

Comonadic Interface Design

Your ticket for the good life

Mitch Stevens

Time to invest in comonads!

- **Duplicate** investment in months

The next big thing

Who's using them?

- co-log ^a

^a<https://hackage.haskell.org/package/co-log>

Comonads

- **Comonads** are dual structure to Monads
- Monads express effectful computations
- Comonads are values in some context

```
class Comonad w where
  extract    :: w a -> a           -- copure
  duplicate  :: w a -> w (w a)    -- cojoin
```

Extracting and Duplicating

- Comonads can be seen as a state transition diagram ¹
- Using `extract`, we can extract the value that we were focusing on

¹A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

Extracting and Duplicating

- Comonads can be seen as a state transition diagram ¹
- Using `extract`, we can extract the value that we were focusing on



Figure 1: A Scomonad focused on something

¹A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

Extracting and Duplicating

- Comonads can be seen as a state transition diagram ¹
- Using `extract`, we can extract the value that we were focusing on



Figure 1: A Scomonad focused on something

- `duplicate` explodes out all the states of the transition

¹A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

The Zipper Comonad

```
data Zipper a = [a] a [a]
```

```
left, right :: Zipper a -> Zipper a
```

```
left  (Zipper (l:ls) v rs) = Zipper ls l (v:rs)
```

```
right (Zipper ls v (r:rs)) = Zipper (v:ls) r rs
```

```
instance Comonad Zipper where
```

```
  extract (Zipper l v r) = v
```

```
  duplicate zipper = Zipper allLefts zipper allRights
```

```
    where ...
```


Extracting and Duplicating

```
data NonEmptyList a = NonEmptyList a [a]

tail :: NonEmptyList a -> NonEmptyList a
tail (NonEmptyList _ xs) = NonEmptyList (head xs) (tail xs)

instance Comonad Zipper where
  extract (NonEmptyList x xs) = x
  duplicate neList = NonEmptyList neList allTails
  where ...
```

Examples

- Identity a
- (e, a)
- Nonempty Graphs ²
- Trees with values in the branches ($\text{Cofree } f$)

²A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

Kliesli and Cokliesli

- A function $a \rightarrow m\ b$ is called a Kleisli arrow
- If m is a monad, we get Kleisli composition for free
$$(=>=) :: (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$$

Kliesli and Cokliesli

- A function $a \rightarrow m\ b$ is called a Kleisli arrow
- If m is a monad, we get Kleisli composition for free

$(\Rightarrow) :: (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$

- The dual to a Kleisli arrow is a Cokliesli arrow
- If w is a comonad, we also get Cokliesli composition

$(\Rightarrow) :: (w\ a \rightarrow b) \rightarrow (w\ b \rightarrow c) \rightarrow (w\ a \rightarrow c)$

Uses for Comonads

Image processing is a natural fit for Kleisli composition³
we can focus on

```
render :: FocusedImage Pixel -> Image  
blur   :: FocusedImage Pixel -> Pixel  
lighten :: FocusedImage Pixel -> Pixel
```

```
lighten =>= blur =>= render
```

³A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

What is a UI?

- The only hard requirement is a rendering function. . .

What is a UI?

- The only hard requirement is a rendering function...
- But we'll also need
 - ▶ Mutable state
 - ▶ initialiser, finaliser
 - ▶ preloaded data
 - ▶ other effects, etc

What is a UI?

- The only hard requirement is a rendering function...
- But we'll also need
 - ▶ Mutable state
 - ▶ initialiser, finaliser
 - ▶ preloaded data
 - ▶ other effects, etc

```
data NaiveUI s h = UI
  { state :: s
  , render :: s -> h
  }
```

- This would allow us to `fmap` over `h` to render to something else.
- UI admits a comonad instance

The store comonad

- The NaiveUI comonad is usually called Store

```
data Store s a = Store (s -> a) s
instance Comonad (Store s) where
  extract (Store render state) = render state
  duplicate (Store render state)
    = Store (Store render) state

Store s (Store s a) = Store (s -> Store s a) s
```

Components using Comonads

```
type Component w = Comonad w => w (UI ())
```

- `extract` will render the component
- `duplicate` will explore future states of a component

```
extract :: Component w -> UI ()           -- render  
duplicate :: Component w -> w (Component w) -- explode  
select :: x -> w (Component w) -> Component w -- choose
```

Adjunctions

- An adjunction is a relationship between two functors f and g .

```
-- from Data.Functor.Adjunction (simplified)
class (Functor f, Functor g) => Adjunction f u where
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b
```

- We call this relationship an **Adjunction**
- There are also a set of Adjunction laws
- If we require `Monad g` and `Comonad f`, this looks like an isomorphism between `Kliesli g` and `Cokliesli f`...

Examples of Monad/Comonad Adjunctions

Monad	Comonad
Identity	Identity
Reader r	Env r
State s	Store s
Writer w	Traced w
Free f	Cofree f

We also have an adjunction between monad/comonad transformers

```
instance Adjunction w m =>  
  instance Adjunction (EnvT r w) (ReaderT r m)
```

The Reader/Env Pairing

```
type Reader r a = r -> a -- Monad m  
type Env      r a = (a, r) -- Comonad w
```

The Reader/Env Pairing

```
type Reader r a = r -> a -- Monad m
type Env      r a = (a, r) -- Comonad w
```

Adjunction requirements:

$$(w\ a \rightarrow b) \rightarrow (a \rightarrow m\ b)$$
$$(a \rightarrow m\ b) \rightarrow (w\ a \rightarrow b)$$

An utterly surprising result!

`m ()` can be used to navigate through `w a`

```
select :: Adjunction w m => m () -> w (w a) -> w a
```

An utterly surprising result!

$m \ ()$ can be used to navigate through $w \ a$

```
select :: Adjunction w m => m () -> w (w a) -> w a
```

If w has a right adjunct m , we get a navigation type for free

Overview

We have developed a way of

```
extract :: Component w -> UI ()           -- render  
duplicate :: Component w -> w (Component w) -- explode  
select :: m () -> w (Component w) -> Component w -- choose
```

Applications

We want to be able to compose comonadic components

*The limerick packs laughs anatomical
In space that is quite economical.
But the good ones I've seen
So seldom are clean
And the clean ones so seldom are comical*

Comonadic Sum

```
data A f a =  
  A (m x -> b) (f x)
```

Comonadic Product

Use case

```
data Day f g a =  
    Day (x -> y -> a) (f x) (g y)  
instance Comonad f, Comonad g => Comonad (Day f g)
```

Homogenous Transformers

Monad Transformers

```
class MonadTrans t where  
  lift :: Monad m => m a -> t m a
```

Homogenous Transformers

Monad Transformers

```
class MonadTrans t where  
  lift :: Monad m => m a -> t m a
```

Comonad Transformers

```
class ComonadTrans t where  
  lower :: Comonad w => t w a -> t a
```

Co

Co is a heterogenous transformer

```
data Co w a = Co { unCo :: w (a -> r) -> r }  
instance Comonad w => Monad (Co w) where ...
```

Co

Co is a heterogenous transformer

```
data Co w a = Co { unCo :: w (a -> r) -> r }  
instance Comonad w => Monad (Co w) where ...
```

Given a comonad w , $\text{Co } w$ is a monad

Co

Co is a heterogenous transformer

```
data Co w a = Co { unCo :: w (a -> r) -> r }  
instance Comonad w => Monad (Co w) where ...
```

Given a comonad w , $\text{Co } w$ is a monad

Whats more, this new monad $\text{Co } w$ is right adjunct to w , meaning we get a way to move around $w \ a$ for free.

Co Zipper

Zippers are an example of a comonad with no obvious monad pairing.⁴

```
left :: Zipper a -> Zipper a
left (Zipper (l:ls) v rs) = Zipper ls l (v:rs)

-- type Co Zipper a = Co (Zipper (a -> r) -> r)

moveLeft :: Co Zipper ()
moveLeft = Co $ \z -> extract (left z) ()
```

⁴A Real-World Application with a Comonadic User Interface, Arthur Xavier, 2018

Handling arbitrary effects

Given that `Co w` is a monad, why not add another parameter `m` for effects

```
newtype CoT w m a = CoT { runCoT :: w (a -> m r) -> m r }
```

Message Passing

- Use Free Monads:
 - ▶ Functor `QueryF a` to model messages to a component
 - ▶ `eval :: Free QueryF a -> m a` evaluates these messages
-

Interesting Ideas

- Comonad transformer stacks
- $\text{Day } f$ is isomorphic to f