# Algebras and Fixpoints

$$S \Rightarrow (S, A)$$

**F A ⇒ A**

$$F[A] \Rightarrow A$$

# Recursive Data

What is it?

**My use cases over the last few years**

- ShipReq: filter expr
- ShipReq: tags M:M
- ShipReq: use case tree
- ShipReq: code trie
- Nyaya: Properties
- TestState
- Sprockets: Expr
- More?

# Recursive Data

**Why is it important?**

- Model reality
  - My friends have friends who have friends who have friends...
  - FS: directories have directories which have directories...
- Efficiency (time) - maps
- Efficiency (space) - tries

# Recursion

Not only about data, can also use existing iterable types.

- String processing
- List processing

# Recursive Data: 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.**

**Can't abstract, define depth.**

# Problem #2

**Recursion is usually hard. Often requires practice.**

**Non-recursion easier than recursion.**

# Problem #3

Can't annotate sub-nodes.

## No annotations

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

## Annotate with ID

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

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

# Problem #3

## Annotate `Calc` with logging

```scala
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

## Annotate `Calc` with whatever

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

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

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

## Problems #4 ~ #23

So obvious that we don't even need to talk about them.

*(is joke)*

# Q: How do we deal with this?

A: Generalise the recursive-type.

```
// No longer recursive.

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]
```

`Calc` no longer references `Calc`.

Notice `Add` / `Multiply` fields.

# Wait...

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

# Type constructors

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

# Types

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

# Type constructors

- `Functor`

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

# Types

- `Functor[List]`

```scala
object ListFunctor extends Functor[List] {
 override def map[A, B](list: List[A])(f: A => B): List[B] =
   list.map(f)
}
```

# 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 MAX_DEPTH_3 : Calc[Calc[Calc[Nothing]]] =
  Multiply(
    Number(3),
    Add(
      Number(1),
      Number(2)))
```

# Fixpoints!

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

# Fixpoints!

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

```scala
type RecursiveCalc = Fix[Calc]
//                 = Fix[Calc[Fix[Calc[Fix[Calc[Fix[Calc[...
```

```scala
val UNLIMITED_DEPTH_! : RecursiveCalc =
  Fix(Multiply(
    Fix(Number(3)),
    Fix(Add(
      Fix(Number(1)),
      Fix(Number(2))))))
```

# How? Why?

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]]
```

```
val calc: Fix[Calc] =
  Fix(Number(1))
```

```
val calc: Fix[Calc] =
  gimmeFix(
    gimmeCalc(
      gimmeFix(
        gimmeCalc(
          gimmeFix(
            gimmeCalc(
              gimmeFix(
                gimmeCalc(
                  gimmeFix(
                    Number(1))))))))))
```

## ALL YOU NEED TO KNOW IS:

- Generalise (use an `A`) in place of a self-reference.
- Use a library like Matryoshka, or copy-and-paste `Fix`.
- Wrap your stuff in `Fix` when you want recursion.

# Algebra

`F A -> A`

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

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

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

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

val listSumMethodAsAlgebra: Algebra[List, Int] =
  listSumMethod
```

# Ok... and?

A special function exists...

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

Like magic!

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

# Let's try it out!

## Well, first there's a prerequisite:

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

      }
  }
```

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

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

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

awesome(eval, c) // 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

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

# Amazing!

Real name is catamorphism.

It goes to the ends of a tree, then calculates it way back up.

Bottom-up.

What is this black magic?

This heresay?

"Lies!"

# Proof that it works...

```
    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)
```

```scala
def cata[F[_], A](alg: Algebra[F, A])(f: 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]
                    calc.map(cata)
```

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

## ALL YOU NEED TO KNOW IS:

- Use a library like Matryoshka, or copy-and-paste `Algebra` and `cata`.
- Create an algebra when you want to fold/reduce a tree.
- Call `cata`. *Done!*

# Next up: CoAlgebras

Everything in category theory can have its arrows reversed - this is called a dual.

## Monad

- `M[A] => (A => M[B]) => M[B]`

## CoMonad

- `W[B] => (W[B] => A) => W[A]`

## Algebra:

- `F[A] => A`

## CoAlgebra

- `A => F[A]`

# What's the purpose?

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

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

# Another magic function

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

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

Real name is anamorphism.

Dual of catamorphism.

```
cata: Fix f -> Alg   f a -> a
ana : a     -> CoAlg f a -> Fix f
```

It starts with the at the top, then calculates its way down until the tree is complete.
Top-down.

The `A` is an instruction/description of the subtree yet to be calculated.

## ALL YOU NEED TO KNOW IS:

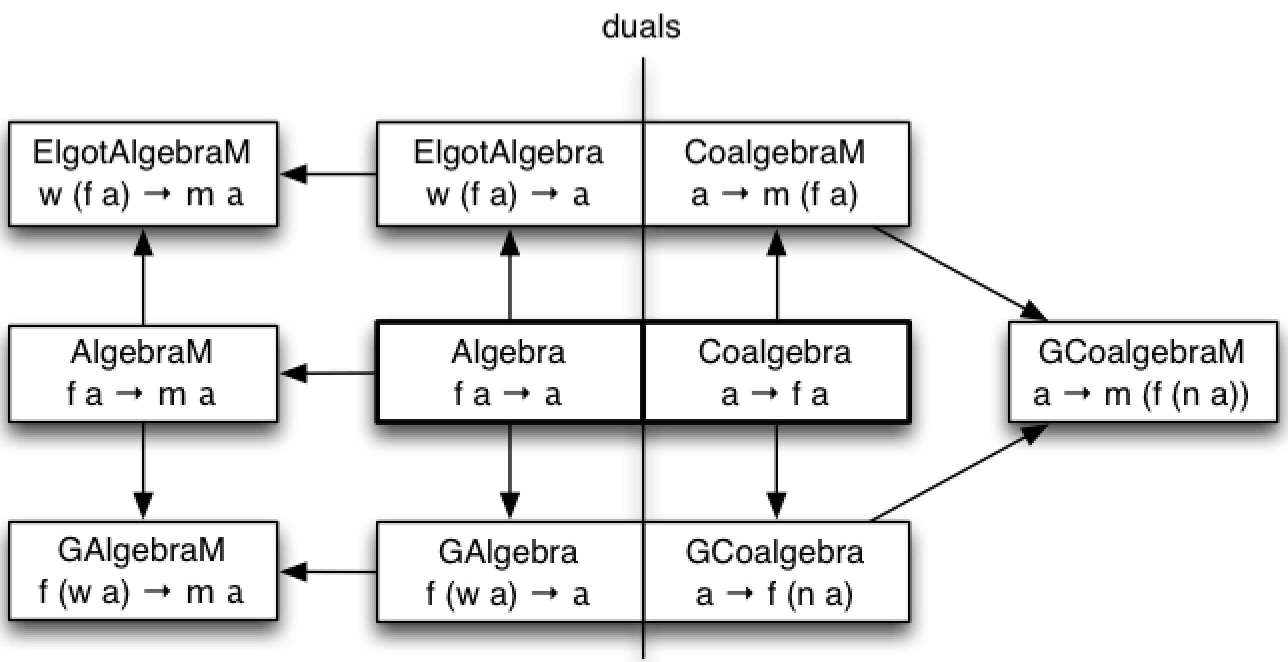
- Use a library like Matryoshka, or copy-and-paste `CoAlgebra` and `ana`.
- Create a coalgebra when you want to build up a tree with context.
- Call `ana`. *Done!*

# There is no more.

No new information required.

# There is more.

There are a bunch of variations and combinations that extend/build on what you've now learned.



duals
ElgotAlgebraM
w (f a) → m a
ElgotAlgebra
w (f a) → a
CoalgebraM
a → m (f a)
AlgebraM
f a → m a
Algebra
f a → a
Coalgebra
a → f a
GCoalgebraM
a → m (f (n a))
GAlgebraM
f (w a) → m a
GAlgebra
f (w a) → a
GCoalgebra
a → f (n a)

# Recursion Schemes

**folds** (tear down a structure)
*algebra f a* → Fix f → a

↔

**unfolds** (build up a structure)
*coalgebra f a* → a → Fix f

| generalized (folds) | folds | unfolds | generalized (unfolds) |
|---|---|---|---|
| **g**eneralized<br>(f w → w f) → (f (w α) → β) | **cata**morphism<br>f a → a | **ana**morphism<br>a → f a | **g**eneralized<br>(m f → f m) → (α → f (m β)) |
| | **prepro**morphism*<br>… after applying a NatTrans<br>(f a → a) → (f → f) | **postpro**morphism*<br>… before applying a NatTrans<br>(a → f a) → (f → f) | |
| | **para**morphism*<br>… with primitive recursion<br>f (Fix f × a) → a | **apo**morphism*<br>… returning a branch or single level<br>a → f (Fix f ∨ a) | |
| | **zygo**morphism*<br>… with a helper function<br>(f b → b) → (f (b × a) → a) | **g apo**morphism<br>(b → f b) → (a → f (b ∨ a)) | |
| **g histo**morphism<br>(f h → h f) → (f (w a) → a) | **histo**morphism<br>… with prev. answers it has given<br>f (w a) → a | **futu**morphism<br>… multiple levels at a time<br>a → f (m a) | **g futu**morphism<br>(h f → f h) → (a → f (m a)) |

**refolds** (build up then tear down a structure)
*algebra g b* → (f → g) → *coalgebra f a* → a → b

**hylo**morphism
cata; ana

**dyna**morphism
histo; ana

**codyna**morphism
cata; futu

**chrono**morphism
histo; futu

**Elgot** algebra
… may short-circuit while building
cata; a → b ∨ f a

**coElgot** algebra
… may short-circuit while tearing
a × g b → b; ana

**g**eneralized
apply the generalizations for both the relevant fold and unfold

**reunfolds** (tear down then build up a structure)
*coalgebra g b* → (a → b) → *algebra f a* → Fix f → Fix g

**meta**morphism
ana; cata

**g**eneralized
apply … both … [un]fold

**combinations** (combine two structures)
*algebra f a* → Fix f → Fix f → a

**zippa**morphism
f a → a

**merga**morphism
… which may fail to combine
(f (Fix f) × f (Fix f)) ∨ f a → a

**others**

**synchro**morphism
???

**exo**morphism
???

**mutu**morphism
???

Stolen from Edward Kmett's http://comonad.com/reader/2009/recursion-schemes/

* This gives rise to a family of related recursion schemes, modeled in recursion-schemes with distributive law combinators

These can be combined in various ways. For example, a "zygohistomorphic prepromorphism" combines the zygo, histo, and prepro aspects into a signature like *(f b → b) → (f → f) → (f (w (b × a)) → a) → Fix f → a*

# Map Fusion

Build-up & tear-down in one pass.

Generate & consume without actually creating the whole tree.

# Monadic variants

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

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

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

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

AlgebraM[String \/ ?, Calc, Int]
//       ^^^^^^^^^^^  ^^^^  ^^^
//       M            F     A
```

# Real Example

# 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 desriable value.

Also useful for:

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

# Nyaya

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

val g: Gen[Example] =
  for {
    e <- Gen.boolean
    p <- Gen.chooseInt(-128, 128).pair
    s <- Gen.ascii.string(1 to 10).option
           .mapBy(Gen.long)(0 to 4)
  } yield Example(e, p, s)

val example: Example = g.sample()

// Example(
//   true,
//   (-30,83),
//   Map(2340946662719216224 -> Some(!\@91u),
//       7161527918171176759 -> None))
```

# DSL

```
sealed trait Expr[+A]

case class Literal(typeAndValue: Type.AndValue) extends Expr[N
case class Proposition[A](op: BoolOp, left: A, right: A) exten
case class Arithmetic[A](op: ArithmeticOp, left: A, right: A)
case class Comparison[A](op: ComparisonOp, left: A, right: A)
case class Not[A](expr: A) extends Expr[A]
case class ReadState(name: String) extends Expr[Nothing]
case class CallBuiltInFunction[A](name: String, args: List[A])
```

# Goal

```
def gen: Gen[Fix[Expr]] =
  ???
```

```scala
def gen: Gen[Fix[Expr]] =
  ???

val coalgebra: CoAlgebra[Expr, ?] =
  ???
```

```scala
/** Specification for how a valid expression should be generat
case class ExprSpec(
  resultType      : Type,
  remainingDepth  : Int)

val coalgebra: CoAlgebraM[Gen, Expr, ExprSpec] =
//           : ExprSpec => Gen[Expr[ExprSpec]] =
  spec => ???

def gen(result: Type, maxDepth: Int = 5): Gen[Fix[Expr]] =
  anamorphismM(coalgebra)(ExprSpec(result, maxDepth))
```

```scala
val coalgebra: CoAlgebraM[Gen, Expr, ExprSpec] = { spec ⇒
  var gens = List.empty[Gen[Expr[ExprSpec]]]

  // Add non-recursive gens
  gens ::= genLiteral(spec.resultType)
  // ...

  // Add recursive gens
  if (spec.remainingDepth > 0) {
    val nextDepth = spec.remainingDepth - 1

    resultType match {
      case Type.Bool ⇒
        val boolSpec = Spec(Type.Bool, nextDepth)
        gens ::= Gen pure Not(boolSpec)

        val longSpec = Spec(Type.Long, nextDepth)
        gens ::= genComparisonOp.map(op =>
                    Comparison(op, longSpec, longSpec))

      // case ... => ...
    }
  }

  Gen.chooseGen(gens) // returns a Gen[Expr[ExprSpec]]
}
```

I don't think about recursion.

Where there'd normally be recusion, I just declare what I want.

**Magic ensures.**

# Summary [1/2]

- Allows you to not think about recursion or implement it.
- Allows you to write code that works with *ANY* structure.
  - eg. generic algebras for size & height.
- You don't need to be a pilot to catch a flight. 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.

# Summary [2/2]

## ALL YOU NEED TO KNOW IS:

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

# Resources

- Matryoshka: Scala library by Slamdata.
- 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"