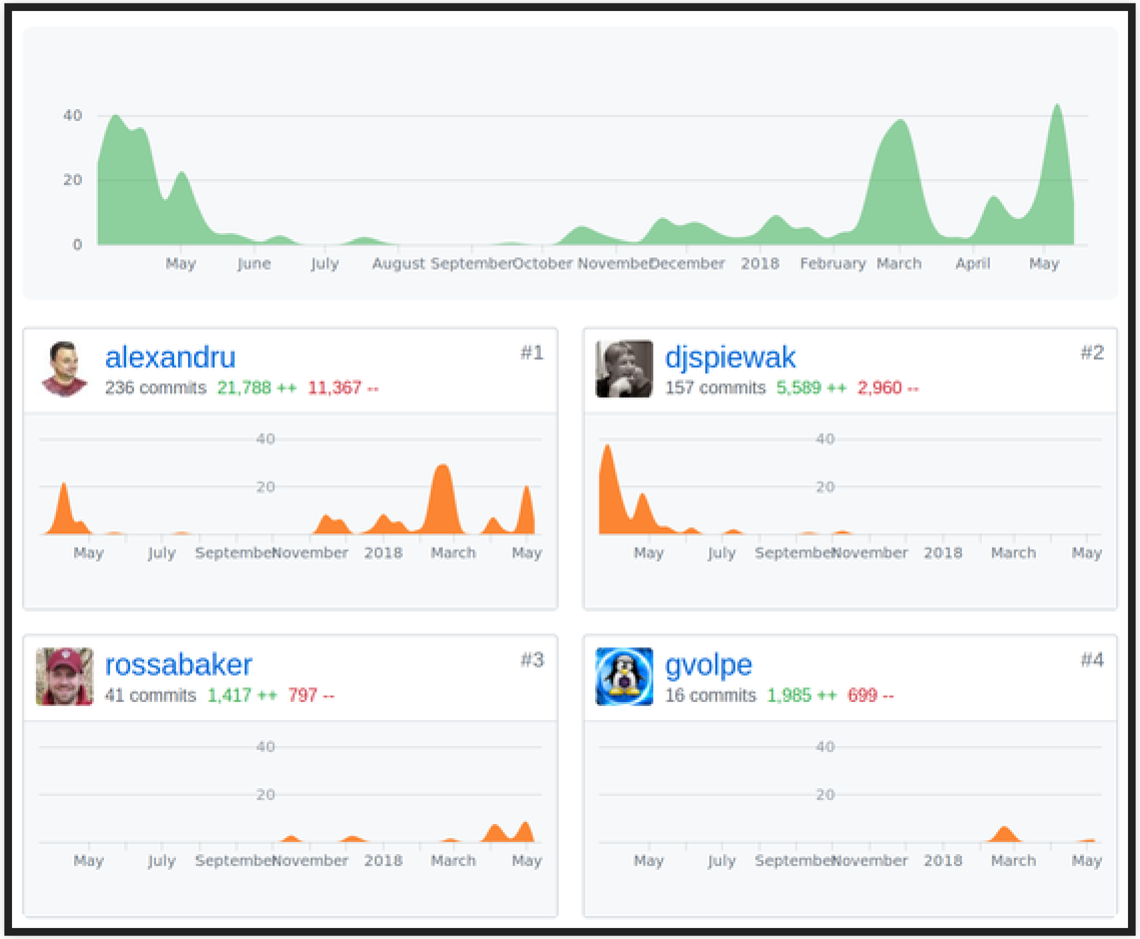
# CATS EFFECT

## THE IO MONAD FOR SCALA

# GABRIEL VOLPE

- Software Engineer at Paidy
- Functional Programing enthusiastic
- Co-organizer of Scala Tokyo Meetup
- Open Source Contributor
  - https://gvolpe.github.io/

# CATS EFFECT CONTRIBUTOR



alexandru #1
236 commits 21,788 ++ 11,367 --

djspiewak #2
157 commits 5,589 ++ 2,960 --

rossabaker #3
41 commits 1,417 ++ 797 --

gvolpe #4
16 commits 1,985 ++ 699 --

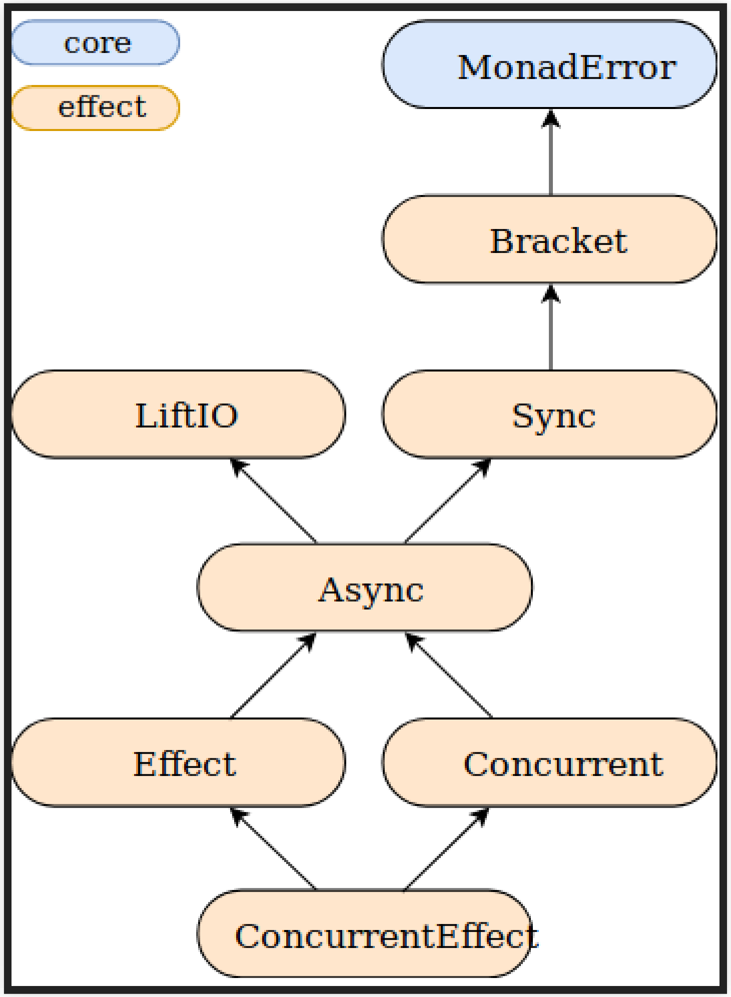
And other libraries such as Fs2, Http4s, Circe, Fs2 Rabbit

# AGENDA

- What is Cats Effect?
- Effects vs Side Effects
- Sync vs Async
- Error Handling
- Resource Safety
- Concurrency
- Parallelism
- Cancellation

# WHAT IS CATS EFFECT?

It provides a hierarchy of typeclasses to describe and compose effects



# WHAT IS CATS EFFECT?

It also provides a set of datatypes:

- IO
- Fiber
- Resource
- Timer

And concurrent primitives:

- Deferred
- MVar
- Ref
- Semaphore

# WHAT IS CATS EFFECT?

Latest version `1.0.0-RC2`

Stable version `1.0.0` coming soon!

# EFFECTS VS SIDE EFFECTS

- What are effects?

# EFFECTS VS SIDE EFFECTS

## EFFECTS

- **Option[A]**: May or may not produce a value A.
- **Either[A, B]**: Either produces a value A or a value B.
- **List[A]**: Produces Zero, One or Many elements of type A.
- **IO[A]**: Produces a value A, fails or never terminate.

# EFFECTS VS SIDE EFFECTS

## SIDE EFFECTS

- **println("Hey!")**: Writes to the console immediately.
- **scala.io.StdIn.readLine()**: Reads from the console immediately.
- **System.nanoTime()**: Retrieves current time from the VM immediately.
- **Future(deleteDB)**: Deletes database immediately.

# REFERENTIAL TRANSPARENCY

## ARE THESE TWO PROGRAMS THE SAME?

```
val expr = 123
(expr, expr)
```

```
(123, 123)
```

# REFERENTIAL TRANSPARENCY

## ARE THESE TWO PROGRAMS THE SAME?

```
val expr = println("Hey!")
(expr, expr)
```

```
(println("Hey!"), println("Hey!"))
```

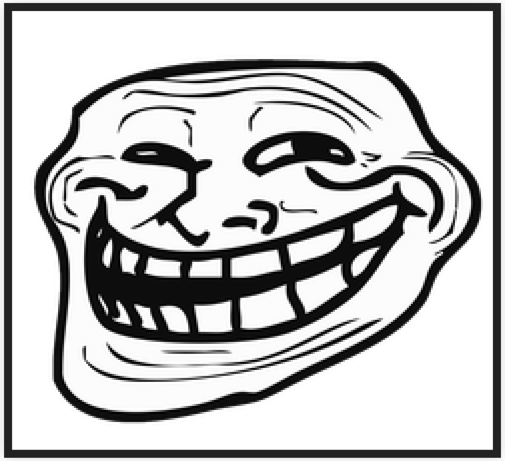
# REFERENTIAL TRANSPARENCY

## ARE THESE TWO PROGRAMS THE SAME?

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
```

```scala
val expr = Future(println("Hey!"))
(expr, expr)
```

```scala
(Future(println("Hey!")), Future(println("Hey!")))
```



# REFERENTIAL TRANSPARENCY

## ARE THESE TWO PROGRAMS THE SAME?

```scala
import cats.effect.IO
```

```scala
val expr = IO(println("Hey!"))
(expr, expr)
```

```scala
(IO(println("Hey!")), IO(println("Hey!")))
```

# REFERENTIAL TRANSPARENCY

```
IO[A]
```

Represents the intention to perform a side effect

# SYNCHRONOUS COMPUTATIONS

## DELAY

```
def delay[A](thunk: => A): F[A] = suspend(pure(thunk))
```

```
import cats.effect.{IO, Sync}

Sync[IO].delay(println("Hey!")) <-> IO(println("Hey!"))
```

## SUSPEND

```
def suspend[A](thunk: => F[A]): F[A]
```

```
val expr = IO(loop)
Sync[IO].suspend(expr) <-> IO.suspend(expr)
```

# ASYNCHRONOUS COMPUTATIONS

## ASYNC

```
def async[A](k: (Either[Throwable, A] => Unit) => Unit): F[A]
```

```
Async[IO].async <-> IO.async
```

## NEVER

```
def never[A]: F[A] = async(_ => ())
```

# ASYNCHRONOUS COMPUTATIONS

```
val iof = IO(myFuture)

val fromFuture: IO[Unit] =
  iof.flatMap { f =>
    IO.async[Unit] { cb =>
      f.onComplete{
        case Success(a) => cb(Right(a))
        case Failure(e) => cb(Left(e))
      }
    }
  }
```

```
val f = IO.fromFuture(iof)
```

# ERROR HANDLING: MONAD ERROR

```
def attempt[A](fa: F[A]): F[Either[E, A]]
def rethrow[A](fa: F[Either[E, A]]): F[A]
def handleErrorWith[A](fa: F[A])(f: E => F[A]): F[A]
def recoverWith[A](fa: F[A])(pf: PartialFunction[E, F[A]]): F[A]
```

```
MonadError[IO, Throwable].raiseError <-> IO.raiseError
MonadError[IO, Throwable].attempt <-> IO.attempt
```

## RAISE AND ATTEMPT

```
val boom: IO[String] = IO.raiseError[String](new Exception("boom"))
val safe: IO[Either[Throwable, String]] = boom.attempt
```

## HANDLING ERRORS

```
val keepGoing: IO[String] = boom.handleErrorWith {
  case NonFatal(e) => IO(println(e.getMessage)) *> IO.pure("Keep going ;)")
}
```

# BRACKET

```
def bracket[A, B](acquire: F[A])(use: A => F[B])
  (release: A => F[Unit]): F[B]
```

## SAFE RESOURCE ACQUISITION

```
val acquireResource: IO[FileOutputStream] =
  IO { new FileOutputStream("test.txt") }

val useResource: FileOutputStream => IO[Unit] =
  fos => IO { fos.write("test data".getBytes()) }

val releaseResource: FileOutputStream => IO[Unit] =
  fos => IO { fos.close() }

acquireResource.bracket(useResource)(releaseResource)
```

# BRACKET

## CAVEATS

Nested resources get messy very quick

```
def putStrLn(str: String): IO[Unit] = IO(println(str))

def acquire(s: String) = putStrLn(s"Acquiring $s") *> IO.pure(s)
def release(s: String) = putStrLn(s"Releasing $s")

acquire("one").bracket { r1 =>
  putStrLn(s"Using $r1") *> acquire("two").bracket { r2 =>
    putStrLn(s"Using $r2") *> acquire("three").bracket { r3 =>
      putStrLn(s"Using $r3")
    } { r3 => release(r3) }
  } { r2 => release(r2) }
} { r1 => release(r1)}
```

# RESOURCE

- Nested resources are released in reverse order of acquisition.
- Outer resources are released even if an inner use or release fails.

```
def allocate: F[(A, F[Unit])]
def use[B, E](f: A => F[B])(implicit F: Bracket[F, E]): F[B] =
  F.bracket(allocate)(a => f(a._1))(_._2)
```

## NESTED RESOURCE ACQUISITION

```
def mkResource(s: String): Resource[IO, String] = {
  val acquire = IO(println(s"Acquiring $s")) *> IO.pure(s)
  def release(s: String) = IO(println(s"Releasing $s"))
  Resource.make[IO, String](acquire)(release)
}

val r = for {
  outer <- mkResource("outer")
  inner <- mkResource("inner")
} yield (outer, inner)

r.use { case (a, b) => IO(println(s"Using $a and $b")) }
```

# CONCURRENCY

## START

```
def start[A](fa: F[A]): F[Fiber[F, A]
```

```
trait Fiber[F[_], A] {
  def cancel: F[Unit]
  def join: F[A]
}
```

## NON-DETERMINISTIC / CONCURRENT EXECUTION

```
for {
  fb1 <- ioa.start
  fb2 <- iob.start
  _   <- fb2.cancel
  rs  <- fb1.join
} yield rs
```

# CONCURRENCY

```scala
def race[A, B](fa: F[A], fb: F[B]): F[Either[A, B]]
def racePair[A,B](fa: F[A], fb: F[B]): F[Either[(A, Fiber[F, B]), (Fiber[F, A], B)]]
```

```scala
Concurrent[IO].race <-> IO.race
```

## CAN YOU GUESS THE RESULT?

```scala
val ioa = IO.sleep(1.second) *> IO(println("A"))
val iob = IO(println("B"))

IO.race(ioa, iob)
IO.racePair(ioa, iob)
```

# CAN YOU GUESS THE RESULT?

```
val ioa = IO.sleep(1.second) *> IO(println("A"))
val iob = IO(println("B"))

IO.race(ioa, iob)       // winner is always B, A gets canceled
IO.racePair(ioa, iob)   // gets a Fiber to join / cancel the loser
```

# CONCURRENCY

## TIMEOUT

```scala
case class TimeOutException(message: String) extends Exception(message)

def timeout[A](ioa: IO[A], after: FiniteDuration): IO[A] = {
  IO.race(ioa, IO.sleep(after)).flatMap {
    case Left(x)  => IO.pure(x)
    case Right(_) => IO.raiseError(TimeOutException(s"Timeout after $after"))
  }
}

val delayedIO = IO.sleep(3.seconds) *> IO(println("delayed io!"))

timeout(delayedIO, 1.second) // TimeOutException: Timeout after 1 second!
```

```scala
def timeout(duration: FiniteDuration)(implicit timer: Timer[IO]): IO[A]
```

# PARALLELISM

```
trait Parallel[M[_], F[_]] {
  def apply: Apply[F]
  def flatMap: FlatMap[M]
  def sequential: F ~> M
  def parallel: M ~> F
}
```

## PAR MAP N

```
val pio1 = IO(println("started pio1")) *> IO.sleep(1.second) *> IO.pure("P1")
val pio2 = IO(println("started pio2")) *> IO.pure("P2")

(pio1, pio2).parMapN { case (a, b) => IO(println(s"$a and $b"))}.flatten
```

# PARALLELISM

## PAR TRAVERSE

```
List(pio1, pio2).parTraverse(io => io *> putStrLn("Traversing"))
```

## PAR SEQUENCE

```
List(pio1, pio2).parSequence // IO(List(P1, P2))
```

# CANCELLATION

```
trait Concurrent[F[_]] extends Async[F] {
  def cancelable[A](k: (Either[Throwable, A] => Unit) => IO[Unit]): F[A]
}
```

```
trait Bracket[F[_], E] extends MonadError[F, E] {
  def uncancelable[A](fa: F[A]): F[A]
}
```

## CANCELABLE

```
val sc: ScheduledExecutorService = Executors.newScheduledThreadPool(2)

def cioSleep(d: FiniteDuration) = IO.cancelable[Unit] { cb =>
  val runnable: Runnable = () => { cb(Right(())) }
  val task = sc.schedule(runnable, d.length, d.unit)

  putStrLn("Triggering cancellation") *> IO(task.cancel(false))
}

val sleep3s      = cioSleep(3.seconds) *> putStrLn("not today")
val cancelableIO = sleep3s.runCancelable(_ => putStrLn("done"))

cancelableIO.flatMap(token => IO.sleep(1.second) *> token)
```

# CANCELLATION

## UNCANCELABLE

```
val nope = IO.sleep(10.seconds).uncancelable.runCancelable(_ => IO.unit)
nope.flatMap(token => IO.sleep(1.second) *> token)
```

## ON CANCEL RAISE ERROR

```
val fa = putStrLn("infinite") *> IO.never

fa.onCancelRaiseError(new Exception("Process Cancelled!"))
  .handleErrorWith(e => putStrLn(e.getMessage))
  .runCancelable(_ => putStrLn("done"))
  .flatMap(token => IO.sleep(2.seconds) *> token *> IO.sleep(100.millis))
```

```
IO.onCancelRaiseError <-> acquire.bracketCase(use) {
                            case (_, ExitCase.Canceled) => IO.raiseError
                          }
```

# CANCELLATION

## CANCEL BOUNDARY

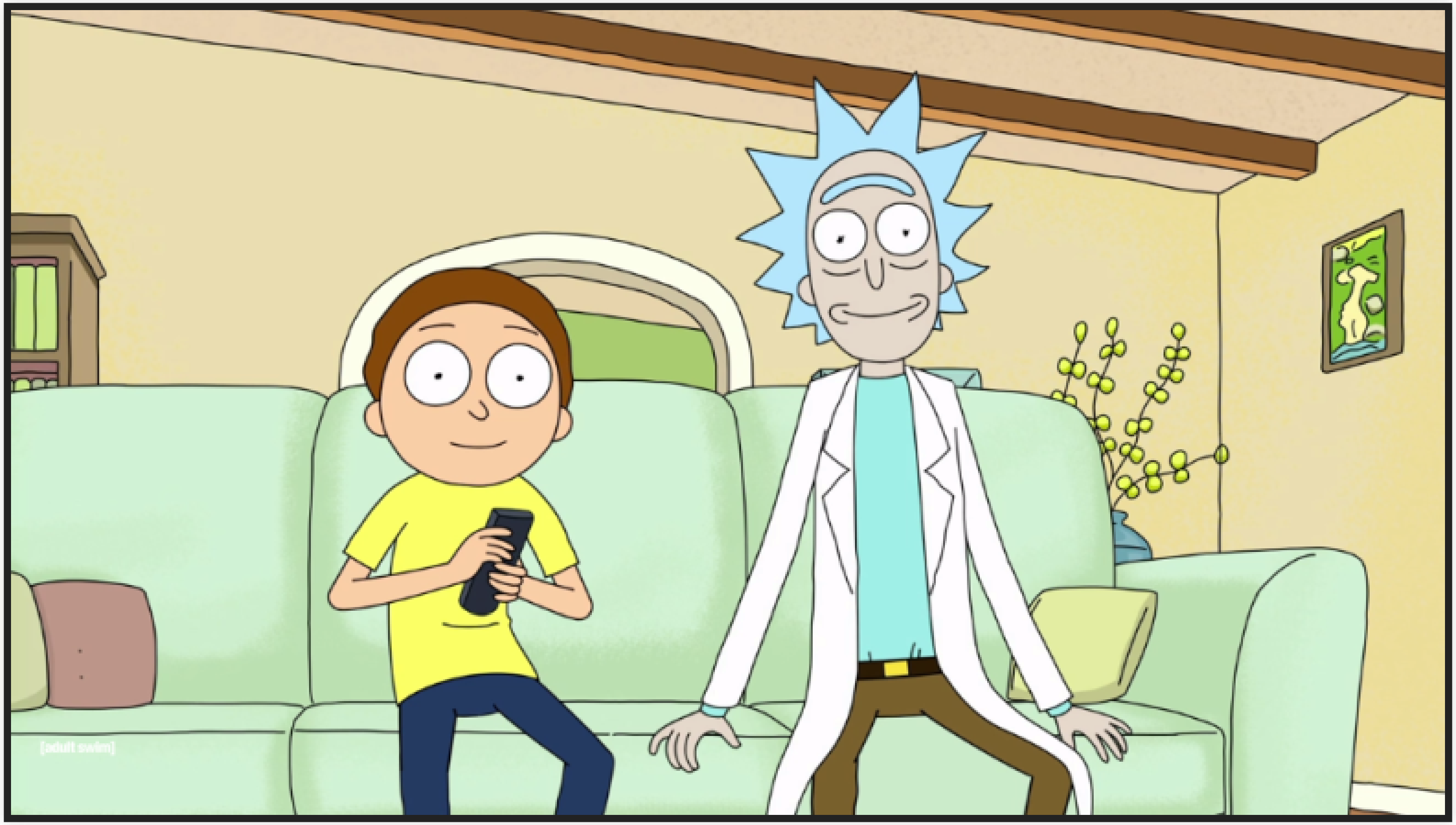
```scala
def cancelableLoop(acc: Int): IO[Unit] =
  IO.suspend {
    val next  = cancelableLoop(acc + 1)
    val sleep = IO.shift *> IO(Thread.sleep(100)) // IO.sleep is cancelable

    if (acc % 10 == 0) {
      putStrLn(s"#$acc >> Checking cancellation status") *> IO.cancelBoundary *> next
    } else {
      putStrLn(s"#$acc") *> sleep *> next
    }
  }

val ioa = cancelableLoop(1).runCancelable(_ => putStrLn("done"))
ioa.flatMap(token => IO.sleep(1.second) *> token <* IO.sleep(1.second))
```

# CANCELLATION

## TAKEAWAYS

- **IO[IO[Unit]]** represents a task that when run it will start the evaluation of the effects giving you back a cancellation token of type **IO[Unit]**.
- **Fiber[IO, Unit]** gives you control over an effect to either cancel it or wait for its completion.
- **uncancelable** changes the nature of any `cancelable` task.
- **cancelBoundary** adds a "cancellation status check" step to any computation, useful in loops.
- **onCancelRaiseError** forces possibly non-terminating tasks to end by raising the given error.

# QUESTIONS?



paidy

カードのいらないカンタン決済

# WE ARE HIRING!

https://engineering.paidy.com/