A TALE OF TWO MONIX STREAMS

Alexandru Nedelcu

@alexelcu | alexn.org

WHAT IS MONIX?

- Scala / Scala.js library
- For composing asynchronous programs
- Exposes Observable, Task, Coeval, Iterant and many concurrency primitives
- Typelevel (see typelevel.org)
- 3.0.0-M1
- See: monix.io

3.0

- Deep integration with **Typelevel Cats**
- **Iterant** data type for lawful pull-based streaming
- major improvements to **Observable**, **Task**, **Coeval** and **CancelableFuture**

MONIFU

- Started on January 2, 2014, at 2:30 a.m.
- Developed at **Eloquentix** for monitoring and controlling power plants
- Inspired by RxJava / ReactiveX (link)
- Renamed to Monix on Dec 30, 2015 (issue #91)
- Monix comes from: *Monads* + *Rx*

MONIFU

- Started on January 2, 2014, at 2:30 a.m.
- Developed at **Eloquentix** for monitoring and controlling power plants
- Inspired by RxJava / ReactiveX (link)
- Renamed to Monix on Dec 30, 2015 (issue #91)
- Monix comes from: *Monads* + *Rx*

MONIFU

- Started on January 2, 2014, at 2:30 a.m.
- Developed at **Eloquentix** for monitoring and controlling power plants
- Inspired by RxJava / ReactiveX (link)
- Renamed to Monix on Dec 30, 2015 (issue #91)
- Monix comes from: *Monads* + *Rx*

MONIFU

- Started on January 2, 2014
- Developed at **Eloquentix** for monitoring and controlling power plants
- Inspired by RxJava / ReactiveX (link)
- Renamed to Monix on Dec 30, 2015 (issue #91)
- Monix comes from: *Monads* + *Rx*

I'M A DEVELOPER, I HAVE NO LIFE

```
commit 9c6ce3106c35de707b5d3f9d26cd63ed4b355c07
Author: Alexandru Nedelcu <noreply@alexn.org>
Date:   Thu Jan 2 02:32:49 2014 +0200

    Initial commit
(END)
```

PUSH - BASED STREAMING

OBSERVABLE[+A]

RX .NET – ORIGINS

- Reactive Extensions (also known as ReactiveX)
- The Observable pattern
- Built at Microsoft by
  - Jeffery Van Gogh
  - Wes Dyer
  - Erik Meijer
  - Bart De Smet

RX .NET – ORIGINS

```
trait Iterator[+A] {
  def hasNext: Boolean
  def next(): A
}

trait Iterable[+A] {
  def iterator: Iterator[A]
}
```

RX .NET – ORIGINS

```
type Iterator[+A] =
  () ⇒ Option[Either[Throwable, A]]

type Iterable[+A] =
  () ⇒ Iterator[A]
```

RX .NET – ORIGINS

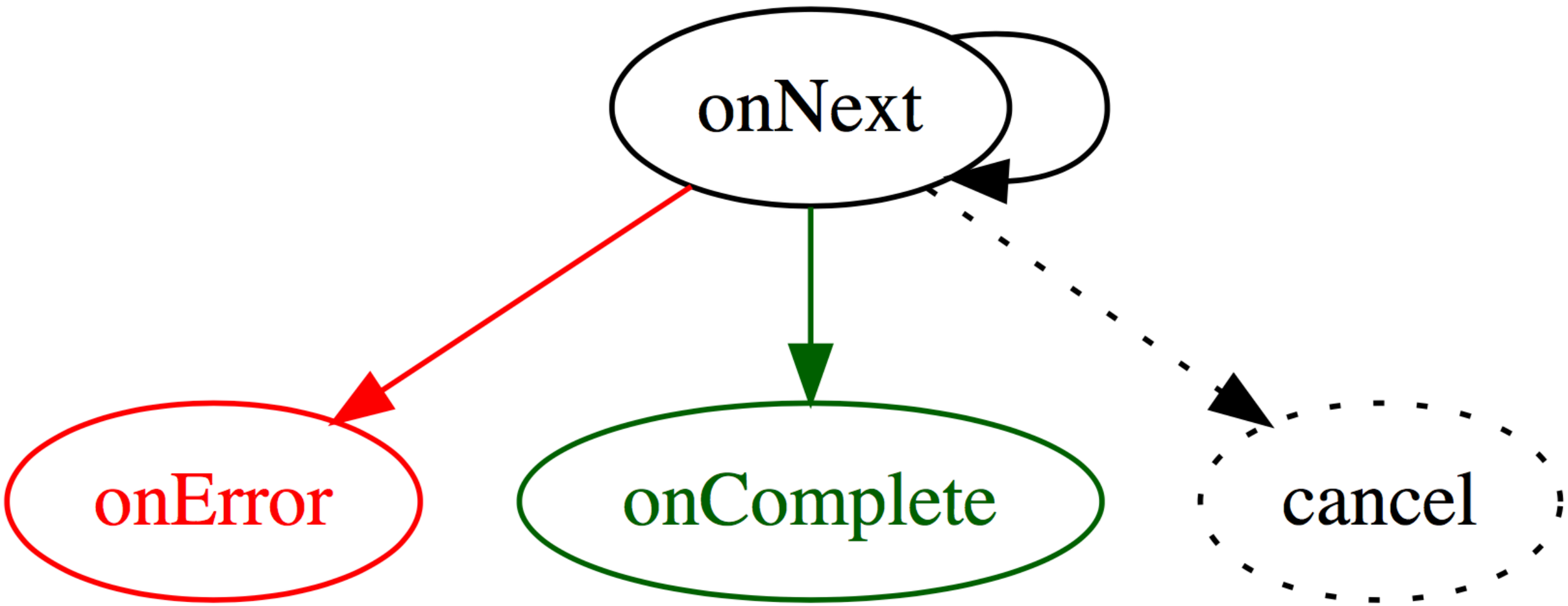
```
type Observer[-A] =
  Option[Either[Throwable, A]] ⇒ Unit

type Observable[+A] =
  Observer[A] ⇒ Unit
```

RX .NET – ORIGINS

```
trait Observable[+A] {
  def subscribe(o: Observer[A]): Cancelable
}
```

```
trait Observer[-A] {
  def onNext(elem: A): Unit
  def onComplete(): Unit
  def onError(e: Throwable): Unit
}
```

RX .NET – ORIGINS



OBSERVABLE IS THE DUAL OF ITERABLE

Erik Meijer

RX .NET – PROBLEMS

- Pure push - based
- No protections against slow consumers
- Unbounded buffers / throttling
- Trivia: **flatMap** is aliased to **mergeMap** because **concatMap** is unsafe

RX .NET – PROBLEMS

- Pure push - based
- No protections against slow consumers
- Unbounded buffers / throttling
- Trivia: **flatMap** is aliased to **mergeMap** because **concatMap** is unsafe

RX .NET – PROBLEMS

- Pure push - based
- No protections against slow consumers
- Unbounded buffers / throttling
- Trivia: **flatMap** is aliased to **mergeMap** because **concatMap** is unsafe

REACTIVE-STREAMS.ORG

```
trait Subscription {
  def request(n: Long): Unit
  def cancel(): Unit
}
```

```
trait Subscriber[A] {
  def onSubscribe(s: Subscription): Unit

  def onNext(elem: A): Unit
  def onComplete(): Unit
  def onError(e: Throwable): Unit
}
```

TOWARD THE FUTURE[A]

IDEA 1: BACK-PRESURE WITH FUTURE

```scala
import scala.concurrent.Future

trait Observer[-A] {
  def onNext(elem: A): Future[Unit]

  def onComplete(): Unit

  def onError(e: Throwable): Unit
}
```

IDEA 2: CONSUMER DRIVEN CANCELATION

```
sealed trait Ack extends Future[Ack] {
  // ...
}
```

```
object Ack {
  /** Signals demand for more. */
  case object Continue extends Ack

  /** Signals demand for early termination. */
  case object Stop extends Ack
}
```

IDEA 2: CONSUMER DRIVEN CANCELATION

```scala
import monix.execution.Ack
import scala.concurrent.Future
```

```scala
trait Observer[-A] {
  def onNext(elem: A): Future[Ack]

  def onComplete(): Unit

  def onError(e: Throwable): Unit
}
```

SIDE EFFECTS, WOOT!

```
class SumObserver(take: Int) extends Observer[Int] {
  private var count = 0
  private var sum   = 0

  def onNext(elem: Int): Ack = {
    count += 1
    sum += elem
    if (count < take) Continue else {
      onComplete()
      Stop
    }
  }

  def onComplete() =
    println(s"Sum: $sum")
  def onError(e: Throwable) =
    e.printStackTrace()
}
```

OBSERVABLE IS HIGH-LEVEL

```scala
val sum: Observable[Long] =
  Observable.range(0, 1000)
    .take(100)
    .map(_ * 2)
    .foldF

// Actual execution
sum.subscribe(result ⇒ {
  println(s"Sum: $result")
  Stop
})
```

OBSERVABLE IS HIGH-LEVEL

```
val sum: Task[Long] =
  Observable.range(0, 1000)
    .take(100)
    .map(_ * 2)
    .foldL

// Actual execution
val f: CancelableFuture[Long] =
  sum.runAsync
```

OBSERVABLE IS HIGH-LEVEL

```
val list: Observable[Long] =
  Observable.range(0, 1000)
    .take(100)
    .map(_ * 2)

val consumer: Consumer[Long, Long] =
  Consumer.foldLeft(0L)(_ + _)

val task: Task[Long] =
  list.consumeWith(consumer)
```

OBSERVABLE IS A MONADIC TYPE

```
def eventsSeq(key: Long): Observable[Long] = ???

observable.flatMap { key ⇒ eventsSeq(key) }
```

```
def someTask(key: Long): Task[Long] = ???

observable.mapTask { key ⇒ someTask(key) }

def someIO(key: Long): IO[Long] = ???

observable.mapEval { key ⇒ someIO(key) }
```

SUSPENDING SIDE EFFECTS

```
def readFile(path: String): Observable[String] =
  Observable.suspend {
    // The side effect
    val lines = Source.fromFile(path).getLines
    Observable.fromIterator(lines)
  }
```

SUSPENDING SIDE EFFECTS

- Does not need IO / Task for *evaluation* or *suspending* effects
- **Observable is IO-ish**

REACTIVE WOOT!

```
observable.mergeMap { key ⇒ eventsSeq(key) }

observable.switchMap { key ⇒ eventsSeq(key) }
```

REACTIVE WOOT!

```
observable.throttleFirst(1.second)

observable.sample(1.second)

observable.debounce(1.second)

observable.echoOnce(1.second)
```

REACTIVE WOOT!

```
observable.sampleRepeated(1.second)

observable.debounceRepeated(1.second)

observable.echoRepeated(1.second)
```

REACTIVE WOOT!

```
observable
  .distinctUntilChanged
  .sample(1.second)
  .echoRepeated(5.seconds)
```

REACTIVE WOOT!

```
observable.publishSelector { hot ⇒
  val a = hot.filter(_ % 2 == 0).map(_ * 2)
  val b = hot.filter(_ % 2 == 1).map(_ * 3)

  Observable.merge(a, b)
}
```

REACTIVE WOOT!

```
import monix.reactive.OverflowStrategy._

observable.whileBusyBuffer(
  DropNewAndSignal(1000, count ⇒ {
    logger.warn(s"$count events dropped")
    None
  }))

observable.asyncBoundary(BackPressure(1000))
```

OBSERVABLE OPTIMISATIONS FOR FLAT-MAP

- Models complex state machine for eliminating **asynchronous boundaries**
- Deals with **Concurrency** by means of one **Atomic**
  - Cache-line padding for avoiding *false sharing*
  - Uses **getAndSet** platform intrinsics
- **monix-execution** ftw

OBSERVABLE OPTIMISATIONS FOR FLAT-MAP

- Models complex state machine for eliminating **asynchronous boundaries**
- Deals with **Concurrency** by means of one **Atomic**
  - Cache-line padding for avoiding *false sharing*
  - Uses **getAndSet** platform intrinsics
- **monix-execution** ftw

OBSERVABLE OPTIMISATIONS FOR FLAT-MAP

- Models complex state machine for eliminating **asynchronous boundaries**
- Deals with **Concurrency** by means of one **Atomic**
  - Cache-line padding for avoiding *false sharing*
  - Uses **getAndSet** platform intrinsics
- **monix-execution** ftw

OBSERVABLE OPTIMISATIONS FOR MERGE-MAP / BUFFERING

- **Using JCTools.org for non-blocking queues**
- MPSC scenarios
- Consumer does not contend with producers

CONSEQUENCES

- **Best in class performance**
  (synchronous ops have ~zero overhead, can optimise synchronous pipelines)
- Referential Transparency
  (`subscribe <-> unsafePerformIO`)
- `Pure API, Dirty Internals`

CONSEQUENCES

- Best in class performance
  (synchronous ops have ~zero overhead, can optimise synchronous pipelines)
- **Referential Transparency**
  `(subscribe <-> unsafePerformIO)`
- **Pure API, Dirty Internals**

PULL-BASED STREAMING

ITERANT[F,A]

ARCHITECTURE IS FROZEN MUSIC

Johann Wolfgang Von Goethe

DATA STRUCTURES ARE FROZEN ALGORITHMS

Jon Bentley

FP DESIGN – KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*
   (Immutable)
2. Think *State Machines*
3. Be *Lazy*

FP DESIGN – KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*
2. Think *State Machines* (most of the time)
3. Be *Lazy*

FP DESIGN – KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*
2. Think *State Machines*
3. Be *Lazy*

   (Strict Values => Functions ;-))

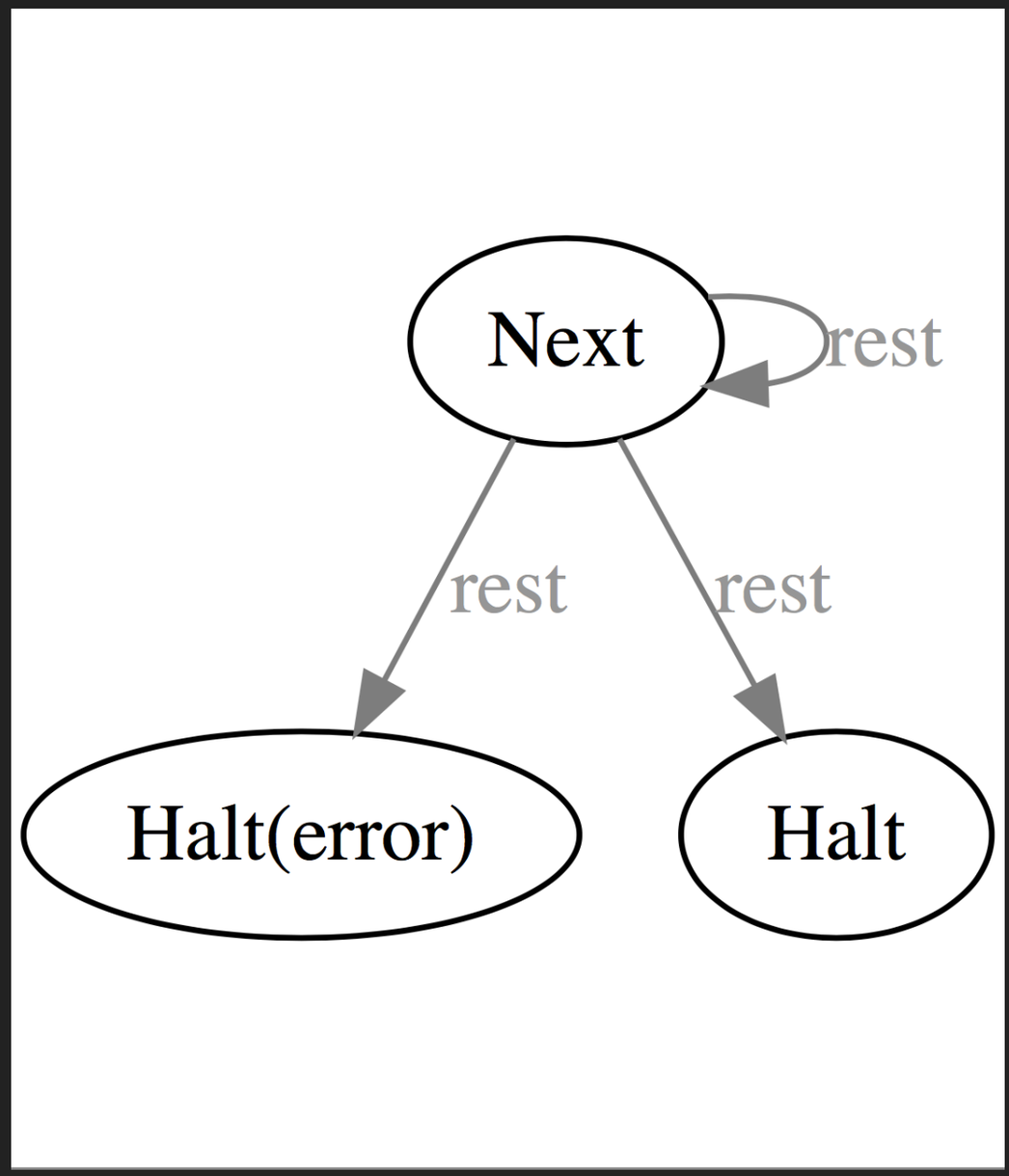

Finite State Machine Cat

LINKED LISTS

```scala
sealed trait List[+A]

case class Cons[+A](
  head: A,
  tail: List[A])
  extends List[A]

case object Nil
  extends List[Nothing]
```

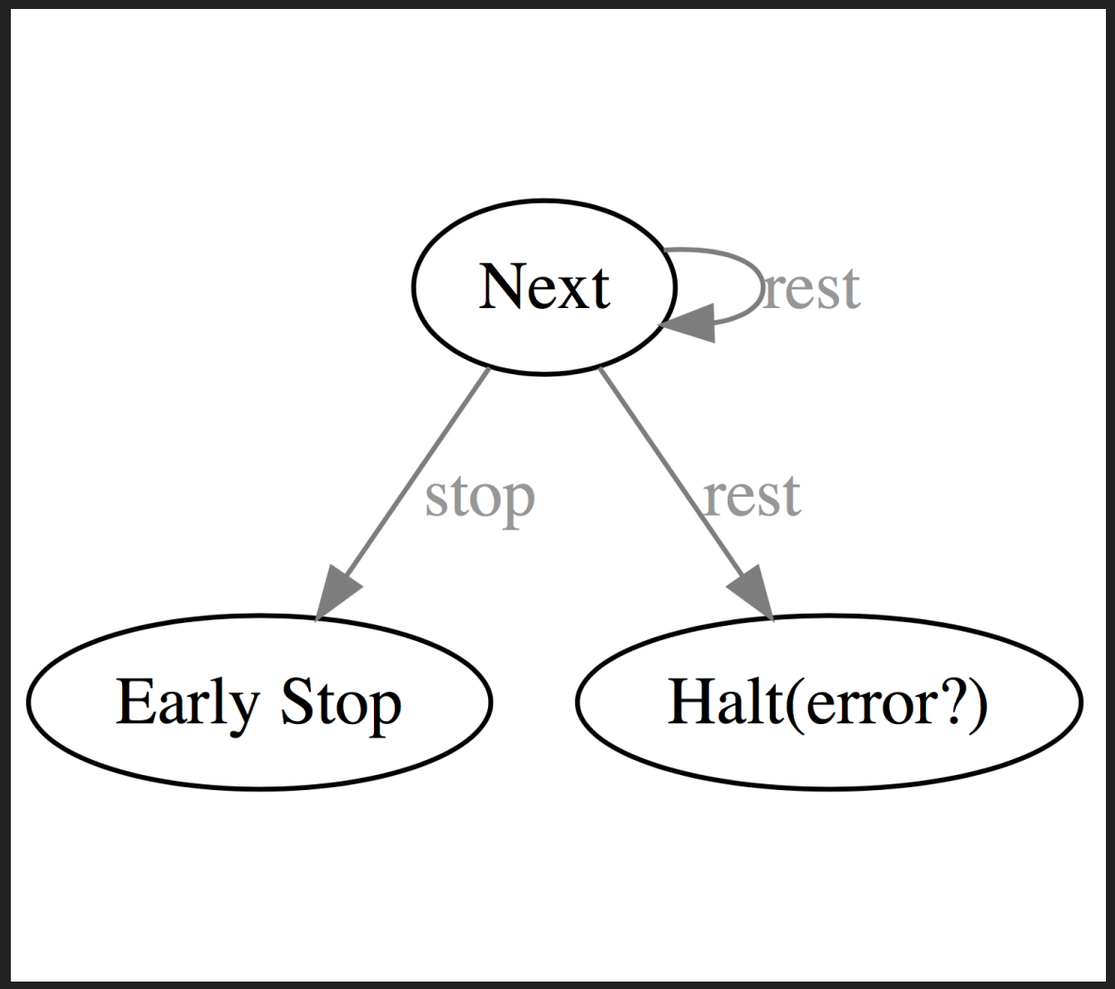
LAZY EVALUATION

```
sealed trait Iterant[A]

case class Next[A](
  item: A,
  rest: () ⇒ Iterant[A])
  extends Iterant[A]

case class Halt[A](
  e: Option[Throwable])
  extends Iterant[A]
```



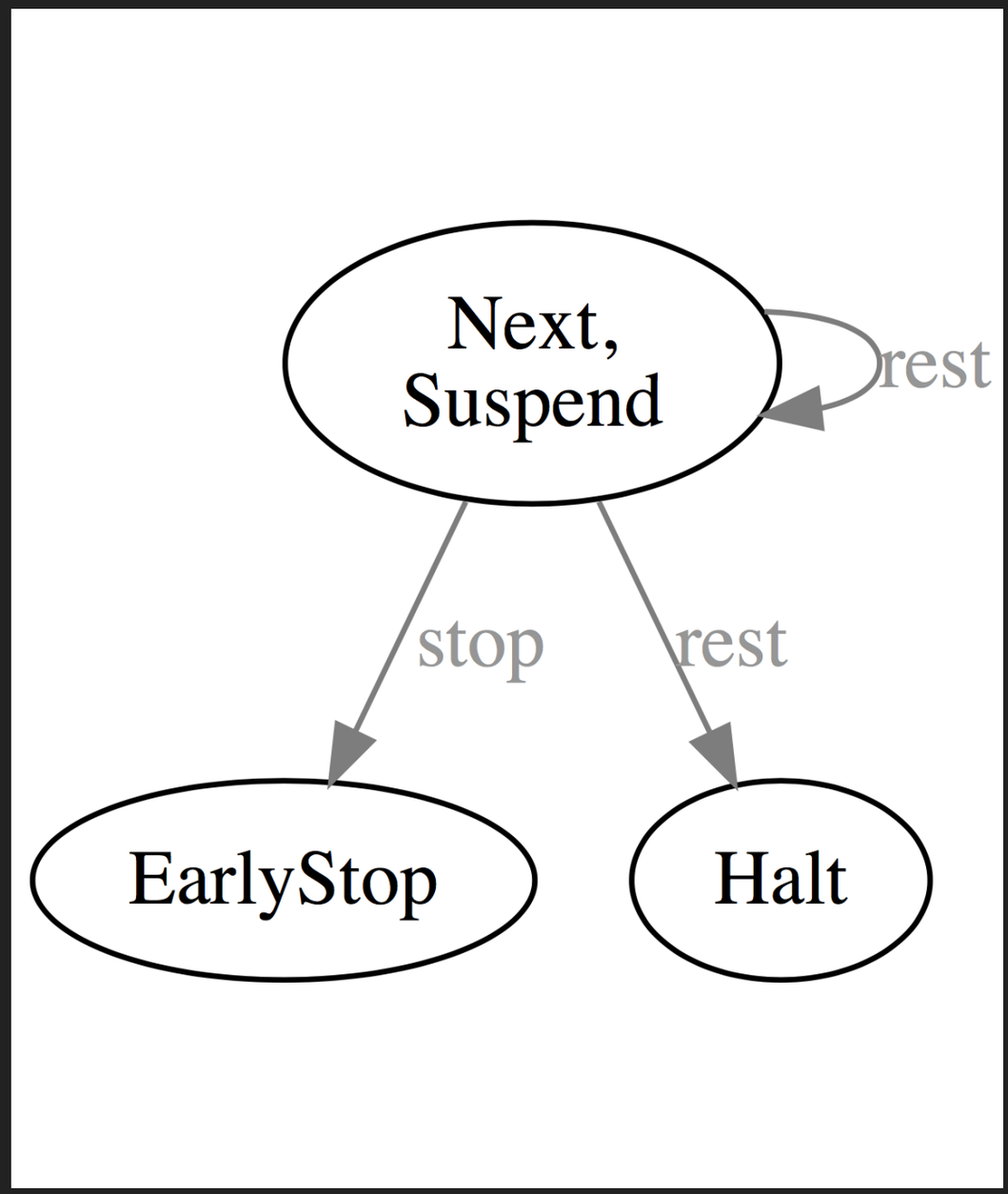
RESOURCE MANAGEMENT

```
sealed trait Iterant[A]

case class Next[A](
  item: A,
  rest: () ⇒ Iterant[A],
  stop: () ⇒ Unit)
  extends Iterant[A]

case class Halt[A](
  e: Option[Throwable])
  extends Iterant[A]
```



DEFERRING

```
sealed trait Iterant[A]

//  ...

case class Suspend[A](
  rest: () ⇒ Iterant[A],
  stop: () ⇒ Unit)
  extends Iterant[A]
```



FILTER

```scala
def filter[A](fa: Iterant[A])(p: A ⇒ Boolean): Iterant[A] =
  fa match {
    case halt @ Halt(_) ⇒ halt
    // ...
  }
```

FILTER

```scala
def filter[A](fa: Iterant[A])(p: A ⇒ Boolean): Iterant[A] =
  fa match {
  // ...
  case Suspend(rest, stop) ⇒
    Suspend(() ⇒ filter(rest())(p), stop)
  // ...
}
```

FILTER

```scala
def filter[A](fa: Iterant[A])(p: A ⇒ Boolean): Iterant[A] =
  fa match {
  // ...
  case Next(a, rest, stop) ⇒
    if (p(a)) Next(a, () ⇒ filter(rest())(p), stop)
    else Suspend(() ⇒ filter(rest())(p), stop)
}
```

TRAMPOLINES

CAN WE DO THIS ?

```
case class Next[A](
  item: A,
  rest: Future[Iterant[F, A]],
  stop: Future[Unit])
extends Iterant[A]
```

CAN WE DO THIS ?

```scala
type Task[+A] = () ⇒ Future[A]
```

```scala
case class Next[A](
  item: A,
  rest: Task[Iterant[F, A]],
  stop: Task[Unit])
  extends Iterant[A]
```

CAN WE DO THIS ?

```scala
import monix.eval.Task
```

```scala
case class Next[A](
  item: A,
  rest: Task[Iterant[F, A]],
  stop: Task[Unit])
  extends Iterant[A]
```

CAN WE DO THIS ?

```scala
import monix.eval.Coeval
```

```scala
case class Next[A](
  item: A,
  rest: Coeval[Iterant[F, A]],
  stop: Coeval[Unit])
  extends Iterant[A]
```

CAN WE DO THIS ?

```scala
import cats.effect.IO

case class Next[A](
  item: A,
  rest: IO[Iterant[F, A]],
  stop: IO[Unit])
  extends Iterant[A]
```

CAN WE DO THIS ?

```scala
import cats.Eval

case class Next[A](
  item: A,
  rest: Eval[Iterant[F, A]],
  stop: Eval[Unit])
  extends Iterant[A]
```

PARAMETRIC POLYMORPHISM

```scala
sealed trait Iterant[F[_], A]

case class Next[F[_], A](
  item: A,
  rest: F[Iterant[F, A]],
  stop: F[Unit])
  extends Iterant[F, A]
```

PARAMETRIC POLYMORPHISM

```scala
import cats.syntax.all._
import cats.effect.Sync

def filter[F[_], A](p: A ⇒ Boolean)(fa: Iterant[F, A])
  (implicit F: Sync[F]): Iterant[F, A] = {
  fa match {
    // ...
    case Suspend(rest, stop) ⇒
      Suspend(rest.map(filter(p)), stop)
    // ...
  }
}
```

BRING YOUR OWN BOOZE

```scala
import monix.eval.Task
```

```scala
val sum: Task[Int] =
  Iterant[Task].range(0, 1000)
    .filter(_ % 2 == 0)
    .map(_ * 2)
    .foldL
```

BRING YOUR OWN BOOZE

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

```scala
val sum: IO[Int] =
  Iterant[IO].range(0, 1000)
    .filter(_ % 2 == 0)
    .map(_ * 2)
    .foldL
```

BRING YOUR OWN BOOZE

```scala
import monix.eval.Coeval
```

```scala
val sum: Coeval[Int] =
  Iterant[Coeval].range(0, 1000)
    .filter(_ % 2 == 0)
    .map(_ * 2)
    .foldL
```

PERFORMANCE PROBLEMS

- Linked Lists are everywhere in FP
- Linked Lists are terrible
- Async or Lazy Boundaries are terrible
- Find Ways to work with Arrays and
- … to avoid lazy/async boundaries

PERFORMANCE SOLUTIONS

- Linked Lists are everywhere in FP
- Linked Lists are terrible
- Async or Lazy Boundaries are terrible
- Find Ways to work with Arrays and
- … to avoid lazy/async boundaries

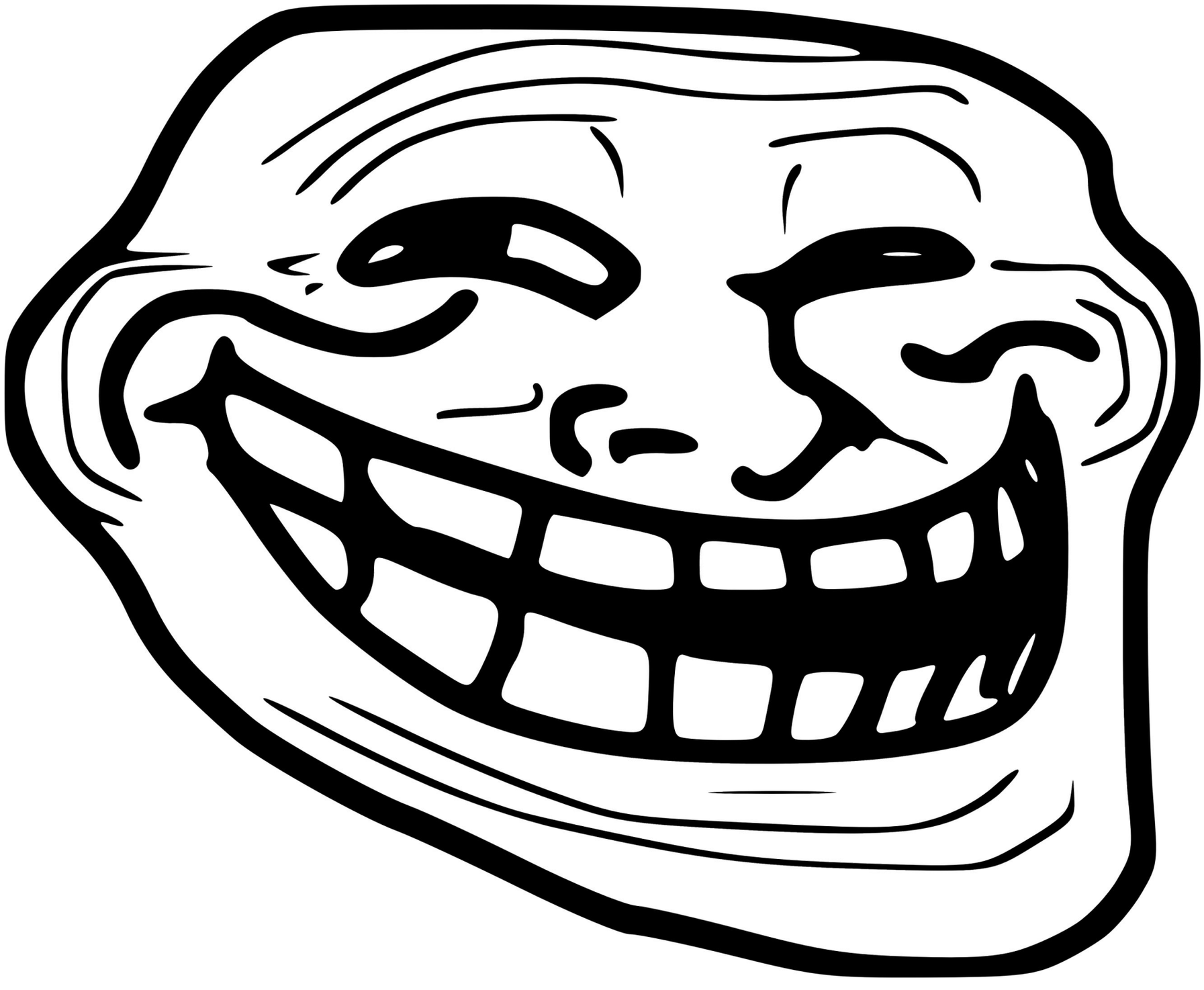
WHAT CAN ITERATE OVER ARRAYS?

WHAT CAN ITERATE OVER ARRAYS?

```
trait Iterator[+A] {
  def hasNext: Boolean
  def next(): A
}

trait Iterable[+A] {
  def iterator: Iterator[A]
}
```

WHAT CAN ITERATE OVER ARRAYS?

```
case class NextBatch[F[_], A](
  batch: Iterable[A],
  rest: F[Iterant[F, A]],
  stop: F[Unit])
  extends Iterant[F, A]

case class NextCursor[F[_], A](
  cursor: Iterator[A],
  rest: F[Iterant[F, A]],
  stop: F[Unit])
  extends Iterant[F, A]
```



OBSERVABLE[+A]

VS

ITERANT[F,A]

Case Study: scanEval

https://github.com/monix/monix/pull/412

Case Study: scanEval

```scala
import cats.effect.Sync

sealed abstract class Iterant[F[_], A] {
  //  ...
  def scanEval[S](seed: F[S])(op: (S, A) ⇒ F[S])
    (implicit F: Sync[F]): Iterant[F, S] = ???
}
```

Case Study: scanEval

```
def loop(state: S)(source: Iterant[F, A]): Iterant[F, S] =
  try source match {
    case Next(head, tail, stop) ⇒
      protectedF(state, head, tail, stop)
    case ref @ NextCursor(cursor, rest, stop) ⇒
      evalNextCursor(state, ref, cursor, rest, stop)
    case NextBatch(gen, rest, stop) ⇒
      val cursor = gen.cursor()
      val ref = NextCursor(cursor, rest, stop)
      evalNextCursor(state, ref, cursor, rest, stop)
    case Suspend(rest, stop) ⇒
      Suspend[F,S](rest.map(loop(state)), stop)
    case Last(item) ⇒
      val fa = ff(state, item)
      Suspend(fa.map(s ⇒ lastS[F,S](s)), F.unit)
    case halt @ Halt(_) ⇒
      halt.asInstanceOf[Iterant[F, S]]
  } catch {
    case NonFatal(ex) ⇒ signalError(source, ex)
  }
```

Case Study: scanEval

```scala
import cats.effect.Effect
```

```scala
abstract class Observable[+A] {
  //…
  def scanEval[F[_], S](seed: F[S])(op: (S, A) ⇒ F[S])
    (implicit F: Effect[F]): Observable[S] = ???
}
```

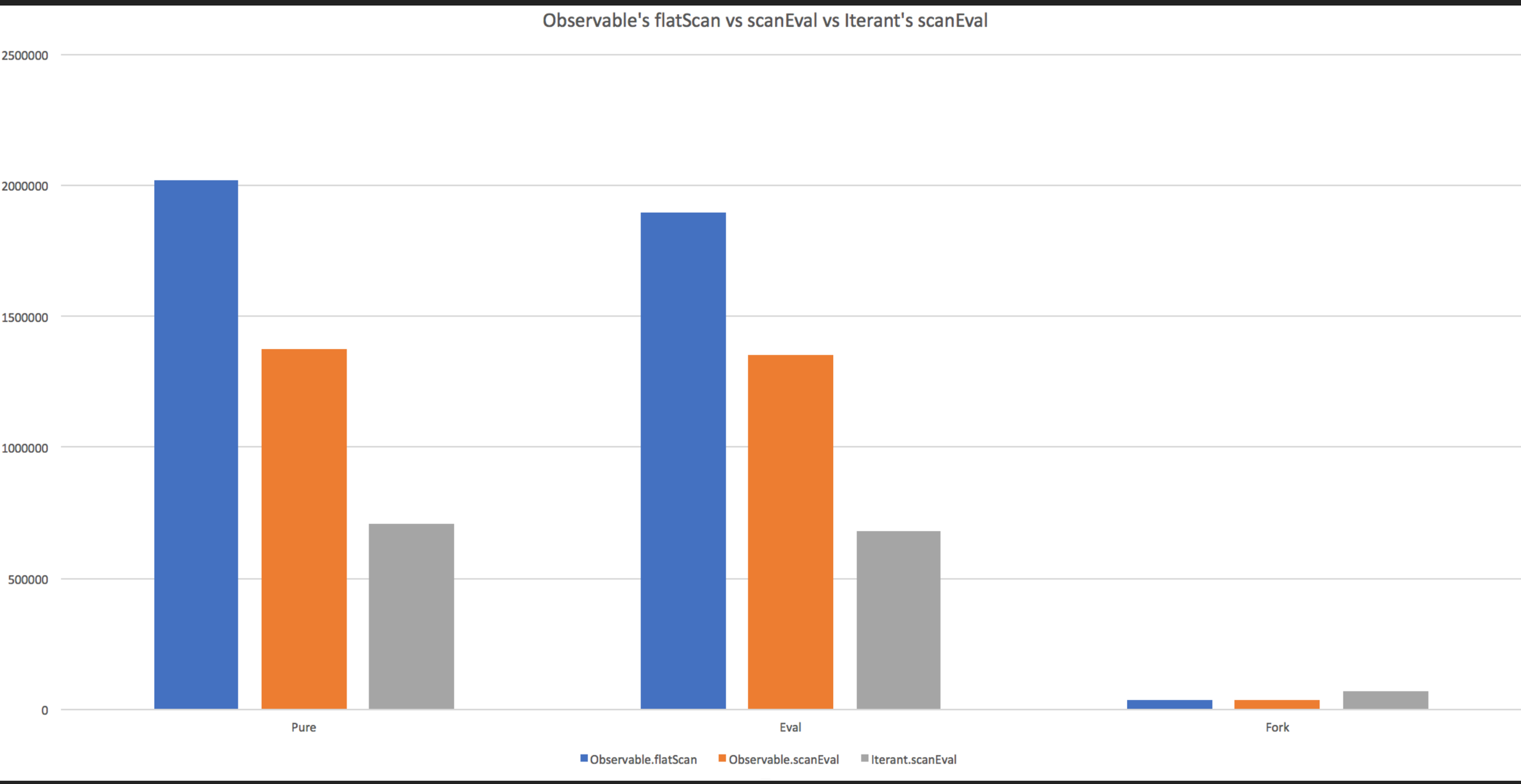
Case Study: scanEval

```scala
abstract class Observable[+A] {
  // ...
  def scanEval[F[_], S](seed: F[S])(op: (S, A) ⇒ F[S])
    (implicit F: Effect[F]): Observable[S] =
    scanTask(Task.fromEffect(seed))((a,e) ⇒ Task.fromEffect(op(a,e)))

  def scanTask[S](seed: Task[S])(op: (S, A) ⇒ Task[S]): Observable[S] =
    ???
}
```

Case Study: scanEval

- Observable's scanTask is flatMap
- With the aforementioned implementation

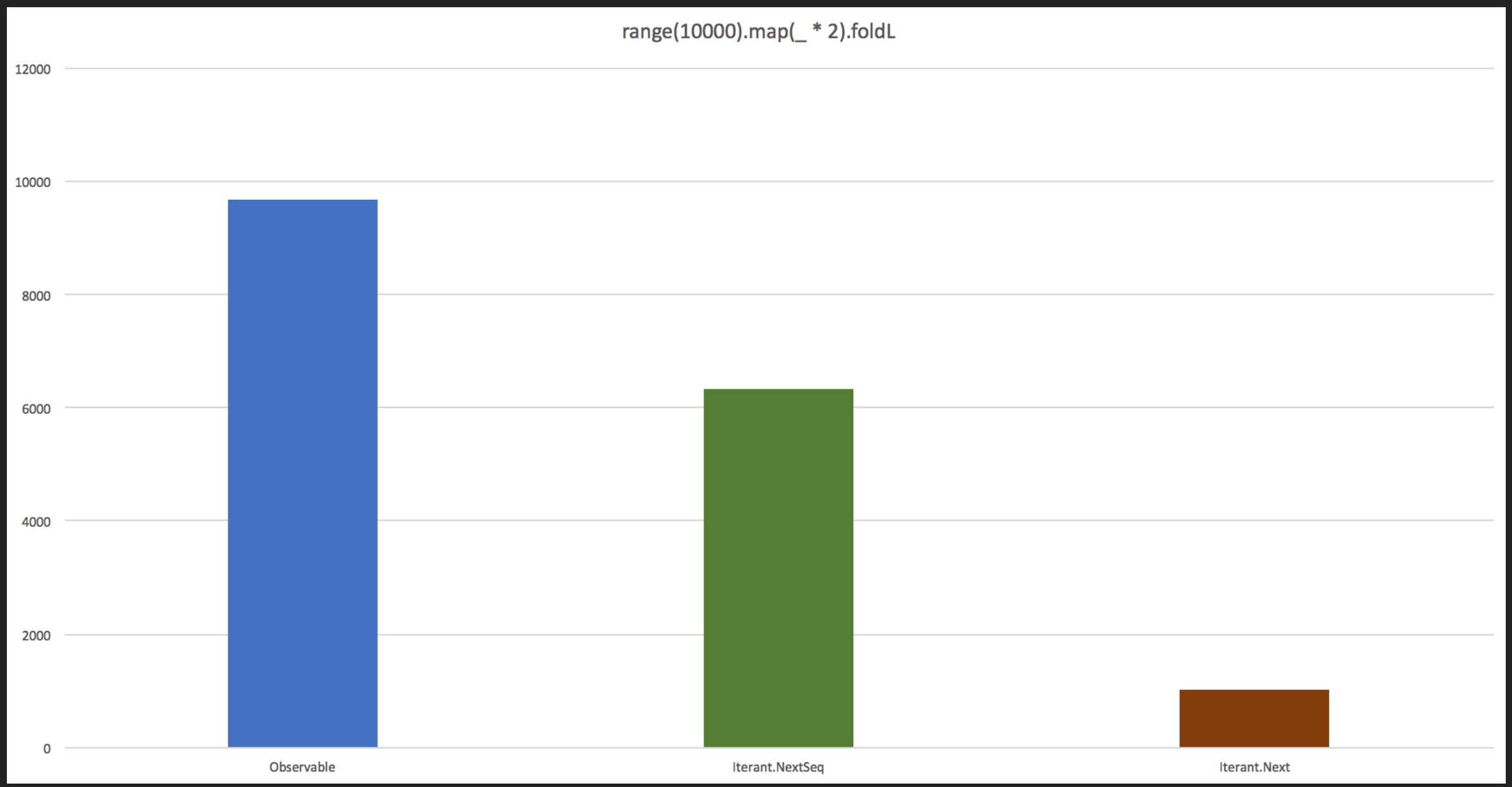
Benchmark: scanEval



Case Study: scanEval

- Observable has performance
- Iterant has reason

Benchmark: map(f).foldL



OBSERVABLE FOLD-RIGHT

- Cannot express **foldRight** for **Observable**
- But can work with substitutes, e.g. **foldWhileLeftL ...**

```
abstract class Observable[+A] {
  //  ...
  def foldWhileLeftL[S](seed: ⇒ S)
    (op: (S, A) ⇒ Either[S, S]): Task[S]
}
```

ITERANT FOLD-RIGHT

```
sealed abstract class Iterable[F[_], A] {
  //   ...
  def foldRightL[B](b: F[B])
    (f: (A, F[B], F[Unit]) ⇒ F[B])
    (implicit F: Sync[F]): F[B]
}
```

ITERANT FOLD-RIGHT

```scala
def exists[F[_], A](ref: Iterant[F, A], p: A ⇒ Boolean)
(implicit F: Sync[F]): F[Boolean] =

ref.foldRightL(F.pure(false)) { (e, next, stop) ⇒
  if (p(e)) stop followedBy F.pure(true)
  else next
}
```

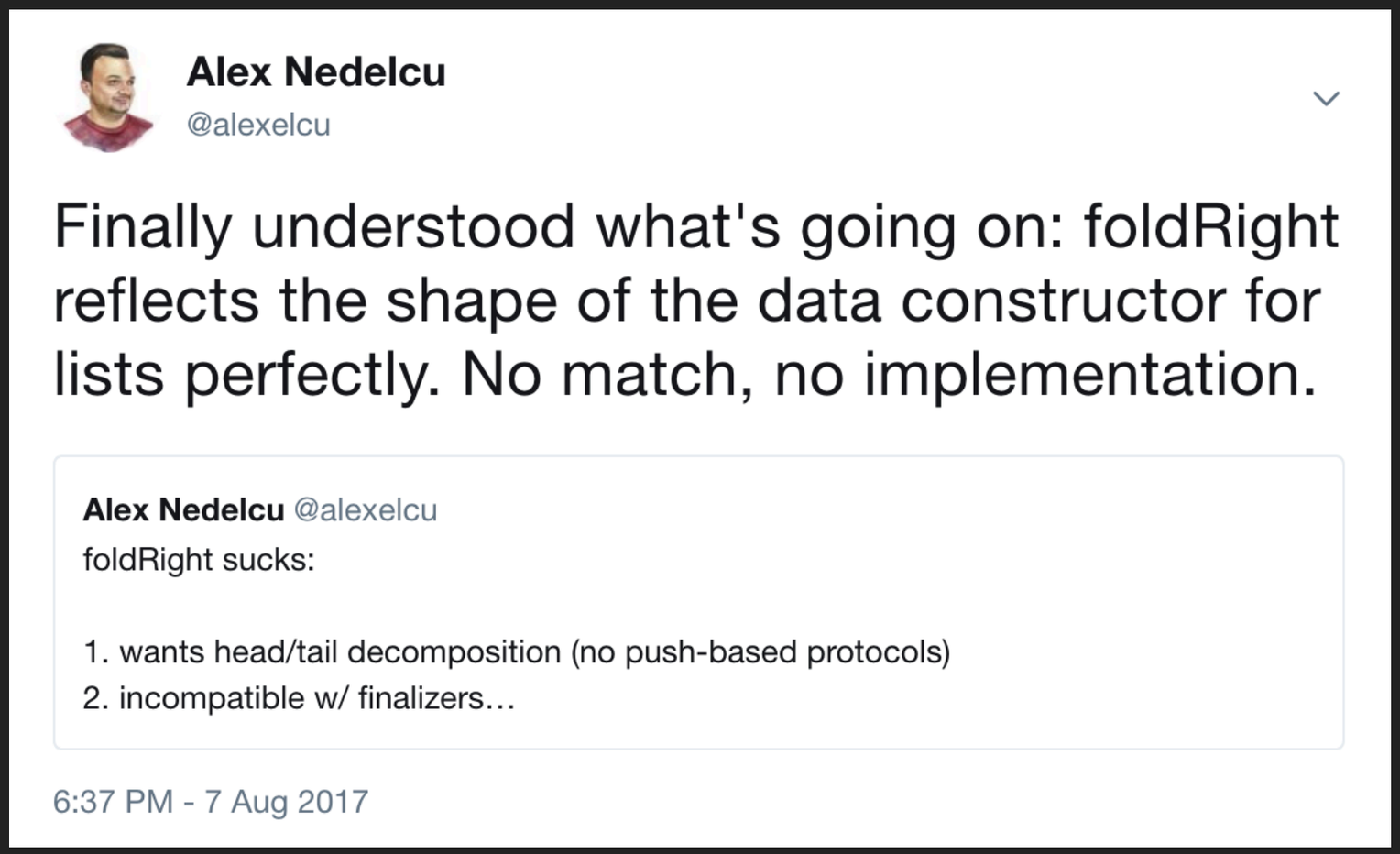
ITERANT FOLD-RIGHT

```
def forall[F[_], A](ref: Iterant[F, A], p: A ⇒ Boolean)
(implicit F: Sync[F]): F[Boolean] =

ref.foldRightL(F.pure(true)) { (e, next, stop) ⇒
  if (!p(e)) stop followedBy F.pure(false)
  else next
}
```

ITERANT FOLD-RIGHT

```
def concat[F[_], A](lh: Iterant[F, A], rh: Iterant[F, A])
(implicit F: Sync[F]): Iterant[F, A] =

Iterant.suspend[F, A] {
  lh.foldRightL(F.pure(rh)) { (a, rest, stop) ⇒
    F.pure(Iterant.nextS(a, rest, stop))
  }
}
```

ITERANT FOLD-RIGHT



Conclusions

- **Observable is best for**
  - Reactive operations
  - Shared data sources
  - Throttling
  - Buffering
  - Performance
- **Iterant is best for**
  - Easier implementation reasoning
  - Converting Task / IO calls into streams

Conclusions

- **Both**
  - Implement reactive-streams.org
  - Can express asynchronous streams
  - Do back-pressuring & safe resource handling

One more thing ...

VS

QUESTIONS?

monix.io

@monix

@monix

alexn.org

@alexelcu

@alexandru