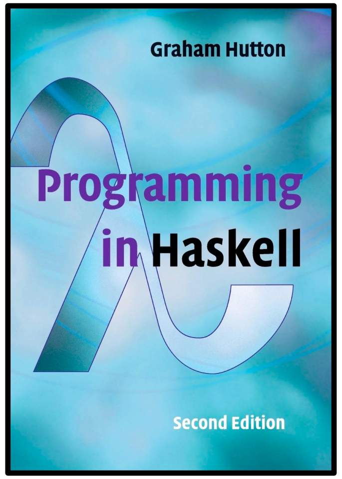
Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as the **impure functions** in the **Game of Life** are translated from **Haskell** into **Scala**,

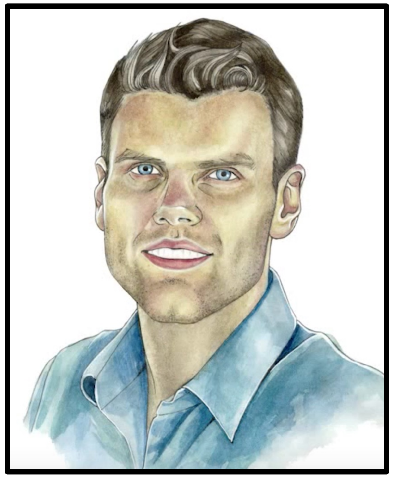
deepening you understanding of the **IO monad** in the process

(Part 2)

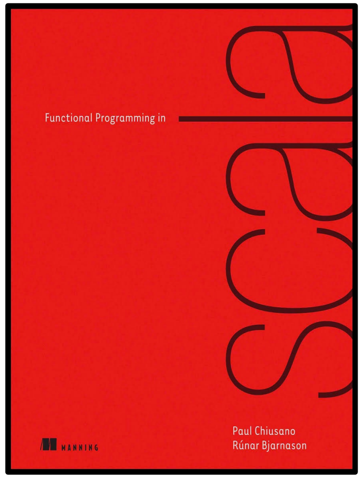
through the work of



Graham Hutton
@haskellhutt



Runar Bjarnason
@runarorama



FP in Scala

Paul Chiusano
@pchiusano

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

@philip_schwarz

In part 1 we translated some of the **Game Of Life** functions from **Haskell** into **Scala**. The functions that we translated were the **pure functions**, and on the next slide you can see the resulting **Scala** functions.

PURE FUNCTIONS

```
type Pos = (Int, Int)

type Board = List[Pos]

val width = 20

val height = 20
```

```
def neighbs(p: Pos): List[Pos] = p match {
  case (x,y) => List(
    (x - 1, y - 1), (x, y - 1),
    (x + 1, y - 1), (x - 1, y    ),
    (x + 1, y    ), (x - 1, y + 1),
    (x,     y + 1), (x + 1, y + 1) ) map wrap }

def wrap(p:Pos): Pos = p match {
  case (x, y) => (((x - 1) % width) + 1,
                  ((y - 1) % height) + 1) }
```

```
def isAlive(b: Board)(p: Pos): Boolean =
  b contains p

def isEmpty(b: Board)(p: Pos): Boolean =
  !(isAlive(b)(p))

def liveneighbs(b:Board)(p: Pos): Int =
  neighbs(p).filter(isAlive(b)).length
```

```
def survivors(b: Board): List[Pos] =
  for {
    p <- b
    if List(2,3) contains liveneighbs(b)(p)
  } yield p

def births(b: Board): List[Pos] =
  for {
    p <- rmdups(b flatMap neighbs)
    if isEmpty(b)(p)
    if liveneighbs(b)(p) == 3
  } yield p

def rmdups[A](l: List[A]): List[A] = l match {
  case Nil => Nil
  case x::xs => x::rmdups(xs filter(_ != x)) }

def nextgen(b: Board): Board =
  survivors(b) ++ births(b)
```

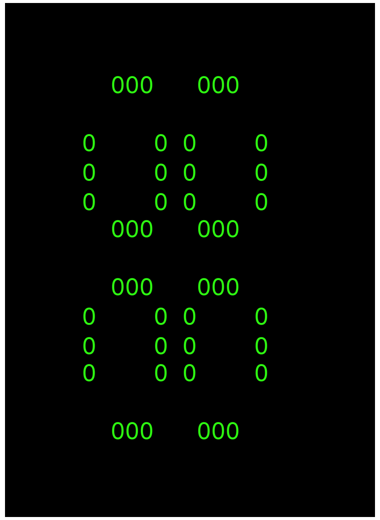
```
val glider: Board = List((4,2),(2,3),(4,3),(3,4),(4,4))

val gliderNext: Board = List((3,2),(4,3),(5,3),(3,4),(4,4))

val pulsar: Board = List(
   (4, 2),(5, 2),(6, 2),(10, 2),(11, 2),(12, 2),

          (2, 4),(7, 4),( 9, 4),(14, 4),
          (2, 5),(7, 5),( 9, 5),(14, 5),
          (2, 6),(7, 6),( 9, 6),(14, 6),
   (4, 7),(5, 7),(6, 7),(10, 7),(11, 7),(12, 7),


   (4, 9),(5, 9),(6, 9),(10, 9),(11, 9),(12, 9),
          (2,10),(7,10),( 9,10),(14,10),
          (2,11),(7,11),( 9,11),(14,11),
          (2,12),(7,12),( 9,12),(14,12),

   (4,14),(5,14),(6,14),(10,14),(11,14),(12,14)])
```

We now want to translate from **Haskell** into **Scala**, the remaining **Game of Life** functions, which are **impure functions**, and which are shown on the next slide.

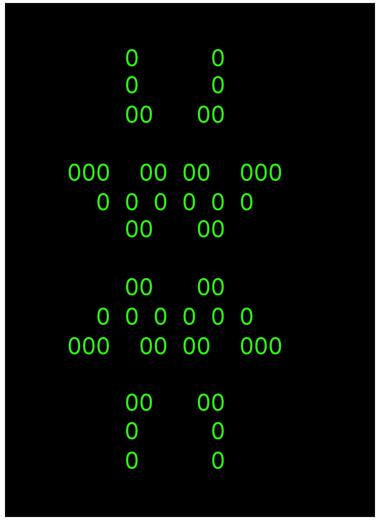
IMPURE FUNCTIONS

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'

cls :: IO ()
cls = putStr "\ESC[2J"

writeat :: Pos -> String -> IO ()
writeat p xs = do goto p
                  putStr xs

goto :: Pos -> IO ()
goto (x,y) =
  putStr ("\ESC[" ++ show y ++ ";"
                  ++ show x ++ "H")
```

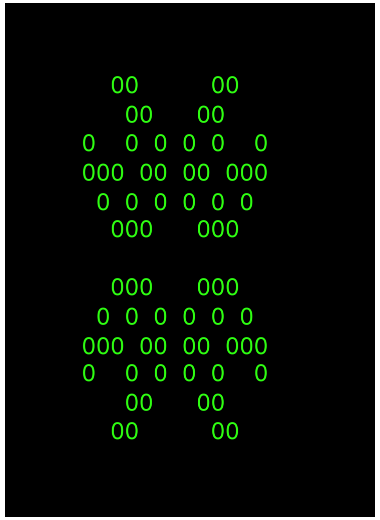
```haskell
life :: Board -> IO ()
life b = do cls
            showcells b
            wait 500000
            life (nextgen b)

showcells :: Board -> IO ()
showcells b = sequence_ [writeat p "O" | p <- b]

wait :: Int -> IO ()
wait n = sequence_ [return () | _ <- [1..n]]

main :: IO ()
main = life(pulsar)
```



```
  OOO   OOO

O    O O    O
O    O O    O
O    O O    O
  OOO   OOO

  OOO   OOO
O    O O    O
O    O O    O
O    O O    O

  OOO   OOO
```



```
    O     O
    O     O
    OO   OO

OOO  OO OO  OOO
  O O O O O O
    OO   OO

    OO   OO
  O O O O O O
OOO  OO OO  OOO

    OO   OO
    O     O
    O     O
```



```
  OO     OO
   OO   OO
O  O O O O  O
OOO OO OO OOO
 O O O O O O
  OOO   OOO

  OOO   OOO
 O O O O O O
OOO OO OO OOO
O  O O O O  O
   OO   OO
  OO     OO
```

While I have also included **putsStr** and **putStrLn**, they are of course **Haskell** predefined (derived) **primitives**.

The difference between the **pure functions** and the **impure functions** is that while the former don't have any **side effects**, the latter do, which is indicated by the fact that their signatures contain the **IO type**. See below for how **Graham Hutton** puts it.

Most of the definitions used to implement the **game of life** are **pure functions**, with **only a small number of top-level definitions involving input/output**. Moreover, **the definitions that do have such side-effects are clearly distinguishable from those that do not, through the presence of IO in their types**.

Graham Hutton

@haskellhutt

Functions with the **IO type** in their signature are called **IO actions**. The first two **Haskell** functions on the previous slide are predefined primitives **putStr** and **putStrLn**, which are **IO actions**. See below for how **Will Kurt** puts the fact that **IO actions** are not really functions, i.e. they are not **pure**.

IO actions work much like functions except they violate at least one of the three rules of functions that make functional programming so predictable and safe

- **All functions must take a value**.
- **All functions must return a value**.
- **Anytime the same argument is supplied, the same value must be returned (referential transparency)**.

getLine is an IO action because it violates our rule that functions must take an argument

putStrLn is an IO action because it violates our rule that functions must return values.

Will Kurt

@willkurt

Also see below how **Alejandro Mena** puts it when discussing **referential transparency** and the **randomRIO** function

Haskell's **solution is to mark those values for which purity does not hold with IO**.

Alejandro Serrano Mena

@trupill

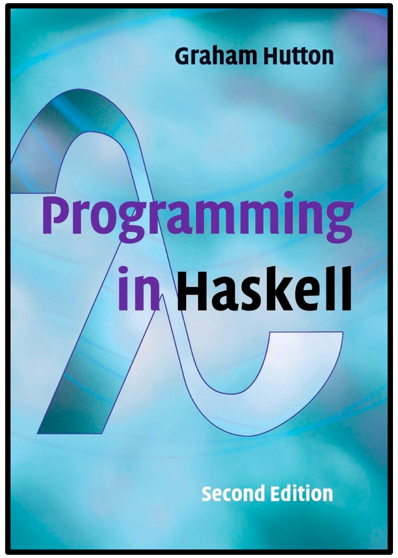
@philip_schwarz

But while there is this clear distinction between **pure functions** and **impure functions**, between functions with **side effects** and functions without **side effects**, between **pure functions** and **IO actions**, there is also the notion that **an IO action *can* be viewed as a pure function** that takes the **current state of the world** as its argument, and produces a **modified world** as its result, in which the **modified world** reflects any **side-effects** that were performed by the program during its execution.

See the next slide for a recap of how **Graham Hutton** puts it.

Graham Hutton

@haskellhutt



Graham Hutton

Programming in Haskell

Second Edition

In Haskell an interactive program is viewed as a pure function that takes the current state of the world as its argument, and produces a modified world as its result, in which the modified world reflects any side-effects that were performed by the program during its execution. Hence, given a suitable type `World` whose values represent states of the world, the notion of an interactive program can be represented by a function of type `World -> World` which we abbreviate as `IO` (short for input/output) using the following type declaration:

```
type IO = World -> World
```

In general, however, an interactive program may return a result value in addition to performing side-effects. For example, a program for reading a character from the keyboard may return the character that was read. For this reason, we generalise our type for interactive programs to also return a result value with the type of such values being a parameter of the `IO` type:

```
type IO a = World -> (a,World)
```

Expressions of type `IO a` are called actions. For example, `IO Char` is the type of actions that return a character, while `IO ()` is the type of actions that return the empty tuple `()` as a dummy result value. Actions of the latter type can be thought of as purely side-effecting actions that return no result value and are often useful in interactive programming.

In addition to returning a result value, interactive programs may also require argument values. However, there is no need to generalise the `IO` type further to take account of this, because this behaviour can already be achieved by exploiting currying. For example, an interactive program that takes a character and returns an integer would have type `Char -> IO Int`, which abbreviates the curried function type `Char -> World -> (Int,World)`.

At this point the reader may, quite reasonably, be concerned about the feasibility of passing around the entire state of the world when programming with actions! Of course, this isn't possible, and in reality the type `IO a` is provided as a primitive in Haskell, rather than being represented as a function type. However, the above explanation is useful for understanding how actions can be viewed as pure functions, and the implementation of actions in Haskell is consistent with this view. For the remainder of this chapter, we will consider `IO a` as a built-in type whose implementation details are hidden:

```
data IO a = ...
```

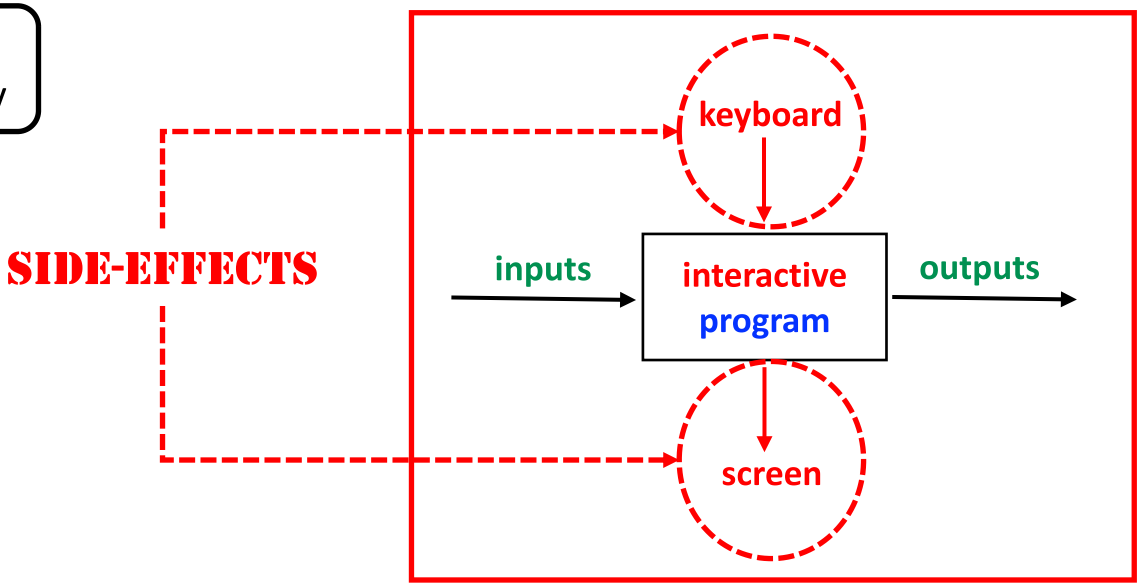
See the next slide for an illustrated recap of how viewing **IO actions** as **pure functions** solves the problem of modeling **interactive programs** as **pure functions**.

See the slide after that for a further illustration of how **IO actions** can be seen as **pure functions**.

visual summary

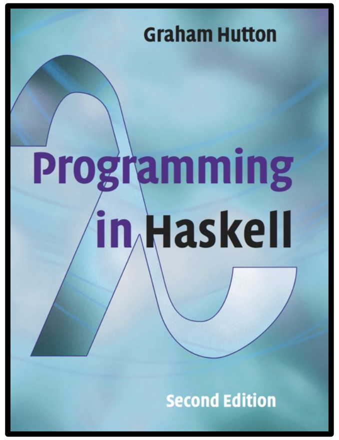
@philip_schwarz

Problem



How can such programs be modelled as **pure functions**?

Graham Hutton
@haskellhutt



Graham Hutton

Programming in Haskell

Second Edition

```
type IO a = World -> (a,World)
```

IO : short for **input/output**

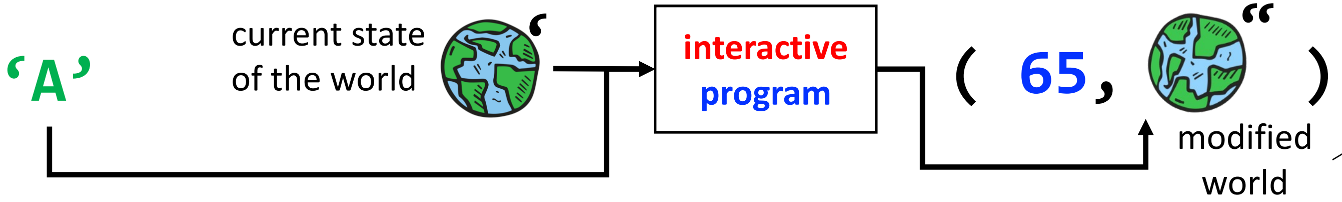
```
Char ──────────> IO Int
```

IO action returning an `Int`

```
Char ───> World ───> (Int,World)
```

Solution

Haskell



reflects any **side-effects** that were performed by the program during its execution.

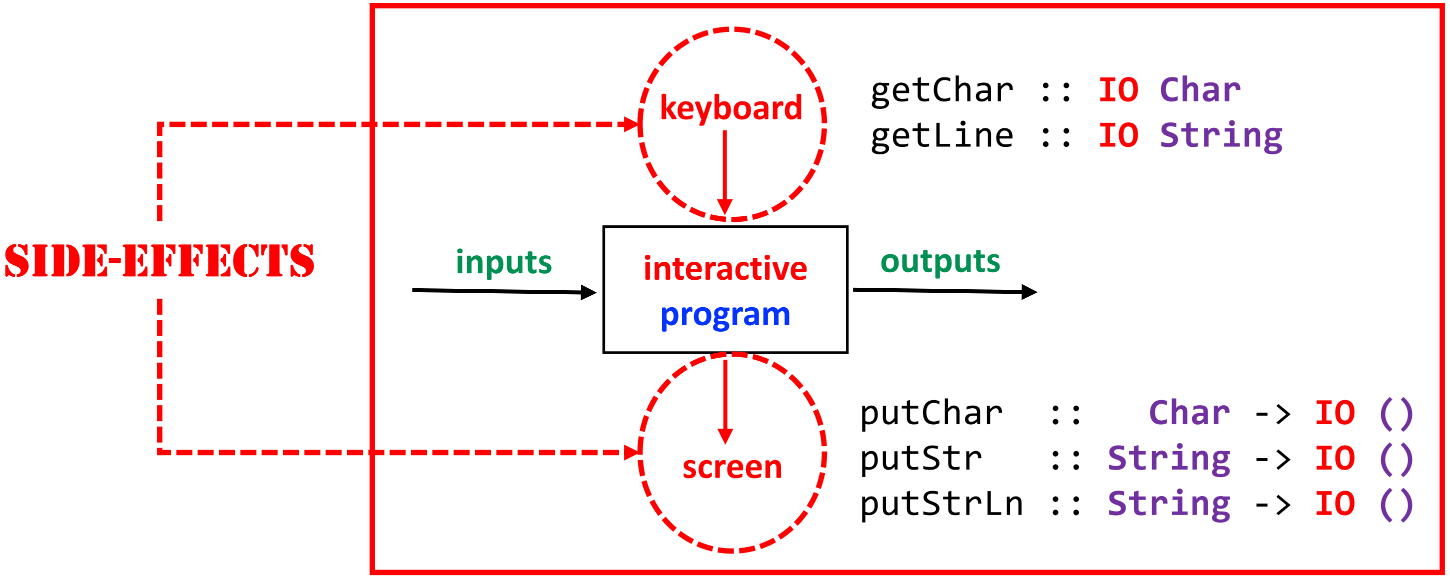
Instead of viewing **IO actions** as **impure functions** that perform the **side effects** of an **interactive program**…

Basic primitive IO actions

```
getChar   :: IO Char
putChar   :: Char -> IO ()
return    :: a -> IO a
```

Derived primitive IO actions

```
getLine   :: IO String
putStr    :: String -> IO ()
putStrLn :: String -> IO ()
```

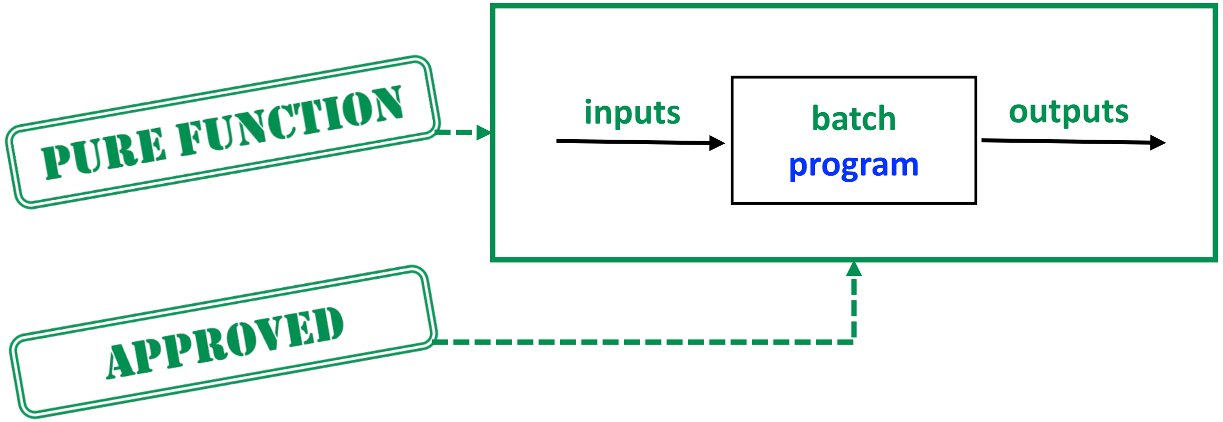


```
getChar :: IO Char
getLine :: IO String
```

```
putChar   ::   Char -> IO ()
putStr    :: String -> IO ()
putStrLn :: String -> IO ()
```

```
                            keyboard
getChar   ::              World -> (Char, World)

                            console
putChar   ::   Char -> World -> ((), World)

return    ::        a -> World -> (a, World)

                            keyboard
getLine   ::              World -> (String, World)

                            console
putStr    :: String -> World -> ((), World)

                            console
putStrLn :: String -> World -> ((), World)
```

…view them as **pure functions** of a **batch program** that take the **current world** and return a **modified world** that reflects any **side effects** performed.



The first two **Haskell** functions that we have to translate into **Scala** are **derived primitive IO actions putStr** and **putStrLn**.

Right after **Graham Hutton** introduced **Haskell's primitive IO actions**, he showed us a simple **strLen** program that made use of the **derived** ones (see below).

```
Basic primitive IO actions

getChar   :: IO Char
putChar   :: Char -> IO ()
return    :: a -> IO a

Derived primitive IO actions

getLine   :: IO String
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
```

Graham Hutton
@haskellhutt

For example, using these **primitives** we can now define an **action** that prompts for a string to be entered from the **keyboard**, and displays its length:

```
strLen :: IO ()
strLen = do putStr "Enter a string: "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

For example

```
> strLen
Enter a string: Haskell
The string has 7 characters
>
```

The next thing we are going to do is see if we can translate the **Haskell derived primitive IO actions** into **Scala** and then use them to write the **Scala** equivalent of the program on the left.


@philip_schwarz

Haskell **primitive IO actions** have the **IO type** in their signature.

```
getLine  :: IO String
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

While in **Scala** there are **primitive functions** for reading/writing from/to the **console**, their signature does not involve any **IO type** and there is no such **predefined** type in **Scala**.

While we can view the **Haskell IO actions** as **pure functions**, which take the **current world** and return a **result** together with a **modified world**, the same cannot be said of the corresponding **Scala** functions, which have **side effects** (they violate one or more of the rules for **pure functions**):

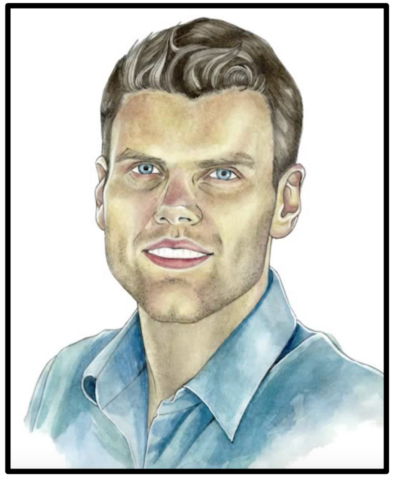
```
getLine  :: World -> (String, World)
putStr   :: String -> World -> ((), World)
putStrLn :: String -> World -> ((), World)
```

```
def readLine()       : String
def print(x: Any)    : Unit
def println(x: Any)  : Unit
```

It is, however, possible to define an **IO type** in **Scala** and we are now going to turn to **Functional Programming in Scala** (**FPiS**) to see how it is done and how such an **IO type** can be used to write **pure Scala functions** that mirror **Haskell**'s **derived primitive IO actions**.

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

13 External effects and I/O

In this chapter, we'll take what we've learned so far about **monads** and **algebraic data types** and extend it to handle **external effects** like **reading from databases** and **writing to files**.

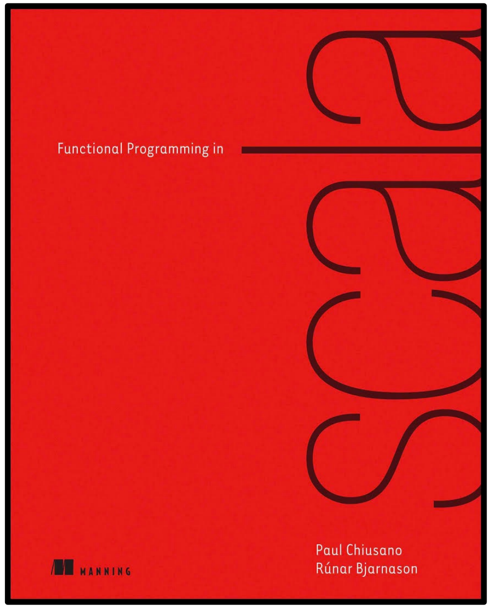
We'll develop a monad for I/O, aptly called IO, that will allow us to handle such external effects in a **purely functional** way.

We'll make an **important distinction** in this chapter between **effects** and **side effects**.

The IO monad provides a straightforward way of embedding imperative programming with I/O effects in a pure program while preserving referential transparency. **It clearly separates effectful code**—**code that needs to have some effect on the outside world**—**from the rest of our program**.

This will also illustrate **a key technique for dealing with external effects**—**using pure functions to compute a description of an effectful computation, which is then executed by a separate interpreter that actually performs those effects**.

Essentially we're crafting **an embedded domain-specific language (EDSL) for imperative programming**. This is a powerful technique that we'll use throughout the rest of part 4. Our goal is to equip you with the skills needed to craft your own EDSLs for describing **effectful programs**.



Functional Programming in Scala

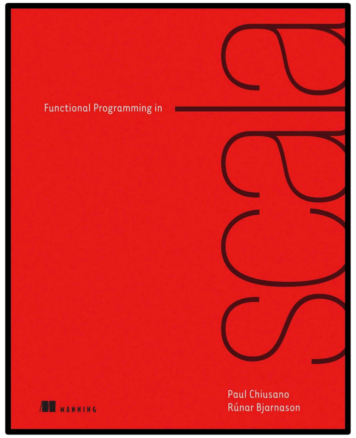
13.1 Factoring effects

We'll work our way up to the `IO` monad by first considering a simple example of a program with side effects.

```scala
case class Player(name: String, score: Int)

def contest(p1: Player, p2: Player): Unit =
  if (p1.score > p2.score)
    println(s"${p1.name} is the winner!")
  else if (p2.score > p1.score)
    println(s"${p2.name} is the winner!")
  else
    println("It's a draw.")
```

The `contest` function couples the I/O code for displaying the result to the pure logic for computing the winner. We can factor the logic into its own pure function, `winner`:

```scala
def contest(p1: Player, p2: Player): Unit = winner(p1, p2) match {
  case Some(Player(name, _)) => println(s"$name is the winner!")
  case None => println("It's a draw.")
}

def winner(p1: Player, p2: Player): Option[Player] =
  if (p1.score > p2.score) Some(p1)
  else if (p1.score < p2.score) Some(p2)
  else None
```

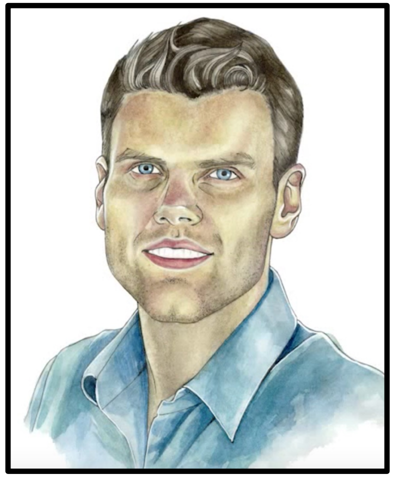

Functional Programming in
Scala
Paul Chiusano
Rúnar Bjarnason
MANNING

Functional Programming in Scala

It is always possible to factor an impure procedure into a pure "core" function and two procedures with side effects: one that supplies the pure function's input and one that does something with the pure function's output. In listing 13.1, we factored the pure function `winner` out of `contest`. Conceptually, `contest` had two responsibilities—it was computing the result of the contest, and it was displaying the result that was computed. With the refactored code, `winner` has a single responsibility: to compute the winner. The `contest` method retains the responsibility of printing the result of `winner` to the console.

Paul Chiusano

@pchiusano

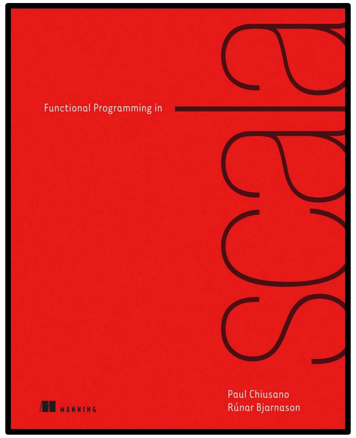


Runar Bjarnason

@runarorama

We can refactor this even further. The `contest` function still has **two responsibilities**: it's **computing** which message to display and then **printing** that message to the **console**. **We could factor out a pure function here as well**, which might be beneficial if we later decide to display the result in some sort of UI or write it to a file instead. Let's perform this refactoring now:

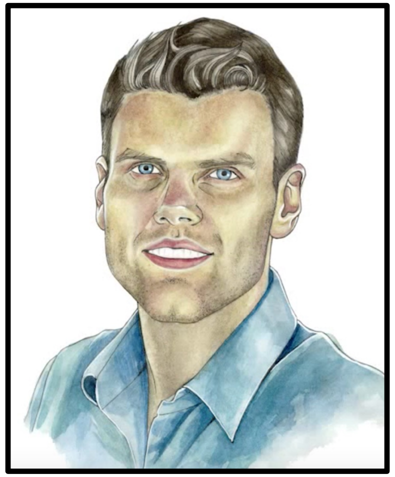
```
def contest(p1: Player, p2: Player): Unit =
  println(winnerMsg(winner(p1, p2)))

def winner(p1: Player, p2: Player): Option[Player] =
  if (p1.score > p2.score) Some(p1)
  else if (p1.score < p2.score) Some(p2)
  else None

def winnerMsg(p: Option[Player]): String = p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."
```



Functional Programming in Scala

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

Note how the side effect, `println`, **is now only in the outermost layer of the program, and what's inside the call to** `println` **is a pure expression**. This might seem like a simplistic example, but **the same principle applies in larger, more complex programs**, and we hope you can see how this sort of refactoring is quite natural. We aren't changing what our program does, just the internal details of how it's factored into smaller functions.

The insight here is that **inside every function with side effects is a pure function waiting to get out**. We can formalize this insight a bit. Given an **impure function** `f` of type `A => B`, we can split `f` into two functions:

- A **pure function** of type `A => D`, where `D` is some **description** of the result of `f`.
- An **impure function** of type `D => B`, which can be thought of as **an interpreter of these descriptions**.

We'll extend this to handle **"input" effects** shortly. For now, **let's consider applying this strategy repeatedly to a program**. **Each time we apply it, we make more functions pure and push side effects to the outer layers. We could call these impure functions the "imperative shell" around the pure "core" of the program**. Eventually, we reach functions that seem to necessitate **side effects** like the built-in `println`, which has type `String => Unit`. What do we do then?

About the following section of the previous slide:

The insight here is that **inside every function with side effects is a pure function waiting to get out**

Given an **impure function** `f` of type `A => B`, we can split `f` into two functions:

- A **pure function** of type `A => D`, where `D` is some **description of the result** of `f`.
- An **impure function** of type `D => B` which can be thought of as **an interpreter of these descriptions**.

When it come to relating the above to the **`contest`** function that we have just seen, I found it straightforward to relate function A => B to the original **`contest`** function, but I did not find it that straightforward to relate functions A => D and D => B to the **`winner`** and **`winnerMsg`** functions.

So on the next slide I show the code before and after the refactoring and in the slide after that I define type aliases A, B and D and show the code again but making use of the aliases.

On both slides, I have replaced methods with functions in order to help us identify the three functions
A => B, A => D and D => B.

I also called the A => D function '**pure**' and the D => B function '**impure**'.

Here on the left is the original **contest** function, and on the right, the refactored version

```scala
def contest: ((Player, Player)) => Unit = {
  case (p1: Player, p2: Player) =>
    if (p1.score > p2.score)
      println(s"${p1.name} is the winner!")
    else if (p2.score > p1.score)
      println(s"${p2.name} is the winner!")
    else
      println("It's a draw.")
}
```

```scala
val contest: ((Player, Player)) => Unit = impure compose pure

val impure : String => Unit = println(_)
```

IMPURE FUNCTIONS

```scala
val pure: ((Player, Player)) => String = winnerMsg compose winner

val winnerMsg: Option[Player] => String = p => p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."

val winner: ((Player, Player)) => Option[Player] = {
  case (p1: Player, p2: Player) =>
    if (p1.score > p2.score) Some(p1)
    else if (p1.score < p2.score) Some(p2)
    else None
}
```

PURE FUNCTIONS

Same as on the previous slide, but using type aliases **A**, **B** and **D**.

@philip_schwarz

```scala
def contest: A => B = {
  case (p1: Player, p2: Player) =>
    if (p1.score > p2.score)
      println(s"${p1.name} is the winner!")
    else if (p2.score > p1.score)
      println(s"${p2.name} is the winner!")
    else
      println("It's a draw.")
}
```

```scala
type A = (Player, Player)
type B = Unit
type D = String
```

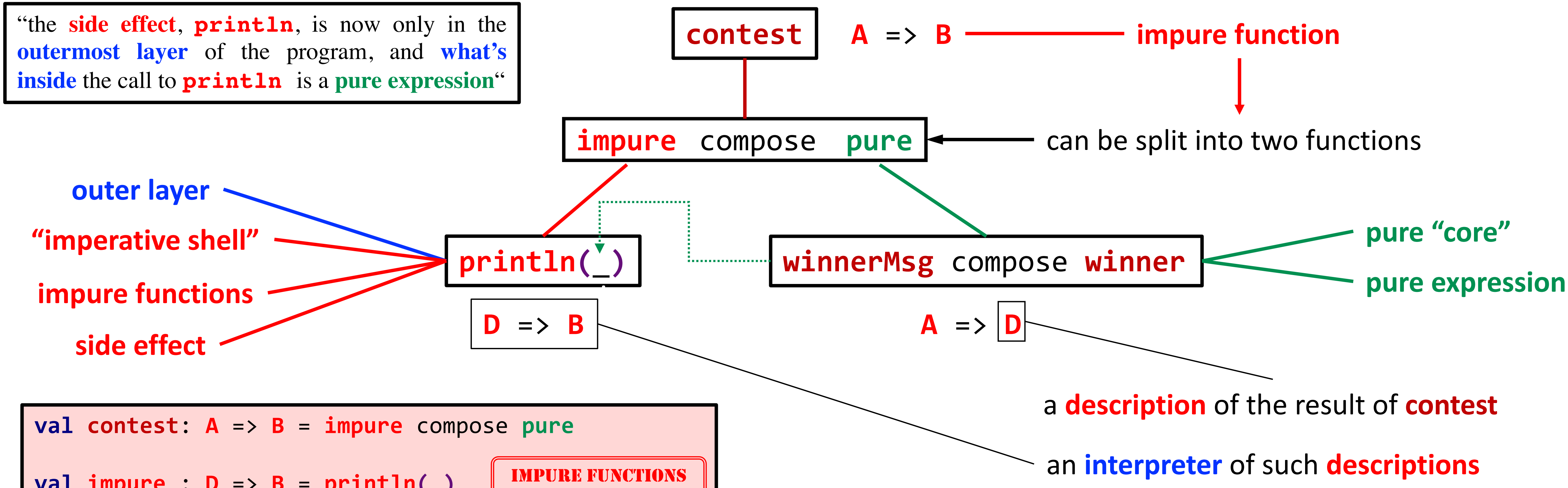
```scala
val contest: A => B = impure compose pure

val impure : D => B = println(_)
```

IMPURE FUNCTIONS

```scala
val pure: A => D = winnerMsg compose winner

val winnerMsg: Option[Player] => D = p => p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."

val winner: A => Option[Player] = {
  case (p1: Player, p2: Player) =>
    if (p1.score > p2.score) Some(p1)
    else if (p1.score < p2.score) Some(p2)
    else None
}
```

PURE FUNCTIONS

"the **side effect**, **`println`**, is now only in the **outermost layer** of the program, and **what's inside** the call to **`println`** is a **pure expression**"



`contest`

`A => B` impure function

can be split into two functions

`impure compose pure`

outer layer

"imperative shell"

impure functions

side effect

`println(_)`

`D => B`

`winnerMsg compose winner`

`A => D`

pure "core"

pure expression

a **description** of the result of **contest**

an **interpreter** of such **descriptions**

```
val contest: A => B = impure compose pure

val impure : D => B = println(_)
```

IMPURE FUNCTIONS

```
val pure: A => D = winnerMsg compose winner

val winnerMsg: Option[Player] => D = p => p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."

val winner: A => Option[Player] = {
  case (p1: Player, p2: Player) =>
    if (p1.score > p2.score) Some(p1)
    else if (p1.score < p2.score) Some(p2)
    else None
}
```

PURE FUNCTIONS

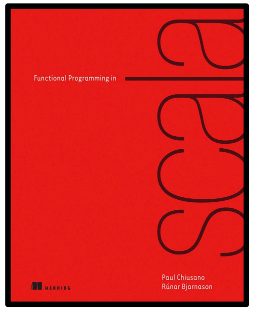
`contest``: A => B` - an **impure function** – it can be split into two functions:
- **`pure`**`: A => D` – a **pure** function producing a **description** of the result
- **`impure`**`: D => B` – an **impure** function, an **interpreter** of such **descriptions**

`type A = (Player, Player)`
`type B = Unit`
`type D = String` - Some **description** of the result of **`contest`**

13.2 A simple IO type

It turns out that even procedures like `println` are doing more than one thing. And they can be factored in much the same way, by introducing a new data type that we'll call `IO`:

```
trait IO { def run: Unit }

def PrintLine(msg: String): IO =
  new IO { def run = println(msg) }

def contest(p1: Player, p2: Player): IO =
  PrintLine(winnerMsg(winner(p1, p2)))
```



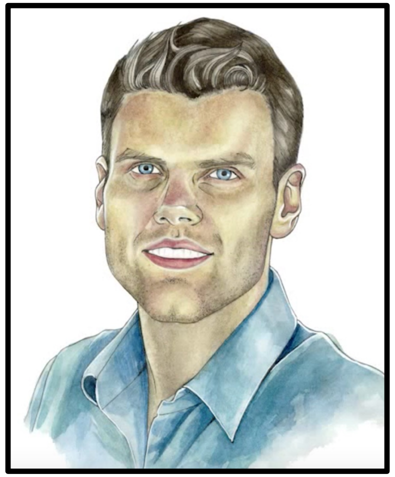
Functional Programming in Scala

Our `contest` function is now pure— it returns an `IO` value, which simply describes an action that needs to take place, but doesn't actually execute it. We say that `contest` has (or produces) an effect or is effectful, but it's only the interpreter of `IO` (its `run` method) that actually has a side effect.

Now `contest` only has one responsibility, which is to compose the parts of the program together: `winner` to compute who the winner is, `winnerMsg` to compute what the resulting message should be, and `PrintLine` to indicate that the message should be printed to the console. But the responsibility of interpreting the effect and actually manipulating the console is held by the `run` method on `IO`.

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

Paul Chiusano
@pchiusano

We’ll make an important distinction in this chapter between effects and side effects.

a key technique for dealing with external effects—using pure functions to compute a description of an effectful computation, which is then executed by a separate interpreter that actually performs those effects.

Our `contest` function is now pure

```scala
def contest(p1: Player, p2: Player): IO =
  PrintLine(winnerMsg(winner(p1, p2)))
```

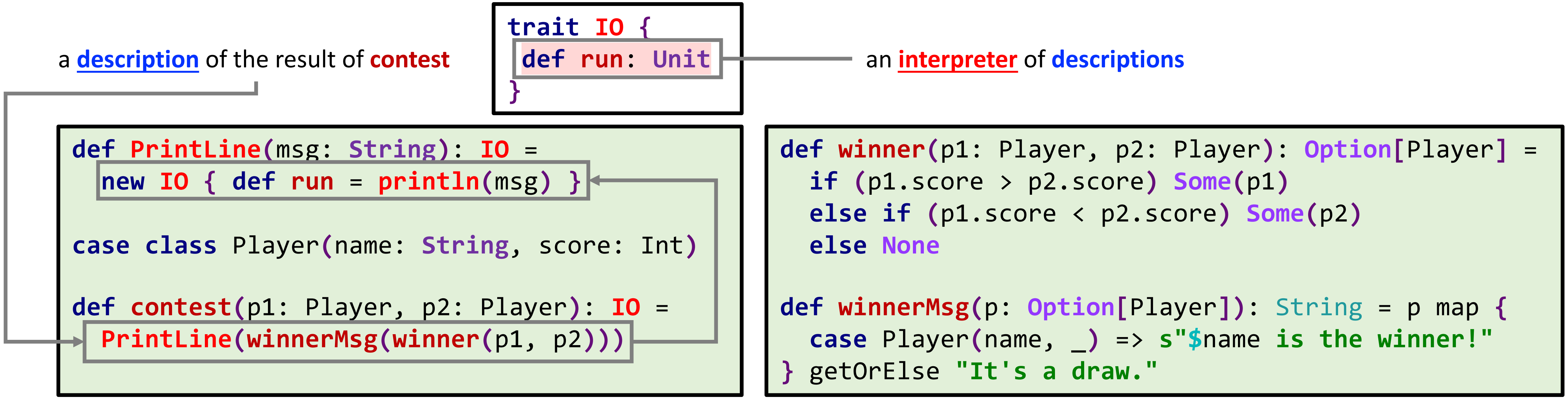
it returns an `IO` value, which simply describes an action that needs to take place, but doesn’t actually execute it

Runar Bjarnason
@runarorama

`contest` has (or produces) an effect or is effectful, but it’s only the interpreter of `IO` (its `run` method) that actually has a side effect

```scala
trait IO { def run: Unit }
```

the responsibility of interpreting the effect and actually manipulating the console is held by the `run` method on `IO`.



a description of the result of contest

```scala
trait IO {
  def run: Unit
}
```

an interpreter of descriptions

```scala
def PrintLine(msg: String): IO =
  new IO { def run = println(msg) }

case class Player(name: String, score: Int)

def contest(p1: Player, p2: Player): IO =
  PrintLine(winnerMsg(winner(p1, p2)))
```

```scala
def winner(p1: Player, p2: Player): Option[Player] =
  if (p1.score > p2.score) Some(p1)
  else if (p1.score < p2.score) Some(p2)
  else None

def winnerMsg(p: Option[Player]): String = p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."
```

Let's try out the program. We first execute the **contest** function, which returns an **IO action** **describing** an **effectful computation**. We then invoke the **description**'s **run** method, which **interprets** the **description** by executing the **effectful computation** that it describes, and which results in the **side effect** of **printing** a message to the **console**.

```
scala> val resultDescription: IO = contest(Player("John",score=10), Player("Jane",score=15))
resultDescription: IO = GameOfLife$$anon$1@544f606e

scala> resultDescription.run
Jane is the winner!

scala>
```

side effect

produced by

simply describes an **action** that needs to take place, but doesn't actually **execute** it

a **description** of the result of **contest**

```
trait IO {
  def run: Unit
}
```

an **interpreter** of **descriptions**

it's only the **interpreter** of **IO** (its **run** method) that actually has a **side effect**.

```
def PrintLine(msg: String): IO =
  new IO { def run = println(msg) }

case class Player(name: String, score: Int)

def contest(p1: Player, p2: Player): IO =
  PrintLine(winnerMsg(winner(p1, p2)))
```

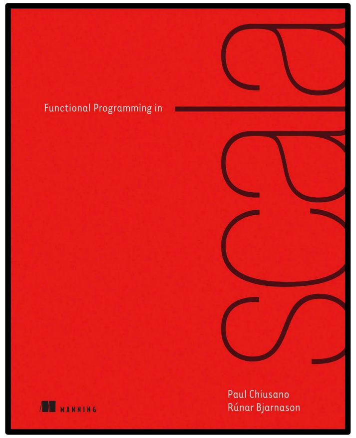
```
def winner(p1: Player, p2: Player): Option[Player] =
  if (p1.score > p2.score) Some(p1)
  else if (p1.score < p2.score) Some(p2)
  else None

def winnerMsg(p: Option[Player]): String = p map {
  case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."
```

Other than technically satisfying the requirements of referential transparency, has the `IO` type actually bought us anything? That's a personal value judgement. **As with any other data type, we can assess the merits of `IO` by considering what sort of algebra it provides—is it something interesting, from which we can define a large number of useful operations and programs, with nice laws that give us the ability to reason about what these larger programs will do?** Not really. Let's look at the operations we can define:

```
trait IO { self =>

  def run: Unit

  def ++(io: IO): IO = new IO {
    def run = { self.run; io.run }
  }

}
object IO {

  def empty: IO = new IO { def run = () }

}
```

The `self` argument lets us refer to this object as `self` instead of `this`.

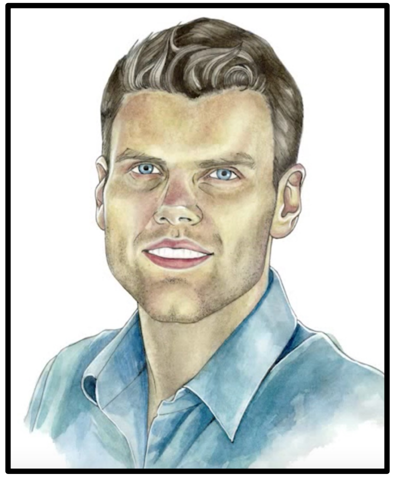
self refers to the outer IO.



Functional Programming in Scala

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

The only thing we can perhaps say about `IO` as it stands right now is that it forms a `Monoid` (`empty` is the identity, and `++` is the associative operation). **So if we have, for example, a `List[IO]`, we can reduce that to a single `IO`**, and the associativity of `++` means that we can do this either by folding left or folding right. On its own, this isn't very interesting. **All it seems to have given us is the ability to delay when a side effect actually happens**.

Now we'll let you in on a secret: you, **as the programmer, get to invent whatever API you wish to represent your computations, including those that interact with the universe external to your program**. This process of crafting pleasing, useful, and composable descriptions of what you want your programs to do is at its core language design.

You're crafting a little language, and an associated interpreter, that will allow you to express various programs. If you don't like something about this language you've created, change it! You should approach this like any other design task.

In the next slide we have a go at running multiple **IO actions**, first by executing each one individually, and then by **folding** them into a single **composite IO action** and executing that.

```
// create IO actions describing the effectful
// computations for three contests.
val gameResultDescriptions: List[IO] = List(
  contest(Player("John", 10), Player(   "Jane", 15)),
  contest(Player("Jane",  5), Player("Charlie", 10)),
  contest(Player("John", 25), Player("Charlie", 25))
)
```

Here are the **IO actions** for three contests.

Before the introduction of new **IO** operations **++** and **empty**, there was no way to **compose IO actions**, so each **action** had to be executed individually. See ①

```
trait IO { self =>
  def run: Unit
  def ++(io: IO): IO = new IO {
    def run = { self.run; io.run }
  }
}
object IO {
  def empty: IO = new IO { def run = () }
}
```

①

```
// run each IO action individually, in sequence
gameResultDescriptions.foreach(_.run)
```

②

```
// use the List.fold function of the standard library
// to fold all the IO actions into a single composite
// IO action.
val gameResultsDescription: IO =
  gameResultDescriptions.fold(IO.empty)(_ ++ _)

// run the composite IO action
gameResultsDescription.run
```

③

```
// define an implicit Monoid instance for IO using an
// FP library like cats, or in this case using a
// hand-rolled Monoid typeclass (see bottom right).
implicit val ioMonoid: Monoid[IO] = new Monoid[IO] {
  def op(x: IO, y: IO): IO = x ++ y
  val zero: IO = IO.empty
}

// fold the actions using the List concatenation function of an
// FP library, or in this case a hand-rolled one (see right).
val gameResultsDescription: IO =
  concatenate(gameResultDescriptions)(ioMonoid)

// run the composite IO action
gameResultsDescription.run
```

```
Jane is the winner!
Charlie is the winner!
It's a draw.
```

@philip_schwarz

Now that **++** and **IO.empty** are available, we can compose multiple **IO actions** into a single one using **++**, and since (**IO**, **++**, **IO.empty**) forms a **monoid**, we can also compose the **actions** by **folding** them with the **monoid**.
In ② we **fold** actions using the standard library's **List.fold** function.
In ③ we **fold** them using an **implicit** instance of the **IO monoid** and a hand-rolled **concatenate** function that uses the **implicit monoid**.

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}

def concatenate[A](as: List[A])(implicit m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.op)
```

13.2.1 Handling input effects

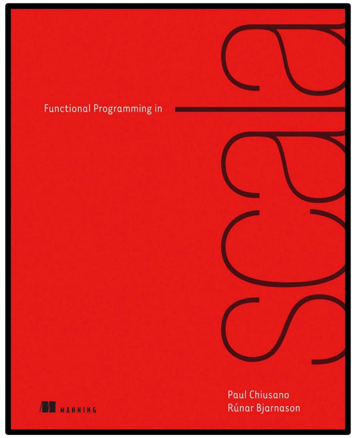
As you've seen before, sometimes when building up a little language you'll encounter a program that it can't express. So far our `IO` type can represent only "output" effects. There's no way to express `IO` computations that must, at various points, wait for input from some external source. Suppose we wanted to write a program that prompts the user for a temperature in degrees Fahrenheit, and then converts this value to Celsius and echoes it to the user. A typical imperative program might look something like this[1].

```scala
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0

def converter: Unit = {
  println("Enter a temperature in degrees Fahrenheit: ")
  val d = readLine.toDouble
  println(fahrenheitToCelsius(d))
}
```

Unfortunately, we run into problems if we want to make `converter` into a pure function that returns an `IO`:

```scala
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0

def converter: IO = {
  val prompt: IO = PrintLine("Enter a temperature in degrees Fahrenheit: ")
  // now what ???
}
```

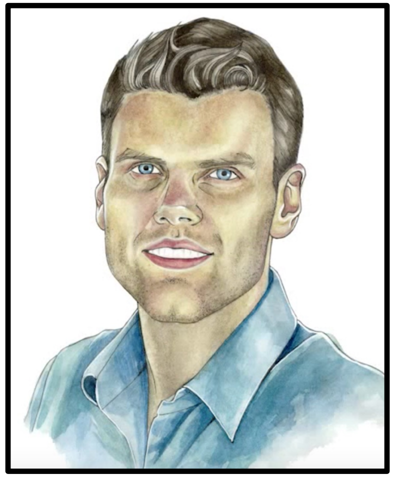
```scala
trait IO {  def run: Unit }

def PrintLine(msg: String): IO =
  new IO { def run = println(msg) }
```



Functional Programming in Scala

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

In Scala, `readLine` is a `def` with the side effect of capturing a line of input from the console. It returns a `String`. We could wrap a call to `readLine` in `IO`, but we have nowhere to put the result! We don't yet have a way of representing this sort of effect. The problem is that our current `IO` type can't express computations that yield a value of some meaningful type—our interpreter of `IO` just produces `Unit` as its output.

```scala
trait IO {
 def run: Unit
}
```

---

[1] We're not doing any sort of error handling here. This is just meant to be an illustrative example.

Should we give up on our `IO` type and resort to using side effects? Of course not! We extend our `IO` type to allow input, by adding a type parameter:

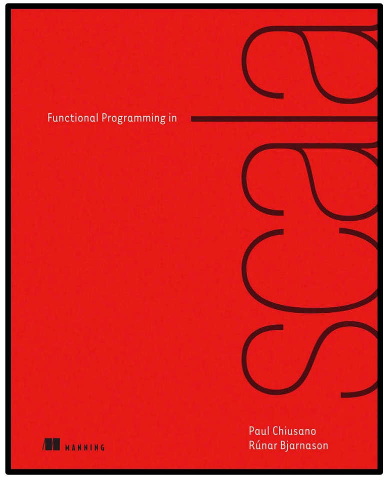
```
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}
```

An `IO` computation can now return a meaningful value. Note that we've added **`map`** and **`flatMap`** functions so `IO` can be used in for-comprehensions. And `IO` now forms a **`Monad`**:

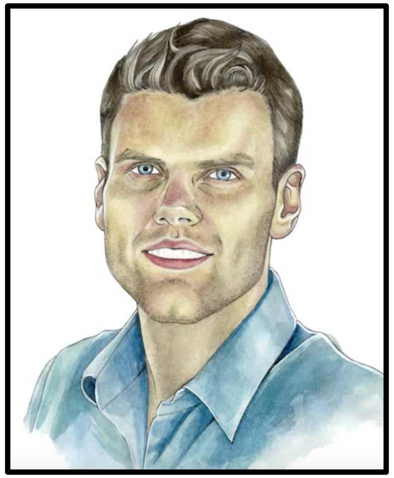
```
object IO extends Monad[IO] {
  def unit[A](a: => A): IO[A] =
    new IO[A] { def run = a }
  def flatMap[A,B](fa: IO[A])(f: A => IO[B]) =
    fa flatMap f
  def apply[A](a: => A): IO[A] =
    unit(a)
}
```

We can now write our `converter` example:

```
def ReadLine: IO[String] = IO { readLine }
def PrintLine(msg: String): IO[Unit] = IO { println(msg) }
def converter: IO[Unit] = for {
  _ <- PrintLine("Enter a temperature in degrees Fahrenheit: ")
  d <- ReadLine.map(_.toDouble)
  _ <- PrintLine(fahrenheitToCelsius(d).toString)
} yield ()
```



Functional Programming in Scala



Runar Bjarnason

@runarorama

Paul Chiusano

@pchiusano

Our `converter` definition no longer has side effects—it's a referentially transparent description of a computation with effects, and `converter.run` is the interpreter that will actually execute those effects. And because `IO` forms a **`Monad`**, we can use all the **monadic combinators** we wrote previously.

The previous slide included the following statements:

- **`map`** and **`flatMap`** functions were added to **`IO`** so that it can be used in **for-comprehensions**
- **`IO`** now forms a **`Monad`**

But

- what does it mean for **IO** to be a **monad**?
- what is a **for comprehension**?
- why does **IO** need **map** and **flatMap** in order to be used in a **for comprehension**?
- how do the particular **Scala idioms** used to implement the **IO monad** compare with **alternative idioms**?

The following ten slides have a quick go at answering these questions. If you are already familiar with **monads** in **Scala** you can safely skip them.

One way to define a **monad** is to say that it is an implementation of the **Monad interface** on the right, such that its **unit** and **flatMap** functions obey the **monad laws** (the three monadic laws are outside the scope of this slide deck. See the following for an introduction: https://www.slideshare.net/pjschwarz/monad-laws-must-be-checked-107011209)

E.g. here we take **Foo**, a class that simply wraps a value of some type A, and we instantiate the **Monad interface** for **Foo** by supplying implementations of **unit** and **flatMap**.

We implement **unit** and **flatMap** in a trivial way, which results in the simplest possible **monad**, one that does nothing (the **identity monad**). We do this so that in this and the next few slides we are not distracted by the details of any particular **monad** and can concentrate on things that apply to all **monads**.

We then show how invocations of the **flatMap** function of the **Foo monad** can be **chained**, allowing us to get hold of the wrapped values and use them to compute a result which then gets wrapped.

In the next slide we turn to a different way of defining a **monad**.

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

```scala
case class Foo[A](value:A)
```

```scala
val fooMonad: Monad[Foo] = new Monad[Foo] {
  def unit[A](a: => A): Foo[A] =
    Foo(a)
  def flatMap[A, B](ma: Foo[A])(f: A => Foo[B]): Foo[B] =
    f(ma.value)
```

```scala
val fooTwo = Foo(2)
val fooThree = Foo(3)
val fooFour = Foo(4)
val fooNine = Foo(9)
```

```scala
val fooResult =
  fooMonad.flatMap(fooTwo) { x =>
    fooMonad.flatMap(fooThree) { y =>
      fooMonad.flatMap(fooFour) { z =>
        fooMonad.unit(x + y + z)
      }
    }
  }

assert(fooResult.value == 9)
```

In the **Scala** language itself there is no built-in, predefined **Monad interface** like the one we saw on the previous slide. Instead, if we give the **Foo** class a **map** function and a **flatMap** function with the signatures shown on the right, then the **Scala** compiler makes our life easier in that instead of us having to implement the computation that we saw on the previous slide by **chaining flatMap** and **map** as shown bottom right, we can use the **syntactic sugar** of a **for comprehension**, as shown bottom left. i.e. the compiler translates (desugars) the **for comprehension** into a **chain** of **flatMaps** ending with a **map**.

@philip_schwarz

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

```scala
val fooTwo = Foo(2)
val fooThree = Foo(3)
val fooFour = Foo(4)
val fooNine = Foo(9)
```

```scala
val fooSweetenedResult =
  for {
    x <- fooTwo
    y <- fooThree
    z <- fooFour
  } yield x + y + z

assert(fooSweetenedResult.value == 9)
```

desugars to

```scala
val fooPlainResult =
  fooTwo flatMap { x =>
    fooThree flatMap { y =>
      fooFour map { z =>
        x + y + z
      }
    }
  }

assert(fooPlainResult.value == 9)
```

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

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

```
val fooTwo = Foo(2)
val fooThree = Foo(3)
val fooFour = Foo(4)
val fooNine = Foo(9)
```

The **first** and **second approach** to implementing a **monad** can happily coexist, so on this slide we see the code for **both approaches** together.

```
val fooMonad: Monad[Foo] = new Monad[Foo] {
  def unit[A](a: => A): Foo[A] =
    Foo(a)
  def flatMap[A, B](ma: Foo[A])(f: A => Foo[B]): Foo[B] =
    f(ma.value)
```

```
val fooResult =
  fooMonad.flatMap(fooTwo) { x =>
    fooMonad.flatMap(fooThree) { y =>
      fooMonad.flatMap(fooFour) { z =>
        fooMonad.unit(x + y + z)
      }
    }
  }

assert(fooResult.value == 9)
```

```
val fooSweetenedResult =
  for {
    x <- fooTwo
    y <- fooThree
    z <- fooFour
  } yield x + y + z

assert(fooSweetenedResult.value == 9)
```

desugars to

```
val fooPlainResult =
  fooTwo flatMap { x =>
    fooThree flatMap { y =>
      fooFour map { z =>
        x + y + z
      }
    }
  }

assert(fooPlainResult.value == 9)
```

```scala
sealed trait Foo[A] { self =>
  def value: A
  def map[B](f: A => B): Foo[B] =
    new Foo[B]{ def value = f(self.value) }
  def flatMap[B](f: A => Foo[B]): Foo[B] =
    f(self.value)
}
```

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

```scala
object Foo extends Monad[Foo] {
  def unit[A](a: => A): Foo[A] =
    new Foo[A]{ def value = a }
  def flatMap[A, B](ma: Foo[A])(f: A => Foo[B]): Foo[B] =
    ma flatMap f
  def apply[A](a: => A): Foo[A] =
    unit(a)
}
```

In **FPiS**, the approach taken to coding the **IO monad** is a **third approach** that is shown on this slide and which ultimately is equivalent to the sum of the previous **two approaches**, but uses different **Scala idioms**.

Instead of a **Foo** class, there is a **Foo** trait.

Instead of instantiating the **Monad interface** by **newing** one up, we define an **object** (a singleton) that implements the **interface**.

The **Foo** singleton **object** defines an **apply function** that allows us to create an instance of the **Foo** class by doing **Foo**(x) rather than **Foo.unit**(x).

Computing the sum of the three wrapped values by **chaining flatMaps** is almost identical to how it was done in the **first approach**.

```scala
val fooTwo = Foo(2)
val fooThree = Foo(3)
val fooFour = Foo(4)
val fooNine = Foo(9)
```

```scala
val fooResult =
  Foo.flatMap(fooTwo) { x =>
    Foo.flatMap(fooThree) { y =>
      Foo.flatMap(fooFour) { z =>
        Foo(x + y + z)
      }
    }
  }

assert(fooResult.value == 9)
```

```scala
val fooSweetenedResult =
  for {
    x <- fooTwo
    y <- fooThree
    z <- fooFour
  } yield x + y + z

assert(fooSweetenedResult.value == 9)
```

desugars to

```scala
val fooPlainResult =
  fooTwo flatMap { x =>
    fooThree flatMap { y =>
      fooFour map { z =>
        x + y + z
      }
    }
  }

assert(fooPlainResult.value == 9)
```

@philip_schwarz

On this slide we compare the **joint first two approaches**, with the **third approach** and show only the code in which the **third approach** differs from the **joint first two approaches** in that it uses different idioms, but ultimately results in code with the same capabilities.

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

different ways of defining **Foo**

```
sealed trait Foo[A] { self =>
  def value: A
  def map[B](f: A => B): Foo[B] =
    new Foo[B]{ def value = f(self.value) }
  def flatMap[B](f: A => Foo[B]): Foo[B] =
    f(self.value)
}
```

```
val fooMonad: Monad[Foo] = new Monad[Foo] {
  def unit[A](a: => A): Foo[A] =
    Foo(a)
  def flatMap[A, B](ma: Foo[A])(f: A => Foo[B]): Foo[B] =
    f(ma.value)
```

different ways of instantiating the **Monad interface**

```
object Foo extends Monad[Foo] {
  def unit[A](a: => A): Foo[A] =
    new Foo[A]{ def value = a }
  def flatMap[A, B](ma: Foo[A])(f: A => Foo[B]): Foo[B] =
    ma flatMap f
  def apply[A](a: => A): Foo[A] =
    unit(a)
}
```

```
val fooResult =
  fooMonad.flatMap(fooTwo) { x =>
    fooMonad.flatMap(fooThree) { y =>
      fooMonad.flatMap(fooFour) { z =>
        fooMonad.unit(x + y + z)
      }
    }
  }

assert(fooResult.value == 9)
```

different ways of referring to the **Monad** instance when **chaining flatMaps**

```
val fooResult =
  Foo.flatMap(fooTwo) { x =>
    Foo.flatMap(fooThree) { y =>
      Foo.flatMap(fooFour) { z =>
        Foo(x + y + z)
      }
    }
  }

assert(fooResult.value == 9)
```

In the past five slides, we had a go at answering the following questions, but concentrating on the **how** rather than the **what** or the **why**:

- what is a **monad**
- what is a **for comprehension**?
- why does **IO** need **map** and **flatMap** in order to be used in a **for comprehension**?
- how do the particular idioms used to implement the **IO monad** compare with alternative idioms?

We looked at the **mechanics** of what a **monad** is and how it can be implemented, including some of the idioms that can be used. We did that by looking at the simplest possible **monad**, the **identity monad**, which does nothing.

In the next four slides we turn more to the **what** and the **why** and go through a very brief recap of what a **monad** is from a **particular point of view** that is useful for our purposes in this slide deck.

If you already familiar with the concept of a **monad**, then feel free to skip the next five slides.

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.

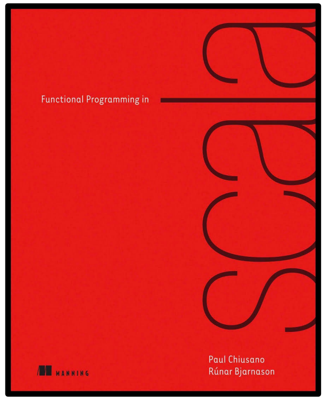
For example, with `Id`, nothing at all occurs except unwrapping and rewrapping in the `Id` constructor.

…

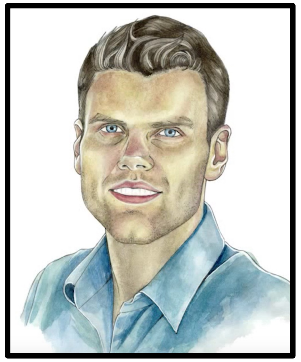
With the `Option` monad, a statement may return `None` and terminate the program.

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.



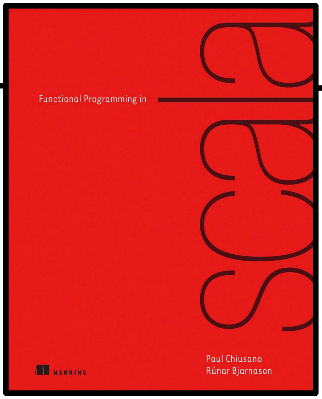
Functional Programming in Scala



@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)
}
```

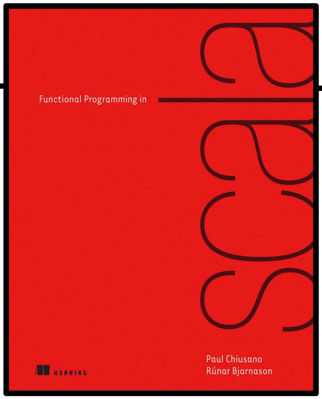
```
val result: Id[String] =
  for {
    hello <- Id("Hello, ")
    monad <- Id("monad!")
  } yield hello + monad

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**”

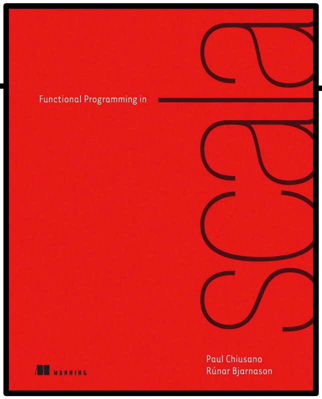
```scala
val result =
  for {
    firstNumber  <- None[String]
    secondNumber <- Some("333")
  } yield firstNumber + secondNumber

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**.



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 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))
    }
}
```

"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**."

```
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))))
)
```

After that recap on **monads** in general, let’s go back to our **IO monad**.

```
def converter: Unit = {
  println("Enter a temperature in degrees Fahrenheit: ")
  val d = readLine.toDouble
  println(fahrenheitToCelsius(d))
}
```

```
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0
```

Here on the left hand side is the initial **converter** program, which has **side effects** because when it runs, it calls **readLine**, which is an **impure function**, since it doesn't take any arguments, and **println**, which is an **impure function** because it doesn't have a return value.

And below is the new **converter** program, which instead of having **side effects**, is an **effectful** program in that when it runs, instead of calling **impure functions** like **println** and **readLine**, which result in **side effects**, simply produces a description of a computation with **side effects**. The result of running the **converter** is an **IO action**, i.e. a **pure value**. Once in possession of an **IO action**, it is possible to **interpret** the **IO action**, i.e. to **execute** the **side-effect** producing **computation** that it describes, which is done by invoking the **run** method of the **IO action.**

@philip_schwarz

The **run** function is the only **impure function** in the whole program. It is polymorphic in **A**. When **A** is **Unit** then **run** is an **impure function** because it doesn't return anything. When **A** is anything else then **run** is an **impure function** because it doesn't take any arguments.

```
def converter: IO[Unit] =
  for {
    _ <- PrintLine("Enter a temperature in degrees Fahrenheit: ")
    d <- ReadLine.map(_.toDouble)
    _ <- PrintLine(fahrenheitToCelsius(d).toString)
  } yield ()
```

```
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0
```

```
def ReadLine: IO[String] = IO { readLine }
def PrintLine(msg: String): IO[Unit] = IO { println(msg) }
```

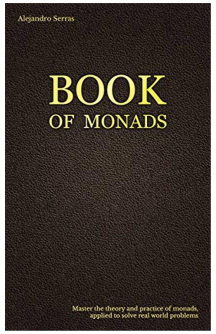
```
trait Monad[F[_]] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]
}
```

```
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}
```

```
object IO extends Monad[IO] {
  def unit[A](a: => A): IO[A] =
    new IO[A] { def run = a }
  def flatMap[A,B](fa: IO[A])(f: A => IO[B]) =
    fa flatMap f
  def apply[A](a: => A): IO[A] =
    unit(a)
}
```

Remember when in part 1 we saw that Haskell's IO values are not executed on the spot, that only expressions that have IO as their outer constructor are executed, and that in order to get nested IO values to be executed we have to use functions like sequence_?

Alejandro
Serrano Mena
@trupill



Description versus execution.

IO values are treated like any other value in Haskell: they can be used as arguments to functions, put in a list, and so on. This raises the question of when the results of such actions are visible to the outside world.

Take the following small expression:

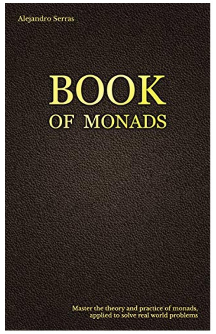
```
map putStrLn ["Alejandro", "John"]
```

If you try to execute it, you will see that nothing is printed on the screen. What we have created is a description of a list of actions that write to the screen. You can see this in the type assigned to the expression, [IO ()]. The fact that IO actions are not executed on the spot goes very well with the lazy nature of Haskell and allows us to write our own imperative control structures:

```
while :: IO Bool -> IO () -> IO ()
while cond action = do c <- cond
                       if c then action >> while cond action
                       else return ()
```

Such code would be useless if the actions given as arguments were executed immediately.

Alejandro
Serrano Mena
@trupill



There are only two ways in which we can execute the description embodied in an IO action. One is entering the expression at the GHC interpreter prompt. The other is putting it in the call trace that starts in the main function of an executable. In any case, only those expressions that have IO as their outer constructor are executed. This is the reason why the previous expression would not print anything, even in main. To get the work done, we need to use sequence_ or mapM_:

```
sequence_ (map putStrLn ["Alejandro", "John"])
-- or equivalently
mapM_ putStrLn ["Alejandro", "John"]
```

This distinction between description and execution is at the core of the techniques explained in this book for creating your own, fine-grained monads. But even for a monad with so many possible side-effects like IO, it is useful for keeping the pure and impure parts of your code separated.

What are the analogous notions when using the Scala IO monad?

One thing is instantiating an IO value, so that it wraps some side-effecting code, and another is having the wrapped side-effecting code executed, which is done by invoking the IO value's run function.

Invoking the converter function results in the instantiation of two IO values, one nested inside another. When we invoke the run function of the outer of those IO values, it results in the following:

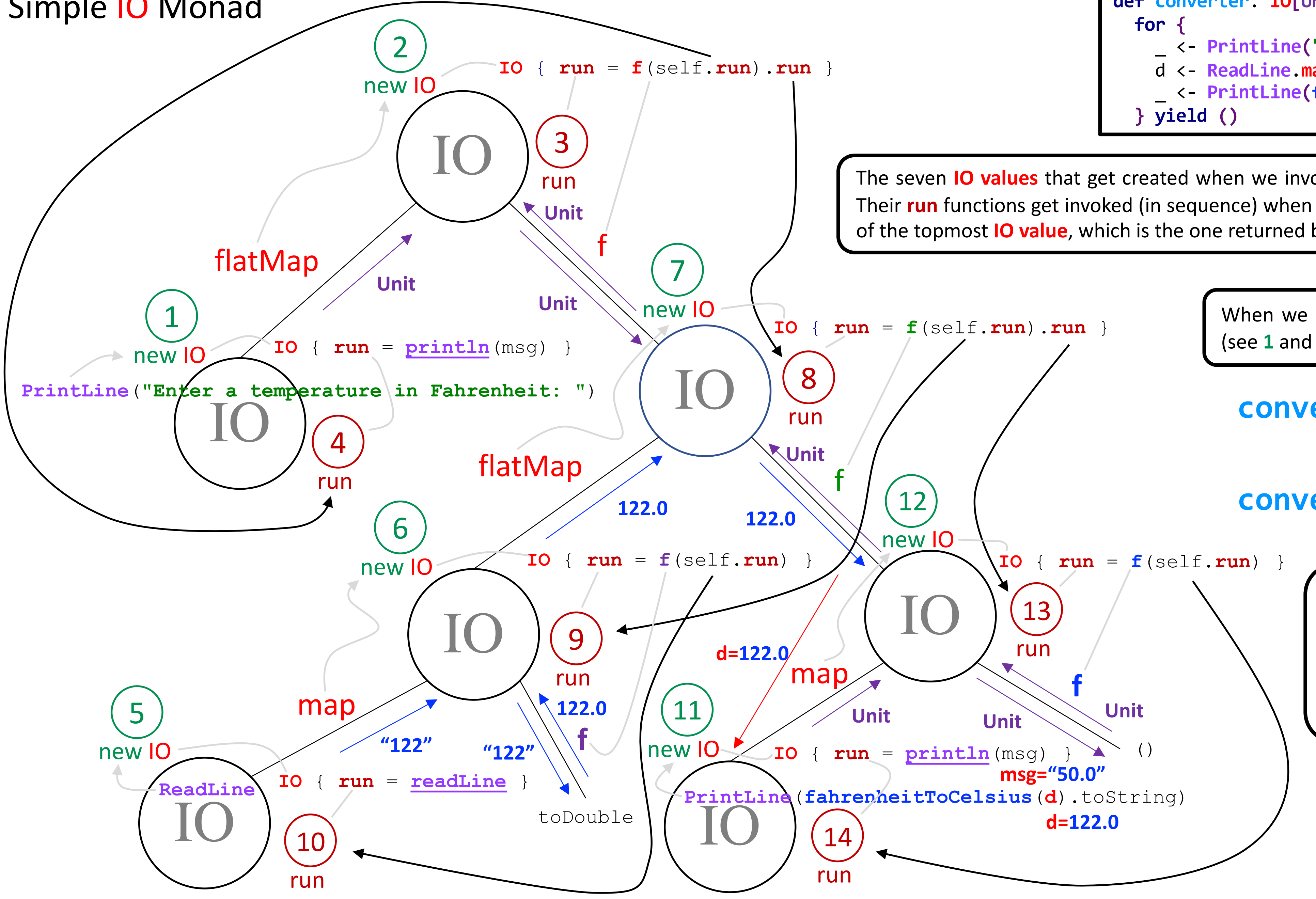
- invocation of the run function of the inner IO value
- instantiation of five further IO values
- Invocation of the run function of the above five IO values

```
def converter: IO[Unit] =
  for {
    _ <- PrintLine("Enter a temperature in degrees Fahrenheit: ")
    d <- ReadLine.map(_.toDouble)
    _ <- PrintLine(fahrenheitToCelsius(d).toString)
  } yield ()
```

On the next slide I have a go at visualising the seven IO values that are created when we first call converter and then invoke the run function of the outer IO value that the latter returns.

Simple IO Monad

```
def converter: IO[Unit] =
  for {
    _ <- PrintLine("Enter a temperature in Fahrenheit: ")
    d <- ReadLine.map(_.toDouble)
    _ <- PrintLine(fahrenheitToCelsius(d).toString)
  } yield ()
```



The seven **IO values** that get created when we invoke the **converter** function. Their **run** functions get invoked (in sequence) when we invoke the **run** function of the topmost **IO value**, which is the one returned by the **converter** function.

When we call **converter**, two **IO values** are created (see **1** and **2**), with the first nested inside the second.

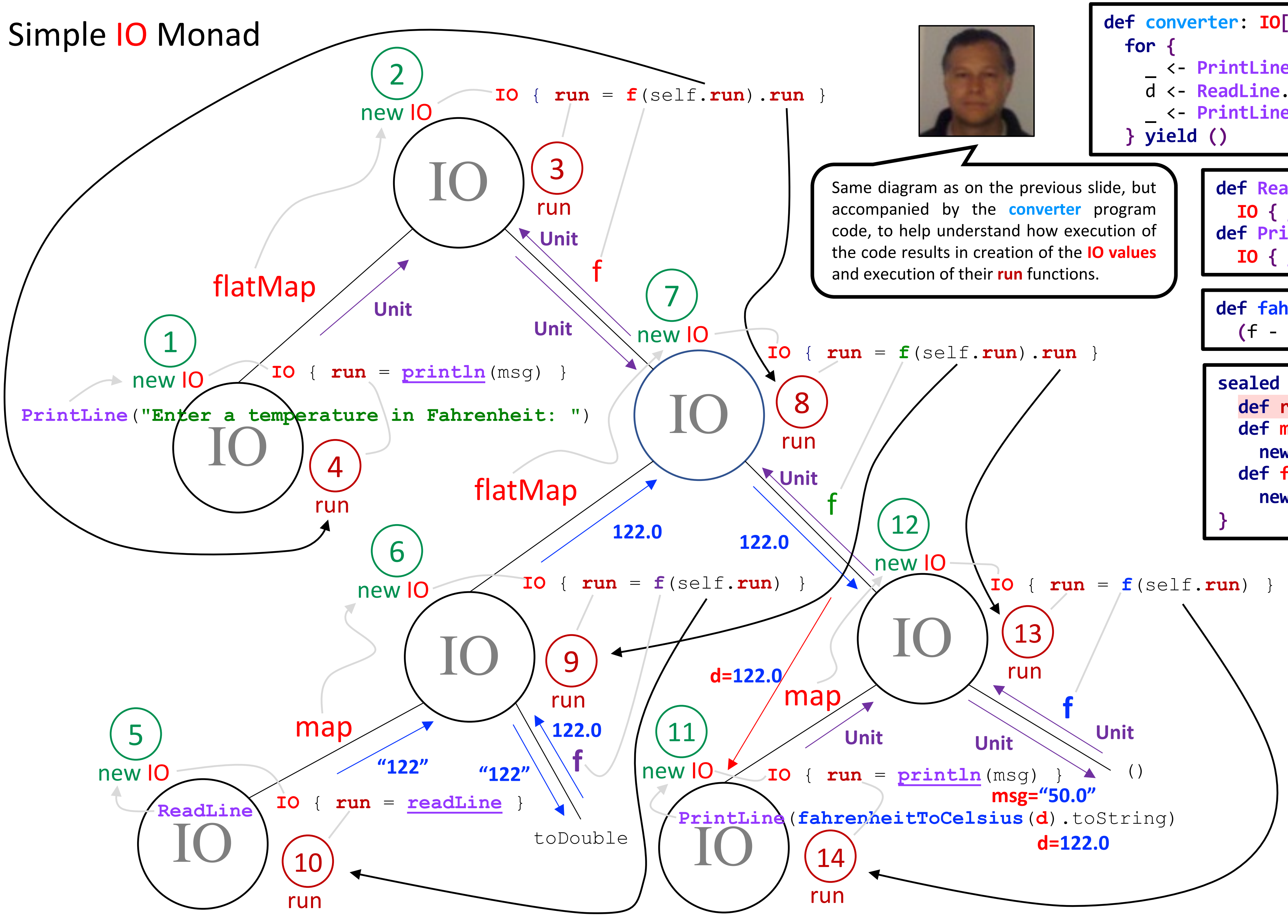
converter ⇨ 1 2
new IO new IO

converter.run ⇨ 1 … 14
new IO run

If we also invoke the **run** function of the outermost **IO value** returned by the **converter** function (see **3**) then five more **IO values** are created (see **5**,**6**,**7**,**11**,**12**) and their run functions are invoked in sequence (see **8**,**9**,**10**,**13**,**14**).

@philip_schwarz

Simple IO Monad



Same diagram as on the previous slide, but accompanied by the **converter** program code, to help understand how execution of the code results in creation of the **IO values** and execution of their **run** functions.

```
def converter: IO[Unit] =
  for {
    _ <- PrintLine("Enter a temperature in Fahrenheit: ")
    d <- ReadLine.map(_.toDouble)
    _ <- PrintLine(fahrenheitToCelsius(d).toString)
  } yield ()
```

```
def ReadLine: IO[String] =
  IO { readLine }
def PrintLine(msg: String): IO[Unit] =
  IO { println(msg) }
```

```
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0
```

```
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}
```

```
object IO extends Monad[IO] {
  def apply[A](a: => A): IO[A] =
    unit(a)
  def unit[A](a: => A): IO[A] =
    new IO[A] { def run = a }
  def flatMap[A,B](fa: IO[A])
                  (f: A => IO[B]) =
    fa flatMap f
}
```

does not get exercised

converter ⇒ 1 new IO 2 new IO

converter.run ⇒ 1 new IO … 14 run

That's quite a milestone. We are now able to write programs which instead of having **side effects**, produce an **IO action** describing a **computation with side effects**, and then at a time of our choosing, by invoking the **run** method of the **IO action**, we can **interpret** the **description**, i.e. execute the **computation**, which results in **side effects**.

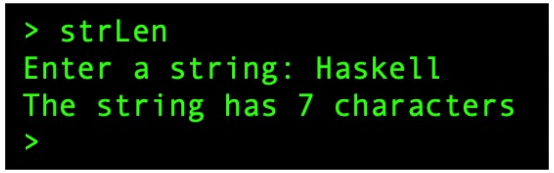
Rather than showing you a sample execution of the temperature converter, let's go back to our current objective, which is to write the **strLen** program in **Scala**.

Graham Hutton
@haskellhutt

For example, using these **primitives** we can now define an **action** that prompts for a string to be entered from the **keyboard**, and displays its length:

```haskell
strLen :: IO ()
strLen = do putStr "Enter a string: "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

For example



```
> strLen
Enter a string: Haskell
The string has 7 characters
>
```

The next thing we are going to do is see if we can translate the **Haskell derived primitive IO actions** into **Scala** and then use them to write the **Scala** equivalent of the program on the left.

To write the **strLen** program in **Scala** we need the **Scala** equivalent of the following **Haskell IO actions**:

```
getLine   :: IO String
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
```

We noticed that while the **Haskell IO actions** can be seen as **pure functions**, the corresponding **Scala** functions are **impure** because they have **side effects** (they violate one or more of the rules for **pure functions**):

```
getLine   :: World -> (String, World)
putStr    :: String -> World -> (()  World)
putStrLn  :: String -> World -> (()  World)
```

```
def readLine()      : String
def print(x: Any)   : Unit
def println(x: Any) : Unit
```

We have now seen how it is possible to define a **Scala IO type** using which we can then implement **pure functions** that are analogous to the **Haskell IO actions**:

```
getLine   :: World -> (String, World)
putStr    :: String -> World -> ((), World)
putStrLn  :: String -> World -> ((), World)
```

```
def ReadLine: IO[String]
def Print(msg: String): IO[Unit]
def PrintLine(msg: String): IO[Unit]
```

(I added this, since we are going to need it)

Conversely, we can think of the **Haskell IO actions** as being **pure**, in that instead of having **side effects**, they return **IO values**, i.e. **descriptions** of **computations** that produce **side effects** but only at the time when the **descriptions** are **interpreted**:

```
getLine   :: IO String
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
```

```
def ReadLine: IO[String]
def Print(msg: String): IO[Unit]
def PrintLine(msg: String): IO[Unit]
```

Here is how we are now able to implement the **Haskell strLen** program in **Scala**.

```
def readLine(): String
def print(x: Any): Unit
def println(x: Any): Unit
```

Scala predefined functions

```
getLine   :: IO String
putStr    :: String -> IO ()
putStrLn :: String -> IO ()

strLen :: IO ()
strLen =
  do
    putStr "Enter a string: "
    xs <- getLine
    putStr "The string has "
    putStr (show (length xs))
    putStrLn " characters”
```

```
def ReadLine: IO[String] = IO { readLine }
def Print(msg: String): IO[Unit] = IO { print(msg) }
def PrintLine(msg: String): IO[Unit] = IO { println(msg) }

def strLen: IO[Unit] =
  for {
     _ <- Print("Enter a string: ")
    xs <- ReadLine; _ <- PrintLine(xs.toString)
     _ <- Print("The string has ")
     _ <- Print(xs.length.toString)
     _ <- PrintLine(" characters")
  } yield ()
```

unlike the **Haskell getLine**, method, the **Scala readLine** method doesn’t echo to the screen, so we added this

When we get the REPL to evaluate the **Haskell** program, the result is an **IO value** that the REPL then executes, which results in **side effects**. When we execute the **Scala** program, it produces an **IO value** whose **run** function we then invoke, which results in **side effects**.

```
> strLen
Enter a string: Haskell
The string has 7 characters

>
```

```
scala> val strLenDescription: IO[Unit] = strLen
strLenDescription: IO[Unit] = StrLen $IO$$anon$12@591e1a98

scala> strLenDescription.run
Enter a string: Haskell
The string has 7 characters

scala>
```

@philip_schwarz

Now that we have successfully translated the **Haskell strLen** program into **Scala**, let’s return to the task of translating into **Scala** the **impure Haskell** functions of the **Game of Life**.

Now that we have a **Scala IO monad**, translating the first three functions is straightforward.

```
putStr :: String -> IO () =
  … (predefined)
```

```
def putStr(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }
```

```
cls :: IO ()
cls = putStr "\ESC[2J"

goto :: Pos -> IO ()
goto (x,y) =
  putStr ("\ESC[" ++ show y ++ ";"
                 ++ show x ++ "H")

writeat :: Pos -> String -> IO ()
writeat p xs = do goto p
                  putStr xs
```

```
def cls: IO[Unit] =
  putStr("\u001B[2J")

def goto(p: Pos): IO[Unit] = p match {
  case (x,y) => putStr(s"\u001B[${y};${x}H")
}

def writeAt(p: Pos, s: String): IO[Unit] =
  for {
    _ <- goto(p)
    _ <- putStr(s)
  } yield ()
```

In order to translate the next two **Haskell** functions, we need the **Scala** equivalent of the Haskell **sequence_** function, which takes a **foldable** structure containing **monadic actions**, and executes them from left to right, ignoring their result.

Earlier in **FPiS** we came across this statement: “because **IO** forms a **Monad**, we can use all the **monadic combinators** we wrote previously.”

In the code accompanying **FPiS**, the **Monad** type class contains many combinators, and among them are two combinators both called **sequence_**.

```
trait Functor[F[_]] {
  def map[A,B](a: F[A])(f: A => B): F[B]
}

trait Monad[F[_]] extends Functor[F] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]

  def map[A,B](a: F[A])(f: A => B): F[B] =
    flatMap(a)(a => unit(f(a)))

  // monadic combinators
  …

 }
```

The first one takes a stream of **monadic actions**, and the second one takes zero or more **monadic actions**.

As you can see, the two **sequence_** combinators rely on several other combinators.

```
…
def sequence_[A](fs: Stream[F[A]]): F[Unit] = foreachM(fs)(skip)
def sequence_[A](fs: F[A]*): F[Unit] = sequence_(fs.toStream)

def as[A,B](a: F[A])(b: B): F[B] = map(a)(_ => b)
def skip[A](a: F[A]): F[Unit] = as(a)(())

def foldM[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[B] =
  l match {
    case h #:: t => flatMap(f(z,h))(z2 => foldM(t)(z2)(f))
    case _ => unit(z)
  }
def foldM_[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[Unit] =
  skip { foldM(l)(z)(f) }

def foreachM[A](l: Stream[A])(f: A => F[Unit]): F[Unit] =
  foldM_(l)(())((u,a) => skip(f(a)))
…
```

“We don’t necessarily endorse writing code this way in **Scala**. But it does demonstrate that **FP** is not in any way limited in its expressiveness—every program can be expressed in a **purely functional** way, even if that functional program is a straightforward embedding of an **imperative program** into the `IO` **monad**” - **FPiS**

We are not going to spend any time looking at the other combinators.

Let’s just try out the second **sequence_** function by passing it two **strLen IO actions**.

That works.

```
scala> val strLenDescriptions: IO[Unit] = IO.sequence_(strLen,strLen)
strLenDescriptions: IO[Unit] = StrLen$IO$$anon$12@3620d817
scala> strLenDescriptions.run
Enter a string: Haskell
The string has 7 characters
Enter a string: Scala
The string has 5 characters
scala>
```

So if we change the parameter type of **Monad**’s **sequence_** function from `F[A]*` to `List[F[A]]`

```
def sequence_[A](fs: List[F[A]]): F[Unit] = sequence_(fs.toStream)
```

then we can translate from **Haskell** into **Scala** the two functions that use the **sequence_** function

```
showcells :: Board -> IO ()
showcells b = sequence_ [writeat p "O" | p <- b]

wait :: Int -> IO ()
wait n = sequence_ [return () | _ <- [1..n]]
```

```
def showCells(b: Board): IO[Unit] =
  IO.sequence_(b.map{ writeAt(_, "O") })

def wait(n:Int): IO[Unit] =
  IO.sequence_(List.fill(n)(IO.unit(())))
```

alternative version using similar syntactic sugar to the **Haskell** one:

```
IO.sequence_( for { p <- b } yield writeAt(p, "O") )
```

And now we can finally complete our **Scala Game of Life** program by translating the last two **impure functions**:

```
 life :: Board -> IO ()
 life b =
   do cls
      showcells b
      wait 500000
      life (nextgen b)


 main :: IO ()
 main = life(pulsar)

> main
```

```
def life(b: Board): IO[Unit] =
  for {
     _ <- cls
     _ <- showCells(b)
     _ <- goto(width+1,height+1) // move cursor out of the way
     _ <- wait(1_000_000)
     _ <- life(nextgen(b))
  } yield ()

val main: IO[Unit] = life(pulsar)

> main.run
```

Here is the **Scala** translation of all the **Haskell** impure functions in the **Game of Life**, plus the **IO monad** that they all use.
We have stopped calling them **impure functions**. We are now calling them **pure IO functions**. Note how the **run** function of **IO** is still highlighted with a red background because it is the only function that is **impure**, the only one that has **side effects**.

PURE IO FUNCTIONS

```scala
def putStr(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }

def cls: IO[Unit] =
  putStr("\u001B[2J")

def goto(p: Pos): IO[Unit] =
  p match { case (x,y) => putStr(s"\u001B[${y};${x}H") }

def writeAt(p: Pos, s: String): IO[Unit] =
  for {
    _ <- goto(p)
    _ <- putStr(s)
  } yield ()

def showCells(b: Board): IO[Unit] =
  IO.sequence_(b.map{ writeAt(_, "O") })

def wait(n:Int): IO[Unit] =
  IO.sequence_(List.fill(n)(IO.unit(())))

def life(b: Board): IO[Unit] =
  for {
    _ <- cls
    _ <- showCells(b)
    _ <- goto(width+1,height+1) // move cursor out of the way
    _ <- wait(1_000_000)
    _ <- life(nextgen(b))
  } yield ()

val main = life(pulsar)
```

IO MONAD

```scala
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}

object IO extends Monad[IO] {
  def unit[A](a: => A): IO[A] =
    new IO[A] { def run = a }
  def flatMap[A,B](fa: IO[A])(f: A => IO[B]) =
    fa flatMap f
  def apply[A](a: => A): IO[A] =
    unit(a)
}
```

The **run** function is the only **impure function** in the whole program. It is polymorphic in A.
When A is **Unit** then **run** is an **impure function** because it doesn't return anything. When A is anything else then **run** is an **impure function** because it doesn't take any arguments.

```scala
trait Monad[F[_]] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]
  …
  def sequence_[A](fs: List[F[A]]): F[Unit] =
    sequence_(fs.toStream)
  def sequence_[A](fs: Stream[F[A]]): F[Unit] =
    foreachM(fs)(skip)
  …
}
```

And here again are the **Scala pure functions** that we wrote in part 1.

```scala
type Pos = (Int, Int)


type Board = List[Pos]


val width = 20


val height = 20
```

```scala
def neighbs(p: Pos): List[Pos] = p match {
  case (x,y) => List(
    (x - 1, y - 1), (x, y - 1),
    (x + 1, y - 1), (x - 1, y    ),
    (x + 1, y    ), (x - 1, y + 1),
    (x,     y + 1), (x + 1, y + 1) ) map wrap }

def wrap(p:Pos): Pos = p match {
  case (x, y) => (((x - 1) % width) + 1,
                  ((y - 1) % height) + 1) }
```

```scala
def isAlive(b: Board)(p: Pos): Boolean =
  b contains p


def isEmpty(b: Board)(p: Pos): Boolean =
  !(isAlive(b)(p))


def liveneighbs(b:Board)(p: Pos): Int =
  neighbs(p).filter(isAlive(b)).length
```

PURE FUNCTIONS

```scala
def survivors(b: Board): List[Pos] =
  for {
    p <- b
    if List(2,3) contains liveneighbs(b)(p)
  } yield p

def births(b: Board): List[Pos] =
  for {
    p <- rmdups(b flatMap neighbs)
    if isEmpty(b)(p)
    if liveneighbs(b)(p) == 3
  } yield p

def rmdups[A](l: List[A]): List[A] = l match {
  case Nil => Nil
  case x::xs => x::rmdups(xs filter(_ != x)) }

def nextgen(b: Board): Board =
  survivors(b) ++ births(b)
```

```scala
val glider: Board = List((4,2),(2,3),(4,3),(3,4),(4,4))

val gliderNext: Board = List((3,2),(4,3),(5,3),(3,4),(4,4))

val pulsar: Board = List(
   (4, 2),(5, 2),(6, 2),(10, 2),(11, 2),(12, 2),

          (2, 4),(7, 4),( 9, 4),(14, 4),
          (2, 5),(7, 5),( 9, 5),(14, 5),
          (2, 6),(7, 6),( 9, 6),(14, 6),
   (4, 7),(5, 7),(6, 7),(10, 7),(11, 7),(12, 7),


   (4, 9),(5, 9),(6, 9),(10, 9),(11, 9),(12, 9),
          (2,10),(7,10),( 9,10),(14,10),
          (2,11),(7,11),( 9,11),(14,11),
          (2,12),(7,12),( 9,12),(14,12),

   (4,14),(5,14),(6,14),(10,14),(11,14),(12,14)])
```

```
~/dev/scala/game-of-life--> sbt run
…
[info] running gameoflife.GameOfLife

   000   000



 0    0 0    0
 0    0 0    0
 0    0 0    0
   000   000



   000   000
 0    0 0    0
 0    0 0    0
 0    0 0    0



   000   000

[error] (run-main-0) java.lang.StackOverflowError
[error] java.lang.StackOverflowError
[error] at gameoflife.GameOfLife$IO$$anon$3.flatMap(GameOfLife.scala:162)
[error] at gameoflife.GameOfLife$IO$.flatMap(GameOfLife.scala:164)
[error] at gameoflife.GameOfLife$IO$.flatMap(GameOfLife.scala:160)
[error] at gameoflife.GameOfLife$Monad.map(GameOfLife.scala:137)
[error] at gameoflife.GameOfLife$Monad.map$(GameOfLife.scala:137)
[error] at gameoflife.GameOfLife$IO$.map(GameOfLife.scala:160)
[error] at gameoflife.GameOfLife$Monad.as(GameOfLife.scala:145)
[error] at gameoflife.GameOfLife$Monad.as$(GameOfLife.scala:145)
[error] at gameoflife.GameOfLife$IO$.as(GameOfLife.scala:160)
[error] at gameoflife.GameOfLife$Monad.skip(GameOfLife.scala:146)
[error] at gameoflife.GameOfLife$Monad.skip$(GameOfLife.scala:146)
[error] at gameoflife.GameOfLife$IO$.skip(GameOfLife.scala:160)
[error] at gameoflife.GameOfLife$Monad.$anonfun$sequence_$1(GameOfLife.scala:141)
[error] at gameoflife.GameOfLife$Monad.$anonfun$foreachM$1(GameOfLife.scala:157)
[error] at gameoflife.GameOfLife$Monad.foldM(GameOfLife.scala:150)
[error] at gameoflife.GameOfLife$Monad.foldM$(GameOfLife.scala:148)
[error] at gameoflife.GameOfLife$IO$.foldM(GameOfLife.scala:160)
[error] at gameoflife.GameOfLife$Monad.$anonfun$foldM$1(GameOfLife.scala:150)
[error] at gameoflife.GameOfLife$IO$$anon$2.run(GameOfLife.scala:123)
[error] at gameoflife.GameOfLife$IO$$anon$2.run(GameOfLife.scala:123)
… hundreds of identical intervening lines
[error] at gameoflife.GameOfLife$IO$$anon$2.run(GameOfLife.scala:123)
[error] stack trace is suppressed; run last Compile / bgRun for the full output
[error] Nonzero exit code: 1
[error] (Compile / run) Nonzero exit code: 1
[error] Total time: 4 s, completed 20-Jun-2020 16:36:01
~/dev/scala/game-of-life-->
```

Let's run the **Scala Game Of Life** program.

It prints the first generation and then fails with a **StackOverflowError**!!!

On the console, the following line is repeated hundreds of times:

```
[error]at gameoflife.GameOfLife$IO$$anon$2.run(GameOfLife.scala:123)
```

Line number **123** is the body of the **IO flatMap** function, highlighted below in grey.

```scala
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}
```

It turns out that if we decrease the parameter that we pass to the **wait** function from 1,000,000 (1M **IO actions**!!!) down to 10,000 then the generations are displayed on the screen at a very fast pace, but the program no longer encounters a **StackOverflowError**.

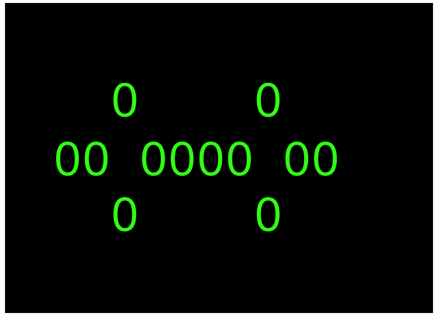
```scala
_ <- wait(10_000)
```

Alternatively, if we increase the **stack size** from the default of 1M up to 70M then the program also no longer crashes, and it displays a new generation every second or so.

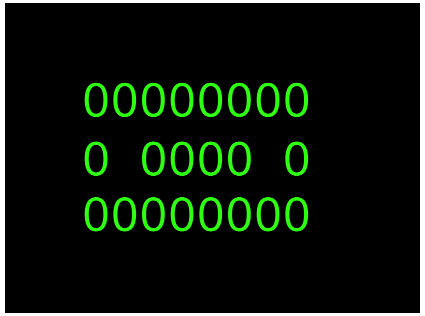
```
~/dev/scala/game-of-life--> export SBT_OPTS="-Xss70M"
```

@philip_schwarz

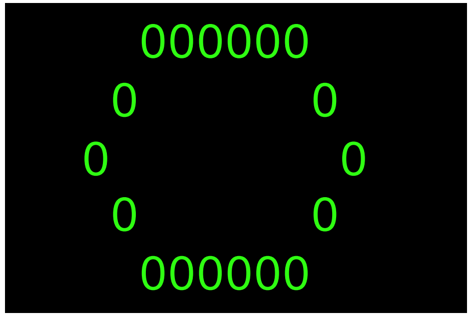
Let's try a **Pentadecathlon**, which is a period-15 **oscillator**



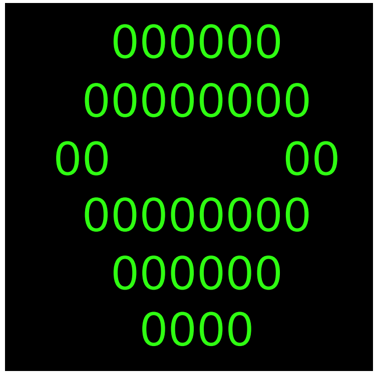
```
  0    0
00 0000 00
  0    0
```



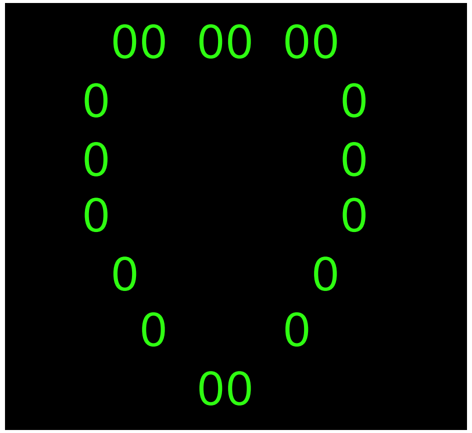
```
00000000
0 0000 0
00000000
```



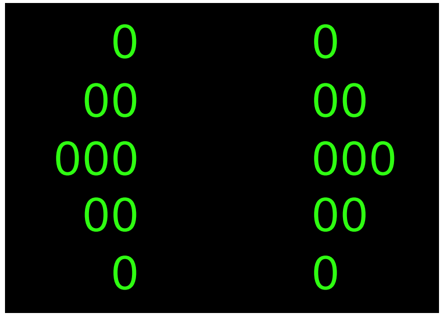
```
  000000
 0      0
0        0
 0      0
  000000
```



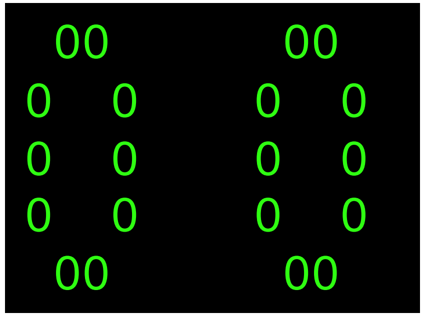
```
  000000
 00000000
00      00
 00000000
  000000
   0000
```



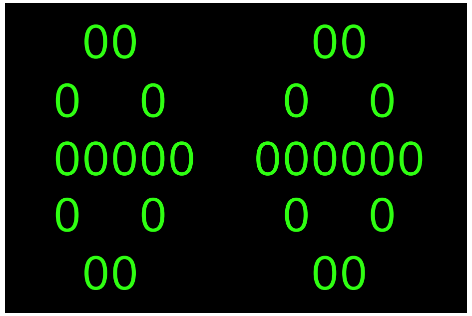
```
 00 00 00
0        0
0        0
0        0
 0      0
  0    0
    00
```



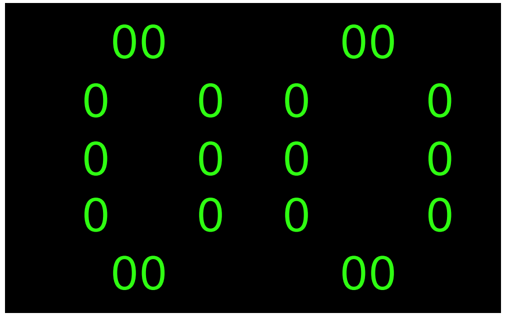
```
  0      0
 00      00
000      000
 00      00
  0      0
```



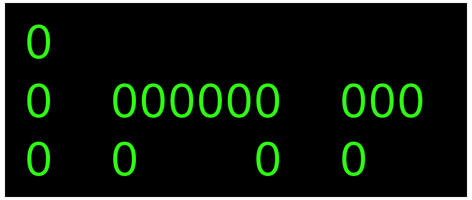
```
 00      00
0  0    0  0
0  0    0  0
0  0    0  0
 00      00
```



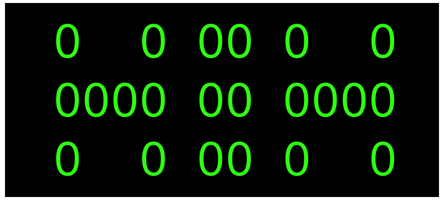
```
 00      00
0  0    0  0
00000  000000
0  0    0  0
 00      00
```



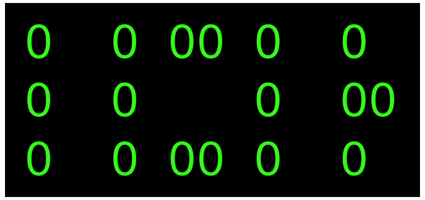
```
 00      00
0   0  0    0
0   0  0    0
0   0  0    0
 00      00
```



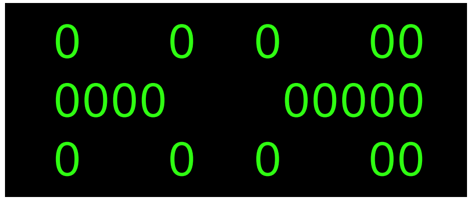
```
0
0  000000  000
0  0    0  0
```



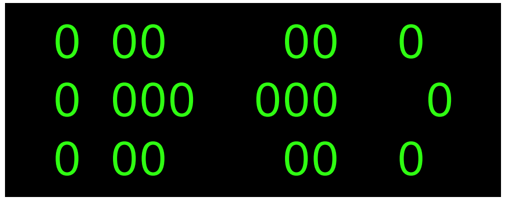
```
0  0 00 0  0
0000 00 0000
0  0 00 0  0
```



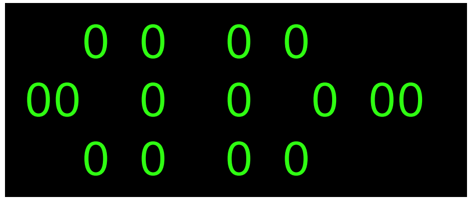
```
0  0 00 0  0
0  0    0  00
0  0 00 0  0
```



```
0   0  0   00
0000    00000
0   0  0   00
```



```
0 00    00  0
0 000  000   0
0 00    00  0
```



```
  0 0  0 0
00  0  0  0 00
  0 0  0 0
```

That’s it for Part 2.

The first thing we are going to do in part 3 is find out why the current **Scala** **IO monad** can result in programs that encounter a **StackOverflowError**, and how the **IO monad** can be improved so that the problem is avoided.

After fixing the problem, we are then going to switch to the **IO monad** provided by the **Cats Effect** library.

Translation of the **Game of Life** into **Unison** also slips to part 3.

See you there!