Comonadic Interface Design

Potentially the next big thing

Mitch Stevens

Comonads

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

```haskell
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

Figure 1: A Scomonad focused on something

[1] A 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

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

NonEmpty List

```
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 ...
```

NonEmpty Graph as a Comonad

```haskell
data NEGraph a = -- Complicated Stuff here

focusUpon :: NEGraph a -> a -> NEGraph a
focusUpon graph focus = -- TODO: focusUpon

instance Comonad NEGraph where
    extract = -- TODO: extract
    duplicate graph = fmap (focusUpon graph) graph
```

Other Comonads

- `Identity a`
- `(e, a)`
- Zippers
- `Trees with values in the branches (Cofree f)`

Kliesli and Cokliesli

- A function `a -> m b` is called a Kliesli arrow
- If `m` is a monad, we get Kliesli composition for free

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Kliesli and Cokliesli

- A function `a -> m b` is called a Kliesli arrow
- If `m` is a monad, we get Kliesli composition for free

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

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

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

Uses for Comonads

Image processing is a natural fit for Cokliesli composition[2]

we can focus on

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

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

[2] A 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

```haskell
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

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

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

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

Adjunctions

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

```haskell
-- 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 is 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 adjuction between monad/comonad transformers

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

The Reader/Env Pairing

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

The Reader/Env Pairing

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

Adjunction requirements:

```haskell
(w a -> b) -> (a -> m b)
(a -> m b) -> (w a -> 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 a new model for modeling UIs. This model can - Eas

```
type Component w = w UI

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

Comonadic Sum

A sum of comonads is itself a comonad

```haskell
-- from Data.Functor.Sum
data Sum f g a = InL (f a) | InR (g a)
instance (Comonad f, Comonad g) => Comonad (Sum f g) where
```

This would represent **two** UI components, with a single component visible at any given time.

For performance and ease of use, we need a comonad that can store both `f` and `g`.

Comonadic Sum

From the paper **Declarative UIs are the Future - And the Future is Comonadic**, we get a different comonadic sum

```
data Sum f g a = Sum Bool (f a) (g a)
instance (Comonad f, Comonad g) => Comonad (Sum f g) where ...
```

Comonadic Product

A comonadic product is poses problems; it doesn't have a comonad instance

```
-- from Data.Functor.Product
data Product f g a = Pair (f a) (g a)
```

Comonadic product

Again from **The future is Comonadic**, Freeman suggests using `Day` to represent a comonadic product:

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

We can use the Day convolution to make combinators for UI components

```
above, below, before, after :: f UI -> g UI -> Day f g UI
```

Homogenous Transformers

Monad Transformers

```haskell
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`, `Co w` is a monad

Co

`Co` is a heterogenous transformer

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

Given a comonad `w`, `Co w` is a monad

Whats more, this new monad `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 an comonad with no obvious monad pairing.[3]

```haskell
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) ()
```

[3] A 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
- `Day f` is isomorphic to `f`