Zippers and Lenses

Daniil Berezun

danya.berezun@gmail.com

2022

Outline for section 1

1. Zippers
   - Motivation
   - List Zipper
   - How to derive a zipper?
   - Tree Zipper
2. "Optics"
   - Lens: Motivation
   - `Control.Lens`
   - More on `Control.Lens`
   - Prism
   - Traversal

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```haskell
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

Immutability and Modification

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

- Let's construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```haskell
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

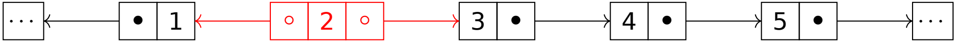
- Let's construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

- Let's construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)



- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

- Let’s construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)

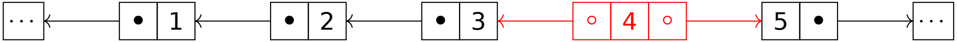
Immutability and Modification

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```haskell
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

- Let's construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)



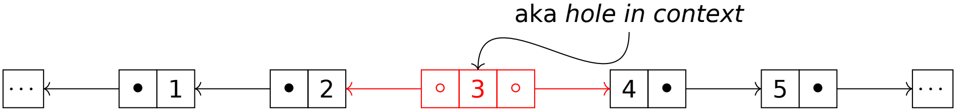
Immutability and Modification

- Functional data structures are immutable ⇒ *hard* and *expensive to modify*

```
data List a = Nil | Cons a (List a)

update :: List a -> Int -> a -> List a
update Nil _ _= Nil
update (Cons _ xs) n _ = Cons a xs
update (Cons x xs) n a = Cons x $ update a xs
```

- Let's construct another data structure s.t.:
  - represents the original data structure
  - has an ability to *navigate through* the structure focusing on some sub-structure
  - *allows* efficient *modification* of the element in focus (aka *hole*)



List Zipper: traversing

```haskell
type ListZipper a = (a, ContextLZ a)
type ContextLZ  a = ([a],[a])
-- construct list zipper
makeLZ :: [a] -> ListZipper a
makeLZ (x:xs) = (x,([],xs))
-- move focus forward
forwardLZ :: ListZipper a -> ListZipper a
forwardLZ (e, (xs, y:ys)) = (y, (e:xs, ys))
-- move focus back
backwardLZ :: ListZipper a -> ListZipper a
backwardLZ (a, (x:xs, ys))) = (x, (xs, a:ys))
-- extract list from list zipper
fromLZ :: ListZipper a -> [a]
fromLZ (x, ([], xs)) = x:xs
fromLZ z = fromLZ . backwardLZ $ z
```

```haskell
-- usage examples
ghci> lz = makeLZ [0..3]
(0,([],[1,2,3]))
ghci> forward lz
(1,([0],[2,3]))
ghci>let lz' =
  (forward . forward) lz
(2,([1,0],[3]))
ghci> backward lz'
(1,([0],[2,3]))
ghci> fromLZ lz'
[0,1,2,3]
```

List Zipper: modify

```
-- update element in hole
updateLZ :: a -> ListZipper a -> ListZipper a
updateLZ a (_, ctx) = (a, ctx)
-- insert element in hole
insertLZ :: a -> ListZipper a -> ListZipper a
insertLZ a (b, (xs, ys)) = (a, (xs, b:ys))
-- remove element in focus from list
removeLZ :: ListZipper a -> ListZipper a
removeLZ (_, (x:xs, []  )) = (x, (xs, []))
removeLZ (_, (xs  , y:ys)) = (y, (xs, ys))
-- usage examples:
ghci> fromLZz . updateLZ 22 . fowrwardLZ . fowrwardLZ . makeLZ $ [0..3]
[0,1,22,3]
ghci> fromLZ . insertLZ 11 . insertLZ 10 . forward . forward . makeLZ $ [0..3]
[0,1,11,10,2,3]
```

Remember Types Algebra?

Tuples

```
type Triple   a = (a     ,(a,a))
type PairPair a = ((a,a),(a,a))
```

How many elements of type?

$$\texttt{Triple} = A * (A * A) = A^3$$
$$\texttt{PairPair} = (A * A)^2 = A^4$$

Remember Types Algebra?

Tuples

```haskell
type Triple   a = (a     ,(a,a))
type PairPair a = ((a,a),(a,a))
```

How many elements of type?

$$\text{Triple} = A * (A * A) = A^3$$
$$\text{PairPair} = (A * A)^2 = A^4$$

Zippers

```haskell
type TripleZ a = (a, CntxTZ a)
data CntxTZ  a = CTZ1 a a | CTZ2 a a | CTZ3 a a

type PairPairZ a = (a, CntxPPZ a)
data CntxPPZ   a = CPPZ1 a a a | CPPZ2 a a a | CPPZ3 a a a | CPPZ4 a a a
```

Remember Types Algebra?

Tuples

```
type Triple   a = (a    ,(a,a))
type PairPair a = ((a,a),(a,a))
```

How many elements of type?

Triple $= A * (A * A) \quad = A^3$

PairPair $= (A * A)^2 \quad = A^4$

Zippers

```
type TripleZ a = (a, CntxTZ a)
data CntxTZ  a = CTZ1 a a | CTZ2 a a | CTZ3 a a

type PairPairZ a = (a, CntxPPZ a)
data CntxPPZ   a = CPPZ1 a a a | CPPZ2 a a a | CPPZ3 a a a | CPPZ4 a a a
```

In term of type theory algebra

TripleZ $(X) = X^3$

CntxTZ $(X) = X^2 + X^2 + X^2 \quad = 3 * X^2$

PairPairZ $(X) = X^4$

CntxPPZ $(X) = X^3 + X^3 + X^3 + X^3 \quad = 4 * X^3$

Remember Types Algebra?

Tuples

```haskell
type Triple    a = (a    ,(a,a))
type PairPair  a = ((a,a),(a,a))
```

How many elements of type?

$$\texttt{Triple} = A * (A * A) = A^3$$
$$\texttt{PairPair} = (A * A)^2 = A^4$$

Zippers

```haskell
type TripleZ a = (a, CntxTZ a)
data CntxTZ  a = CTZ1 a a | CTZ2 a a | CTZ3 a a

type PairPairZ a = (a, CntxPPZ a)
data CntxPPZ   a = CPPZ1 a a a | CPPZ2 a a a | CPPZ3 a a a | CPPZ4 a a a
```

In term of type theory algebra

$$\texttt{TripleZ}(X) = X^3$$
$$\texttt{CntxTZ}(X) = X^2 + X^2 + X^2 = 3 * X^2$$

derivative!!!

$$\texttt{PairPairZ}(X) = X^4$$
$$\texttt{CntxPPZ}(X) = X^3 + X^3 + X^3 + X^3 = 4 * X^3$$

Back to Lists

Lists

$$
\begin{aligned}
L(X) &= 1 + X + X^2 + X^3 + \dots \\
L(X) &= 1 + X * (1 + X + X^2 + X^3 + \dots) \\
L(X) &= 1 + X * L(X)
\end{aligned}
$$

Further

$$
\begin{aligned}
L(X) - X * L(X) &= 1 \\
L(X) * (1 - X) &= 1 \\
L(X) &= \frac{1}{1-X}
\end{aligned}
$$

Back to Lists

Lists

$$
\begin{aligned}
L(X) &= 1 + X + X^2 + X^3 + \dots \\
L(X) &= 1 + X * (1 + X + X^2 + X^3 + \dots) \\
L(X) &= 1 + X * L(X)
\end{aligned}
$$

Further

$$
\begin{aligned}
L(X) - X * L(X) &= 1 \\
L(X) * (1 - X) &= 1 \\
L(X) &= \frac{1}{1-X}
\end{aligned}
$$

Derivate

$$
\begin{aligned}
L(X) &= \frac{1}{1-X} \\
L'(X) &= \frac{1}{(1-X)^2} \\
L'(X) &= L(X) * L(X)
\end{aligned}
$$

Derivate: Alternative syntax

$$
\begin{aligned}
L &= 1 + X * L \\
\frac{\partial L}{\partial X} &= \frac{\partial}{\partial X}(1 + X * L) &&= L + X * \frac{\partial L}{\partial X} \\
\frac{\partial L}{\partial X} &= \frac{L}{1-X} &&= L^2
\end{aligned}
$$

Lists

$$
\begin{aligned}
L(X) &= 1 + X + X^2 + X^3 + \dots \\
L(X) &= 1 + X * (1 + X + X^2 + X^3 + \dots) \\
L(X) &= 1 + X * L(X)
\end{aligned}
$$

Further

$$
\begin{aligned}
L(X) - X * L(X) &= 1 \\
L(X) * (1 - X) &= 1 \\
L(X) &= \frac{1}{1-X}
\end{aligned}
$$

Derivate

$$
\begin{aligned}
L(X) &= \frac{1}{1-X} \\
L'(X) &= \frac{1}{(1-X)^2} \\
L'(X) &= L(X) * L(X)
\end{aligned}
$$

Derivate: Alternative syntax

$$
\begin{aligned}
L &= 1 + X * L \\
\frac{\partial L}{\partial X} &= \frac{\partial}{\partial X}(1 + X * L) &= L + X * \frac{\partial L}{\partial X} \\
\frac{\partial L}{\partial X} &= \frac{L}{1-X} &= L^2
\end{aligned}
$$

Our list zipper exactly! (actually, the context)

```haskell
type ListZ a = (a, CntxL a)
type CntxL a = ([a], [a])
-- or
type ListZipper a = ([a], [a])
-- or
type ListZipper a = ([a], a, [a])
```

Example: Tree Zipper

```
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

Example: Tree Zipper

```
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

$$T\ (X) = 1 + X * T^2(X)$$

Example: Tree Zipper

```haskell
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

$$T(X) = 1 + X * T^2(X)$$

$$T'(X) = T^2(X) + X * 2 * T(X) * T'(X)$$

Example: Tree Zipper

```
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

$$T\ (X) = 1 + X * T^2(X)$$

$$T'(X) = T^2(X) + X * 2 * T(X) * T'(X)$$

$$T'(X) = \frac{T^2(X)}{1 - 2 * X * T(X)}$$

Example: Tree Zipper

```haskell
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

$$T\,(X) = 1 + X * T^2(X)$$

$$T'(X) = T^2(X) + X * 2 * T(X) * T'(X)$$

$$T'(X) = \frac{T^2(X)}{1 - 2 * X * T(X)}$$

$$T'(X) = T^2(X) * L(2 * X * T(X))$$

```haskell
type TreeZipper  a = (a, TreeContext a)
type TreeContext a =
```

Example: Tree Zipper

```
data Tree a = Leaf | Node A (Tree A) (Tree A)
```

$$T(X) = 1 + X * T^2(X)$$

$$T'(X) = T^2(X) + X * 2 * T(X) * T'(X)$$

$$T'(X) = \frac{T^2(X)}{1 - 2 * X * T(X)}$$

$$T'(X) = T^2(X) * L(2 * X * T(X))$$

```
type TreeZipper  a = (a, TreeContext a)
type TreeContext a =
    (Tree a,  -- left  subtree of the hole
     Tree a,  -- right subtree of the hole
     [(       -- list of tuples
           Bool,  -- direction we come from: left or right
           a,     -- value of the parent node
           Tree a -- another subtree of the parent node
      )])
-- Alternative definition
type TreeZipper' =
    (
       Tree a,  -- tree in the hole
       [(       -- list of tuples
           Direction, -- left or right subtree of the parent node
           a,         -- value in the parent node
           Tree a     -- another child of the parent node
       )])
```

Outline for section 2

1. Zippers
   - Motivation
   - List Zipper
   - How to derive a zipper?
   - Tree Zipper

2. “Optics”
   - Lens: Motivation
   - `Control.Lens`
   - More on `Control.Lens`
   - Prism
   - Traversal

Lens: Motivation

- Consider some basic data type with *getter* and *setter*:

```
data Athlete = Athlete String

getName :: Athlete -> String
getName (Athlete name) = name

setName :: Athlete -> String -> Athlete
setName (Athlete _) name = Athlete name
```

- Consider some basic data type with *getter* and *setter*:

```
data Athlete = Athlete String

getName :: Athlete -> String
getName (Athlete name) = name

setName :: Athlete -> String -> Athlete
setName (Athlete _) name = Athlete name
```

- This works, but it’s *tedious*; Let’s use *record* syntax instead

```
data Athlete = Athlete { name :: String }

main :: IO ()
main = putStrLn nameOfRealAthlete where
   athleteWithoutName = Athlete ""
   realAthlete        = athleteWithoutName { name = "Athlete's name" }
   nameOfRealAthlete  = name realAthlete
```

Lens: Motivation

- Consider some basic data type with *getter* and *setter*:

```haskell
data Athlete = Athlete String

getName :: Athlete -> String
getName (Athlete name) = name

setName :: Athlete -> String -> Athlete
setName (Athlete _) name = Athlete name
```

- This works, but it's *tedious*; Let's use *record* syntax instead

```haskell
data Athlete = Athlete { name :: String }

main :: IO ()
main = putStrLn nameOfRealAthlete where
   athleteWithoutName = Athlete ""
   realAthlete        = athleteWithoutName { name = "Athlete's name" }
   nameOfRealAthlete  = name realAthlete
```

- But what happens when we introduce a new data type with the *same* field name?

```haskell
data Athlete = Athlete { name :: String }
data Club = Club { name :: String }
```

```
Error: Multiple
       declarations of 'name'
```

```haskell
data Athlete = Athlete { name :: String }
data Club = Club { name :: String }
```

```
Error: Multiple
        declarations of 'name'
```

- Even if we will use different files:

```haskell
-- Athlete.hs
data Athlete = Athlete { name :: String }

-- Club.hs
data Club = Club { name :: String }

-- Main.hs
import Athlete
import Club

blankAthlete = Athlete { name = "" }
-- Ambiguous occurrence `name'
-- It could refer to either `Athlete.name'
-- or `Club.name'
```

⊳ Ok, let's use aliases

```haskell
-- Main.hs
module Main where

import Athlete as A
import Club    as C

main :: IO ()
main = putStrLn $ nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete ""
  realAthlete        = athleteWithoutName { A.name = "A name" }
  nameOfRealAthlete  = A.name realAthlete
  clubWithoutName    = Club ""
  realClub           = clubWithoutName { C.name = "C name" }
  nameOfRealClub     = C.name realClub
```

⊳ Ok, let's use aliases

```haskell
-- Main.hs
module Main where

import Athlete as A
import Club    as C

main :: IO ()
main = putStrLn $ nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete ""
  realAthlete        = athleteWithoutName { A.name = "A name" }
  nameOfRealAthlete  = A.name realAthlete
  clubWithoutName    = Club ""
  realClub           = clubWithoutName { C.name = "C name" }
  nameOfRealClub     = C.name realClub
```

⊳ This may work, but ... module number and aliases can grow!

⋗ Fine, let's use different names for fields

```haskell
-- Club.hs
module Club where
data Club = Club { clubName :: String }
-- Athlete.hs
module Athlete where
data Athlete = Athlete { athleteName :: String }
-- Main.hs
import Athlete
import Club

main = putStrLn $ nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete ""
  realAthlete        = athleteWithoutName { athleteName = "A name" }
  nameOfRealAthlete  = athleteName realAthlete
  clubWithoutName    = Club ""
  realClub           = clubWithoutName { clubName = "C name" }
  nameOfRealClub     = clubName realClub
```

⋗ Again, works but it is not what we really want

⊳ Let’s define a type class instead:

```
class HasName a where
    getName :: a -> String
    setName :: String -> a -> a

instance HasName Athlete where
    getName athlete = athleteName athlete
    setName newName athlete = athlete { athleteName = newName }

instance HasName Club where
    getName club = clubName club
    setName newName club = club { clubName = newName }

main = putStrLn $ nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete ""
  realAthlete        = setName "A name" athleteWithoutName
  nameOfRealAthlete  = getName realAthlete
  clubWithoutName    = Club ""
  realClub           = setName "C name" clubWithoutName
  nameOfRealClub     = getName realClub
```

⊳ Let's get rid of **String**; Maybe someone wants to redefine it

```haskell
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
class HasName a b where
    getName :: a -> b
    setName :: b -> a -> a

instance HasName Athlete Text where
    getName athlete = athleteName athlete
    setName newName athlete = athlete { athleteName = newName }

instance HasName Club String where
    getName club = clubName club
    setName newName club = club { clubName = newName }

main = putStrLn $ unpack nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete empty
  realAthlete        = setName (pack "A name") athleteWithoutName
  nameOfRealAthlete  = getName realAthlete
  clubWithoutName    = Club ""
  realClub           = setName "C name" clubWithoutName
  nameOfRealClub     = getName realClub
```

⊳ Can we do better? It's functional programming: it should be *brief* and *elegant*

Finally, Lens

```haskell
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

import Data.Text
import Athlete
import Club

data Lens a b = Lens { get :: a -> b
                     , set :: b -> a -> a}
athleteNameLens :: Lens (Athlete a) a
athleteNameLens = Lens { get = \athlete -> athleteName athlete
                       , set = \newName athlete -> athlete { athleteName = newName }}
clubNameLens :: Lens Club String
clubNameLens = Lens { get = \club -> clubName club
                    , set = \newName club -> club { clubName = newName }}
class HasName a b where name :: Lens a b
instance HasName (Athlete a) a where name = athleteNameLens
instance HasName Club String where name = clubNameLens

main = putStrLn $ unpack nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete empty
  realAthlete        = set name (pack "A name") athleteWithoutName
  nameOfRealAthlete  = get name realAthlete
  clubWithoutName    = Club ""
  realClub           = set name "C name" clubWithoutName
  nameOfRealClub     = get name realClub
```

Control.Lens

```haskell
{-# LANGUAGE TemplateHaskell, MultiParamTypeClasses, FlexibleInstances #-}

import Control.Lens
import Data.Text

data Athlete a = Athlete { _athleteName :: a }
makeLenses ''Athlete

data Club = Club { _clubName :: String }
makeLenses ''Club

class HasName a b where name :: Lens' a b

instance HasName (Athlete a) a where name = athleteName

instance HasName Club String where name = clubName

main = putStrLn $ unpack nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete empty
  realAthlete        = set name (pack "A name") athleteWithoutName
  nameOfRealAthlete  = view name realAthlete
  clubWithoutName    = Club ""
  realClub           = set name "C name" clubWithoutName
  nameOfRealClub     = view name realClub
```

```haskell
{-# LANGUAGE TemplateHaskell,  MultiParamTypeClasses, FlexibleInstances,
  FunctionalDependencies #-}

import Control.Lens
import Data.Text

data Athlete a = Athlete { _athleteName :: a }
makeFields ''Athlete

data Club = Club { _clubName :: String }
makeFields ''Club

main = putStrLn $ unpack nameOfRealAthlete ++ ", " ++ nameOfRealClub where
  athleteWithoutName = Athlete empty
  realAthlete        = set name (pack "A name") athleteWithoutName
  nameOfRealAthlete  = view name realAthlete
  clubWithoutName    = Club ""
  realClub           = set name "C name" clubWithoutName
  nameOfRealClub     = view name realClub
```

More on Control.Lens

_1

```
ghci> view _1 (1,2)
1
ghci> view _3 (1,2,3)
3
```

Composition; infix notation

```
ghci> view (_1 . _2) ((1,2),3)
2
ghci> ((1,2),3) ^. _1
(1,2)
ghci> ((1,2),3) ^. _1 . _2
2
```

Modification

```
ghci> set _1 3 (1,2)
(3,2)
ghci> set _1 "Hello" (1,2)
("Hello",2)
ghci> over _1 length ("Hello","World")
(5,"World")
```

Infix notation

```
ghci> _1 .~ "Hello" $ (1,2)
("Hello",2)
ghci> (1, 2) & _1 .~ "Hello"
("Hello",2)
ghci> _1 %~ (^2) $ (2,3)
(4,3)
```

_1

```
ghci> view _1 (1,2)
1
ghci> view _3 (1,2,3)
3
```

Composition; infix notation

```
ghci> view (_1 . _2) ((1,2),3)
2
ghci> ((1,2),3) ^. _1
(1,2)
ghci> ((1,2),3) ^. _1 . _2
2
```

Modification

```
ghci> set _1 3 (1,2)
(3,2)
ghci> set _1 "Hello" (1,2)
("Hello",2)
ghci> over _1 length ("Hello","World")
(5,"World")
```

Infix notation

```
ghci> _1 .~ "Hello" $ (1,2)
("Hello",2)
ghci> (1, 2) & _1 .~ "Hello"
("Hello",2)
ghci> _1 %~ (^2) $ (2,3)
(4,3)
```

Lens laws

```
view l (set l v s)     ≡ v
set l (view l s) s     ≡ s
set l v' (set l v s)   ≡ set l v' s
```

Prism

- Prism for sum types is the same as lens for product type

Examples

```
ghci> preview _Left (Left 1)
Just 1
ghci> preview _Right (Left 1)
Nothing
ghci> review _Left "abc"
Left "abc"
```

Prism

- Prism for sum types is the same as lens for product type

Examples

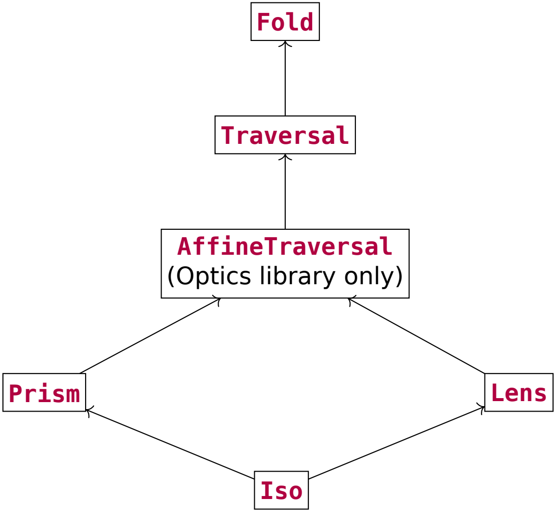
```
ghci> preview _Left (Left 1)
Just 1
ghci> preview _Right (Left 1)
Nothing
ghci> review _Left "abc"
Left "abc"
```

Composition of Lenses and Prisms

```
ghci> Left (1,2,3) ^? _Left . _2
Just 2
ghci> (Left 1,Left 1,Right "abc")
  ^? _3 . _Right
Just "abc"
ghci> (Left 1,Left 2,Right "abc")
  ^? _3 . _Left
Nothing
```

- Lenss and prisms are closed under composition
- Composition of prisms and lenses is a **Trevarsal**
- **Traverse** can have a zero, one or more focuses

Optics hierarchy



Traversal

```haskell
data Atom     = Atom     { _element :: String, _point :: Point  } deriving (Show)
data Point    = Point    { _x       :: Double, _y     :: Double } deriving (Show)
data Molecule = Molecule { _atoms   :: [Atom]                   } deriving (Show)
$(makeLenses ''Atom)
$(makeLenses ''Point)
$(makeLenses ''Molecule)
```

Traversal

```haskell
data Atom     = Atom     { _element :: String, _point :: Point  } deriving (Show)
data Point    = Point    { _x       :: Double, _y     :: Double } deriving (Show)
data Molecule = Molecule { _atoms   :: [Atom]                   } deriving (Show)
$(makeLenses ''Atom)
$(makeLenses ''Point)
$(makeLenses ''Molecule)
```

Q: What is a lens?
A: a first class getter and setter for a value We could pretend that it is a record with two fields:

```haskell
data Lens a b = Lens
    { view :: a -> b
    , over :: (b -> b) -> (a -> a)
    }
```

Q: What is a traversal?
A: first class getter and setter for an arbitrary number of values Think of a traversal as a record with two fields:

```haskell
data Traversal' a b = Traversal'
    { toListOf :: a -> [b]
    , over     :: (b -> b) -> (a -> a)
    }
```

Traversal

```haskell
data Atom     = Atom     { _element :: String, _point :: Point  } deriving (Show)
data Point    = Point    { _x       :: Double, _y     :: Double } deriving (Show)
data Molecule = Molecule { _atoms   :: [Atom]                   } deriving (Show)
$(makeLenses ''Atom)
$(makeLenses ''Point)
$(makeLenses ''Molecule)
```

Q: What is a lens?
A: a first class getter and setter for a value We could pretend that it is a record with two fields:

```haskell
data Lens a b = Lens
    { view :: a -> b
    , over :: (b -> b) -> (a -> a)
    }
```

Q: What is the type of a lens?

```haskell
point :: Lens' Atom  Point
x     :: Lens' Point Double
```

Q: What is a traversal?
A: first class getter and setter for an arbitrary number of values Think of a traversal as a record with two fields:

```haskell
data Traversal' a b = Traversal'
    { toListOf :: a -> [b]
    , over     :: (b -> b) -> (a -> a)
    }
```

Q: What is the type of a traversal?

```haskell
atoms   :: Traversal' Molecule [Atom]
```

Traversal

```haskell
data Atom     = Atom     { _element :: String, _point :: Point  } deriving (Show)
data Point    = Point    { _x       :: Double, _y     :: Double } deriving (Show)
data Molecule = Molecule { _atoms   :: [Atom]                   } deriving (Show)
$(makeLenses ''Atom)
$(makeLenses ''Point)
$(makeLenses ''Molecule)
```

Q: What is a lens?
A: a first class getter and setter for a value We could pretend that it is a record with two fields:

```haskell
data Lens a b = Lens
    { view :: a -> b
    , over :: (b -> b) -> (a -> a)
    }
```

Q: What is the type of a lens?

```haskell
point :: Lens' Atom  Point
x     :: Lens' Point Double
```

The actual definition of **Lens'** is:

```haskell
type Lens' a b =
  forall (f :: * -> *). Functor f =>
    (b -> f b) -> (a -> f a)
  = Lens s s a a
type Lens s t a b =
  forall (f :: * -> *). Functor f =>
    (a -> f b) -> s -> f t
```

Q: What is a traversal?
A: first class getter and setter for an arbitrary number of values Think of a traversal as a record with two fields:

```haskell
data Traversal' a b = Traversal'
    { toListOf :: a -> [b]
    , over     :: (b -> b) -> (a -> a)
    }
```

Q: What is the type of a traversal?

```haskell
atoms    :: Traversal' Molecule [Atom]
```

The actual definition of **Traversal'** is:

```haskell
type Traversal' a b =
  forall (f :: * -> *). Applicative f =>
    (b -> f b) -> (a -> f a)
  = Traversal s s a a
type Traversal s t a b =
  forall (f :: * -> *). Applicative f =>
    (a -> f b) -> s -> f t
```

Traversal: Example

```haskell
data Atom     = Atom     { _element :: String, _point :: Point  } deriving (Show)
data Point    = Point    { _x       :: Double, _y     :: Double } deriving (Show)
data Molecule = Molecule { _atoms   :: [Atom]                   } deriving (Show)
$(makeLenses ''Atom)
$(makeLenses ''Point)
$(makeLenses ''Molecule)

shiftAtomX :: Atom -> Atom
shiftAtomX = over (point . x) (+ 1)

shiftMoleculeX :: Molecule -> Molecule
shiftMoleculeX = over (atoms . traverse . point . x) (+ 1)

main =
  let atom1 = Atom { _element = "C", _point = Point { _x = 1.0, _y = 2.0 } }
      atom2 = Atom { _element = "O", _point = Point { _x = 3.0, _y = 4.0 } }
      molecule = Molecule { _atoms = [atom1, atom2] }
  in do
    print $ shiftAtomX     atom1
    print $ shiftMoleculeX molecule

-- Atom {_element = "C", _point = Point {_x = 2.0, _y = 2.0}}
-- Molecule {_atoms = [Atom {_element = "C", _point = Point {_x = 2.0, _y = 2.0}},
--                     Atom {_element = "O", _point = Point {_x = 4.0, _y = 4.0}}]}
```

Consuming Lenses and Traversals

```
view :: Lens' a b -> a -> b
over :: Lens' a b -> (b -> b) -> a -> a

set  :: Lens' a b ->       b  -> a -> a
set lens b = over lens (\_ -> b)
---
over :: Traversal' a b -> (b -> b) -> a -> a

set  :: Traversal' a b ->       b  -> a -> a
set traversal b = over traversal (\_ -> b)

toListOf :: Traversal' a b -> a -> [b]
```

Operators

prefix	infix
`view _1 (1,2)`	`(1,2) ^. _1`
`set _1 7 (1,2)`	`(_1 .~ 7) (1,2)`
`over _1 (2 *) (1,2)`	`(_1 \%~ (2 *)) (1,2)`
`toListOf traverse [1..4]`	`[1..4] ^.. traverse`
`preview traverse []`	`[] ^? traverse`

The End