Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as **Trampolining** is used to overcome **Stack Overflow** issues with the simple **IO monad**

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

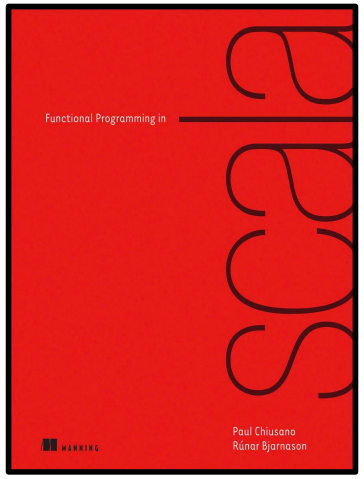
See **Game of Life IO actions** migrated to the **Cats Effect IO monad**, which is **trampolined** in its **flatMap** evaluation

(Part 3)

through the work of

Runar Bjarnason

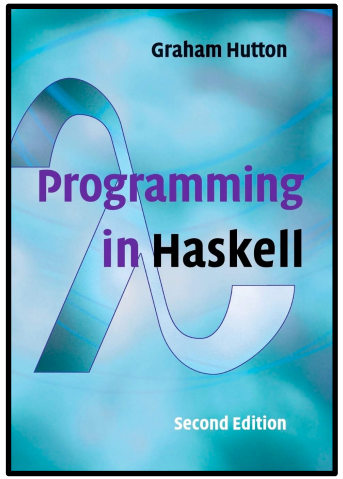
@runarorama



FP in Scala

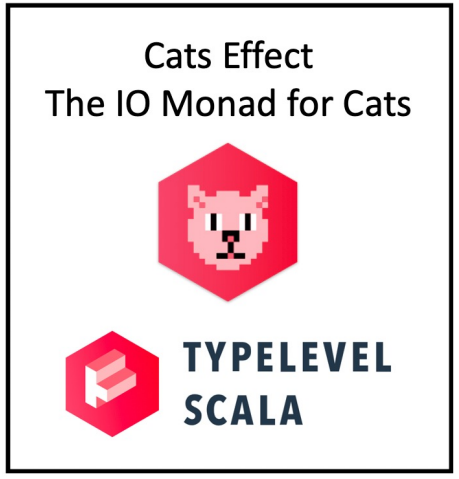
Paul Chiusano

@pchiusano



Graham Hutton

@haskellhutt



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

@philip_schwarz

At the end of part 2, when we first tried to run the **Scala Game of Life** program, it encountered a **StackOverflowError**.

We implemented the game's **I/O functions** with a particular **Scala IO Monad** which can cause programs that use it to encounter a **StackOverflowError**.

The first thing we are going to do next is see how **Functional Programming in Scala** describes the problem.

Many IO programs will overflow the runtime call stack and throw a `StackOverflowError`. If you haven't encountered this problem yet in your own experimenting, you'd certainly run into it if you were to write larger programs using our current IO type…

13.3 Avoiding the StackOverflowError

To better understand the `StackOverflowError`, consider this very simple program that demonstrates the problem:

```
val p = IO.forever(PrintLine("Still going..."))
```

```
def forever[A,B](a: F[A]): F[B] = {
  lazy val t: F[B] = flatMap(a)(_ => t)
  t
}
```

If we evaluate `p.run`, it will crash with a `StackOverflowError` after printing a few thousand lines. If you look at the stack trace, you'll see that `run` is calling itself over and over. The problem is in the definition of `flatMap`:

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

This method creates a new `IO` object whose `run` definition calls `run` again before calling `f`. This will keep building up nested `run` calls on the stack and eventually overflow it.

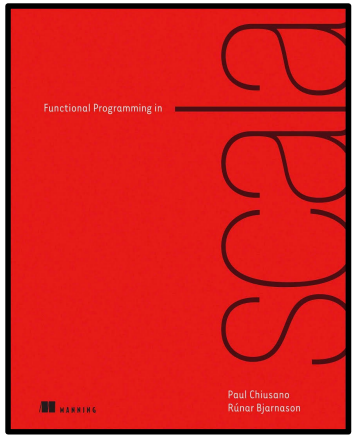
What can be done about this?

13.3.1 Reifying control flow as data constructors

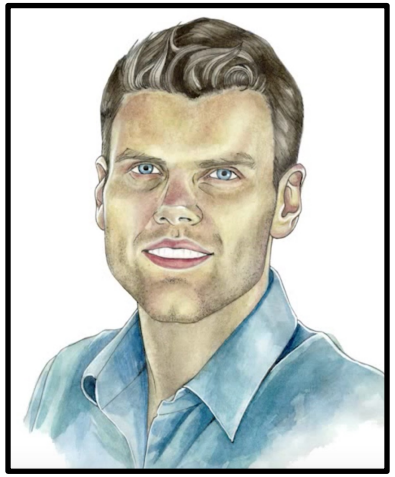
The answer is surprisingly simple. Instead of letting program control just flow through with function calls, we explicitly bake into our data type the control flow that we want to support. For example, instead of making `flatMap` a method that constructs a new `IO` in terms of `run`, we can just make it a data constructor of the `IO` data type. Then the interpreter can be a tail-recursive loop. Whenever it encounters a constructor like `FlatMap(x,k)`, it will simply interpret `x` and then call `k` on the result.

Paul Chiusano

@pchiusano



Functional Programming in Scala



Runar Bjarnason

@runarorama

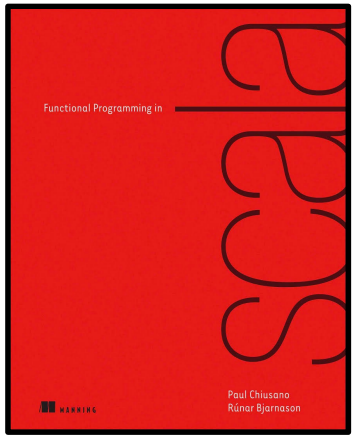
Here's a **new IO type** that implements that idea.

```
sealed trait IO[A] {
  def flatMap[B](f: A => IO[B]): IO[B] =
    FlatMap(this, f)
  def map[B](f: A => B): IO[B] =
    flatMap(f andThen (Return(_)))
}
case class Return[A](a: A) extends IO[A]
case class Suspend[A](resume: () => A) extends IO[A]
case class FlatMap[A,B](sub: IO[A], k: A => IO[B]) extends IO[B]
```

A **pure computation** that immediately returns an `A` without any further steps. When `run` sees this constructor, it knows **the computation has finished**.

A **suspension** of the **computation** where **resume** is a function that takes no arguments, but has some **effect** and yields a result.

A **composition** of two steps. **Reifies flatMap as a data constructor rather than a function**. When **run** sees this, it should first process the **subcomputation sub** and then **continue** with **k** once **sub** produces a result.



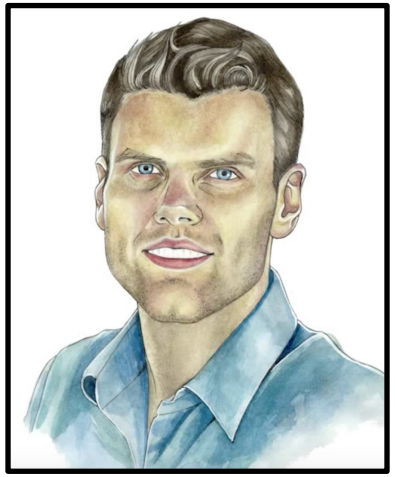
Functional Programming in Scala

This **new `IO` type** has **three data constructors**, representing the **three different kinds of control flow** that we want the interpreter of this data type to support. **`Return`** represents **an `IO` action that has finished**, meaning that we want to return the value **`a`** without any further steps. **`Suspend`** means that **we want to execute some effect to produce a result**. And the **`FlatMap`** data constructor lets us **extend or continue an existing computation by using the result of the first computation to produce a second computation**.

The `flatMap` method's implementation can now simply call the `FlatMap` data constructor and return immediately. When the interpreter encounters `FlatMap(sub, k)`, it can interpret the subcomputation `sub` and then remember to call the continuation `k` on the result. Then `k` will continue executing the program.

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

We'll get to the **interpreter** shortly, but first let's rewrite our `printLine` example to use this **new** `IO` **type**:

```
def printLine(s: String): IO[Unit] =
  Suspend(() => println(s))

val p: IO[Unit] = IO.forever(printLine("Still going..."))
```

What this actually creates is an **infinite nested structure**, much like a `Stream`. The "head" of the stream is a `Function0`, and the rest of the computation is like the "tail":

```
FlatMap(Suspend(() => println(s)),
        _ => FlatMap(Suspend(() => println(s)),
                     _ => FlatMap(...)))
```

head

tail

And here's the **tail-recursive interpreter that traverses the structure and performs the effects**:

```
@annotation.tailrec def run[A](io: IO[A]): A = io match {
  case Return(a) => a
  case Suspend(r) => r()
  case FlatMap(x, f) => x match {
    case Return(a) => run(f(a))
    case Suspend(r) => run(f(r()))
    case FlatMap(y, g) => run(y flatMap (a => g(a) flatMap f))
  }
}
```

We could just say `run(f(run(x)))` here, but then the inner call to `run` wouldn't be in **tail position**. Instead, we match on `x` to see what it is.

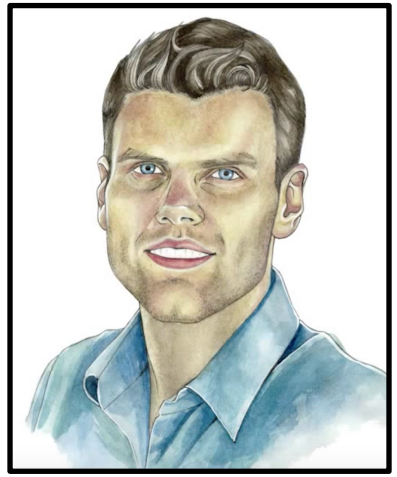
Here `x` is a `Suspend(r)`, so we force the `r` **thunk** and call `f` on the result.

In this case, `io` is an expression like `FlatMap(FlatMap(y, g), f)`. **We reassociate this to the right** in order to be able to call `run` in tail position, and the next iteration will match on `y`.

Functional Programming in Scala

Paul Chiusano

@pchiusano



Runar Bjarnason

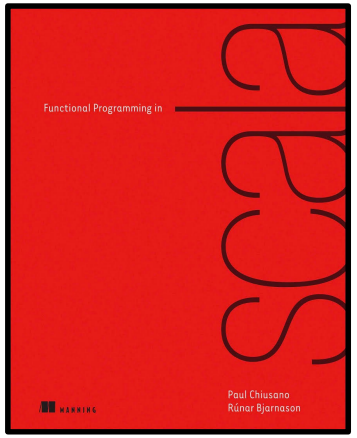
@runarorama

In the hope of aiding comprehension, on the next slide I have renamed some variables as follows:

`r` → **`resume`**
`x` → **`subComputation`**
`f` → **`continuation`**
`y` → **`subSubComputation`**
`g` → **`subContinuation`**

```scala
@annotation.tailrec
def run[A](io: IO[A]): A = io match {
  case Return(a) => a
  case Suspend(resume) => resume()
  case FlatMap(subComputation, continuation) => subComputation match {
    case Return(a) => run(continuation(a))
    case Suspend(resume) => run(continuation(resume()))
    case FlatMap(subSubComputation, subContinuation) =>
      run(subSubComputation flatMap (a => subContinuation(a) flatMap continuation))
```

Functional Programming in Scala



Note that instead of saying `run(continuation(run(subComputation)))` in the `FlatMap(subComputation, continuation)` case (thereby losing tail recursion), we instead pattern match on `subComputation`, since it can only be one of three things. If it's a `Return`, we can just call `continuation` on the pure value inside. If it's a `Suspend`, then we can just execute its resumption, call `FlatMap` with `continuation` on its result, and recurse. But if `subComputation` is itself a `FlatMap` constructor, then we know that `io` consists of two `FlatMap` constructors nested on the left like this: `FlatMap(FlatMap(subSubComputation, subContinuation), continuation)`.

In order to continue running the program in that case, the next thing we naturally want to do is look at `subSubComputation` to see if it is another `FlatMap` constructor, but **the expression may be arbitrarily deep and we want to remain tail-recursive. We reassociate this to the right, effectively turning**

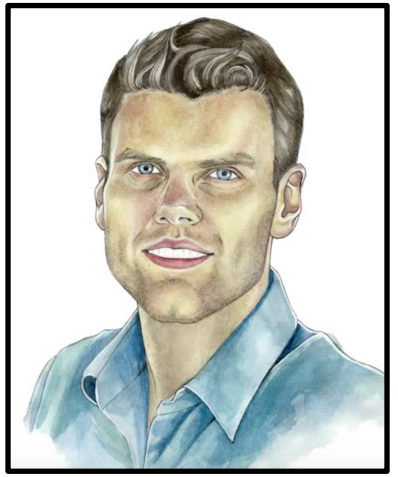
`(subSubComputation flatMap subContinuation) flatMap continuation` into
`subSubComputation flatMap (a => subContinuation(a) flatMap continuation)`.

We're just taking advantage of the **monad associativity law**! Then we call `run` on the rewritten expression, letting us remain **tail-recursive**. Thus, when we actually **interpret** our program, it will be incrementally rewritten to be a **right-associated** sequence of `FlatMap` constructors:

```scala
FlatMap(a1, a1 =>
  FlatMap(a2, a2 =>
    FlatMap(a3, a3 =>
      ...
      FlatMap(aN, aN => Return(aN)))))
```

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

@philip_schwarz

The **monadic Law of Associativity** played a key role on the previous slide, so here is a description of the law, as a reminder, or as a quick introduction.

The **flatMap** function of a **monad** is subject to the following **monadic Law of Associativity**:

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

This holds for all values `m`, `f` and `g` of the appropriate types (see right).

While on the left hand side of the equation the invocations of **flatMap** are being **chained**, on the right hand side of the equation the invocations are being **nested**.

an operation ✻ is **associative** if it doesn't matter whether we parenthesize it
((x ✻ y) ✻ z) or (x ✻ (y ✻ z))

```
m: M[A]
f: A => M[B]
g: B => M[C]
```

e.g. in the **run interpreter** function on the previous slide we have the following:

```
(subSubComputation flatMap subContinuation) flatMap continuation ≡
 subSubComputation flatMap (a => subContinuation(a) flatMap continuation)
```

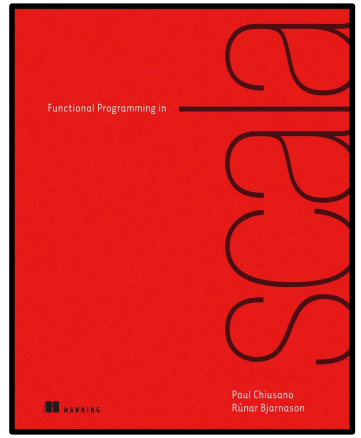
If we now pass our example program `p` to `run`, it'll continue running indefinitely without a stack overflow, which is what we want. Our `run` function won't overflow the stack, even for infinitely recursive `IO` programs.

```
scala> val p: IO[Unit] = IO.forever(printLine("Still going..."))
p: IO[Unit] = FlatMap(Suspend(Main$$$Lambda$5081/83769710@e380654),
                      Monad$$Lambda$5082/300931283@7bf87bb4)
```

Let's try it out.

```
scala> run(p)
Still going...
Still going...
Still going...
Still going...
Still going...
Still going...
<carries on indefinitely>
```

Yes, it works.

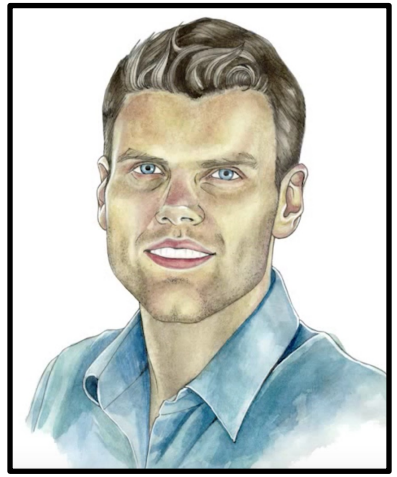


Functional Programming in Scala

Paul Chiusano

@pchiusano

What have we done here? When a program running on the JVM makes a function call, it'll push a frame onto the call stack in order to remember where to return after the call has finished so that the execution can continue. We've made this program control explicit in our `IO` data type. When `run` interprets an `IO` program, it'll determine whether the program is requesting to execute some effect with a `Suspend(s)`, or whether it wants to call a subroutine with `FlatMap(subComputation, continuation)`. Instead of the program making use of the call stack, `run` will call `subComputation()` and then continue by calling `continuation` on the result of that. And `continuation` will immediately return either a `Suspend`, a `FlatMap`, or a `Return`, transferring control to `run` again. Our `IO` program is therefore a kind of coroutine[6] that executes cooperatively with `run`. It continually makes either `Suspend` or `FlatMap` requests, and every time it does so, it suspends its own execution and returns control to `run`. And it's actually `run` that drives the execution of the program forward, one such suspension at a time. A function like `run` is sometimes called a trampoline, and the overall technique of returning control to a single loop to eliminate the stack is called trampolining.



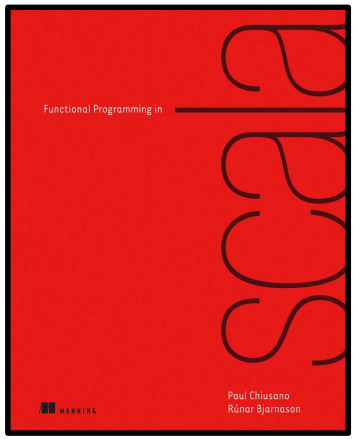
Runar Bjarnason

@runarorama

13.3.2 Trampolining: a general solution to stack overflow

Nothing says that the `resume` functions in our `IO` monad have to perform side effects. The `IO` type we have so far is in fact a general data structure for trampolining computations— even pure computations that don't do any I/O at all! The `StackOverflowError` problem manifests itself in Scala wherever we have a composite function that consists of more function calls than there's space for on the call stack. This problem is easy to demonstrate:

```
scala> val f = (x: Int) => x
f : Int => Int = $$Lambda$4211/1718489889@2303c77b
scala> val g = List.fill(100_000)(f).foldLeft(f)(_ compose _)
g: Int => Int = scala.Function1$$Lambda$4243/448419723@35da7769
scala> g(42)
java.lang.StackOverflowError
  at scala.runtime.java8.JFunction1$mcII$sp.apply(JFunction1$mcII$sp.scala:17)
  at scala.Function1.$anonfun$compose$1(Function1.scala:77)
  …<hundreds of identical intervening lines>
  at scala.Function1.$anonfun$compose$1(Function1.scala:77)
scala>
```



Functional Programming in Scala

And it'll likely fail for much smaller compositions. Fortunately, we can solve this with our `IO` monad:

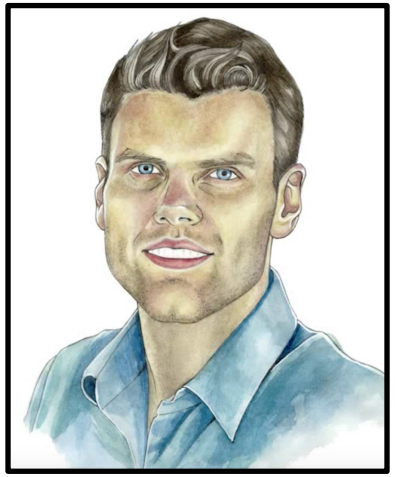
```
scala> val f: Int => IO[Int] = (x: Int) => Return(x)
f: Int => IO[Int] = $$Lambda$4429/341969473@3ab7d11c
scala> val g = List.fill(100_000)(f).foldLeft(f) {
     |   (a, b) => x => Suspend(() => a(x).flatMap(b))
     | }
g: Int => IO[Int] = $$Lambda$4432/1504554235@272067f
scala> val x1 = run(g(0))
x1: Int = 0
scala> val x2 = run(g(42))
x2: Int = 42
scala>
```

Create a large, left-nested chain of `flatMap` calls.

But there's no I/O going on here at all. So `IO` is a bit of a misnomer. It really gets that name from the fact that `Suspend` can contain a side-effecting function. But what we have is not really a monad for I/O—it's actually a monad for tail-call elimination!

Paul Chiusano

@pchiusano



Runar Bjarnason

@runarorama

Let's change its name to reflect that:

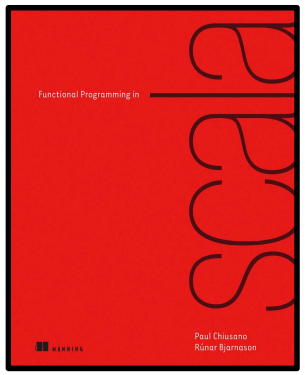
```scala
sealed trait IO[A] {
  def flatMap[B](f: A => IO[B]): IO[B] =
    FlatMap(this, f)
  def map[B](f: A => B): IO[B] =
    flatMap(f andThen (Return(_)))
}
case class Return[A](a: A) extends IO[A]
case class Suspend[A](resume: () => A) extends IO[A]
case class FlatMap[A,B](sub: IO[A],
                        k: A => IO[B]) extends IO[B]
```

```scala
sealed trait TailRec[A] {
  def flatMap[B](f: A => TailRec[B]): TailRec[B] =
    FlatMap(this, f)
  def map[B](f: A => B): TailRec[B] =
    flatMap(f andThen (Return(_)))
}
case class Return[A](a: A) extends TailRec[A]
case class Suspend[A](resume: () => A) extends TailRec[A]
case class FlatMap[A,B](sub: TailRec[A],
                        k: A => TailRec[B]) extends TailRec[B]
```

We can use the `TailRec` data type to add trampolining to any function type `A => B` by modifying the return type `B` to `TailRec[B]` instead. We just saw an example where we changed a program that used `Int => Int` to use `Int => TailRec[Int]`. The program just had to be modified to **use `flatMap` in function composition[7] and to `Suspend` before every function call**. Using `TailRec` can be slower than direct function calls, but its advantage is that we gain predictable stack usage.

7 This is just **Kleisli composition** from chapter 11. In other words, the **trampolined** function uses **Kleisli composition** in the **`TailRec` monad** instead of **ordinary function composition**.

Paul Chiusano @pchiusano



Functional Programming in Scala

Runar Bjarnason @runarorama

The errata page of **FPiS** informs us of an error in the code for function **g** on the slide before last.

@philip_schwarz

It makes sense for the **suspend** function to live in the **TailRec** object, which reminds me that we have not yet migrated the **IO** object to **TailRec** (the **FPiS** book did not mention it). If we look in the **FPiS github** repo, we see that there is a **TailRec** object and that the **suspend** function has been added to it.

https://github.com/fpinscala/fpinscala/wiki/Errata

Pg 240: REPL session has a typo, should be:

```
val g = List.fill(100000)(f).foldLeft(f) {
  (a, b) => x => Suspend(() => ()).flatMap { _ => a(x).flatMap(b)}
}
```

Note: we could write a little helper function to make this nicer:

```
def suspend[A](a: => IO[A]) = Suspend(() => ()).flatMap { _ => a }

val g = List.fill(100000)(f).foldLeft(f) {
  (a, b) => x => suspend { a(x).flatMap(b) }
}
```

```
object TailRec extends Monad[TailRec] {
  def unit[A](a: => A): TailRec[A] =
    Return(a)
 def flatMap[A,B](a: TailRec[A])(f: A => TailRec[B]): TailRec[B] =
    fa flatMap f
 def suspend[A](a: => TailRec[A]) =
    Suspend(() => ()).flatMap { _ => a }
}
```

Just for reference, here on the right is the **IO object** as it was at the time that **TailRec** was introduced

```
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)
}
```

Here is the corrected **g** function *without* using the new helper function.

And here is the corrected **g** function again, but this time using the new helper function, which is very convenient.

```
val g = List.fill(100_000)(f).foldLeft(f) {
   (a, b) => x => Suspend(() => ()).flatMap { _ => a(x).flatMap(b)}
 }
```

```
val g = List.fill(100_000)(f).foldLeft(f) {
   (a, b) => x => TailRec.suspend { a(x).flatMap(b) }
 }
```

We have just seen that a program that **composes** a function with itself a sufficiently large number of times results in a **StackOverflowError**.

E.g. if we take a function **f** that does nothing (i.e. it simply returns its argument)

```
val f = (x: Int) => x
```

and we compose **f** with itself 100,000 times, producing composite function **g**

```
val g = List.fill(100_000)(f).foldLeft(f)(_ compose _)
```

then running **g** results in a **StackOverflowError**:

```
scala> g(42)
java.lang.StackOverflowError
```

But if we make the following changes to the program and the function:

1. add **trampolining** to the function by changing it to return its result wrapped in **TailRec** i.e. change the function's type from **A => B** to **A => TailRec[B]**
2. use **flatMap** in function composition
3. **Suspend** before every function call

e.g. if we redefine **f** as follows:

```
val f: Int => TailRec[Int] = (x: Int) => Return(x)
```

and compose the **f**s as follows:

```
val g = List.fill(100_000)(f).foldLeft(f) {
  (a, b) => x => TailRec.suspend { a(x).flatMap(b) }
}
```

Then running **g** no longer results in a **StackOverflowError**:

```
assert(run(g(42)) == 42)
```

See the next slide for a recap of all the code necessary to run the above example.

```scala
sealed trait TailRec[A] {
  def flatMap[B](f: A => TailRec[B]): TailRec[B] =
    FlatMap(this, f)
  def map[B](f: A => B): TailRec[B] =
    flatMap(f andThen (Return(_)))
}

case class Return[A](a: A) extends TailRec[A]
case class Suspend[A](resume: () => A) extends TailRec[A]
case class FlatMap[A,B](sub: TailRec[A], k: A => TailRec[B]) extends TailRec[B]

implicit object TailRec extends Monad[TailRec] {
  def unit[A](a: => A): TailRec[A] = Return(a)
  def flatMap[A,B](a: TailRec[A])(f: A => TailRec[B]): TailRec[B] = a flatMap f
  def suspend[A](a: => TailRec[A]) =
    Suspend(() => ()).flatMap { _ => a }
}

@annotation.tailrec def run[A](t: TailRec[A]): A = t match {
  case Return(a) => a
  case Suspend(r) => r()
  case FlatMap(x, f) => x match {
    case Return(a) => run(f(a))
    case Suspend(r) => run(f(r()))
    case FlatMap(y, g) => run(y flatMap (a => g(a) flatMap f))
  }
}
```

```scala
val f: Int => TailRec[Int] = (x: Int) => Return(x)
val g = List.fill(100_000)(f).foldLeft(f) {
  (a, b) => x => TailRec.suspend { a(x).flatMap(b) }
}

assert(run(g(0)) == 0)
assert(run(g(42)) == 42)
```

```scala
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)))
}
```

To better understand **how to add trampolining to a function, let's add it to a very simple and familiar function: factorial**.

```
def factorial(n: Int): Int = {
  if (n == 0) 1
  else n * factorial(n - 1)
}
```

On my laptop, if I first run `export SBT_OPTS=-Xss1M` to arrange for the JVM stack size to be set to 1MB, and then run `sbt console`, I can then use the above **factorial** function to compute up to **12!**

```
scala> assert( factorial(12) == 479_001_600 )
```

But **13!** is **6,227,020,800** and does not fit into an **Int**, so the function computes the incorrect value **1,932,053,504**.

If in the signature of **factorial**, we switch from **Int** to **Long**, then we are able to compute up to **20!**

```
scala> assert( factorial(20) == 2_432_902_008_176_640_000L )
```

But **21!** is **51,090,942,171,709,440,000** and does not fit into a **Long**, so the function computes the incorrect value **-4,249,290,049,419,214,848**.

If in the signature of **factorial**, we switch from **Long** to **BigDecimal**, then we are able to compute **factorial** for some very large numbers, e.g.

```
scala> assert( factorial(3_249) == BigDecimal("6.412337688276552183884096303056808E+10000"))
```

But if we try to compute **!10,000** say, we get a **stack overflow error**:

```
scala> factorial(10_000)
java.lang.StackOverflowError
```

n	*n*!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	**479001600**
13	**6227020800**
14	87178291200
15	1307674368000
16	20922789888000
17	355687428096000
18	6402373705728000
19	121645100408832000
20	**2432902008176640000**
25	1.551121004×10^{25}
50	3.041409320×10^{64}
70	$1.197857167\times10^{100}$
100	$9.332621544\times10^{157}$
450	$1.733368733\times10^{1000}$
1000	$4.023872601\times10^{2567}$
3249	$\mathbf{6.412337688\times10^{10000}}$
10000	$\mathbf{2.846259681\times10^{35659}}$
25206	$1.205703438\times10^{100000}$
100000	$2.824229408\times10^{456573}$
205023	$2.503898932\times10^{1000004}$
1000000	$8.263931688\times10^{5565708}$
10^{100}	$10^{10101.998109775482}$

Let's fix that by adding **trampolining** to **factorial**.

Step 1: refactor expression `n * factorial(n - 1)` by extracting functions `f` and `g` and rewriting the expression in terms of the composition of `f` and `g`, i.e. `(f andThen g)(n - 1)`

```
def factorial(n: BigDecimal): BigDecimal = {
  if (n == 0) 1
  else n * factorial(n - 1)
}
```

refactor

```
def factorial(n: BigDecimal): BigDecimal = {
  if (n == 0) 1
  else {
    val f: BigDecimal => BigDecimal = factorial _
    val g: BigDecimal => BigDecimal = res => n * res
    (f andThen g)(n - 1)
  }
}
```

Step 2: apply **trampolining rules**: ① return result wrapped in **TailRec**, ② use **flatMap** in **function composition**, ③ **Suspend** before every function call

```
def factorial(n: BigDecimal): BigDecimal = {
  if (n == 0) 1
  else {
    val f: BigDecimal => BigDecimal = factorial _
    val g: BigDecimal => BigDecimal = res => n * res
    (f andThen g)(n - 1)
  }
}
```

ordinary function composition: f **andThen** g = g **compose** f

refactor

```
def factorial(n: BigDecimal):①TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else {                  ①
    val f: BigDecimal =>①TailRec[BigDecimal] = factorial _
    val g: BigDecimal =>①TailRec[BigDecimal] = res => Return(n * res)
                                                        ①
    TailRec.suspend(f(n-1).flatMap(g))
  }                  ③                ②
}
```

Kleisli composition using **flatMap**

```
def suspend[A](a: => TailRec[A]) =
  Suspend(() => ()).flatMap { _ => a }
```

```
def flatMap[B](f: A => TailRec[B]): TailRec[B] =
  FlatMap(this, f)
```

Step 3: inline f and g

```scala
def factorial(n: BigDecimal): TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else {
    val f: BigDecimal => TailRec[BigDecimal] = factorial _
    val g: BigDecimal => TailRec[BigDecimal] = res => Return(n * res)
    TailRec.suspend(f(n-1).flatMap(g))
  }
}
```

refactor

```scala
def factorial(n: BigDecimal): TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else TailRec.suspend(
    factorial(n-1).flatMap(res => Return(n * res))
  )
}
```

Now that we have added **trampolining**, we are able to compute the **factorial** of large numbers without **stack overflow**. E.g. earlier we were unable to compute **!10,000**, but now we can compute not only that, but even much larger ones, say **!1,000,000**.

@philip_schwarz

```
10,000!

scala> assert(run(factorial(10_000)) == BigDecimal("2.8462596809170545189064132121198 39E+35659") )

25,206!

scala> assert(run(factorial(25_206)) == BigDecimal("1.2057034381592326935615853755159 68E+100000") )

1,000,000!

scala> assert(run(factorial(1_000_000)) == BigDecimal("8.2639316883312400623766461031744 63E+5565708") )
```

n	*n*!
0	1
1	1
2	2
3	6
4	24
5	120
...	...
3249	$6.412337688\times10^{10000}$
10000	**$2.846259681\times10^{35659}$**
25206	**$1.205703438\times10^{100000}$**
100000	**$2.824229408\times10^{456573}$**
205023	$2.503898932\times10^{1000004}$
1000000	$8.263931688\times10^{5565708}$
10^{100}	$10^{10101.998109775482}$

About the second rule used to add **trampolining** to **factorial**:

> "use **`flatMap`** in function composition"

Remember the following footnote to that rule?

> This is just **Kleisli composition** from chapter 11. In other words, the **trampolined** function uses **Kleisli composition** in the **`TailRec` monad** instead of **ordinary function composition**.

What does that mean e.g. in the context of our **trampolined** factorial function?

The next 5 slides are a quick reminder of (or intro to) **Kleisli composition**.

Consider three functions **f, g** and **h** of the following types:

```
f: A => B
g: B => C
h: C => D
```

e.g.

```
val f:          Int => String       = _.toString
val g:       String => Array[Char] = _.toArray
val h: Array[Char] => String       = _.mkString(",")
```

We can compose these functions ourselves:

```
assert( h(g(f(12345))) == "1,2,3,4,5" )
```

Or we can compose them into a single function using **compose**, the **higher-order** function for composing ordinary functions :

```
val hgf = h compose g compose f

assert( hgf(12345) == "1,2,3,4,5" )
```

Alternatively, we can compose them using **andThen**:

```
val hgf = f andThen g andThen h

assert( hgf(12345) == "1,2,3,4,5" )
```

What about **Kleisli arrows**, which are functions of types like A => F[B], where **F** is a **monadic type constructor**: how can they be composed?

Consider three **Kleisli arrows** **f**, **g** and **h**:

```
f: A => F[B]
g: B => F[C]
h: C => F[D]
```

How can we compose f, g and h?

We can do so using **Kleisli composition**. Here is how we compose the three functions using the **fish operator**, which is the infix operator for **Kleisli Composition**:

```
f >=> g >=> h
```

And here is the signature of the **fish operator**:

```
((A => F[B]) >=> (B => F[C])) => (A => F[C])
```

e.g. here are the **Kleisli composition** functions for **Option** and **List**:

```
// Kleisli composition for Option
implicit class OptionFunctionOps[A, B](f: A => Option[B] ) {
  def >=> [C](g: B => Option[C]): A => Option[C] =
    a => f(a) match {
      case Some(b) => g(b)
      case None    => None
    }
}

// Kleisli composition for List
implicit class ListFunctionOps[A, B](f: A => List[B] ) {
  def >=> [C](g: B => List[C]): A => List[C] =
    a => f(a).foldRight(List[C]())((b, cs) => g(b) ++ cs)
}
```

@philip_schwarz

Here are examples of using **Kleisli composition** for **Option** and **List**.

>=> example for Option

```
case class Insurance(name:String)
case class Car(insurance: Option[Insurance])
case class Person(car: Option[Car])

val car: Person => Option[Car] =
  person => person.car
val insurance: Car => Option[Insurance] =
  car => car.insurance
```

```
val carInsurance: Person => Option[Insurance] =
  car >=> insurance
```

```
val non-driver= Person(car=None)
val uninsured = Person(Some(Car(insurance=None)))
val insured = Person(Some(Car(Some(Insurance("Acme")))))

assert(carInsurance(non-driver).isEmpty )
assert(carInsurance(uninsured).isEmpty )
assert(carInsurance(insured).contains(Insurance("Acme")))
```

>=> example for List

```
val toChars: String => List[Char] = _.toList
val toAscii: Char => List[Char] =
_.toInt.toString.toList

assert( toChars("AB") == List('A','B'))
assert( toAscii('A') == List('6','5') )
```

```
val toCharsAscii: String => List[Char] =
  toChars >=> toAscii
```

```
assert(toCharsAscii("AB") == List('6','5','6','6'))
```

Note that it is the **OptionFunctionOps** and **ListFunctionOps** **implicits** that are making the fish operator **>=>** available on **car** and **toChars** (by wrapping around them):

```
OptionFunctionOps(car) >=> insurance
ListFunctionOps(toChars) >=> toAscii
```

While we have seen **Kleisli composition** defined separately for **Option** and for **List**, it is possible to define it in terms of the **Monad** interface, i.e. define a **Kleisli composition** operator that works for every instance of the **Monad** interface

```
  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)))
  }

implicit class MonadFunctionOps[F[_], A, B](f: A => F[B] ) {
  def >=>[C](g: B => F[C])(implicit M:Monad[F]): A => F[C] =
    a => M.flatMap(f(a))(g)
}
```

Note that unlike the >=> for **Option** and **List**, the generic **>=>** above takes an **implicit Monad** instance as a parameter.

Let's go back to the question that led us to look at **Kleisli composition**.

What does **FPiS** mean, e.g. in the context of our **trampolined factorial** function, when it first says that to add **trampolining** we must

> "use **`flatMap`** in function composition"

and then says the following about **flatMap**

> This is just **Kleisli composition** from chapter 11. In other words, the **trampolined** function uses **Kleisli composition** in the **`TailRec` monad** instead of **ordinary function composition**.

Now that we have defined a generic **>=>** operator, let's go back to how the origina **factorial** function and the **trampolined factorial** function looked like before inlining **f** and **g**

```
def factorial(n: BigDecimal): BigDecimal = {
  if (n == 0) 1
  else {
    val f: BigDecimal => BigDecimal = factorial _
    val g: BigDecimal => BigDecimal = res => n * res
    (f andThen g)(n - 1)
  }
}
```

```
def factorial(n: BigDecimal): TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else {
    val f: BigDecimal => TailRec[BigDecimal] = factorial _
    val g: BigDecimal => TailRec[BigDecimal] = res => Return(n * res)
    TailRec.suspend(f(n-1).flatMap(g))
  }
}
```

On the next slide we are going to see the above two function definitions again, but after extracting (in both) a function **h** that is the **composition** of **f** and **g**.

```scala
def factorial(n: BigDecimal): BigDecimal = {
  if (n == 0) 1
  else {
    val f: BigDecimal => BigDecimal =
      factorial _
    val g: BigDecimal => BigDecimal =
      res => n * res
    val h = f andThen g
    h(n - 1)
  }
}
```

original factorial function composes f and g using ordinary function **composition**

```scala
def factorial(n: BigDecimal): TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else {
    val f: BigDecimal => TailRec[BigDecimal] =
      factorial _
    val g: BigDecimal => TailRec[BigDecimal] =
      res => Return(n * res)
    val h = a => f(a).flatMap(g)
    TailRec.suspend(h(n - 1))
  }
}
```

trampolined factorial function composes f and g using **flatMap**

The f and g in the original **factorial** function are ordinary functions of type `BigDecimal => BigDecimal`, whereas the f and g in the **trampolined** factorial are **Kleisli arrows** of type `BigDecimal => TailRec[BigDecimal]`.

In the original **factorial** function we are composing two functions f and g using ordinary function **composition**, which in **Scala** can be written either as f **andThen** g or as g **compose** f (g after f), and which in mathematics is written as $g \circ f$. In the refactored **factorial** function instead, we are composing **Kleisli arrows** using **Kleisli composition**, which becomes more obvious when we consider the following definition of **Kleisli composition** in terms of **flatMap**:

$$f \Longrightarrow g \equiv a \Rightarrow f(a) \text{ flatMap } g$$

Let's make that even more explicit by refactoring the **trampolined** function so that instead of computing the **Kleisli composition** of f and g using **flatMap**, it does so using the **fish operator** (notice the **TailRec monad**, in light gray, being passed into the **fish operator** implicitly).

```scala
// composition of ordinary functions
g ∘ f

// composition of Kleisli arrows using flatMap
a ⇒ f(a) flatMap g

// composition of Kleisli arrows using fish operator
(f ⟹ g)(TailRec)
```

```scala
def factorial(n: BigDecimal): TailRec[BigDecimal] = {
  if (n == 0) Return(1)
  else {
    val f: BigDecimal => TailRec[BigDecimal] =
      factorial _
    val g: BigDecimal => TailRec[BigDecimal] =
      res => Return(n * res)
    val h = (f >=> g)(TailRec)
    TailRec.suspend(h(n - 1))
  }
}
```

trampolined factorial function that composes f and g using >=>

@philip_schwarz

After that look at **Klesli composition**, let's now go back to solving the problem that we are facing with our **Scala Game of Life** program, i.e. that the **IO monad** currently used by the program causes the program to encounter a **StackOverflowError**, unless we either reduce the number of **IO actions** created by the **wait** function from 1 million down to 10,000, which speeds up the rate at which new generations are displayed on the screen, or we increase the stack size to a hefty 70MB.

FPiS says the following:

> "If we use **TailRec** as our **IO type**, this solves the **stack overflow** problem"

So let's have a go at doing that.

On the next slide we take all the **Game of Life** functions that reference the **IO monad** and replace those references with references to **TailRec**.

```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: IO[Unit] = life(pulsar)
```

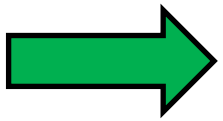
```scala
def putStr(s: String): TailRec[Unit] =
  TailRec { scala.Predef.print(s) }

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

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

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

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

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

def life(b: Board): TailRec[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: TailRec[Unit] = life(pulsar)
```

That was easy, but there is a problem: as you may have noticed when we looked at the **FPiS** errata page, the **TailRec** object that we got from the **FPiS github** repository does not have an **apply function**, so the body of the **putStr** function does not compile.

@philip_schwarz

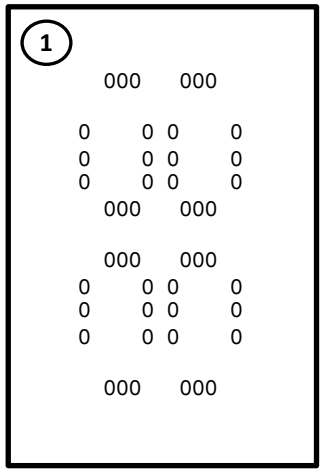
Let's take the **apply function** provided by the **IO monad** and migrate it to **TailRec**.

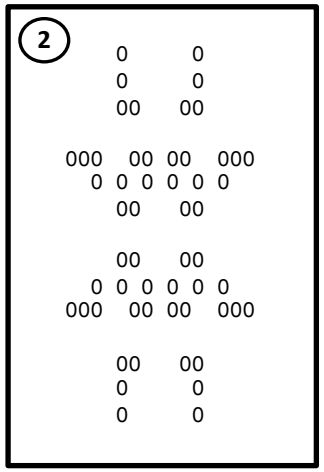
```
def apply[A](a: => A): IO[A] =
  unit(a)
```

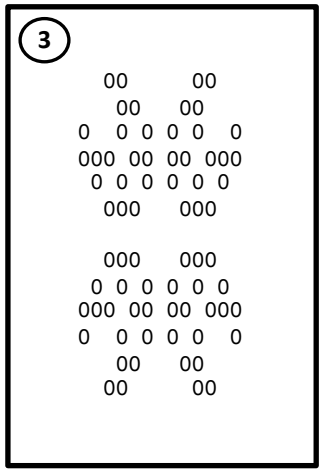


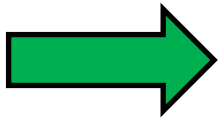
```
def apply[A](a: => A): TailRec[A] =
  unit(a)
```

So now we are ready to run the program. But there is a surprise. If we execute `run(main)`, we see that the three repeating **pulsar** generations now get flattened!!!









oooooooooooooooooooooooooooooooooooooooooooooooooo 1

ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 2

oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo 3

If we simply call the **showCells** function with a board consisting of four cells lying on a diagonal line, and then run the resulting **TailRec** action, we also see some strange behavior.

```
scala>

scala>

scalprogram: GameOfLife.TailRec[Unit] =
FlatMap(FlatMap(FlatMap(FlatMap(FlatMap(Return(()),GameOfLife$$$Lambda$4228/1447
24440@5150d661),GameOfLife$Monad$$Lambda$4233/1670634729@288d85be),GameOfLife$Mo
nad$$Lambda$4233/1670634729@599881b0),GameOfLife.Monad$$Lambda$4234/423129645@91
2eafe),GameOfLife$Monad$$Lambda$4233/1670634729@5ea6c663)

scala> run(program)
0000

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala> val program: TailRec[Unit] = showCells(List((2,2),(3,3),(4,4),(5,5)))
```

2

And when I later pass the **program** variable to the **run** function, the cells *do* get displayed on the console, as expected, but not at the expected screen positions: they are displayed on the next line and in a straight line.

1

When I call **showCells**, the cells do not get displayed on the console, as expected, but oddly, the REPL cursor moves to screen positions (2,2), (3,3), (4,4) and (5,5) in sequence, and as a result, the REPL's console output showing the value of the **program** variable gets displayed at screen position (5,5)!

If we look back at how **showCells** is implemented, we are reminded that both the **actions** that move the cursor about on the screen and the **actions** that display the cells, are created by the **putStr** function, which creates an **action** by calling the **apply** function of the **TailRec** object.

If we make calls to putStr and goto, we see that as things stand, both of them actually have side effects, despite returning a **TailRec**[**Unit**]:

```
// actually moves the cursor about on the screen
goto((15,15))

// actually displays ‘O’ on the screen
putStr("O")
```

Why is that? Since **goto** is defined in terms of **putStr**, we can just examine the behavior of **putStr**. The following

```
TailRec { print("X") }
```

does not cause X to be displayed on the screen, because the argument of **TailRec**’s **apply** function is a **thunk** (a **by-name** parameter). The above evaluates to the following

```
TailRec.unit( print("X") )
```

which also does not cause X to be printed to the screen, because the argument of the **TailRec monad**’s **unit** function is also a **thunk**. The above evaluates to the following

```
Return( print("X") )
```

Which, of course, does cause X to be displayed on the screen, since the parameter of **case class** constructor **Return** is not a **thunk**, it is a plain, **by-value** parameter.

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

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

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

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

```
def apply[A](a: => A): TailRec[A] =
  unit(a)
```

```
def unit[A](a: => A): TailRec[A] =
  Return(a)
```

```
case class Return[A](a: A) extends TailRec[A]
```

By the way, if you could do with an explanation of the term **by-name parameter**, then see the following

slideshare https://www.slideshare.net/pjschwarz/non-strict-functions-bottom-and-scala-byname-parameters

non-strict functions, bottom and Scala by-name parameters

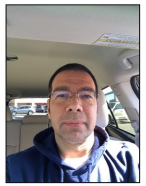
'a close l⊙⊙k'

through the work of

Runar Bjarnason
@runarorama

Paul Chiusano
@pchiusano

Martin Odersky @odersky
Bill Venners @bvenners
Lex Spoon
https://www.linkedin.com/in/lex-spoon-65352816/



Alvin Alexander
@alvinalexander

Mark Lewis
@DrMarkCLewis

Aleksandar Prokopec
@alexprokopec

slides by  @philip_schwarz

So TaiRec's apply function is not suspending the computations that it is being passed: it allows the computations to be carried out, which results in side effects happening immediately, when actions get created, rather than at a later time when the actions are interpreted by TailRec's run function.

But why is it that when we execute `run(life(pulsar))` we see each generation of cells being displayed as a flat line of cells? It is as if the the side effects of the actions created by the goto function all happen together, before or after the side effects of the actions that display the cells, rather than interspersed with them.

The reason is that writeAt first calls goto and then calls putStr, and it does so in a for comprehension, which means that the two calls get composed using flatMap.

While direct calls to goto and putStrln immediately result in side effects, when the two are chained together using flatMap, only the first one results in an immediate side effect.

e.g. invoking `goto((15,15))` in the REPL causes the cursor to move on the console, and invoking `putStr("O")` in the REPL causes "O " to be displayed on the console, but executing `goto((15,15)).flatMap(_ => putStr("O"))` only causes the cursor movement, because while `goto((15,15))` gets evaluated and its side effect takes place, rather than flatMap invoking `_ => putStr("O")` with the result, thereby causing a second side effect, flatMap just creates a FlatMap object containing both the result and the `_ => putStr("O")` function.

```
scala>         res0:
gameoflife.GameOfLifeFPiSTrampolinedIO.TailRec[Unit] =
FlatMap(Return(()),$$Lambda$4234/1708992557@50969c5e)

scala>
…
scala> goto((15,15)).flatMap(_ => putStr("O"))
```

The `putStr("O")` in the FlatMap object will only get executed if/when the result of the flatMap gets interpreted by TailRec's run function.

```
def apply[A](a: => A): TailRec[A] =
  unit(a)
```

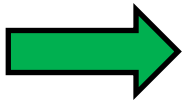
```
def writeAt(p: Pos, s: String): TailRec[Unit] =
  for {
    _ <- goto(p)
    _ <- putStr(s)
  } yield ()
```

That explains why the cursor movements for a generation in the Game of Life all happen before any of the generation's cells get displayed, which results in all the cells being displayed on a single line, one after the other.

So all we have to do to fix the problem is change the apply function of **TailRec** so that it suspends the computation represented by the parameter it is passed.

@philip_schwarz

```
def apply[A](a: => A): TailRec[A] =
  unit(a)
```



```
def apply[A](a: => A): TailRec[A] =
  Suspend(() => a)
```

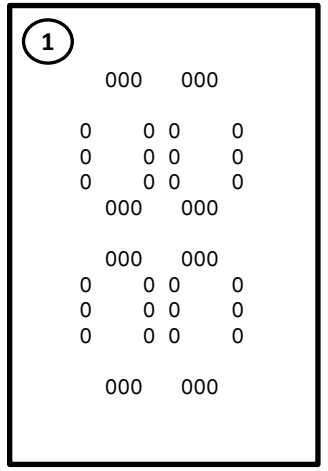
```
case class Suspend[A](resume: () => A) extends TailRec[A]
```

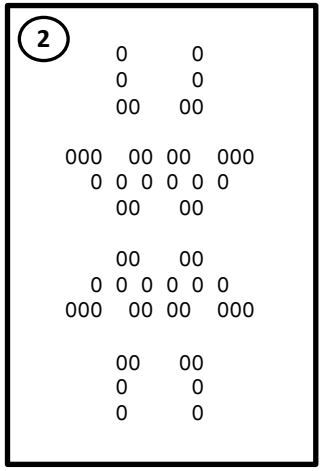
Now if we decrease the stack from 70MB down to a very low 0.5MB (`export SBT_OPTS=-Xss512K`), and we bring the number of dummy actions created by the **wait** function back up from **10,000** to **1,000,000**, and invoke `run(main)` in the **REPL**, the flattened generations are replaced by two-dimensional ones, new generations are displayed at a convenient rate of once per second, and the program just keeps running and running: it no longer encounters a **StackOverflowError**.

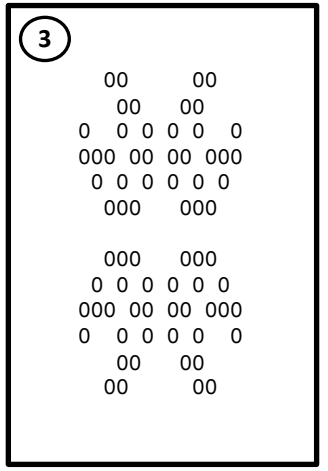
```
oooooooooooooooooooooooooooooooooooooooooooooooooo  1
```

```
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo  2
```

```
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo  3
```







We have learned how to write an **IO monad**, albeit one susceptible to **StackOverflowError**, and how to write **TailRec**, a **monad** that improves on **IO** by eliminating the **StackOverflowError**.

While **Haskell** provides a predefined **IO monad**, **Scala** does not.

But **Scala** developers are not expected to write monads like **IO** and **TailRec** by themselves. The **Cats** functional programming library, for example provides an **IO monad**, and its **flatMap** function is **trampolined**:

https://typelevel.org/cats-effect/datatypes/io.html

Stack Safety

`IO` is trampolined in its `flatMap` evaluation. This means that you can safely call `flatMap` in a recursive function of arbitrary depth, without fear of blowing the stack:

```
def fib(n: Int, a: Long = 0, b: Long = 1): IO[Long] =
  IO(a + b).flatMap { b2 =>
    if (n > 0)
      fib(n - 1, b, b2)
    else
      IO.pure(a)
  }
```

Cats Effect
The IO Monad for Cats

TYPELEVEL
SCALA

Note how the **FPiS IO.unit** function is called **IO.pure** in **Cats** (**IO** is actually in **Cats Effect** btw, but sometimes we'll just say it is in **Cats**).

Let's see, on the next slide, how the **factorial** function, which we have already migrated to the **FPiS TailRec monad** (in order to avoid **stack overflow** errors), can be migrated to the **Cats IO monad**.

initial **factorial** function

using the **FPiS IO monad**

```
def factorial(n: BigDecimal)
    : BigDecimal =

  if (n == 0)
    1
  else
    n * factorial(n-1)
```

```
def factorial(n: BigDecimal)
    : IO[BigDecimal] =

  if (n == 0)
    IO.unit(1)
  else
    factorial(n-1).flatMap(
      res => IO.unit(n * res))
```

```
val fac10: BigDecimal =
  factorial(10)

assert(
  fac10
  ==
  3_628_800
)
```

```
val fac10: IO[BigDecimal] =
  factorial(10)

assert(
  fac10.run
  ==
  3_628_800
)
```

```
factorial(1_000_000)
```

```
factorial(1_000_000).run
```

java.lang.**StackOverflowError**

using the **FPiS TailRec monad**

using the **Cats IO monad**

```
def factorial(n: BigDecimal)
    : TailRec[BigDecimal] =

  if (n == 0)
    Return(1)
  else
    TailRec.suspend(
      factorial(n-1).flatMap(
        res => Return(n * res)))
```

```
import cats.effect.IO

def factorial(n: BigDecimal)
    : IO[BigDecimal] =

  if (n == 0)
    IO.pure(1)
  else
    IO.suspend(
      factorial(n-1).flatMap(
        res => IO.pure(n * res)))
```

```
val fac10: TailRec[BigDecimal] =
  factorial(10)

assert(
  run(fac10)
  ==
  3_628_800
)
```

```
val fac10: IO[BigDecimal] =
  factorial(10)

assert(
  fac10.unsafeRunSync
  ==
  3_628_800
)
```

```
run(factorial(1_000_000))
```

```
factorial(1_000_000).unsafeRunSync
```

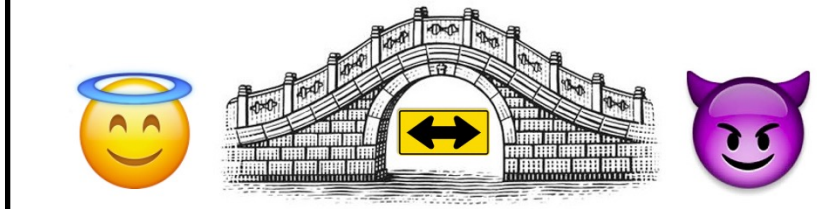
BigDecimal("8.263931688331240062376646103174463E+5565708")

@philip_schwarz

What is that **unsafeRunSync** method that we are calling on the **Cats IO monad** returned by **factorial**?

```
factorial(1_000_000).unsafeRunSync
```

Remember how in **Haskell**, the **unsafePerformIO** function is the opposite of the **return** function?



Graham Hutton

@haskellhutt

The function return provides a bridge from pure expressions without side-effects to impure actions with side-effects.

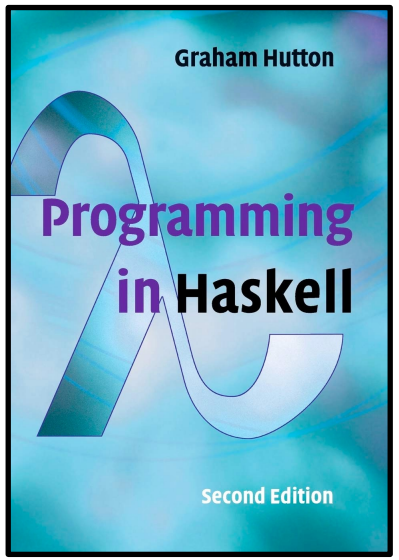
```
return :: a -> IO a
return v = ...
```

Crucially, there is no bridge back — once we are impure we are impure for ever, with no possibility for redemption!

As a result, we may suspect that **impurity** quickly permeates entire programs, but in practice this is usually not the case. For most **Haskell** programs, the vast majority of functions do not involve **interaction**, with this being handled by a relatively small number of **interactive** functions at the outermost level.

…

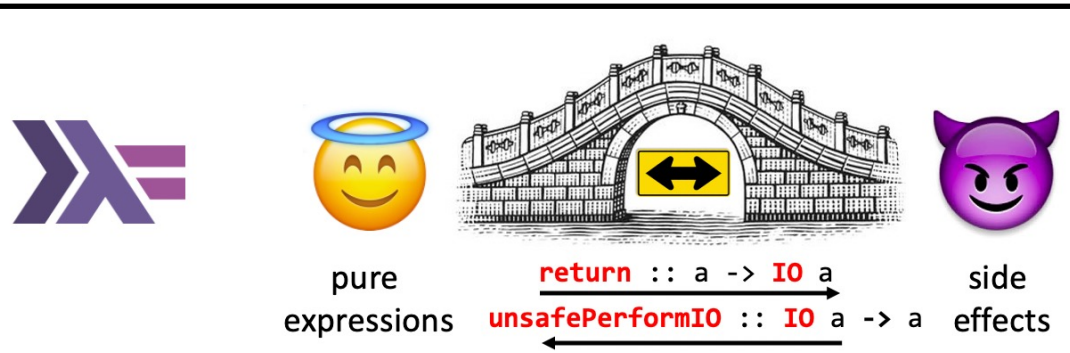
For specialised applications, a bridge back from impure actions to pure expressions is in fact available via the function **unsafePerformIO :: IO a -> a** in the library **System.IO.Unsafe** However, as suggested by the naming, this function is unsafe and should not be used in normal **Haskell** programs as it compromises the purity of the language



IO.pure and IO.unsafeRunSync are the Cats equivalent of Haskell's return and unsafePerformIO functions.

Also shown on this slide, the IO.suspend function that we used when we migrated factorial to Cats.

```
/**
 * Suspends a pure value in `IO`.
 *
 * This should ''only'' be used if the value in question has
 * "already" been computed!  In other words, something like
 * `IO.pure(readLine)` is most definitely not the right thing to do!
 * However, `IO.pure(42)` is correct and will be more efficient
 * (when evaluated) than `IO(42)`, due to avoiding the allocation of
 * extra thunks.
 */
def pure[A](a: A): IO[A] = Pure(a)
```




```
/**
 * Produces the result by running the encapsulated effects as impure
 * side effects.
 *
 * If any component of the computation is asynchronous, the current
 * thread will block awaiting the results of the async computation.
 * On JavaScript, an exception will be thrown instead to avoid
 * generating a deadlock. By default, this blocking will be
 * unbounded.  To limit the thread block to some fixed time, use
 * `unsafeRunTimed` instead.
 *
 * Any exceptions raised within the effect will be re-thrown during
 * evaluation.
 *
 * As the name says, this is an UNSAFE function as it is impure and
 * performs side effects, not to mention blocking, throwing
 * exceptions, and doing other things that are at odds with
 * reasonable software.  You should ideally only call this function
 * *once*, at the very end of your program.
 */
final def unsafeRunSync(): A = ...
```

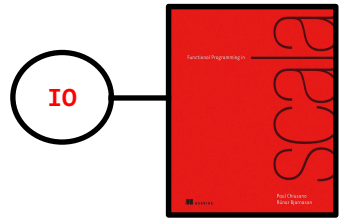
```
/**
 * Suspends a synchronous side effect which produces an `IO` in `IO`.
 *
 * This is useful for trampolining (i.e. when the side effect is
 * conceptually the allocation of a stack frame).  Any exceptions
 * thrown by the side effect will be caught and sequenced into the
 * `IO`.
 */
def suspend[A](thunk: => IO[A]): IO[A] =
  Suspend(() => thunk)
```

What we are going to do next is take our **Scala Game of Life actions** and adapt them so that instead of using the hand-rolled **IO** and **TailRec** abstractions that we have seen so far, they use the **Cats Effect IO monad**.

Cats Effect
The IO Monad for Cats

TYPELEVEL
SCALA

```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_( for { p <- b } yield writeAt(p, "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)
    _ <- wait(1_000_000)
    _ <- life(nextgen(b))
  } yield ()

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

main.run
```

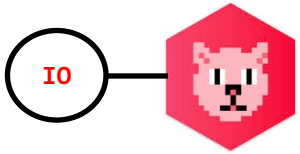


To migrate the **IO actions** from the **FPiS IO monad** to the **Cats Effect IO monad**, there is very little to do!

See the next slide for why in **Cats** we are able to invoke **sequence_** this way.

Also, in **Cats Effect**, **IO.unit** is an alias for **IO.pure**(()).

```scala
import cats.implicits._
import cats.effect.IO

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

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

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

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

main.unsafeRunSync
```



In our two usages of **sequence_** on the previous slide we have some list **x** of type **List[IO[Unit]]**, and **Cats** makes available type class instances **Foldable[List]** and **Applicative[IO]** (available implicitly and also summonable), so we can say **Foldable[List].sequence_(x)**.

```
def sequence_[G[_], A](fga: F[G[A]])(implicit arg0: Applicative[G]): G[Unit]
```

Sequence `F[G[A]]` using `Applicative[G]`.

This is similar to `traverse_` except it operates on `F[G[A]]` values, so no additional functions are needed.

For example:

```
scala> import cats.implicits._
scala> val F = Foldable[List]
scala> F.sequence_(List(Option(1), Option(2), Option(3)))
res0: Option[Unit] = Some(())
scala> F.sequence_(List(Option(1), None, Option(3)))
res1: Option[Unit] = None
```

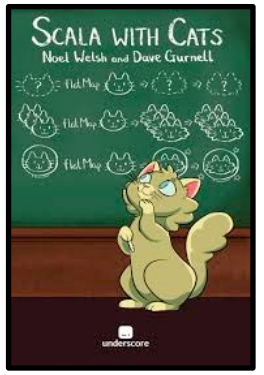
```
Foldable[List]; Applicative[IO]
F = List; G = IO; A = Unit; F[G[A]] = List[IO[Unit]]
sequence_(fga: List[IO[Unit]]): IO[Unit]
```

typelevel.org/cats/api/cats/Foldable.html

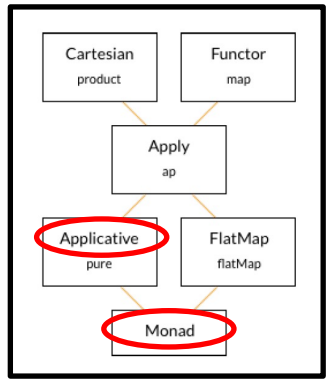
Search

cats

Foldable



Every **Monad**, e.g. **IO**, is also an **Applicative**




```
def traverse_[G[_], A, B](fa: F[A])(f: (A) ⇒ G[B])(implicit G: Applicative[G]): G[Unit]
```

Traverse `F[A]` using `Applicative[G]`.

A values will be mapped into `G[B]` and combined using `Applicative#map2`.

For example:

```
scala> import cats.implicits._
scala> def parseInt(s: String): Option[Int] = Either.catchOnly[NumberFormatException](s.toInt).toOption
scala> val F = Foldable[List]
scala> F.traverse_(List("333", "444"))(parseInt)
res0: Option[Unit] = Some(())
scala> F.traverse_(List("333", "zzz"))(parseInt)
res1: Option[Unit] = None
```

This method is primarily useful when `G[_]` represents an action or effect, and the specific `A` aspect of `G[A]` is not otherwise needed.

But we prefer to take advantage of **syntax** (**syntactic sugar**), which allows us instead to just say **x.sequence_**.

```
scala> val x: IO[Unit] = List(
  IO{ print("a") }, IO{ print("b") }, IO{ print("c") }
).sequence_
scala> x.unsafeRunSync
abc
scala>
```

cats.syntax

NestedFoldableOps

```
final class NestedFoldableOps[F[_], G[_], A](private val fga: F[G[A]]) extends AnyVal {
  def sequence_(implicit F: Foldable[F], G: Applicative[G]): G[Unit] = F.sequence_(fga)
…
```

We can exploit the fact that **IO** is an **Applicative**, by using Applicative's **right shark** function (AKA **right bird**) to simplify the following two functions

@philip_schwarz

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

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

In the above **for comprehensions**, we are forced to assign the result of an **effectful** function call to a **val**, even if we are not interested in the result, which in this case is **IO**[**Unit**]. Although this is made slightly more palatable by choosing _ as the **val**, it is still annoying.

By using **Applicative**'s **right shark** function ***>**, we can get rid of the **for comprehensions** altogether.

cats.effect IO

```
def *>[B](another: IO[B]): IO[B]
```

Runs the current IO, then runs the parameter, keeping its result. The result of the first action is ignored. If the source fails, the other action won't run.

Let's do it

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

simplify

```
def writeAt(p: Pos, s: String): IO[Unit] =
  goto(p) *> putStr(s)
```

If you are new to **Applicative**, then the next four slides provide a quick introduction to ***>**. Feel free to skip them if you are on familiar ground. If you could do with an in-depth introduction to **Applicative**, then the following slide decks may be of interest:

slideshare

https://www.slideshare.net/pjschwarz/applicative-functor-116035644
https://www.slideshare.net/pjschwarz/applicative-functor-part-2
https://www.slideshare.net/pjschwarz/applicative-functor-part-3

Applicative Functor

learn how to use an Applicative Functor to handle multiple independent effectful values through the work of


Sergei Winitzki
sergei-winitzki-11a6431


Runar Bjarnason
@runarorama


Paul Chiusano
@pchiusano


Debasish Ghosh
@debasishg


Adelbert Chang
@adelbertchang

slides by @philip_schwarz

Applicative Functor

Part 2

Learn more about the canonical definition of the Applicative typeclass by looking at a great Haskell validation example by Chris Martin and Julie Moronuki
Then see it translated to Scala


@chris_martin @argumatronic

slides by @philip_schwarz

Applicative Functor

Part 3

learn how to use Applicative Functors with Scalaz through the work of


Sam Halliday
@fommil


Debasish Ghosh
@debasishg


John A De Goes
@jdegoes


Julie Moronuki
Chris Martin
@chris_martin
@argumatronic

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

To explain *>, I am first going to explain <*> (AKA Tie Fighter, or angry parent). Consider a function that squares its parameter:

```
def square: Int => Int = n => n * n
```

This function can only be applied to an Int. What if we want to apply it to an effectful value, e.g. an Option[Int], a number that may or may not be available?

Option is a Functor:

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

Notice how the signature of the above map function differs slightly from the following, more customary signature:

```
def map[A,B](ma: F[A])(f: A => B): F[B]
```

We have simply swapped the first and second parameters of the above map function and then pushed the second parameter into the return type. The map function with the less customary signature lifts any function into context F, i.e. it is a function which, given any function f, returns a new function which takes an A wrapped in an effect F, gains access, if possible, to the wrapped a of type A, computes f(a), and returns the resulting B wrapped in F.

So if we get a Functor instance for Option

```
implicit val optionFunctor = new Functor[Option] {
  def map[A,B](f: A => B): Option[A] => Option[B] =
    oa => oa map f
}
```

We can now use map to apply square to an Option[Int] value

```
import optionFunctor.map

assert( map(square)(Option(3)) == Option(9))
assert( map(square)(None)      == None)
```

What if the function we are interested in does not take just one parameter, but many? e.g. if we have a function that takes three integers and returns the largest?

```
val max3: Int => Int => Int => Int = x => y => z => Math.max(x, Math.max(y, z))
```

What if we want to apply the function to **effectful values**, e.g. to **Option**[**Int**] values, numbers that may or may not be available? **Option** can be modeled as an **Applicative**:

```
sealed trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: => A): F[A]
  def ap[A,B](fab: F[A => B]): F[A] => F[B]
}
```

Just as I did for the **map** function in **Functor**, I have rearranged the signature of **ap** (AKA **apply**) so that it returns a function `F[A] => F[B]`, just as it does in **Haskell**.

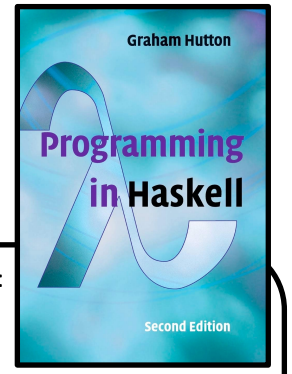
While **Functor**'s **map** function allows us to lift any one-arg function into a context F, the combination of **Applicative**'s **pure** and **ap** functions allows us to lift into a context F any function with multiple arguments. If we get an **Applicative** instance for **Option**

```
implicit val maybeApplicative = new Applicative[Option] {
  def pure[A](a: => A): Some(a)
  def ap[A,B](of: Option[A => B]): Option[A] => Option[B] = oa =>
    (of, oa) match {
      case (None, _) => None
      case (_, None) => None
      case (Some(f), Some(a)) => Some(f(a))
    }
  def map[A,B](f: A => B): Option[A] => Option[B] =
    a => ap(Some(f))(a)
}
```

```
(<*>) :: f (a -> b) -> f a -> f b
```
Sequential application.

and we define some syntax allowing us to call **ap** as infix operator **<*>**

```
implicit class ApplicativeOps[A,B,F[ ]](of: F[A => B]) {
  def <*>(oa: F[A])(implicit AP: Applicative[F]) = AP.ap(of)(oa)
}
```



A typical use of **pure** and **<*>** has the following form:

pure g **<*>** x_1 **<*>** x_2 **<*>** ... **<*>** x_n

Such expressions are said to be in **applicative style**, because of the similarity to normal function application notation **g** $\mathbf{x_1}$ $\mathbf{x_2}$ ... $\mathbf{x_n}$.

In both cases, **g** is a **curried** function that takes **n** arguments of type $\mathbf{a_1}$ ... $\mathbf{a_n}$ and produces a result of type **b**. However, in **applicative style**, each argument $\mathbf{x_i}$ has type **f** $\mathbf{a_i}$ rather than just $\mathbf{a_i}$, and the overall result has type **f b** rather than **b**.

then we can use **pure** and **<*>** to apply **max3** to **Option**[**Int**] values:

```
assert( pure(max3) <*> Option(3) <*> Option(2) <*> Option(4) == Option(4) )
```

Like every **monad**, the **IO monad** is also an **Applicative**, so let's see how we can use the **pure** and **<*>** functions provided by the **Cats Effect IO monad** to apply the **max3** function to values in an **IO context**.

Let's define a function to read an **Int** from the console:

```
def getInt: IO[Int] = IO { scala.io.StdIn.readInt() }
```

```
val max3: Int => Int => Int => Int = x => y => z =>
  Math.max(x, Math.max(y, z))
```

We can use **pure** and **<*>** to apply **max3** to three **Int** values read from the console:

```
val maximum: IO[Int] = IO.pure(max3) <*> getInt <*> getInt <*> getInt
```

```
(<*>) :: f (a -> b) -> f a -> f b
Sequential application.
```

If we now run the **maximum IO action**:

```
println(s"the maximum is ${maximum.unsafeRunSync}")
```

We can enter three numbers at the console and watch how the maximum of the three numbers gets announced on the console:

```
3
2
4
the maximum is 4
```

Now consider a scenario in which rather than reading a number of integers from the console, we are writing them to the console. So we have **putStr**

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

We call **putStr** a number of times, and each time we do that, we don't care about the **IO[Unit]** value that it returns. This is where Applicative's ***>** function comes in handy:

```
scala> val putStrings = putStr("a") *> putStr("b") *> putStr("c")
putStrings: cats.effect.IO[Unit] = IO$591439568
scala> putStrings.unsafeRunSync
abc
scala>
```

```
(*>) :: f a -> f b -> f b
Sequence actions, discarding the value of the first argument.
```

The reason why I explained <*> before explaining *> is that once you understand <*>, it is very easy to grasp what *> does and why it might be useful, especially in an IO context.

@philip_schwarz

```
(<*>) :: f (a -> b) -> f a -> f b
```
Sequential application.

```
(*>) :: f a -> f b -> f b
```
Sequence actions, discarding the value of the first argument.

```
IO.pure(max3) <*> getInt <*> getInt <*> getInt
```

<*> consumes **effectful expressions** wrapping values that we are interested in because we want to feed the values to **max3**

```
putStr("a") *> putStr("b") *> putStr("c")
```

*> consumes **effectful expressions** wrapping values that we are not interested in because it is the **side effects** produced by the expressions that are of value to us

We have already used *> to simplify the **writeAt IO action**

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

simplify

```
def writeAt(p: Pos, s: String): IO[Unit] =
  goto(p) *> putStr(s)
```

On the next slide, we do the same for the **life** function

```scala
def life(b: Board): IO[Unit] =
  for {
    _ <- cls
    _ <- showCells(b)
    _ <- goto(width+1,height+1)
    _ <- wait(1_000_000)
    _ <- life(nextgen(b))
  } yield ()
```

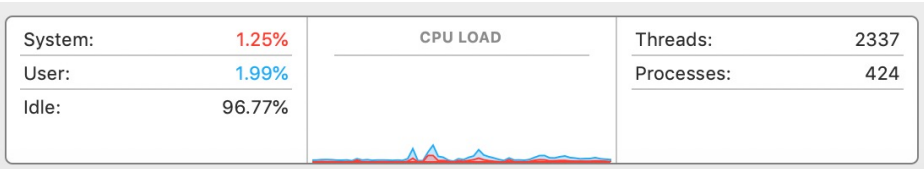
simplify

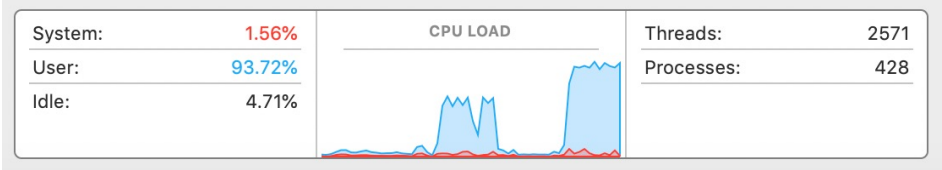
```scala
def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) *>
  life(nextgen(b))
```

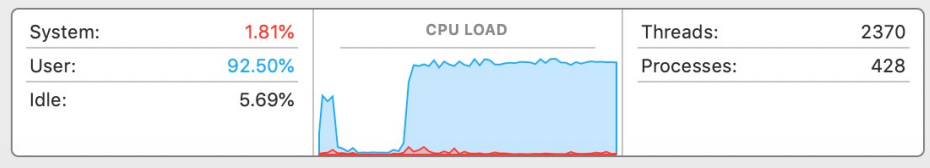
Unfortunately, if we do that, the **Game of Life** program stops displaying anything and the CPU load shoots up.

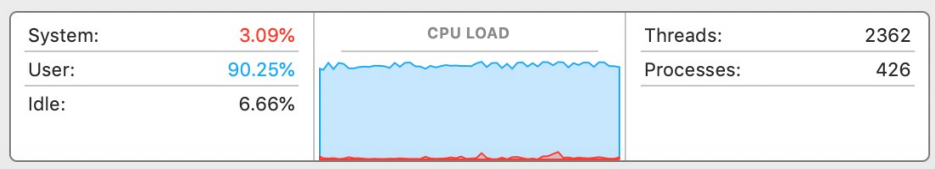
Until I stop the program, at which point the CPU load goes gradually back to normal.

CPU load

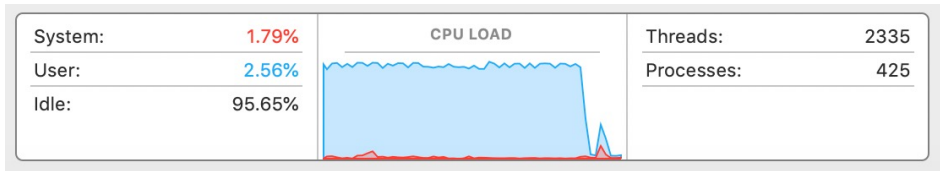


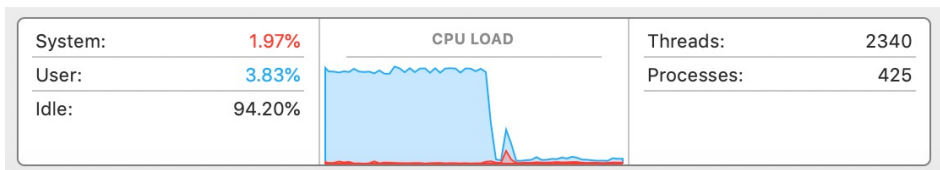


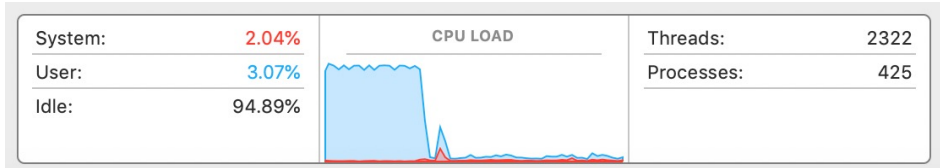


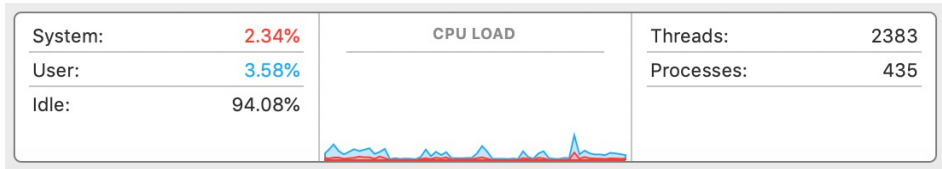


CPU load









I asked for help in https://gitter.im/typelevel/cats and **Fabio Labella** immediately identified the issue.

The problem is that *> is eager in its single parameter, i.e. its argument is passed in **by-value**, so it gets evaluated before *> executes, therefore when *> is passed `life(nextgen(b))`, which is an **infinite computation** (since the **life** function calls itself **recursively**), the body of *> never gets to run!

```
/**
 * Runs the current IO, then runs the parameter, keeping its result.
 * The result of the first action is ignored.
 * If the source fails, the other action won't run.
 * */
def *>[B](another: IO[B]): IO[B] = ...
```

```
def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) *>  this invocation of *> never gets to run
  life(nextgen(b))    since this expression never terminates
```

Fabio also pointed out that there is an **alternative operator** called >> that can be used to avoid the problem, because its single parameter is passed in **by-name** rather than **by-value**.

@philip_schwarz

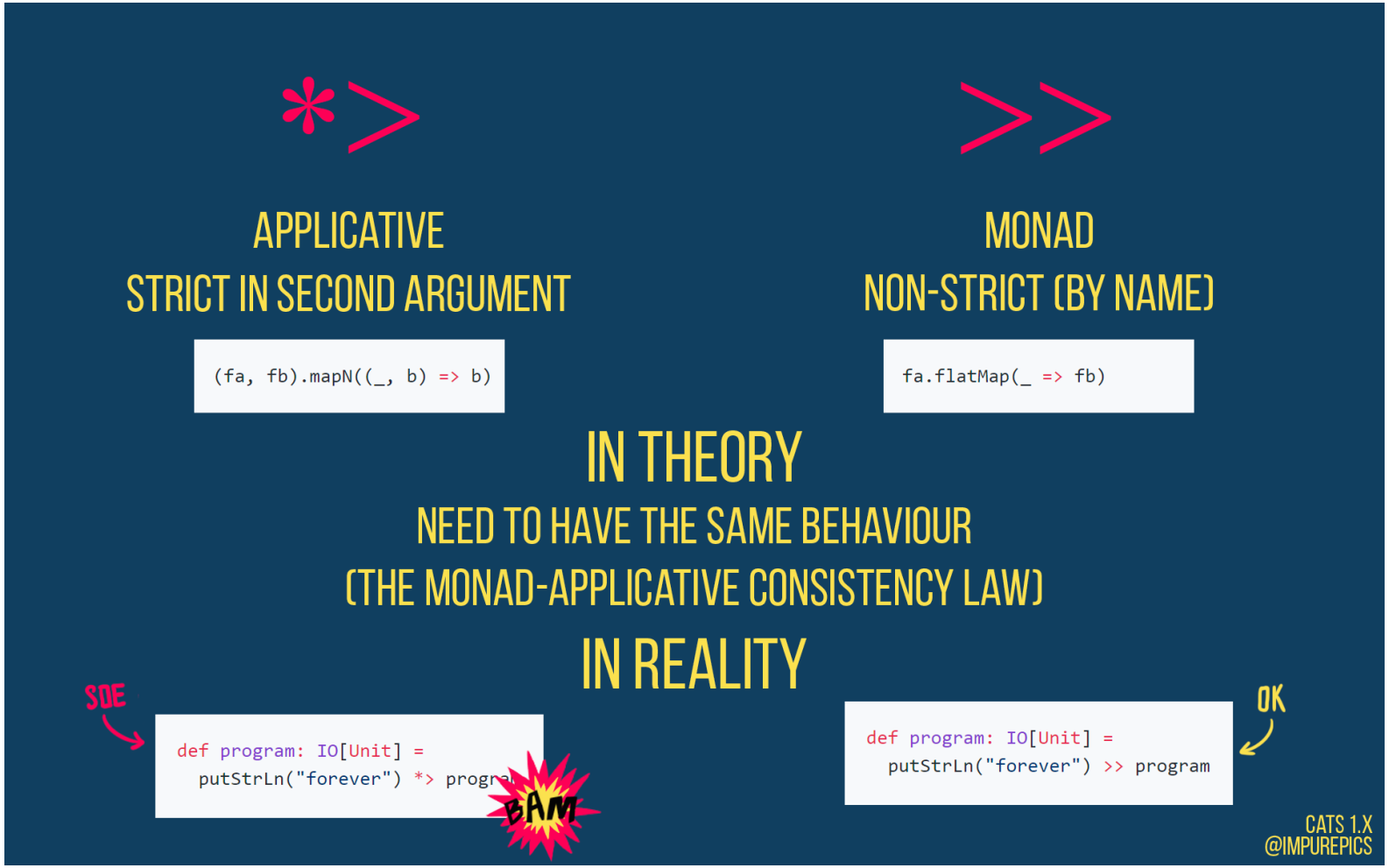
So all we have to do is replace the last usage of *> in the **life** function with >> and the **Game of Life** program goes back to running normally.

```
/**
 * Alias for `fa.flatMap(_ => fb)`.
 *
 * Unlike `*>`, `fb` is defined as a by-name parameter, allowing this
 * method to be used in cases where computing `fb` is not stack safe
 * unless suspended in a `flatMap`.
 */
def >>[B](fb: => F[B])(implicit F: FlatMap[F]): F[B] = ...
```

```
def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) >>
  life(nextgen(b))
```

As I was putting the finishing touches on this slide deck I realised that **@impurepics** has a diagram on this potential problem with ***>**

https://impurepics.com/posts/2019-02-09-operator-wars-reality.html



By the way, another point that **Fabio Labella** made, this time about the original version of the **recursive life** function, the one using the **for comprehension**, is that there is a potential memory problem if **better-monadic-for** is not used, since those `yield ()` get transformed into a massive chain of **map** calls.

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

https://github.com/oleg-py/better-monadic-for

A **Scala** compiler plugin to give patterns and **for-comprehensions** the love they deserve

Final map optimization

Eliminate calls to `.map` in comprehensions like this:

```
for {
  x <- xs
  y <- getYs(x)
} yield y
```

Standard desugaring is

```
xs.flatMap(x => getYs(x).map(y => y))
```

This plugin simplifies it to

```
xs.flatMap(x => getYs(x))
```

To conclude this slide deck, the next two slides show the final **Scala** code for the **Game of Life**.

FUNCTIONAL CORE

```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
```

```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)])
```

IMPERATIVE SHELL

```scala
import cats.implicits._, cats.effect.IO

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] =
  goto(p) *> putStr(s)

def showCells(b: Board): IO[Unit] =
  ( for { p <- b } yield writeAt(p, "O") ).sequence_

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

def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) >>
  life(nextgen(b))

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

main.unsafeRunSync
```

FPiS simple IO monad replaced by Cats Effect 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)
}
```

```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)
  …
}
```

That’s it for Part 3.

Translation of the **Game of Life** into **Unison** slips to part 4.

See you there!