MONAD FACT #5

A chain of **monadic flatMap** calls (or an equivalent **for-comprehension**) is like an **imperative program** with **statements** that **assign** to **variables** and the **monad** specifies what occurs at **statement boundaries**

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] \arrow[dr, phantom, "\text{FACT \#3}"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

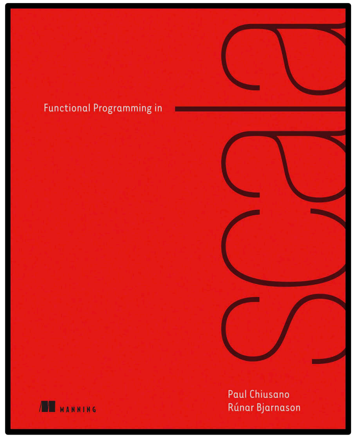
slides by

@philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

The title of this slide deck comes straight from **Functional Programming in Scala** and the aim of the deck is purely to emphasize a concept that is explained therein.

Hopefully the slides make the concept more vivid for you and reinforce your grasp of it.

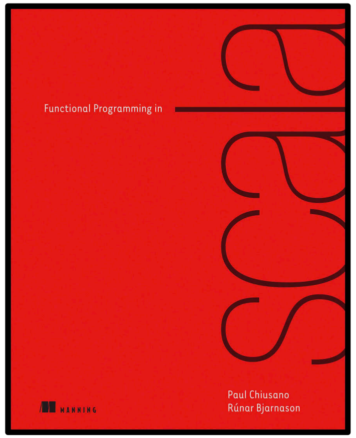


Functional Programming in Scala

11.5.1 The identity monad

To distill **monads** to their essentials, let's look at the simplest interesting specimen, the **identity monad**

```
trait Monad[F[_]] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
}

case class Id[A](value: A) {
  def map[B](f: A => B): Id[B] = Id(f(value))
  def flatMap[B](f: A => Id[B]): Id[B] = f(value)
}

object Id {
  val idMonad = new Monad[Id] {
    def unit[A](a: => A): Id[A] = Id(a)
    def flatMap[A,B](ma: Id[A])(f: A => Id[B]): Id[B] = ma flatMap f
  }
}
```

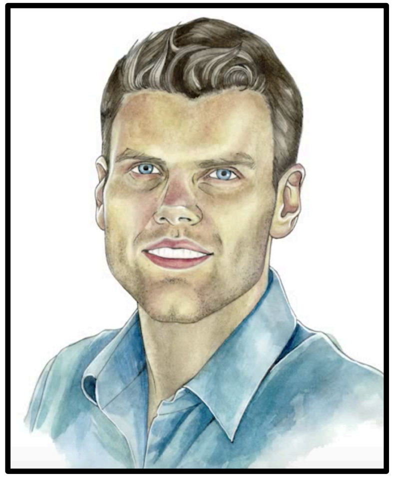


Functional Programming in Scala

Now, **`Id`** is just a simple wrapper. It doesn't really add anything. Applying **`Id`** to `A` is an **identity** since the wrapped type and the unwrapped type are totally **isomorphic** (we can go from one to the other and back again without any loss of information).

Paul Chiusano

@pchiusano

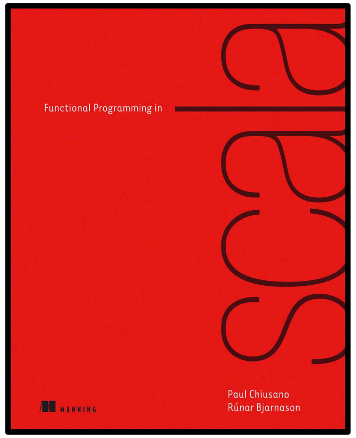


Runar Bjarnason

@runarorama

But **what is the meaning of the identity monad**? Let's try using it in the REPL:

```
scala>  Id("Hello, ") flatMap (a =>
     |     Id("monad!") flatMap (b =>
     |       Id(a + b)))
res0: Id[String] = Id(Hello, monad!)
```



Functional Programming in Scala

When we write the exact same thing with a **for-comprehension**, it might be clearer:

```
scala>  for {
     |    a <- Id("Hello, ")
     |    b <- Id("monad!")
     |  } yield a + b
res1: Id[String] = Id(Hello, monad!)
```

So **what is the action of `flatMap` for the identity monad**? **It's simply variable substitution**. **The variables `a` and `b` get bound to `"Hello, "` and `"monad!"`, respectively, and then substituted into the expression `a + b`**. We could have written the same thing without the `Id` wrapper, using just **Scala**'s own variables:

```
scala> val a = "Hello, "
a: String = "Hello, "

scala> val b = "monad!"
b: String = monad!

scala> a + b
res2: String = Hello, monad!
```

Besides the `Id` wrapper, there's no difference.

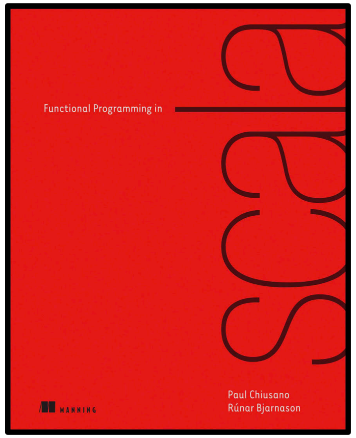
So now we have at least **a partial answer to the question of what monads mean**. We could say that **monads provide a context for introducing and binding variables, and performing variable substitution**.

Let's see if we can get the rest of the answer.

11.5.2 The State monad and partial type application

Look back at the discussion of the **State** data type in chapter 6. Recall that we implemented some combinators for **State**, including **map** and **flatMap**.

```
case class State[S, A](run: S => (A, S)) {

  def map[B](f: A => B): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      (f(a), s1)
    })

  def flatMap[B](f:A => State[S, B]): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      f(a).run(s1)
    })
}
```



Functional Programming in Scala

It looks like **State** definitely fits the profile for being a **monad**. But its type constructor takes two type arguments, and **Monad** requires a type constructor of one argument, so we can't just say `Monad[State]`. But if we choose some particular `S`, then we have something like `State[S, _]`, which is the kind of thing expected by **Monad**. So **State** doesn't just have one **monad** instance but a whole family of them, one for each choice of `S`. We'd like to be able to partially apply **State** to where the `S` type argument is fixed to be some concrete type. This is much like how we might partially apply a function, except at the type level.

FPiS then goes on to find a solution to the problem of partially applying `State[S,A]` so that the S type argument is fixed to be some concrete type, whereas the A argument remains variable.

The solution consists of using a type lambda that looks like this

```
({type f[x] = State[S,x]})#f
```

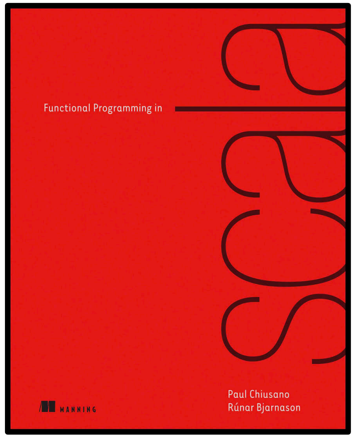
Explaining type lambdas is out of scope for this slide deck, but as **FPiS** says, their syntax can be jarring when you first see it, so here is the **Scala** 3 equivalent (dotty 0.22.0-RC1) of the above lambda, which is much easier on the eye:

```
[A] =>> State[S,A]
```

In the next slide I have replaced the **Scala** 2 type lambda with the **Scala** 3 equivalent.

A type constructor declared inline like this is often called a **type lambda** in **Scala**. We can use this trick to partially apply the `State` type constructor and declare a `StateMonad` trait. An instance of `StateMonad[S]` is then a **monad** instance for the given state type `S`:

```
def stateMonad[S] = new Monad[[A] =>> State[S,A]] {

  def unit[A](a: => A): State[S,A] =
    State(s => (a, s))

  def flatMap[A,B](st: State[S,A])(f: A => State[S,B]): State[S,B] =
    st flatMap f
}
```



Functional Programming in Scala

Again, just by giving implementations of `unit` and `flatMap`, we get implementations of all the other **monadic** combinators for free.

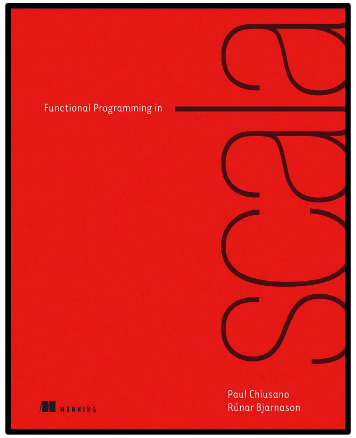
Let's now look at the difference between the `Id` monad and the `State` monad. Remember that the primitive operations on `State` (besides the **monadic** operations `unit` and `flatMap`) are that we can read the current state with `getState` and we can set a new state with `setState`:

```
def getState[S]: State[S, S]
def setState[S](s: => S): State[S, Unit]
```

Remember that we also discovered that **these combinators constitute a minimal set of primitive operations** for `State`. So **together with the monadic primitives (`unit` and `flatMap`) they completely specify everything that we can do with the `State` data type**. **This is true in general for monads**—**they all have `unit` and `flatMap`**, **and each monad brings its own set of additional primitive operations that are specific to it**.

What does this tell us about the meaning of the `State` monad? Let’s study a simple example. The details of this code aren’t too important, but notice the use of `getState` and `setState` in the `for` block.

```
val F = stateMonad[Int]

def zipWithIndex[A](as: List[A]): List[(Int,A)] =
  as.foldLeft(F.unit(List[(Int, A)]()))((acc,a) => for {
    xs <- acc
    n  <- getState
    _  <- setState(n + 1)
  } yield (n, a) :: xs).run(0)._1.reverse
```



Functional Programming in Scala

This function numbers all the elements in a list using a `State` action. It keeps a state that’s an `Int`, which is incremented at each step. We run the whole composite `State` action starting from 0. We then reverse the result since we constructed it in reverse order.

Note what’s going on with `getState` and `setState` in the for-comprehension. We’re obviously getting variable binding just like in the `Id` monad—we’re binding the value of each successive state action (`getState`, `currentStateAction`, and then `setState`) to variables. But there’s more going on, literally between the lines. At each line in the for-comprehension, the implementation of `flatMap` is making sure that the current state is available to `getState`, and that the new state gets propagated to all actions that follow a `setState`.

If you are not very familiar with the **State** monad then you might like some help to fully understand how the code on this slide works.
While **FPiS** says that the details of the code aren’t too important, I think this is a really useful example of the **State monad** and so in order to aid its comprehension

1) the next slide is the equivalent of this one but with the **FPiS** code replaced with a more verbose version in which I have a go at additional naming plus alternative or more explicit naming
2) the slide after that consists of the verbose version of the code plus all the **State** code that it depends on

What does this tell us about the meaning of the **`State` monad**? Let's study a simple example. The details of this code aren't too important, but notice the use of `getState` and `setState` in the `for` block.

```
def zipWithIndex[A](as: List[A]): List[(Int,A)] =
  val emptyIndexedList = List[(Int, A)]()
  val initialStateAction = stateMonad[Int].unit(emptyIndexedList)
  val finalStateAction =
    as.foldLeft(initialStateAction)(
      (currentStateAction, currentElement) => {
        val nextStateAction =
          for {
            currentIndexedList <- currentStateAction
            currentIndex       <- getState
            _                  <- setState(currentIndex + 1)
            nextIndexedList = (currentIndex, currentElement) :: currentIndexedList
          } yield nextIndexedList
        nextStateAction
      }
    )
  val firstIndex = 0
  val (indexedList,_) = finalStateAction.run(firstIndex)
  indexedList.reverse
```

This function numbers all the elements in a list using a **`State` action**. It keeps a **state** that's an **`Int`**, which is incremented at each step. We run the whole **composite `State` action** starting from 0. We then reverse the result since we constructed it in reverse order.

Note what's going on with `getState` and `setState` in the **for-comprehension**. We're obviously getting **variable binding** just like in the **`Id` monad**—we're **binding the value** of each successive **state action** (`getState`, `currentStateAction`, and then `setState`) to **variables**. But there's more going on, literally between the lines. At each line in the **for-comprehension**, the implementation of **`flatMap`** is making sure that the **current state** is available to `getState`, and that the **new state** gets **propagated** to all **actions** that follow a `setState`.

```scala
trait Monad[F[_]] {
   def unit[A](a: => A): F[A]
   def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
 }

def stateMonad[S] = new Monad[[A] =>> State[S,A]] {

   def unit[A](a: => A): State[S,A] =
     State(s => (a, s))

   def flatMap[A,B](st: State[S,A])(f: A => State[S,B]): State[S,B] =
     st flatMap f
 }
```

```scala
def zipWithIndex[A](as: List[A]): List[(Int,A)] =
  val emptyIndexedList = List[(Int, A)]()
  val initialStateAction = stateMonad[Int].unit(emptyIndexedList)
  val finalStateAction =
    as.foldLeft(initialStateAction)(
      (currentStateAction, currentElement) => {
        val nextStateAction =
          for {
            currentIndexedList <- currentStateAction
            currentIndex       <- getState
            _                  <- setState(currentIndex + 1)
            nextIndexedList =
              (currentIndex, currentElement) :: currentIndexedList
          } yield nextIndexedList
        nextStateAction
      }
    )
  val firstIndex = 0
  val (indexedList,_) = finalStateAction.run(firstIndex)
  indexedList.reverse
```

Note that while **FPiS** tends to use **monads** via the **Monad** trait, in this particular example we are only using the trait's **unit** function: the **for comprehension** desugars to invocations of the **map** and **flatMap** functions of the **State** class.

```scala
case class State[S, A](run: S => (A, S)) {

   def map[B](f: A => B): State[S, B] =
     State(s => {
       val (a, s1) = run(s)
       (f(a), s1)
     })

   def flatMap[B](f:A => State[S, B]): State[S, B] =
     State(s => {
       val (a, s1) = run(s)
       f(a).run(s1)
     })
 }

 object State {

   def getState[S]: State[S, S] =
     State(s => (s, s))

   def setState[S](s: => S): State[S, Unit] =
     State(_ => ((), s))

 }
```

```scala
assert( zipWithIndex( List("a", "b", "c"))
        == List((0,"a"), (1,"b"), (2,"c")) )
```

What does the difference between the action of `Id` and the action of `State` tell us about monads in general?

We can see that a chain of `flatMap` calls (or an equivalent for-comprehension) is like an imperative program with statements that assign to variables, and the monad specifies what occurs at statement boundaries.

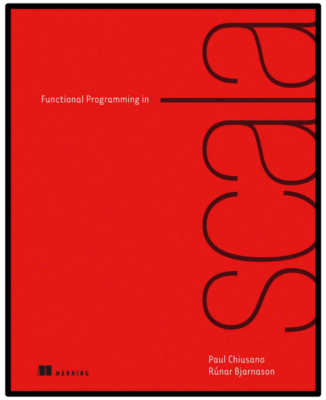
For example, with `Id`, nothing at all occurs except unwrapping and rewrapping in the `Id` constructor.

With `State`, the most current state gets passed from one statement to the next.

With the `Option` monad, a statement may return `None` and terminate the program.

With the `List` monad, a statement may return many results, which causes statements that follow it to potentially run multiple times, once for each result.

The `Monad` contract doesn't specify what is happening between the lines, only that whatever is happening satisfies the laws of associativity and identity.

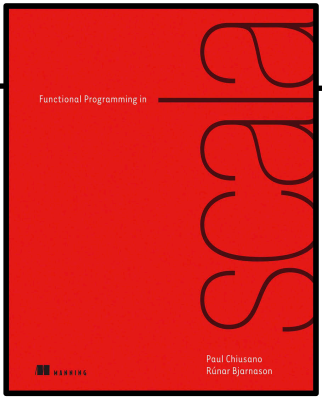


Functional Programming in Scala

@runarorama

@pchiusano

A chain of **flatMap** calls (or an equivalent **for-comprehension**) is like an **imperative program** with **statements** that **assign** to **variables**, and the **monad** specifies what occurs at **statement boundaries**.



For example, with `Id`, nothing at all occurs except unwrapping and rewrapping in the `Id` constructor.

Functional Programming in Scala

```
// the Identity Monad – does absolutely nothing
case class Id[A](a: A) {

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }

  def flatMap[B](f: A => Id[B]): Id[B] =
    f(a)
}
```

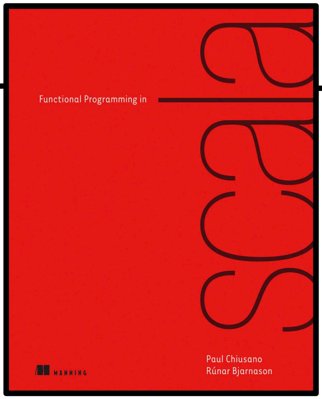
```
val result: Id[String] =
  for {
    hello <- Id("Hello, ")
    monad <- Id("monad!")
  } yield hello + monad

assert( result == Id("Hello, monad!"))
```

“An **imperative program** with **statements** that **assign** to **variables**“

“the **monad** specifies what occurs at **statement boundaries**“

“with `Id` nothing at all occurs except **unwrapping** and **rewrapping** in the `Id` constructor”

A chain of **flatMap** calls (or an equivalent **for-comprehension**) is like an **imperative program** with **statements** that **assign** to **variables**, and the **monad** specifies what occurs at **statement boundaries**.



"With the `Option` **monad**, **a statement may** return **None** and **terminate the program**"

Functional Programming in Scala

```scala
// the Option Monad
sealed trait Option[+A] {

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case None => None
      case Some(a) => f(a)
    }

}
case object None extends Option[Nothing] {
  def apply[A] = None.asInstanceOf[Option[A]]
}
case class Some[+A](get: A) extends Option[A]
```

"the **monad** specifies what occurs at **statement boundaries**"

"An **imperative program** with **statements** that **assign** to **variables**"

"With the `Option` **monad**, **a statement may** return **None** and **terminate the program**"

```scala
val result =
  for {
    firstNumber  <- None[String]
    secondNumber <- Some("333")
  } yield firstNumber + secondNumber

assert( result == None )
```

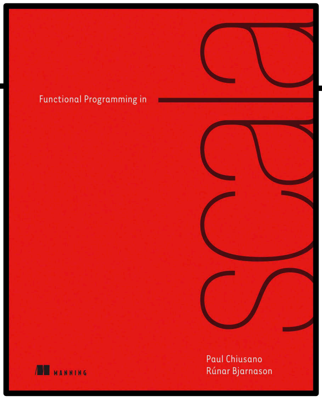
```scala
val result =
  for {
    firstNumber  <- Some(333)
    secondNumber <- Some(666)
  } yield firstNumber + secondNumber

assert( result == Some(999) )
```

```scala
val result =
  for {
    firstNumber  <- Some(333)
    secondNumber <- None[Int]
  } yield firstNumber + secondNumber

assert( result == None )
```

```scala
val result =
  for {
    firstNumber  <- Some("333")
    secondNumber <- Some("666")
  } yield firstNumber + secondNumber

assert( result == Some("333666") )
```

A chain of **`flatMap`** calls (or an equivalent **for-comprehension**) is like an **imperative program** with **statements** that **assign** to **variables**, and the **monad** specifies what occurs at **statement boundaries**.

```scala
// The List Monad
sealed trait List[+A] {

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a), Nil) }

  def flatMap[B](f: A => List[B]): List[B] =
    this match {
      case Nil =>
        Nil
      case Cons(a, tail) =>
        concatenate(f(a), (tail flatMap f))
    }
}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {
  def concatenate[A](left:List[A], right:List[A]):List[A] =
    left match {
      case Nil =>
        right
      case Cons(head, tail) =>
        Cons(head, concatenate(tail, right))
    }
}
```



With the **`List`** **monad**, **a statement may return many results**, **which causes statements that follow it to potentially run multiple times**, **once for each result**.

Functional Programming in Scala

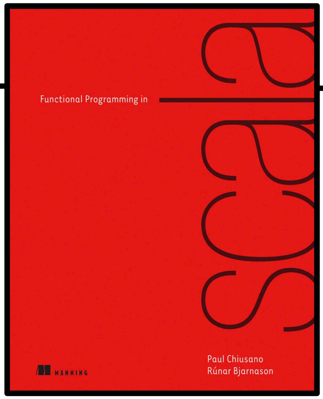
"the **monad** specifies what occurs at **statement boundaries**"

"An **imperative program** with **statements** that **assign** to **variables**"

"With the **`List`** **monad**, **a statement may return many results**, **which causes statements that follow it to potentially run multiple times**, **once for each result**."

```scala
val result =
  for {
    letter <- Cons("A", Cons("B", Nil))
    number <- Cons(1, Cons(2, Nil))
  } yield letter + number

assert(
  result
  == Cons("A1",Cons("A2",Cons("B1",Cons("B2",Nil))))
)
```

A chain of **`flatMap`** calls (or an equivalent **for-comprehension**) is like an **imperative program** with **statements** that **assign** to **variables**, and the **monad** specifies what occurs at **statement boundaries**.



With **`State`**, **the most current** **state gets passed from one statement to the next**.

Functional Programming in Scala

```scala
 // The State Monad
case class State[S, A](run: S => (A, S)) {

   def map[B](f: A => B): State[S, B] =
     this flatMap { a => State { s => (f(a), s) } }

   def flatMap[B](f:A => State[S, B]): State[S, B] =
     State(s => {
       val (a, s1) = run(s)
       f(a).run(s1)
     })
}
```

```scala
object State {
  def getState[S]: State[S, S] =
    State(s => (s, s))

  def setState[S](s: => S): State[S, Unit] =
    State(_ => ((), s))
}
```

```scala
val F = stateMonad[Int]

def zipWithIndex[A](as: List[A]): List[(Int,A)] =
  as.foldLeft(F.unit(List[(Int, A)]()))((acc,a) =>
    for {
      xs <- acc
      n  <- getState
      _  <- setState(n + 1)
    } yield (n, a) :: xs).run(0)._1.reverse
```

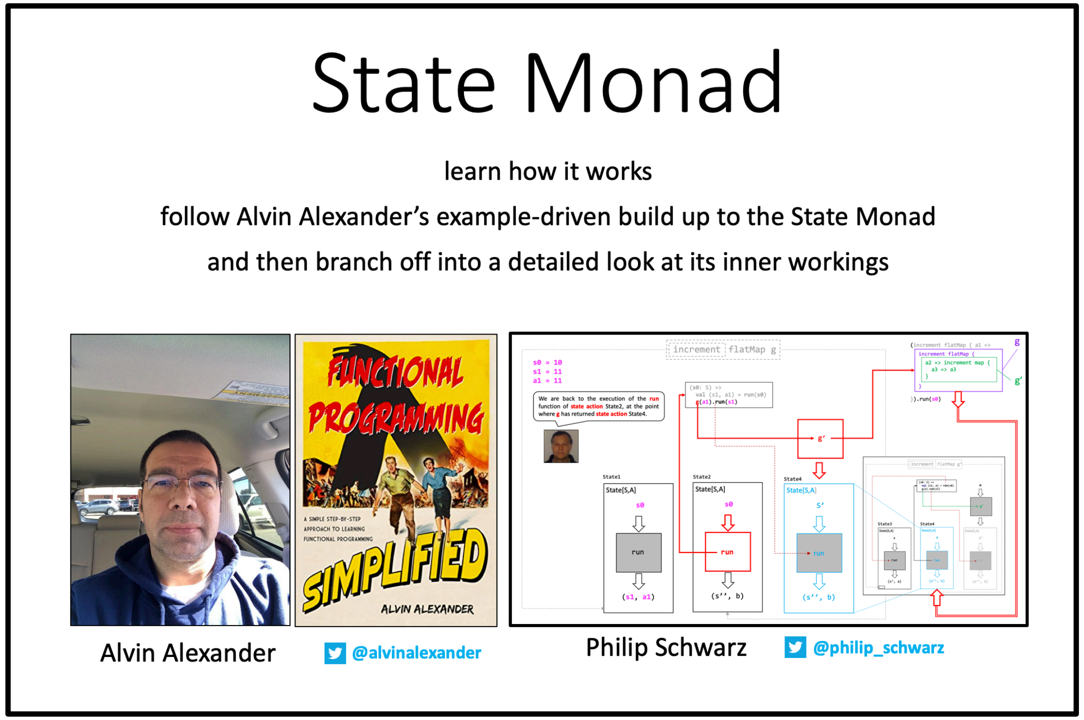
“the **monad** specifies what occurs at **statement boundaries**“

“An **imperative program** with **statements** that **assign** to **variables**“

“With **`State`**, **the most current** **state gets passed from one statement to the next**. “

```scala
assert( zipWithIndex( List("a", "b", "c"))
        == List((0,"a"), (1,"b"), (2,"c")) )
```

If you are interested in an introduction to the **State** monad then see the following slide deck at https://www.slideshare.net/pjschwarz



State Monad

learn how it works

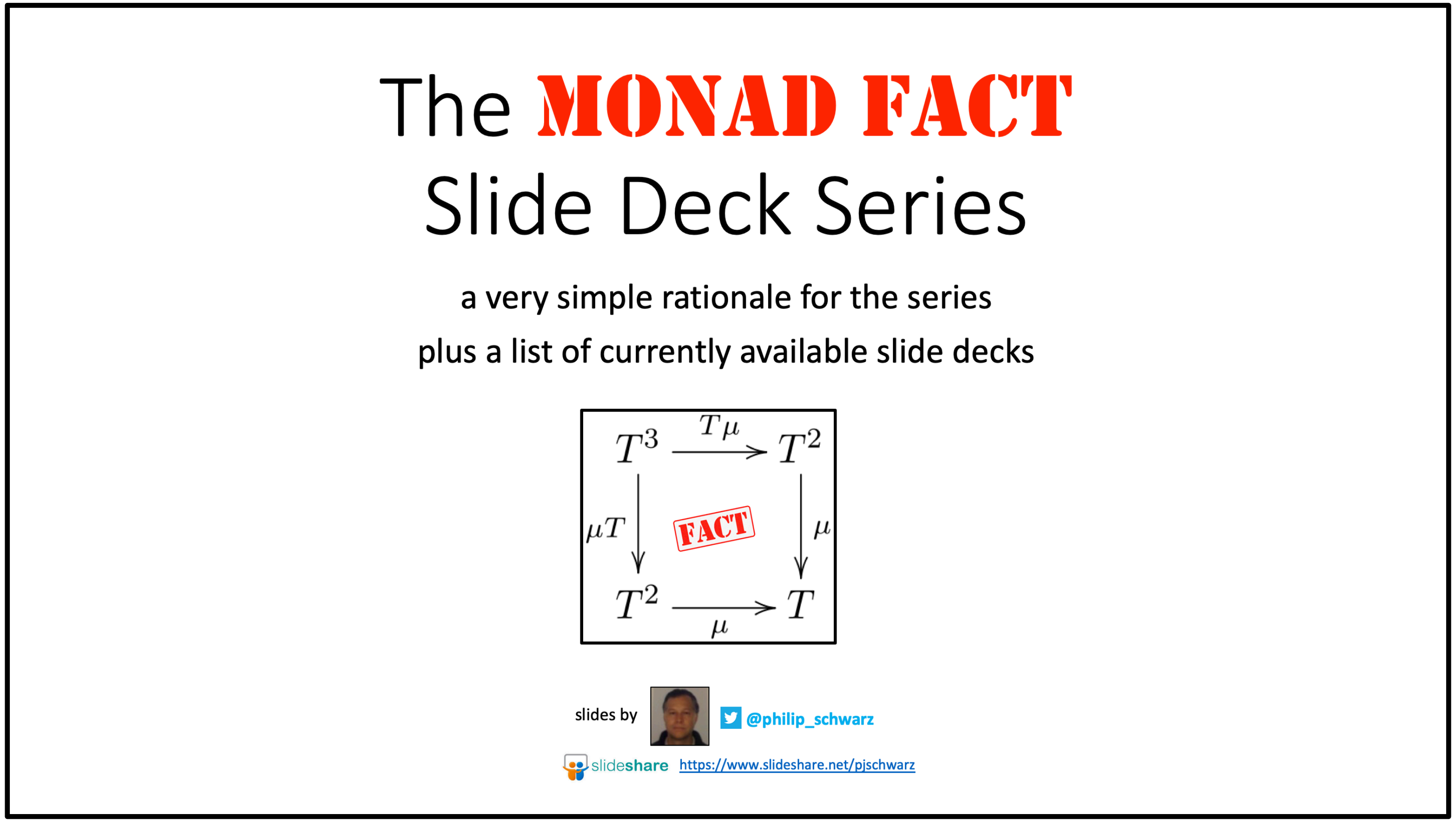
follow Alvin Alexander's example-driven build up to the State Monad

and then branch off into a detailed look at its inner workings

Alvin Alexander @alvinalexander

Philip Schwarz @philip_schwarz

See the following slide deck for the list of all available decks in the **MONAD FACT** series



The **MONAD FACT**
Slide Deck Series

a very simple rationale for the series

plus a list of currently available slide decks

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz