Comonads

Monads are popular.

Monads are not special.

Category theory redux

Categories

A collection C of objects connected with arrows, such that:

$$\forall object\ X \in Obj(C)\ .\ id_X = X \to X \in Arr(C)$$

$$\begin{aligned} \forall objects\ X, Y, Z \in Obj(C)\ . \\ f = X \to Y \in Arr(C) \wedge \\ g = Y \to Z \in Arr(C) \\ \Rightarrow f \circ g = X \to Z \in Arr(C) \end{aligned}$$

Category theory redux

Objects

The only evidence we have for the existence of any object X is the arrow $id_X = X \to X$. The inequality of two id arrows is what distinguishes two objects. Otherwise, the objects have no content or properties.

Category theory redux

Arrows (Morphisms)

Defined by their **source** and **target**. They define the semantics of a category.

Category theory redux

Functors

Structure-preserving mappings between categories. They map objects to objects, and morphisms between two objects to morphisms between their respective corresponding objects.

Category theory redux

Functors

Structure-preserving mappings between categories. They map objects to objects, and morphisms between two objects to morphisms between their respective corresponding objects.

Category theory redux

Duals

Flipping all arrows sometimes yields great results.

Useful Functorial Structures

map, covariant functor:

$$(A \Rightarrow B) \Rightarrow (F[A] \Rightarrow F[B])$$

contramap, contravariant functor:

$$(B \Rightarrow A) \Rightarrow (F[A] \Rightarrow F[B])$$

apply, applicative:

$$F[A \Rightarrow B] \Rightarrow (F[A] \Rightarrow F[B])$$

flatMap, monad:

$$(A \Rightarrow F[B]) \Rightarrow (F[A] \Rightarrow F[B])$$

coflatMap, comonad:

$$(F[A] \Rightarrow B) \Rightarrow (F[A] \Rightarrow F[B])$$

Monad

```scala
trait Monad[F[_]] extends Functor[F] {
  // aka return
  def wrap[A]: A => F[A]

  // aka join
  def flatten[A]: F[F[A]] => F[A]

  // aka bind
  def flatMap[A, B]: (A => F[B]) => (F[A] => F[B])
}
```

Comonad

```scala
trait Comonad[F[_]] extends Functor[F] {
  // aka coreturn
  def extract[A]: F[A] => A

  // aka cojoin
  def duplicate[A]: F[A] => F[F[A]]

  // aka cobind, coflatMap
  def extend[A, B]: (F[A] => B) => F[A] => F[B]
}
```

Comonad

Comonadic laws

```
extend extract = id
extract . extend f = f
extend f . extend g = extend (f . extend g)
```

Reasoning

Monads: *effectful* computations required to *produce* values

Comonads: *contextual* computations required to *consume* values

Uses for Comonads

- Annotated structures
- "Pointed" structures
- Functional Reactive Programming, Signal Processing

Uses for Comonads

Annotated structures: `F[A] => B` interpreted as creating annotations of type `B` given a value of type `F[A]` (`F` is a functor, so `fmap` guarantees that the annotated structure will keep the same structure)

Uses for Comonads

Annotated structures

A non-empty tree.

```
case class Tree[A](tip: A, sub: List[Tree[A]])
```

Uses for Comonads

Annotated structures

A tree of all subtrees.

```
def duplicate: Tree[Tree[A]] =
    Tree(this, sub.map(_.duplicate))
```

Uses for Comonads

Annotated structures

A tree of all subtrees that we can **map** over!

```
def duplicate: Tree[Tree[A]] =
    Tree(this, sub.map(_.duplicate))

duplicate(tree).map(f)
// is equivalent to
extend(tree)(f)
```

`f` takes a tree and performs some computation that required that tree's information (including not only its value but also its subtree)

Uses for Comonads

Annotated structures

A tree of all subtrees that we can **map** over!

```
def duplicate: Tree[Tree[A]] =
    Tree(this, sub.map(_.duplicate))

duplicate(tree).map(f)
// is equivalent to
extend(tree)(f)
```

The result is a new tree mirroring `tree`, except that each node has `f` applied over the corresponding subtree (annotating it).

Uses for Comonads

Annotated structures

A tree of all subtrees that we can **map** over!

```
// alias extend with =>>

exprTree =>> annotateTypes
```

The result is a new tree mirroring `exprTree`, except that each node has `annotateTypes` applied over the corresponding subtree.

Uses for Comonads

"Pointed" structures: Duplicate can be understood as pointing at the input `F[A]` and giving us all "neighboring" substructures.

```
def duplicate[A]: F[A] => F[F[A]]
```

Uses for Comonads

"Pointed" structures

Zipper-like structures, traversers, iterators...

```
case class Zip[A]
      (pre: List[A], now: A, post: List[A])
```

Uses for Comonads

"Pointed" structures

Zipper-like structures, traversers, iterators...

```
Zip([ -2, -3, ...], -1, [ 0, 1, ... ])
```

Uses for Comonads

"Pointed" structures

Zipper-like structures, traversers, iterators...

```
// helper function
def iterate[A](app: A => A, start: A): List[A] =
    start :: iterate(app, app(start))

def fmap[B](f: A => B): Zip[B] =
  Zip[B](pre.map(f), f(now), post.map(f))

def duplicate =
  Zip(
    iterate(shiftLeft _, this).tail, this,
    iterate(shiftRight _, this).tail)
```

Uses for Comonads

"Pointed" structures

Zipper-like structures, traversers, iterators...

```
// these are trivial

def extract: A = now

def extend[B](f: Zip[A] => B): Zip[B] =
  coflatten.fmap(f)
```

Uses for Comonads

"Pointed" structures

Cellular automata-like rule applications by `extend`ing over every point and getting its neighborhood from `duplicate`.

```
def rule(cell: Zip[Boolean]): Boolean = cell match {
  case Zip(a :: _, b, c :: _) =>
    !(a && b && !c || (a == c))
}
```

Uses for Comonads

"Pointed" structures

Simple example of usage, in an image libraries (remember CS162? Look at this).

```
def blur(pixel: Pixel[Double]) = {
  val focus = pixel.get
  val before = pixel.shift(pixel.index - 1)
  val after = pixel.shift(pixel.index + 1)
  0.25 * before + 0.5 focus + 0.25 * after
}

def scale(pixel: Pixel[Double]) =
  pixel.get * 2
```

Uses for Comonads

"Pointed" structures

Useful for image libraries (remember CS162? Look at this).

```
// We write functions that only "think" locally,
// in their environment
def blur(pixel: Pixel[Double]) = {
  val focus = pixel.get
  val before = pixel.shiftLeft.get
  val after = pixel.shiftRight.get
  0.25 * before + 0.5 focus + 0.25 * after
}

def scale(n: Double)(pixel: Pixel[Double]) =
  pixel.get * n

val image = Pixels.fromSeq(1, 1, 1, 1, 1, 0, 0, 0, 0)
val result = image =>> blur =>> scale(0.5)
```

Uses for Comonads

"Pointed" structures

Useful for image libraries (remember CS162? Look at this).

Possibly useful for shader languages which do pipelining!

Uses for Comonads

Functional Reactive Programming, **Signal Processing**: Both of these use streams, and streams are inherently comonadic.

Uses for Comonads

Modelling OO programming: objects (collections of fields and member functions) can be built from scratch using three comonads (`Traced + Stream + Store = Command Pattern`).

How do we get comonads for cofree?

We can generalize the comonadic structure by taking a functor (*any* functor) and putting it into the `Cofree` comonad (it's a comonad cogenerator!).

```
case class Cofree[F[_], A]
  (counit: A, sub: F[Cofree[F,A]]) {

  def duplicate(implicit F: Functor[F]):
    Cofree[F,Cofree[F,A]] =

      Cofree(this, F.map(sub)(_.duplicate))

  def extract = counit
}
```

In Practice

With (functional) languages being more used to working with monads, sampling code from libraries shows that monads are used for **external** interfaces, while many **internal** contracts are based on chained comonads.

In Practice

Both Dan Piponi and Edward Kmett say to never compose comonads over monads, as they end up being very unoptimized. Composing the other way around seems to be the natural solution for languages we have now (monads out, comonads inside).

In Type Systems

Monads represent side-effects, and can be added to the type system as **effects**. These effects are requirements to get to the output.

$$\text{print} \;: \text{string} \rightarrow \; \text{unit} \;\&\; \{\ \text{io}\ \}$$

Comonads represent context, and can be added to the type system as **coeffects**. These effects are requirements on the input to even start the computation.

$$\text{stopwatch} \;: \text{unit} \;@\; \{\ \text{clock}\ \} \rightarrow \text{int}$$

Coestions

Cothank you