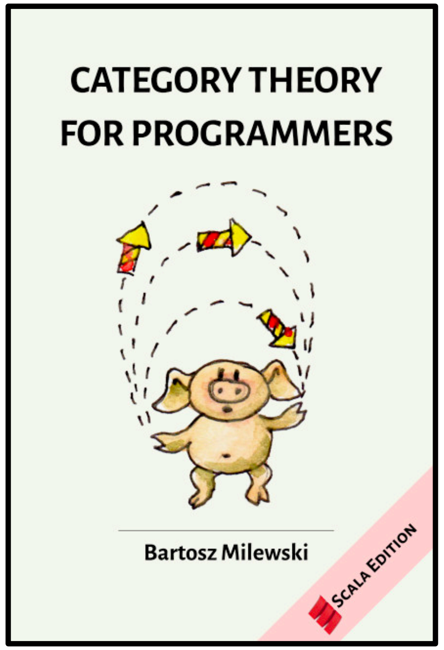
MONAD FACT #6

a **monad** is an **overloading** of the **semicolon**

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] \arrow[dr, phantom, "\text{FACT \#1}"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

slides by @philip_schwarz

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

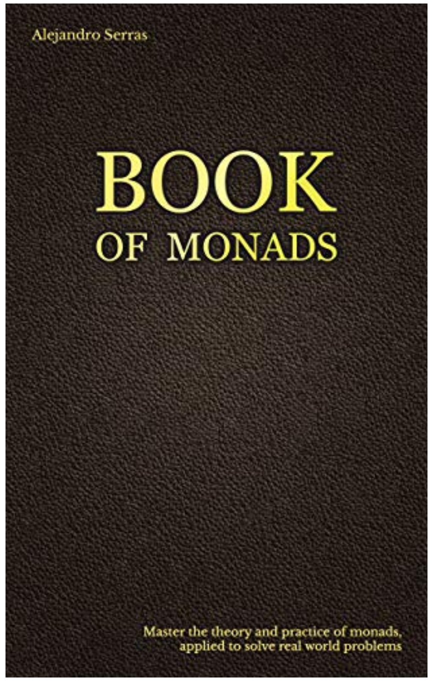


Functional Geekery Episode 69 – Bartosz Milewski

Functional Geeks, Geeking Functionally

Steven Proctor
@stevenproctor

Bartosz Milewski
@BartoszMilewski


Alejandro Serras
BOOK
OF MONADS
Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena
@trupill

20 Monads: Programmer's Definition

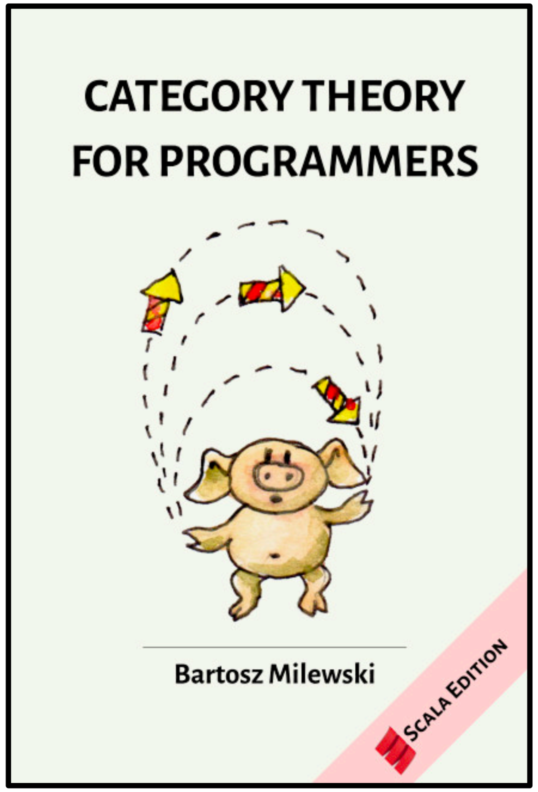
Programmers have developed a whole mythology around monads. It's supposed to be one of the most abstract and difficult concepts in programming. There are people who "get it" and those who don't. **For many, the moment when they understand the concept of the monad is like a mystical experience**. **The monad abstracts the essence of so many diverse constructions that we simply don't have a good analogy for it in everyday life**. We are reduced to groping in the dark, like those blind men touching different parts of the elephant end exclaiming triumphantly: "It's a rope," "It's a tree trunk," or "It's a burrito!"

Let me set the record straight: The whole mysticism around the monad is the result of a misunderstanding. **The monad is a very simple concept**. **It's the diversity of applications of the monad that causes the confusion**.

As part of research for this post I looked up duct tape (a.k.a., duck tape) and its applications. Here's a little sample of things that you can do with it:

- sealing ducts
- fixing CO2 scrubbers on board Apollo 13
- wart treatment
- fixing Apple's iPhone 4 dropped call issue
- making a prom dress
- building a suspension bridge

Now imagine that you didn't know what duct tape was and you were trying to figure it out based on this list. Good luck! So I'd like to add one more item to the collection of "the monad is like…" clichés: **The monad is like duct tape. Its applications are widely diverse, but its principle is very simple: it glues things together. More precisely, it composes things**.



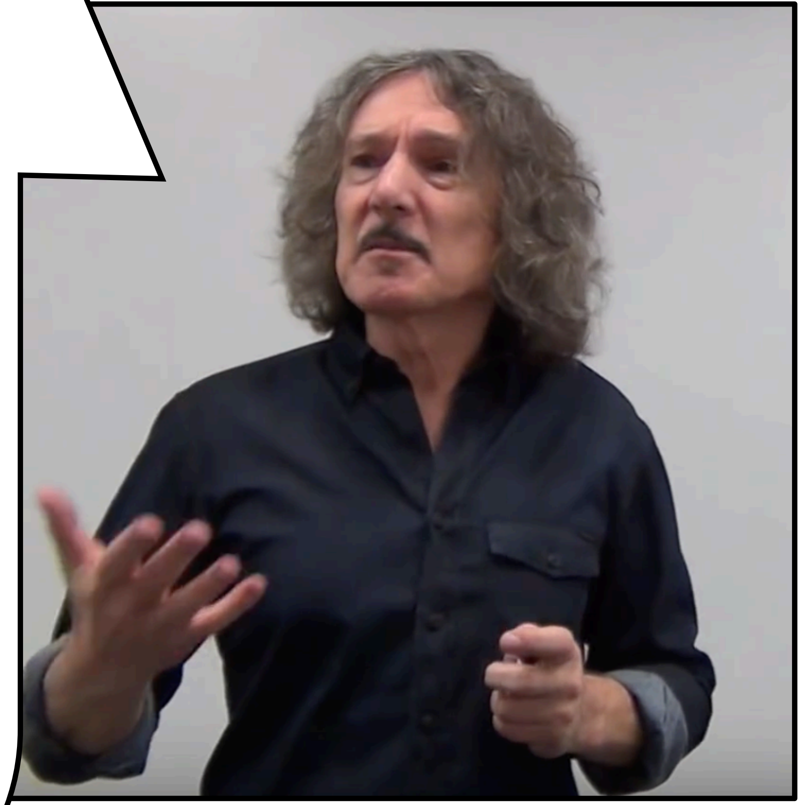
Essentially you have **just one category that we use in programming**, that's **the category of types and functions**.

And that's the starting point. And you have to start thinking about a program and when you use FP it is much easier to think of it this way, but in general this is sort of hidden under a lot of noise, is that **a program really consists of functions and you build larger programs or larger functions from smaller functions by composing them**.

So **the basic way of structuring your program** is, I am calling a function with an argument, I am getting a result, and I am passing this result to another function as an argument and this function produces a result, and so on. So **it is just the chaining of functions**, and that is called composition and **composition of functions** or in category theory they are more general things than functions, they are called morphisms, but in our case, in programming, these morphisms are just functions.

The **composition of functions**, the **composition is the essence of a category** and this is just one example, when you just take **straightforward functions** and you compose them, you build larger functions and so on, so this is how we programmers work with functions.

Functional Geekery Episode 69
Bartosz Milewski



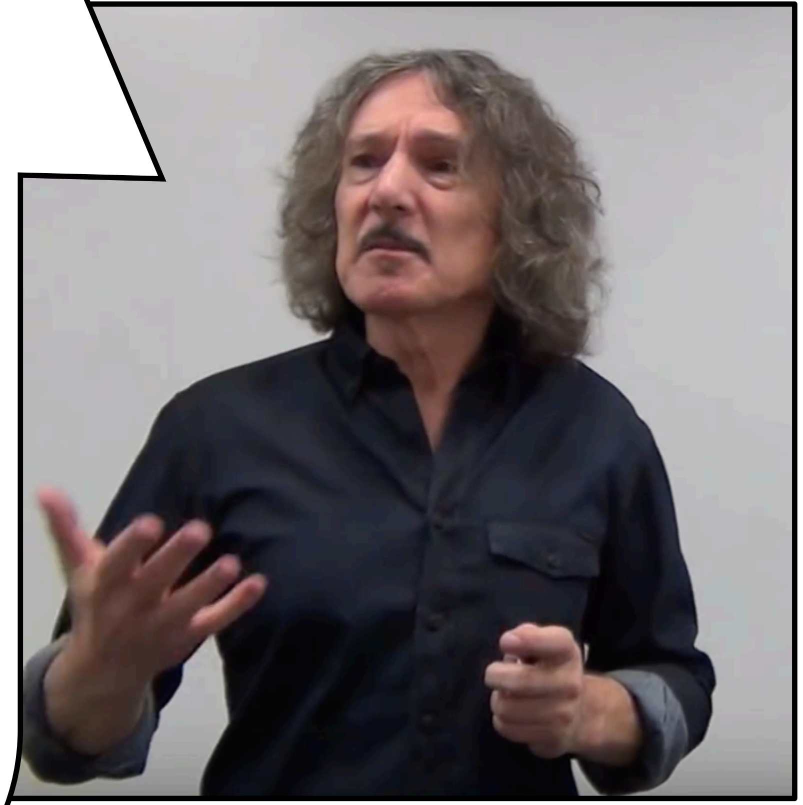
@BartoszMilewski

But then where does the monad come in? When you have functions that have side effects. Functions that do something. They are not mathematical functions. Most functions that you use in C++ for instance, they have side effects. You call them multiple times and they might give you different results every time you call them. Or they modify some external state. They are not pure functions.

So this was the biggest problem in FP, how to describe these side effects, how to describe things that are not really pure functions, and the answer is, if you have all these different kinds of impurity that you can have, and they are very different you know, like you have functions that throw exceptions, you have functions that take input from the user, you have functions that produce output to the screen and so on, there are so many different ones, you have functions that have hidden state, and so on. All these very different ways of being non-pure, can actually be translated into pure functions with some additional structure.

And the question is, if normal programming means just composing functions to create bigger functions, how do you compose these special functions, the functions that have side effects or modify state, and so on? How can you compose them, because they are not your traditional functions, and the answer is, for all these very complex ways of encoding side effects, for all of them the answer is 'use a monad'.

Functional Geekery Episode 69
Bartosz Milewski



@BartoszMilewski

A monad is a way of composing functions that have side effects, essentially.

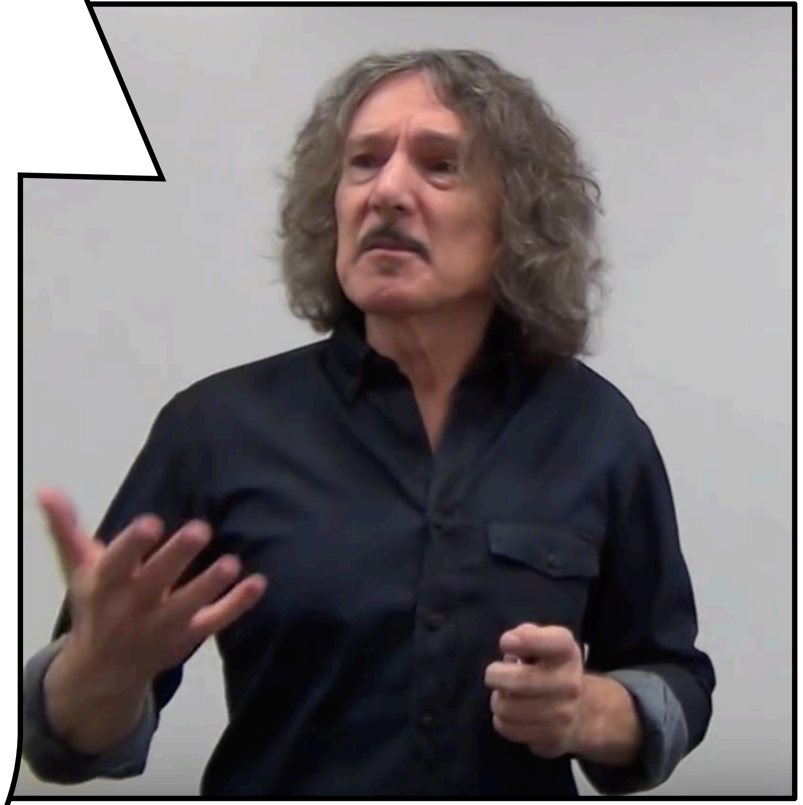
And because there is such a huge variety of side effects that you can describe this way, people have problems understanding what a monabd is, because they say, OK, a **monad** is about **exceptions**. No, a **monad** is about **state**. No, a **monad** is about **IO**. No, a **monad** is about this and that.

No, a monad is just about how compose these things. It is about composition. Once you understand this, it is just a pattern for composing things and that's easy.

There is a lot of confusion because of this and the word became sort of like a cuss word, like people try to avoid it.

I don't know why people have no problems with the word object in OO programming, because it is a common word. **Try defining an object in OO programming and you'll get into much bigger trouble than trying to define a monad in FP**. It is just that it is a word that is used more often than the word **monad**. But **whatever you call it, the idea is not that hard**.

Functional Geekery Episode 69
Bartosz Milewski



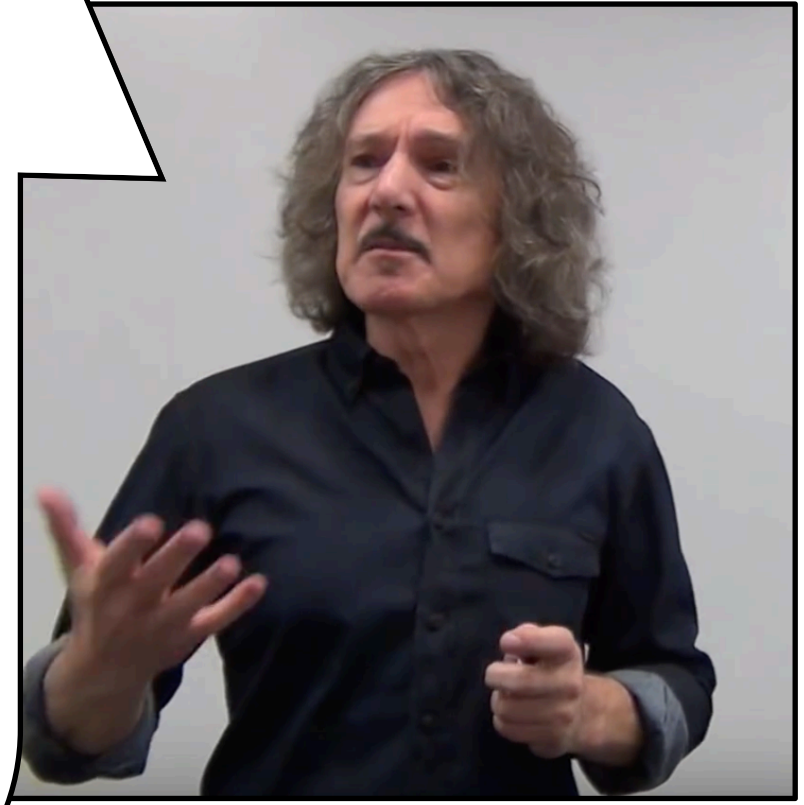
@BartoszMilewski

I think the complexity comes in because we are so used to the imperative style of programming, where we actually split things into very small steps, so for instance for an imperative programmer this idea of composing functions is not immediately obvious. Function composition in an imperative language looks more like this:

```
int x = f(y)

… <something, something, something> …

string s = g(x)
```

and you already forgot that x was obtained by calling the function f previously, so you don't really see this composition of two functions since it was obscured by creating these temporary variables to begin with, I mean you created a variable x just to hold the result of one function and then you pass this x to another function. This process of naming these variables is just a side effect of the way we do imperative programming. So this idea that we are composing functions, that we are passing results of one function as argument to another function, that's so deeply hidden that it is almost invisible and there is a lot of glue code in between function calls that do little things. This glue code can also be abstracted into functions and then you end up with just composing functions.

Functional Geekery Episode 69
Bartosz Milewski



@BartoszMilewski

The problem is that we are not used to thinking of programming in terms of function composition. This is understandable.

We often give names to intermediate values rather than pass them directly from function to function. We also inline short segments of glue code rather than abstract them into helper functions.

Here's an imperative-style implementation of the vector-length function in C:

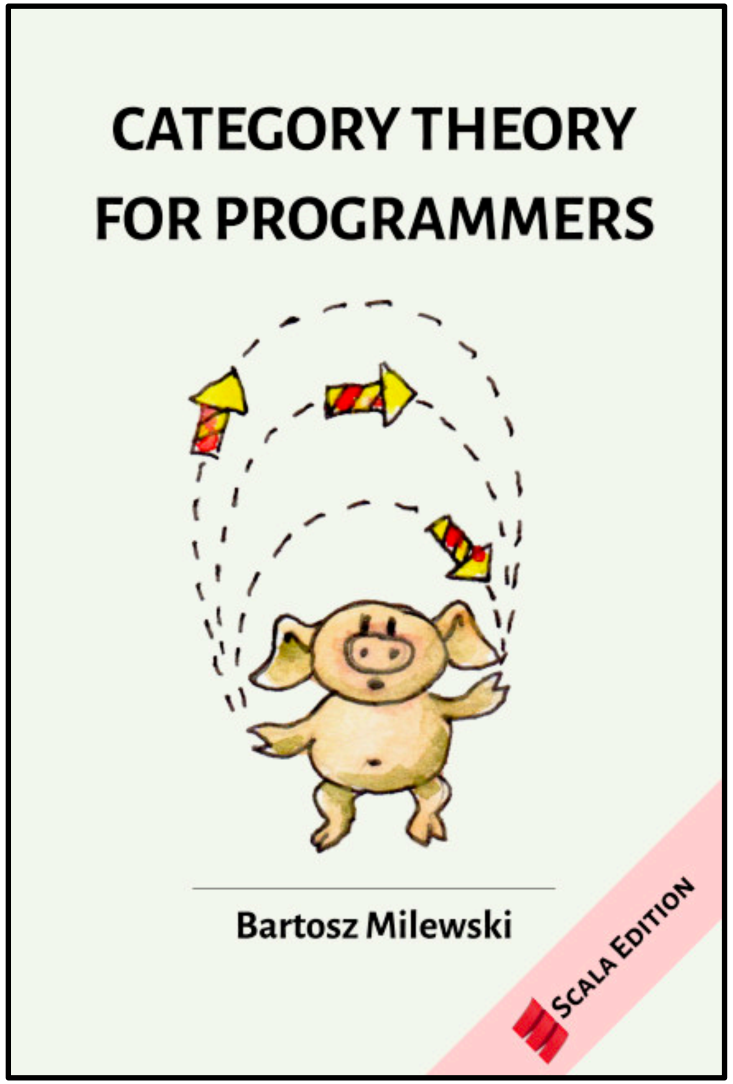
```
double vlen(double * v) {
    double d = 0.0;
    int n;
    for (n = 0; n < 3; ++n)
        d += v[n] * v[n];
    return sqrt(d);
}
```

Compare this with the (stylized) Haskell version that makes function composition explicit:

```
vlen = sqrt . sum . fmap (flip (^) 2)
```

(Here, to make things even more cryptic, I partially applied the exponentiation operator (^) by setting its second argument to 2.)

I'm not arguing that Haskell's point-free style is always better, just that function composition is at the bottom of everything we do in programming. And even though we are effectively composing functions, Haskell does go to great lengths to provide imperative-style syntax called the do notation for monadic composition.



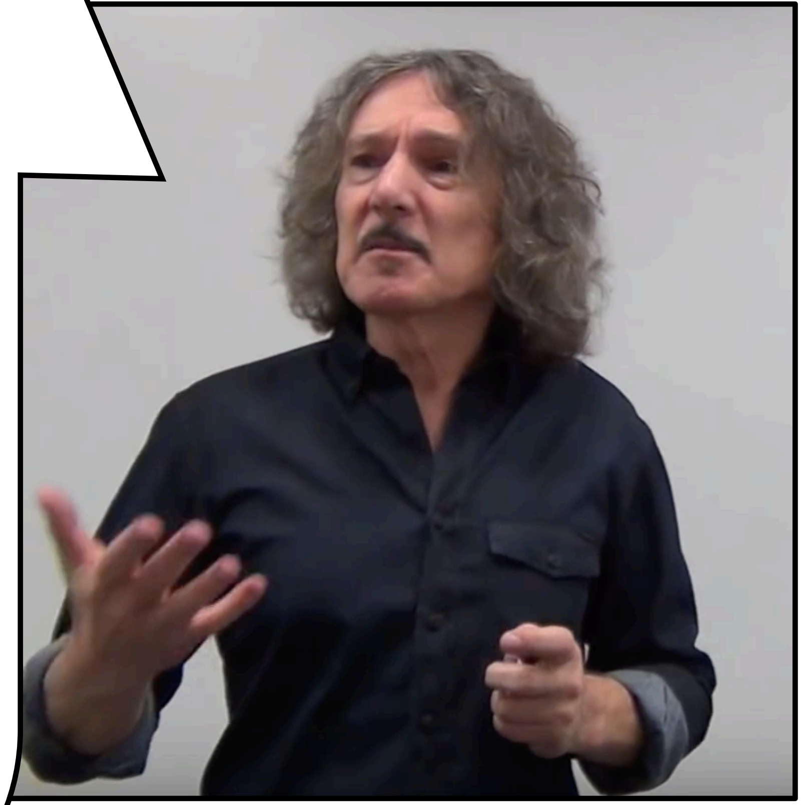
And I am not saying that code written as a composition of functions is the most readable code, sometimes it's not.

So sometimes it's really better to give names to your temporary variables. It's good because you can give them meaningful names and they sort of serve as comments in your code because they have additional names, they specify what is the semantics of what I am doing, but once you know FP, you start looking at imperative code differently and you see that it is really about function composition and it could be that composing functions in this imperative style that is much more verbose is actually easier to understand, as long as you don't lose this idea that you are actually composing functions.

So if you take imperative code you can translate it into functional code by just doing function composition, and function composition, you now, like in C++ there isn't even a library function called compose. There is no higher-order function that takes two functions and produces a new function that is the composition of these two. There isn't even a function like this.

So this just shows you how far away imperative programming is from this idea of composing functions. Whereas in Haskell, if you want to compose two functions you just put a period, a dot in between functions. That's a composition operator.

Functional Geekery Episode 69
Bartosz Milewski



@BartoszMilewski

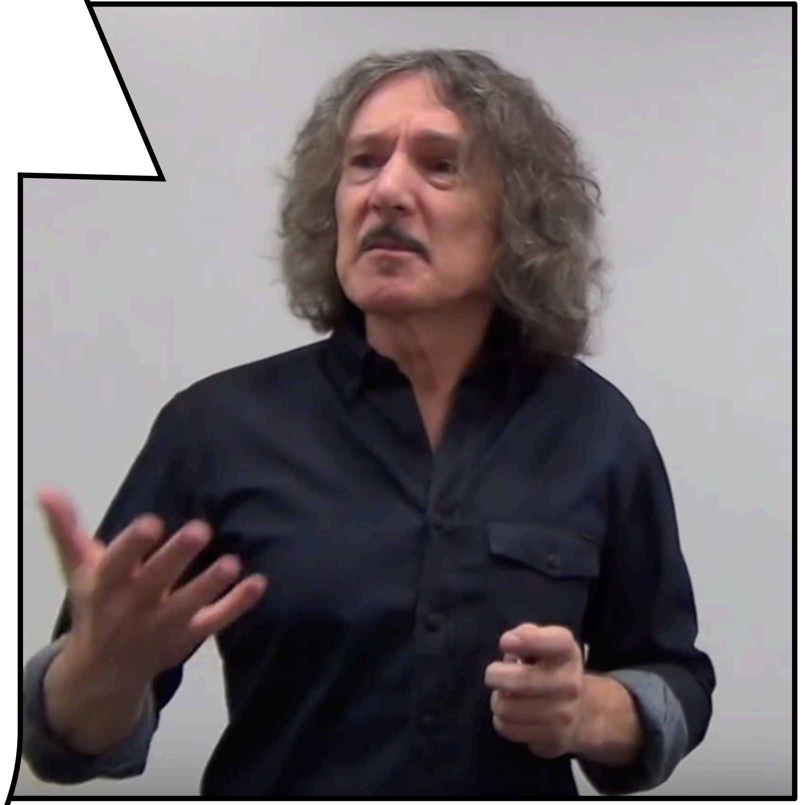
So you see composition in Haskell code. You won't see it in C++ code. And in Haskell code if you say now I want my functions to actually do some **additional stuff**, for instance **exceptions**, ot **state**, or so on, **I will replace this dot, with something else, and in fact replacement for the dot when you go into a monadic composition is called the fish operator, it is like >=> and that replaces the dot**. So once you understand that you are doing **function composition**, you have two things.

You have **functions** and you have **composition**. And **you can modify your functions or you can modify the way you compose them**, and **this tweaking of the way functions are composed is done through a monad**, or through **applicative**, that's an even simpler way of doing this, but **the tweaking** of **composition**, **this is why people sometimes say that**

a monad is an overloading of the semicolon

If you want to explain it to an imperative programmer, a semicolon is something that's between functions, and composition is also something that's between functions, you just overload it, you say 'after you call this function and before you call the next function, do this additional stuff', and that is what a monad is.

Functional Geekery Episode 69
Bartosz Milewski



For the fish operator >=> see **MONAD FACT #2** and **#3**

20.3 The do Notation

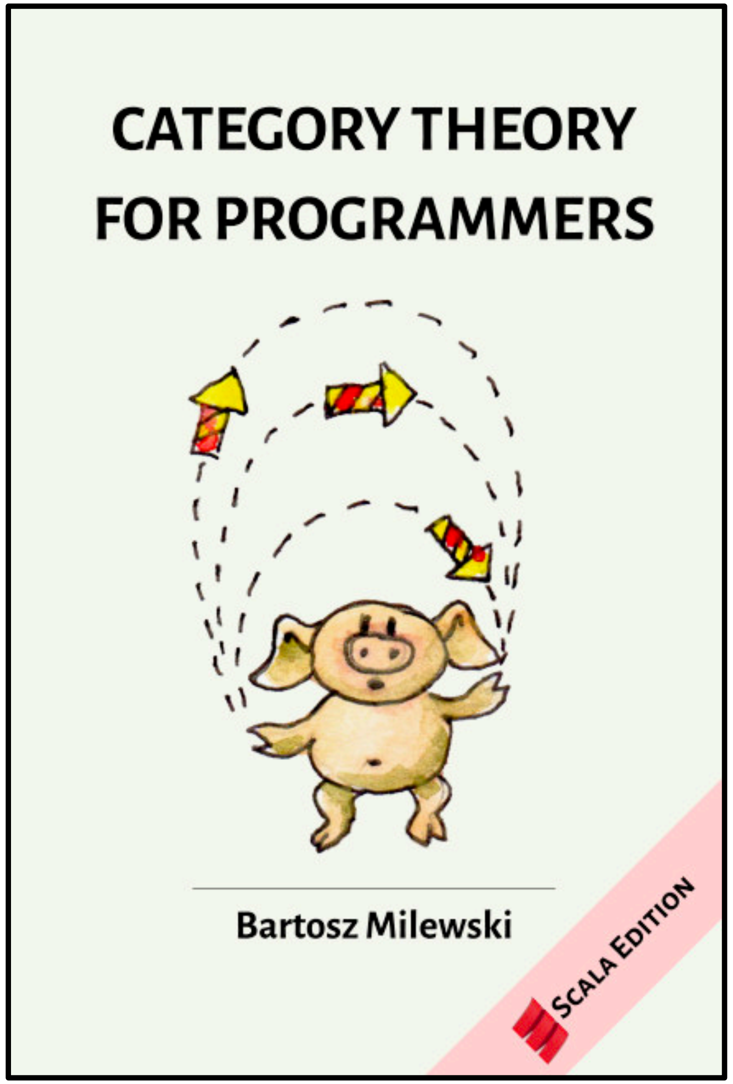
One way of writing code using monads is to work with Kleisli arrows — composing them using the fish operator. This mode of programming is the generalization of the point-free style. Point-free code is compact and often quite elegant. In general, though, it can be hard to understand, bordering on cryptic. That's why most programmers prefer to give names to function arguments and intermediate values.

When dealing with monads it means favoring the bind operator over the fish operator. Bind takes a monadic value and returns a monadic value. The programmer may chose to give names to those values. But that's hardly an improvement. What we really want is to pretend that we are dealing with regular values, not the monadic containers that encapsulate them. That's how imperative code works — side effects, such as updating a global log, are mostly hidden from view. And that's what the do notation emulates in Haskell.

You might be wondering then, why use monads at all? If we want to make side effects invisible, why not stick to an imperative language?

The answer is that the monad gives us much better control over side effects. For instance, the log in the Writer monad is passed from function to function and is never exposed globally. There is no possibility of garbling the log or creating a data race. Also, monadic code is clearly demarcated and cordoned off from the rest of the program.

The do notation is just syntactic sugar for monadic composition. On the surface, it looks a lot like imperative code, but it translates directly to a sequence of binds and lambda expressions.



For the fish operator >=> see MONAD FACT #2 and #3

In the **Book of Monads**, **Alejandro Serrano** looks at both **Haskell**'s **do notation** and **Scala**'s equivalent, i.e. the **for comprehension**.

He refers to both as **block notation**. He looks first at the problems that arise when we use monadic operators directly and then at how block notation solves those problems.

@trupill

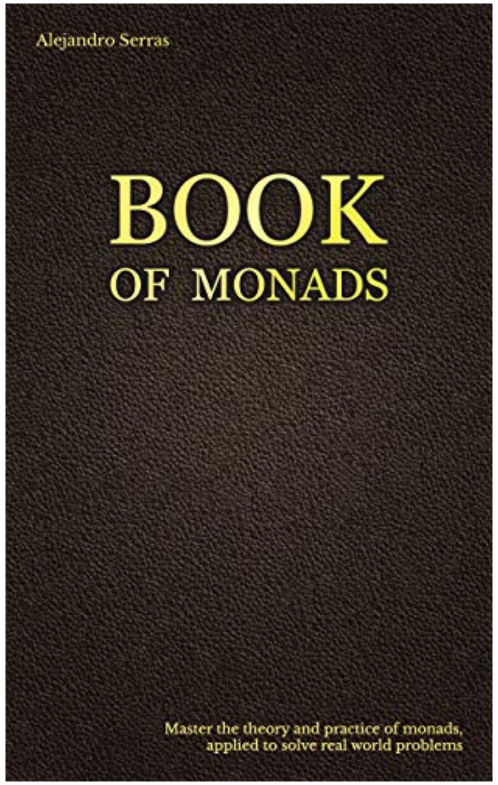
Even in these simple examples, we can see the usual pattern of **monads**. We have a bunch of **monadic operations** — functions with a return type of the form **m** a or **M**[A] for the corresponding **monad** — and after each of them, we apply a **bind function** in order to do something with the value the monad contains in subsequent computations. Eventually, we use **return** or **point** to construct the final value, which is wrapped in — or "**returned** into" — the **monadic context**. Note that this use of "**return**" is different from the keyword of the same name that often appears in imperative languages. In **Haskell**, **return** is just another function, and it does not imply that control flow escapes from its **monadic context** in any way.

Given the pervasiveness of this pattern, many programming languages have dedicated **syntactic sugar** to describe **monadic computations**.

>>= is the **bind** function, known in **Scala** as **flatMap**

```
validatePerson name age =
  validateName name >>= \name' ->
  validateAge age >>= \age' ->
  return (Person name' age')
```

```
type Name = String
data Person = Person { name :: Name, age :: Int }

validateName :: String -> Maybe Name = ???
validateAge :: Int -> Maybe Int = ???

Monad m where
  return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```


Alejandro Serras
BOOK
OF MONADS
Master the theory and practice of monads, applied to solve real world problems

bind is another name for **flatMap**, and **point** is another name for **unit** and **pure**

```
def validatePerson(s: String, n: Int) =
  validateName(s) bind { name =>
    validateAge(n) bind { age =>
      point(Person(name, age))
    }
  }
```

```
type Name = String
case class Person(name: Name, age: Int)

def validateName(s: String): Option[Name] = ???
def validateAge(n: Int): Option[Int] = ???

trait Monad[ M[_]] {
  def point[A](x: A): M[A]
  def bind[A, B](x: M[A]) (f: A => M[B]): M[B]
}
```

```scala
def validatePerson(s: String, n: Int)(implicit M: Monad[Option]): Option[Person] =
  M.bind(validateName(s))  { name =>
    M.bind(validateAge(n)) { age =>
      M.point(Person(name, age))
    }
  }
```

Scala

@philip_schwarz

Here is the full **Scala** code for the example on the previous slide. Note the following:

1. In the previous slide, **bind** and **point** are in **infix position**. I wasn't able to quickly get that to compile, so here I don't use **infix notation**.
2. For didactic reasons, **Alejandro Serrano** chose to give **Option**'s implementation of **bind** the name **then** (which, by the way, is a reserved word in **Scala**, though it only leads to a warning).

```scala
trait Monad[ M[_]] {
  def point[A](x: A): M[A]
  def bind[A, B](x: M[A]) (f: A => M[B]): M[B]
}
```

```scala
type Name = String

case class Person(name: Name, age: Int)

def validateName(s: String): Option[Name] =
  if (s.length > 1 && s.length < 15)
    Some(s)
  else None

def validateAge(n: Int): Option[Int] =
  if (n > 0 && n <= 110)
    Some(n)
  else None
```

```scala
assert(validatePerson("Fred", 35) == Some(Person("Fred", 35)))
assert(validatePerson("F", 35) == None)
assert(validatePerson("Fred", 200) == None)
```

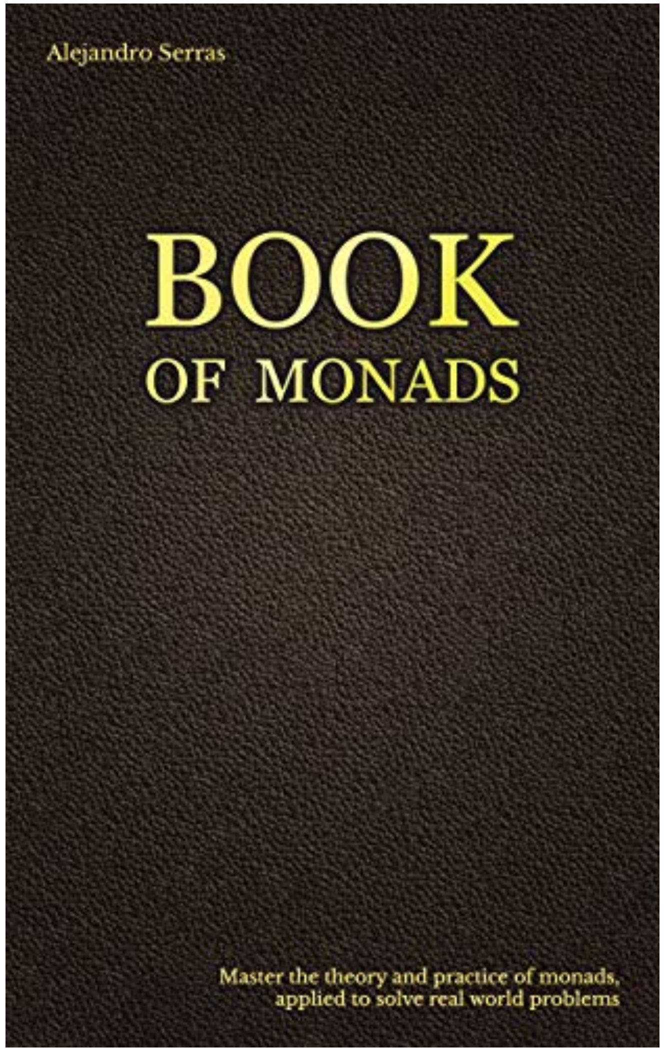
```scala
sealed abstract class Option[+A]{

  def then[B](f: A => Option[B]): Option[B] =
    this match {
      case None => None
      case Some(x) => f(x)
    }
}
case object None extends Option[Nothing]
case class Some[+A](value: A) extends Option[A]

object Option {

  implicit val optionMonad: Monad[Option] = new Monad[Option] {

    def point[A](x: A): Option[A] =
      Some(x)

    def bind[A, B](x: Option[A])(f: A => Option[B]): Option[ B] =
      x then f
  }
}
```

Block Notation

There are **three main problems with writing code using monadic operators directly**:

1. We need to **repeat the name of the bind function over and over**. **Haskell** tries to “hide” it under a symbolic name, (**>>=**), but you **still need to write it on every line**. And sometimes **Haskell**’s solution does not work very well: **beginners may feel intimidated by so many strange symbols on their screens**!
2. **In most programming languages, giving a name to a value takes the form name = value**, maybe with a slightly different equality or assignment symbol. **Monadic code breaks this pattern, since the name we give to the element in the monad** — for example, **name** or **age** in the preceding code — **is written after the expression that produces it**.
3. **In those languages where anonymous functions are delineated by parentheses or braces, such as Scala**, **we need to keep track of how many to close at the end of the expression**. This seems like a small annoyance, but it makes it **harder to refactor monadic code**. In this respect, the **Haskell** syntax works a bit better, because anonymous functions always extend until the end of the expression. In practice, this is a good default for **monadic** code.

In order to combat these problems, several functional programming languages natively support the notion of monadic blocks. In a monadic block, code may be written in a more pleasing way and is transformed by the compiler into nested calls to the bind operation. Unfortunately, there is no consensus about the extent to which monadic operations should be hidden with syntax, which has led to many different variations of **monadic blocks**.


Alejandro Serras
BOOK OF MONADS
Master the theory and practice of monads, applied to solve real world problems

The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

Do Notation

Haskell introduced do blocks in one of the first revisions of its specification, and it has become a flagship feature of the language. Nowadays, do notation is also available in other Haskell-inspired languages, such as PureScript and Idris.

The key idea of do notation is simple:

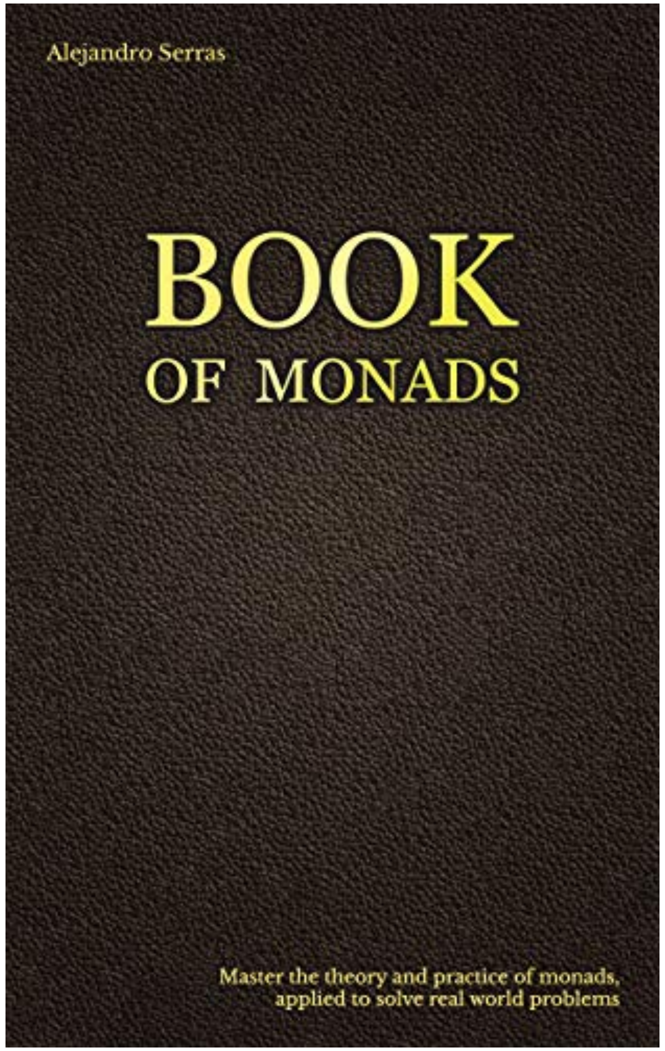
- fix problem (1) by automatically inserting calls to (>>=)
- fix problem (2) by providing specialized syntax for naming in a monadic context

The two code blocks at the beginning of this chapter could be written as follows:

```
validatePerson name age =
  do name' <- validateName name
     age’  <- validateAge age
     return (Person name' age’)
```

```
def validatePerson(s: String, n: Int) =
  for {
    name <- validateName(s)
    age  <- validateAge(n)
  } yield Person(name, age)
```

Alejandro Serrano Mena
@trupill


Alejandro Serras
BOOK OF MONADS
Master the theory and practice of monads, applied to solve real world problems
The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

```scala
def validatePerson(s: String, n: Int): Option[Person] =
  for {
    name <- validateName(s)
    age  <- validateAge(n)
  } yield Person(name, age)
```

Here is the full **Scala** code again, but this time, rather than using **bind** and **point** functions, the **validatePerson** function uses the **syntactic sugar** of a **for comprehension**.

Also, note the following:

1. I added a **map** function to **Option**, since it is needed in order to permit desugaring of the **for comprehension**
2. I renamed **Option**'s **then** function to **flatMap** for the same reason

```scala
trait Monad[ M[_]] {
  def point[A](x: A): M[A]
  def bind[A, B](x: M[A]) (f: A => M[B]): M[B]
}
```

```scala
type Name = String
case class Person(name: Name, age: Int)

def validateName(s: String): Option[Name] =
  if (s.length > 1 && s.length < 15)
    Some(s)
  else None

def validateAge(n: Int): Option[Int] =
  if (n > 0 && n <= 110)
    Some(n)
  else None
```

```scala
assert(validatePerson("Fred", 35) == Some(Person("Fred", 35)))
assert(validatePerson("F", 35) == None)
assert(validatePerson("Fred", 200) == None)
```

for desugaring of a **for comprehension**, see **MONAD FACT #1**

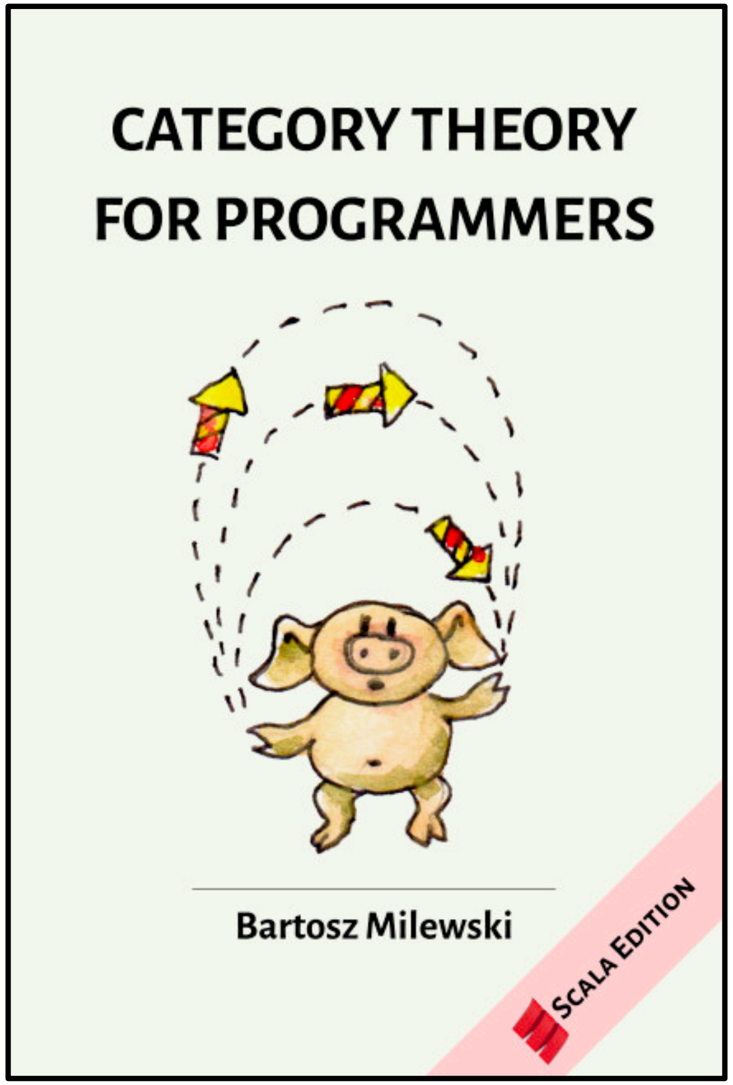
```scala
sealed abstract class Option[+A]{
  def map[B](f: A => B): Option[B] =
    this match {
      case None => None
      case Some(x) => Some(f(x))
    }
  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case None => None
      case Some(x) => f(x)
    }
}
case object None extends Option[Nothing]
case class Some[+A](value: A) extends Option[A]

object Option {

  implicit val optionMonad: Monad[Option] = new Monad[Option] {
    def point[A](x: A): Option[A] =
      Some(x)
    def bind[A, B](x: Option[A])(f: A => Option[B]): Option[ B] =
      x flatMap f
  }
}
```

In general, **do blocks consist of lines (or sub-blocks) that either use the left arrow to introduce new names that are then available in the rest of the code**, **or are executed purely for side-effects**.

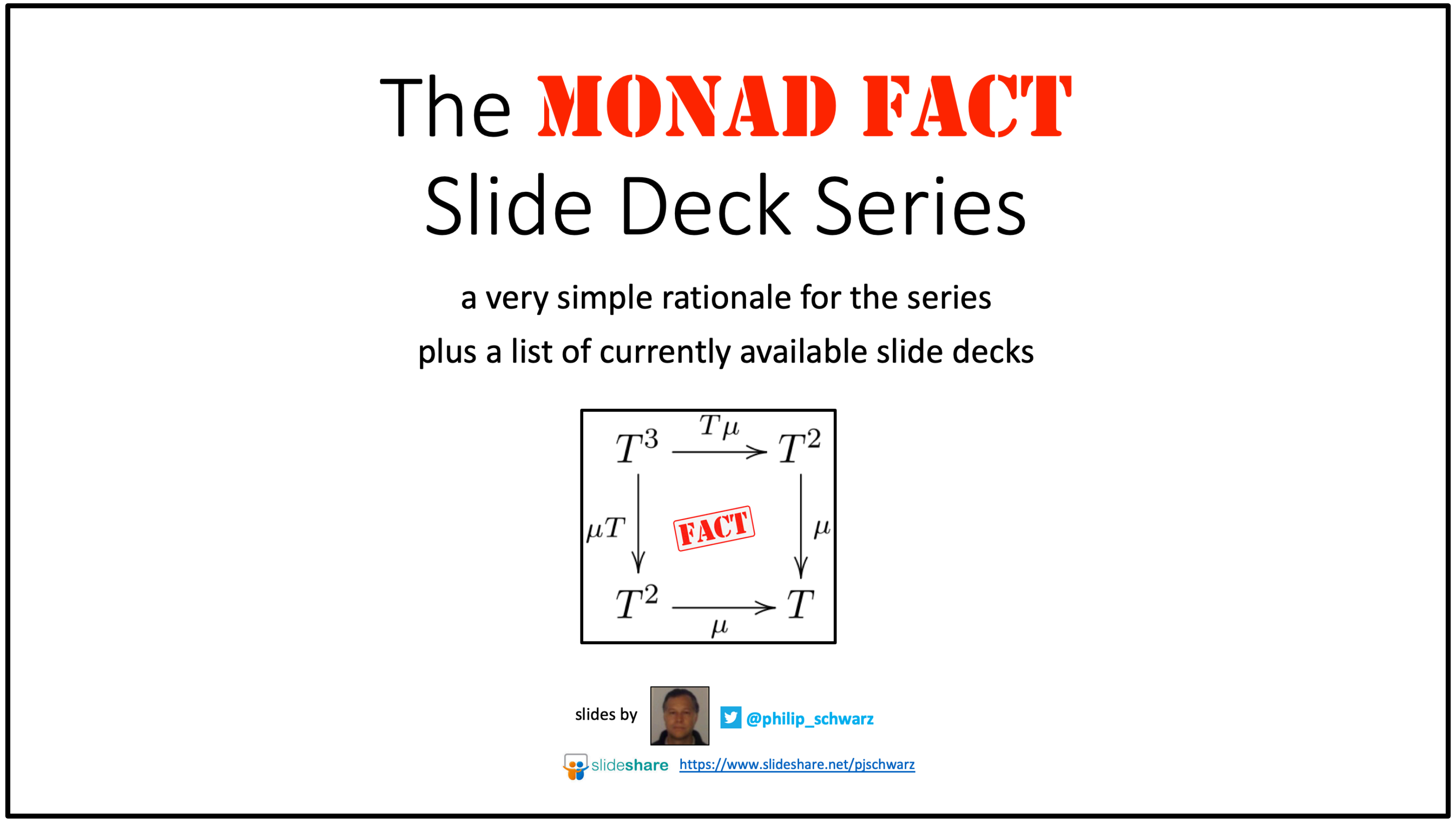
Bind operators are implicit between the lines of code. **Incidentally, it is possible, in Haskell, to replace the formatting in the do blocks with braces and semicolons.**

This provides the justification for describing the monad as a way of overloading the semicolon.

Notice that the nesting of **lambdas** and **bind operators** when desugaring the **do notation** has the effect of influencing the execution of the rest of the **do block** based on the result of each line. This property can be used to introduce complex control structures, for instance to simulate exceptions.



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

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

slides by @philip_schwarz

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