Table of Contents for Programming Scala, 2nd Edition

Let's return to functional programming and discuss some advanced concepts. You can skip this chapter if you're a beginner, but come back to it if you hear people using terms like *Algebraic Data Types*, *Category Theory*, and *Monads*.

The goal here is to give you a sense of what these concepts are and why they are useful without getting bogged down into too much theory and notation.

Algebraic Data Types

There are two common uses of the "ADT" acronym, *abstract data types* and *algebraic data types*, which is confusing. The former meaning is common in the object-oriented programming world. It includes our familiar friends like `Seq`, an abstraction for any of the sequential collections. In contrast, *algebraic data types* comes from the functional programming world. It may be less familiar, but it's equally important.

The term *algebraic data type* arises because the kinds of data types we'll discuss obey *algebraic*, i.e., mathematical properties. This is important because if we can prove properties about our types, it raises our confidence that they are bug-free and it promotes safe composition to create more complex data structures and algorithms.

Sum Types Versus Product Types

Scala types divide into *sum types* and *product types*.

Most of the classes you know are product types. When you define a `case class`, for example, how many unique instances can you have? Consider this simple example:

```
case class Person(name: Name, age: Age)
```

You can have as many instances of `Person` as the allowed `Name` instances *times* the allowed values for `age`. Let's say that `Name` encapsulates nonempty strings and disallows nonalphabetic characters (for some alphabet). There will effectively still be infinitely many values, but we'll say it's *N*. Similarly, `Age` limits integer values, let's say between 0 and 130. Why not use `String` and `Int`, respectively? We've been arguing all along that your types should express the allowed states and prevent invalid states, wherever possible.

Because we can combine any `Name` value with any `Age` value to create a `Person`, the number of possible `Person` instances is *131 * N*. For this reason, such types are called *product types*. Most of the types we're accustomed to using fall into this category.

It's also the source of the name for Scala's `Product` type, a parent of all `TupleN` types and case classes, which we learned about in Products, Case Classes, and Tuples.

We learned in Inferring Type Information that the single instance of `Unit` has the mysterious name `()`. This odd-looking name actually makes sense if we think of it as a zero-element tuple. Whereas a one-element tuple of `Int` values, `(Int)` or `Tuple1[Int]`, can have 2^{32} values, one for each of the possible `Int` values, a no-element tuple can only have one instance, because it can't carry any state.

Consider what happens if we start with a two-element tuple, `(Int, String)`, and construct a new type by

appending `Unit`:

```
type unitTimesTuple2 = (Int, String, Unit)
```

How many instances does this type have? It's exactly the same as the number of types that `(Int, String)` has. In product terms, it's as if we *multiplied* the number of types by 1. Hence, this is the origin of the name `Unit`, just as one is the "unit" of multiplication and zero is the unit of addition.

Is there a zero for product types? We need a type with zero instances: `scala.Nothing`. Combining `Nothing` with any other type to construct a new type must have zero instances because we don't have an instance to "inhabit" the `Nothing` field.

An example of sum types is enumerations. Recall this example from Chapter 3:

```
//
src/main/scala/progscala2/rounding/enumeration.sc

object Breed extends Enumeration {
  type Breed = Value
  val doberman = Value("Doberman Pinscher")
                       "Yorkshire
  val yorkie   = Value(Terrier"           )
                       "Scottish
  val scottie  = Value(Terrier"          )
                       "Great
  val dane     = Value(Dane"       )
                       "Portuguese Water
  val portie   = Value(Dog"                 )
}
```

There are exactly five instances. Note that the values are mutually exclusive. We can't have combinations of them (ignoring the realities of actual dog procreation). The breed is one and only one of these values.

Another way to implement a sum type is to use a sealed hierarchy of objects:

```
sealed trait Breed { val name: String }
case object doberman extends Breed { val name = "Doberman Pinscher" }
                                                 "Yorkshire
case object yorkie   extends Breed { val name = Terrier"            }
                                                 "Scottish
case object scottie  extends Breed { val name = Terrier"           }
                                                 "Great
case object dane     extends Breed { val name = Dane"        }
                                                 "Portuguese Water
case object portie   extends Breed { val name = Dog"
}
```

Tip

Use enumerations when you just need "flags" with an index and optional user-friendly strings. Use a sealed hierarchy of objects when they need to carry more state information.

Properties of Algebraic Data Types

In mathematics an *algebra* is defined by three aspects:

1. A set of *objects*: not to be confused with our the OO notion of objects. They could be numbers or almost anything.
2. A set of *operations*: how elements are combined to create new elements.
3. A set of *laws*: these are rules that define the relationships between operations and objects. For example, for numbers, *(x + (y + z)) == ((x + y) + z)* (associativity law).

Let's consider product types first. The informal arguments we made about the numbers of instances are more formally described by operations and laws. Consider again the *operation* of "adding" `Unit` to `(Int, String)`. We have a commutativity law:

```
Unit x (Int,String) == (Int,String) x Unit
```

This is true from the standpoint of the number of instances. It's analogous to *1 * N = N * 1* for any number *N*. This generalizes to combinations with nonunit types:

```
Breeds x (Int,String) == (Int,String) x Breeds
```

Just like *M * N = N * M* for any numbers *M* and *N*. Similarly, multiplication with "zero" (`Nothing`) commutes:

```
Nothing x (Int,String) == (Int,String) x Nothing
```

Turning to sum types, it's useful to recall that sets have unique members. Hence we could think of our allowed dog breeds as forming a set. That implies that adding `Nothing` to the set returns the same set. Adding `Unit` to the set would create a new set with all the original elements plus one, `Unit`. Similarly, if we added a nonunit type to the set, the new set would have all the instances of the additional type plus the original dog breeds. The same algebraic laws apply that we expect for addition:

```
Nothing + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Nothing
Unit    + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Unit
Person  + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Person
```

There is even a *distributive law* of the form *x*(a + b) = x*a + x*b*. I'll let you convince yourself that it actually works.

Final Thought on Algebraic Data Types

There are more properties we could explore, but I'll refer you to an excellent series of blog posts by Chris Taylor for more details.

What does all this have to do with programming? This kind of precise reasoning encourages us to examine our types. Do they have precise meanings? Do they constrain the allowed values to just those that make sense? Do they compose to create new types with precise behaviors?

My favorite example of "unexpected" precision is the `List` type in Scala. It is an abstract class with exactly two

concrete subtypes, `Nil` and `::`, the nonempty list. It might sound trivial to say a list is either empty or it isn't, but with these two types we can construct all lists and reason precisely about how all lists must behave.

Category Theory

Perhaps the most controversial debate in the Scala community is how much to embrace *Category Theory*, a branch of mathematics, as a source of what I'll argue are *Functional Design Patterns*. This section introduces you to the basic ideas of Category Theory and to a few of the actual *categories* most commonly used in functional programming. They are powerful tools, at least for those development teams willing to master them.

The use of Category Theory is controversial because of its intimidating mathematical foundation. Accessible documentation has been hard to find. Category Theory generalizes all of mathematics in ways that enable reasoning about global properties. Hence, it offers deep and far-reaching abstractions, but when applied to code, many developers struggle with extreme abstraction. It's easier for most of us to understand code with concrete, specific details, but we already know that abstractions are useful, as well. Striking a balance is key and where you feel comfortable striking that balance will determine whether or not Category Theory works for you.

However, Category Theory now occupies a central place in advanced functional programming. Its use was pioneered in Haskell to solve various design problems and to push the envelope of functional thinking. Implementations of the common categories are now available in most functional languages.

If you are an advanced Scala developer, you should learn the rudiments of Category Theory as applied to programming, then decide whether or not it is right for your team and project. Unfortunately, I've seen situations where libraries written by talented proponents of Category Theory have failed in their organizations, because the rest of the team found the libraries too difficult to understand and maintain. If you embrace Category Theory, make sure you consider the full life cycle of your code and the social aspects of development.

Scalaz (pronounced "Scala Zed") is the primary library for Scala that implements categories. It is a good vehicle for learning and experimentation. We already used its `Validation` type in Scalaz Validation. I'll use simplified category implementations in this chapter to minimize what's new to learn.

In a sense, this section continues where Higher-Kinded Types left off. There, we discussed abstracting over parameterized types. For example, if we have a method with `Seq[A]`, can we generalize it to `M[A]` instead, where `M` is a type parameter for any type that is itself parameterized by a single type parameter? Now we'll abstract over the functional *combinators*, `map`, `flatMap`, and so forth.

About Categories

Let's start with the general definition of a *category*, which contains three "entities" (recall our discussion of algebra):

1. A *class* consisting of a set of *objects*. These aren't the same terms from OOP, but they have similar implications.
2. A *class* of *morphisms*, also called *arrows*. A generalized notion of functions and written *f: A -> B* (`f: A => B` in Scala). For each morphism, *f*, one object is the *domain* of *f* and one is the *codomain*. It might seem strange to use the singular "object," but in some categories each *object* is itself a collection of values or another category.
3. A *binary operation* called *morphism composition* with the property that for *f: A -> B* and *g: B -> C*, the composition *g ∘ f: A -> C* exists.

Two axioms are satisfied by morphism composition:

1. Each object *x* has one and only one identity morphism, i.e., where the domain and codomain are the same, ID_X and composition with identity has the following property: $f \circ ID_X = ID_X \circ f$.
2. *Associativity*. For *f: A -> B*, *g: B -> C*, *h: C -> D*, $(f \circ g) \circ h = f \circ (g \circ h)$.

The categories we'll discuss next have these properties and laws. We'll look at just two categories that are used in software development (of the many categories known to mathematics), *Functor* and *Monad*. We'll also mention two more, *Applicative* and Arrow.

The Functor Category

Functor abstracts the `map` operation. We introduced it in Type Lambdas to set up an example where a *type lambda* was necessary. We're going to implement it in a slightly different way here, first defining the abstraction and then implementing it for three concrete types, `Seq`, `Option`, and `A => B` :

```
// src/main/scala/progscala2/fp/categories/Functor.scala
package progscala2.fp.categories
import scala.language.higherKinds

trait Functor[F[_]] {                                                    //
❶
  def map[A, B](fa: F[A])(f: A => B): F[B]                              // ❷
}

object SeqF extends Functor[Seq] {                                       //
❸
  def map[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map f
}

object OptionF extends Functor[Option] {
  def map[A, B](opt: Option[A])(f: A => B): Option[B] = opt map f
}

object FunctionF {
// ❹
  def map[A,A2,B](func: A => A2)(f: A2 => B): A => B = {              // ❺
    val functor = new Functor[({type λ[β] = A => β})#λ] {              // ❻
      def map[A3,B](func: A => A3)(f: A3 => B): A => B = (a: A) => f(func(a
))
    }
    functor.map(func)(f)                                                 //
❼
  }
}
```

❶

In contrast to the previous version in Type Lambdas, this `map` method takes an instance of an `F` (which will be some kind of container) as an argument. Also, we don't use the name `map2` here, like we did before.

❷

The `map` arguments are the functor `F[A]`, a function `A => B` . An `F[B]` is returned.

❸

Define implementation objects for `Seq` and `Option`.

❹

Define an implementation object for mapping one function to another; not easy!

❺

`FunctionF` defines its own `map` method, constructed so invoking `map` has the same syntax for functions as it does for `Seq`, `Option`, and any other transformations we might implement. This `map` takes the initial function we're transforming and the function that does the transformation. Note the types: we're converting a `A2` `A =>` to `B` `A =>`, which means the second function argument `f` is `B` `A2 =>`. In other words, we're *chaining* functions.

❻

The implementation of `map` constructs a `Functor` with the correct types to do the transformation.

❼

Finally, `FunctionF.map` invokes the `Functor`. The return value of `FunctionF.map` is `B` `A =>`.

`FunctionF` is nontrivial. To understand it, keep in mind that we aren't changing the initial type `A`, just chaining a second function call that takes the `A2` output of `func` and calls `f` with it.

Let's try these types:

```
// src/main/scala/progscala2/fp/categories/Functor.sc
import progscala2.fp.categories._
import scala.language.higherKinds

val fii: Int => Int       = i => i * 2
val fid: Int => Double    = i => 2.1 * i
val fds: Double => String = d => d.toString

SeqF.map(List(1,2,3,4))(fii)
// Seq[Int]: List(2, 4, 6,
8)
                                          // Seq[Int]:
SeqF.map(List.empty[Int])(fii)            List()

OptionF.map(Some(2))(fii)                 // Option[Int]: Some(4)
OptionF.map(Option.empty[Int])(fii)       // Option[Int]: None

val fa = FunctionF.map(fid)(fds)
// ❶
                                          // String:
fa(2)                                     4.2

// val fb = FunctionF.map(fid)(fds)                                    ❷
val fb = FunctionF.map[Int,Double,String](fid)(fds)
fb(2)

val fc = fds compose fid
// ❸
                                          // String:
fc(2)                                     4.2
```

❶

Chain `Int =>` `Double` and `Double =>` `String` functions together, creating a new function, then call it on the next line.

❷

Unfortunately, the argument types can't be inferred, so explicit types are required in the function literals or on `FunctionF.map`.

❸

Note that `FunctionF.map(f1)(f2) == f2 compose f1` , not `f1 compose f2` !

So, why is the parameterized type with a `map` operation called a "Functor"? Let's look at the `map` declaration again. We'll redefine it with `Seq` for simplicity, then again with the argument lists switched:

```
       def map[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map
scala> f

       def map[A, B](f: A => B)(seq: Seq[A]): Seq[B] = seq map
scala> f
```

Now note the type of the new function returned when we use partial application on the second version:

```
scala> val fm = map((i: Int) => i * 2.1) _
fm: Seq[Int] => Seq[Double] = <function1>
```

So, this `map` method *lifts* a function `A => B` to `Seq[A] => Seq[B]`! In general, `Functor.map` morphs `A => B`, for all types `A` and `B`, to `F[A] => F[B]` for many `F` (`F` has to be a category itself). Put another way, *Functor* allows us to apply a pure function (`f: A => B`) to a "context" holding one or more `A` values. We don't have to extract those values ourselves to apply `f`, then put the results into a new instance of the "context." The term *Functor* is meant to capture this abstraction of enabling the use of pure functions in this way.

In Category Theory terms, other categories are the objects and the morphisms are the mapping between categories. For example, `List[Int]` and `List[String]` would be two categories whose own objects are all possible lists of `Int`s and `String`s, respectively.

`Functor` has two additional properties that fall out of the general properties and axioms for Category Theory:

1. A Functor `F` preserves identity. That is, the identity of the *domain* maps to the identity of the *codomain*.
2. A Functor `F` preserves composition. *F(f ∘ g) = F(f) ∘ F(g)*.

For an example of the first property, an empty list is the "unit" of lists; think of what happens when you concatenate it with another list. Mapping over an empty list always returns a new empty list, possibly with a different list element type.

Are the common and `Functor`-specific axioms satisfied? The following *ScalaCheck* property test verifies them:

```
// src/test/scala/progscala2/fp/categories/FunctorProperties.scala
package progscala2.fp.categories
import org.scalatest.FunSpec
import org.scalatest.prop.PropertyChecks

class FunctorProperties extends FunSpec with PropertyChecks {

  // Lift identity method to a function
  def id[A] = identity[A] _

  def testSeqMorphism(f2: Int => Int) = {                          // ❶
    val f1: Int => Int = _ * 2
    import SeqF._
    forAll { (l: List[Int]) =>
      assert( map(map(l)(f1))(f2) === map(l)(f2 compose f1) )
    }
  }

  def testFunctionMorphism(f2: Int => Int) = {                     // ❷
    val f1: Int => Int = _ * 2
    import FunctionF._
    forAll { (i: Int) =>
      assert( map(f1)(f2)(i) === (f2 compose f1)(i) )              // ❸
```

```
  }
}

          "Functor morphism
describe (composition"                  ) {                          // ❹
      "works for Sequence
  it (Functors"                     ) {
    testSeqMorphism(_ + 3)
  }
      "works for Function
  it (Functors"                     ) {
    testFunctionMorphism(_ + 3)
  }
}

          "Functor identity composed with a another function
describe (commutes"                                                    ) {
      "works for Sequence
  it (Functors"                     ) {                              // ❺
    testSeqMorphism(id[Int])
  }
      "works for Function
  it (Functors"                     ) {
    testFunctionMorphism(id)
  }
}

          "Functor identity maps between the identities of the
describe (categories"                                                  ) {
      "works for Sequence
  it (Functors"                     ) {                              // ❻
    val f1: Int => String = _.toString
    import SeqF._
    assert( map(List.empty[Int])(f1) === List.empty[String] )
  }
      "works for Function
  it (Functors"                     ) {
    val f1: Int => Int = _ * 2
                                // Lift method to a
    def id[A] = identity[A] _   function
    import FunctionF._
    forAll { (i: Int) =>
      assert( map(id[Int])(f1)(i) === (f1 compose id[Int])(i) )
    }
  }
}

          "Functor morphism composition is
describe (associative"                                 ) {
      "works for Sequence
  it (Functors"                     ) {                              // ❼
    val f1: Int => Int = _ * 2
    val f2: Int => Int = _ + 3
    val f3: Int => Int = _ * 5
    import SeqF._
    forAll { (l: List[Int]) =>
      val m12 = map(map(l)(f1))(f2)
```

```
        val m12 = map(map(f)(f1))(f2)
        val m23 = (seq: Seq[Int]) => map(map(seq)(f2))(f3)
        assert( map(m12)(f3) === m23(map(l)(f1)) )
      }
    }
        "works for Function
    it (Functors"                    ) {
      val f1: Int => Int = _ * 2
      val f2: Int => Int = _ + 3
      val f3: Int => Int = _ * 5
      val f:  Int => Int = _ + 21
      import FunctionF._
      val m12 = map(map(f)(f1))(f2)
      val m23 = (g: Int => Int) => map(map(g)(f2))(f3)
      forAll { (i: Int) =>
        assert( map(m12)(f3)(i) === m23(map(f)(f1))(i) )
      }
    }
  }
}
```

❶

A helper function that verifies morphism composition for `SeqF`. Essentially, does mapping the `Functor` object once with one function then mapping the output with the second function produce the same result as composing the functions and then mapping once?

❷

A similar helper function that verifies morphism composition for `FunctionF`.

❸

Note that we “morph” the functions, then verify they produce equivalent results by applying to a set of generated `Int` values.

❹

Verify morphism composition for both `SeqF` and `FunctionF`.

❺

Verify the identity property for both `SeqF` and `FunctionF`.

❻

Verify the `Functor`-specific axiom that identities are mapped to identities.

❼

Verify the `Functor`-specific axiom of associativity of morphisms.

Back to programming, is it of practical use to have a separate abstraction for `map`, given the sophistication of this code? In general, abstractions with mathematically provable properties enable us to reason about program structure and behavior. For example, once we had a generalized abstraction for mapping, we could apply it to many different data structures, even functions. This reasoning power of Category Theory is being applied in several areas of Computer Science research.

The Monad Category

If `Functor` is an abstraction for `map`, is there a corresponding abstraction for `flatMap`? Yes, *Monad*, which is named after the term *monas* used by the Pythagorean philosophers of ancient Greece that roughly translates "the Divinity from which all other things are generated."

Here is our definition of Monad:

```
// src/main/scala/progscala2/fp/categories/Monad.scala
package progscala2.fp.categories
import scala.language.higherKinds

trait Monad[M[_]] {                                                    // ❶
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]                      // ❷
  def unit[A](a: => A): M[A]                                           // ❸

  // Some common aliases:                                              ❹
  def bind[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def >>=[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def pure[A](a: => A): M[A] = unit(a)
                                                  // backticks to avoid
  def `return`[A](a: => A): M[A] = unit(a)        keyword
}

object SeqM extends Monad[Seq] {
  def flatMap[A, B](seq: Seq[A])(f: A => Seq[B]): Seq[B] = seq flatMap f
  def unit[A](a: => A): Seq[A] = Seq(a)
}

object OptionM extends Monad[Option] {
  def flatMap[A, B](opt: Option[A])(f: A => Option[B]):Option[B]= opt flatMap
f
  def unit[A](a: => A): Option[A] = Option(a)
}
```

❶

Use `M[_]` for the type representing a data structure with "monadic" properties. As for `Functor`, it takes a single type parameter.

❷

Note that the function `f` passed to `flatMap` has the type `A => M[B]`, not `A => B`.

❸

Monad has a second function that takes a (by-name) value and returns it inside a Monad instance. In Scala, this is typically implemented with constructors and case class `apply` methods.

❹

Mathematics and other programming languages use different terms. The `>>=` and `return` names are the standard in Haskell. However, in Scala both names are problematic. The `=` at the end of `>>=` causes funny behavior due to the operator precedence of `=`. The name `return` collides with the keyword, unless escaped as shown.

Sometimes an abstraction with just `flatMap`, a.k.a. `bind`, is called *Bind*.

More commonly, an abstraction with just `unit` or `pure` is called *Applicative*. Note how `unit` resembles a case class `apply` method, where a value is passed in and an enclosing instance of a "wrapper" type is returned! Applicative is very interesting as an abstraction over construction. Recall from Constraining Allowed Instances and CanBuildFrom how `CanBuildFrom` is used in the collections library to construct new collection instances. Applicative is an alternative, if less flexible.

Let's try our Monad implementation:

```
// src/main/scala/progscala2/fp/categories/Monad.sc
import progscala2.fp.categories._
import scala.language.higherKinds

val seqf: Int => Seq[Int] = i => 1 to i
val optf: Int => Option[Int] = i => Option(i + 1)

SeqM.flatMap(List(1,2,3))(seqf)
// Seq[Int]: List(1,1,2,1,2,3)
                                          // Seq[Int]:
SeqM.flatMap(List.empty[Int])(seqf)       List()

OptionM.flatMap(Some(2))(optf)            // Option[Int]: Some(3)
OptionM.flatMap(Option.empty[Int])(optf)  // Option[Int]: None
```

One way to describe `flatMap` is that it extracts an element of type `A` from the container on the left and *binds* it to a new kind of element in a new container instance, hence the alternative name. Like `map`, it removes the burden of knowing how to extract an element from `M[A]`. However, it looks like the function argument now has the burden of knowing how to construct a new `M[B]`. Actually, this is not an issue, because `unit` can be called to do this. In an OO language like Scala, the actual Monad type returned could be a subtype of `M`.

The *Monad Laws* are as follows.

`unit` behaves like an identity (so it's appropriately named):

```
flatMap(unit(x))(f) == f(x)      Where x is a value
flatMap(m)(unit) == m            Where m is a Monad
instance
```

Like morphism composition for `Functor`, flat mapping with two functions in succession behaves the same as flat mapping over one function that is constructed from the two functions:

```
flatMap(flatMap(m)(f))(g) == flatMap(m)(x => flatMap(f(x))
(g))
```

The code examples contain a property test to verify these properties (see *src/test/scala/progscala2/toolslibs/fp/MonadProperties.scala*).

The Importance of Monad

Ironically, `Functor` is more important to Category Theory than Monad, while its application to software is somewhat trivial compared to Monad's impact.

In essence, Monad is important because it gives us a principled way to wrap context information around a value, then propagate and evolve that context as the value evolves. Hence, it minimizes coupling between the values and contexts while the presence of the Monad wrapper informs the reader of the context's existence.

This "pattern" is used frequently in Scala, inspired by the pioneer usage in Haskell. We saw several examples in Options and Other Container Types, including `Option`, `Either`, `Try`, and `scalaz.Validation`.

All are *monadic*, because they support `flatMap` and construction (case class `apply` methods instead of `unit`). All can be used in `for` comprehensions. All allow us to sequence operations and handle failures in different ways, usually through returning a subclass of the parent type.

Recall the simplified signature for `flatMap` in `Try`:

```
sealed abstract class Try[+A] {
  def flatMap[B](f: A => Try[B]): Try[B]
}
```

It's similar in the other types. Now consider processing a sequence of steps, where the previous outcome is fed into the next step, but we stop processing at the first failure:

```
// src/main/scala/progscala2/fp/categories/for-tries-
steps.sc

import scala.util.{ Try, Success, Failure }

type Step = Int => Try[Int]                                    //
❶

val successfulSteps: Seq[Step] = List(                         //
❷
  (i:Int) => Success(i + 5),
  (i:Int) => Success(i + 10),
  (i:Int) => Success(i + 25))
val partiallySuccessfulSteps: Seq[Step] = List(
  (i:Int) => Success(i + 5),
  (i:Int) => Failure(new RuntimeException("FAIL!")),
  (i:Int) => Success(i + 25))

def sumCounts(countSteps: Seq[Step]): Try[Int] = {             // ❸
  val zero: Try[Int] = Success(0)
  (countSteps foldLeft zero) {
    (sumTry, step) => sumTry flatMap (i => step(i))
  }
}

sumCounts(successfulSteps)
// Returns: scala.util.Try[Int] =
Success(40)

sumCounts1(partiallySuccessfulSteps)
// Returns: scala.util.Try[Int] = Failure(java.lang.RuntimeException:
FAIL!)
```

❶

Alias for “step” functions.

❷

Two sequences of steps, one successful, one with a failed step.

❸

A method that works through a step sequence, passing the result of a previous step to the next step.

The logic of `sumCounts` handles sequencing, while `flatMap` handles the `Try` containers. Note that subtypes are actually returned, either `Success` or `Failure`. We’ll see that `scala.concurrent.Future` is also monadic in Futures.

The use of Monad was pioneered in Haskell,[] where functional purity is more strongly emphasized. For example, Monads are used to compartmentalize input and output (I/O) from pure code. The `IO` Monad handles this *separation of concerns*. Also, because it appears in the type signature of functions that use it, the reader and compiler know that the function isn’t pure. Similarly, `Reader` and `Writer` Monads have been defined in many languages for the same purposes.

A generalization of Monad is *Arrow*. Whereas Monad lifts a *value* into a context, i.e., the function passed to `flatMap` is `A => M[B]`, an *Arrow* lifts a function into a context, `(A => B) => C[A => B]`. Composition of Arrows makes it possible to reason about sequences of processing steps, i.e., `A => B`, then `B => C`, etc., in a referentially transparent way, outside the context of actual use. In contrast, a function passed to `flatMap` is explicitly aware of its context, as expressed in the return type!

Recap and What's Next

I hope this brief introduction to more advanced concepts was informative enough to help you understand a few ideas you'll hear people mention, and why they are powerful, if also challenging to understand.

Scala's standard library uses an object-oriented approach to add functions like `map`, `flatMap`, and `unit`, rather than implementing categories. However, with methods like `flatMap`, we get "monadic" behaviors that make `for` comprehensions so concise.

I've casually referred to *Monad*, *Functor*, *Applicative*, and *Arrow* as examples of *Functional Design Patterns*. While the term has a bad connotation for some functional programmers, overuse of patterns in the OOP world doesn't invalidate the core idea of reusable constructs, in whatever form they take.

Unfortunately, categories have been steeped in mystery because of the mathematical formalism and their names, which are often opaque to ordinary developers. But distilled to their essence, they are abstractions of familiar concepts, with powerful implications for program correctness, reasoning, concision, and expressiveness. Hopefully, these concepts will become more accessible to a wider range of developers.

Appendix A lists some books, papers, and blog posts that explore functional programming further. A few are worth highlighting here. Two other functional structures that you might investigate are *Lenses*, for getting or setting (with cloning) a value nested in an instance graph, and *Monad Transformers*, for composing Monads.

Functional Programming in Scala, by Paul Chiusano and Rúnar Bjarnason (Manning) is a great next step for pushing your FP knowledge further with lots of exercises in Scala. The authors are some of the main contributers to Scalaz. Eugene Yokota wrote an excellent series of blog posts on learning Scalaz.

Shapeless also explores advanced constructs, especially in the type system. The aggregator site *http://typelevel.org* has some other instructive projects.

The next chapter returns to more practical grounds, the very important topic of writing concurrent software with Scala.