FUNCTIONAL PROGRAMMING INCEPTION

Alexandru Nedelcu

Software Developer @ eloquentix.com

@alexelcu / alexn.org

WHAT IS FUNCTIONAL PROGRAMMING?

WHAT IS FUNCTIONAL PROGRAMMING?

A: Programming with *Mathematical Functions*

PROPERTIES OF FP

- FP <=> Programming with Values
- Referential Transparency
- Composability, Reason

ITERATOR

CASE STUDY ON THE WORLD MOST FAMOUS OOP ABSTRACTION

HOW DID ITERATOR HAPPEN?

```
int[] array = ???;
int sum = 0;

for (int i = 0; i < length(array); i += 1)
  sum += array[i];
```

HOW DID ITERATOR HAPPEN?

```
val array: Array[Int] = ???
var sum = 0

var i = 0
while (i < array.length) {
  sum += array(i)
  i += 1
}
```

HOW DID ITERATOR HAPPEN?

```
val array: Array[Int] = ???
var sum = 0


var i = 0 //<-- Start!
while (i < array.length) { //<-- Has Current?
   sum += array(i) //<-- Get Current
   i += 1 //<-- Next Cycle Please
}
```

HOW DID ITERATOR HAPPEN?

```
package scala.collection

trait Iterator[+A] {

  def hasNext: Boolean
  def next(): A
}
```

HOW DID ITERATOR HAPPEN?

```
val array: Array[Int] = ???
var sum = 0


val cursor = array.iterator //<-- Start!
while (cursor.hasNext) { //<-- Has Next?
  // Get Current & Advance
  sum += cursor.next()
}
```

PROBLEMS ?

PROBLEMS ?

- Synchronous Only
  - blocks threads for async stuff
- no way around it, it’s in the signature

PROBLEMS ?

- Synchronous Only

- No Backed-in Resource Managed

```
// Managed language devs have no discipline ;-)
iterator.take(100).sum
```

PROBLEMS ?

- Synchronous Only
- No Backed-in Resource Managed
- Minefield for Stack Overflows

```
def range(from: Long, until: Long): Iterator[Long] = {
  if (from < until)
    Iterator(from) ++ range(from + 1, until)
  else
    Iterator.empty
}

range(0, 1000000).sum
//java.lang.StackOverflowError (in Scala 2.11)
```

FP DESIGN

HOW TO

ARCHITECTURE IS FROZEN MUSIC

Johann Wolfgang Von Goethe

DATA STRUCTURES ARE FROZEN ALGORITHMS

Jon Bentley

KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*
   (Immutable)

KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*

2. Think *State Machines*
(most of the time)

KEY INSIGHTS

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

   (Strict Values => Functions ;-))

KEY INSIGHTS

1. Freeze Algorithms into *Data-Structures*
2. Think *State Machines*
3. Be *Lazy*
4. Evaluate Effects w/ Stack-safe *Monads*
   (e.g. IO, Task, Free)

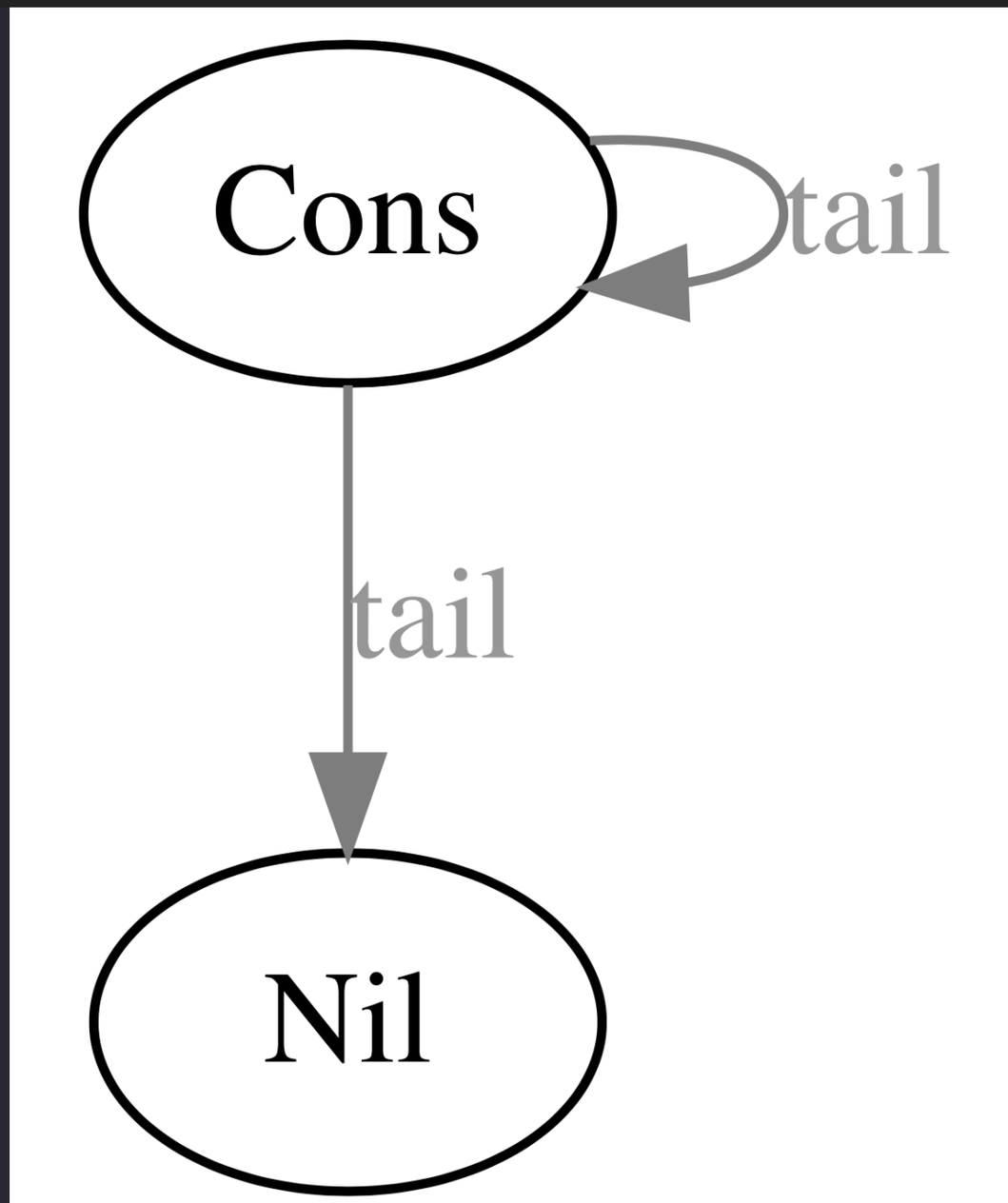

Finite State Machine Cat

EXAMPLE: LINKED LISTS

```
sealed trait List[+A]

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

case object Nil
  extends List[Nothing]
```

EXAMPLE: LINKED LISTS

```
sealed trait List[+A]

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

case object Nil
  extends List[Nothing]
```



EXAMPLE: LINKED LISTS

```
sealed trait List[+A]

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

case object Nil
  extends List[Nothing]
```

ITERANT

A PURELY FUNCTIONAL ITERATOR

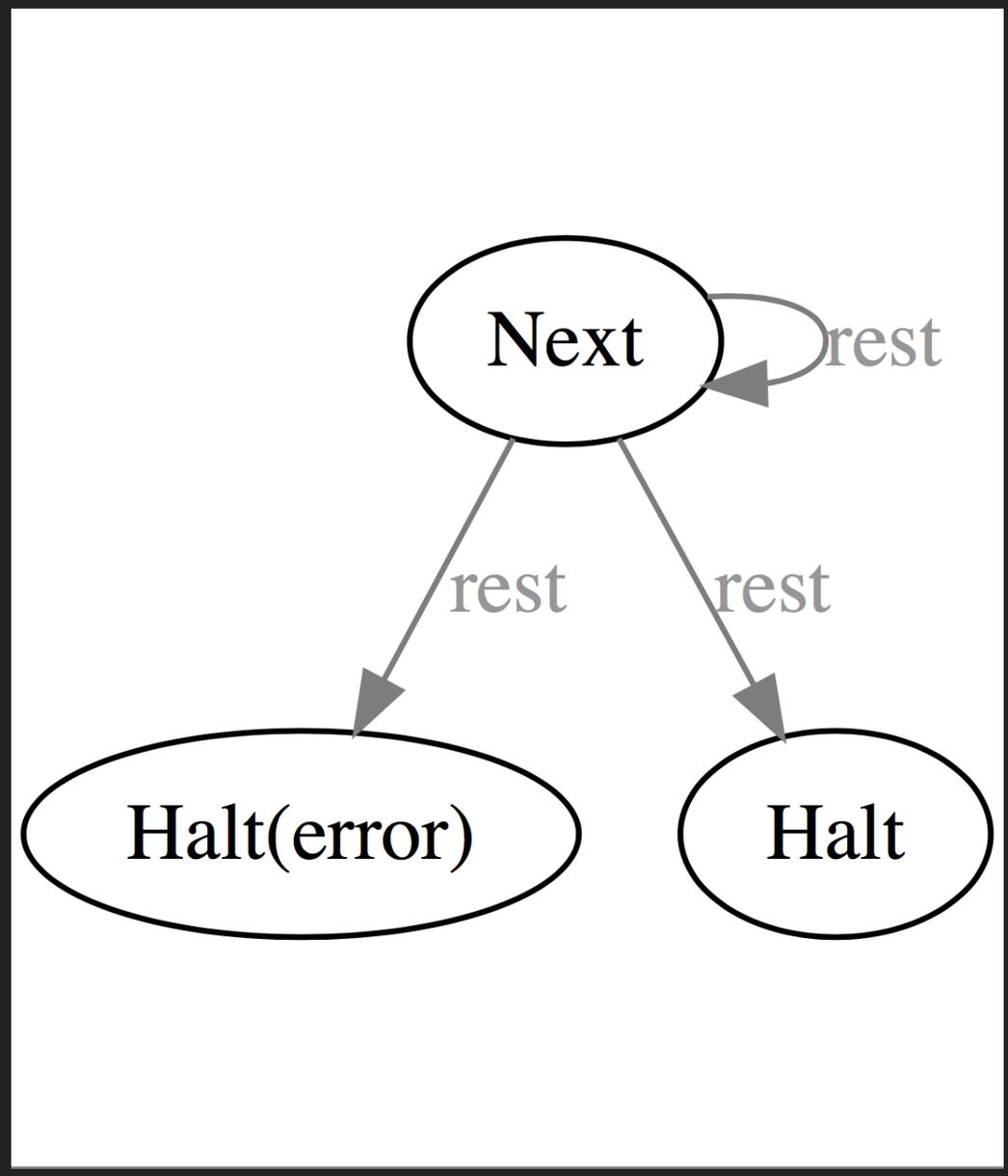
LAZY EVALUATION

```scala
sealed trait Iterant[+A]

case class Next[+A](
  item: A,
  rest: () => Iterant[A])
  extends Iterant[A]

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

λ-calculus: using anonymous functions because of privacy concerns

LAZY EVALUATION

```
sealed trait Iterant[+A]

case class Next[+A](
  item: A,
  rest: () => Iterant[A])
  extends Iterant[A]

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



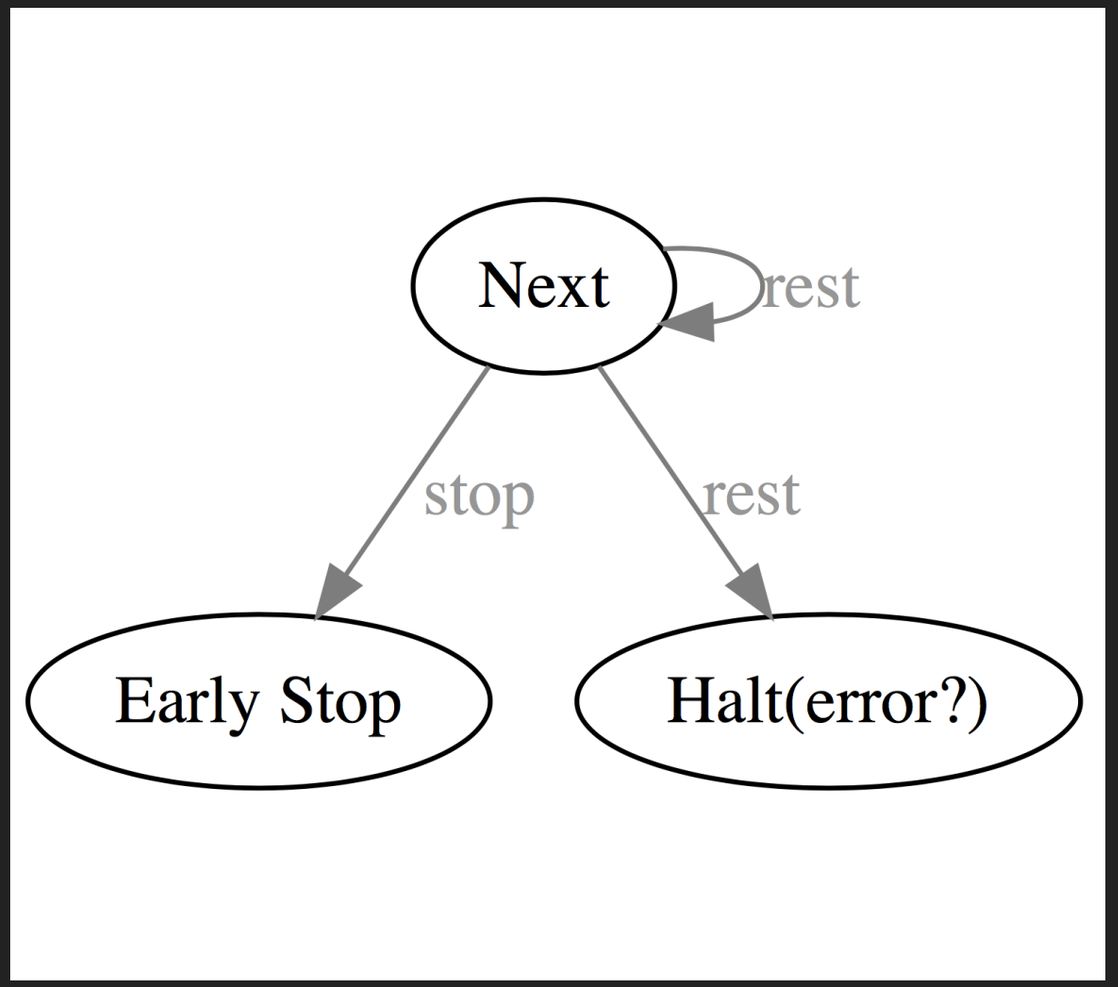
λ-calculus: using anonymous functions because of privacy concerns

USAGE

```scala
def sum(ref: Iterant[Int], acc: Int): Int
  ref match {
  case Halt(None) => acc
  case Halt(ex) => throw ex
  case Next(a, rest) =>
    sum(rest(), acc + a)
}
```

RESOURCE MANAGEMENT

```scala
sealed trait Iterant[+A]

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

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



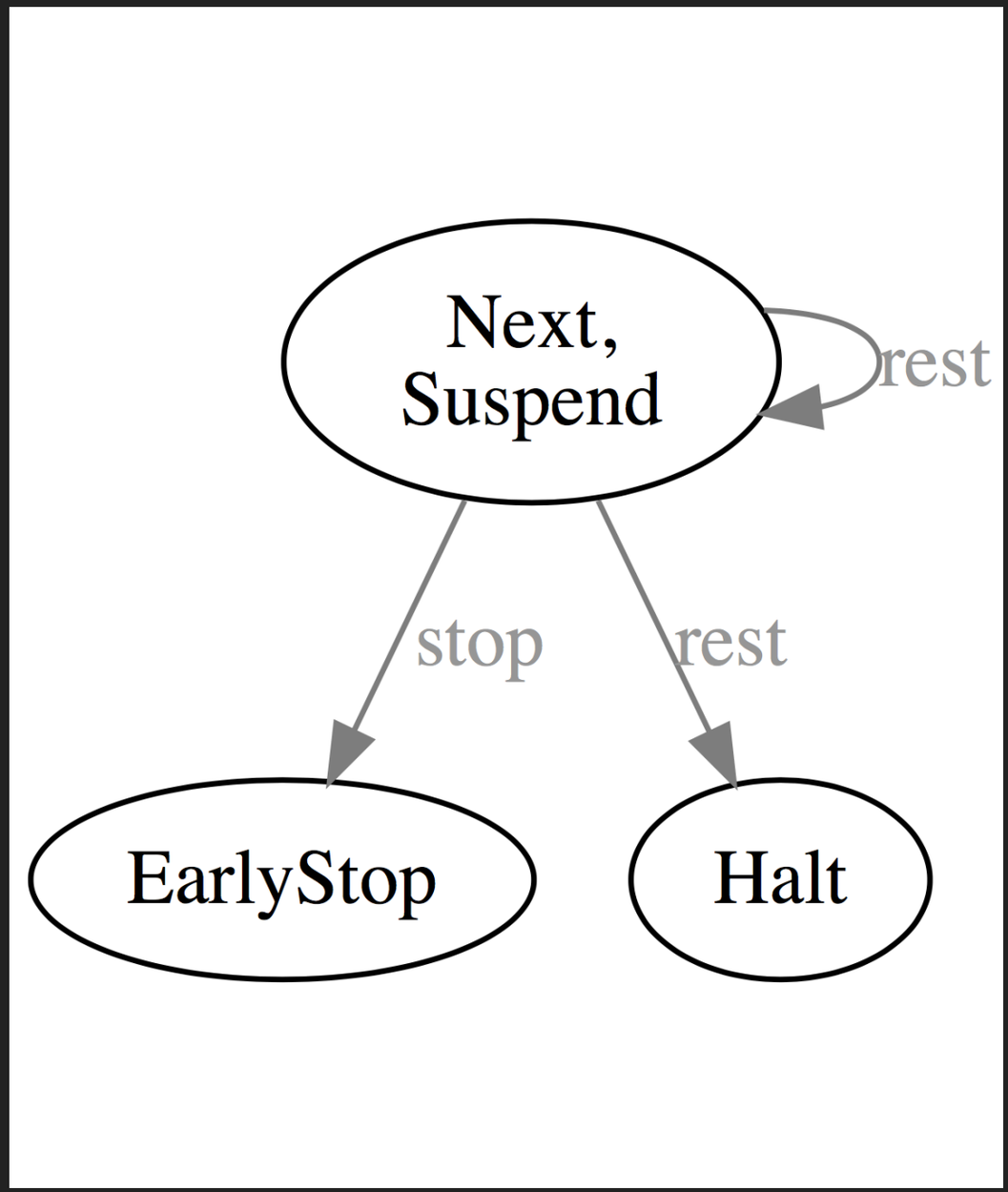
USAGE

```
def map[A,B](fa: Iterant[A])(f: A => B): Iterant[B] =
  fa match {
    case halt @ Halt(_) => halt
    case Next(a, rest, stop) =>
      try Next(f(a), map(rest)(f), stop)
      catch { case NonFatal(ex) =>
        stop(); Halt(Some(ex))
      }
}
```

Not pure yet, not referentially transparent

DEFERRING

```
sealed trait Iterant[+A]

// ...

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



USAGE

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

USAGE

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

USAGE

```
def filter[A](fa: Iterant[A])(p: A => Boolean): Iterant[A] =
  fa match {
    //...
  case Next(a, rest, stop) =>
    try {
      val continue = () => filter(rest())(p)
      if (p(a)) Next(a, continue, stop)
      else Suspend(continue, stop)
    } catch { case NonFatal(ex) =>
      Suspend(() => { stop(); Halt(Some(ex)) }, stop)
    }
}
```

ASYNCHRONY

CONCURRENCY, NON-DETERMINISM

QUICK INTRO

```
type Callback[-A] =
  (A) => Unit
```

QUICK INTRO

```
type Callback[-A] =
  (A) => Unit

type Async[+A] =
  (Callback[A]) => Unit
```

QUICK INTRO

```
type Callback[-A] =
  (A) => Unit

type Async[+A] =
  (Callback[A]) => Unit

type Future[+A] =
  (Callback[A], ExecutionContext) => Unit
```

CAN WE DO THIS ?

```
case class Next[+A](
  items: Iterator[A],
  rest: Future[Iterant[A]],
  stop: Future[Unit])
extends Iterant[A]
```

EVALUATION IN SCALA

	Eager	Lazy
	A	**() => A**

EVALUATION IN SCALA

	Eager	Lazy
Synchronous	**A**	**() => A**
Asynchronous	**(A => Unit) => Unit**	**() => (A => Unit) => Unit**

EVALUATION IN SCALA

	Eager	Lazy
Synchronous	**A**	**() => A**
		Function0[A]
Asynchronous	**(A => Unit) => Unit**	**() => (A => Unit) => Unit**
	Future[A]	**Task[A]**

“A FUTURE REPRESENTS A VALUE, DETACHED FROM TIME”

Viktor Klang

GOING LAZY (AGAIN)

```
type Task[+A] = () => Future[A]

case class Next[+A](
  items: Iterator[A],
  rest: Task[Iterant[A]],
  stop: Task[Unit])
  extends Iterant[A]
```

MONIX TASK

- High-performance
- Lazy, possibly asynchronous behaviour
- Allows for cancelling of a running computation
- https://monix.io/docs/2x/eval/task.html

GOING LAZY (AGAIN)

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

HIGHER-KINDED POLYMORPHISM

Bring Your Own Booze

CAN WE DO THIS ?

```
import scalaz.effects.IO

case class Next[+A](
  items: Iterator[A],
  rest: IO[Iterant[A]],
  stop: IO[Unit])
  extends Iterant[A]
```

CAN WE DO THIS ?

```
import cats.Eval

case class Next[+A](
  items: Iterator[A],
  rest: Eval[Iterant[A]],
  stop: Eval[Unit])
  extends Iterant[A]
```

GENERICS OF A HIGHER KIND

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

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

GENERICS OF A HIGHER KIND

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

GENERICS OF A HIGHER KIND

```
def foldLeftL[F[_], A, S](fa: Iterant[F, A])
  (seed: => S)(op: (S,A) => S)
  (implicit F: Monad[F]): F[S] = {

  // Checkout https://github.com/monix/monix/pull/280
}
```

OOP VS PARAMETRIC POLYMORPHISM

OOP VS PARAMETRIC POLYMORPHISM

- OOP is about *Information Hiding* (in types too)
- OOP handles Heterogeneity

OOP VS PARAMETRIC POLYMORPHISM

- OOP is about *Information Hiding* (in types too)
- OOP handles Heterogeneity
- Parametric Polymorphism is compile-time
- Fundamentally changes behaviour based on plugged-in types

OOP VS PARAMETRIC POLYMORPHISM

- ArrayIterator vs ListIterator
- Iterant[Task] vs Iterant[Eval]

OOP VS PARAMETRIC POLYMORPHISM

- ArrayIterator vs ListIterator
- Iterant[Task, _] vs Iterant[Eval, _]

- One is hiding implementation details
- The other is about composition

PROBLEMS

- Pushes compiler to its limits

```
[error] /Users/alex/Projects/monix/monix/monix-tail/shared/src/main/scala/monix/tail/internal/Iter
[error]  found   : monix.tail.Iterant.NextSeq[F,?A2] where type ?A2 <: A (this is a GADT skolem)
[error]  required: monix.tail.Iterant.NextSeq[F,A]
[error] Note: ?A2 <: A, but class NextSeq is invariant in type A.
[error] You may wish to define A as +A instead. (SLS 4.5)
[error]         evalNextSeq(ref, cursor, rest, stop)
[error]                     ^
```


```
[error]  found   : monix.tail.Iterant.NextSeq[F,?A2] where type ?A2 <: A (this is a GADT skolem)
[error]  required: monix.tail.Iterant.NextSeq[F,A]
[error] Note: ?A2 <: A, but class NextSeq is invariant in type A.
[error] You may wish to define A as +A instead. (SLS 4.5)
[error]         evalNextSeq(ref, cursor, rest, stop)
[error]                     ^
```

PROBLEMS

- Pushes compiler to its limits

- Unfamiliarity for users

PROBLEMS

- Pushes compiler to its limits
- Unfamiliarity for users
- Not all needed type-classes are available, design can be frustrating

https://github.com/typelevel/cats/pull/1552

(39 comments and counting)

UPSIDE

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

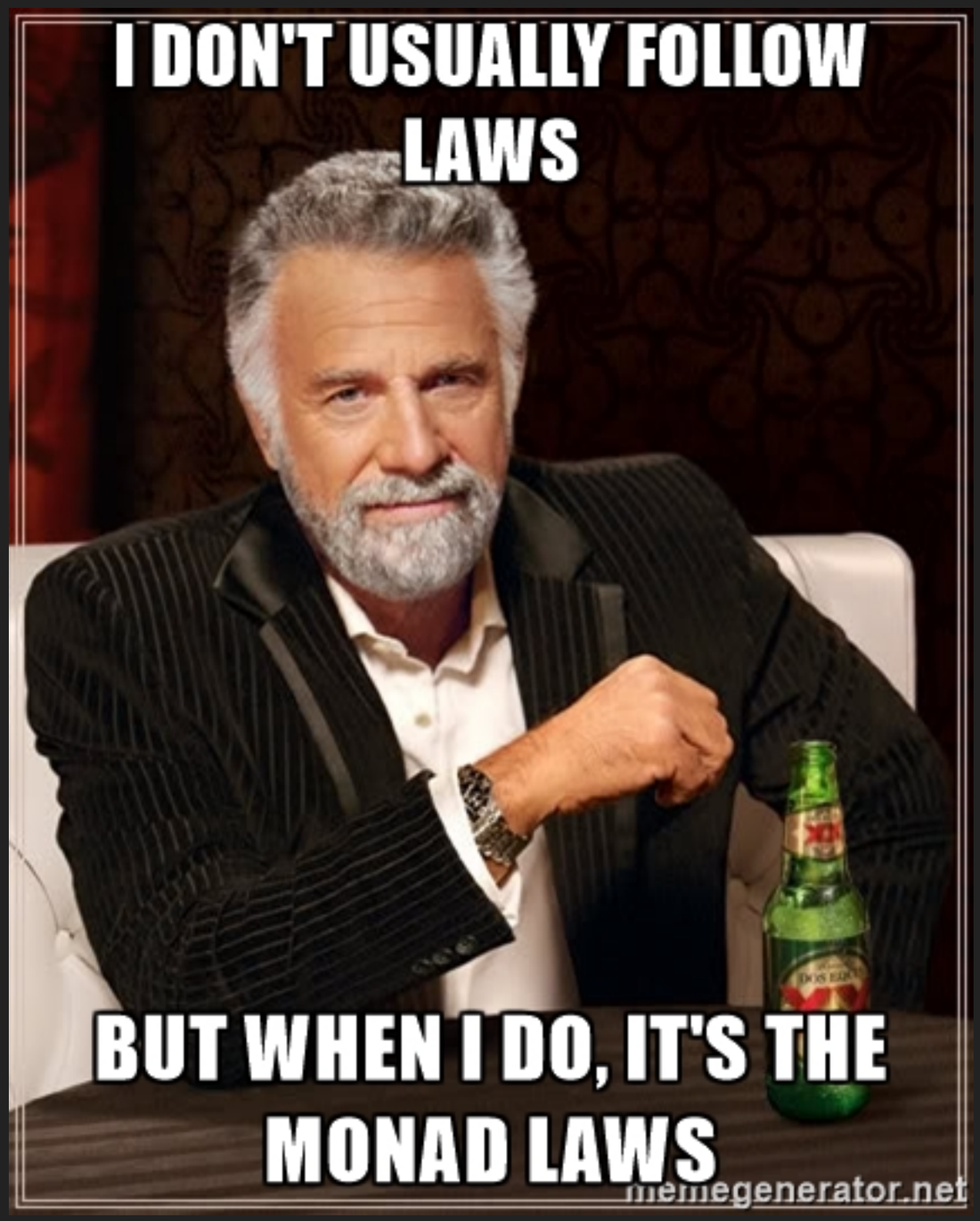
```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def map2[A,B,R](fa: A, fb: B)(f: (A,B) => R): F[R]
}

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

LAWS

```
// Left Identity
pure(a).flatMap(f) <-> f(a)

// Right Identity
m.flatMap(pure) <-> m

// Associativity
fa.flatMap(f).flatMap(g) <-> fa.flatMap(a => f(a).flatMap(g))
```

LAWS

- Typelevel Cats
- Typelevel Discipline
- ScalaCheck



PERFORMANCE PROBLEMS

- Linked Lists are everywhere in FP
- Linked Lists are terrible
- Async or Lazy Boundaries are terrible

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

PERFORMANCE SOLUTIONS

Efficient

head/tail

decomposition

needed ;-)

```
case class NextGen[+A](
  items: Iterable[A],
  rest: Task[Iterant[A]],
  stop: Task[Unit])
  extends Iterant[A]

case class NextSeq[+A](
  items: Iterator[A],
  rest: Task[Iterant[A]],
  stop: Task[Unit])
  extends Iterant[A]
```

OTHER PROBLEMS

- Recursion is terrible
- Space leaks are hard to fix

OTHER PROBLEMS

- Recursion is terrible
- Space leaks are hard to fix
- Solvable with pain and YourKit

TAKEAWAYS

TAKEAWAYS

- Freeze Algorithms into Immutable Data-Structures
- Describe State Machines
- Be lazy, suspend side-effects with Task/Free/IO
- Be lawful, use ScalaCheck/QuickCheck
- Performance matters (for libraries)

TAKEAWAYS

- Libraries:
  Monix, Cats, ScalaCheck
- Generic Iterant implementation:
  https://github.com/monix/monix/pull/280
- Simplified Task-based implementation:
  https://github.com/monix/monix/pull/331

QUESTIONS?