 Lambda Expression 9

4.9 Higher Order Function 9

4.10 OOP versus FP 10

5 Modeling API behavior **10**

5.1 Pure and Impure Functions 10

6 Category Theory **11**

6.1 Type Class & Monad 12

6.2 Function Structure 13

6.3 Function Composition 14

6.4 Async Computing 14

7 Summary **15**

List of Figures

1 Data Flow diagram 6

DRAFT

2 Core of Pure Functions and Outer Shell of Impure Functions 11

Listings

1 Immutable 7
2 Immutable 8
3 Referential Transparency 8
4 Closure 8
5 Pure Function 8
6 Recursion 9
7 Tail Recursion 9
8 Lambda Expression 9
9 Higher Order Function 9
10 Typical Java OO pseudo-code 10
11 Typical FP pseudo-code 10
12 Payment Method 10
13 Type Class 12
14 Data Structure 13
15 Function 13
16 Input data wrapped in a Type Class 14
17 List of Functions 14
18 Function Composition 14
19 Function Composition 14
20 Function Composition 14
21 Asynchronous Computating 14

DRAFT

1 Introduction

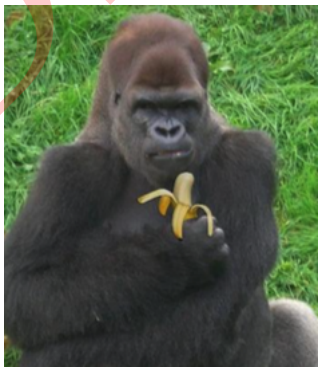
The purpose of this article is to help developers learn the art and science of designing APIs using the functional style of programming that will make their design robust while increasing their productivity by at least 100%. Functional Programming is based on sound mathematical principles such immutability, referential transparency, functional composition and monads that have persisted the test of time. But learning functional programming can be a daunting process as it brings with it unfamiliar terminology that may appear to be difficult but is worth understanding given the benefits.

The distinguished mathematician, Prof. G.H. Hardy once said, "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. But 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 pragmatic use in software engineering and demonstrate practical solutions through code samples.

Kotlin has been chosen as the language for implementing the ideas. It is 100% compatible with Java and its implementation is pragmatic. 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, should be used for developing in Kotlin. Thereafter, one should be able to easily migrate to the functional style of programming, leverage type classes and write function compositions, which are parallely executatble, all in just about a week.

2 The Trouble with Objected Oriented Paradigm

The Object Oriented Programming (OOP) paradigm typically results in obfuscating code. It is very common to see developers write layers and layers of abstractions. The code follows a verbose, iterative, if-then style to encode business logic. As a result, making changes become tedious, time consuming and error prone. For a business logic of 10 lines, it is not uncommon to see developers write 100 lines in the form of Interfaces, Abstract Classes, Concrete Classes, Factories, Abstract Factories, Builders and Managers! In general, an OOP code is typically about 50% larger than a functional code.



The problem with object-oriented languages is they've 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.

2.1 Resusability & Extensibilty

In reality, 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. In any case, reusability and extensibility concepts promoted by OOP through use of Interfaces and Factories are easily achievable through the use of higher order functions and function compositions.

DRAFT

2.2 Dependencies

Dependency injection (DI) is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used as a service. When class A uses some functionality of class B, then its said that class A has a dependency of class B. Transferring the task of creating the object to someone else and directly using the dependency is called dependency injection. Inversion of control which is the concept behind DI, states that a class should not configure its dependencies statically but should be configured by some other class from outside. DI adds a layer of abstraction between the developer and the 'new' keyword. When there are lots of dependencies, the benefits are noticable as DI eliminates the "wiring" code as DI builds and maintain a large object graph.

Dependency Injection and Design Patterns that are widely prevalent in OOP. With DI, many compile time errors are now pushed to run-time. DI frameworks are implemented with reflection which hinders use of IDE automation, such as "find references", "show call hierarchy" and safe refactoring. Since the object graph is built at run time, traditional debugging techniques are no longer effective . In summary, its overuse has lead to serious in problem management of software development.

An highly influential book on programming is "Structure and Interpretation of Computer Programs by Abel and Sussman of MIT". In there it is stated, *"In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs" [section 3.1.3]*. It strongly advocates functional programming as a powerful technique and DI is fundamentally a functional paradigm. The injector is in the business of functions and argument-lists, and treats them as first-class objects to pass around. With the control of the function passed to the injector, the injector can do much of the same high-level optimizations that are common in a functional language such as Lisp and Haskell.

2.3 Testing

Since the OOP style of programming uses lots of dependencies, it requires lots of testing and testing is not easy. 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. 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 α and β quality software that require a magnitude more of effort towards fixes. A functional code is devoid of complexities of DI, Mocking, and Mutations, making the code and its testing much easier.

3 Language for Functional Descriptions

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 through 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 constructed from 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

DRAFT

words, the combinatorial expressive power of which is explosive. More importantly, the semantic parsing of the phrases can often be ambiguous. Consider the 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 functionality desired. No much how much you try, an architect cannot build a house from a word document. Architectural drawings, structural drawings, electrical and HVAC drawings are essential to build a proper house. A building cannot be constructed out of descriptions provided in a word document.

In software development, one often encouner a multitude of documents or artifacts such as *Business Requirement Document*, *Product Requirement Document*, *High Level Design*, *Low Level Design* and so on. These artifacts are verbose, qualitative in nature, have varying degrees of rigor depending on the author and is more than often inadequate to describe the software correctly.

3.1 Business Processes and Data Flow

Business processes are essentially a flow of information or data encapsulated in *entities* between various *activities*. Examples of activities are validation, transformation, computation and so on. Entities are nouns while 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 which outputs entities.

Data flows are best captured in a *Markdown* editor that supports the *Javascript Mermaid* plugin. It is simple to model entities and activities using a markdown editor such as *Typora*. Figure 1 depicts an example of a typical data flow.

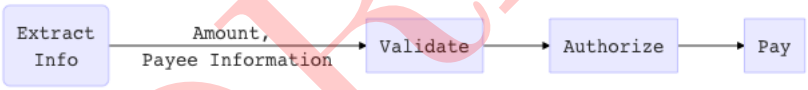


Figure 1: Data Flow diagram

Here the words inside the boxes are verbs that represent activities while entities which are nouns flow from one activity to another.

3.2 Kotlin

Kotlin has been used extensively in this article to demonstrate various concepts. 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

DRAFT

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.

4 Functional Programming

It should now be amply clear on the need to move from object dependencies to functional dependencies and ultimately evolve APIs towards behavior instead of objects.

Functional Programming (FP) 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 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 OOP. With concepts of higher order functions and composition, FP does away with all such layers.

4.1 First Class Functions

A function is a first-class citizen, i.e., it is just another value. Higher-order Functions either take functions as parameters, return functions or both.

4.2 Side Effects

A function or an expression is said to have a side effect if it modifies some state outside of its local environment, that is to say has an observable interaction with the outside world, besides returning a value. Modifying a non-local variable or an argument passed by reference and performing I/O are some examples of side effects. With side effects, a program's behaviour may depend on it execution history, i.e., the order of evaluation matters. Understanding and debugging a function with side effects requires knowledge about the context and its possible states and, therefore, can get very complicated.

In functional programming, side effects are rarely used. The lack of side effects makes it easier to do formal verifications of a program. Functional languages such as Scheme, Kotlin, Scala and Standard ML do not restrict side effects, but it is customary to avoid them. Haskell expresses side effects such as I/O and other stateful computations using monadic actions.

4.3 Mutation

Immutable means, once created, an object or a variable cannot be changed. That means, to change a property, you will have to copy (clone) the entire object. Writing mutable code comes naturally to us. We do not create a new bank account each time we make a deposit, we change the balance.

Immutable objects are thread safe. No race conditions, no concurrency problems, no need to sync and better readable code. But writing immutable code can mean cloning, cloning involves object creation and extra memory. Below is an example of immutable code:

Listing 1: Immutable

```
val listOfProduct = listOf(
     Product("A", 100),
     Product("B", 200),
     Product("C", 300)
)
val total: Int = listOfProduct.map { it.quantity }.sum()
```

DRAFT

4.4 Referential Transparency

Referential Transparency is a prerequisite for full composability of functions. In Functional Programming, Referential transparency is a property of expressions that allows the expression to be replaced by other expressions having the same value without changing the result in any way. Consider the following:

Listing 2: Immutable

```
int add(int a, int b) {
    return a + b
}
int mult(int a, int b) {
    return a * b;
}
int x = add(2, mult(3, 4));
```

The method is referentially transparent because any call to it may be replaced with the corresponding return value, i.e., can be replaced by . And so is referentially transparent. Now consider:

Listing 3: Referential Transparency

```
int add(int a, int b) {
    int result = a + b;
    System.out.println("Returning " + result);
    return result;
}
```

Replacing with will alter the program as the message will no longer be printed. Hence , in this case, is not referentially transparent.

4.5 Closure

In Kotlin, a lambda expression or anonymous function (as well as a local function and an object expression) can access the variables declared in the outer scope. Example:

Listing 4: Closure

```
var sum = 0
var ints = listOf(1,2,3,4)
ints.filter { it > 0 }.forEach {
    sum += it
}
```

4.6 Pure Function

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.

Listing 5: Pure Function

```
fun mySum(a:Int,b:Int):Int{
    return a+b
}
```

Pure functions are easy to test and lend themselves for parallel processing.

DRAFT

4.7 Tail Recursion

Examples of recurions using *fold* and *reduce*.

Listing 6: Recursion

```
val total = listOf(1, 2, 3, 4, 5).fold(0, { total, next -> total + next })
var mul = listOf(1, 2, 3, 4, 5).reduce({ mul, next -> mul * next })
```

Will excessive reliance on recursion cause stack overflows? That brings us to to the topic of Tail Recursion. A recursive function is eligible for tail recursion if the function call to itself is the last operation it performs. Since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use. The Kotlin (as well as many others, but not Java) compiler is smart enough to recognize this and effectively converts into a while loop and jumps to the called function. The compiler optimization for tail recursive functions removes the possibility of stack overflow. You have to mark the function with *tailrec* for the Kotlin compiler to perform tail recursion optimization.

Listing 7: Tail Recursion

```
tailrec fun fibonacci(n: Int, a: Long, b: Long): Long {
    return if (n == 0) b else fibonacci(n-1, a+b, a)
}
```

4.8 Lambda Expression

Lambda expression or simply lambda is an anonymous function; a function without name. One can pass them as arguments to methods, return them, or do any other thing we could do with a normal object.

Listing 8: Lambda Expression

```
val lambfunSum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
// or
val lambfunSum = { x: Int, y: Int -> x + y }
// and invoke as
var sum = lambfunSum(x,y)
```

4.9 Higher Order Function

In Kotlin, 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. Below is definition of a higher order function that takes another as a parameter.

Listing 9: Higher Order Function

```
fun sub(a: Int, b: Int, op: (Int, Int) -> Int, op2: (Int, Int) ->Int) : Int {
    val result: Int = subtr(a,b);
    return (result+result2)
}

fun subtr(p: Int, q: Int) : Int {
    return (p - q);
}

var result = sub(10,3, this::subtr)
```

DRAFT

4.10 OOP versus FP

Listing 10: Typical Java OO pseudo-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 11: Typical FP pseudo-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.
```

5 Modeling API behavior

Consider the following example of a higher order function *pay* that 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 12: Payment Method

```
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) {
    ...

}

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

DRAFT

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.

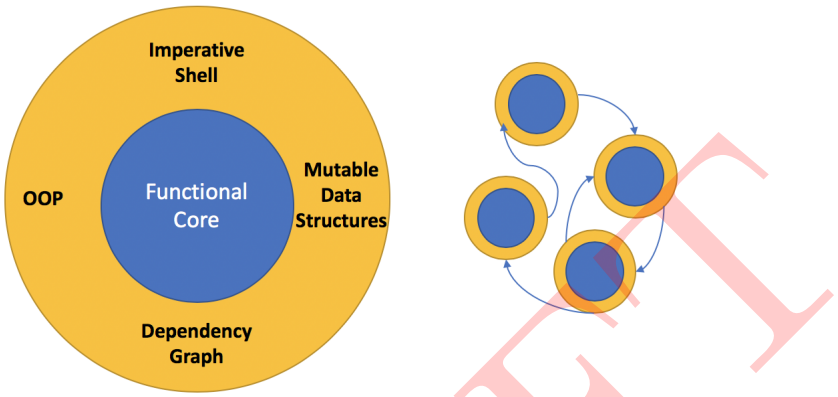
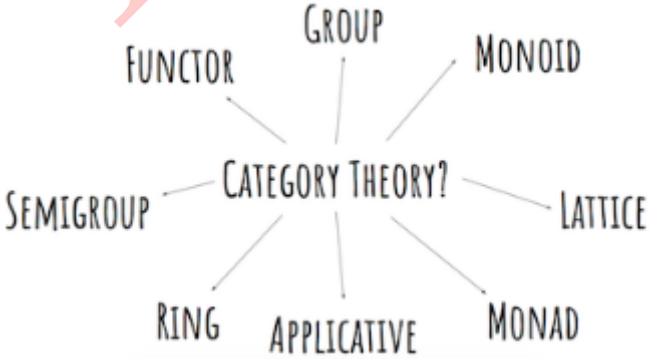


Figure 2: Core of Pure Functions and Outer Shell of Impure Functions

6 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 Monoid is a set that obeys certain rules. In a statically typed language, we can translate the notion of set into the notion of type.

A functor is simply a map between categories. An endofunctor is defined as a functor from one category back to the same category.



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.

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

DRAFT

6.1 Type Class & Monad

In computer science, function composition is an act or mechanism to combine simple functions to build more complicated ones. Now, when you apply a function to this value, you will get different results depending on the context. Furthermore, any exception in the composition must not upset the data flow. This is where the concept of Type Classes are useful. 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. We do not 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.

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.

We will now formalize mathematically valid structures and operations that guarantee correct result at all times. We will develop associate code so that future tasks become almost trivial.

Listing 13: Type Class

```
/** Type Class class is a type system construct that supports ad hoc
 *  polymorphism. This is achieved by adding constraints to type
 *  variables in parametrically polymorphic types. Such a constraint
 *  typically involves a type class T and a type variable a, and means
 *  that a can only be instantiated to a type whose members support the
 *  overloaded operations associated with T.
 *  The keyword out is necessary as otherwise line 22 will
 *  throw a type mismatch error at compile time
 */
sealed class TC<out A> {

    object None: TC<Nothing>()
    /** Value is actually a data class DC and
     *  value is dc i.e., an instance of the data class DC and
        extends the type class TC as below
     */
    data class Value<out A>(val value: A): TC<A>()

    /** Apply a function to a wrapped data, i.e. a type class
        and return a wrapped data using flatMap
        (liftM or >>= in Haskell)
     */
    inline infix fun <B> flatMap(f: (A) -> TC<B>): TC<B> = when (this) {
        is None  -> this
        is Value -> f(value)
    }
}
```

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

DRAFT

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.

We eliminate the issue of impedance mismatch by requiring that all functions that correspond to the *flatmap f* have the same argument type and returns that type wrapped in a type class. Consider the following data structure that is common to the functions in a Type Class.

Listing 14: Data Structure

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

6.2 Function Structure

Now consider three functions and take note of their structure, i.e. how the output is wrapped in a Type Class of the same data structure.

Listing 15: Function

```
fun mysqrt(a: DC) = when {
    a.data >= 0 -> {
        var y = kotlin.math.sqrt(a.data)
        a.data = y
        a.rem = a.rem + "mysqrt ok\n"
        println(y)
        TC.Value(a)
    }
    else -> {
        a.data = 0.0
        a.rem = a.rem + "mysqrt Error: Argument is negative\n"
        TC.Value(a)
    }
}

fun mylog(a: DC) = when {
    a.data > 0  -> {
        var y: Double = kotlin.math.ln(a.data)
        a.data = y
        a.rem = a.rem + "mylog ok\n"
        println(y)
        TC.Value(a)
    }
    else -> {
        a.data = 0.0
        a.rem = a.rem + "mylog Error: Argument is 0; "
        TC.Value(a)
    }
}

fun myinv(a: DC) = when {
    a.data > 0 -> {
    var y: Double = 1/(a.data)
    a.data = y
    a.rem = a.rem + "myinv ok\n"
    println(y)
    TC.Value(a)
}
else -> {
```

DRAFT

```
    a.data = 0.0
    a.rem = a.rem + "myinv Error: Denominator is 0; "
    TC.Value(a)
    }
}
```

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. Notice how the exceptions are handled in a function, the composition still works and error messages are cascaded even if individual functions fail.

Listing 16: Input data wrapped in a Type Class

```
var dc = DC(100.0,"Start\n")
inp = TC.Value(dc)
```

6.3 Function Composition

We next create the list of functions as follows.

Listing 17: List of Functions

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

With the helper function *execute*, we can now compose as follows:

Listing 18: Function Composition

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

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

Note that the above composition is identical to:

Listing 19: Function Composition

```
var out = inp.flatMap(::mysqrt)
             .flatMap(::mylog)
             .flatMap(::myinv) as TC.Value
```

Since the Type Class supports *infix* notation, we could also simply write:

Listing 20: Function Composition

```
var out = inp flatMap ::mysqrt flatMap ::mylog flatMap ::myinv
```

6.4 Async Computing

Function compositions can be easily invoked in an *async* fashion as shown below:

Listing 21: Asynchronous Computating

```
runBlocking {
    val startTime = System.currentTimeMillis()
    val deferred = async { execute(inp, listOfFunctions) }
```

DRAFT

```
    val value = deferred.await() as TC.Value<DC>
    println("${Thread.currentThread().name} : ${value.value}")
    val endTime = System.currentTimeMillis()
    println("Time taken: ${endTime - startTime}")
}
```

7 Summary

We have established a mathematically valid computational structure leveraging Type Classes and functional composition techniques that eliminates code complexity, smooth error handling and computationally efficient as it easily lends itself to *async* computation.

DRAFT