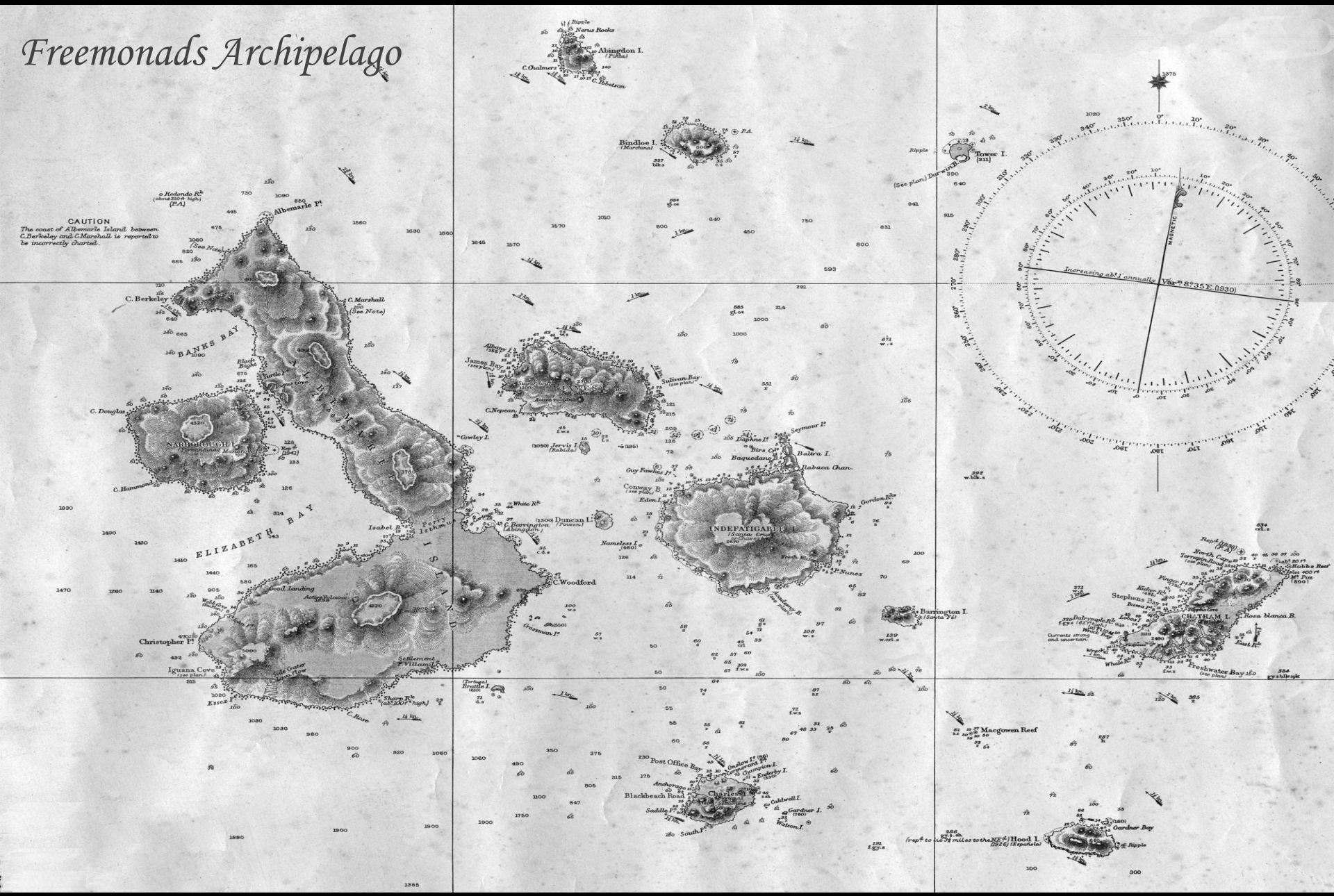
and also FreeMonads, Coproducts, Pure Functional DI

Functional Oceans

Freemonads Archipelago



Different Rites & Unknown Cultures

Great Explorers

Your Faithful & Obedient Servant

Pascal Voitot

twitter/github: @mandubian

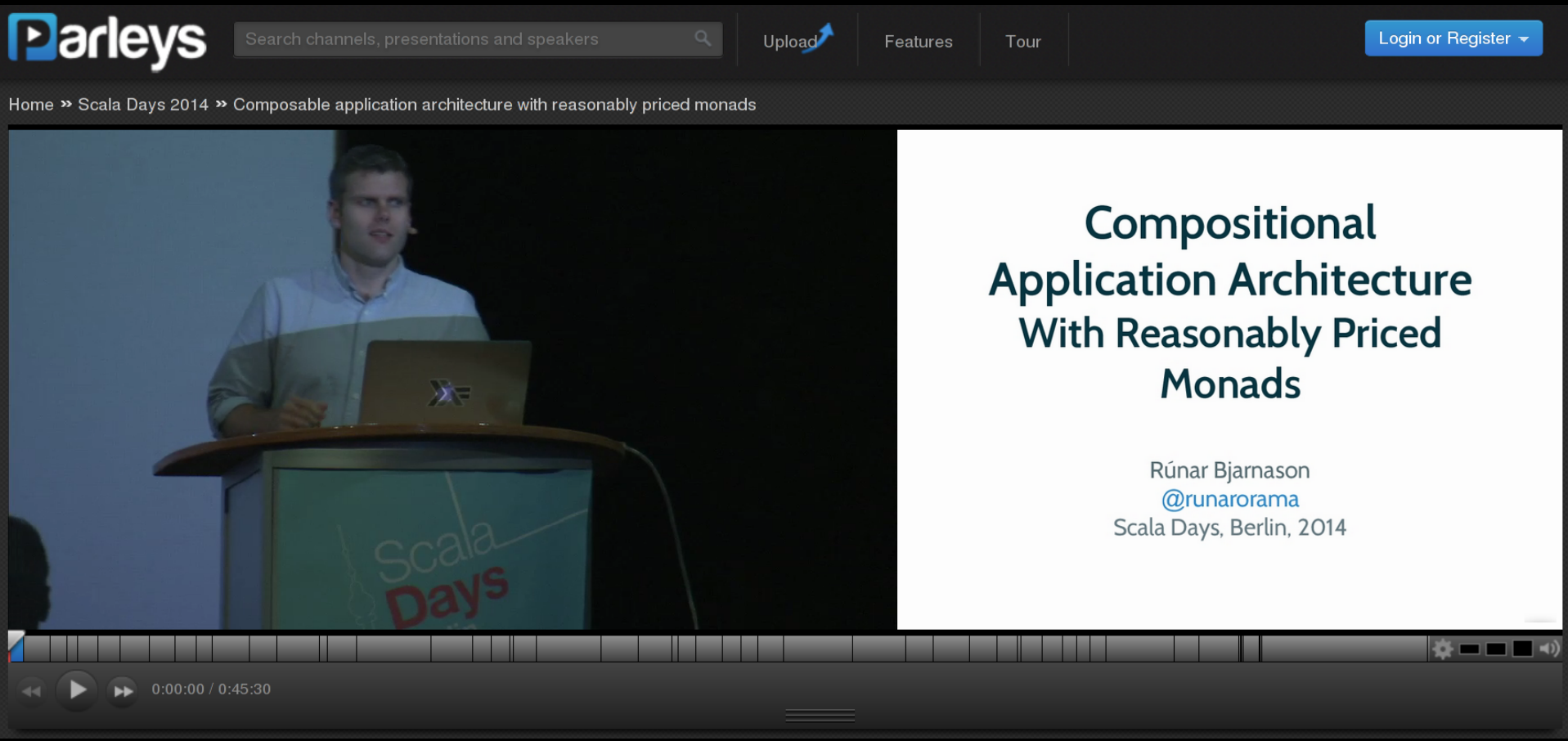
Backend / FP / Math / OSS

FreeMonadic Applications

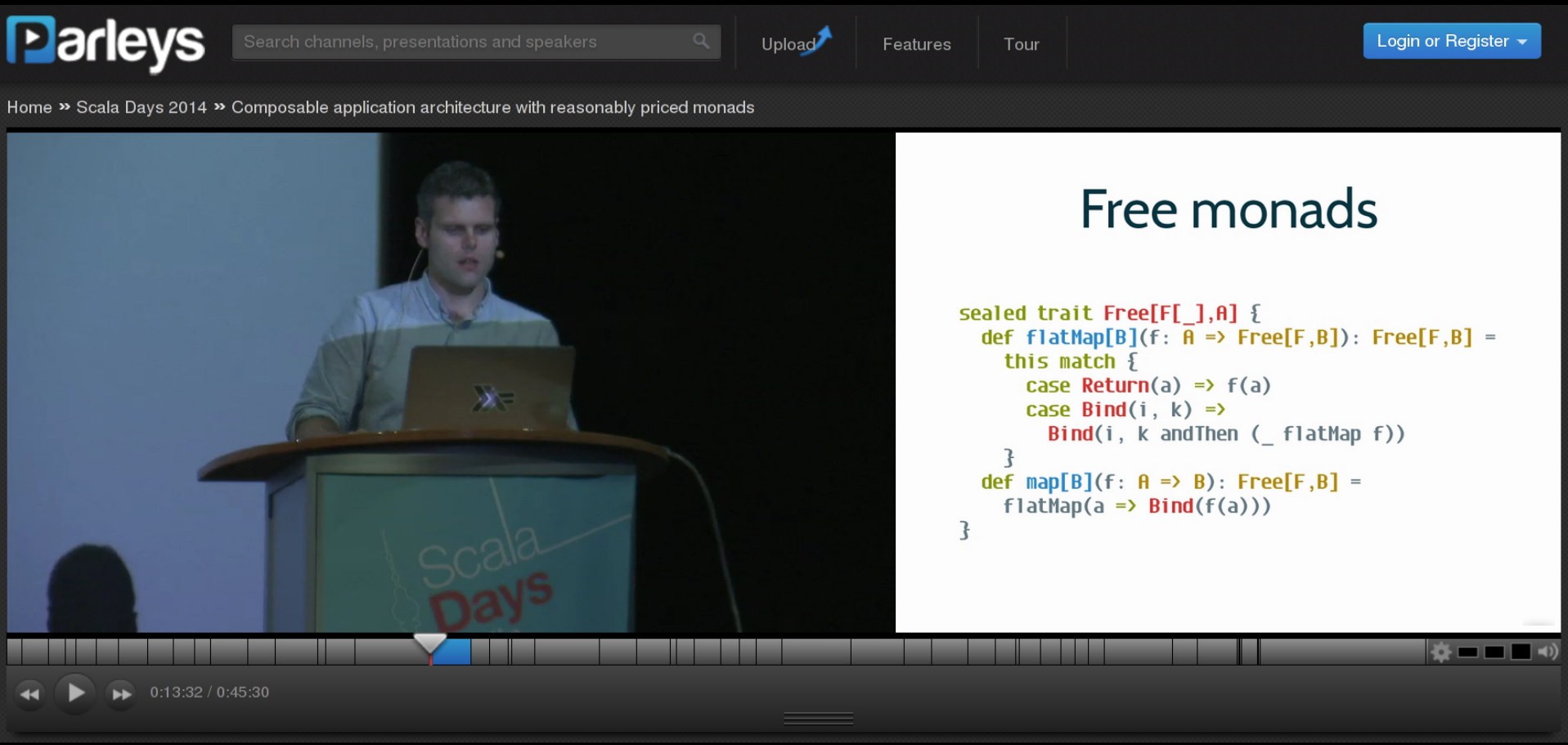
Bibliography

- Functional Programming in Scala (Chuisano & Bjarnason)
- ScalaDays 2014 Runar Bjarnason's talk
- Scala.IO 2014 Yorick Laupa's talk
- Articles:
  - DataTypes à la Carte (Swierstra)
  - Stackless Scala with Free Monads (Bjarnason)
  - Reflection without Remorse (Van der Ploeg, Kiselyov)

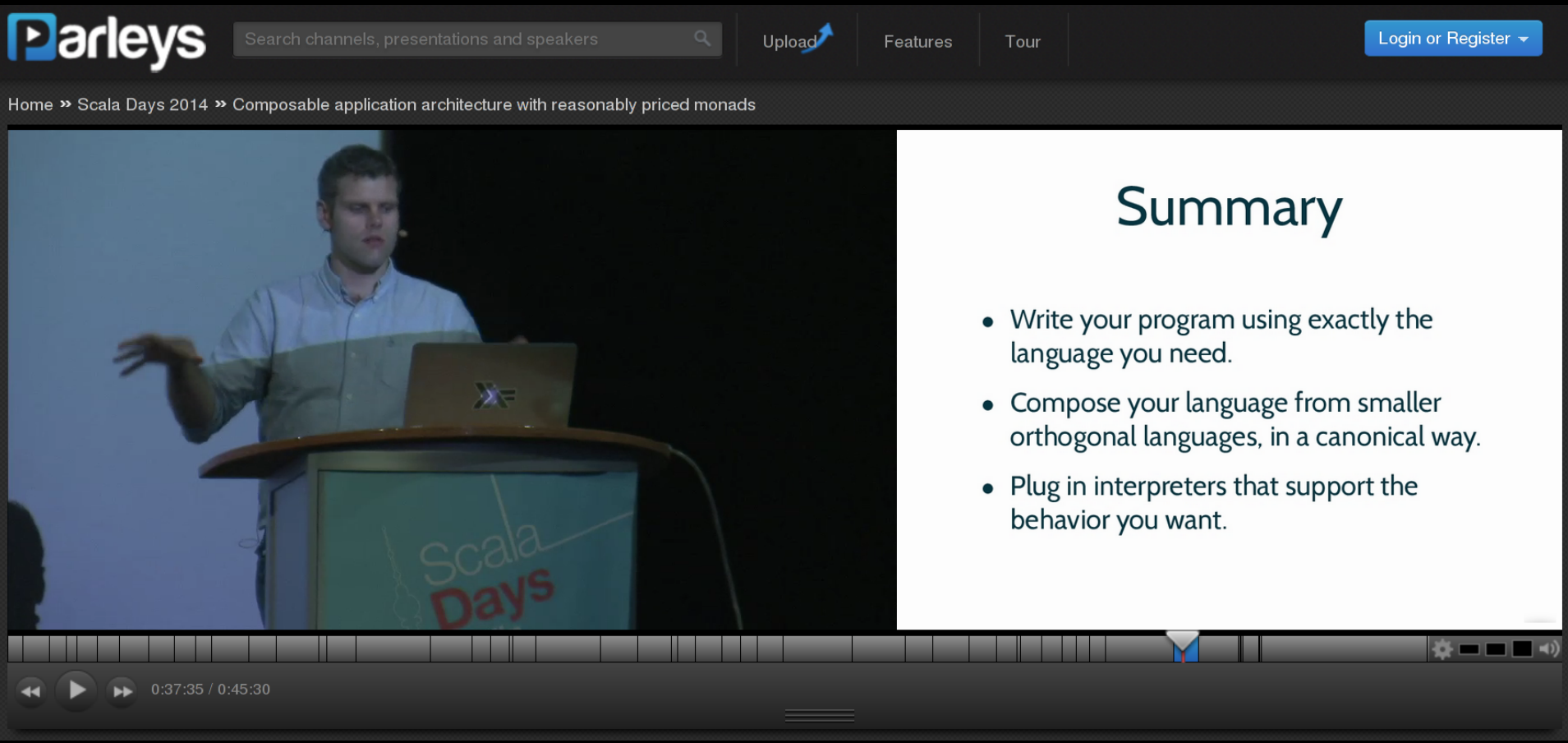
Reminders on Runar's talk



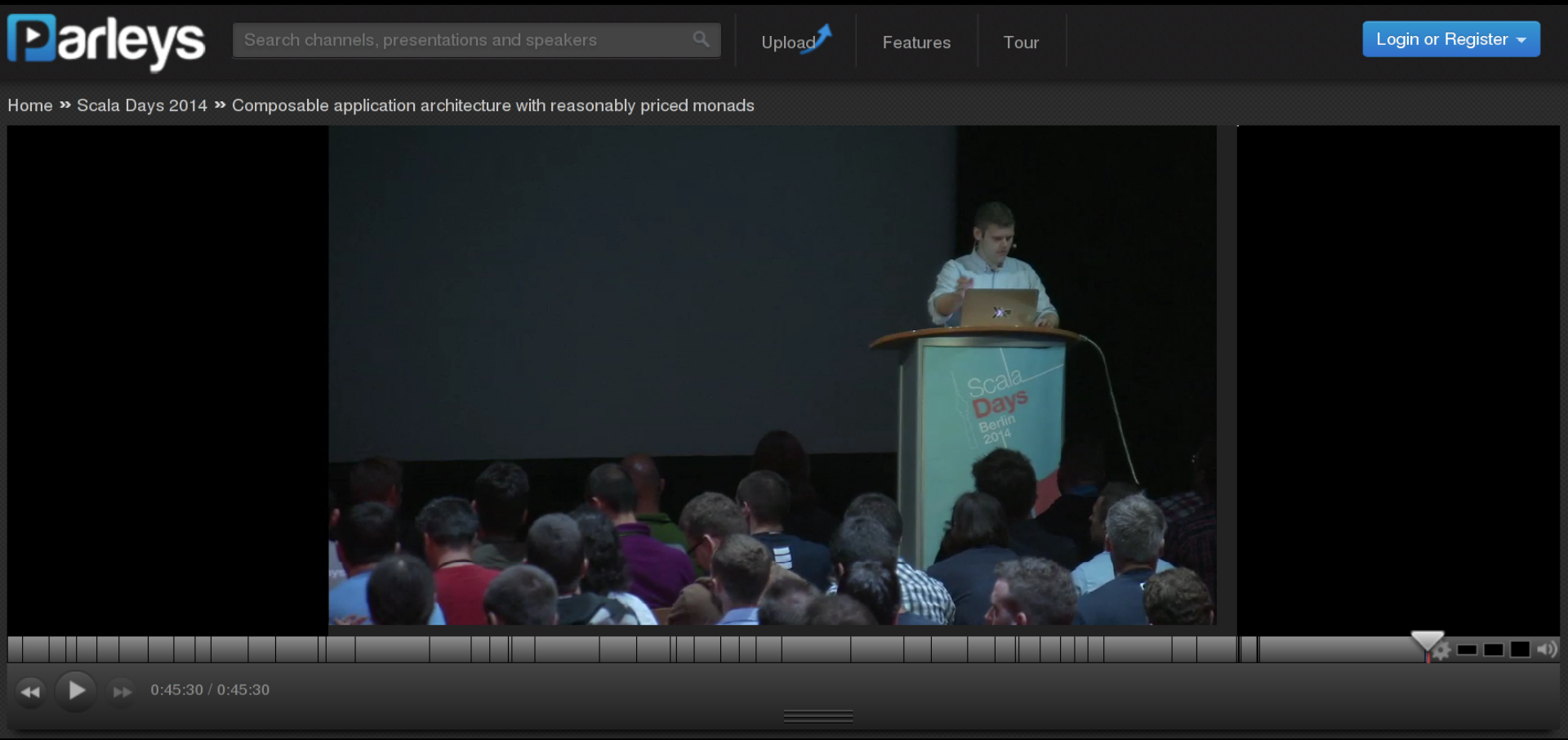
Reminders on Runar's talk



Reminders on Runar's talk



Reminders on Runar's talk



Dive Into Freemonadic Applications

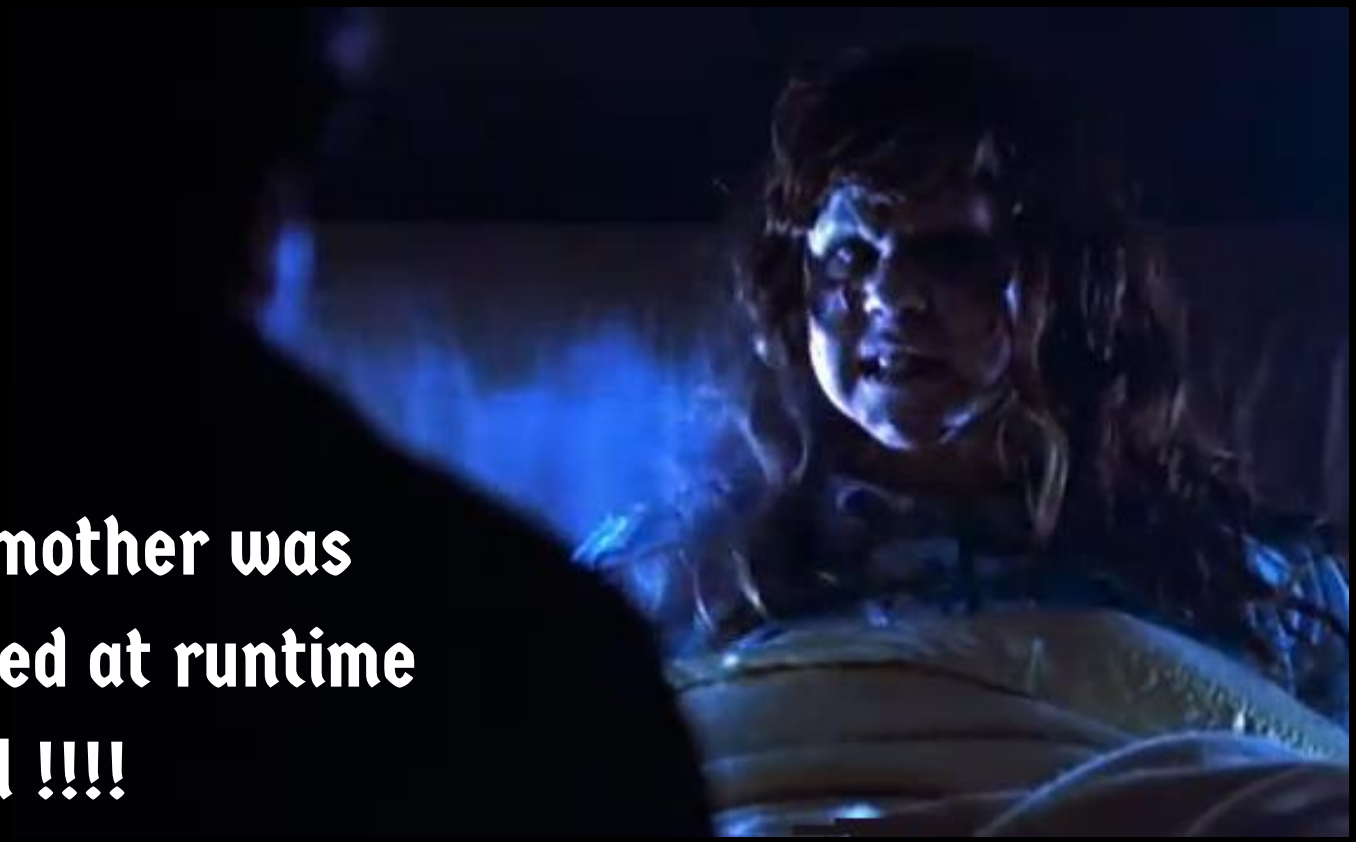
Our Vision of Application

- Modular
- Contractual
- Mockable
- Testable

- ~~Reusable~~
- ~~Pluggable~~

Runtime DI

Runtime DI is evil



Compile-Time DI is acceptable

- Cake Pattern
- Monad Transformers
- Scala macros (MacWire)
- ...

The Sample

Web Server

- Receive HttpRequest
- Handle request
- Generate HttpResponse
- Send HttpResponse

Classic Application

```
trait HttpInteract {
  def receive(): HttpReq
  def respond(resp: HttpResp): SendStatus
  def stop()
}

trait HttpHandle {
  def handle(req: HttpReq): HttpResp
}

trait App extends HttpInteract with HttpHandle {
  def main() = {
    val req = receive()
    val resp = handle(req)
    val status = respond(resp)
    if(status == Ack) main() else stop()
  }
}

object App extends App with HttpInteractImpl with HttpHandleImpl
```

Exploring alternatives to build applications

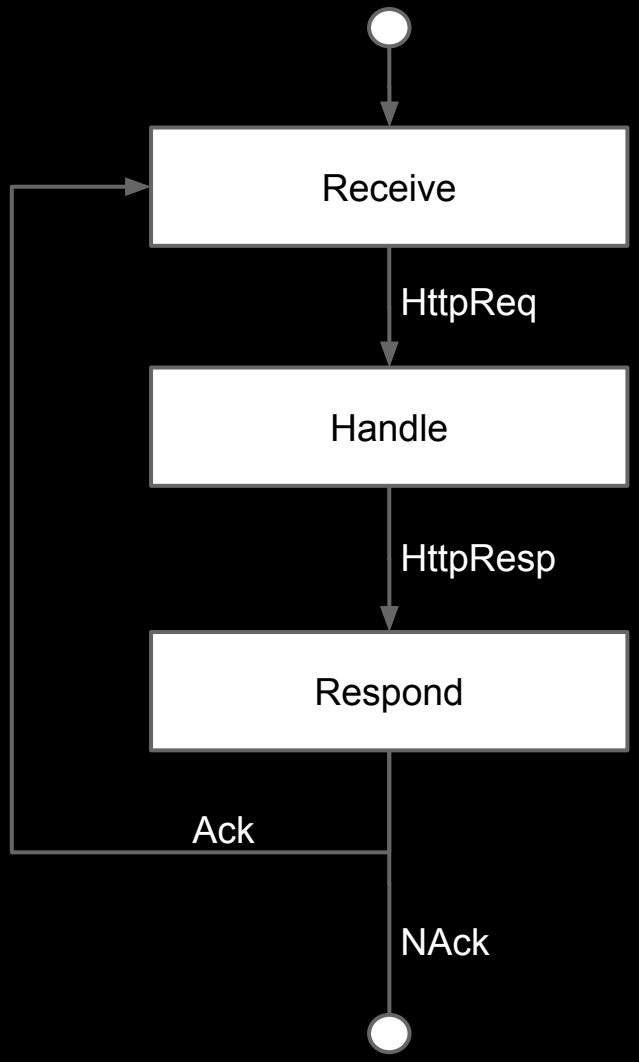
Less

- “imperative” code
- prog lang facilities dependent

More

- descriptive (data structures)
- nearer automata / continuous flow
- FP
- ?? composable / extensible / pluggable / reusable ??

State Automata



Anyone imagined “Runtime BPM”?

Remember
Runtime DI & BPM = Bad Taste

Quizz?

In FP, what helps us describe :

- a process / computation / dirty side-effect that can be continued

- a something that does something with things and produces other things that can be used to cause other somethings

In a safe & pure way ???


Monads guide you through the happy path !!!!!
(& to psychedelic t-shirts too)

Monadic App

```
val serve(): App[SendStatus] = for {
  req          <- receive()
  _            <- log(req)
  resp         <- handle(req)
  status       <- respond(resp)
  lastStatus   <- if(status == Ack) serve()
                  else stop(status)
} yield (lastStatus)
```

App as :

- data structure == process
- contained data == result

App as Algebraic Data Types

```
val serve(): App[SendStatus] = for {
  req          <- HttpReceive[HttpReq]()
  _            <- LogInfo[Unit](req)
  resp         <- HttpHandle[HttpResp](req)
  status       <- HttpRespond[SendStatus](resp)
  lastStatus   <- if(status == Ack) serve()
                  else HttpStop[SendStatus](status)
} yield (lastStatus)
```

Grammar based on Type Constructors F[A]

- F === instructions / computations
- A === effective payload / bound value

Algebraic Data Types

```
sealed trait HttpInteract[A]
case object HttpReceive extends HttpInteract[HttpReq]
case class HttpRespond(data: HttpResp) extends HttpInteract[SendStatus]
case class Stop(err:SendStatus) extends HttpInteract[SendStatus]
```

```
sealed trait HttpHandle[A]
case class HttpHandleResult(resp: HttpResp) extends HttpHandle
[HttpResp]
```

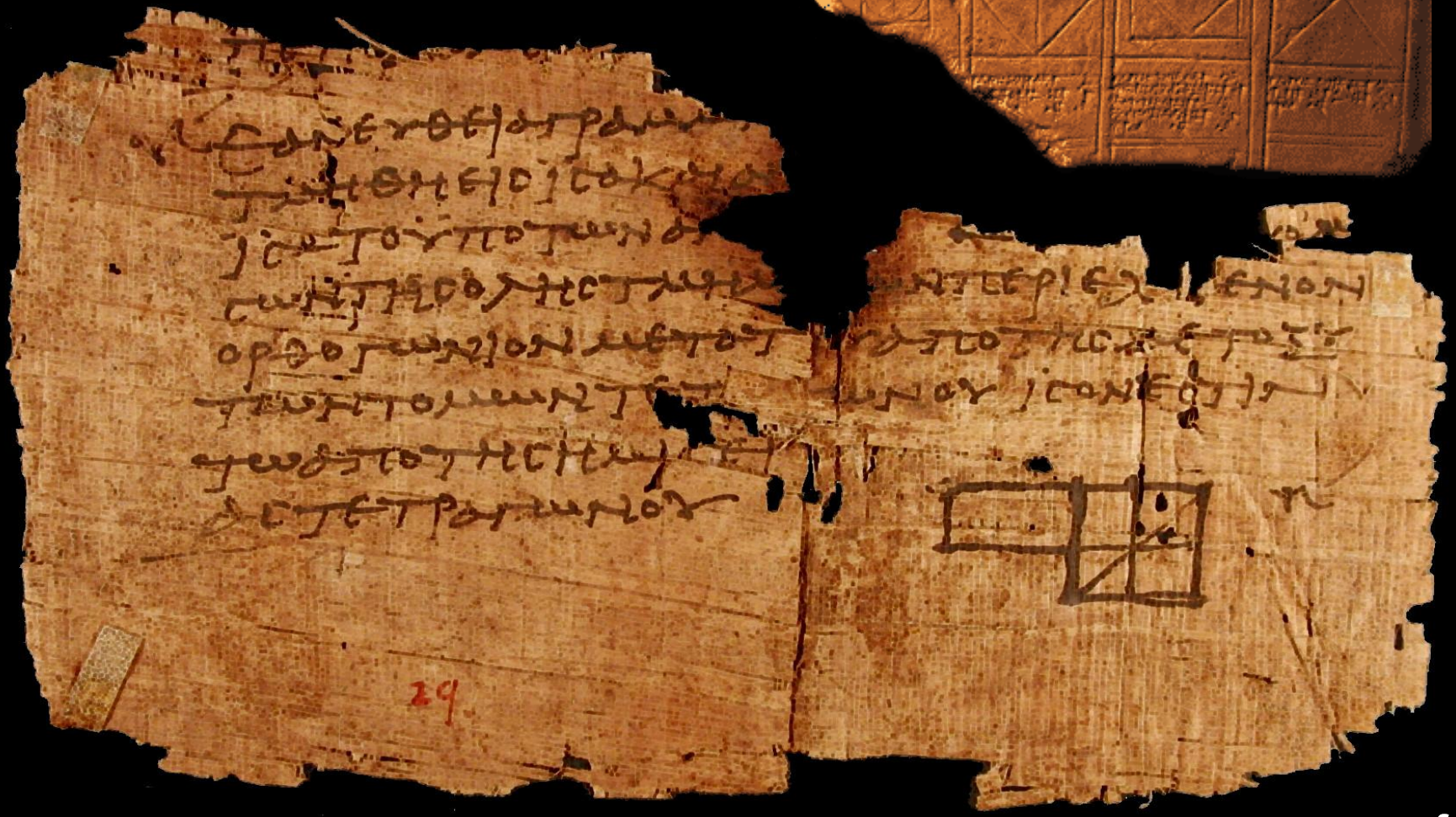
```
sealed trait Log[A] { val level: LogLevel }
case class LogInfo(msg: String, level = INFO) extends Log[Unit]
case class LogDebug(msg: String, level = DEBUG) extends Log[Unit]
```

Hey, your F[A] aren't Monads !

(& when I close the blinds, my t-shirt turns blue, nice side-effect isn't it?

Magic from Freemonads Archipelago

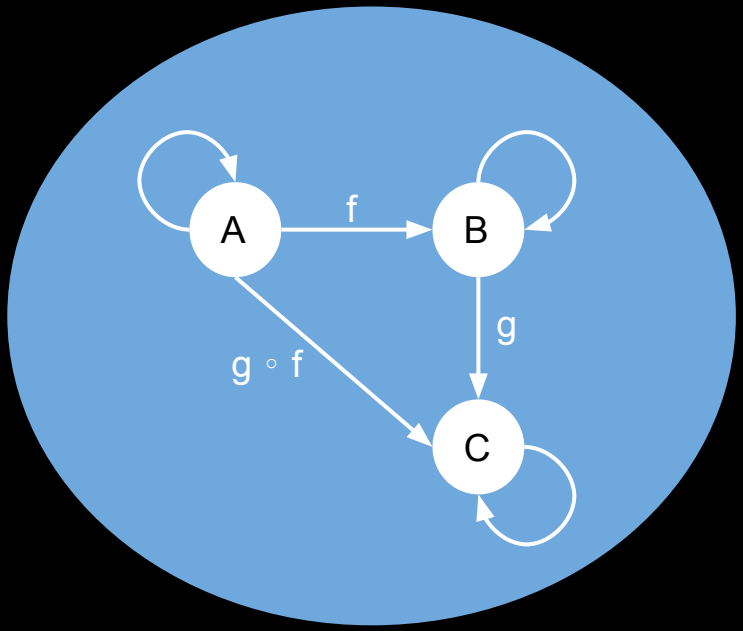
Math is The Only Magic



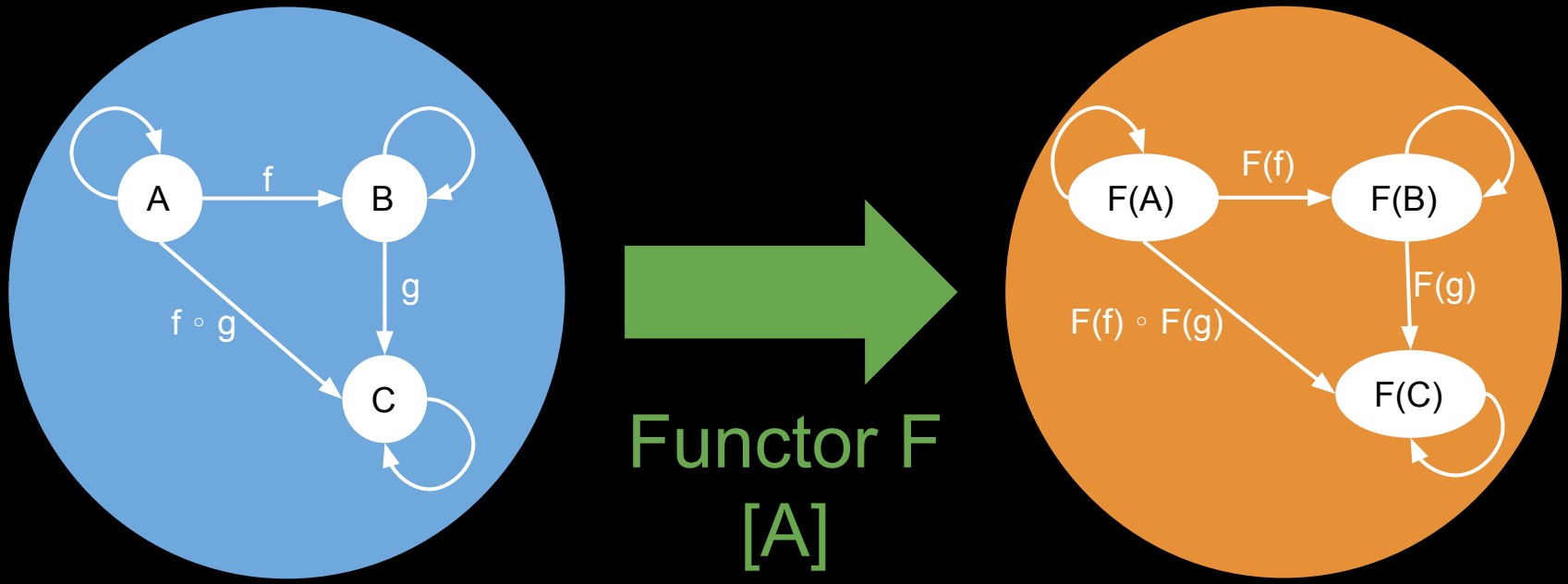
Algebra (Category Theory)

- structures/properties based on groups/sets of elements
- relations between elements
- NOT about nature of elements

Category

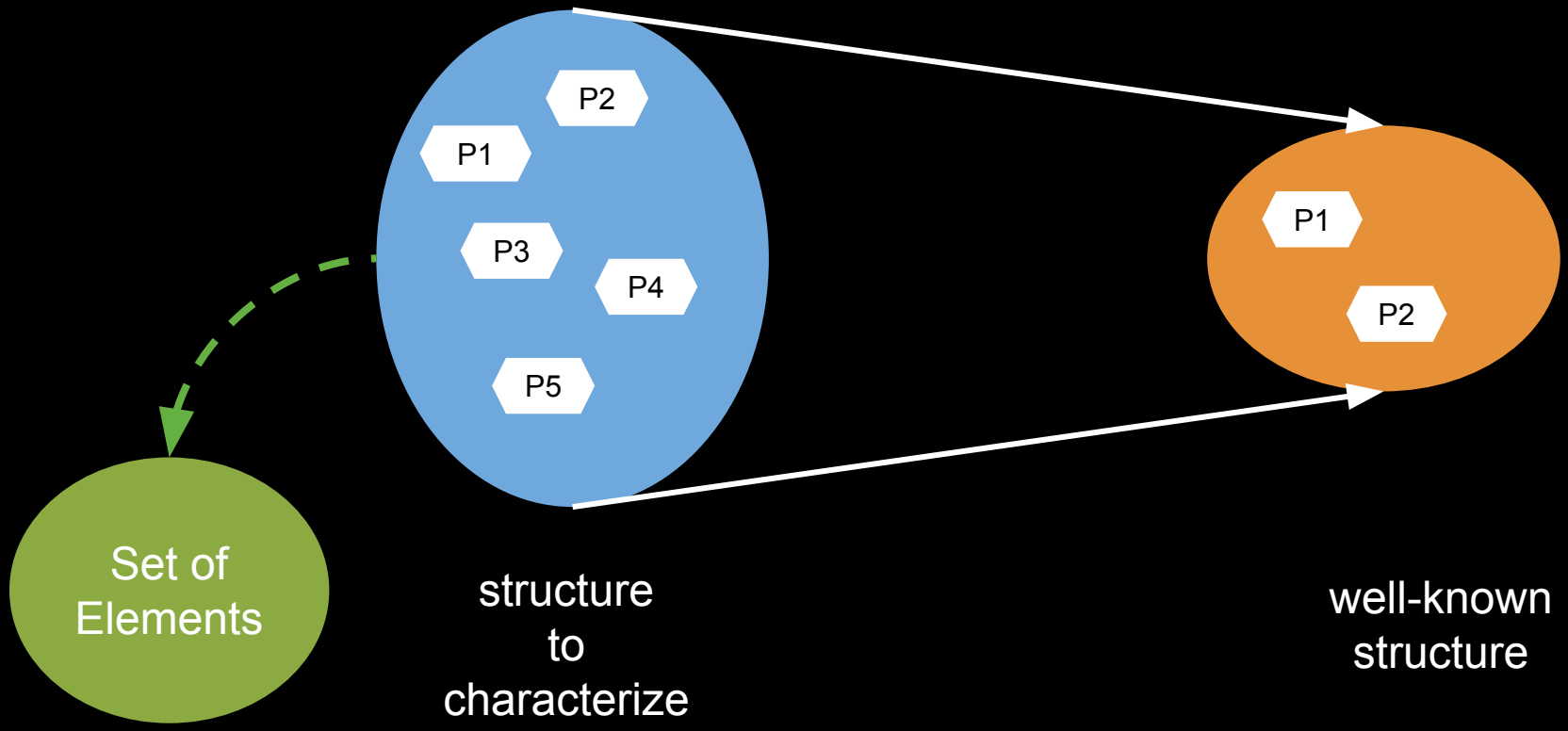


Functor

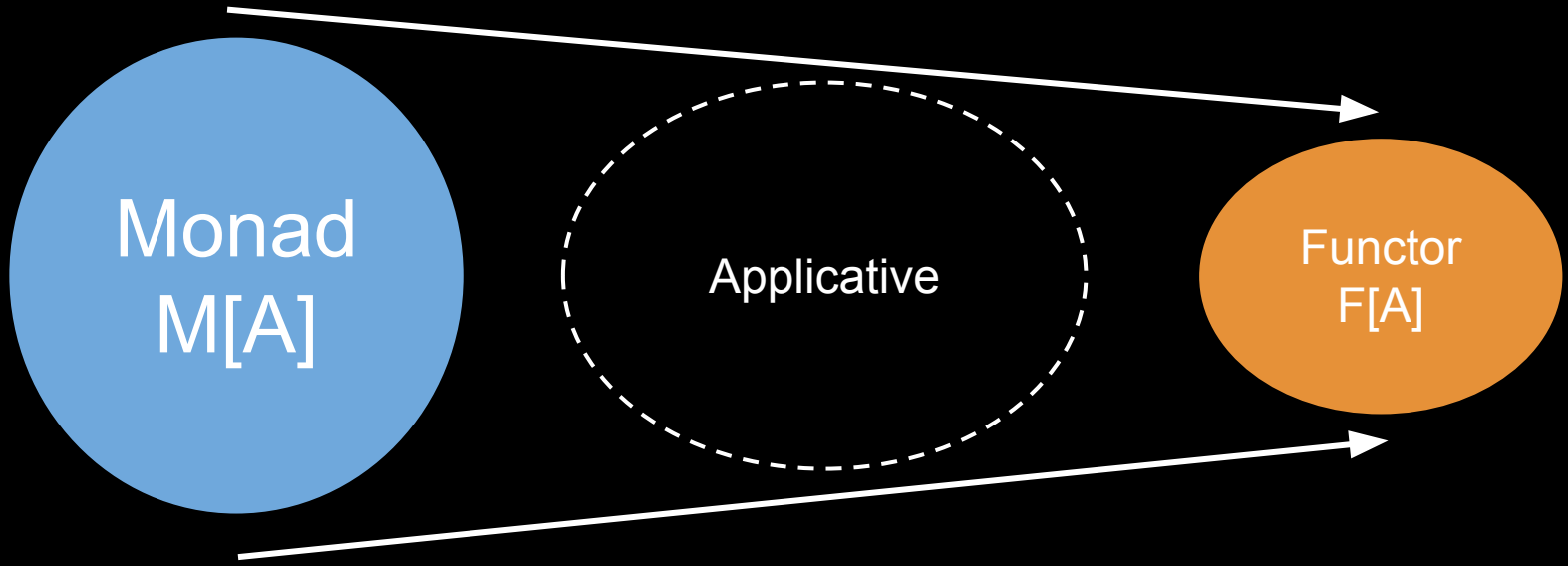


- Mapping that keeps structure
- “““Computation with post-processing”””

Algebraic structures



Reduce your structures



1/ point(a: A): M[A]

2/ flatMap(ma:M[A])(f: A =>M[B]): M[B]

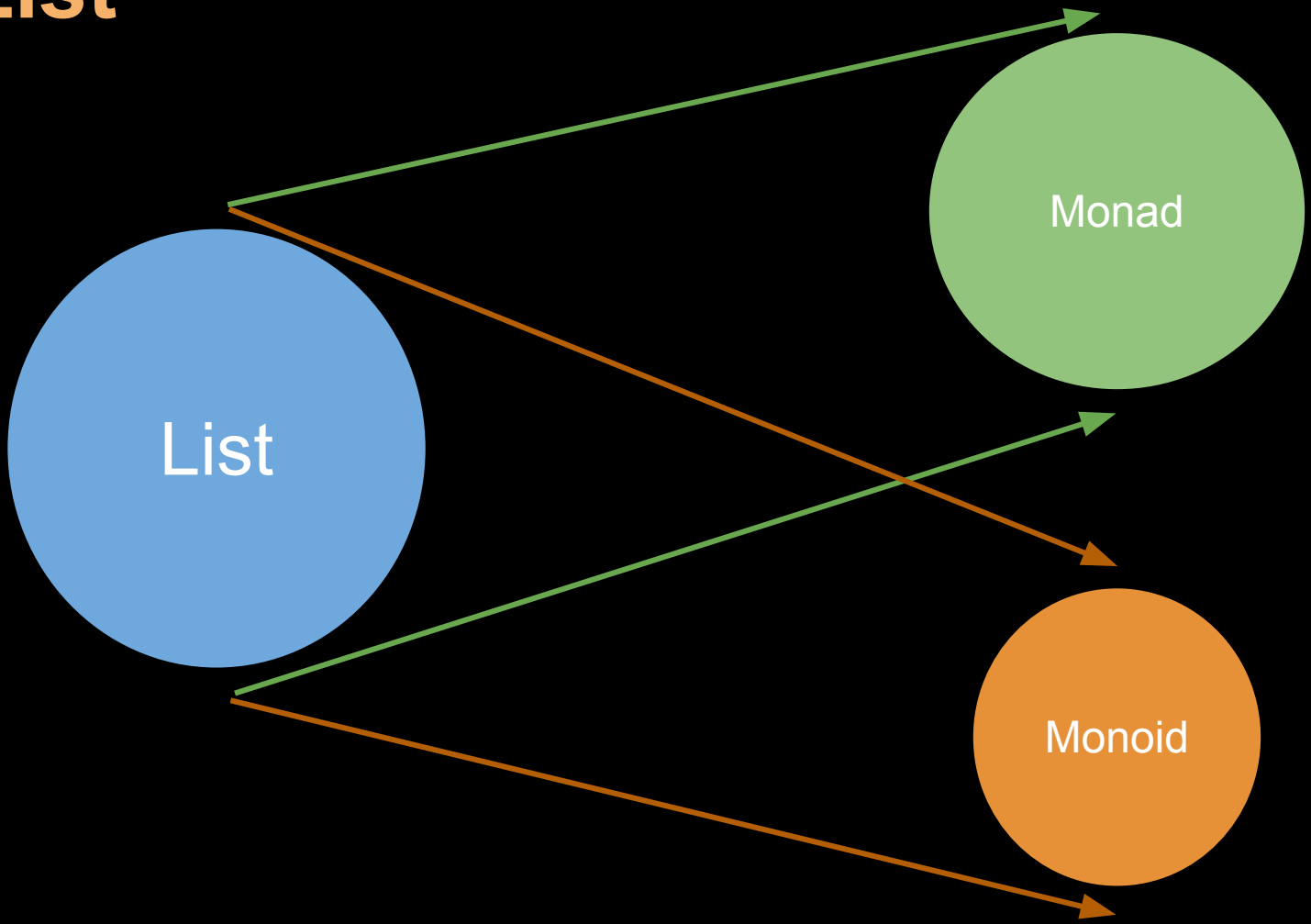
Computation with
Runtime Continuation

1/ map(fa:F[A])(f:A =>B): F[B]

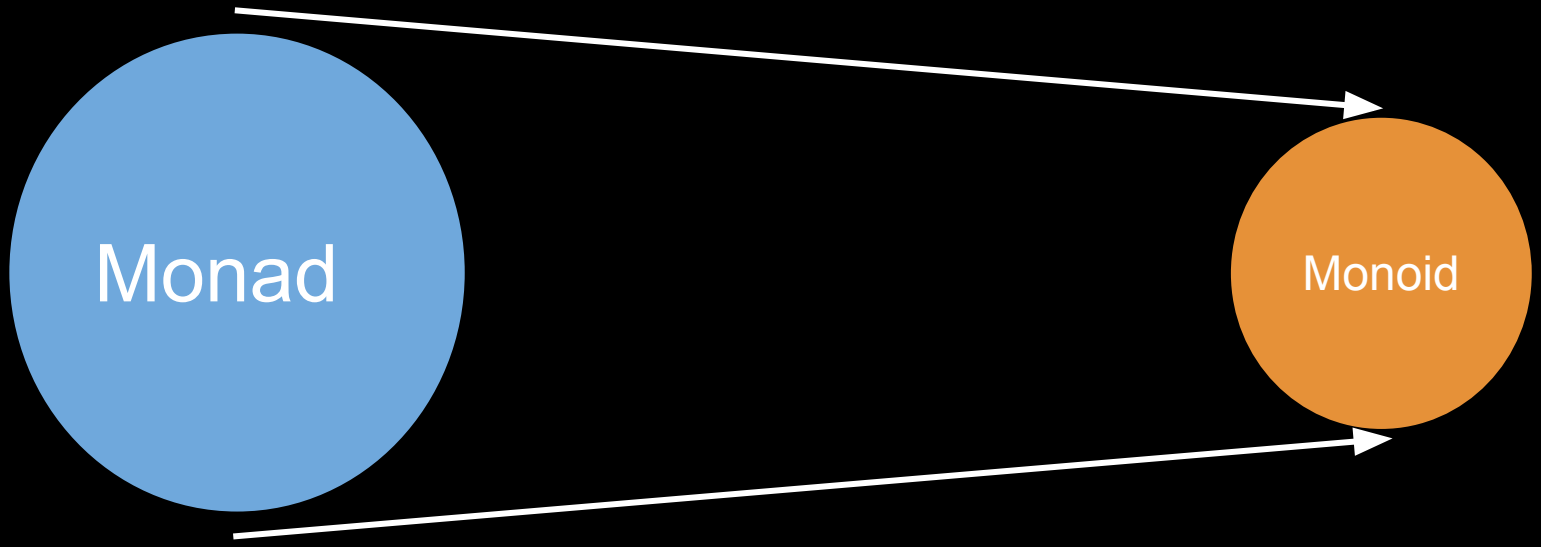
= flatMap(fa)(a => point(f(a))

Computation with
Post-processing

List

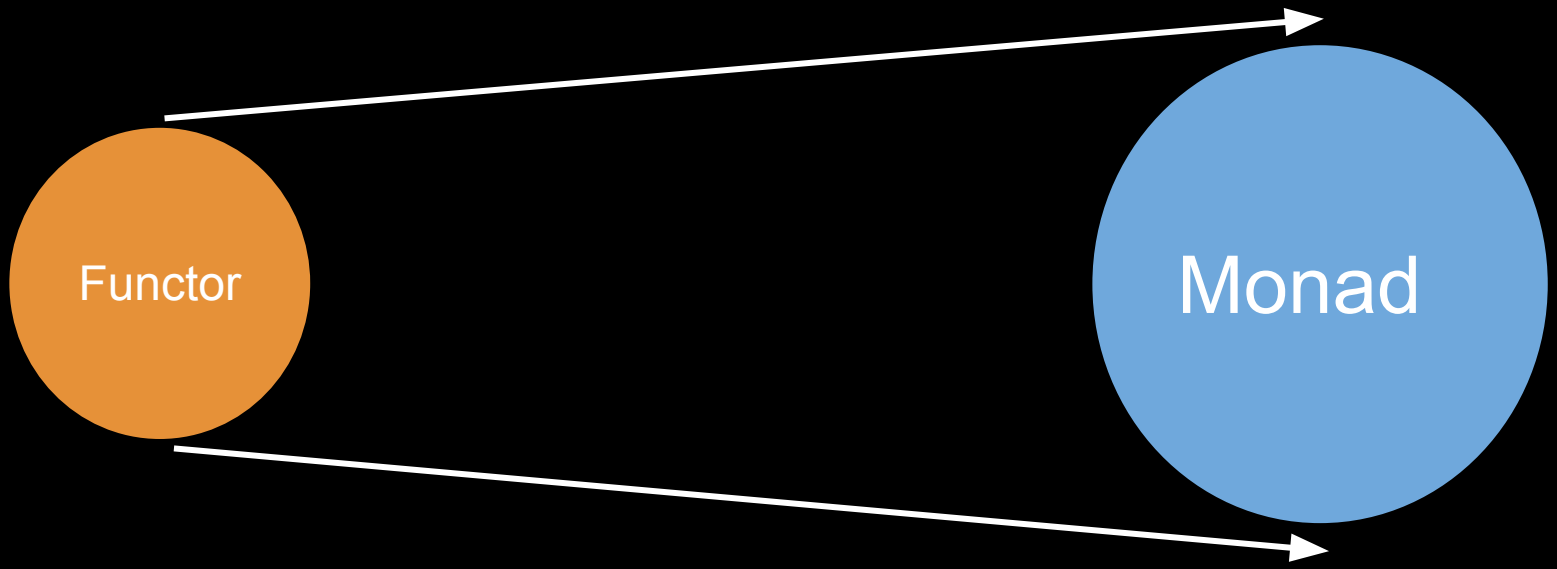


Delicatessen



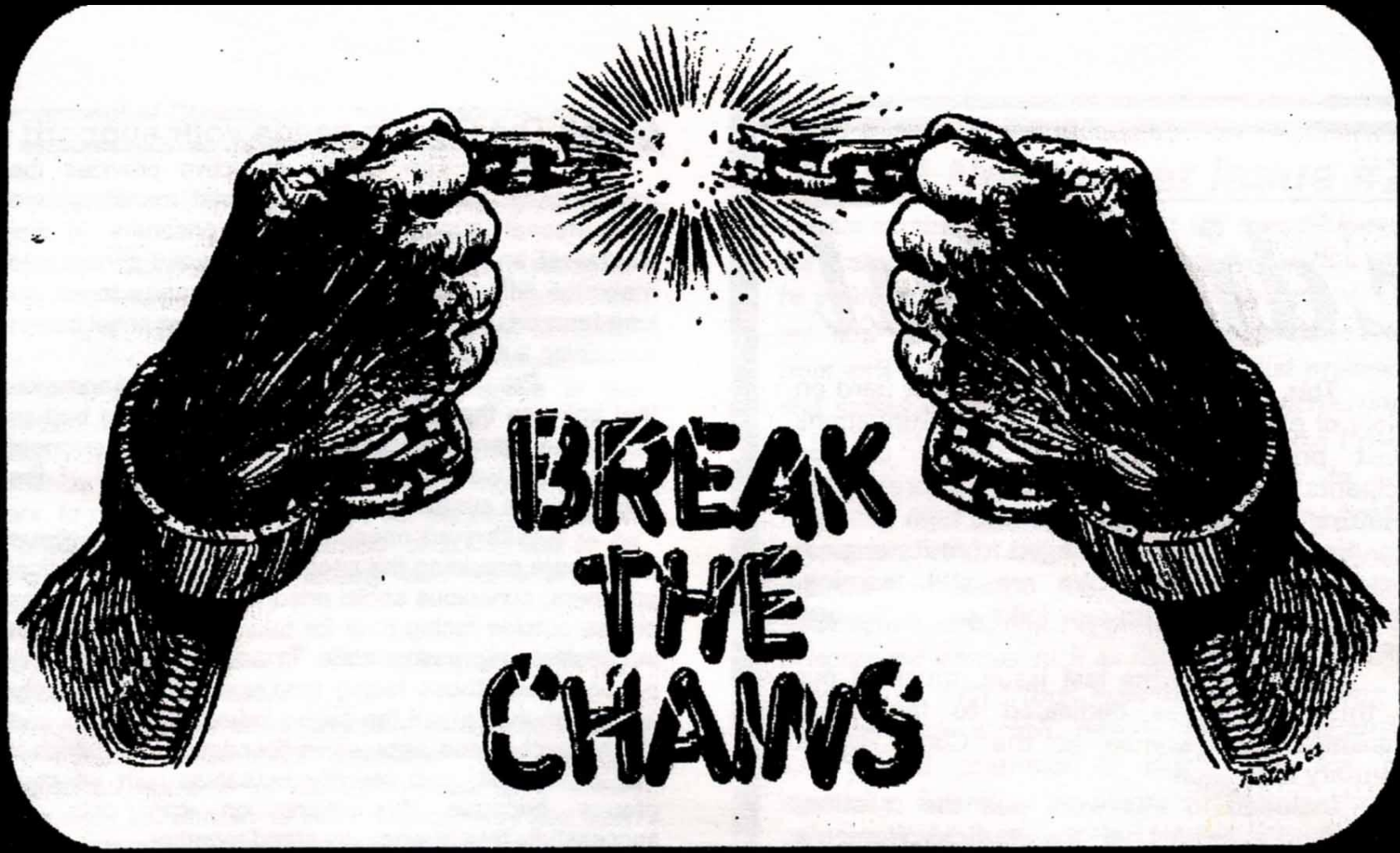
Monads are monoids in the category of endofunctors

Enlarge your structures

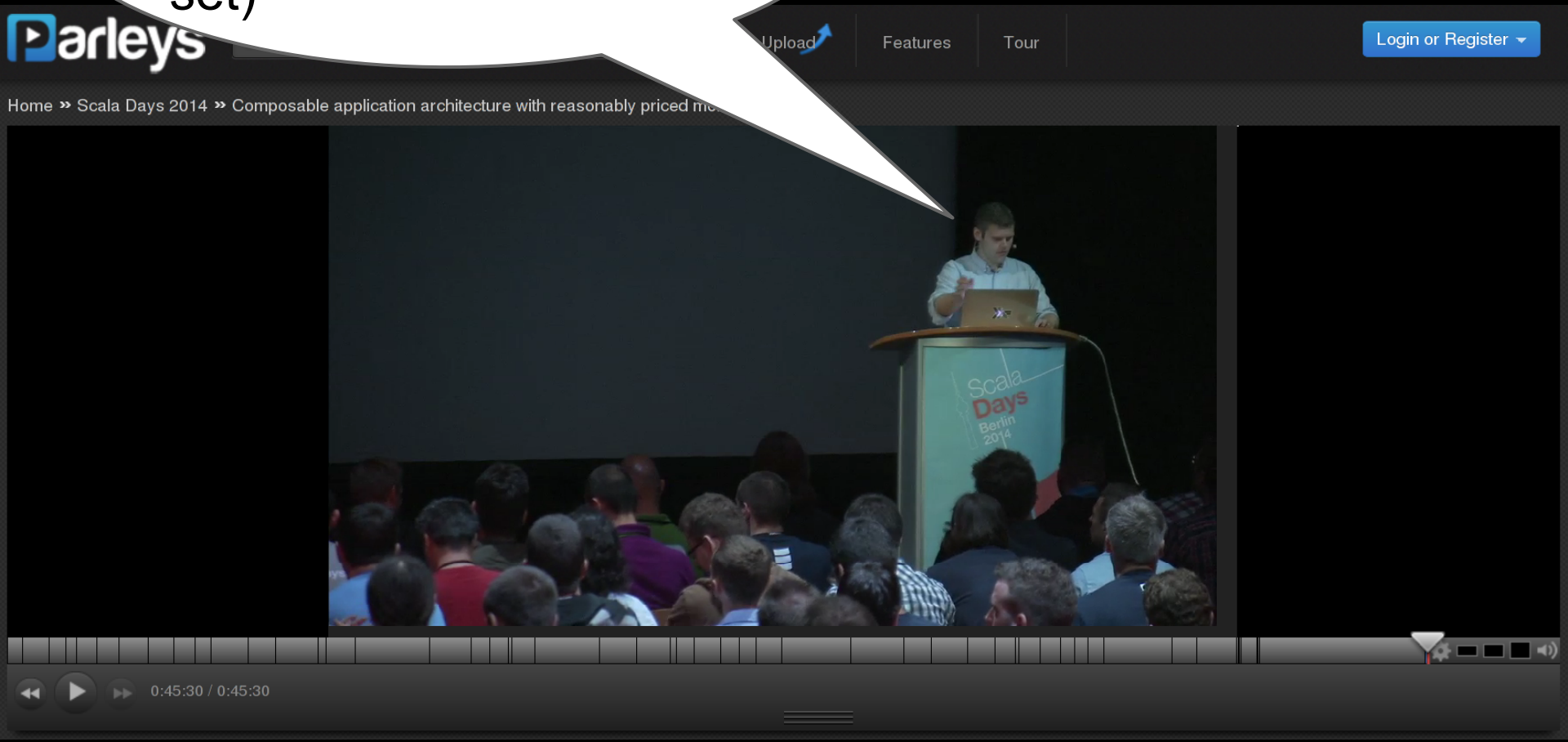


Idea = Build a Monad from a Functor

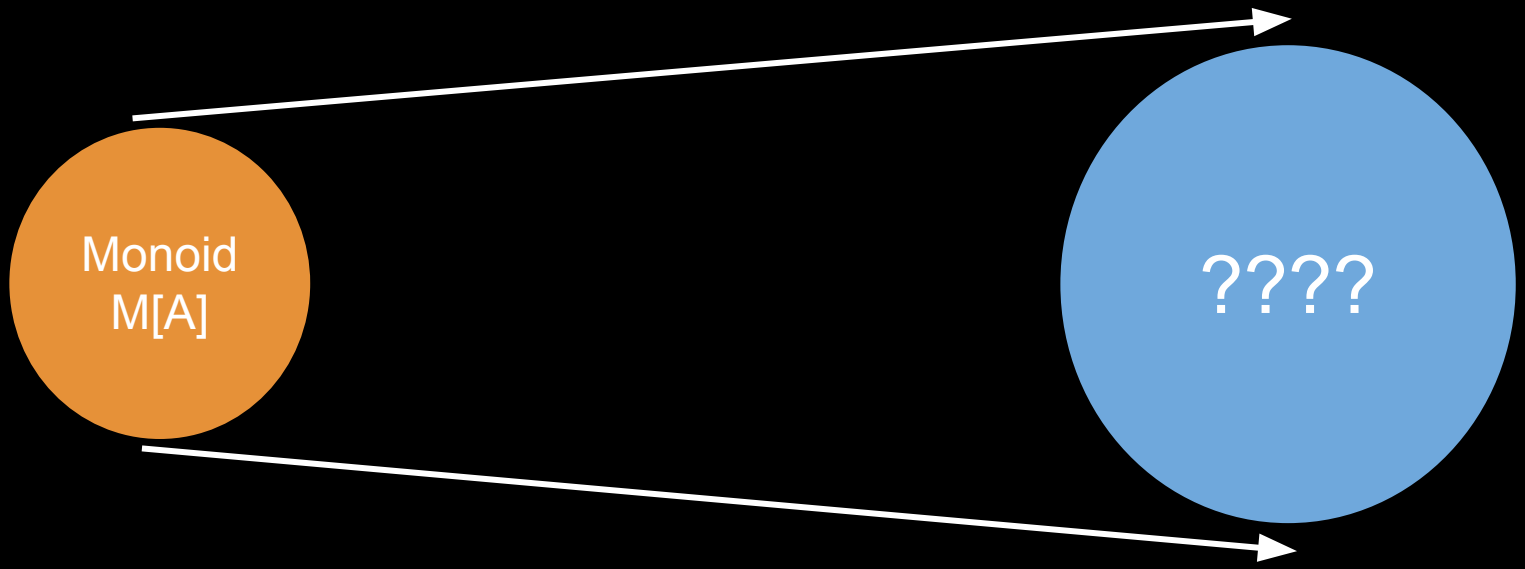
Free your structures



Free F is the simplest structure that has the properties of F (built on set)



Free Monoid (on a Set)

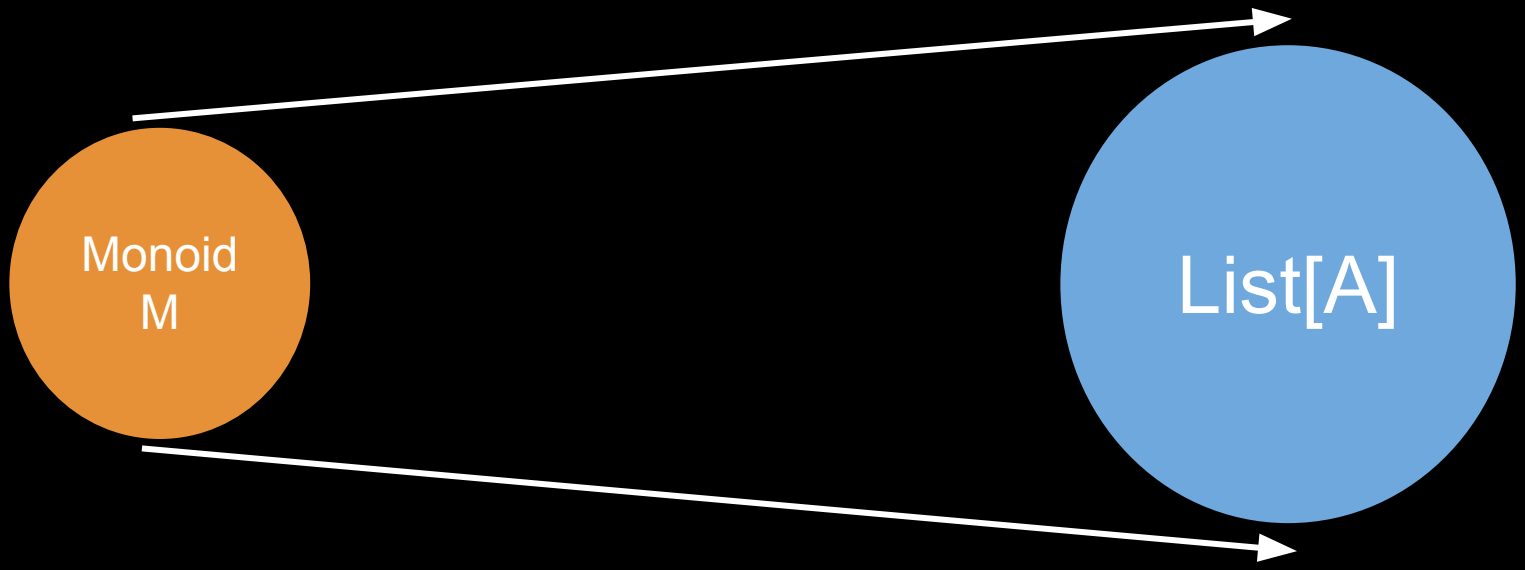


1/ zero: M

2/ append(m1: M, m2: M): M

What is the simplest structure that has the same properties for a set of elements?

Free Monoid (on a Set)



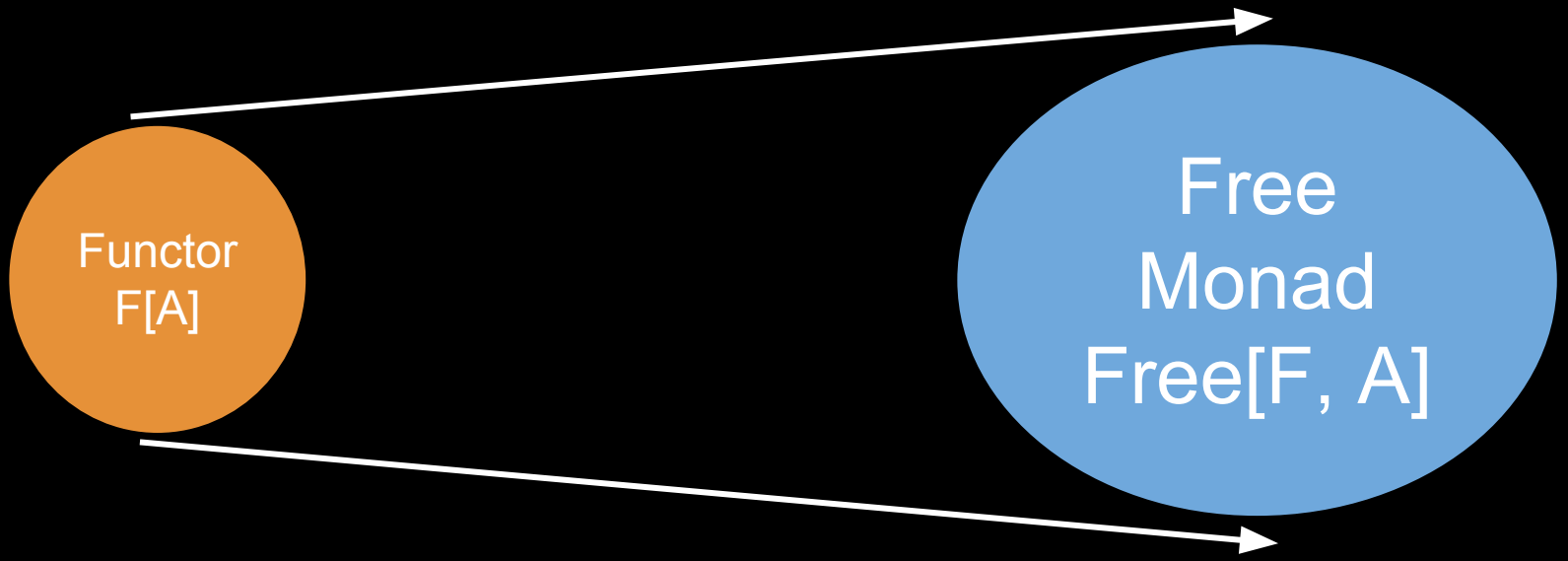
1/ zero: M

2/ append(m1: M, m2: M): M

1/ zero == Nil

2/ append == ++

Free Monad M[A] of Functor F[A]



Idea = build the simplest Monad M[A] from Functor F[A] by reifying point/flatMap using F

Free Monad M[A] of Functor F[A]

```
sealed abstract class Free[F[_], A]
```

Reifying Point

```
case class Return[F[_], A](a: A) extends Free[F, A]
```

```
def point[A](a: => A): Free[F, A] = Return(a)
```

Reifying FlatMap

// flatMap = (M[A], f) => M[B]
// reify >>>>> recursive data structure

F[A] *// use A as a carrier of recursion*

F[Free[F, A]] *// wrap that to make it Free[F, A]*

Wrap[F[Free[F, A]] extends *Free*[F, A]

```
case class Suspend[F[_], A](a: F[Free[F, A]])extends
Free[F, A]
```

Classic Free Monad representation

```
sealed abstract class Free[F[_], A]
case class Return[F[_], A](a: A) extends Free[F, A]
case class Suspend[F[_], A](a: F[Free[F, A]]) extends Free
[F, A]
```

- Recursive structure with an end

```
Suspend[ F[Suspend[ F[Suspend[ F[Return[A]] ] ] ] ] ]
```

- Continuation/Suspend conditioned by Functor

Classic Free Monad representation

```
implicit def freeMonad[S[_]: Functor] = new Monad[({ type l[A] = Free
[S, A] })#l] {

  def point[A](a: => A): Free[F, A] = Return(a)

  def flatMap[B](f: A => Free[F, B])(implicit F: Functor[F]): Free[F,
B] = this match {
    case Return(a)  => f(a)
    case Suspend(s) => Suspend(F.map(s){ free => free flatMap f })
  }


  def >>=[B](f: A => Free[F, B])(implicit F: Functor[F]) = flatMap(f)
}
```

- Continuation (>>=/bind) conditioned by Functor

Functor Samples

```
type F[A] = ()
Return(a)
Suspend(())

type Option[A] = Free[({ type l[x] = () })#l, A]
               = Free[[x] => (), A] (TypeLevel Scala)

//////////////

type F[A] = () => A
Suspend(() => Suspend(() => Suspend(() => Return(a))))

type Trampoline[A] = Free[[x] => (() => x), A]

// Stack recursion to heap recursion
// Stack Safety / Computation reified on Heap
```

Functor Samples

```
type F[I, A] = I => A
Suspend(i => Suspend(i => Suspend(i => Return(4)))))

type Iteratee[I, O] = Free[[x] => (I => x), O]



//////////////

type F[E, A] = (E, A)
Suspend( (1, Suspend( (2, Suspend( (3, Return(4)) ) ) ) ) )
)

type Source[E] = Free[[x] => (E, x), A]

/// LIST[E] / Free Monoid ???
```

Lift your Functors to FreeMonads

```
/** Suspends a value within a functor in a single step.
Monadic unit for a higher-order monad. */
def liftF[S[_], A](value: => S[A])(
    implicit S: Functor[S]
): Free[S, A] =
     Suspend(S.map(value)(Return[S, A]))
```

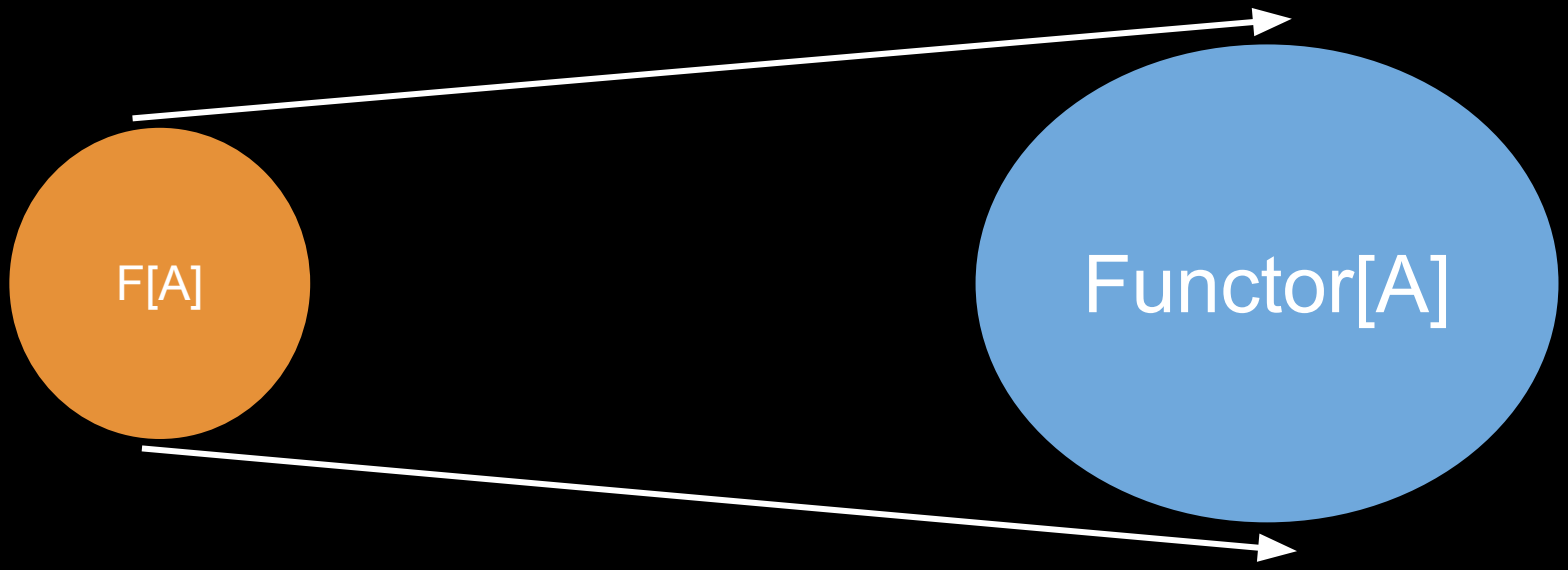
From any Functor F[A], we can obtain a Monad [A]

Back to sample

```
val serve(): App[SendStatus] = for {
  req          <- liftF(HttpReceive[HttpReq]())
  _            <- liftF(LogInfo[Unit](req.toString))
  resp         <- liftF(HttpHandle[HttpResp](req))
  status       <- liftF(HttpRespond[SendStatus](resp))
  lastStatus   <- if(status == Ack) liftF(serve())
                  else liftF(HttpStop[SendStatus](status))
} yield (lastStatus)
```

Hey, your F[A] aren't Functors!
(and your solution is as far away as the door behind me)

Functorize your *->* ?



1/ map(fa:F[A])(f:A =>B): M[B]

Coyoneda Lemma

"Any value F[A] can be 'destructured' into :

- *at least some type I*
- *a function I => A*
- *a value F[I]" (Mr Coyoneda)*

Dual of extremely important Yoneda lemma

Proof

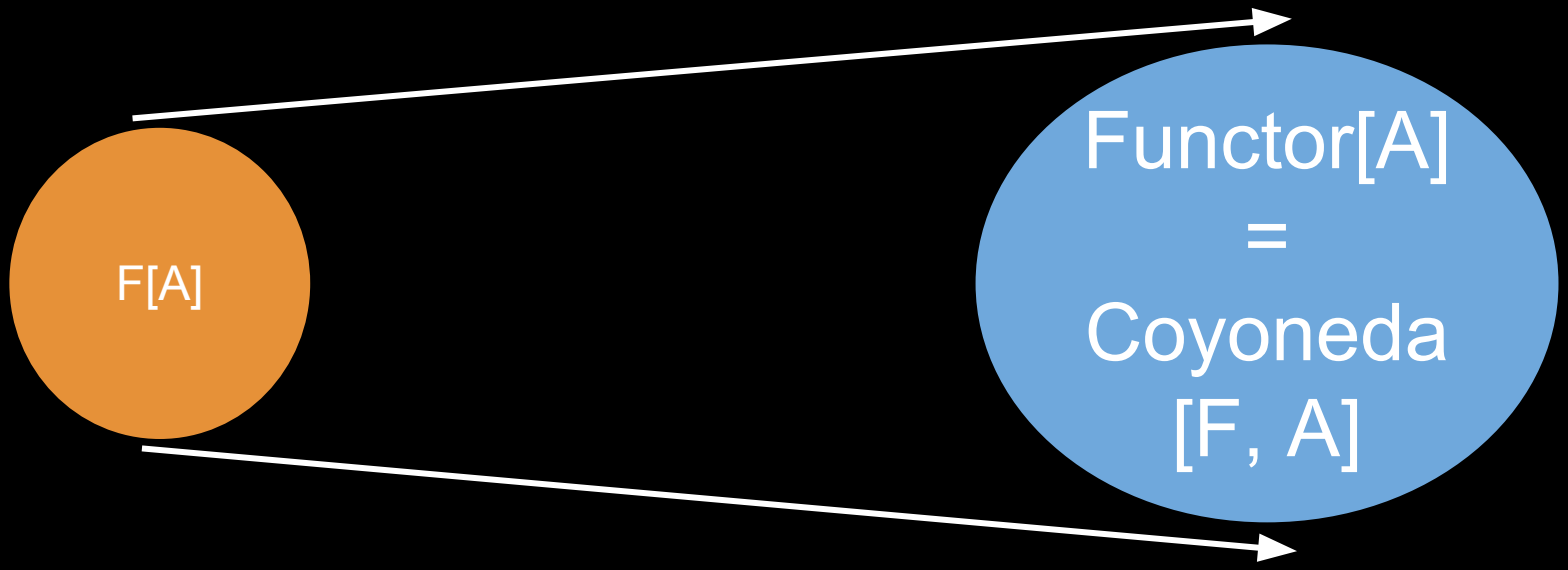
```
trait CoYoneda[F[_], A] {
  type I
  def f: I => A
  def fi: F[I]
}


def toCoYo[F[_],A](fa: F[A]) = new CoYoneda[F,A] {
  type I = A
  val f = (a: A) => a
  val fi = fa
}
```

Coyoneda[F[_], A] is a Functor

```
/** `Coyoneda[F,_]` is a functor for any `F` */
implicit def coyonedaFunctor[F[_]]: Functor[({type λ[α] = Coyoneda
[F, α]})#λ] =
  new Functor[({type λ[α] = Coyoneda[F,α]})#λ] {

    def map[A, B](ya: Coyoneda[F, A])(f: A => B) =
      new Coyoneda[F, B] {
        type I = ya.I
        val f = f compose ya.f // I => A => B == I => B
        val fi = ya.fi
      }
  }
```

Post-processing added to F[A]

Coyoneda is the simplest...



1/ map(fa:F[A])(f:A =>B): M[B]

Coyoneda is “Free Functor”

Lift your F[A] to FreeMonad

```
/** A free monad over a free functor of `F`. */
def liftFC[F[_], A](fa: F[A]):Free[[x]=>Coyoneda[F, x], A] =
    liftF(Coyoneda.toCoyo(fa))
```

Back to sample

```
val serve(): App[SendStatus] = for {
  req          <- liftFC(HttpReceive[HttpReq]())
  _            <- liftFC(LogInfo[Unit](req.toString))
  resp         <- liftFC(HttpHandle[HttpResp](req))
  status       <- liftFC(HttpRespond[SendStatus](resp))
  lastStatus   <- if(status == Ack) liftFC(serve())
                  else liftFC(HttpStop[SendStatus](status))
} yield (lastStatus)
```


Hey, I’m happy for you but your Monads aren’t the same in the for-comprehension

Homogenize Monads

```
type FreeC[S[_], A] = Free[[x] => Coyoneda[S, x], A]

val serve(): App[SendStatus] = for {
  req          <- FreeC[HttpInteract, HttpReq]
  _            <- FreeC[Log, Unit]
  resp         <- FreeC[HttpHandle, HttpResp]
  status       <- FreeC[HttpInteract, SendStatus]
  lastStatus   <- if(status == Ack) App[SendStatus]
                  else FreeC[HttpInteract, SendStatus]
} yield (lastStatus)
```

Homogenize Monads

```
type App[A] =
    FreeC[HttpInteract, A] or
    FreeC[Log, A] or
    FreeC[HttpHandle, A]


A : HttpReq => Unit => HttpResp => SendStatus
```

(Shapeless) Coproduct

```
type App[A] =
FreeC[HttpInteract, A] :+:
FreeC[Log, A] :+:
FreeC[HttpHandle, A] :+:
CNil

// Coproduct of FreeC is
// FreeC of Coproduct


// Grammar of instructions
type App[A] = HttpInteract[A] :+: Log[A] :+: HttpHandle
[A] :+: CNil
// Application flow
type FreeApp[A] = FreeC[App, A]
```

Coproduct Injection

```
val c1: App[HttpReq] =
        Coproduct[App](HttpReceive[HttpReq]())

val c2: App[HttpResp] =
        Coproduct[App](HttpHandle[HttpResp](req))

object Coproduct {
  def apply[C <: Coproduct, T](implicit inj: Inject[C, T]): C = inj(t)
                              // Inject proves C can be inserted
                              // into Coproduct C
}

trait Inject[C <: Coproduct, I] {
  def apply(i: I): C
}
```

Coproduct is simply math generalization of Dependency Injection

Lifting helpers

```
// APP DEFINITION
type App[A] = HttpInteract[A] :+: HttpHandle[A] :+: Log[A] :+: CNil
type FreeApp[A] = Free.FreeC[App, A]

// HELPERS
def lift[F[_], A](a: F[A])(implicit inj: Inject[App[A], F[A]]): FreeApp[A] =
Free.liftFC(Coproduct[App[A]](fa))

object HttpInteract {
  def receive() = lift(HttpReceive)
  def respond(data: HttpResp) = lift(HttpRespond(data))
  def stop(err: RecvError \/ SendStatus) = lift(Stop(err))
}

object Log {
  def info(msg: String)       = lift(LogMsg(InfoLevel, msg))
}

object HttpHandle {
  def result(resp: HttpResp) = lift(HttpHandleResult(resp))
}
```

Final Application

```
// Application
val serve(): FreeApp[SendStatus] = for {
  req         <- receive()
  _           <- log("Received: "+req.toString)
  resp        <- handle(req)
  status      <- respond(resp)
  lastStatus  <- if(status == Ack) serve() else stop(status)
} yield (lastStatus)


// Handle action
def handle(req: HttpReq): FreeApp[HttpResp] = req.url match {
  case "/foo" => result(HttpResp(status = Ok, body = e.toString))

  case _      => result(HttpResp(status = InternalServerError))
}
```

Victory !!!

Runnnnn (the app)!!!

(Plz note: in the future, they draw curves on whiteboards)

Runar's Free = FreeC[F, A]

```
sealed abstract class Free[F[_], A]

case class Return[F[_], A](a: A)
extends Free[F, A]

case class Suspend[F[_], A](a: F
[Free[F, A]]) extends Free[F, A]
```

```
trait CoYoneda[F[_], A] {
  type I
  def f: I => A
  def fi: F[I]
}
```

```
sealed trait Free[F[_],A]

case class Return[F[_],A](a: A)
  extends Free[F,A]

case class Bind[F[_],I,A](
  i: F[I],
  k: I => Free[F,A]) extends Free[F,A]
```

FreeC // List

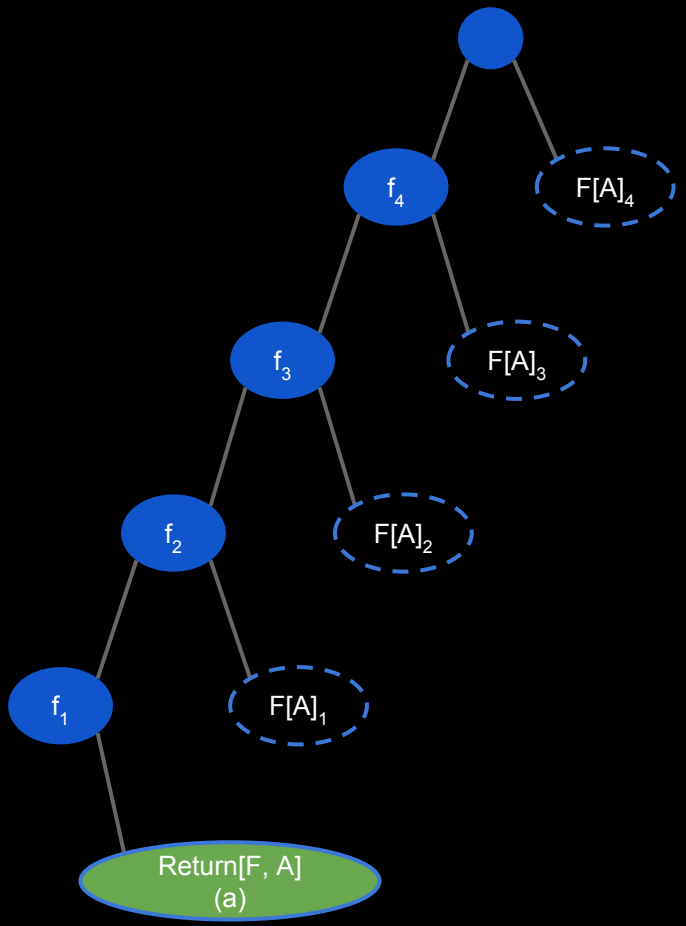
```scala
// FreeC[F[_], A]
sealed trait Free[F[_], A]

case class Return[F[_], A](a: A)
  extends Free[F, A]

case class Bind[F[_], A](
  i: F[I],
  f: I => Free[F, A]
) extends Free[F, A]
```

```scala
// FreeMonoid[A]
sealed trait List[A]

case class Nil
  extends List[Nothing]

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

FreeC ~ List (
  head = F[A],
  tail = generate next step / instruction
)

FreeC is a sequence of computations

```
val fi: (a:A) => Free[F, A] = ???
val free: Free[F, A] = Return(a) >>= f1 >>= f2 >>= f3 >>= f4
```



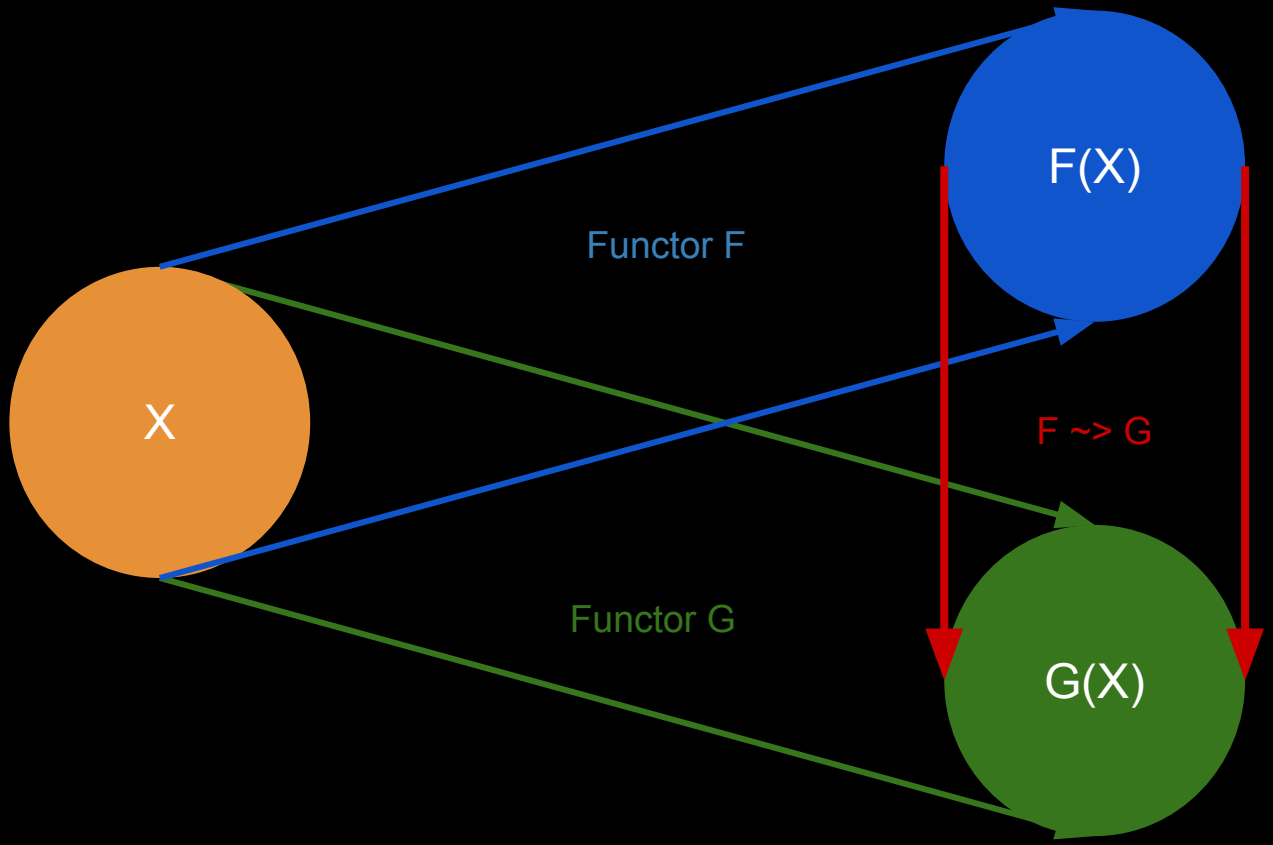
To obtain a single/final result, FOLD the Free monad as you fold a list

(aka Catamorphism)

Running Free ~ FoldR List

```
// Natural Transformation
trait ~>[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}


/**
 * Catamorphism for `Free`.
 * Runs to completion, mapping the suspension with the given transformation
at each step and
 * accumulating into the monad `M`.
 */
def foldMap[M[_]](f: S ~> M)(implicit S: Functor[S], M: Monad[M]): M[A] =
  this match {
    case Suspend(s) => Monad[M].bind(f(s))(_.foldMap(f))
    case Return(r) => Monad[M].pure(r)
  }
```

Natural Transformation



F[A] is programming grammar / language

F ~> G are interpreters

compiles language F to G

Mock (X ~> Id)

```
object Logger extends (Log ~> Id) {
  def apply[A](a: Log[A]) = a match {
    case LogMsg(lvl, msg) =>
      println(s"$lvl $msg")
  }
}

object HttpHandler extends (Http.HttpHandle ~> Id) {
  def apply[A](a: Http.HttpHandle[A]) = a match {
    case Http.HttpHandleResult(resp) =>
      println(s"Handling $resp")
      resp
  }
}

object HttpInteraction extends (Http.HttpInteract ~> Id) {
  def apply[A](a: Http.HttpInteract[A]) = a match {
    case Http.HttpReceive       => Http.GetReq("/foo")

    case Http.HttpRespond(resp) => Http.Ack

    case Http.Stop(err) => err
  }
}
```

(More realistic) X ~> Future

```
def Logger(implicit ctx: ExeCtx) = new (Log ~> Future) {
  def apply[A](a: Log[A]) = a match {
    case LogMsg(lvl, msg) =>
      Future { println(s"$lvl $msg") }
  }
}

def HttpHandler(implicit ctx: ExeCtx)= new (Http.HttpHandle ~> Future) {
  def apply[A](a: Http.HttpHandle[A]) = a match {
    case Http.HttpHandleResult(resp) =>
      Future {
        println(s"Handling $resp")
        resp
       }
  }
}

def HttpInteraction(implicit ctx: ExeCtx) = (Http.HttpInteract ~> Future) {
  def apply[A](a: Http.HttpInteract[A]) = a match {
    case Http.HttpReceive       => Future { Http.GetReq("/foo") }

    case Http.HttpRespond(resp) => Future { Http.Ack }

    case Http.Stop(err) => Future { err }
  }
}
```

Coproduct of NatTransf

```
// If we have
HttpInteract ~> F
HttpHandle ~> F
Log ~> F

// Then we have
HttpInteract :+: HttpHandle :+: Log :+: CNil ~> F
```

Run the app

```
val httpInterpreter: Http.App ~> Id =
  HttpInteractior ||: HttpHandler ||: Logger


val httpInterpreterCoyc: Http.CoyoApp ~> Id =
  liftCoyoLeft(httpInterpreter)


val lastStatus =
  Http.serve().foldMap(httpInterpreterCoyo)
```

Summary

Modules / Contracts	Algebraic Data Types + Coproduct
Flow Description	FreeMonad + Coyo
Mock/Test	Interpreters (NatTrans)

Reusability

```
// DB Interaction Application
object DB {
  // DB ADT
  type Entity = Map[String, String]

  sealed trait DBError
  case object NotFound extends DBError

  sealed trait DBInteract[A]
  case class FindById(id: String) extends DBInteract[DBError \/ Entity]
  case class Insert(...) extends DBInteract[...]

  // APP DEFINITION
  type App[A] = DBInteract[A] :+: Log[A] :+: CNil
  type CoyoApp[A] = Coyoneda[App, A]
  type FreeApp[A] = Free.FreeC[App, A]

  def findById(id: String): FreeApp[DBError \/ Entity] =
    for {
      _   <- Log.debug("Searching for entity id:" +id)
      res <- liftFC(FindById(id))
      _   <- Log.debug("Search result:" +res)
    } yield (res)
}
```

Reusability

```
type App[A] =
HttpInteract[A] :+: HttpHandle[A] :+: Log[A] :+: DB.FreeApp[A] :+: CNil

// Handle action
def handle(req: HttpReq): FreeApp[HttpResp] = req.url match {
  case "/foo" =>
    for {
      dbRes <-  liftFC( DB.findById("foo"))

      resp  <-  HttpHandle.result(
                 dbRes match {
                   case -\/(err) => HttpResp(status = InternalServerError)
                   case \/-(e)   => HttpResp(status = Ok, body = e.toString)
                 }
               )
    } yield (resp)

  case _ => HttpHandle.result(HttpResp(status = InternalServerError))
}
```

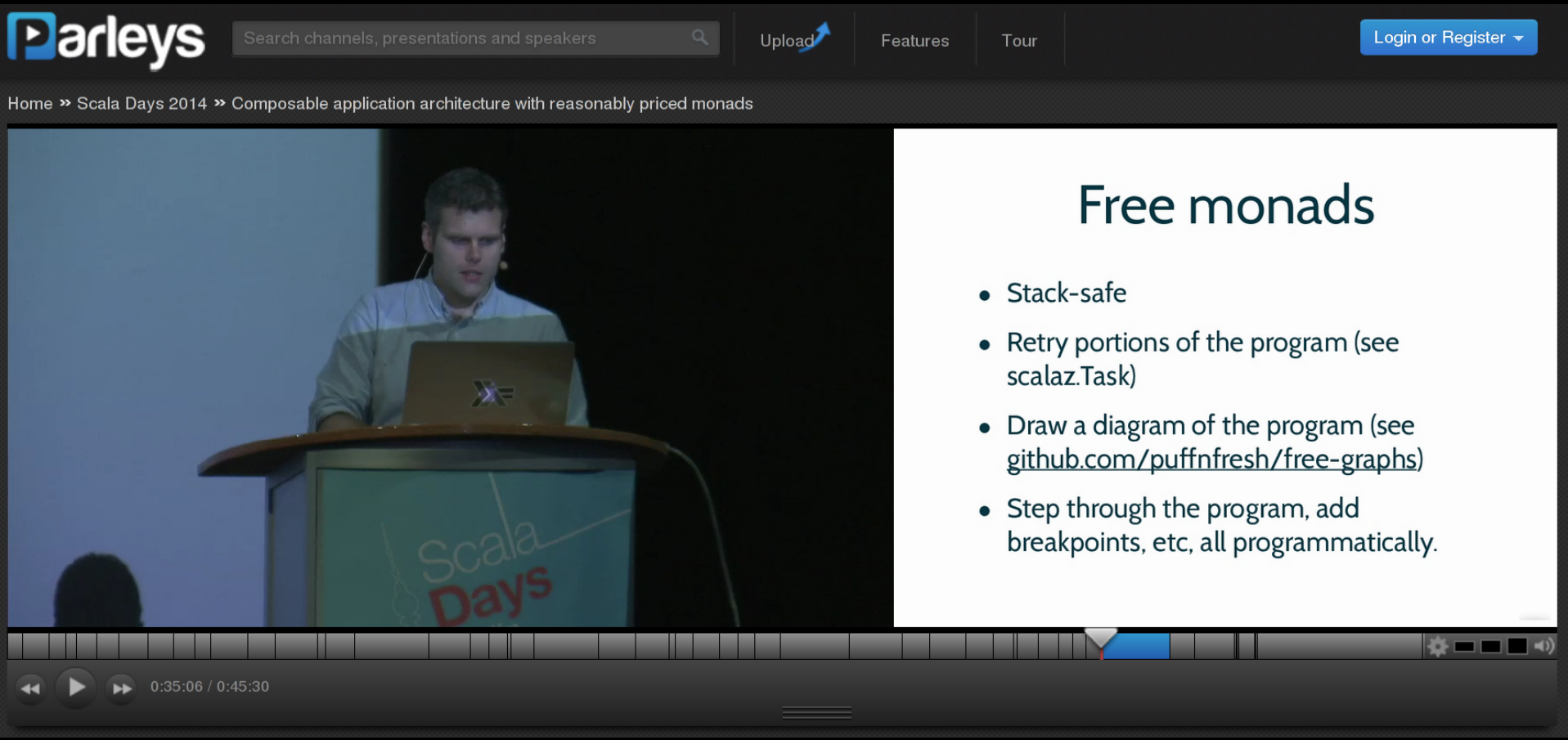
Reusability

```
val dbInterpreter: DB.App ~> Id = DBManager ||: Logger
val dbInterpreterCoyo: DB.CoyoApp ~> Id = liftCoyoLeft(dbInterpreter)
val dbInterpreterFree: DB.FreeApp ~> Id = liftFree(dbInterpreterCoyo)
```

```
val httpInterpreter: Http.App ~> Id =
  HttpInteraction ||: HttpHandler ||: Logger ||: dbInterpreterFree

val httpInterpreterCoyo: Http.CoyoApp ~> Id = liftCoyoLeft(httpInterpreter)
```

```
Http.serve().foldMap(httpInterpreterCoyo)
```

Other Ideas


Parleys
Search channels, presentations and speakers
Upload
Features
Tour
Login or Register
Home » Scala Days 2014 » Composable application architecture with reasonably priced monads
Free monads
• Stack-safe
• Retry portions of the program (see scalaz.Task)
• Draw a diagram of the program (see github.com/puffnfresh/free-graphs)
• Step through the program, add breakpoints, etc, all programmatically.
0:35:06 / 0:45:30

Other Ideas

- Free is a pure data structure that can be manipulated

- Chained Apps (App ~> FreeApp2) (pluggable?)

- specific FreeMonads folds

- Other representations of FreeMonads based on sequence ideas

Algebra

“from Arabic *al-jebr* meaning “reunion of broken parts” ”
(Wikipedia)

Scalac

“from Old Swiss-French meaning “Scala Type inference
has broken parts & it makes some developers crazy” ”

Dive into the great depths

Coming Soon...

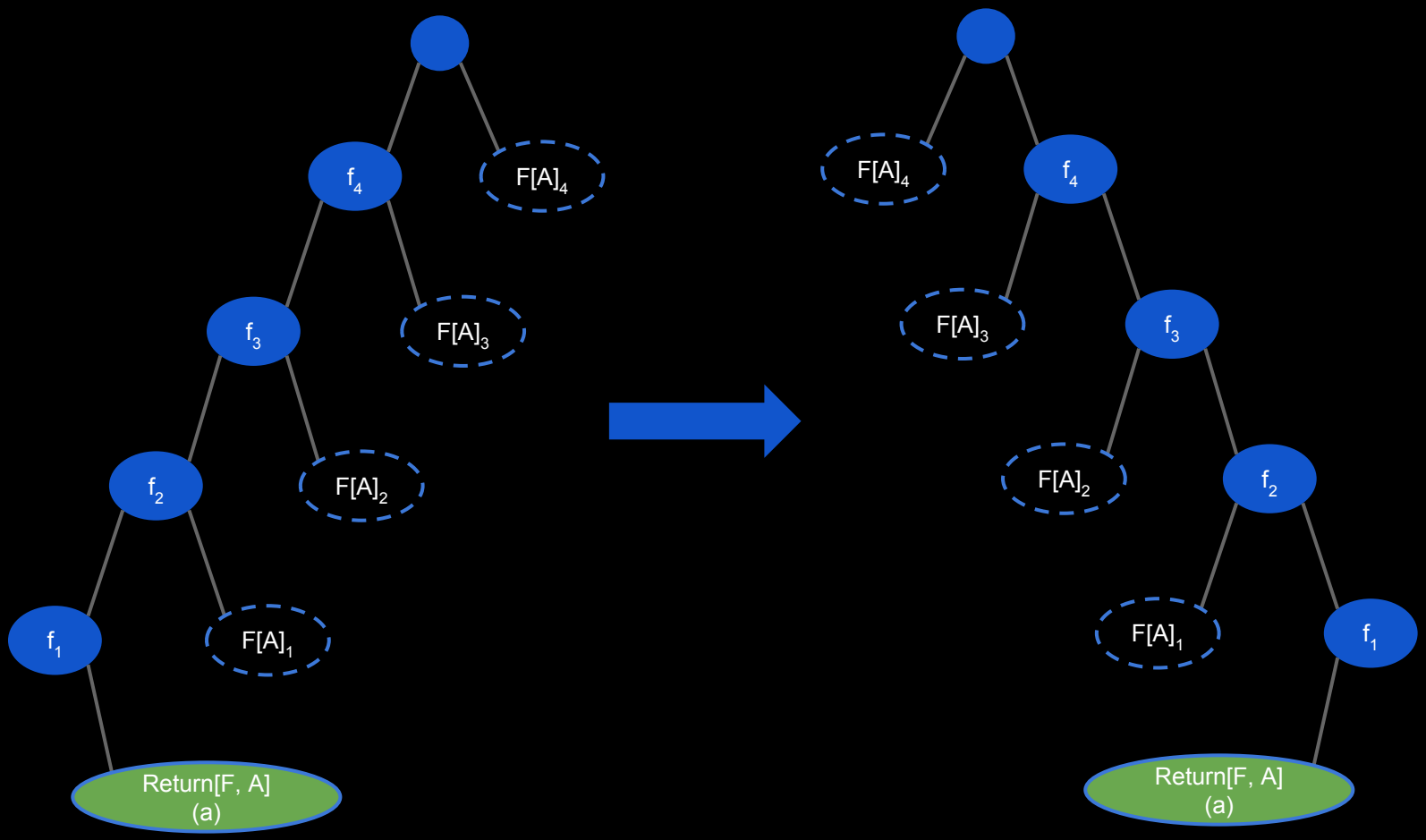
FreeMonad Performance issues

More complexity

```
(((a1 ++ a2) ++ a3) ++ .. ++ an))) =>
  a1 ++ (a2 ++ (a3 ++ (.. ++ an)))

(((a1 >>= a2) >>= a3) >>= .. >>= an))) =>
  a1 >>= (a2 >>= (a3 >>= (.. >>== an)))
```

$$\sum_{i=0}^{n-1} (n-1)|a_i| \approx \frac{n(n-1)}{2} \approx O(n^2)$$

More esoteric schemas

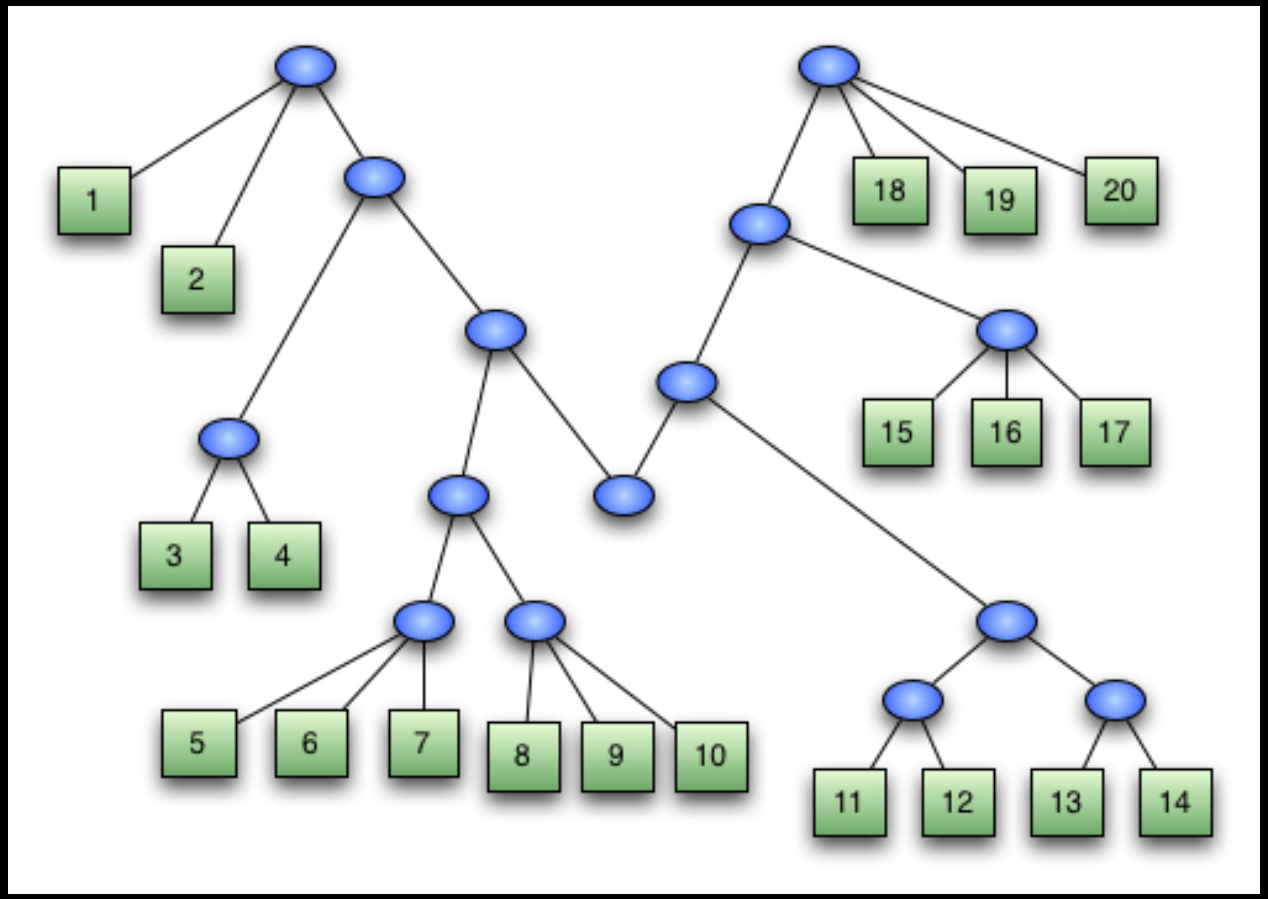


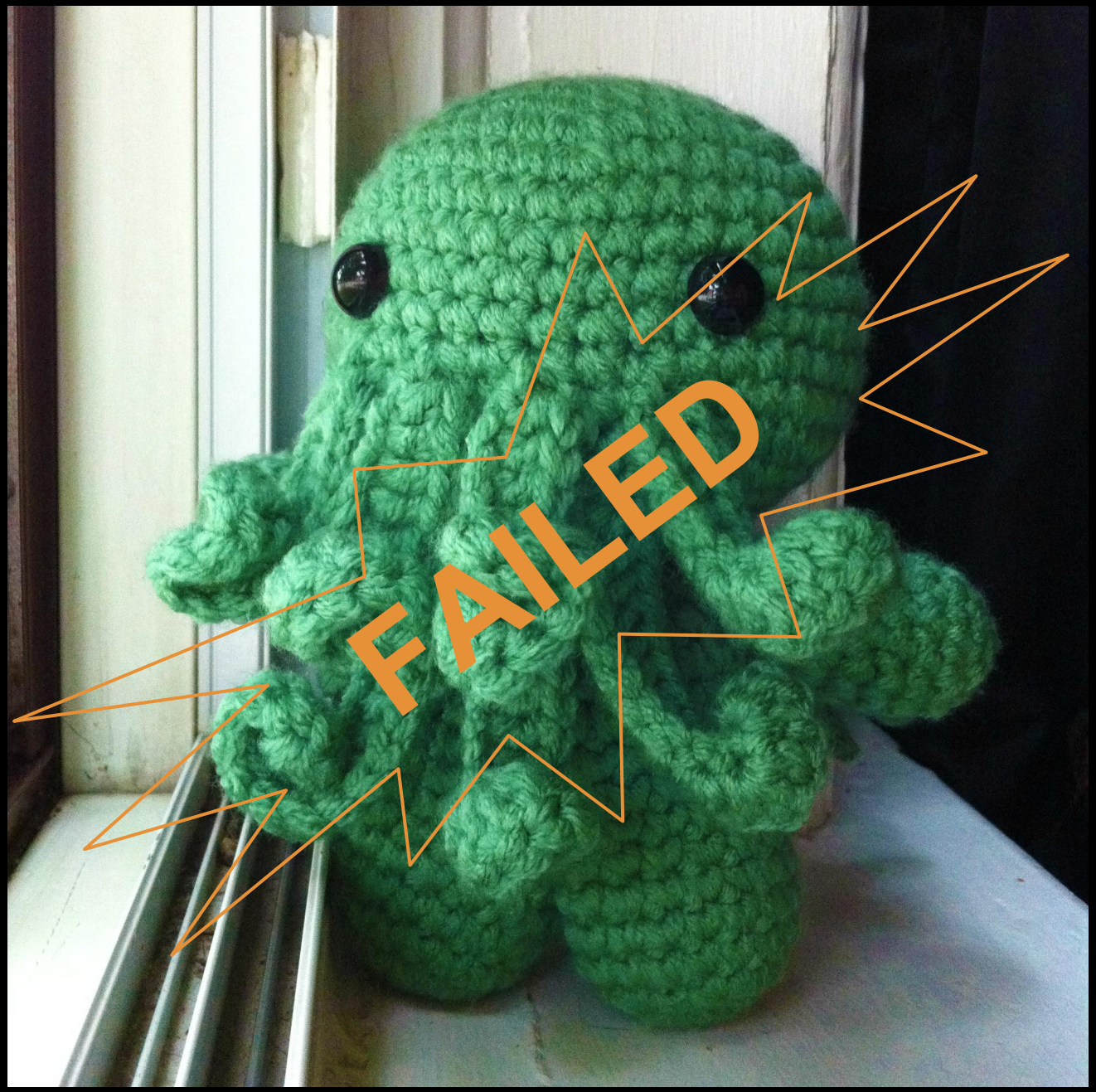
More Math Jargon

- Left/Right association
- Continuation-passing style
- Codensity
- Type-aligned structures

The Observation dilemma

FreeMonads from outer space




FAILED

To Be

Continued

Code @ https://github.com/mandubian/scalaio-2014

Thanks / Questions?