Algebras and Fixpoints:
Recursion Non-Recursively!

Who am I?

David Barri. *Hi !*

Github: `japgolly`
Twitter: `@japgolly`

Programming 28 years.
Scala & FP: 4 years. (14%)

What'cha workin' on?

- **ShipReq**: Software requirements startup.
- **Open Source**: github.com/japgolly
  - scalajs-react
  - scalajs-benchmark
  - ScalaCSS
  - Test State
  - Nyaya
  - univeq
  - microlibs

FP = Amazing

Great experience.
My startup & OSS is all FP.

Humbling. Can do so much with so little.

$S \Rightarrow (S, A)$

$$F[A] \Rightarrow A$$

Quick Foreward

- Who knows how to write a red-black tree?

Quick Foreward

- Who knows how to use `Map` or `Set`?
- Who finds they get benefit when using `Map` or `Set`?

Quick Foreward

Of the things I'll present tonight,

if you leave not understanding **how/why it works**, that's ok.
It might take a few attempts; explore later at your leisure.

More important to understand **how to use it** for your benefit.

Structure

1. When/where should I use this stuff?
2. Really awesome thing.
3. Really awesome thing backwards!
4. Two extensions.
5. Example & live demo.
6. Summary & resources.

Recursive Data

Often pops up in my experience.

What is it?

What are some examples?

Where is this talk applicable?

Recursive Data: ShipReq

- Use cases

```
1.0.   Create account.
1.0.1.   System prompts for username & password.
1.0.2.   User enters details.
1.0.2.a.   User says "der I'm a user"
1.0.2.b.   User mashes keyboard.
1.0.3.   System crashes cos actors have no type safety.
         Goto 1.0.1.
```

Recursive Data: ShipReq

- Filter expression

```
type != blah and (active || foo == bar)
```

Recursive Data: ShipReq

- Trie (Prefix tree)

  ```
  com.blah.cool_library
  com.blah.cool_library.dao
  com.blah.cool_library.lib
  com.blah.cool_library.util
  ```

  modeled as:

  ```
  com
    blah
      cool_library
        dao
        lib
        util
  ```

Recursive Data: ShipReq

- Self-referential many-to-many relationship.

  Focus on a node, get a recursive tree of children or parents.

Recursive Data

- **Nyaya (OSS):** properties

  eg. `P = A ∧ ¬B ∧ (C → D)`

- **Test-State (OSS):** actions, properties, assertions.

  eg. `getMilk = fridge.open >> take(milk) >>= drink >> fridge.close`

That's me just recently. Quite common.

Everyday Recursive Data Types

- Linked lists
- Maps, sets
- JSON

Why is recursive data useful?

- Models reality
    - FriendBook: *My friends have friends who have friends who have friends...*
    - File system: directories have directories which have directories...

Why is recursive data useful?

- Efficiency
  - Time - hashmap, binary search
  - Space - trie, interval trees

Typical Recursive Model

Calculator example.

```scala
sealed trait Calc
case class Number  (i: Int)             extends Calc
case class Add     (a: Calc, b: Calc) extends Calc
case class Multiply(a: Calc, b: Calc) extends Calc

// 1 + 2
Add(
  Number(1),
  Number(2))

// 3 * (1 + 2)
Multiply(
  Number(3),
  Add(
    Number(1),
    Number(2)))
```

Problem #1

No control over the recursion.
It's hardcoded to be always recursive, and always infinitely.
Can't abstract, define depth.

Problem #2

Recursion is usually hard. Often requires practice.

Non-recursion easier than recursion.

Problem #3

Can't annotate nodes.

Problem #3 (Node annotation)

No annotations

```scala
case class Person(name   : String,
                  age    : Int,
                  friends: List[Person])
```

Annotated with IDs

```scala
case class PersonId(value: Long) extends AnyVal

case class PersonWithId(id     : PersonId,
                        name   : String,
                        age    : Int,
                        friends: List[PersonWithId])
```

Problem #3 (Node annotation)

Calculator annotated with logging:

```
sealed abstract class Calc(val log: String)

case class Number  (i: Int)             extends Calc(i.toString)
case class Add     (a: Calc, b: Calc) extends Calc(s"(${a.log} + ${b.log})")
case class Multiply(a: Calc, b: Calc) extends Calc(s"(${a.log} * ${b.log})")
```

Problem #3 (Node annotation)

How would we allow ***any*** annotation to our calculator?

Problem #3 (Node annotation)

```scala
sealed abstract class Calc[A] {
  val ann: A
}

case class Number  [A](ann: A, i: Int)                 extends Calc[A]
case class Add     [A](ann: A, a: Calc[A], b: Calc[A]) extends Calc[A]
case class Multiply[A](ann: A, a: Calc[A], b: Calc[A]) extends Calc[A]

// 3 * (1 + 2)
Multiply("3 * (1 + 2)",
  Number("3", 3),
  Add("(1 + 2)",
    Number("1", 1),
    Number("2", 2)))
```

😓 😓

Is there a better way? 🤔

Step 1: Generalise the recursive-type.

Before:

```scala
sealed trait Calc
case class Number  (i: Int)            extends Calc
case class Add     (a: Calc, b: Calc) extends Calc
case class Multiply(a: Calc, b: Calc) extends Calc
```

After:

```scala
sealed trait Calc[A]
case class Number  [A](i: Int)     extends Calc[A]
case class Add     [A](a: A, b: A) extends Calc[A]
case class Multiply[A](a: A, b: A) extends Calc[A]
```

Wait...

```
val WHY_YOU_NO_WORK_?! : Calc[Calc] =
  Multiply(
    Number(3),
    Add(
      Number(1),
      Number(2)))
```

Type constructors

- `List`
- `Map`
- `Future`

Types

- `Int`
- `String`
- `Unit`

Type constructors

- `List`
- `Map`
- `Future`

Types

- `List[Int]`
- `Map[Int, String]`
- `Future[List[Int]]`

Type constructors

- `Calc`

Types

- `Calc[Unit]`
- `Calc[List[Int]]`
- `Calc[Nothing]`
- `Calc[Calc[Nothing]]`
- `Calc[Calc[Calc[Nothing]]]`
- `Calc[Calc[Calc[Calc[Nothing]]]]`
- `Calc[Calc[Calc[Calc[Calc[Nothing]]]]]`

This works but isn't nice.

```
val MaxDepthOf3: Calc[Calc[Calc[Nothing]]] =
  Number(456)
```

```
val MaxDepthOf3: Calc[Calc[Calc[Nothing]]] =
  Multiply(
    Number(3),
    Add(
      Number(1),
      Number(2)))
```

How do we get unlimited recursion back?

Step 2: Fixpoints!

```
final case class Fix[F[_]](unfix: F[Fix[F]])
```

```scala
final case class Fix[F[_]](unfix: F[Fix[F]])
```

```scala
   Fix[F] = F[ Fix[F] ]
//             ^^^^^^
          = F[F[ Fix[F] ]]
//               ^^^^^^
          = F[F[F[ Fix[F] ]]]
//                 ^^^^^^
```

Before:

```
val calc: Calc =
  Multiply(
    Number(3),
    Add(
      Number(1),
      Number(2)))
```

After:

```
val calc: Fix[Calc] =
  Fix(Multiply(
    Fix(Number(3)),
    Fix(Add(
      Fix(Number(1)),
      Fix(Number(2))))))
```

It goes back and forth.

```
def gimmeFix (c: Calc[Fix[Calc]]):      Fix[Calc]  = Fix[Calc](c)
def gimmeCalc(f:      Fix[Calc] ): Calc[Fix[Calc]] = f.unfix
```

```
    Fix[Calc]

        |  ↑
unfix   |  | Fix
        ↓  |

  Calc[Fix[Calc]]
```

ALL YOU NEED TO KNOW IS:

- Generalise (use an `A`) in place of a self-reference.
- Wrap your stuff in `Fix` when you want recursion.

Algebra

`F A -> A`

```
type Algebra[F[_], A] = F[A] => A
```

Scala has many ways of representing the same thing...

```scala
val listSumAlg: Algebra[List, Int] =
  _.sum

val listSumFn: List[Int] => Int =
  _.sum

def listSumMethod(list: List[Int]): Int =
  list.sum
```

They're all algebras.

```scala
listSumAlg   : Algebra[List, Int]
listSumMethod: Algebra[List, Int]
listSumFn    : Algebra[List, Int]
```

Ok...and?

A special function exists...

```
def awesome[F[_]: Functor, A](data: Fix[F], alg: Algebra[F, A]): A
```

```scala
def awesome[F[_]: Functor, A](data: Fix[F], alg: Algebra[F, A]): A
```

Algebra needs a `F[A]`.

No `A` in `Fix[F]`.

Like magic!

Let's try it out!

Well, first there's a prerequisite:

```
import scalaz.Functor

implicit val functor: Functor[Calc] =
  new Functor[Calc] {
    override def map[A, B](c: Calc[A])(f: A => B): Calc[B] = ???
  }
```

s/Calc/YourDataType/g

```scala
import scalaz.Functor

implicit val functor: Functor[Calc] =
  new Functor[Calc] {
    override def map[A, B](c: Calc[A])(f: A => B): Calc[B] =
```

```scala
      c match {
        case Number  (i)    => Number  (i)
        case Add     (a, b) => Add     (f(a), f(b))
        case Multiply(a, b) => Multiply(f(a), f(b))
      }
```

```scala
  }
```

Let's try it out!

```
val eval: Algebra[Calc, Int] = {
  case Number  (i)    => i
  case Add     (a, b) => a + b
  case Multiply(a, b) => a * b
}
```

```scala
def awesome[F[_]: Functor, A](data: Fix[F], alg : Algebra[F, A]): A

val eval: Algebra[Calc, Int] = {
  case Number  (i)    => i
  case Add     (a, b) => a + b
  case Multiply(a, b) => a * b
}
```

```scala
val c: Fix[Calc] = ...

awesome(c, eval) // returns 9
```

```scala
val explain: Algebra[Calc, String] = {
  case Number  (i)    => i.toString
  case Add     (a, b) => s"($a + $b)"
  case Multiply(a, b) => s"($a * $b)"
}
```

```scala
val explain: Algebra[Calc, String] = ...
val eval   : Algebra[Calc, Int]    = ...

val explainAndEval: Algebra[Calc, (String, Int)] =
  explain zip eval
```

```scala
val c: Fix[Calc] = ...

awesome(c, explainAndEval) // returns ("3 * (1 + 2)", 9)
```

Amazing!

Real name of `awesome` is catamorphism.
It goes to the ends of a tree, then calculates its way back up.
Bottom-up.

How does that work?!

```
    Fix[Calc]

       |  ↑
unfix  |  | Fix
       ↓  |

  Calc[Fix[Calc]]
```

```
                     cata = ???
    Fix[Calc]      - - - - - - - ->  A

       |  ↑
unfix  |  | Fix
       ↓  |

  Calc[Fix[Calc]]
```

```
                    cata = ???
    Fix[Calc]      - - - - - - - ->  A

         |  ↑
unfix    |  |  Fix
         ↓  |

  Calc[Fix[Calc]]  ---------------->  Calc[A]
                     calc.map(cata)
```

```
                      cata = ???
    Fix[Calc]      - - - - - - - ->  A

       |  ↑                          ↑
unfix  |  | Fix                      | alg: Algebra[Calc, A]
       ↓  |                          |

  Calc[Fix[Calc]]  ---------------->  Calc[A]
                     calc.map(cata)
```

```
                             cata
    Fix[Calc]      - - - - - - - ->  A

        |  ↑                         ↑
unfix   |  | Fix                     | alg: Algebra[Calc, A]
        ↓  |                         |

  Calc[Fix[Calc]]  --------------->  Calc[A]
                    calc.map(cata)
```

```
def cata: Fix[Calc] => A =
  fix => {
    val calc : Calc[Fix[Calc]] = fix.unfix
    val calcA: Calc[A]         = calc.map(cata)
    val a    : A               = alg(calcA)
    a
  }
```

```
                          cata
    Fix[Calc]      - - - - - - - ->  A

       |  ↑                          ↑
unfix  |  | Fix                      | alg
       ↓  |                          |

  Calc[Fix[Calc]]  ---------------->  Calc[A]
                    calc.map(cata)
```

```
def cata[F[_], A](alg: Algebra[F, A])(data: Fix[F])
                 (implicit F: Functor[F]): A =
  alg(F.map(f.unfix)(cata(alg)))
```

```
                        cata
    Fix[Calc]      - - - - - - - ->  A

      |  ↑                           ↑
unfix |  | Fix                       | alg
      ↓  |                           |

  Calc[Fix[Calc]]  --------------->  Calc[A]
                   fmap (cata alg)
```

```haskell
cata :: Functor f => (Algebra f a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unfix
```

ALL YOU NEED TO KNOW IS:

- Create a functor.
- Create an algebra when you want to fold/reduce a tree.
- Call `cata`. *Done!*

Next up: CoAlgebras

Everything in category theory can go backwards - this is called a dual.

Algebra

- `F[A] => A`

Coalgebra

- `A => F[A]`

Algebra

- `F[A] => A`
- *squash* a structure (`F`) into a single value (`A`)

Coalgebra

- `A => F[A]`
- *expand* a single value (`A`) into a structure (`F`)

Algebra

- `F[A] => A`
- *fold* a structure (`F`) into a single value (`A`)

Coalgebra

- `A => F[A]`
- *unfold* a single value (`A`) into a structure (`F`)

```scala
type CoAlgebra[F[_], A] = A => F[A]
```

```scala
val factors: CoAlgebra[Calc, Int] = i =>
  if (i > 2 && i % 2 == 0)
    Multiply(2, i / 2)
  else
    Number(i)
```

Another magic function

```scala
def emosewa[F[_]: Functor, A](data: A, alg: CoAlgebra[F, A]): Fix[F]
```

Notice, there's no `A` in the result.

Real name is anamorphism.

Dual of catamorphism.

```
cata: Fix f -> (f a -> a  ) -> a
ana : a     -> (a   -> f a) -> Fix f
```

It starts with the at the root, then calculates its way down to the nodes until complete.

Top-down.

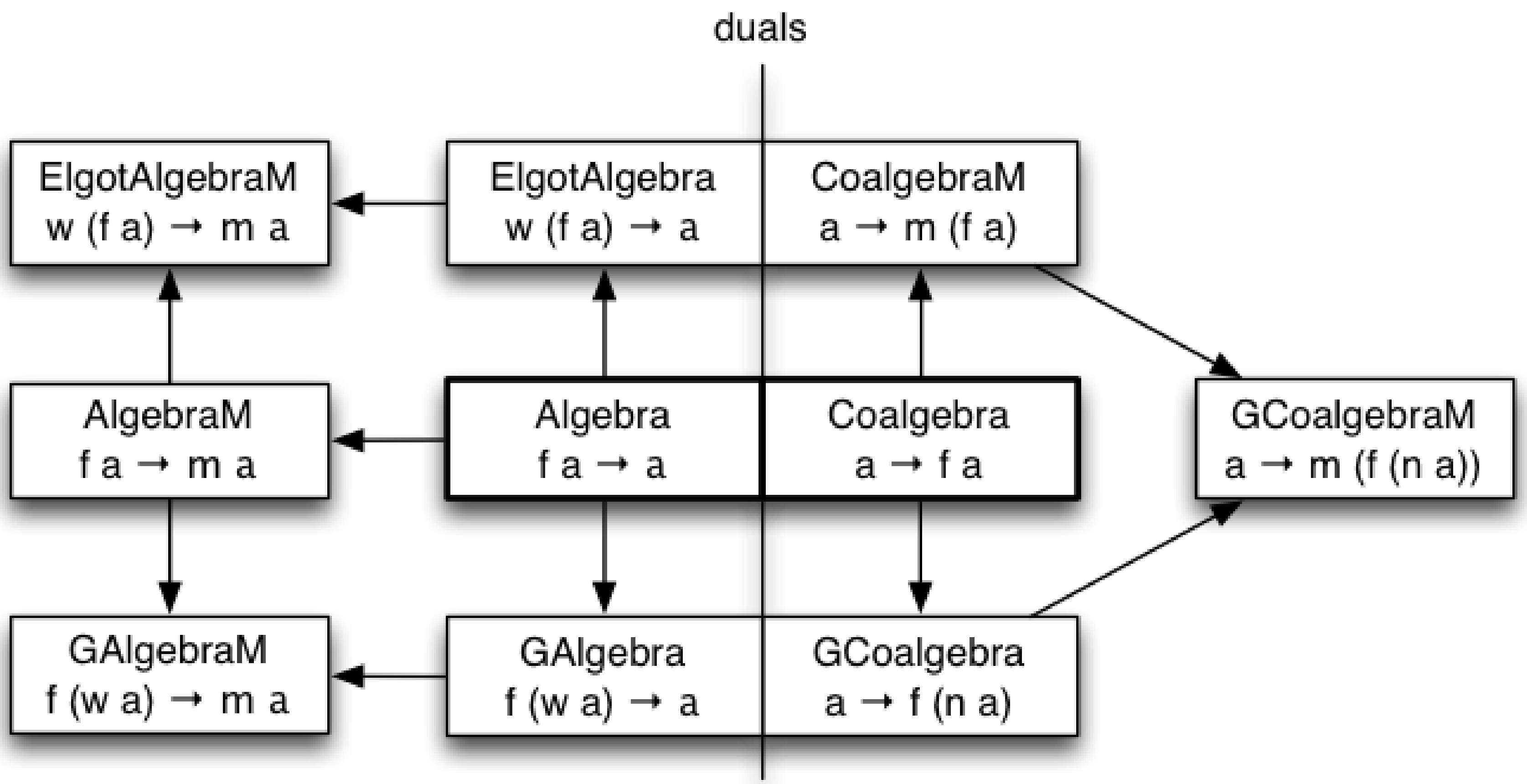
ALL YOU NEED TO KNOW IS:

- Create a coalgebra when you want to build up a structure, using the seed/instruction/specification as the input.
- Call `ana`. *Done!*

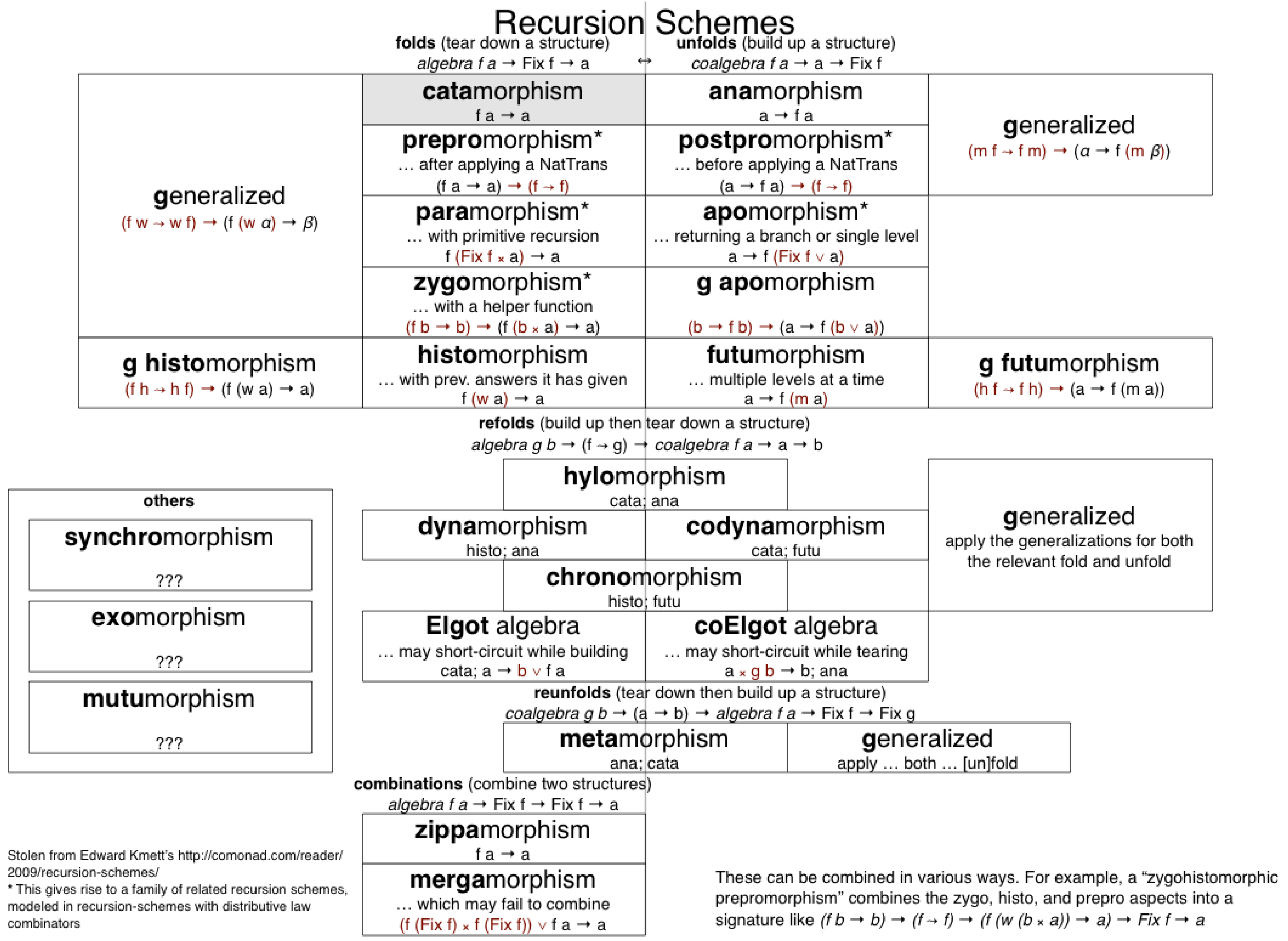
Recursion Scheme Basics

- Fixpoints
- (Co)Algebra
- {cata,ana}morphism

Most everything else builds on the above.



https://github.com/slamdata/matryoshka/blob/master/resources/algebras.png



https://github.com/slamdata/matryoshka/blob/master/resources/recursion-schemes.png

Two very useful extensions...

1. Operation Fusion

Combine catamorphism & anamorphism into a single operation (called a hylomorphism).

Hylomorphism sounds scary...

...but it's really simple. Pass the same arguments to 1 method instead of 2.

```
def ana[F[_]: Functor, A]
      (coalg: Coalgebra[F, A])(a: A): Fix[F]

def cata[F[_]: Functor, B]
      (alg: Algebra[F, B])(f: Fix[F]): B
```

```
def hylo[F[_]: Functor, A, B]
      (coalg: Coalgebra[F, A], alg: Algebra[F, B])(a: A): B
```

...but it's really simple. Pass the same arguments to 1 method instead of 2.

```
def unfold[F[_]: Functor, A]
      (coalg: Coalgebra[F, A])(a: A): Fix[F]

def fold[F[_]: Functor, B]
      (alg: Algebra[F, B])(f: Fix[F]): B
```

```
def unfoldIntoFold[F[_]: Functor, A, B]
      (coalg: Coalgebra[F, A], alg: Algebra[F, B])(a: A): B
```

Build-up & tear-down in one pass.

Generate & consume without actually creating the whole tree.

$\Theta(n)$ instead of $\Theta(2n)$.

2. Monadic versions

Algebras can return monads.

What's a monad?

Oh god...

That's a separate talk *but*, speaking *extremely* loosely:

- A composable wrapper around data or intent.
- Something with `map` and `flatMap` methods.
- Something you can use in `for` comprehensions.

Some monads you've probably already used:

- `Option[A]`
- `List[A]`
- `Future[A]`
- `Either[A, B]` / Scalaz's disjunction `A \/ B`

```scala
val eval: Calc[Int] => String \/ Int = {
  case Num(i)    => \/-(i)
  case Div(_, 0) => -\/("Division by zero: Australia says no!")
  case Div(a, b) => \/-(a / b)
}
```

```scala
val eval: Calc[Int] => Either[String, Int] = {
  case Num(i)    => Right(i)
  case Div(_, 0) => Left("Division by zero: Australia says no!")
  case Div(a, b) => Right(a / b)
}
```

```scala
type Algebra  [F[_], A] = F[A] => A
type CoAlgebra[F[_], A] = A => F[A]
```

```scala
type AlgebraM  [M[_], F[_], A] = F[A] => M[A]
type CoAlgebraM[M[_], F[_], A] = A => M[F[A]]
```

```scala
def unfold[F[_]: Functor, A]
      (coalg: Coalgebra[F, A])(a: A): Fix[F]

def fold[F[_]: Functor, B]
      (alg: Algebra[F, B])(f: Fix[F]): B
```

```scala
def monadicUnfold[M[_]: Monad, F[_]: Traverse, A]
      (coalg: CoalgebraM[M, F, A])(a: A): M[Fix[F]]

def monadicFold[M[_]: Monad, F[_]: Traverse, B]
      (alg: AlgebraM[M, F, B])(f: Fix[F]): M[B]
```

- (Co)Algebra -> (Co)AlgebraM
- Result is now `M[_]`
- Functor -> Traverse

```scala
type AlgebraM  [M[_], F[_], A] = F[A] => M[A]
type CoAlgebraM[M[_], F[_], A] = A => M[F[A]]
```

```scala
val eval: Calc[Int] => String \/ Int = {
  case Num(i)    => \/-(i)
  case Div(_, 0) => -\/("Division by zero detected.")
  case Div(a, b) => \/-(a / b)
}
```

```scala
// Calc[Int] => String \/ Int
// F   [A]   => M         [A]

type StringOr[A] = String \/ A

val eval: AlgebraM[StringOr, Calc, Int] = {
```

Great! We've just added error handling and short-circuiting.

Real Example

Random JSON generator.

Random Data

Generating random data is important.

It's the secret sauce of property testing that ensures that, (asymptotically), you test your code with every possible, legal or desirable value.

Also useful for:

- load testing
- stress testing
- benchmarking (if generator supports determinism)

Nyaya

```scala
case class Whatever(enabled : Boolean,
                    position: (Int, Int),
                    stuff   : Map[Long, Option[String]])
```

```scala
import nyaya.gen._

val genWhatever: Gen[Whatever] =
  for {
    enabled  <- Gen.boolean
    position <- Gen.chooseInt(-128, 128).pair
    stuff    <- Gen.long.mapTo(Gen.string.option)
  } yield Whatever(enabled, position, stuff)
```

```scala
case class Whatever(enabled : Boolean,
                    position: (Int, Int),
                    stuff   : Map[Long, Option[String]])
```

```scala
println(genWhatever.sample())

// Whatever(
//  true,
//   (-30, 83),
//   Map(
//     2340946662719216224 -> Some("!XM91u"),
//     7161527918171176759 -> None
//   )
// )
```

Hint: `Gen` is a monad.

```
for {
  enabled  <- Gen.boolean
  position <- Gen.chooseInt(-128, 128).pair
  stuff    <- Gen.long.mapTo(Gen.string.option)
} yield Whatever(enabled, position, stuff)
```

Our imaginary app uses Play JSON.

Play JSON (like everything else) has hardcoded recursion.

We need to abstract over recursion and use fixpoints...

Fixpoint JSON

```scala
sealed trait JsonF[A]
object JsonF {
  case class Null[A]()                              extends JsonF[A]
  case class Bool[A](value: Boolean)                extends JsonF[A]
  case class Str [A](value: String)                 extends JsonF[A]
  case class Num [A](value: Double)                 extends JsonF[A]
  case class Arr [A](values: List[A])               extends JsonF[A]
  case class Obj [A](fields: List[(String, A)])     extends JsonF[A]
}
```

Fixpoint JSON

```
sealed trait JsonF[+A]
object JsonF {
  case object Null                                  extends JsonF[Nothing]
  case class  Bool  (value: Boolean)                extends JsonF[Nothing]
  case class  Str   (value: String)                 extends JsonF[Nothing]
  case class  Num   (value: Double)                 extends JsonF[Nothing]
  case class  Arr[A](values: List[A])               extends JsonF[A]
  case class  Obj[A](fields: List[(String, A)])     extends JsonF[A]
}
```

Traverse (skeleton)

```
import scalaz._, Scalaz._

implicit val traverse: Traverse[JsonF] = new Traverse[JsonF] {
  override def traverseImpl[G[_], A, B](fa: JsonF[A])(f: A => G[B])
                                         (implicit G: Applicative[G]): G[JsonF[B]] =
      ???
}
```

s/JsonF/YourDataType/g

Traverse (body)

```
fa match {
  case x@ Null      => G.pure(x)
  case x@ Bool(_)   => G.pure(x)
  case x@ Str(_)    => G.pure(x)
  case x@ Num(_)    => G.pure(x)
  case Arr(values) => values.traverse(f).map(Arr(_))
  case Obj(fields) => fields.traverse{ case (s,a) => f(a).map((s,_)) }.map(Obj(_))
}
```

0x `A` : Use `G.pure` .

1x `A` : Use `G.apply` .

2x `A` : Use `G.apply2` .

3x `A` : Use `G.apply3` .

...

Collection of `A` s: Use `.traverse` then `.map` .

Done:

- `Traverse` & `Functor` instance

TODO:

- Generate JSON (using custom data type)
- Convert custom JSON into Play JSON

Live code time...

In summary...

Recursive data types:

- Trees, indented lists
- Tries
- DSLs, expression languages
- Anything self-referential
- Logical propositions, assertions
- Composable actions/data

- Allows you to not think about, or implement recursion.
- Where there'd normally be recursion, just declare what you want.
  Magic will grant your wish.
- Traditionally high barrier to entry. It's needless.
  You don't need to be a pilot to catch a plane.
  You don't *need* to understand Greek or category theory to use this.
- It wasn't *discovered* simply but now that it is, it's simple to use.

ALL YOU NEED TO KNOW TO BENEFIT IS:

- Use a library or copy-and-paste little snippets that you need. They're tiny.
- Parametise your data; use `Fix` instead of self-referencing.
- When writing your logic, use plain old functions--just use the right shapes.
- 🎉

Libraries:

My recursion micro-library

- https://github.com/japgolly/microlibs-scala
- Minimalistic: Few algebras & morphisms. Only supports `Fix`.
- Fast.
- Beginner focused.
  - Contains an `EasyRecursion` module with English instead of Greek.
  - Hopefully less intimidating for non-FP teams.
  - Nice stepping-stone before graduating to Matryoshka.

Libraries:

Matryoshka

- https://github.com/slamdata/matryoshka
- *Very* comprehensive. Kitchen-sink of recursion.
- Will become part of Typelevel suite.
- Currently undergoing a lot of change.
- Super-smart people seriously working on it.
- Going to be awesome.

Resources

- Youtube: Any talks by Greg Pfeil.
- Youtube: "Pure functional database programming with fixpoint types" by Rob Norris.
- Google: "recursion scheme cheatsheets".
- Paper: "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire"

All done; Thank you!

Huge thanks to Rob Norris & Greg Pfeil for teaching me!

Github: `japgolly`
Twitter: `@japgolly`