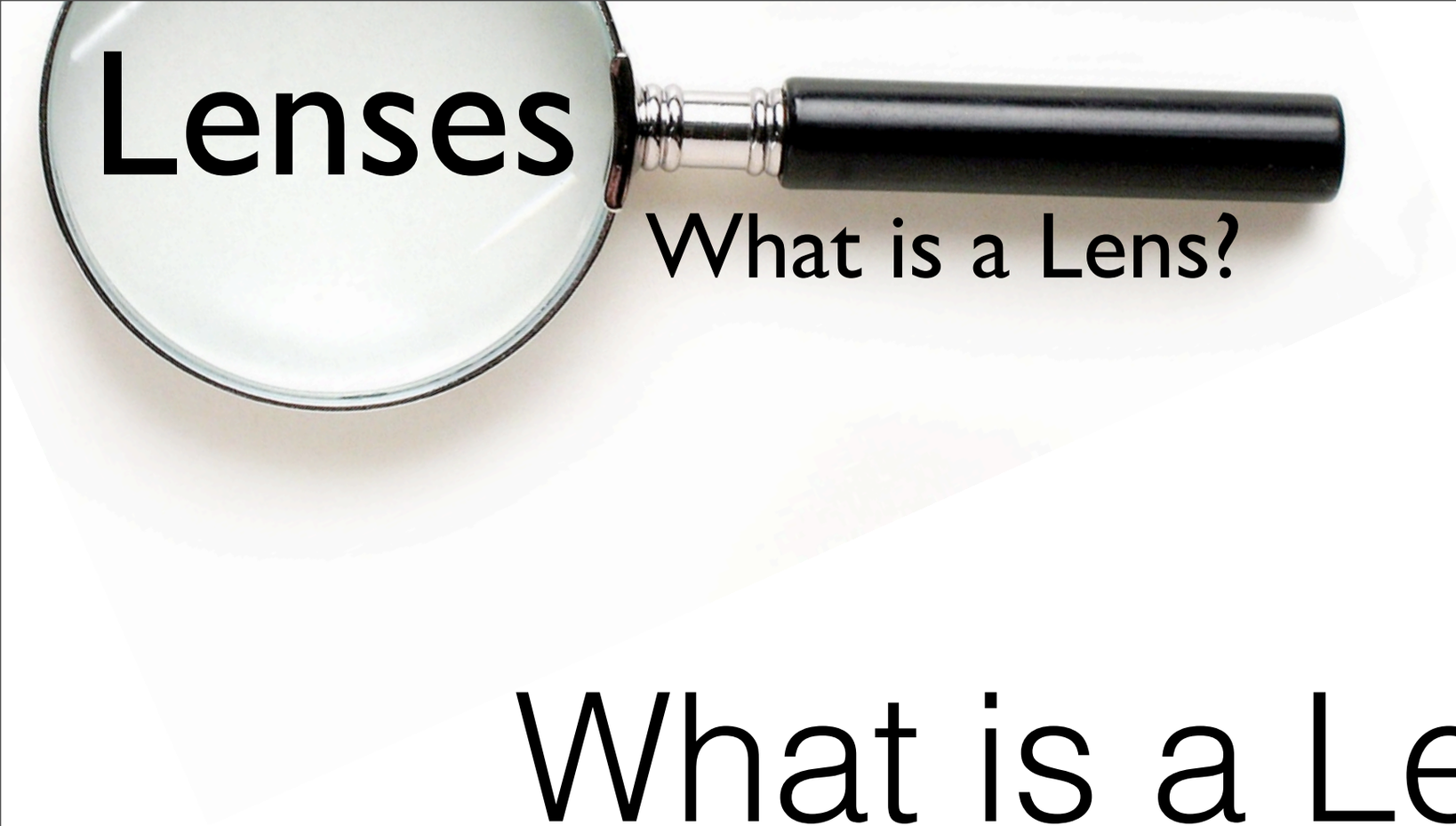
Lenses, Folds, and Traversals

Edward Kmett

Lenses

Overview

- What is a Lens?
- “The Power is in the Dot”
  - Semantic Editor Combinators
  - Setters
  - Traversals
  - Folds
  - Lenses
  - Getters
  - Overloading Application
- Extras
  - Uniplate
  - Zippers



What is a Lens?

Lenses

What is a Lens?

Costate Comonad Coalgebra
is equivalent of Java's
member variable update
technology for Haskell

@PLT_Borat

Lenses

What is a Lens?

```
data Lens s a = Lens { set  :: s -> a -> s
                     , view :: s -> a
                     }

view :: Lens s a -> s -> a
set  :: Lens s a -> s -> a -> s
```

Laws:

```
1.) set l (view l s) s = s
2.) view l (set l s a) = a
3.) set l (set l s a) b = set l s b
```

Lenses

What is a Lens?

```
data Lens s a = Lens { set  :: s -> a -> s
                     , view :: s -> a
                     }
```

Lenses

What is a Lens?

```
data Lens s a = Lens { set  :: s -> a -> s
                     , view :: s -> a
                     }
```

Fusion (and data-lens).

```
data Lens s a = Lens (s -> (a -> s, a))
```

Lenses

What is a Lens?

```
data Lens s a = Lens { set  :: s -> a -> s
                     , view :: s -> a
                     }
```

Fusion (and data-lens).

```
data Lens s a = Lens (s -> (a -> s, a))

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

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

Lenses

What is a Lens?

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

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

instance Category Lens where
  id = Lens (Store id)
  Lens f . Lens g = Lens $ \r -> case g r of
    Store sr s -> case f s of
      Store ts t -> Store (sr . ts) t
```

The Power is in the Dot

Lenses

The Power is in the Dot

```
(.)          :: (a -> b) -> (c -> a) -> c -> b
```

Lenses

The Power is in the Dot

```
(.)          :: (a -> b) -> (c ->          a) -> c          -> b
```

Lenses

The Power is in the Dot

```
(.)        :: (a -> b) -> (c ->          a) -> c          -> b
(.).(.)    :: (a -> b) -> (c -> d ->     a) -> c -> d     -> b
```

Lenses

The Power is in the Dot

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b
```

Lenses

The Power is in the Dot

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

Semantic Editor Combinators

Lenses

Semantic Editor Combinators

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

These are sometimes known as **Semantic Editor Combinators**.

```
result = (.)

element = fmap

second = fmap

first f (a,b) = (f a, b)
```

Lenses

Semantic Editor Combinators

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

These are sometimes known as **Semantic Editor Combinators**.

```
type SEC s t a b = (a -> b) -> s -> t

result :: SEC (e -> a) (e -> b) a b
result = (.)
element :: SEC [a] [b] a b
element = fmap
second :: SEC (c,a) (c,b) a b
second = fmap
first :: SEC (a,c) (b,c) a b
first f (a,b) = (f a, b)
```

Lenses

Semantic Editor Combinators

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

These are sometimes known as **Semantic Editor Combinators**.

```
type SEC s t a b = (a -> b) -> s -> t

fmap :: Functor f => SEC (f a) (f b) a b
```

```
first :: SEC (a,c) (b,c) a b
first f (a,b) = (f a, b)
```

Setters

Lenses

Setters

```
(.)         :: (a -> b) -> (c ->           a) -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d ->      a) -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

```
traverse                    :: (Traversable f, Applicative m)
  => (a -> m b) -> f a          -> m (f b)
traverse.traverse           :: (Traversable f, Traversable g, Applicative m)
  => (a -> m b) -> f (g a)      -> m (f (g b))
traverse.traverse.traverse :: (Traversable f, Traversable g, Traversable h, Applicative m)
  => (a -> m b) -> f (g (h a)) -> m (f (g (h b)))
```

Lenses

Setters

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
class (Functor f, Foldable f) => Traversable f where
  traverse :: Applicative m => (a -> m b) -> f a -> m (f b)
  ...
```

Lenses

Setters

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
```

```
traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)
```

Lenses

Setters

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
```

```
traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)
```

Lenses

Setters

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault f = runIdentity . traverse (Identity . f)
```

```
traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)

newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance Applicative Identity where
  pure = Identity
  Identity f <*> Identity x = Identity (f x)
```

Lenses

Setters

```
fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))

fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault f = runIdentity . traverse (Identity . f)
```

```
traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)

newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance Applicative Identity where
  pure = Identity
  Identity f <*> Identity x = Identity (f x)
```

Lenses

Setters

```
fmap            :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap       :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap  :: (Functor f, Functor g, Functor h)  => (a -> b) -> f (g (h a)) -> f (g (h b))
```

```
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault = runIdentity . traverse (Identity . f)
```

```
over l f = runIdentity . l (Identity . f)
```

Lenses

Setters

```
fmap           :: Functor f                        => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)           => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

```
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault = runIdentity . traverse (Identity . f)
```

```
over l f = runIdentity . l (Identity . f)

over traverse f = runIdentity . traverse (Identity . f)
                = fmapDefault f
                = fmap f
```

Lenses

Setters

```
fmap           :: Functor f                        => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)           => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

```
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault = runIdentity . traverse (Identity . f)

over :: (a -> Identity b) -> s -> Identity t) -> (a -> b) -> s -> t
over l f = runIdentity . l (Identity . f)

over traverse f = runIdentity . traverse (Identity . f)
                = fmapDefault f
                = fmap f
```

Lenses

Setters

```
fmap           :: Functor f                        => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)           => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))
```

```
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault = runIdentity . traverse (Identity . f)

over :: Setter s t a b -> (a -> b) -> s -> t
over l f = runIdentity . l (Identity . f)

over traverse f = runIdentity . traverse (Identity . f)
                = fmapDefault f
                = fmap f

type Setter s t a b = (a -> Identity b) -> s -> Identity t
```

Lenses

Setters

```
mapped                :: Functor f => Setter (f a) (f b) a b
mapped.mapped         :: (Functor f, Functor g) => Setter (f (g a)) (f (g b)) a b
mapped.mapped.mapped :: (Functor f, Functor g, Functor h) => Setter (f (g (h a))) (f (g (h b))) a b
```

```
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault = runIdentity . traverse (Identity . f)

over :: Setter s t a b -> (a -> b) -> s -> t
over l f = runIdentity . l (Identity . f)

over traverse f = runIdentity . traverse (Identity . f)
                = fmapDefault f
                = fmap f

type Setter s t a b = (a -> Identity b) -> s -> Identity t

mapped :: Functor f => Setter (f a) (f b) a b
mapped f = Identity . fmap (runIdentity . f)

over mapped f = runIdentity . mapped (Identity . f)
              = runIdentity . Identity . fmap (runIdentity . Identity . f)
              = fmap f
```

Lenses

Setters

```
mapped                :: Functor f => Setter (f a) (f b) a b
mapped.mapped         :: (Functor f, Functor g) => Setter (f (g a)) (f (g b)) a b
mapped.mapped.mapped :: (Functor f, Functor g, Functor h) => Setter (f (g (h a))) (f (g (h b))) a b

over :: Setter s t a b -> (a -> b) -> s -> t

over mapped :: Functor f => (a -> b) -> f a -> f b
over (mapped.mapped) :: (Functor f, Functor g) => (a -> b) -> f (g a) -> f (g b)
...

chars :: (Char -> Identity Char) -> Text -> Identity Text
chars f = fmap pack . mapped f . unpack

over chars :: (Char -> Char) -> Text -> Text
over (mapped.chars) :: Functor f => (Char -> Char) -> f Text -> f Text
over (traverse.chars) :: Traversable f => (Char -> Char) -> f Text -> f Text
```

Lenses

Setters

```
mapped                :: Functor f => Setter (f a) (f b) a b
mapped.mapped         :: (Functor f, Functor g) => Setter (f (g a)) (f (g b)) a b
mapped.mapped.mapped :: (Functor f, Functor g, Functor h) => Setter (f (g (h a))) (f (g (h b))) a b

over :: Setter s t a b -> (a -> b) -> s -> t

over mapped :: Functor f => (a -> b) -> f a -> f b
over (mapped.mapped) :: (Functor f, Functor g) => (a -> b) -> f (g a) -> f (g b)
...

chars :: (Char -> Identity Char) -> Text -> Identity Text
chars f = fmap pack . mapped f . unpack

over chars :: (Char -> Char) -> Text -> Text
over (mapped.chars) :: Functor f => (Char -> Char) -> f Text -> f Text
over (traverse.chars) :: Traversable f => (Char -> Char) -> f Text -> f Text
```

Lenses

Setters

```
mapped                :: Functor f => Setter (f a) (f b) a b
mapped.mapped         :: (Functor f, Functor g) => Setter (f (g a)) (f (g b)) a b
mapped.mapped.mapped :: (Functor f, Functor g, Functor h) => Setter (f (g (h a))) (f (g (h b))) a b

over :: Setter s t a b -> (a -> b) -> s -> t
```

Functor Laws:

1.) fmap id = id
2.) fmap f . fmap g = fmap (f . g)

Setter Laws for a legal Setter *l*.

1.) over *l* id = id
2.) over *l* f . over *l* g = over *l* (f . g)

```
both :: Setter (a,a) (b,b) a b
both f (a,b) = (,) <$> f a <*> f b

first :: Setter (a,c) (b,c) a b
first f (a,b) = (,b) <$> f a
```

Lenses

Setters (are like Functors)

```
type Simple f s a = f s s a a

sets :: ((a -> b) -> s -> t) -> Setter s t a b
mapped :: Functor f => Setter (f a) (f b) a b

over, mapOf, (%~) :: Setter s t a b -> (a -> b) -> s -> t
set, (.~) :: Setter s t a b -> b -> s -> t
(+~), (-~), (*~) :: Num c => Setter s t c c -> c -> s -> t
(//~) :: Fractional c => Setter s t c c -> c -> s -> t
(||~), (&&~) :: Setter s t Bool Bool -> Bool -> s -> t

assign :: MonadState s m => Setter s s a b -> b -> m ()
(.=) :: MonadState s m => Setter s s a b -> b -> m ()
(%=) :: MonadState s m => Setter s s a b -> (a -> b) -> m ()
(+=), (-=), (*=) :: (MonadState s m, Num a) => Simple Setter s a -> a -> m ()
(//=) :: (MonadState s m, Fractional a) => Simple Setter s a -> a -> m ()
(||=), (&&~) :: MonadState s m => Simple Setter s Bool -> Bool -> m ()
```

Traversals

Lenses

Traversals

```
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t

traverse           :: Traversable f => Traversal (f a) (f b) a b
traverse.traverse :: (Traversable f, Traversable g) => Traversal (f (g a)) (f (g b)) a b

over traverse f = runIdentity . traverse (Identity . f)
               = fmapDefault f
               = fmap f
```

Lenses

Traversals

```
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t

mapM :: (Traversable f, Monad m) => (a -> m b) -> f a -> m (f b)
mapM f = unwrapMonad . traverse (WrapMonad . f)

mapMOf :: Monad m => Traversal s t a b -> (a -> m b) -> s -> m t
mapMOf l f = unwrapMonad . l (WrapMonad . f)

traverse          :: Traversable f => Traversal (f a) (f b) a b
traverse.traverse :: (Traversable f, Traversable g) => Traversal (f (g a)) (f (g b)) a b

over traverse f = runIdentity . traverse (Identity . f)
               = fmapDefault f
               = fmap f

mapMOf traverse f = unwrapMonad . traverse (WrapMonad . f)
                  = mapM f
```

Lenses

Traversals

```
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
```

Traversable Laws:

1.) traverse pure = pure
2.) Compose . fmap (traverse f) . traverse g ≡ traverse (Compose . fmap f . g)

Laws for a valid Traversal l:

1.) l pure = pure
2.) Compose . fmap (l f) . l g = l (Compose . fmap f . g)

Lenses

Traversals

```
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t

traverse :: Traversable f => Traversal (f a) (f b) a b

both :: Traversal (a,a) (b,b) a b
both f (a,b) = (,) <$> f a <*> f b

traverseLeft :: Traversal (Either a c) (Either b c) a b
traverseLeft f (Left a) = Left <$> f a
traverseLeft f (Right c) = pure (Right c)

traverseRight :: Traversal (Either c a) (Either c b) a b
traverseRight f (Left c) = pure (Left c)
traverseRight f (Right a) = Right <$> f a

traverseLeft.both :: Traversal (Either (a,a) c) (Either (b,b) c) a b

>>> over (traverseLeft.both) (+1) $ Left (2,3)
Left (3,4)

(.~) = over
traverseLeft.both .~ (+1) $ Left (2,3)
traverseLeft.both +~ 1 $ Left (2,3)
```

Lenses

Traversals (are like Traversables)

```
type LensLike f s t a b = (a -> f b) -> s -> f t
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t

traverse :: Traversable t => Traversal (t a) (t b) a b
ignored :: Traversal s s a b
traverseLeft :: Traversal (Either a c) (Either b c) a b
traverseRight :: Traversal (Either c a) (Either c b) a b
both :: Traversal (a, a) (b, b) a b

traverseOf :: LensLike f s t a b -> (a -> f b) -> s -> f t
forOf :: LensLike f s t a b -> s -> (a -> f b) -> f t
mapMOf :: LensLike (WrappedMonad m) s t a b -> (a -> m b) -> s -> m t
forMOf :: LensLike (WrappedMonad m) a b c d -> s -> (a -> m b) -> m t
```

Folds

Lenses

Folds

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Foldable f where
  foldMap :: Monoid m => (a -> m) -> f a -> m
  ...

class (Functor f, Foldable f) => Traversable f where
  traverse :: Applicative m => (a -> m b) -> f a -> m (f b)
  ...
```

Lenses

Folds

```
(.)         :: (a -> b) -> (c -> a)          -> c           -> b
(.).(.)     :: (a -> b) -> (c -> d -> a)     -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap           :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap      :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))

foldMap                 :: (Foldable f, Monoid m)
  => (a -> m) -> f a         -> m
foldMap.foldMap         :: (Foldable f, Foldable g, Monoid m)
  => (a -> m) -> f (g a)     -> m
foldMap.foldMap.foldMap :: (Foldable f, Foldable g, Foldable h, Monoid m)
  => (a -> m) -> f (g (h a)) -> m
```

Lenses

Folds

```
(.)          :: (a -> b) -> (c -> a)           -> c           -> b
(.).(.)      :: (a -> b) -> (c -> d -> a)      -> c -> d      -> b
(.).(.).(.) :: (a -> b) -> (c -> d -> e -> a) -> c -> d -> e -> b

fmap            :: Functor f                          => (a -> b) -> f a         -> f b
fmap.fmap       :: (Functor f, Functor g)             => (a -> b) -> f (g a)     -> f (g b)
fmap.fmap.fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) -> f (g (h b))

foldMap                  :: (Foldable f, Monoid m)
  => (a -> m) -> f a         -> m
foldMap.foldMap          :: (Foldable f, Foldable g, Monoid m)
  => (a -> m) -> f (g a)     -> m
foldMap.foldMap.foldMap :: (Foldable f, Foldable g, Foldable h, Monoid m)
  => (a -> m) -> f (g (h a)) -> m

traverse                    :: (Traversable f, Applicative m)
  => (a -> m b) -> f a         -> m (f b)
traverse.traverse           :: (Traversable f, Traversable g, Applicative m)
  => (a -> m b) -> f (g a)     -> m (f (g b))
traverse.traverse.traverse :: (Traversable f, Traversable g, Traversable h, Applicative m)
  => (a -> m b) -> f (g (h a)) -> m (f (g (h b)))
```

Lenses

Folds

```
fmap        :: Functor f     => (a -> b) -> f a -> f b

foldMap        :: (Foldable f, Monoid m)    => (a -> m) -> f a -> m

traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)
```

Lenses

Folds

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b

foldMap        :: (Foldable f, Monoid m)    => (a -> m) -> f a -> m
foldMapDefault :: (Traversable f, Monoid m) => (a -> m) -> f a -> m

traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)
```

Lenses

Folds

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault f = runIdentity . traverse (Identity . f)

foldMap        :: (Foldable f, Monoid m)    => (a -> m) -> f a -> m
foldMapDefault :: (Traversable f, Monoid m) => (a -> m) -> f a -> m
foldMapDefault f = getConst . traverse (Const . f)

traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)

newtype Const m a = Const { getConst :: m }

instance Functor (Const m) where
  fmap _ (Const m) = Const m

instance Monoid m => Applicative (Const m) where
  pure _ = Const mempty
  Const m <*> Const n = Const (m <> n)
```

Lenses

Folds

```
foldMap        :: (Foldable f, Monoid m)    => (a -> m) -> f a -> m
foldMapDefault :: (Traversable f, Monoid m) => (a -> m) -> f a -> m
foldMapDefault f = getConst . traverse (Const . f)

foldMapOf l f = getConst . l (Const . f)

folded :: Foldable f => Fold (f a) a
folded.folded :: (Foldable f, Foldable g) => Fold (f (g a)) a
folded.folded.folded :: (Foldable f, Foldable g, Foldable h) => Foldable (f (g (h a))) a

folded f = Const . foldMap (getConst . f)

foldMapOf folded f = getConst . Const . foldMap (getConst . Const . f)
                   = foldMap f

view l = getConst . l Const

view folded = getConst . Const . foldMap (getConst . Const)
            = foldMap id
            = fold
```

```
type Fold s a = forall m. (a -> Const m a) -> s -> Const m s
```

Lenses

Folds

```
foldMapOf l f = getConst . l (Const . f)
anyOf l f = getAny . foldMapOf l (Any . f)
sumOf l = getSum . foldMap Sum

>>> sumOf both (10,20)
30

>>> sumOf (traverse.both) [(10,20),(30,40)]
100
```

More Than a Traversal

Lenses

More Than a Traversal

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t

type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s


view l = getConst . l Const


foldMapOf l f = getConst . l (Const . f)


toListOf l = getConst . l (\c -> Const [c])


_1 f (a,b) = (,b) <$> f a

>>> view _1 (10,20)
10

_1 :: Traversal (a,c) (b,c) a b
>>> view _1 (10,20)
No instance of Monoid for a
No instance of Num for a
```

Lenses

More Than a Traversal

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t

view :: Getting a s t a b -> s -> a
view l = getConst . l Const

foldMapOf :: Getting m s t a b -> (a -> m) -> s -> m
foldMapOf l f = getConst . l (Const . f)

toListOf :: Getting [a] s t a b -> s -> [a]
toListOf l = getConst . l (\a -> Const [a])

_1 :: Lens (a,c) (b,c) a b
_1 f (a,b) = (,b) <$> f a

>>> view _1 (10,20)
10
```

Lenses

More Than a Traversal

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t

view :: Getting a s t a b -> s -> a
view l = getConst . l Const

set :: Setter s t a b -> b -> s -> t
set l d = runIdentity . l (Identity . const d)

lens :: (s -> a) -> (b -> s -> t) -> Lens s t a b
lens sa bst afb s = (`bst` s) <$> afb (sa s)
```

A Lens updates a *single* target. A Traversal can update (and read from) *many*.

A lens is just a valid traversal that targets a single element by only using the Functor out of whatever instance it is supplied.

Lenses

More Than a Traversal

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t

_1 :: Lens (a,c) (b,c) a b
_1 f (a,b) = (,b) <$> f a

_2 :: Lens (c,a) (c,b) a b
_2 f (a,b) = (a,) <$> f b

at :: Ord k => k -> Simple Lens (Map k a) a
at

traverse._1 :: Applicative m => (a -> m b) -> [(a,c)] -> m [(a,c)]

>>> over (traverse._1) length $ [("hello","San"),("Francisco",":)")]
[(5,"San"),(9,":)"]

view _1 (10,20)
10
```

Lenses

Folds (are like Foldables)

```
type Getting r s t a b = (a -> Const r b) -> s -> Const r t

(^?) :: s -> Getting (First a) s t a b -> Maybe a
(^..) :: s -> Getting [a] s t a b -> [a]
folding :: (Foldable f, Applicative g, Gettable g) => (s -> f a) -> LensLike g s t a b
folded :: Foldable f => Fold (f a) a

unfolded :: (s -> Maybe (a, s)) -> Fold s a
iterated :: (a -> a) -> Fold a a
backwards :: LensLike (Backwards f) s t a b -> LensLike f s t a b
repeated :: Fold a a
replicated :: Int -> Fold a a
takingWhile :: (Gettable f, Applicative f)
            => (a -> Bool) -> Getting (Endo (f s)) s s a a -> LensLike f s s a a

foldMapOf :: Getting r s t a b -> (a -> r) -> s -> r
foldOf :: Getting a s t a b -> s -> a
foldrOf :: Getting (Endo r) s t a b -> (a -> r -> r) -> r -> s -> r
toListOf :: Getting [a] s t a b -> s -> [a]
anyOf :: Getting Any s t a b -> (a -> Bool) -> s -> Bool
traverseOf_ :: Functor f => Getting (Traversed f) s t a b -> (a -> f r) -> s -> f ()
```

Getters

Lenses

Getters

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t
type Getter s a        = forall r.                  (a -> Const r a)  -> s -> Const r s

to :: (s -> a) -> Getter s a
-- to :: (s -> a) -> (a -> Const r a) -> s -> Const r s
```

Lenses

Getters

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t
type Getter s a        = forall r.                  (a -> Const r a)  -> s -> Const r s

to :: (s -> a) -> Getter s a
-- to :: (s -> a) -> (a -> Const r a) -> s -> Const r s

to_ :: (s -> a) -> (a -> r) -> s -> r
to_ f g = g . f
```

Lenses

Getters

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t
type Getter s a        = forall r.                  (a -> Const r a)  -> s -> Const r s

to :: (s -> a) -> Getter s a
to f g = Const . getConst . g . f

view :: Getting a s t a b -> s -> a
view l = getConst . l Const

view (to f) = getConst . to f Const
            = getConst . Const . getConst . Const . f
            = f
```

Lenses

Getters

```
type Traversal s t a b = forall f. Applicative f => (a -> f b)        -> s -> f t
type Lens s t a b      = forall f. Functor f     => (a -> f b)        -> s -> f t
type Setter s t a b    =                            (a -> Identity b) -> s -> Identity t
type Fold s a          = forall m. Monoid m      => (a -> Const m a)  -> s -> Const m s
type Getting r s t a b =                            (a -> Const r b)  -> s -> Const r t
type Getter s a        = forall r.                  (a -> Const r a)  -> s -> Const r s

to :: (s -> a) -> Getter s a
to f g = Const . getConst . g . f

view :: Getting a s t a b -> s -> a
view l = getConst . l Const

view (to f) = getConst . to f Const
            = getConst . Const . getConst . Const . f
            = f
```

Lenses

Getters (are like functions)

```
class Functor f => Gettable f where
  coerce :: f a -> f b

type Getter a c = forall f. Gettable f => (c -> f c) -> a -> f a
newtype Accessor r a = Accessor { runAccessor :: r }

type Getting r a b c d = (c -> Accessor r d) -> a -> Accessor r b

to :: (s -> a) -> Getter s a
to f g = coerce . g . f

(^.) :: s -> Getting a s t a b -> a
view, (^$) :: Getting a s t a b -> s -> a

use :: MonadState s m => Getting a s t a b -> m a
```

Lenses

Lenses (Both Getter and Traversal)

```
type Lens s t a b = forall f. Functor f => (a -> f b) -> s -> f t

lens     :: (s -> a) -> (s -> b -> t) -> Lens s t a b
resultAt :: Eq e => e -> Simple Lens (e -> a) a
chosen   :: Lens (Either a a) (Either b b) a b

(<+~), (<-~), (<*~) :: Num a => LensLike ((,) a) s t a a -> a -> s -> (a, t)
(<//~)              :: Fractional a => LensLike ((,) a) s t a a -> a -> s -> (a, t)
(<||~), (<&&~)      :: LensLike ((,) Bool) s t Bool Bool -> Bool -> s -> (Bool, t)
(%%~)               :: LensLike f s t a b -> (a -> f b) -> s -> f t

(<+=),(<-=),(<*=) :: (MonadState s m, Num a) => SimpleLensLike ((,) a) s a -> a -> m a
(<//=)            :: (MonadState s m, Fractional a) => SimpleLensLike ((,) a) s a -> a -> m a
(<||=),(<&&=)     :: MonadState s m => SimpleLensLike ((,) Bool) s Bool -> Bool -> m Bool
(%%=)             :: MonadState s m => LensLike ((,) r) s s a b -> (a -> (r, b)) -> m r
```

Isomorphisms

Lenses

Fun Overloading

```
class Category k => Isomorphic k where
  isomorphic :: (a -> b) -> (b -> a) -> k a b

instance Isomorphic (->) where
  isomorphic = const

data Isomorphism a b = Isomorphism (a -> b) (b -> a)

instance Category Isomorphism where
  id = Isomorphism id id
  Isomorphism bc cb . Isomorphism ab ba = Isomorphism (bc . ab) (ba . cb)

instance Isomorphic Isomorphism where
  isomorphic = Isomorphism

type a <-> b = forall k. Isomorphic k => k a b

from :: Isomorphism a b -> b <-> a
from (Isomorphism a b) = isomorphic b a
```

Lenses

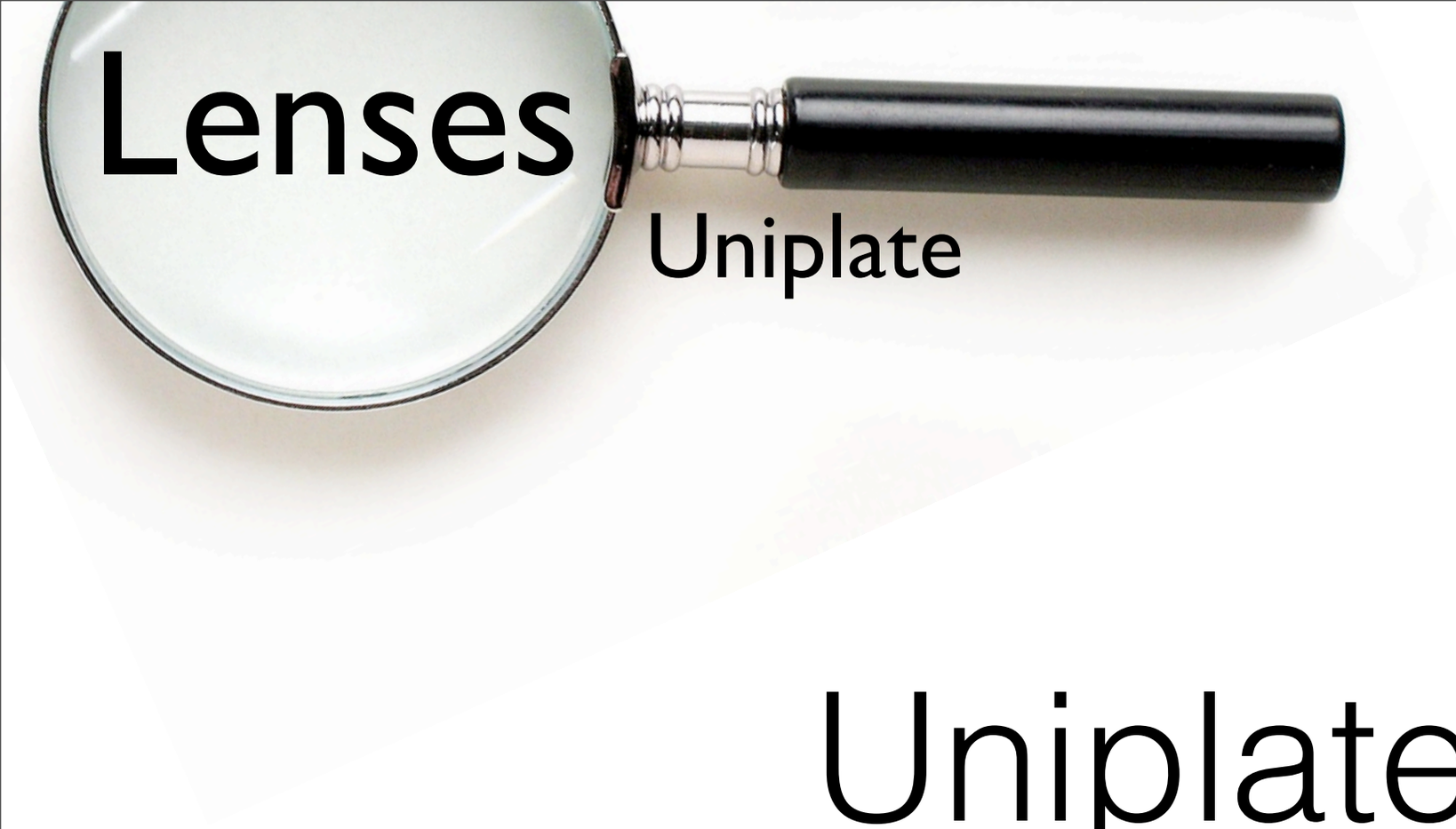
Fun Overloading

```
class Category k => Isomorphic k where
  isomorphic :: (a -> b) -> (b -> a) -> k a b

inc :: Num a => a -> a <-> a
inc = isomorphic (+1) (subtract 1)

>>> inc 4
5

>>> from inc 5
4
```

Lenses

Fun Overloading

```
type Traversal s t a b = forall f. Applicative f                => (a -> f b) -> s -> f t
type Lens s t a b      = forall f. Functor f                    => (a -> f b) -> s -> f t
type Iso s t a b       = forall k f. (Isomorphic k, Functor f) => k (a -> f b) (s -> f t)

iso :: (a -> b) -> (b -> a) -> Simple Iso a b
iso ab ba = isos ab ba ab ba

isos :: (a -> c) -> (c -> a) -> (b -> d) -> (d -> b) -> Iso a b c d
isos ac ca bd db = isomorphic (\cfd a -> db <$> cfd (ac a))
                              (\afb c -> bd <$> afb (ca c))

packed :: Simple Iso String Text
packed = iso pack unpack

text :: Simple Traversal Text Char
text = from packed . traverse

>>> anyOf (both.text) (=='c') ("chello"^.packed,"world"^.packed)
True
```



Uniplate

Lenses

Uniplate

```
class Plated a where
  plate :: Simple Traversal a a
  plate = ignored

instance Plated (Tree a) where
  plate f (Node a as) = Node a <$> traverse f as

template :: (Data a, Typeable b) => Simple Traversal a b
uniplate :: Data a => Simple Traversal a a
biplate :: (Data a, Typeable b) => Simple Traversal a b

children :: Plated a => a -> [a]
children = toListOf plate

rewriteOf :: Simple Setter a a -> (a -> Maybe a) -> a -> a
rewriteOf l f = go where go = transformOf l (\x -> maybe x go (f x))

contextsOf     :: SimpleLensLike (Bazaar a a) a a -> a -> [Context a a a]
holesOf        :: LensLike (Bazaar c c) s t c c -> s -> [Context c c t]
paraOf         :: Getting [a] a b a b -> (a -> [r] -> r) -> a -> r
partsOf        :: LensLike (Bazaar c c) s t c c -> Lens s t [c] [c]
unsafePartsOf :: LensLike (Bazaar c d) s t c d -> Lens s t [c] [d]
```

Zippers

Lenses

Zippers

```
zipper ("hello","world")
  % down _1
  % fromWithin traverse
  % focus .~ 'J'
  % rightmost
  % focus .~ 'y'
  % rezip
("Jelly","world")
```

```
zipper :: a -> Top :> a
up :: (a :> b :> c) -> a :> b
down :: Simple Lens b c -> (a :> b) -> a :> b :> c
within :: Simple Traversal b c -> (a :> b) -> Maybe (a :> b :> c)
fromWithin :: Simple Traversal b c -> (a :> b) -> a :> b :> c

left, right  :: (a :> b) -> Maybe (a :> b)
lefts, rights :: Int -> (h :> a) -> Maybe (h :> a)
leftmost, rightmost :: (a :> b) -> a :> b

save :: (a :> b) -> Tape (a :> b)
restore :: Tape (h :> a) -> Zipped h a -> Maybe (h :> a)
```

Lenses

References

- J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient Lenses. *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, British Columbia*, September, 2008.
- J. Gibbons, M. Johnson. Lenses, coalgebraically: View updates through the looking glass.
- E. Kmett. scalaz.Lens source code
- - lenses, fclabels, data-accessor - which library for structure access and mutation is better. Stack Overflow
- R. O'Connor. Functor is to Lens as Applicative is to Biplate. arXiv: 1103.2841
- T. van Laarhoven. Talk on Lenses http://twanvl.nl

Lenses

Questions

Any
Questions

?

Extra Slides

Lenses

The Power is in the Dot

```
fmap        :: Functor f     => (a -> b) -> f a -> f b
fmapDefault :: Traversable f => (a -> b) -> f a -> f b
fmapDefault f = runIdentity . traverse (Identity . f)

foldMap        :: (Foldable f, Monoid m)    => (a -> m) -> f a -> m
foldMapDefault :: (Traversable f, Monoid m) => (a -> m) -> f a -> m
foldMapDefault f = getConst . traverse (Const . f)

traverse :: (Traversable f, Applicative m) => (a -> m b) -> f a -> m (f b)

newtype Const m a = Const m

instance Functor (Const m) where
  fmap _ (Const m) = Const m

instance Monoid m => Applicative (Const m) where
  pure _ = Const mempty
  Const m <*> Const n = Const (m <> n)
```

Lenses

Isomorphism-Based Lenses

```
data a ↔ b
  = Iso (a -> b) (b -> a)

newtype Lens a b =
  forall c. Lens (a ↔ (c,b))
```

Lenses

Isomorphism-Based Lenses

```
data a ↔ b
  = Iso (a -> b) (b -> a)

newtype Lens a b =
  forall c. Lens (a ↔ (c,b))

(%%~) :: Functor f =>
         Lens a b -> (b -> f b) -> a -> f a
Lens (Iso to fro) f a = case to a of
  (c, b) -> fro . ((,) c) <$> f b
```

Lenses

Isomorphism-Based Lenses

```
data a ↔ b
  = Iso (a -> b) (b -> a)

newtype Lens a b =
  forall c. Lens (a ↔ (c, b))

(%%~) :: Functor f =>
         Lens a b -> (b -> f b) -> a -> f a
Lens (Iso to fro) f a = case to a of
  (c, b) -> fro . ((,) c) <$> f b
```

Lenses

van Laarhoven Lenses

```
type Lens a b = forall f.
  Functor f => (b -> f b) -> a -> f a

(%%~) :: Lens a b -> (b -> f b) -> a -> f a
(%%~) = id
```

Lenses

van Laarhoven Lenses

```
type Lens a b = forall f.
  Functor f => (b -> f b) -> a -> f a

(%%~) :: Lens a b -> (b -> f b) -> a -> f a
(%%~) = id

(.~) :: Lens a b -> b -> a -> a
l .~ b = runIdentity . l (Identity . const b)

(^.) :: a -> Lens a b -> b
a ^. l = getConst (l Const a)
```

Lenses

van Laarhoven Lens Families

```
type Lens a b c d = forall f.
  Functor f => (c -> f d) -> a -> f b

(%%~) :: Lens a b c d -> (c -> f d) -> a -> f b
(%%~) = id

(.~) :: Lens a b c d -> d -> a -> b
l .~ b = runIdentity . l (Identity . const b)

(^.) :: a -> Lens a b c d -> c
a ^. l = getConst (l Const a)
```

Lenses

van Laarhoven Lens Families

```
type Lens a b c d = forall f.
  Functor f => LensLike f a b c d

type LensLike f a b c d =
  (c -> f d) -> a -> f b

(%%~) :: Lens a b c d -> (c -> f d) -> a -> f b
(%%~) = id

(.~) :: Lens a b c d -> d -> a -> b
l .~ b = runIdentity . l (Identity . const b)

(^.) :: a -> Lens a b c d -> c
a ^. l = getConst (l Const a)
```

Lenses

van Laarhoven Lens Families

```
type Lens a b c d = forall f.
  Functor f => LensLike f a b c d

type LensLike f a b c d =
  (c -> f d) -> a -> f b

(%%~) :: Lens a b c d -> (c -> f d) -> a -> f b
(%%~) = id

(.~) :: Lens a b c d -> d -> a -> b
l .~ b = runIdentity . l (Identity . const b)

(^.) :: a -> Lens a b c d -> c
a ^. l = getConst (l Const a)
```

Lenses

van Laarhoven Lens Families

```
type Lens a b c d = forall f.
  Functor f => LensLike f a b c d

type LensLike f a b c d =
  (c -> f d) -> a -> f b

(%%~) :: Lens a b c d -> (c -> f d) -> a -> f b
(%%~) = id

(.~) :: LensLike Identity a b c d -> d -> a -> b
l .~ b = runIdentity . l (Identity . const b)

(^.) :: a -> LensLike (Const c) a b c d -> c
a ^. l = getConst (l Const a)
```

Lenses

Indexed Lenses, Traversals, etc.

```
class Indexed i k where
  index :: ((i -> a) -> b) -> k a b

type Indexable i a b = forall k. Indexed i k => k a b

instance Indexed i (->) where
  index f = f . const

newtype Index i a b = Index { withIndex :: (i -> a) -> b }

instance i ~ j => Indexed i (Index j) where
  index = Index

indexed :: Indexed Int k => LensLike (Indexing f) a b c d) -> k (c -> f d) (a -> f b)
```

Lenses

Indexed Lenses, Traversals, etc.

```
type Overloaded k f a b c d = k (c -> f d) (a -> f b)

-- turn a normal traversal or setter into an indexed traversal or setter.
indexed :: Indexed Int k => LensLike (Indexing f) a b c d) -> k (c -> f d) (a -> f b)
class At k m | m -> k where
  at :: k -> SimpleIndexedLens k (m v) (Maybe v)

traverseAt :: At k m => k -> SimpleIndexedTraversal k (m v) v
value :: (k -> Bool) -> SimpleIndexedTraversal k (k, v) v
iwhereOf :: (Indexed i k, Applicative f) =>
  Overloaded (Index i) f a b c c -> (i -> Bool) -> Overloaded k f a b c c

... and there are similarly a ton of combinators for these, too...
```

Lenses
Overview

Overview

What are Lenses?

The Power is in the Dot

Families of Generalized van Laarhoven Lenses

General Purpose Combinators

Indexed Traversals (and Isomorphisms)

Uniplate

Zippers

References

Lenses
Overview

Overview

What are Lenses?

The Power is in the Dot

Families of Generalized van Laarhoven Lenses

General Purpose Combinators

Indexed Traversals (and Isomorphisms)

Uniplate

Zippers

References

Lenses
Overview

Lenses

Overview

What are Lenses?

The Power is in the Dot

Families of Generalized van Laarhoven Lenses

General Purpose Combinators

Indexed Traversals (and Isomorphisms)

Uniplate

Zippers

References