Optics

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

August, 2023

Goals

- `Semigroup` & `Monoid`
- `Identity`
- `Traversable`
- Lenses
- Traversals

Semigroup & Monoid

Semigroup

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

Semigroup

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

Law

The operation `(<>)` should be associative:

`x <> (y <> z) == (x <> y) <> z`

Monoid

```haskell
class Semigroup m => Monoid m where
  mempty :: m
```

Monoid

```haskell
class Semigroup m => Monoid m where
  mempty :: m
```

Laws

`mempty` should be a neutral element for `(<>)`:

`x <> mempty == mempty <> x == x`

Example: Lists

```haskell
instance Semigroup [a] where
  (<>) = (++)
```

```haskell
instance Monoid [a] where
  mempty = []
```

```
GHCi> "Haskell" <> mempty <> "Kenya"
"HaskellKenya"
```

Example: `Sum`

```haskell
newtype Sum a = Sum {getSum :: a}
```

```haskell
instance Num a => Semigroup (Sum a) where
  Sum a <> Sum b = Sum (a + b)
```

```haskell
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
```

```
GHCi> getSum $ Sum 3 <> mempty <> Sum 7
10
```

Example: `Product`

```haskell
newtype Product a = Product {getProduct :: a}
```

```haskell
instance Num a => Semigroup (Product a) where
  Product a <> Product b = Product (a * b)
```

```haskell
instance Num a => Monoid (Product a) where
  mempty = Product 1
```

```
GHCi> getProduct $
  Product 3 <> mempty <> Product 7
21
```

Example: `First`

```
newtype First a = First {getFirst :: Maybe a}
```

```
instance Semigroup (First a) where
  First (Just a) <> _ = First (Just a)
  First Nothing  <> x = x
```

```
instance Monoid (First a) where
  mempty = First Nothing
```

```
GHCi> getFirst $
  mempty <> First (Just 'x') <> First (Just 'y')
Just 'x'
```

Example: `Last`

```haskell
newtype Last a = Last {getLast :: Maybe a}
```

```haskell
instance Semigroup (Last a) where
  _ <> Last (Just a) = Last (Just a)
  x <> Nothing       = x
```

```haskell
instance Monoid (Last a) where
  mempty = Last Nothing
```

```
GHCi> getLast $
  mempty <> Last (Just 'x') <> Last (Just 'y')
Just 'y'
```

Example: `Endo`

```haskell
newtype Endo a = Endo {appEndo :: a -> a}
```

```haskell
instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)
```

```haskell
instance Monoid (Endo a) where
  mempty = Endo id
```

```
GHCi> (Endo succ <> mempty <> Endo (*2))
   `appEndo` 5
11
```

Identity

The `Identity` Monad

```haskell
newtype Identity a = Identity {runIdentity a}
```

```haskell
instance Applicative Identity where
  pure = Identity
  (<*>) = ap
```

```haskell
instance Monad Identity where
  return = pure
  Identity a >>= cont = cont a
```

```
GHCi> runIdentity $ do
  x <- Identity 42
  let y = 8
  pure $ x + y
50
```

Traversable

mapM for lists

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM _ []       = pure []
mapM f (a : as) = do
  b  <- f a
  bs <- mapM f as
  pure (b : bs)
```

```
GHCi> mapM print [1, 2, 3]
1
2
3
[(), (), ()]
```

Do we need the `Monad` constraint?

```haskell
mapA :: Applicative f => (a -> f b) -> [a] -> f [b]
mapA _ []       = pure []
mapA f (a : as) = (:) <$> f a <*> mapA f as
```

```
GHCi> mapA print [1, 2, 3]
1
2
3
[(), (), ()]
```

Other uses of `mapA`

Can we use `mapA` in place of `map`? What `Applicative f` would we have to use?

Other uses of `mapA`

Can we use `mapA` in place of `map`? What `Applicative` `f` would we have to use?

Let's try `Identity`!

```
GHCi> runIdentity
  (mapA (Identity . succ) [1, 2, 3])
[2, 3, 4]
```

Other uses of `mapA`

Can we use `mapA` in place of `map`? What `Applicative` `f` would we have to use?

Let's try `Identity`!

```
GHCi> runIdentity
  (mapA (Identity . succ) [1, 2, 3])
[2, 3, 4]
```

So we can define `map` in terms of `mapA`:

```
map :: (a -> b) -> [a] -> [b]
map f = runIdentity . mapA (Identity . f)
```

Other uses of `mapA` (cntd.)

What about `foldMap`? Can we get that with `mapA` as well?

```
foldMap :: Monoid m => (a -> m) -> [a] -> m
```

Other uses of `mapA` (cntd.)

What about `foldMap`? Can we get that with `mapA` as well?

```
foldMap :: Monoid m => (a -> m) -> [a] -> m
```

We need an `Applicative` `f` that can store a value of type `m`, independent of the type it is applied to.

Other uses of `mapA` (cntd.)

What about `foldMap`? Can we get that with `mapA` as well?

```
foldMap :: Monoid m => (a -> m) -> [a] -> m
```

We need an `Applicative` `f` that can store a value of type `m`, independent of the type it is applied to.

```
data Const a b = Const {getConst :: a}
```

```
instance Functor (Const a) where
  fmap _ (Const a) = Const a
```

Making `Const` `Applicative`

But is `Const` an instance of `Applicative`?

Making `Const` `Applicative`

But is `Const` an instance of `Applicative`?

Not in general, but we only need it to be when `a` is a `Monoid`:

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

`foldMap` via `mapA`

Now we can implement `foldMap` in terms of `mapA`:

```
foldMap :: Monoid m => (a -> m) -> [a] -> m
foldMap f = getConst . mapA (Const . f)
```

```
GHCi> getSum (foldMap Sum [1, 2, 3])
6
```

`foldMap` via `mapA`

Now we can implement `foldMap` in terms of `mapA`:

```
foldMap :: Monoid m => (a -> m) -> [a] -> m
foldMap f = getConst . mapA (Const . f)
```

```
GHCi> getSum (foldMap Sum [1, 2, 3])
6
```

It seems `mapA` is very powerful, providing us with `Functor` and `Foldable` instances for `[]`, in addition to doing effectful mappings.

`mapA` for trees

Having seen the power of `mapA` for lists, what about other "container" types like trees?

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
  deriving Show
```

```
mapT :: Applicative f
   => (a -> f b) -> Tree a -> f (Tree b)
mapT f (Leaf a)  = Leaf <$> f a
mapT f (Bin l r) = Bin <$> mapT f l <*> mapT f r
```

```
GHCi> mapT print (Bin (Leaf 1) (Leaf 2))
1
2
Bin (Leaf ()) (Leaf ())
```

`mapA` for trees

Having seen the power of `mapA` for lists, what about other "container" types like trees?

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
  deriving Show
```

```
mapT :: Applicative f
   => (a -> f b) -> Tree a -> f (Tree b)
mapT f (Leaf a)  = Leaf <$> f a
mapT f (Bin l r) = Bin <$> mapT f l <*> mapT f r
```

```
GHCi> runIdentity (mapT (Identity . succ))
  (Bin (Leaf 1) (Leaf 2))
Bin (Leaf 2) (Leaf 3)
```

`mapA` for trees

Having seen the power of `mapA` for lists, what about other "container" types like trees?

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
  deriving Show
```

```
mapT :: Applicative f
   => (a -> f b) -> Tree a -> f (Tree b)
mapT f (Leaf a)  = Leaf <$> f a
mapT f (Bin l r) = Bin <$> mapT f l <*> mapT f r
```

```
GHCi> getSum (getConst (mapT (Const . Sum))
  (Bin (Leaf 1) (Leaf 2))
3
```

The `Traversable` class

From `Data.Traversable`:

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

Similarly to how we can use `liftM` and `ap` to define `Functor` and `Applicative` instances, once we have defined `return` and `(>>=)`, `Data.Traversable` provides `fmapDefault` and `foldMapDefault` to implement `Functor` and `Foldable`, once we have defined `traverse`.

`Tree` as `Traversable`

```
instance Functor Tree where
  fmap = fmapDefault
```

```
instance Foldable Tree where
  foldMap = foldMapDefault
```

```
instance Traversable Tree where
  traverse = mapT
```

Note

Intuitively, for a `Traversable` `t`, `t a` is like a "container" for `a`'s that you can inspect and manipulate.

Composing `traverse`

```
GHCi> :t traverse . traverse
(Traversable s, Traversable t, Applicative f)
  => (a -> f b) -> s (t a) -> f (s (t b))
```

Composing several `traverse`'s let's us traverse nested containers!

```
GHCi> (traverse . traverse) print
  (Bin (Leaf [1, 2]) (Leaf [3, 4]))
1
2
3
4
Bin (Leaf [(), ()]) (Leaf [(), ()])
```

Lenses

Record types

```
data Company = Company {_staff   :: [Person]}
data Person  = Person  {_name    :: String
                       , _address :: Address }
data Address = Address {_city    :: String  }
```

```
karina = Person
  {_name    = "Karina"
  , _address = Address {_city = "Zacatecas"}
  }
lars = Person
  {_name    = "Lars"
  , _address = Address {_city = "Regensburg"}
  }
iog = Company {_staff = [karina, lars]}
```

Record types

```
data Company = Company {_staff   :: [Person]}
data Person  = Person  {_name    :: String
                       , _address :: Address }
data Address = Address {_city    :: String  }
```

As a quick exercise, implement a function

```
goTo :: String -> Company -> Company
```

that takes the name of city and a `Company` and moves all company staff to that city!

Record types

```haskell
data Company = Company {_staff   :: [Person]}
data Person  = Person  {_name    :: String
                       , _address :: Address }
data Address = Address {_city    :: String  }
```

```haskell
goTo :: String -> Company -> Company
goTo there c = c {_staff = map movePerson (_staff c)}
  where
    movePerson p = p {_address = (_address p)
                                  {_city = there}}
```

Taking stock

- What have we learned in this exercise?
- While record accessors are fine for flat records, they become a pain for handling (deeply) nested records.
- If we were in a language like C, Java or Python, we would use the `.`-accessor to navigate deeply into a nested data structure and manipulate it in place.
- In Haskell, we prefer to use immutable data structures when possible.
- Does that mean we are doomed?

Taking stock

- What have we learned in this exercise?
- While record accessors are fine for flat records, they become a pain for handling (deeply) nested records.
- If we were in a language like C, Java or Python, we would use the `.`-accessor to navigate deeply into a nested data structure and manipulate it in place.
- In Haskell, we prefer to use immutable data structures when possible.
- Does that mean we are doomed?

Plan

This is Haskell, after all! We have a programmable `;`, so let's create a programmable `.`!

Teaser

By the end of this lecture, we will be able to write `goTo` like this:

```
goTo :: String -> Company -> Company
goTo s c = set (staff . each . address . city) c s
```

Getters & setters

```
_staff :: Company -> [Person]
```

Record accessors are just functions, and therefore composable, which is good. But if we want to *update* a record, we have to use record syntax, and composability breaks down.

Let's change that and make accessors "first-class citizens"!

Getters & setters

```haskell
_staff :: Company -> [Person]
```

Record accessors are just functions, and therefore composable, which is good. But if we want to *update* a record, we have to use record syntax, and composability breaks down.

Let's change that and make accessors "first-class citizens"!

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

Writing lenses

```haskell
staff :: Lens Company [Person]
staff = Lens _staff (\ c ps -> c {_staff = ps})
```

```haskell
name :: Lens Person String
name = Lens _name (\ p n -> p {_name = n})
```

```haskell
address :: Lens Person Address
address = Lens _address (\ p a -> p {_address = a})
```

```haskell
city :: Lens Address String
city = Lens _city (\ a c -> a {_city = c})
```

Writing lenses (cntd.)

This is easy, but fairly mechanical and boring.

So mechanical and boring, in fact, that it can be automated via Template Haskell or datatype-generic programming, topics we will hopefully learn about later in this course.

Trying our lenses

Let's take our shiny new lenses for a spin!

```
GHCi> get name lars
"Lars"
```

```
GHCi> set name lars "Dr. Lars"
Person {_name = "Dr. Lars",
  _address = Address {_city = "Regensburg"}}
```

So far, so good. But not much better than plain old record syntax – yet.

Other lenses

Even though we motivated lenses with records, the concept applies to many more situations:

```
_1 :: Lens (a, b) a
_1 = Lens fst (\ (_, b) a -> (a, b))
```

```
_2 :: Lens (a, b) b
_2 = Lens snd (\ (a, _) b -> (a, b))
```

```
GHCi> set _2 ('x', False) True
('x', True)
```

Other lenses (cntd.)

We can even leave the realm of product types altogether:

```haskell
data Sign = Plus | Zero | Minus
```

```haskell
sign :: Lens Int Sign
sign = Lens gt st
  where
    gt n
      | n > 0      = Plus
      | n == 0     = Zero
      | otherwise = Minus
    st n Plus  = if n == 0 then 1 else abs n
    st _ Zero  = 0
    st n Minus = if n == 0 then (- 1) else - (abs n)
```

Other lenses (cntd.)

We can even leave the realm of product types altogether:

```haskell
data Sign = Plus | Zero | Minus
```

```haskell
sign :: Lens Int Sign
```

```
GHCi> get sign (- 42)
Minus
GHCi> set sign (- 42) Plus
42
GHCi> set sign 111 Zero
0
```

Lawful lenses

There are lens laws, too; every lens should obey them, but some don't and might still be useful:

- get set: You get back what you set: `get l (set l s a) = a`
- set get: Setting what you got does not change anything: `set l s (get l s) = s`
- set set: Setting twice is the same as setting once: `set l (set l s a') a = set l s a`

The "lens" on the last slide actually violates one of the laws – can you spot which one?

An Iso example

```haskell
import qualified Data.ByteString      as S
import qualified Data.ByteString.Lazy as L
```

```haskell
lazy :: Lens S.ByteString L.ByteString
lazy = Lens L.fromStrict (\ s a -> L.toStrict a)
```

This lens obeys the laws and is quite useful. But it is even stronger than a "normal" lens, it is a so-called iso.

We will talk about isos in the next lecture.

Another useful lens

```haskell
at :: Ord k => k -> Lens (Map k v) (Maybe v)
at k = Lens gt st
  where
    gt = lookup k
    st m Nothing  = delete k m
    st m (Just v) = insert k v m
```

```
GHCi> let m = set (at "Kenya") empty
  (Just "Nairobi")
GHCi> get (at "Kenya") m
Just "Nairobi"
GHCi> get (at "USA") m
Nothing
```

Another useful lens

```haskell
at :: Ord k => k -> Lens (Map k v) (Maybe v)
at k = Lens gt st
  where
    gt = lookup k
    st m Nothing  = delete k m
    st m (Just v) = insert k v m
```

```
GHCi> set (at "USA") m
  (Just "Washington DC")
fromList [("Kenya", "Nairobi"),
  ("USA", "Washington DC")]
GHCi> set (at "Kenya") m Nothing
fromList []
```

Composing lenses

Having `city :: Lens Address String` and `address :: Lens Person Address`, we would like to compose those two to get a `Lens Person String`. Let's do that next!

```
compose :: Lens a x -> Lens s a -> Lens s x
compose ax sa = Lens
  {get = get ax . get sa
  , set = \ s x -> set sa s (set ax (get sa s) x)
  }
```

Composing lenses (cntd.)

```
GHCi> let ca = compose city address
GHCi> get ca karina
"Zacatecas"
```

```
GHCi> set ca karina "Nairobi"
Person {_name = "Karina",
  _address = Address {_city = "Nairobi"}}
```

The `Category` class

From `Control.Category`:

```
class Category (cat :: k -> k -> *) where
  id  :: forall (a :: k) . cat a a
  (.) :: forall (b :: k) (c :: k) (a :: k) .
         cat b c -> cat a b -> cat a c
```

In order to use it, you have to hide `(.)` and `id` from the `Prelude`:

```
import Prelude hiding ((.), id)
```

Turning lenses into a category

Before we can write a `Category` instance for our `Lens` type, we need an *identity lens*, i.e. one that zooms in into itself, but that is easy.

```
instance Category Lens where
  id  = Lens id (\ _ a -> a)
  (.) = compose
```

```
GHCi> get (city . address) lars
"Regensburg"
```

Taking stock again

What does `goTo` look like now?

```haskell
goTo :: String -> Company -> Company
goTo there c = set staff c
  (map movePerson (get staff c))
  where
    movePerson p = set (city . address) p there
```

The `movePerson`-part is much nicer now, but overall composability still leaves room for improvement.

Updating

Instead of just getting and setting, we would like to be able to update parts of data, too. We can of course do that by combining getting and setting:

```
over :: Lens s a -> (a -> a) -> s -> s
over sa f s = set sa s (f (get sa s))
```

```
GHCi> over name (map toUpper) lars
Person {_name = "LARS",
  _address = Address {_city = "Regensburg"}}
```

Changing the `Lens` type

Update works, but it is not very efficient for composed lenses to descend deep into a data structure, grab the value, apply a function, then descend all the way down again to put in the new value.

Seeing as setting is just a special form of updating, why don't we promote `over` to constructor status?

```
data Lens s a = Lens {get  :: s -> a
                     , over :: (a -> a) -> s -> s
                     }
```

```
set :: Lens s a -> s -> a -> s
set sa s a = over sa (const a) s
```

Changing the `Lens` type (cntd.)

We can still construct a lens from getter and setter:

```
lens :: (s -> a) -> (s -> a -> s) -> Lens s a
lens gt st = Lens gt (\ f s -> st s (f (gt s)))
```

Then we only have to slightly change the implementation of our sample lenses.

Rewriting our lenses

```haskell
staff :: Lens Company [Person]
staff = lens _staff (\ c ps -> c {_staff = ps})
```

```haskell
name :: Lens Person String
name = lens _name (\ p n -> p {_name = n})
```

```haskell
address :: Lens Person Address
address = lens _address (\ p a -> p {_address = a})
```

```haskell
city :: Lens Address String
city = lens _city (\ a c -> a {_city = c})
```

Rewriting our `Category` instance

And we have to adapt our `Category` instance:

```
compose :: Lens a x -> Lens s a -> Lens s x
compose ax sa = Lens
  { get  = get ax . get sa
  , over = over sa . over ax
  }
```

```
instance Category Lens where
  id  = Lens id ($)
  (.) = compose
```

Note

Note how nicely `over` composes!

Effectful updates

What if we want effectful updates of parts of our data structures?

```
overIO :: Lens s a -> (a -> IO a) -> s -> IO s
```

One solution would be to add another constructor to our `Lens` type:

```
data Lens s a = Lens
  {get    :: s -> a
  , over   :: (a -> a) -> s -> s
  , overIO :: (a -> IO a) -> s -> IO s
  }
```

Effectful updates

What if we want effectful updates of parts of our data structures?

But we don't have to! Instead, we can implement `overIO` just using `get` and `set`:

```haskell
overIO :: Lens s a -> (a -> IO a) -> s -> IO s
overIO sa g s = set sa s <$> g (get sa s)
```

Effectful updates

What if we want effectful updates of parts of our data structures?

We have used no special properties of `IO`, not even that it is a monad – we only used `fmap`. So we can generalize:

```
overF :: Functor f
  => Lens s a -> (a -> f a) -> s -> f s
overF sa g s = set sa s <$> g (get sa s)
```

This looks a lot like the signature of `traverse`!

Effectful update example

Let's try this!

```
askName :: String -> IO String
askName n = do
  putStrLn ("old name was: " ++ n)
  getLine
```

```
GHCi> overF name askName lars
old name was: Lars
LARS
Person {_name = "LARS",
  _address = Address {_city = "Regensburg"}}
```

`overF` instead of `over`

We can replace `over` with `overF` in our definition of lens …

```
data Lens s a = Lens
  { get   :: s -> a
  , overF :: forall f . Functor f
      => (a -> f a) -> s -> f s
  }
```

… and recover `over` using `Identity` for `f`:

```
over :: Lens s a -> (a -> a) -> s -> s
over sa f s =
  runIdentity (overF sa (Identity . f) s)
```

`overF` is enough

We can drop `get` from the definition …

```
data Lens s a = Lens
  {overF :: forall f . Functor f
     => (a -> f a) -> s -> f s}
```

… and still define `get` using `Const a` for `f`:

```
get :: Lens s a -> s -> a
get sa s = getConst (overF sa Const s)
```

van Laarhoven lenses

At this point, we see that we do not even *need* a `data` declaration any longer. We can just define a type synonym:

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

These lenses are called van Laarhoven lenses.

Changing from a data type to a type synonym is a double edged sword. There are clear advantages to keeping a data type abstract.
In this case though, we will see that we gain two important advantages:

- Easy composability and
- A form of "subtyping" (which we'll understand better once we'll have learned about traversals, prisms and isos later today and in the next lecture).

van Laarhoven lenses

At this point, we see that we do not even *need* a `data` declaration any longer. We can just define a type synonym:

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

These lenses are called van Laarhoven lenses.

We can recover `over` and `get`:

```
over :: Lens s a -> (a -> a) -> s -> s
over sa f s =
  runIdentity (sa (Identity . f) s)
```

```
get :: Lens s a -> s -> a
get sa s = getConst (sa Const s)
```

van Laarhoven lenses

At this point, we see that we do not even *need* a `data` declaration any longer. We can just define a type synonym:

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

These lenses are called van Laarhoven lenses.

And we can still construct a lens from a getter and a setter:

```
lens :: (s -> a) -> (s -> a -> s) -> Lens s a
lens gt st f s = st s <$> f (gt s)
```

This means that the definition of all our example lenses remains literally the same!

Composing van Laarhoven lenses

One of the nicest features of our previous attempts was composability. Now we do not even have a `data` type definition anymore, so we can't define a `Category` instance for our new lenses ...

Composing van Laarhoven lenses

One of the nicest features of our previous attempts was composability. Now we do not even have a `data` type definition anymore, so we can't define a `Category` instance for our new lenses ...

...But we don't have to! Van Laarhoven lenses are just functions, and we know how to compose functions!

```
GHCi> get (address . city) lars
"Regensburg"
```

Note

Note that the order of composition has swapped! Now it looks like object accessors in languages like Java.

Where are we?

By using van Laarhoven lenses, we have drastically simplified our `Lens` type.

Composition of lenses is just function composition now.

We also have “built in” effectful updates, in addition to the more basic features of getting, setting and updating.

However, `goTo` still looks the same (except for the swapped order of composition).

But now we are in a position to fix that!

Traversals

Motivation

Lenses allow us to ”zoom in” on one part of a structure.

They are naturally composable, because they are just functions.

Given a structure with many parts (of the same type), we would like to zoom in on those ”simultaneously”.

We also want to compose such Traversals with lenses and with eachother, so they should have a similar shape.

Method `traverse` of class `Traversable` has a very promising signature. This leads us to our definition of `Traversal`.

Traversal

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

A `Traversable` functor `t` gives us a `Traversal` via `traverse`:

```haskell
each :: Traversable t => Traversal (t a) a
each = traverse
```

`over` and `set` for `Traversal`'s

We defined `over` for lenses, but looking back at the definition, we don't actually *need* the full power of a lens. We only need the special case `f = Identity`. So let's change the signature of `over`, the implementation can stay exactly as it was:

```
over :: ((a -> Identity a) -> s -> Identity s)
     -> (a -> a) -> s -> s
over sa f s =
  runIdentity (sa (Identity . f) s)
```

`over` and `set` for `Traversal`'s

We defined `over` for lenses, but looking back at the definition, we don't actually *need* the full power of a lens. We only need the special case `f = Identity`. So let's change the signature of `over`, the implementation can stay exactly as it was:

And we do the same for `set`:

```
set :: ((a -> Identity a) -> s -> Identity s)
    -> s -> a -> s
set sa s a = over sa (const a) s
```

`over` and `set` for `Traversal`'s

We defined `over` for lenses, but looking back at the definition, we don't actually *need* the full power of a lens. We only need the special case `f = Identity`. So let's change the signature of `over`, the implementation can stay exactly as it was:

Let's try it!

```
GHCi> set each [1, 2, 3] 0
[0, 0, 0]
```

both

As another example for a `Traversal`, we can traverse over both components of a pair if both have the same type:

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

```
GHCi> set both (1, 2) 0
(0, 0)
```

`view` for `Traversal`'s

We can do the same we did for `over` and `set` for `get` – but for historical reasons, we call the resulting function `view`.

```
view :: ((a -> Const a a) -> s -> Const a s)
     -> s -> a
view sa s = getConst (sa Const s)
```

`view` for `Traversal`'s

We can do the same we did for `over` and `set` for `get` – but for historical reasons, we call the resulting function `view`.

For lenses, this works as expected:

```
GHCi> view name lars
"Lars"
```

`view` for `Traversal`'s

We can do the same we did for `over` and `set` for `get` – but for historical reasons, we call the resulting function `view`.

For traversals, however, we seem to be out of luck...

```
GHCi> view both (True, False)
 < interactive >: 6 : 6 : error :
   No instance for (Monoid Bool) arising from a use of  both
   In the first argument of  view, namely  both
   In the expression : view both (True, False)
   In an equation for  it  : it = view both (True, False)
```

...and we remember that `Const a` is only `Applicative` if `a` is a `Monoid`.

`view` for `Traversal`'s

We can do the same we did for `over` and `set` for `get` – but for historical reasons, we call the resulting function `view`.

```
GHCi> view both ([True], [False])
[True, False]
```

To make viewing `Traversal`s easier, we define:

```
toListOf :: ((a -> Const [a] a) -> s -> Const [a] s)
         -> s -> [a]
toListOf sa s = getConst (sa (Const . pure) s)
```

```
GHCi> toListOf both (True, False)
[True, False]
```

Composing `Lens`es & `Traversal`s

Because `Lens`es and `Traversal`s are just functions of a compatible shape, we can compose them freely:

Composing two `Lens`es gives a `Lens`.

```
GHCi> set (address . city) lars "Nairobi"
Person {_name = "Lars",
  _address = Address {_city = "Nairobi"}}
```

Composing `Lens`es & `Traversal`s

Because `Lens`es and `Traversal`s are just functions of a compatible shape, we can compose them freely:

Composing two `Traversal`s gives a `Traversal`:

```
GHCi> set (each . each) [[1], [2, 3]] 0
[[0], [0, 0]]
```

Composing `Lens`es & `Traversal`s

Because `Lens`es and `Traversal`s are just functions of a compatible shape, we can compose them freely:

Composing a `Traversal` with a `Lens` gives a `Traversal`...

```
GHCi> set (each . _1) [(1, 'x'), (2, 'y')] 0
[(0, 'x'), (0, 'y')]
```

Composing `Lens`es & `Traversal`s

Because `Lens`es and `Traversal`s are just functions of a compatible shape, we can compose them freely:

…and so does composing a `Lens` with a `Traversal`:

```
GHCi> set (_2 . each) (True, "Nairobi") 'x'
(True, "xxxxxxx")
```

`goTo` again

Now we are finally in a position to write `goTo` in a very nice and compact way:

```
goTo :: String -> Company -> Company
goTo s c = set (staff . each . address . city) c s
```

```
GHCi> goTo "Nairobi" iog
Company
  {_staff =
    [Person {_name    = "Karina"
            , _address = Address {_city = "Nairobi"}
            }
    , Person {_name    = "Lars"
            , _address = Address {_city = "Nairobi"}
            }
    ]
  }
```