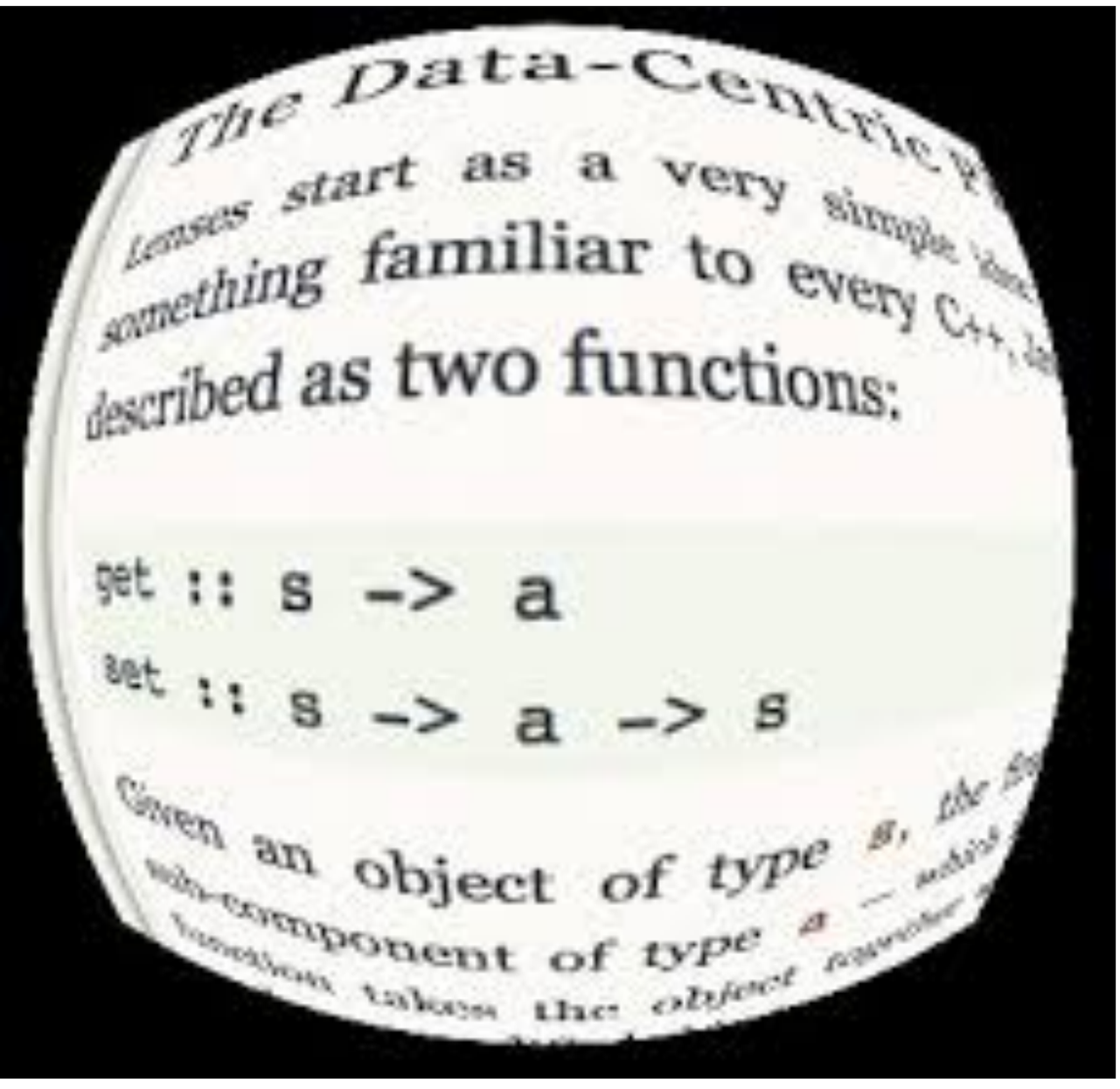
LENSES

For the mere mortal

Lorenzo López-Sancho Abraham

source code & slides: *https://github.com/batou99/haskell_lenses*



WHAT ARE LENSES

- A Lens is an abstraction from functional programming which helps to deal with a problem of updating complex immutable nested objects

BASIC TYPES

```
type Point2D = (Float, Float)

origin :: Point2D
origin = (0, 0)
```

VIEW

```haskell
viewX :: Point2D -> Float
viewX (x, _) = x

viewY :: Point2D -> Float
viewY (_, y) = y
```

SET

```haskell
setX :: Float -> Point2D -> Point2D
setX newX (_, y) = (newX, y)

setY :: Float -> Point2D -> Point2D
setY newY (x, _) = (x, newY)
```

OVER

```haskell
overX :: (Float -> Float) -> Point2D -> Point2D
overX fn (x, y) = (fn x, y)
```

```haskell
overY :: (Float -> Float) -> Point2D -> Point2D
overY fn (x, y) = (x,fn y)
```

TUPLES

LENS

```haskell
xLens :: Lens' Point2D Float
xLens = lens getter setter
  where
    getter = fst
    setter (_, y) newX = (newX, y)
```

LENS (BETTER)

```haskell
yLens :: Lens' Point2D Float
yLens = _2
```

STAB

```
forall s t a b. Lens s t a b
```

- s (Original whole)
- t (New whole)
- a (Original focus)
- b (New focus)

LENS (MORE GENERAL)

```haskell
yGeneralLens :: Lens' (a, b) b
yGeneralLens = _2
```

LENS (EVEN MORE GENERAL)

```haskell
yEvenMoreGeneralLens :: Lens (ignored, a) (ignored, b) a b
yEvenMoreGeneralLens = _2
```

LENS LAWS

what makes a lens a lens?

SET-GET

- view views what set sets

```
setGet = view _1 $ set _1 10 origin -- 10
```

GET-SET

- if you set what you get, nothing changes

```
getSet = set _1 (view _1 origin) origin -- No changes
```

SET-SET

- set is idempotent

```
-- Idempotent
setSet = set _1 4 $ set _1 4 origin
```

RECORDS

RECORDS

```haskell
data Person = P { _name :: String
                , _addr :: Address
                , _salary :: Int }  deriving Show

data Address = A { _road :: String
                 , _city :: String
                 , _postcode :: String } deriving Show
```

FIRST LEVEL RECORDS

```haskell
viewName :: Person -> String
viewName = _name

setName :: String -> Person -> Person
setName name person = person { _name = name }

overName :: (String -> String) -> Person -> Person
overName fn person = person { _name = fn $ _name person }
```

NESTED RECORDS

```haskell
viewPersonRoad :: Person -> String
viewPersonRoad = _road . _addr

setPersonRoad :: String -> Person -> Person
setPersonRoad newRoad person = person { _addr = newAddr }
  where
    oldAddr = _addr person
    newAddr = oldAddr { _road = newRoad }

overPersonRoad :: (String -> String) -> Person -> Person
overPersonRoad fn person = setPersonRoad (fn oldPersonRoad) person
  where
    oldPersonRoad = viewPersonRoad person
```

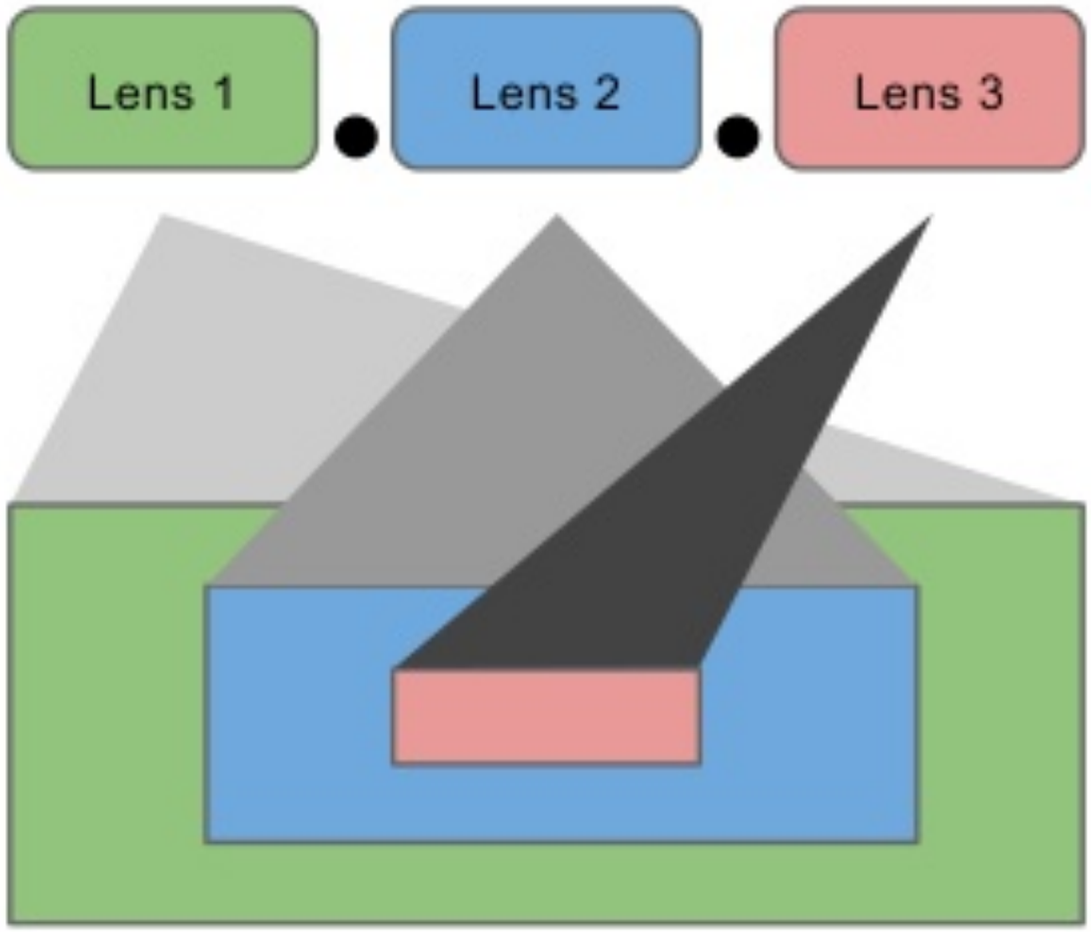
WITH LENSES

```haskell
salaryLens :: Lens' Person Int
salaryLens = lens _salary (\p s -> p { _salary = s })

addrLens :: Lens' Person Address
addrLens = lens _addr (\p a -> p { _addr = a})

roadLens :: Lens' Address String
roadLens = lens _road (\a r -> a { _road = r })
```

COMPOSITION




```haskell
personRoadLens :: Lens' Person String
personRoadLens = addrLens . roadLens
```

EXAMPLES

```haskell
viewSalary :: Person -> Int
viewSalary = view salaryLens

setSalary :: Int -> Person -> Person
setSalary = set salaryLens

overSalary :: (Int -> Int) -> Person -> Person
overSalary = over salaryLens

viewRoad :: Person -> String
viewRoad = view $ addrLens . roadLens
```

AUTOLENSES

```
makeLenses ''Person
makeLenses ''Address
```

MAPS

TYPES

```haskell
data Product = Product { _productId :: Int
                       , _productName :: String
                       , _productPrice :: Float } deriving Show

type Catalog = Map Int Product

data Store = Store { _storeName :: String
                   , _storeCatalog :: Catalog } deriving Show
```

```haskell
makeLenses ''Product
makeLenses ''Store
```

MANUAL LENS

```haskell
product1Lens :: Lens' Catalog (Maybe Product)
product1Lens = lens getter setter
  where
    getter = Map.lookup 1
    setter catalog wrapped =
      case wrapped of
        Just prod -> Map.insert 1 prod catalog
        Nothing   -> Map.delete 1 catalog
```

GENERAL LENS

```haskell
atKeyLens :: Int -> Lens' Catalog (Maybe Product)
atKeyLens key = lens getter setter
  where
    getter = Map.lookup key
    setter catalog wrapped =
      case wrapped of
        Just prod -> Map.insert key prod catalog
        Nothing   -> Map.delete key catalog
```

AUTO LENS

```haskell
atKeyLens' :: Int -> Lens' Catalog (Maybe Product)
atKeyLens' = at
```

COMPOSITION (1)

```haskell
storeProduct1 :: Lens' Store (Maybe Product)
storeProduct1 = storeCatalog . at 1
```

COMPOSITION (2)

```haskell
storeProduct1 :: Lens' Store (Maybe Product)
storeProduct1 = storeCatalog . at 1
```

TRAVERSALS

operating on whole collections

PREDEFINED OPTIC (FOR ANY TRAVERSABLE)

```
set traversed 3 [1, 2] -- [3, 3]

over traversed negate [1, 2] -- [-1, -2]
over traversed negate (Just 3) -- (Just -3)

view traversed ["g", "o", "o", "d"] -- "good"

toListOf traversed [1, 2, 3] -- [1, 2, 3]
```

VIEW FOR NON MONOIDAL TYPES

```
getSum $ view traversed $ over traversed Sum [1, 2, 3] -- 6
```

FOCUSING ON INDIVIDUAL VALUES

```
firstOf traversed [1, 2, 3] -- Just 1
preview traversed [1, 2, 3] -- Just 1

lastOf traversed [1, 2, 3] -- Just 3

over (element 1) (*2) [1, 2, 3] -- [1, 4, 3]
```

COMPOSING TRAVERSALS (1)

```haskell
negateAll :: [[Int]] -> [[Int]]
negateAll = over (traversed . traversed) negate

negateAll [[1, 2], [3]] -- [[-1, -2], [-3]]
```

UNWRAP WITH TRAVERSED

- traversed optic can be also used to unwrap the type after another optic.

```haskell
discountProduct :: Store -> Int -> Float -> Store
discountProduct store pid percent =
  over storeProductPriceLens (* percent) store
  where
    storeProductPriceLens =
      storeCatalog . at pid . traversed . productPrice
```

SINGLE ELEMENTS

in arrays and fixed-size maps

INDEX

```
over (ix 1) negate [1, 2, 3] -- [1, -2, 3]
over (element 1) negate [1, 2, 3] -- [1, -2, 3]
```

INDEX: COMPARING WITH AT (1)

```haskell
_at1 = at 1
_ix1 = ix 1

m = Map.singleton 1 "a"

view _at1 m     -- (Just "a")
view _ix1 m     -- "a" (monoidal append)
preview _ix1 m --(Just "a")
```

INDEX: COMPARING WITH AT (2)

- Because an At lens can create or delete, set takes a Maybe

```
set _at1 (Just "new value") m -- (fromFoldable [(Tuple 1 "new value")])
```

- Because an Index optic doesn't create or delete, it takes an unwrapped value

```
set _ix1 "new value" m -- (fromFoldable [(Tuple 1 "new value")])
```

- if the Index optic is used to set an object that doesn't exist. The structure is silently unchanged

```
set _ix1 "new value" Map.empty -- (fromFoldable [])
```

INDEX: COMPARING WITH AT (3)

- Index lenses can be also used with arrays (At lenses can't)

```
preview _ix1 [1, 2] -- (Just 2)

set _ix1 "ignored" [] -- []

over _ix1 negate [0, 1] -- [0,-1]
```

PRISMS

lenses over sum types

SUM TYPES

```
newtype Percent = Percent Float
  deriving (Eq, Show)
data Point      = Point Float Float
  deriving (Eq, Show)

data Fill
  = Solid Color
  | LinearGradient Color Color Percent
  | RadialGradient Color Color Point
  | NoFill deriving (Eq, Show)

data Color = Color Int Int Int Float
  deriving (Eq, Show)
```

USAGE

- Only get something if color is solid

```
preview _solidFill $ Solid Color.white
-- Just Color 255 255 255 1.0
preview _solidFill  NoFill -- Nothing

review _solidFill Color.white
-- Solid Color 255 255 255 1.0

set _solidFill Color.white $ Solid Color.black
-- Solid Color 255 255 255 1.0
set (_1 . _solidFill) Color.white (Solid Color.black, 5)
-- (Solid Color 255 255 255 1.0, 5)
```

MAKING A PRISM

```haskell
_solidFill :: Prism' Fill Color
_solidFill = prism' constructor focuser
  where
    constructor = Solid
    focuser fill = case fill of
      Solid color -> Just color
      _other      -> Nothing

_solidWhite :: Prism' Fill Unit
_solidWhite = only (Solid Color.white)

-- is _solidWhite $ Solid Color.black :: Boolean

_solidWhite' :: Prism' Fill Unit
_solidWhite' = nearly (Solid Color.white) case _ of
  Solid color -> color == Color.white
  _ -> false
```

PRISM LAWS

- review-preview

```haskell
reviewPreview :: Maybe Color
reviewPreview = preview _solidFill solidColorWhite
  where
    solidColorWhite = review _solidFill Color.white
```

- preview-review

```haskell
previewReview :: Maybe Fill
previewReview = review _solidFill <$> maybeColor
  where
    maybeColor = preview _solidFill $ Solid Color.white
```

VALUE CONSTRUCTORS WITH MORE THAN ONE ARGUMENT

- This is not a valid lens

```
_centerPoint :: Prism' Fill Point
_centerPoint = prism' constructor focuser
  where
    focuser x = case x of
                  RadialGradient _ _ point -> Just point
                  _ -> Nothing
    constructor = RadialGradient Color.black Color.white
```

VALUE CONSTRUCTORS WITH MORE THAN ONE ARGUMENT

- This is a valid lens

```haskell
data RadialInterchange = RI Color Color Point

centerPoint' :: Prism' Fill RadialInterchange
centerPoint' = prism' constructor focuser
  where
  constructor (RI color1 color2 center) =
    RI color1 color2 center
  focuser x =
    case x of
      RI color1 color2 center -> Just $ RI color1 color2 center
      _ -> Nothing
```

CREDITS

- Brian Marick (Lenses for the mere mortal)
- https://leanpub.com/lenses