Functional Effects

Part 2

learn about functional effects through the work of

John A De Goes
@jdegoes

slides by @philip_schwarz
slideshare https://www.slideshare.net/pjschwarz

In this slide deck we go through the following two sections of **One Monad to Rule Them All**, a great talk by **John A De Goes**:

- Intro to **Functional Effects**
- Tour of the **Effect** Zoo

@philip_schwarz

John A De Goes

@jdegoes

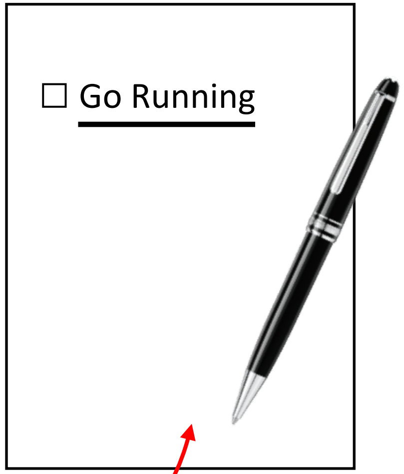
slideshare https://www.slideshare.net/jdegoes/one-monad-to-rule-them-all

YouTube https://youtu.be/POUEz8XHMhE

John A De Goes
@jdegoes

Every **effect** can be thought of as **doing something**.

If we want to **turn that into a** **value**, then **instead of** **doing something**, we turn that into a **description** of **doing something**.



So **instead of** **going running**, we'll turn that into a **piece of paper** **that says** **go running** **on it**.

Only, in **Scala** we don't actually write on a **piece of paper**, we end up **building a** **data structure** **that** **describes** **the act of** **going running**.

John A De Goes
@jdegoes

I'll give you a **very simple example** here of **a little program that has effects in it**.

```
def monitor: Boolean = {
  if (sensor.tripped) {
    securityCompany.call()
    true
  } else false
}
```

It is actually **doing stuff** and this is **not a value** right now. This is a **side effecting procedure**. It's a method called **monitor** and what it does is it checks to see if a sensor is tripped and if it is tripped it calls the security company returning true, otherwise if the sensor is not tripped it returns false.

Now this is **a piece of side effecting code**. The simplest possible way for us to **transform this into a value** is to build a **mini language**.

We will build a **data structure**, that's that **sheet of paper**, that will allow us to **describe** these operations.

John A De Goes
@jdegoes

You can do that more or less by rote. In this case I am going to call this data structure an **alarm**, and this **alarm** is going to be a **sealed trait**, so it's going to be an **enumeration**, a **sum type**, and it is going to have **three different instructions** in it

```
sealed trait Alarm[+A]
case class Return      [A](v   : A)                       extends Alarm[A]
case class CheckTripped[A](f   : Boolean => Alarm[A])  extends Alarm[A]
case class Call        [A](next: Alarm[A])                extends Alarm[A]
```

It is going to have a `Return` **instruction**, which **returns a value**, it is going to have a `CheckTripped` **instruction** which allows us to look and see if the sensor has been tripped and to choose to **return different** `Alarms` in the case that it is tripped or not tripped, and finally it is going to have a `Call` **instruction** that allows us to describe the act of calling the security company, **as well as whatever we want to do after** calling the security company.

John A De Goes
@jdegoes

Now, using this extraordinarily simple, immutable data structure, **we can create a model of the side effecting program** that you saw before. And it is quite simple, the type of our value will be **Alarm** of **Boolean**, this is an ordinary, immutable value. And what we do is we use the **CheckTripped operation** as the first **operation** in our program. And we pass it a function that will be passed a Boolean value, whether or not the sensor was tripped, and if it was tripped we are going to **Return** another **Alarm**, value, which is going to call the security company and then **Return true**, and if the sensor is not tripped we are just going to immediately **Return false**.

```scala
val check: Alarm[Boolean] =
  CheckTripped( tripped =>
    if (tripped)
      Call(Return(true))
    else
      Return(false)
  )
```

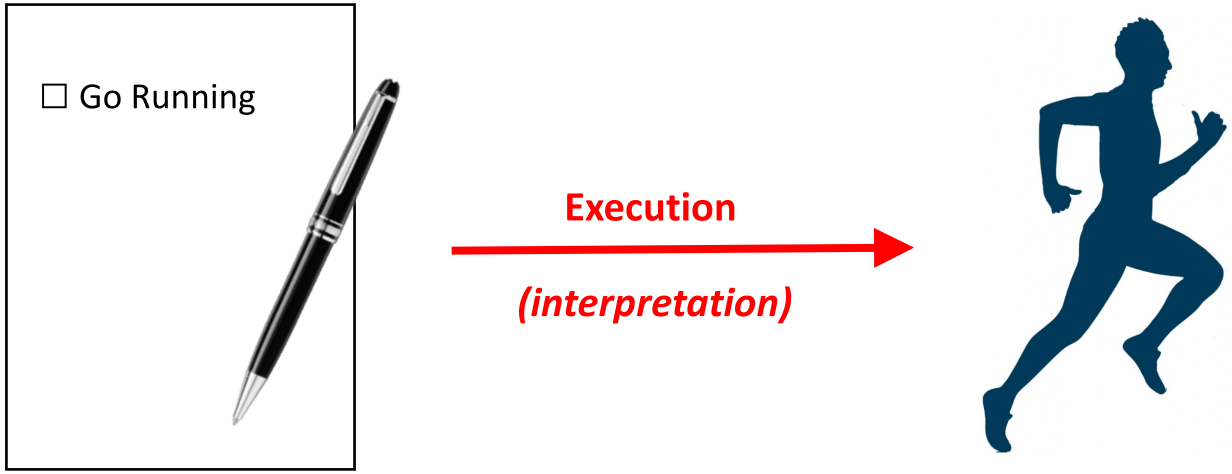
```scala
sealed trait Alarm[+A]
case class Return       [A](v    : A)                        extends Alarm[A]
case class CheckTripped[A](f    : Boolean => Alarm[A]) extends Alarm[A]
case class Call         [A](next: Alarm[A])                extends Alarm[A]
```

We have created a declarative description of the preceding side effecting program. **There are no side effects here, everything is a data structure and in fact it is an immutable data structure**.

Now, this value is useful as an intermediate form in our program, because now **we can store it in data structures, we can accept it in our functions, we can return it from our functions, we can build combinators that act on values of this type**, **however, it is not actually going to interact with the real world**.

John A De Goes
@jdegoes

To interact with the real world, we need to interpret this data structure into the side effects that it represents. And this is called execution, or interpretation.



To do this in the case of the Alarm data structure, we simply match against the three different cases, and if it is Return, we return that value, if it is CheckTripped, we do what we did before, we check if the sensor is tripped, we feed that result into f, and then we interpret the result of calling that. And then finally, for Call(next) we call the security company and then we interpret the remainder of the program.

```
def interpret[A](alarm: Alarm[A]): A = alarm match {
  case Return(v) => v
  case CheckTripped(f) => interpret(f(sensor.tripped))
  case Call(next) => securityCompany.call(); interpret(next)
}
```

John A De Goes

@jdegoes

```scala
def interpret[A](alarm: Alarm[A]): A = alarm match {
  case Return(v) => v
  case CheckTripped(f) => interpret(f(sensor.tripped))
  case Call(next) => securityCompany.call(); interpret(next)
}
```

This interpret function is not a pure function, it takes this immutable data structure and it interprets it into the side effects that it describes, allowing us to regain the sort of real world practicality of the preceding program, without sacrificing the fact that now we can use this data structure in most places in our program.

So this is a typical example of what a functional effect is, but in general:

> "A **functional effect** is an **immutable data type** equipped with a set of core **operations** that together provide a complete, type-safe model of a **domain concern**."
>
> — John A. De Goes

Every single functional effect out there satisfies that definition. They look quite different. Not all of them look like `Alarm`. I specifically chose `Alarm` because you have never seen anything like it before and probably never will again. It is not very realistic but it is an example of a functional effect, because we had our data type, it was immutable, it had three different operations in it and together we were able to create a complete model of the preceding side-effecting code.

quick recap

immutable data structure that can be used to create a model of the side effecting code

2

```
sealed trait Alarm[+A]
case class Return      [A](v    : A)                        extends Alarm[A]
case class CheckTripped[A](f    : Boolean => Alarm[A]) extends Alarm[A]
case class Call        [A](next: Alarm[A])                extends Alarm[A]
```

1

a piece of side effecting code

```
def monitor: Boolean = {
  if (sensor.tripped) {
    securityCompany.call()
    true
  } else false
}
```

immutable data structure (no side effects here) that is a declarative description of the side effecting code

```
val check: Alarm[Boolean] =
  CheckTripped( tripped =>
    if (tripped)
      Call(Return(true))
    else
      Return(false)
  )
```

3

impure function that takes the immutable data structure and interprets it into the side effects that it describes

4

```
def interpret[A](alarm: Alarm[A]): A = alarm match {
  case Return(v) => v
  case CheckTripped(f) => interpret(f(sensor.tripped))
  case Call(next) => securityCompany.call(); interpret(next)
}
```

```scala
sealed trait Alarm[+A]
case class Return      [A](v    : A)                   extends Alarm[A]
case class CheckTripped[A](f    : Boolean => Alarm[A]) extends Alarm[A]
case class Call        [A](next: Alarm[A])             extends Alarm[A]
```

```scala
def interpret[A](alarm: Alarm[A]): A = alarm match {
  case Return(v) => v
  case CheckTripped(f) => interpret(f(sensor.tripped))
  case Call(next) => securityCompany.call(); interpret(next)
}
```

Let's have a go at using the **interpret** function:

- once when the sensor is **tripped**
- once when the sensor is **not tripped**

@philip_schwarz

```scala
sensor.tripped = true
```

```scala
interpret(
  CheckTripped( tripped =>
    if (tripped)
      Call(Return(true))
    else
      Return(false)
  )
)
```

```scala
interpret(
  if (true)
    Call(Return(true))
  else
    Return(false)
)
```

```scala
interpret(
  Call(Return(true))
)
```

```scala
securityCompany.call(); interpret(next)
```

```scala
securityCompany.call(); interpret(Return(true))
```

```scala
interpret(Return(true))
```

```scala
true
```

```scala
sensor.tripped = false
```

```scala
interpret(
  CheckTripped( tripped =>
    if (tripped)
      Call(Return(true))
    else
      Return(false)
  )
)
```

```scala
interpret(
  if (false)
    Call(Return(true))
  else
    Return(false)
)
```

```scala
interpret(
  Return(false)
)
```

```scala
false
```

John A De Goes
@jdegoes

For every concern out there, there already exists, or you can create, a functional effect to describe that domain, and I'll give you some common examples:

Concern	Effect	Execution
Optionality	Option[A]	null or A
Disjunction	Either[A,B]	A or B
Nondeterminism	List[A]	Option[A]
Input/Output	IO[A]	throw or A

If your concern is optionality, that is you want to compute and sometimes you are going to try and compute something but it is not going to be there, then you can use the effect called Option, which is built into Scala. It's a functional effect. It's an immutable data type and it has a set of instructions, operations, that allow us to build up programs that use the feature of that functional effect, which is optionality. And like all functional effects, we execute it, and at the end of the day, when we execute it, we get back either nothing, if it wasn't there, or we get back the A that was in the Option.

Disjunction, is another concern. So in some class of computation, we'll either produce one type of result or a different type of result entirely. An example would be errorful computation, so computation that can fail with some specific error. That's an example of the effect of disjunction, right? We are either going to fail with something on the left or we are going to succeed with something on the right. And this functional effect is also built into Scala using the Either data type. And when we execute it we either get back a Left of an A or we get back a Right of a B.

John A De Goes
@jdegoes

Concern	Effect	Execution
Optionality	Option[A]	null or A
Disjunction	Either[A,B]	A or B
Nondeterminism	List[A]	Option[A]
Input/Output	IO[A]	throw or A

Nondeterminism, less frequently used, is to do, for example, search, to solve problems in search, where we are looking for a solution satisfying a particular requirements, there is the List data type, of course we use the List data type just for ordinary storing of data, but we can also use it as a functional effect, a functional effect that allows us to explore possible solutions to a given problem and to filter those by one satisfying a given set of conditions. And when we execute or interpret that effect, we'll either get back a solution, or maybe our top ranked solution, or no solution at all, if no solutions were found, which corresponds to calling headOption on a List .

And then also another example of a functional effect not baked into Scala but also extremely important is the effect of Input/Output. So, when our programs interact with the external world, that functional effect is described by IO-like data types. So Cats IO, Monix Task, ZIO's ZIO data type and so on, Scalaz 7's Task type, all these allow us to describe input/output effects, effects between our program and the external sorrounding environment. And when we execute them, which is not a functional operation, we either get back some exception, some code failed, or we get back the A value that they succeeded with.

Optionality

(the Painful Way)

A functional effect for optionality

John A De Goes

@jdegoes

So I am going to give you an example of, a single fully worked example of a functional effect, and this is the effect of optionality, but it's in a way you have never seen before.

You know what the `Option` data type looks like in Scala. It has either Some or None, right? It is a simplification of the real deal that we are going to look at now.

The real deal is a full model of the functional effect of optionality. I'll talk about the relationship with `Option` at the end.

So I'll call this data type Maybe. This is going to be a functional effect for optionality, so if we are concerned with things that may or may not be there, this is the functional effect that we want to use.

A Maybe[A] can succeed with values of type A. However it can also fail to produce any value of type A.

`Maybe[A]`

succeeds with values of type A

A **functional effect**
for **optionality**

John A De Goes
@jdegoes

We are going to need four different **operations** to completely describe this **functional effect**:

Operation	Signature
Present	`A => Maybe[A]`
Absent	`Maybe[Nothing]`
Map	`(Maybe[A], A => B) => Maybe[B],`
Chain	`(Maybe[A], A => Maybe[B]) => Maybe[B],`

One **operation**, which I'll call Present, allows us to **take an A and stick it inside a Maybe of A**. This is when we have something and we want to stick it inside the **functional effect** to represent the fact that it's there.

Absent on the other hand is when we don't have anything but we need to create a Maybe that has some type. We are going to use that **operation** when we don't have it and we want to indicate that. We want to indicate that we don't have a value of that type, so we use the Absent **operation**.

The Map **operation** is when we have a Maybe of A and we also want to **map** that A into a B by supplying a function, so the pair of a Maybe of A and a function from A to B, you provide the Map **operation** those two things, and you get back a Maybe of B.

And then finally the Chain **operation** will be used when we have a Maybe of A and then based on that A we want to produce a Maybe of B, for some type B, and we want to combine both the Maybe of A together with that callback, into a single Maybe of B.

A **functional effect** for **optionality**

John A De Goes

@jdegoes

In order to implement this **functional effect**, all we have to do is define a **sealed trait** with the four **operations** we know we need:

```
sealed trait Maybe[+A]
case class Present[A](value: A) ............................................ extends Maybe[A]
case object Absent ......................................................... extends Maybe[Nothing]
case class     Map[A, B](maybe: Maybe[A], mapper:   A => B) ................. extends Maybe[B]
case class   Chain[A, B](first: Maybe[A], callback: A => Maybe[B]) extends Maybe[B]
```

- The Present **operation** simply stores the A.
- The Absent **operation** doesn't store anything.
- The Map **operation** stores the **Maybe** and the mapper function.
- And then the Chain **operation** stores the **Maybe** and then the callback.

Once we have defined these four **operations**, we can then define **map** and **flatMap** on the **Maybe** data type:

```
sealed trait Maybe[+A] { self =>
  def map[B](f: A => B): Maybe[B] = Map(self, f)
  def flatMap[B](f: A => Maybe[B]): Maybe[B] = Chain(self, f)
  …
}
```

Furthermore, we can define Present and Absent on the maybe companion object:

```
object Maybe {
  def present[A](value: A): Maybe[A] = Present(value)
  val absent: Maybe[Nothing] = Absent
}
```

A **functional effect** for **optionality**

John A De Goes
@jdegoes

We now have everything necessary to define an **interpreter** for the **effect** of **optionality**. This **interpreter** matches the **Maybe** data type, and it has to handle each of the four cases.

```
def interpret[Z, A](ifAbsent: Z, f: A => Z)(maybe: Maybe[A]): Z =
  maybe match {
    case Present(a) => f(a)
    case Absent => ifAbsent
    case Map(old, f0) => interpret(ifAbsent, f.compose(f0))(old)
    case Chain(old, f) => interpret(ifAbsent, a => interpret(ifAbsent, f)(f(a)))(old)
  }
```

This function doesn't compile as it stands, but the problem is easily resolved (see next two slides).

This function is going to **interpret** it (the **Maybe[A]**) to some type **Z** and the user who is **interpreting** this data type, has to supply **some function or some value** called **ifAbsent**, which **will be returned in the event that the computation fails to produce a value of type A**. And also they have to supply **another function**, which I am calling **f** here, it could be called **ifPresent**, which **will be called if the computation succeeds to produce an A**. And in the end the **interpreter** is going to return a **Z**…

In the **interpret** function above, the names of the fields of **Map** and **Chain** differ from their corresponding names in the **Maybe** trait (defined in the previous slide), and this may be confusing, so here is the trait again, just for reference.

```
sealed trait Maybe[+A]
case class Present[A](value: A)                                 extends Maybe[A]
case object Absent                                              extends Maybe[Nothing]
case class     Map[A, B](maybe: Maybe[A], mapper:   A => B)      extends Maybe[B]
case class   Chain[A, B](first: Maybe[A], callback: A => Maybe[B]) extends Maybe[B]
```

Here are the errors I got when I tried to compile the **interpret** function.
See next slide for how I addressed them.

```scala
def interpret[Z, A](ifAbsent: Z, f: A ⇒ Z)(maybe: Maybe[A]): Z =
  maybe match {
    case Present(a) ⇒ f(a)
    case Absent ⇒ ifAbsent
    case Map(old, f0) ⇒
      interpret(ifAbsent, f.compose(f0))(old)
    case Chain(old, f) ⇒
      interpret(ifAbsent, a ⇒ interpret(ifAbsent, f)(f(a)))(old)
  }
```

Type mismatch, expected: Maybe[Nothing], actual: Maybe[A]

Here on the left is John's original **interpret** function, and on the right you can see the changes that I made to get it to compile and to make it slightly easier for me to understand.

@philip_schwarz

```scala
def interpret[Z, A](ifAbsent: Z, f: A => Z)(maybe: Maybe[A]): Z =
  maybe match {
    case Present(a) => f(a)
    case Absent => ifAbsent
    case Map(old, f0) =>
      interpret(ifAbsent, f.compose(f0))(old)
    case Chain(old, f) =>
      interpret(ifAbsent, a => interpret(ifAbsent, f)(f(a)))(old)
  }
```

```scala
def interpret[Z, A, B](ifAbsent: Z, f: B => Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) => f(b)
    case Absent => ifAbsent
    case Map(old: Maybe[A], g: (A => B)) =>
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A => Maybe[B])) =>
      interpret(ifAbsent, (a:A) => interpret(ifAbsent, f)(g(a)))(old)
  }
```

```scala
def interpret[Z, A](ifAbsent: Z, f: A ⇒ Z)(maybe: Maybe[A]): Z =
  maybe match {
    case Present(a) ⇒ f(a)
    case Absent ⇒ ifAbsent
    case Map(old, f0) ⇒
      interpret(ifAbsent, f.compose(f0))(old)
    case Chain(old, f) ⇒
      interpret(ifAbsent, a ⇒ interpret(ifAbsent, f)(f(a)))(old)
  }
```

```scala
def interpret[Z, A, B](ifAbsent: Z, f: B ⇒ Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) ⇒ f(b)
    case Absent ⇒ ifAbsent
    case Map(old: Maybe[A], g: (A ⇒ B)) ⇒
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A ⇒ Maybe[B])) ⇒
      interpret(ifAbsent, (a:A) ⇒ interpret(ifAbsent, f)(g(a)))(old)
  }
```

```scala
def interpret[Z, A](ifAbsent: Z, f: A => Z)(maybe: Maybe[A]): Z =
  maybe match {
    case Present(a) => f(a)
    case Absent => ifAbsent
    case Map(old, f0) =>
      interpret(ifAbsent, f.compose(f0))(old)
    case Chain(old, f) =>
      interpret(ifAbsent, a => interpret(ifAbsent, f)(f(a)))(old)
  }
```

```scala
def interpret[Z, A, B](ifAbsent: Z, f: B => Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) => f(b)
    case Absent => ifAbsent
    case Map(old: Maybe[A], g: (A => B)) =>
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A => Maybe[B])) =>
      interpret(ifAbsent, (a:A) => interpret(ifAbsent, f)(g(a)))(old)
  }
```

A **functional effect** for **optionality**

John A De Goes

@jdegoes

```scala
def interpret[Z, A, B](ifAbsent: Z, f: B => Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) => f(b)
    case Absent => ifAbsent
    case Map(old: Maybe[A], g: (A => B)) =>
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A => Maybe[B])) =>
      interpret(ifAbsent, (a:A) => interpret(ifAbsent, f)(g(a)))(old)
  }
```

This is the slightly modified version of **interpret** mentioned in the previous slide, so in John's commentary below I have replaced A with B and got him to mention **g** (which in his version is called f0).

…this function is polymorphic in **Z**, so it ends up getting a **Z**, either from **ifAbsent**, if there was no **A** inside that **Maybe**, or it gets it from **f** if there was an **A** inside that **Maybe**. It is going to get it from one of those two places and end up returning that **Z**. How it does this is it matches against the **Maybe** data type:

- If the **B** is present it calls **f**(b) to immediately return a **Z**
- If the **B** is absent, then it just returns the **ifAbsent** value, to return the **default**.
- If the case is Map, then it interprets the thing that is being **mapped** and composes the **mapper function g** together with the **f** function and passes along the **ifAbsent default** value
- And then finally in the case of Chain it's another relatively straightforward **recursion**, it just passes things along, drills down into the inner data structure and then maps the output by the **f** function.

So you can **follow the types** here if you want to do this, the compiler will help you write this function, you don't have to think about the implications, just try to get the types right and you'll end up with something that is correct.

Here again is the **Maybe** trait, just for reference

```scala
sealed trait Maybe[+A]
case class Present[A](value: A)                                  extends Maybe[A]
case object Absent                                               extends Maybe[Nothing]
case class     Map[A, B](maybe: Maybe[A], mapper:   A => B)       extends Maybe[B]
case class   Chain[A, B](first: Maybe[A], callback: A => Maybe[B]) extends Maybe[B]
```

```scala
def interpret[Z, A, B](ifAbsent: Z, f: B => Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) => f(b)
    case Absent => ifAbsent
    case Map(old: Maybe[A], g: (A => B)) =>
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A => Maybe[B])) =>
      interpret(ifAbsent, (a:A) => interpret(ifAbsent, f)(g(a)))(old)
  }
```

Again, this is the slightly modified **interpret** fuction.

A **functional effect** for **optionality**

John A De Goes

@jdegoes

So this **interprets** the four cases and notice how **it is much more complex than the Option type built into Scala**. **Why?**

The Option type built into Scala only has two cases. And that's because the Option type built into Scala does just-in-time interpretation of the instructions map and flatMap:

```scala
sealed abstract class Option[+A] extends … { self =>
…
def get: A
…
@inline final def map[B](f: A => B): Option[B] =
  if (isEmpty) None else Some(f(this.get))
…
@inline final def flatMap[B](f: A => Option[B]): Option[B] =
  if (isEmpty) None else f(this.get)
…
}
```

```scala
object Option {
…
def apply[A](x: A): Option[A] =
  if (x == null) None else Some(x)

def empty[A] : Option[A] = None
…
final def isEmpty: Boolean = this eq None
…
}
```

Rather than building up a full description of the effect, what happens is that it takes shortcuts. The **map** and **flatMap** on Option will look a that data type and if it is None for example, it will immediately return None. If it is Some, it will immediately deconstruct that Some, apply your mapper function to it and return a new Some.

So in essence, what is happening is **the Option data type in Scala, even though it is a functional effect, it is doing a type of just-in-time interpretation, it is taking shortcuts** and returning you a maximally reduced data structure right away, which is why **it can afford to be a whole lot simpler** than the version I have shown you.

But keep in mind that is a simplification and **in many types of real world functional effects, you can't make that simplification, you need to store every single instruction that you want to expose to the end user of your API**.

```scala
object Maybe {
  def present[A](value: A): Maybe[A] =
    Present(value)
  val absent: Maybe[Nothing] =
    Absent
}
```


```scala
def interpret[Z, A, B](ifAbsent: Z, f: B => Z)(maybe: Maybe[B]): Z =
  maybe match {
    case Present(b) => f(b)
    case Absent => ifAbsent
    case Map(old: Maybe[A], g: (A => B)) =>
      interpret(ifAbsent, f compose g)(old)
    case Chain(old: Maybe[A], g: (A => Maybe[B])) =>
      interpret(ifAbsent, (a:A) => interpret(ifAbsent, f)(g(a)))(old)
  }
```

```scala
sealed trait Maybe[+A] { self =>
  def map[B](f: A => B): Maybe[B] = Map(self, f)
  def flatMap[B](f: A => Maybe[B]): Maybe[B] = Chain(self, f)
}
case class Present[A](value: A)                                      extends Maybe[A]
case object Absent                                                   extends Maybe[Nothing]
case class     Map[A, B](maybe: Maybe[A], mapper:   A => B)          extends Maybe[B]
case class   Chain[A, B](first: Maybe[A], callback: A => Maybe[B])   extends Maybe[B]
```

```scala
val increment: Int => Int = n => n + 1
val double: Int => Int = n => 2 * n
```

```scala
def headMaybe: List[Int] => Maybe[Int] =
  as => if (as.isEmpty) Maybe.absent else Maybe.present(as(0))
```

```scala
// Option.empty[Int].fold(0)(double)
assert( interpret(0, double)(Maybe.absent) == 0)

// Option.empty[Int].map(increment).fold(0)(double)
val noInt: Maybe [Int] = Maybe.absent
assert( interpret(0 , double)(noInt.map(increment)) == 0)

// Option.empty[List[Int]].flatMap(_.headOption).fold(0)(double)
val noIntList: Maybe [List[Int]] = Maybe.absent
assert( interpret(0 , double)(noIntList.flatMap(headMaybe)) == 0)

// Some(123).fold(0)(double)
assert( interpret(0, double)(Maybe.present(123)) == 246)

// Some(123).map(increment).fold(0)(double)
assert( interpret(0, double)(Maybe.present(123).map(increment)
) == 248)
```

```scala
// Some(List(1,2,3)).flatMap(_.headOption).fold(0)(double)
assert( interpret(0, double)
  (Maybe.present(List(1,2,3)).flatMap(headMaybe)) == 2)

// Some(List(1,2,3)).flatMap(_.headOption).map(increment).fold(0)(double)
assert( interpret(0, double)(Maybe.present(List(1,2,3))
  flatMap { xs => headMaybe(xs)
    map { y => increment(y) } }
) == 4)

// Some(List(1,2,3)).flatMap(_.headOption).map(increment).fold(0)(double)
assert( interpret(0, double)(
  for {
    xs <- Maybe.present(List(1, 2, 3))
    y  <- headMaybe(xs)
  } yield increment(y)
) == 4)
```

```
def monitor: Boolean = {
  if (sensor.tripped) {
    securityCompany.call()
    true
  } else false
}
```

```
val check: Alarm[Boolean] =
  CheckTripped( tripped =>
    if (tripped)
      Call(Return(true))
    else
      Return(false)
  )
```

Note how earlier, when we wanted to model the side-effecting code that sent an email if a sensor was tripped, we created the **model** by manually putting together a data structure ourselves…

…whereas when it comes to **modeling** code exhibiting **optionality**, we programmatically call **Maybe** functions that create the data structure (**model**) for us.

```
Maybe.present(List(1,2,3)) flatMap { xs =>
  headMaybe(xs) map { y =>
    increment(y)
  }
}
```

```
for {
  xs <- Maybe.present(List(1, 2, 3))
  y <- headMaybe(xs)
} yield increment(y)
```

Every common operation in a functional effect system has a corresponding type class in functional programming. You don't need to know this, but it is helpful to know this, if you have ever used type classes from Cats or Scalaz, you have run into Applicative and Functor and Monad and Apply and MonadPlus and lots of other type classes. It turns out that every single type class gives you an operation that you can use in your functional effect. More powerful functional effects have more operations. And the set of all operations given to you by your functional effect type determines how powerful it is and what types of things you can do with it.

Operation	Signature	Type Class
`pure/point`	`A => F[A]`	`Applicative`
`empty/zero`	`F[Nothing]`	`MonadPlus`
`map`	`(F[A], A => B) => F[B]`	`Functor`
`flatMap`	`(F[A], A => F[B]) => F[B]`	`Monad`
`zip/ap`	`(F[A], F[B]) => F[(A,B)]`	`Apply`

John A De Goes

@jdegoes

The pure or point operation from Applicative gives you the ability to take an A and lift it up into your functional effect system. It is the equivalent of returning a value inside your functional effect. It is the Some constructor in Option. It is the List singleton's constructor in List. It is the Future.successful in Future. And so forth.

empty or zero, some types have this notion of an empty or failure type.

Other types have the ability to map over them. All functional effects, almost all, have the ability to map over their contents, which corresponds to taking that return statement and changing it into a value of another type, turning an Option of an Int into an Option of a String by converting the Int to a String.

Operation	Signature	Type Class
pure/point	A => F[A]	Applicative
empty/zero	F[Nothing]	MonadPlus
map	(F[A], A => B) => F[B]	Functor
flatMap	(F[A], A => F[B]) => F[B]	Monad
zip/ap	(F[A], F[B]) => F[(A,B)]	Apply

flatMap is a very powerful capability allowing you to chain two functional effects together in sequence such that the second functional effect depends on the runtime value produced by the first. When you call flatMap, you supply the first functional effect and then you also specify a callback and that callback will be called with the value of the first functional effect, assuming one is ever produced. Of course some **functional effects**, like **Option**, can fail, in which case they'll never call your callback, but also some functional effects like **Future**, for example, can succeed at some point in the future, in which case your callback will be called and you'll get a chance to return the rest of your computation and **the flatMap operation is responsible for fusing those two things together, the old functional effect and its chained successor, into a single functional effect**.

And then finally **zip**, otherwise known as **ap**, is capable of **taking two functional effects**, F[A] **and** F[B] **and zipping them together to get an** F **of a tuple of** A **and** B. **It is not as powerful as flatMap but it is still a powerful operation and it is the minimum needed to have compositional semantics on your functional effect**, **you need to zip two options together, zip two parsers together, zip two futures together, you need the ability to take two different effects and combine them together into a single effect to solve most classes of problems**.

Nearly all functional effects in existence support the pure/point operation, the map operation and then zip. If you don't support zip, if you just support pure and map, it is not that useful, of a functional effect, almost all of your functional effects out there are going to support at least zip and some extremely powerful ones support Monad which gives you flatMap, which allows you to do two operations in sequence such that the second one depends on the runtime value produced by the first one.

John A De Goes

@jdegoes

John A De Goes

@jdegoes

A lot of functional effects support flatMap, a lot. Parsers, Futures, Options, Lists, all kinds of functional effects support this capability. Even the Alarm one that I showed you supports this capability. And that's because a lot of the real world is sequential. You do something and then you do something later and the thing that you do later depends on what you did before. Depends on the result of what you did before. That is a sequential flow, it is the most complex kind of sequential flow, because it is context sensitive. You can change your mind and do different things based on what happened before. That's flatMap. That's Monadic.

As a result, Scala actually has a special syntax for data types that support flatMap and map and this is the for comprehension syntax, and it reads very procedurally:

```
for {
  user    <- lookupUser(userId)
  profile <- user.profile
  pic     <- profile.picUrl
} yield pic
```

```
lookupUser(userId).flatMap(user =>
  user.profile.flatMap(profile =>
    profile.picUrl.map(pic =>
      pic)))
```

```
case class User(profile: Option[Profile])
case class Profile(picUrl: Option[Url])
case class Url(value: String)
def lookupUser(userId: Int): Option[User] = ???
var userId = ???
```

You are going to look up your **user**, and then you are going to get their **profile** and then you are going to get their **picture url**, and you can imagine all these things returning `Option`. And Scala will desugar this into a bunch of flatMaps, followed by a final map, allowing you to use functional effects that support sequentiality in a way whose visual appearance resembles that sequential flow of operations, with the scoping rules that you would expect, that is to say, inside this for comprehension on the left, in the line that says **pic**, I have access to both **profile** and **user**, I have access to both of those variables in that scope. In the line that says profile I have access to **user**, and in the yield statement I have access to all three variables, which is the way you would expect scoping to work if these were statements.

Tour of the Effect Zoo

John A De Goes
@jdegoes

So every functional effect is an immutable data type, together with the operations it provides for addressing some business concern, and at the end of the day, every functional effect system, we need to be able to interpret it into something else that gives it meaning. This interpretation is fold on Option, it is unsafeRun on Task, there is always an interpretation function for all of these, it is run on the State Monad.

It allows us to take this model that describes our business concern and translate it into something that we can use.

Let's take a brief look at some of the effects out there in the wild, some of which you have already seen because they are built into Scala, but a couple of which may be new for you.

Tour of the Effect Zoo

Option[A] – the functional effect of optionality

John A De Goes
@jdegoes

First the effect of optionality. Either something is there or it is not:

```
sealed trait Option[+A]
final case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

The core operations of Option are the Some and None constructors and map and flatMap.

```
// Core operations:
def some[A](v: A): Option[A] = Some(v)
val none: Option[Nothing] = None
def map[A, B](o: Option[A], f: A => B): Option[B]
def flatMap[A, B](o: Option[A], f: A => Option[B]): Option[B]
```

And then its execution/interpretation is the fold function on Option:

```
// Execution / Interpretation:
def fold[Z](z: Z)(f: A => Z)(o: Option[A]): Z
```

We specify what to do, what to return if it wasn’t there and what to return if it was there.

Tour of the Effect Zoo

`Option[A]` – the **functional effect** of **optionality**

John A De Goes

@jdegoes

And we can use it in **for comprehensions**, like I showed before:

```
for {
  user    <- lookupUser(userId)
  profile <- user.profile
  pic     <- profile.picUrl
} yield pic
```

Tour of the Effect Zoo

Either[A,B] – the functional effect of failure

John A De Goes
@jdegoes

The functional effect of failure is used when we have computations that may fail with a specific type of value. Either has two types of value, a left and a right. Left is used to indicate failure and Right is used to indicate success:

```
sealed trait Either[+E, +A]
final case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]
```

Its core operations are constructing a left and a right, mapping and flatMapping:

```
// Core operations:
def left[E](e: E): Either[E, Nothing] = Left(e)
def right[A](a: A): Either[Nothing, A] = Right(a)
def map[E, A, B](o: Either[E, A], f: A => B): Either[E, B]
def flatMap[E, A, B](o: Either[E, A], f: A => Either[E, B]): Either[E, B]
```

And then to execute or interpret an Either we fold over it

```
// Execution / Interpretation:
def fold[Z, E, A](left: E => Z, right: A => Z)(e: Either[E, A]): Z
```

specifying what to do on the left hand case and what to do on the right hand case.

Tour of the Effect Zoo

Either[A,B] – the functional effect of failure

John A De Goes

@jdegoes

And because this supports **map** and **flatMap** we can use it in **for comprehensions**, in which case we are **flatMapping** over the success case

```
for {
  user    <- decodeUser(json1)
  profile <- decodeProfile(json2)
  pic     <- decodeImage(profile.encPic)
} yield (user, profile, pic)
```

So if there is a `Left` case, if one of these methods, like decodeProfile returns `Left`, that **short-circuits the entire computation, we achieve the short-circuiting behaviour of exception handling without actually having exceptions in our code**.

Tour of the Effect Zoo

Writer[W,A] – the functional effect of logging

John A De Goes
@jdegoes

The Writer functional effect is less familiar, you may not have seen this before. Writer is actually dual to Either, only in this case I am using a very specialised variant of Writer that happens to be the most common. Writer is basically a tuple. On the LHS it accumulates a vector of some type W, that's your log. So Writer allows you to log stuff, like log strings, whatever, and those get accumulated on the LHS of the tuple. And every Writer effect can also produce a value of type A. So the Writer functional effect, it cannot fail, it can only succeed, and it can accumulate a log as you are succeding with values of different type.

```
final case class Writer[+W, +A](run: (Vector[W], A))
```

The core operations of Writer are pure, which allows you to lift a value into the Writer effect, write, which allows you to add to that log, and then map and flatMap like we have seen before. :

```
// Core operations:
def pure[A](a: A): Writer[Nothing, A] = Writer((Vector(), a))
def write[W](w: W): Writer[W, Unit] = Writer((Vector(w), ()))
def map[W, A, B](o: Writer[W, A], f: A => B): Writer[W, B]
def flatMap[W, A, B](o: Writer[W, A], f: A => Writer[W, B]): Writer[W, B]
```

And then how you run that, you just pull out the tuple of the Vector and then your success value:

```
// Execution / Interpretation:
def run[W, A](writer: Writer[W, A]): (Vector[W], A)
```

That gives you the log and then the value that the Writer data type succeeded with.

Tour of the Effect Zoo

`Writer[W,A]` – the **functional effect** of **logging**

Because it has **map** and **flatMap** like the other ones you can use this inside **for comprehensions**

```
for {
  user <- pure(findUser())
     _ <- log(s"Got user: $user")
     _ <- pure(getProfile(user))
     _ <- log(s"Got profile: $profile")
} yield user
```

And you can interleave, for example, success values with **log** statements, and you end up accumulating those **log** statements, in this case strings, inside the vector that you get when you **run** that **functional effect**.

John A De Goes
@jdegoes

In the above, either **log** should be **write** or **log** is an alias for **write**

Tour of the Effect Zoo

State[S,A] – the functional effect of state

John A De Goes
@jdegoes

State is another very common functional effect that allows you to model stateful computations. And the State functional effect is basically a function. At least this is the short-circuited version. We could do the full on different instruction version that I did for optionality but we were only going to do that once. Here we are taking a shortcut and we are defining it as a function that takes the old state and returns the new state and a value of type A. So State cannot fail. State can only change the state, when you call run, it can change the state, and it is always going to succeed with a value of type A.

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

The core operations of State are to take an A value and to succeed with that value without changing state, to get the state and to set the state, and then of course map and flatMap, like we have seen with all these functional effects:

```
// Core operations:
def pure[S, A](a: A): State[S, A] = State[S, A](s => (s, a))
def get[S]: State[S, S] = State[S, S](s => (s, s))
def set[S](s: S): State[S, Unit] = State[S, S](_ => (s, ()))
def map[S, A, B](o: State[S, A], f: A => B): State[S, B]
def flatMap[S, A, B](o: State[S, A], f: A => State[S, B]): State[S, B]
```

To run a State we have to supply the initial state as well as the state type, and then out of that we get the new state and the success value:

```
// Execution / Interpretation:
def run[S, A](s: S, state: State[S, A]): (S, A)
```

Tour of the Effect Zoo

`State[S,A]` – the **functional effect** of **state**

John A De Goes
@jdegoes

Because this **functional effect**, like the other ones, supports **map** and **flatMap**, it means that we can use it inside **for comprehensions**:

```
for {
  _ <- set(0)
  v <- get
  _ <- set(v + 1)
  v <- get
} yield v
```

And we can write code that looks like this, like it is actually incrementing stuff. It is **setting** a value to be zero, it is **getting** it, **setting** it to zero plus one, and then it is **getting** it again, and if you actually **run** that **functional effect**, then you are going to end up with 1 out of that, which is what you would expect, **it looks like procedural code but in fact it is not, it is purely functional and it is operating on immutable data**.

Tour of the Effect Zoo

Reader[R,A] – the functional effect of reader

John A De Goes
@jdegoes

Another less common type is the Reader effect, and the Reader functional effect allows us to thread access to some environment of type R throughout our program without having to do any of that plumbing. And we can access that R at any point we want. So it is there, always in the background, it is like a context, it is the environment in which our program runs, and we can pull it out of thin air any time we want, but we don't have to deal with it unless we want to. And it can be defined by a simple function from R to A:

```
final case class Reader[-R, +A](run: R => A)
```

The core operations are pure, like we have seen before, allowing us to take an A and lift it up into an effect, the Reader functional effect, environment, which basically allows us to pull that R into the success value of the Reader, and then map and flatMap:

```
// Core operations:
def pure[A](a: A): Reader[Any, A] = Reader[Any, A](_ => a)
def environment: Reader[R, R] = Reader[R, R](r => r)
def map[R, A, B](r: Reader[R, A], f: A => B): Reader[R, B]
def flatMap[R, A, B](r: Reader[R, A], f: A => Reader[R, B]): Reader[R, B]
```

And then to execute or interpret this functional effect we have to give it an R. That's the R required by the Reader, and then it can give us back the A:

```
// Execution / Interpretation:
def provide[R, A](r: R, reader: Reader[R, A]): A
```

Tour of the Effect Zoo

Reader[R,A] – the functional effect of reader

John A De Goes
@jdegoes

Because it supports map and flatMap, we can use this in for comprehensions:

```
for {
  port    <- environment[Config].map(_.port)
  server  <- environment[Config].map(_.server)
  retries <- environment[Config].map(_.retries)
} yield (port, server, retries)
```

In this case I just pull the config out of the environment and I separately pull out the port and the server and the retries and I yield a tuple of the results.

Tour of the Effect Zoo

IO[A] – the functional effect of asynchronous input/output

And finally, the last functional effect that we'll look at is the effect of asynchronous input and output, and you can define your own very simple type for async I/O, by creating a case class with that unsafeRun signature. The unsafeRun, you give it a callback and it will call it at some point in the future. This is the essence of asynchronous I/O:

```
final case class IO[+A](unsafeRun: (Try[A] => Unit) => Unit)
```

And the core operations are sync for synchronous I/O, async for asynchronous I/O, fail, if you want to fail this thing, and then map and flatMap:

```
// Core operations:
def sync[A](v: => A): IO[A] = IO(_(Success(v)))
def async[A](r: (Try[A] => Unit) => Unit): IO[A] = IO(r)
def fail(t: Throwable): IO[Nothing] = IO(_(Failure(t)))
def map[A, B](o: IO[A], f: A => B): IO[B]
def flatMap[A, B](o: IO[A], f: A => IO[B]): IO[B]
```

And then unsafeRun, you have to give it the IO which you want to run and then you give it a callback and it will call your callback at some point later with either a success or a failure:

```
// Execution / Interpretation:
def unsafeRun[A](io: IO[A], k: Try[A] => Unit): Unit
```

John A De Goes
@jdegoes

Tour of the Effect Zoo

John A De Goes

@jdegoes

So what do all these things have in common? They are all immutable data structures. Every single one of them.

They are all equipped with operations that allow us to compose these things together.

Nearly all of them supported, actually all of them, supported pure and map and flatMap, which allow us to build up and compose sequential things together, which is very very common when you are dealing with functional effects.

And then all of them, without exception, had some way to interpret or execute them.

These are the building blocks of functional effects.

Functional effects are always, always, always, immutable data types that declaratively describe a bunch of different operations in some business domain, that you can end up interpreting to translate into something that is lower level than that specific concern, like we can translate away from optionality by providing a default value. You can translate away from error handling by unifying the left and right of Either, and so on and so forth, they all allow us to escape that concern and move it into something that's lower level, which is a key property of building programs compositionally and modularly.

I liked John's talk a lot. I found it very instructive. There is a lot more great content in it. Go take a look, if you haven't already.

@philip_schwarz

John A De Goes

@jdegoes

slideshare https://www.slideshare.net/jdegoes/one-monad-to-rule-them-all

YouTube https://youtu.be/POUEz8XHMhE