Comonadic Interface Design

Potentially the next big thing

Mitch Stevens

History of Comonadic UIs

Phil Freeman wrote exploratory blog posts + a short paper about a novel way of creating UIs

History of Comonadic UIs

Phil Freeman wrote exploratory blog posts + a short paper about a novel way of creating UIs

Later Freeman would supervise a bachelor thesis by Arthur Xavier extending his orginal concept

History of Comonadic UIs

Phil Freeman wrote exploratory blog posts + a short paper about a novel way of creating UIs

Later Freeman would supervise a bachelor thesis by Arthur Xavier extending his orginal concept

This presentation is an overview of posts/papers on the subject thus far

Comonads

Whats a comonad?

- **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

- Using `extract`, we can extract the value that we were focusing on

Figure 1: A Scomonad focused on something

Extracting and Duplicating

- 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

NonEmpty List

```haskell
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
```

Comonads

Other Comonads

- `Identity a`
- `(e, a)`
- Zippers
- `Trees with values in the branches (Cofree f)`
- Streams (But not *actual* streaming libraries)

Uses for Comonads

Image processing is a natural fit for Cokliesli composition[1]

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

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

- Celullar automata
- Sudoku solvers

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

Component based UI

- Components are small, composable pieces of a UI
- They live in a hierarchy, the root component is the whole page
- These components pass messages, usually between parents and children
- They have their own internal state

Component based UI

- Components are small, composable pieces of a UI
- They live in a hierarchy, the root component is the whole page
- These components pass messages, usually between parents and children
- They have their own internal state

- This usually requires a lot of type variables

Component Example

```
data Component s = ...
```

Component Example

```
data Component s = ...

data Component s i o = ...
```

Component Example

```
data Component s = ...

data Component s i o = ...

data Component s i o (q :: * -> *) = ...
```

Component Example

```
data Component s = ...

data Component s i o = ...

data Component s i o (q :: * -> *) = ...

data Component s i o (q :: * -> *) (m :: * -> *) = ...
```

Component Example

Adding more type parameters for children, child query type, slot for addressing children yields

Component'

Source

```
type Component' h s f g p i o m = { initialState :: i -> s, render :: s -> h
    (ComponentSlot h g m p (f Unit)) (f Unit), eval :: f ~> (HalogenM s f g p
    o m), receiver :: i -> Maybe (f Unit), initializer :: Maybe (f Unit),
    finalizer :: Maybe (f Unit), mkOrdBox :: p -> OrdBox p }
```

The "private" type for a component.

- `h` is the type that will be rendered by the component, usually `HTML`
- `s` is the component's state
- `f` is the query algebra for the component itself
- `g` is the query algebra for child components
- `p` is the slot type for addressing child components
- `i` is the input value type that will be mapped to an `f` whenever the parent of this component renders
- `o` is the type for the component's output messages

Minimalist Components

- The only hard requirement is a rendering function

Minimalist Components

- The only hard requirement is a rendering function
- But we also want all the fancy stuff
  - Mutable state
  - initialiser, finaliser
  - preloaded data
  - other effects, etc

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
```

Repeated application of `duplicate` and `select` gives us a way to manipulate the component, but it is not clear what `x` should be

Adjunctions

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

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

- 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

```
type Component w = w UI

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

This is the bare minimum, we need much more than this

Other Gadgetry

- Sums of Components
- Products of Components
- Comonad Transformers
- Monads from Comonads (Action monads for free)
- Message Passing

Comonadic Sum

A sum of comonads is itself a comonad

```
-- 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

```haskell
-- 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
```

Haskell Transformers

Monad Transformers

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

Haskell 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
```

Comonad Transformer Stack

As with monads, comonad transformers also preserve comonad nature.

```haskell
data StoreT s w a = Store (w (s -> a)) s
```

Comonad Transformer Stack

As with monads, comonad transformers also preserve comonad nature.

```
data StoreT s w a = Store (w (s -> a)) s
```

As with monads, these transformers have classes so you don't have to dig through the stack

```
class ComonadStore s w | w -> s where ...
```

Monads from Comonads

Co is a heterogenous transformer

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

Monads from Comonads

`Co` is a heterogenous transformer

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

Whats more, this new monad `Co w` is paired with `w`, meaning we get a way to move around `w a` for free.

Parents and Children

Using `StoreT`, we can embed child comonads

```
type Parent = StoreT s Child
```

We can also lift the actions for the child into actions for the parent

```
liftAction :: ComonadTrans t => Co w a -> Co (t w) a
```

Co Zipper actions

Zippers are an example of an comonad with no obvious monad pairing.[2]

```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) ()
```

[2] 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 }
```

Since `CoT w` is a monad transformer, we can lift arbitrary effects into

Unfortunatly `CoT w m a` does not pair with `w`. It is unknown how this might be solved.

Message Passing

Use `Cofree f` as a base comonad, where `f` is a query algebra:

`Cofree f` is adjoint to `Free f`, so we have a very familiar way to sequence messages

Yet to be solved

This model has some wrinkles:

- Message passing between components is not
- `CoT w m a` does not pair with `w` anymore