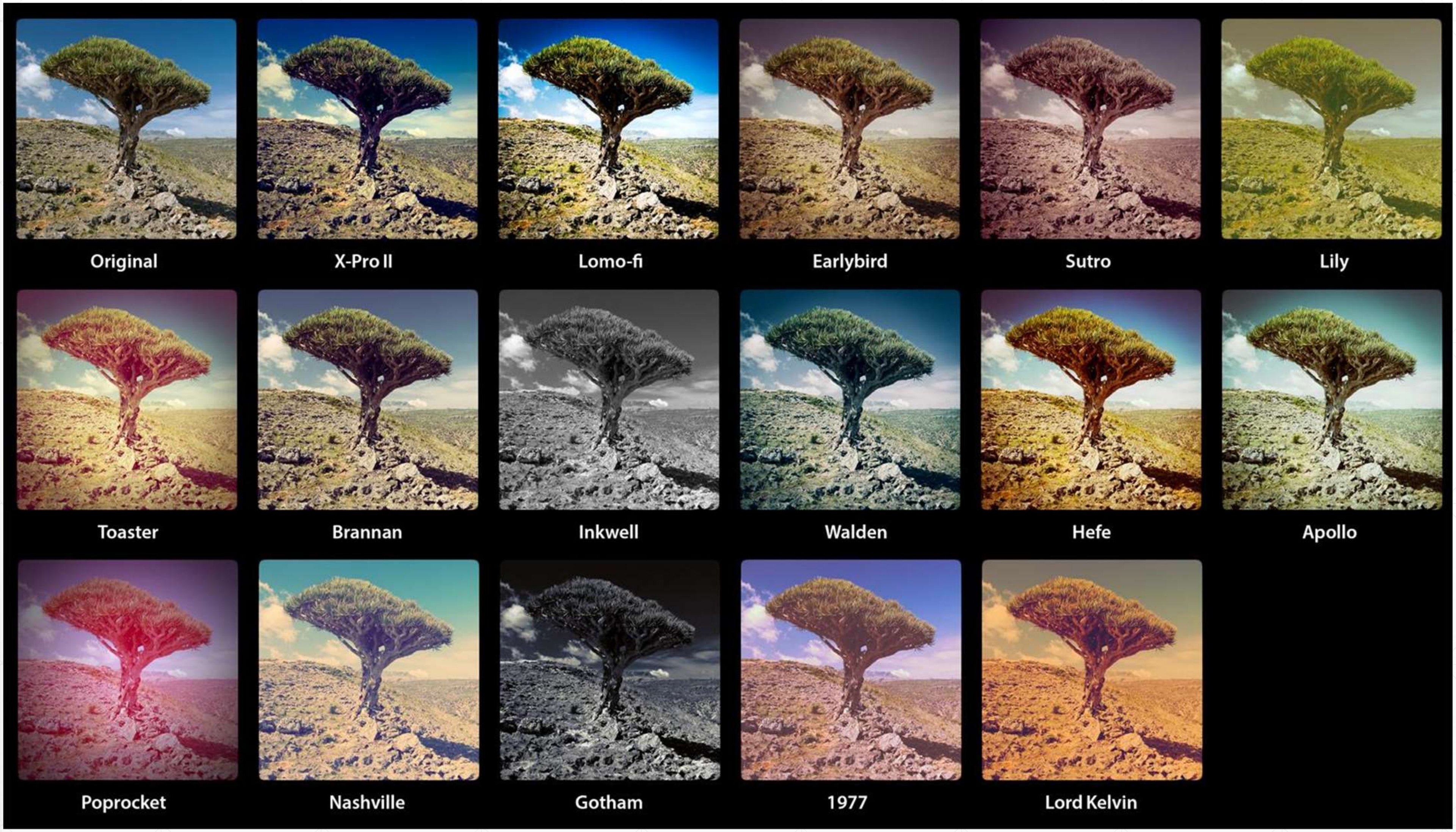
Functors, Comonads, and Digital Image Processing

Lambdaconf 2016, Boulder Colorado, Justin Le (http://jle.im) (github.com/mstksg)



getglasses.com.au

Image Filters!

- Blur
- Colorize
- Rotate
- Skew
- Edge-detect
- Sharpen
- Laplacians
- Visibility masks

The Traditional Image Filter

- `foo :: Image -> Image`
- `type Image = [Pixel]`

  ```
  foo :: [Pixel] -> [Pixel]
  ```
- `bar :: [Pixel] -> [Pixel]`

  ```
  fooThenBar :: [Pixel] -> [Pixel]
  fooThenBar = bar . foo
  ```

What’s in a Type?

- The **less structure**, the **more information**
  - Developer intent
  - Restriction of implementations
  - Algorithmic simplicity
  - Equational manipulability
- It’s simply the *Haskell Way*™!

What does [Pixel] -> [Pixel] tell us?

- Nothing
- Well, nothing useful.
- It’s pure!

What do you think!

- What's wrong with `[Pixel] -> [Pixel]`? (Besides it being a list)
- Problems for the **writer**
  - Too many ways to implement incorrectly
  - Extra things to worry about:
    - How to handle bounadries?
    - Parallelism?
- Problems for the **user**, with respect to **composition**
  - Parallel composition?
  - Algorithmic (structural) composition?

The Functor Design Pattern

- http://www.haskellforall.com/2012/09/the-functor-design-pattern.html
- **F**unctor **D**riven **D**evelopment
- **FDD** Manifesto:
  - Choosing the Category/Subcategory you work in gives you power
  - Feel free to write different parts of your logic in different categories
  - Write **Functors** to unite them how you please

You wouldn’t steal a car

- You wouldn’t:

  -
    ```
    lenSqs :: StateT Int (MaybeT IO) [a] -> StateT Int (MaybeT IO) Int
    lenSqs action = do
      l <- action
      return (length l ^ 2)
    ```
- Functor-aware:

  -
    ```
    lenSq :: [a] -> Int
    lenSq l = length l ^ 2
    ```
- Composing:
  - `lenSqs . lenSqs`
  - `fmap lenSq . fmap lenSq` ***or*** `fmap (lenSq . lenSq)`

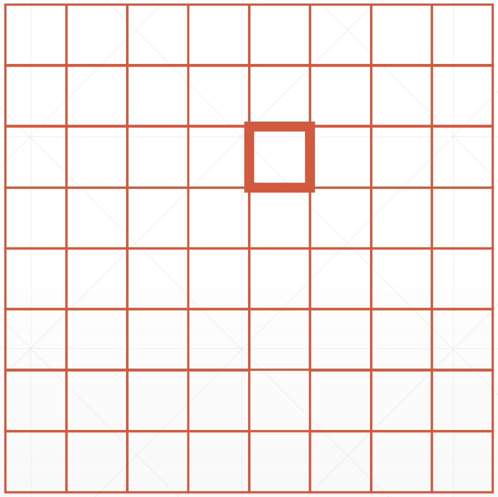
Big gain if fmap is inefficient!!

$$\begin{tikzcd}
 & \texttt{f, g :: a -> a} \arrow[dl, "\text{compose}"'] \arrow[dr, "\text{fmap}"] & \\
\texttt{f . g} \arrow[d, "\text{fmap}"'] & & \texttt{fmap f, fmap g} \arrow[d, "\text{compose}"] \\
\boxed{\texttt{fmap (f . g)}} & & \texttt{fmap f . fmap g}
\end{tikzcd}$$

`:: f a -> f a`

The Search for Better Types

Image With Focus



- Store an image *with* a “focused” index

-
  ```
  data Focused a = F [a] Coord
  ```

-
  ```
  toFocused :: [a] -> Focused a
  toFocused xs = F xs 0
  ```

-
  ```
  fromFocused :: Focused a -> [a]
  fromFocused (F xs _) = xs
  ```

-
  ```
  extract :: Focused a -> a
  extract (F xs c) = xs !! c
  ```

-
  ```
  instance Functor Focused where
    fmap f (F xs c)
        = F (fmap f xs) c
  ```

Arrows as Filters

- `Focused a -> b`
  - “Specify the new pixel value at that location”
- `(Focused a -> b) -> (Focused a -> [b])`
  - “From a specification of a new pixel value at a given location, create a whole new image”
-
  ```
  extendOut :: (Focused a -> b) -> (Focused a -> [b])
  extendOut f (F xs c) = fmap (\d -> f (F xs d)) allCoords
  ```
-
  ```
  extend :: (Focused a -> b) -> (Focused a -> Focused b)
  extend f foc@(F _ c) = F (extendOut foc) c
  ```

Composition

- `(Focused b -> c) -> (Focused a -> b) -> (Focused a -> c)`
  - Sequencing two “`Focused a -> b`”s

-
  ```
  (=<=) :: (Focused b -> c)
        -> (Focused a -> b)
        -> (Focused a -> c)
  (=<=) f g = \x -> let y :: Focused b
                        y = extend g x
                    in  f y
  ```

  (potentially inefficient)

Déjà vu

-
  ```
  extract :: Focused a -> a
  extend  :: (Focused a -> b) -> (Focused a -> Focused b)
  (=<=)   :: (Focused b -> c) -> (Focused a -> b)
                              -> (Focused a -> c)
  ```

-
  ```
  extract :: w a -> a
  extend  :: (w a -> b) -> (w a -> w b)
  (=<=)   :: (w b -> c) -> (w a -> b  ) -> (w a -> c  )
  ```

-
  ```
  return  :: a -> m a
  (=<<)   :: (a -> m b) -> (m a -> m b)  -- aka “bind”
  (<=<)   :: (b -> m c) -> (a   -> m b) -> (a   -> m c)
  ```

Anti-Monads

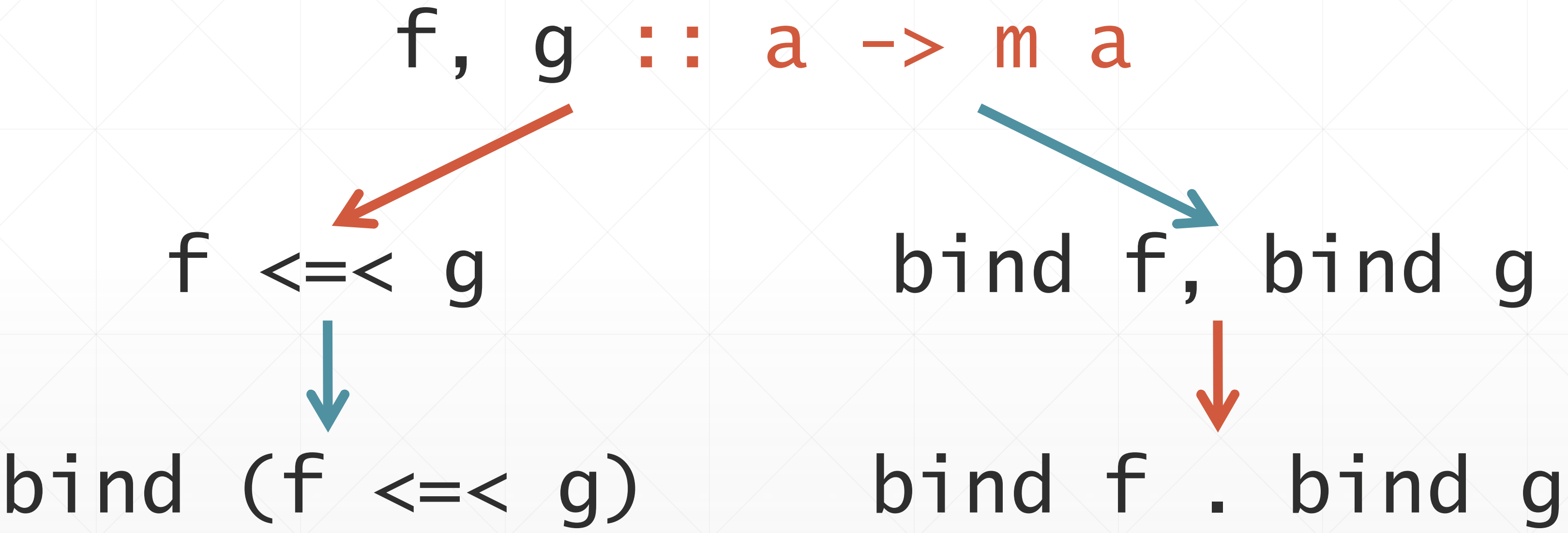
Star Trek

Comonads

-
  ```
  class Functor w => Comonad w where
      extract :: w a -> a
      extend  :: (w a -> b) -> (w a -> w b)
  ```
- “It is well known that *effects* correspond to *monads* … quite interestingly, *coeffects* correspond to the dual concept called *comonads*” *(Tomas Petricek)*
- **extract**
  - From *this context*, yield a value.
- **extend**
  - Take your fancy (`w a -> b`) jargon and bring it back into the real world like everyone else
  - Turned our Comonadic CoKleisli ~~Co~~filter back into a “normal” traditional filter.

Why Monads?

Because Math

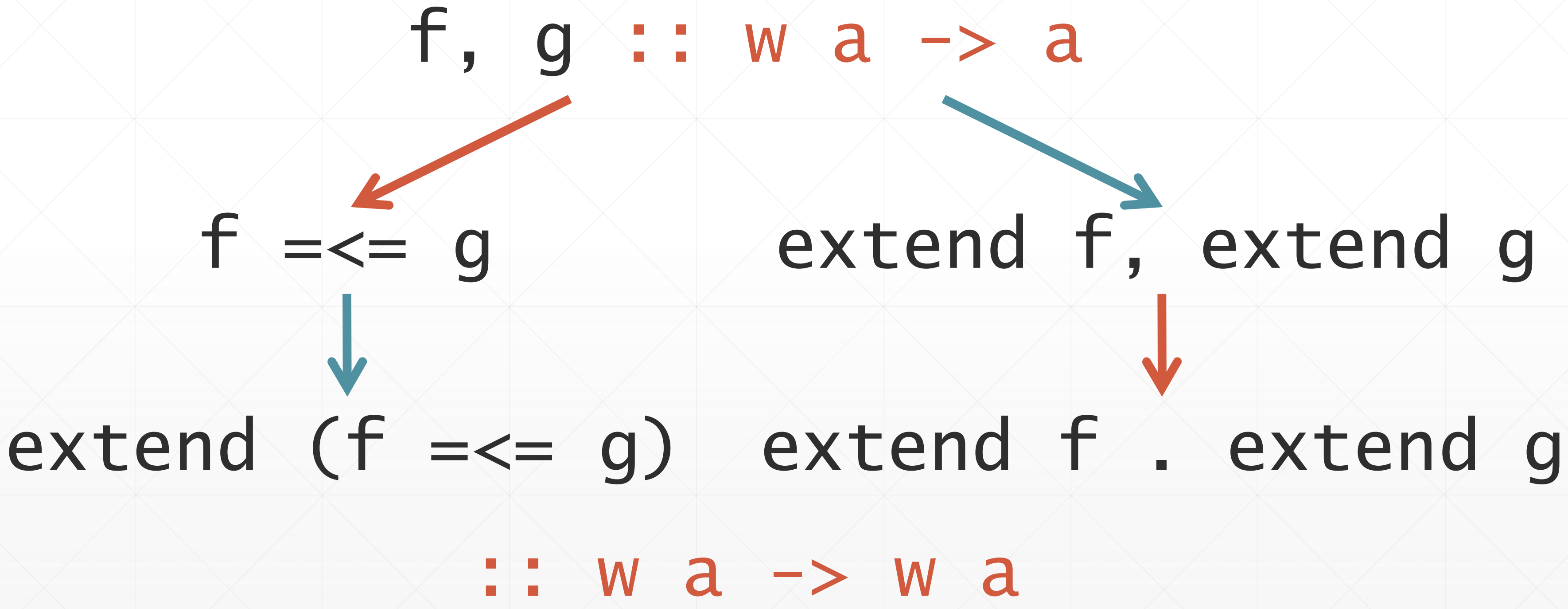


```
f, g :: a -> m a

f <=< g
bind f, bind g

bind (f <=< g)
bind f . bind g
```

```
:: m a -> m a
```

Why Comonads?

Because Math (Part Deux: Comonads)



```
f, g :: w a -> a

f =<= g                    extend f, extend g

extend (f =<= g)           extend f . extend g

:: w a -> w a
```

Power of Choice

- Many ways to implement `extend` and `(=<=)`!
  - Parallelism and concurrency
  - Memoization
  - Behavior at boundaries
- Comonad laws + Equational Reasoning allow us to **interchange** and **reassociate**.
- We can stay “in the world of CoKleislis” for efficient composition
  - Exit only at the end with the final `extend`.
  - https://blog.jle.im/entry/inside-my-world-ode-to-functor-and-monad.html

Laws

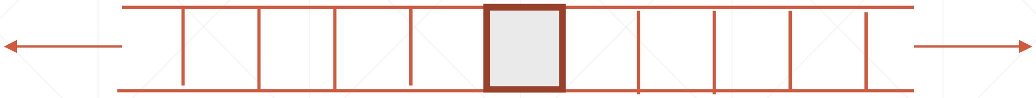
Comonads

- `extend extract = id`
- `extract =<= f  = f`
- `extract . extend f = f`
- etc.
- (Just trust me)

Monads

- `bind return  = id`
- `return <=< f = f`
- `bind f . return = f`
- etc.

Neighborhoods



-
  ```
  data Tape a = Tape { tLeft  :: [a]
                     , tVal   :: a
                     , tRight :: [a]
                     }
  ```

-
  ```
  shiftRight :: Tape a -> Tape a
  shiftRight (Tape (l:ls) v rs) = Tape ls l (v:rs)
  ```

-
  ```
  shiftRightN :: Int -> Tape a -> Tape a
  shiftRightN n t = iterate shiftRight t !! n
  ```

Arrows as Local Filters

-
  ```
  blur :: Fractional a => Tape a -> a
  blur (Tape (l:_) v (r:_)) = (l + v + r) / 3
  ```

-
  ```
  sharpen :: Num a => Tape a -> a
  sharpen (Tape (l:_) v (r:_)) = 2*v - l - r
  ```

-
  ```
  deriv :: Fractional a => Tape a -> a
  deriv (Tape (l:_) v (r:_)) = ((v - l) + (r - v)) / 2
  ```

Comonads Everywhere

- **It makes sense** to “compose” (`Tape a -> b`)’s
  - That’s the smell of a comonad!

-
  ```
  instance Comonad Tape where
    extract (Tape _ v _)       = v
    extend f t@(Tape ls _ rs) = Tape ls’ (f t) rs’
      where
        (_:ls’) = fmap f (iterate shiftRight t)
        (_:rs’) = fmap f (iterate shiftLeft  t)
  ```
- The power of composition
  - `deriv2     = deriv <=< deriv`

Functors and Natural Transformations

-
  ```
  globalize :: d -> (Tape a -> b) -> (Focused a -> b)
  globalize d f (F xs _) = f (listToTape xs)
    where
      listToTape (x:xs) = Tape (repeat d) x (xs ++ repeat d)
  ```

- A **Functor** from the **Tape Cokliesli Category** to the **Focused Cokleisli Category**, where the morphism mapper is `globalize d`.

- We have some choices!
  - Boundary behaviors?

A New World of CoKleisli Categories

- Arrows from different categories are everywhere
  - Kernel matrices
  - Affine transformation matrices
  - Finite or dependently typed neighorhoods
- Be creative with Functors, get assurances with mathematics
- Dimension agnostic:
  - Videos + Compression
  - Physical simulations
  - Difference equation modeling

From f, g :: Tape a -> a …

```
extend (glob (f <=< g))
```

- or -

```
extend (glob f <=< glob g)
```

- or -

```
extend (glob f) . extend (glob g)
```

```
:: Focused a -> Focused a
```

More Resources

- http://hub.darcs.net/ertes/articles/browse/media-processing.lhs
  - Media Processing -- Ertugrul Söylemez
- https://jaspervdj.be/posts/2014-11-27-comonads-image-processing.html
  - Image Processing with Comonads -- Jasper Van der Jeugt
- https://github.com/mstksg/lambdaconf-2016-usa/tree/master/Functors, Comonads, and Digital Image Processing
  - Slides online