Discovering lenses

Anton Dubovik

EPAM Systems

#kievfprog2017.1

Update problem

```
data Person = Person
  { schedule :: Schedule
  , street   :: String
  , father   :: Person
  }
```

```
street person
```

```
person { street = "Pearl street" }
```

```
person { street = street person ++ " avenue"}
```

```
person { father = (father person)
  { street = street (father person) ++ " avenue" } }
```

Update problem

```
person { father = (father person)
  { father = (father (father person))
    { street = street (father (father person)) ++ " avenue"
} } }
```

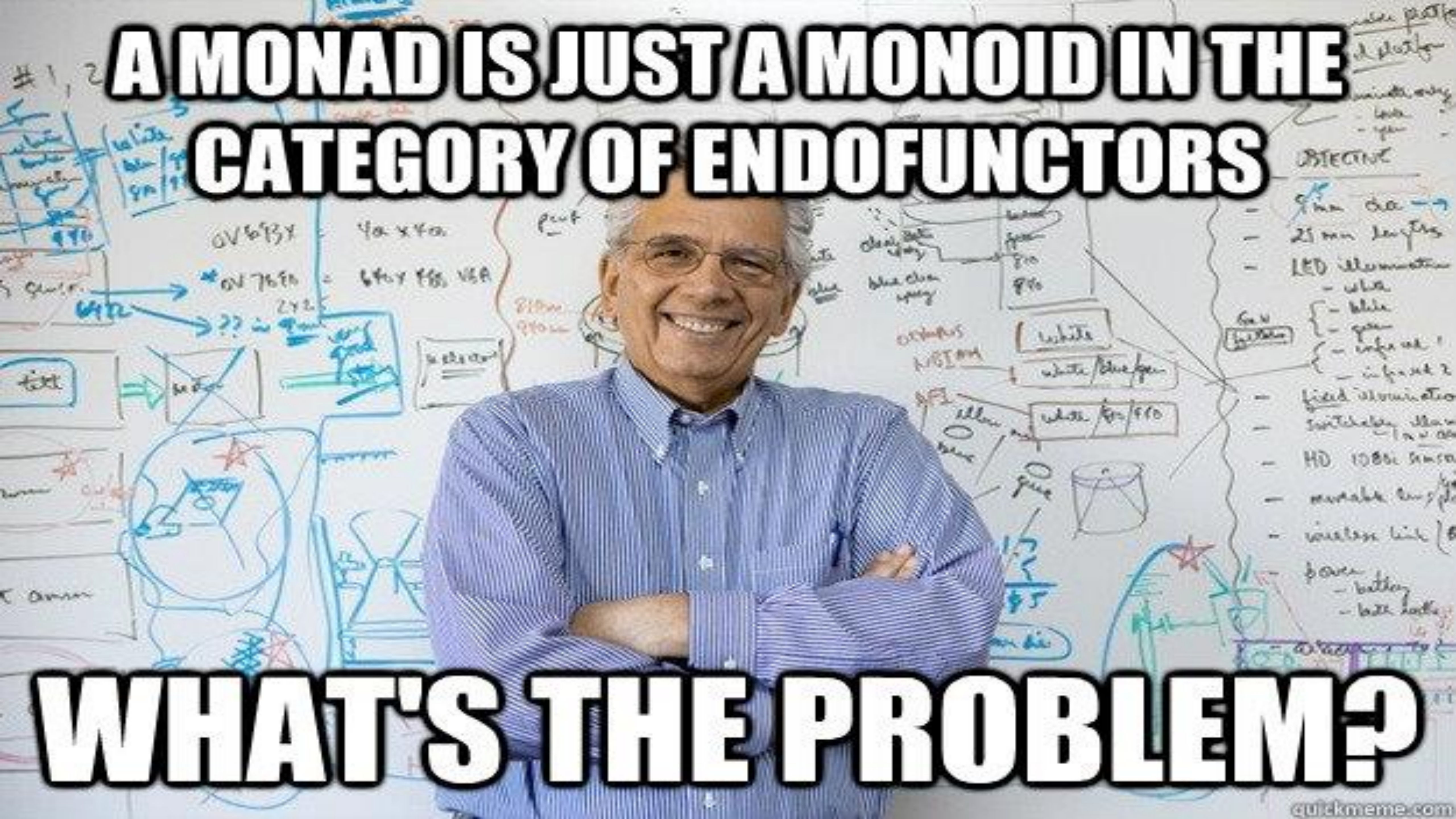
Update problem

C++

```
person.father.father.street += " avenue";
```

Haskell

```
person' = person &
   ((fatherL . fatherL . streetL)
     `modify` (++ " avenue"))
```


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHAT'S THE PROBLEM?
quickmeme.com

PLT Borat

@PLT_Borat

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

Twan van Laarhoven’s lenses

Edward Kmett’s lens package

```haskell
data Person = Person
  { schedule :: Schedule
  , street   :: String
  , father   :: Person
  }

person { street = street person ++ " avenue"}


modifyStreet :: (String -> String)
             -> (Person -> Person)
modifyStreet f pers = pers {street = f (street pers)}

modifyStreet (++ " avenue") person
```

```
person { father = (father person)
  { street = street (father person) ++ " avenue" } }

modifyFather :: (Person -> Person)
             -> (Person -> Person)
modifyFather f pers = pers {father = f (father pers)}


(modifyFather . modifyStreet) (++ " avenue") person


(modifyFather . modifyFather . modifyStreet)
  (++ " avenue")
  person
```

```
(modifyFather . modifyFather . modifyStreet)
  (++ " avenue")
  person
```

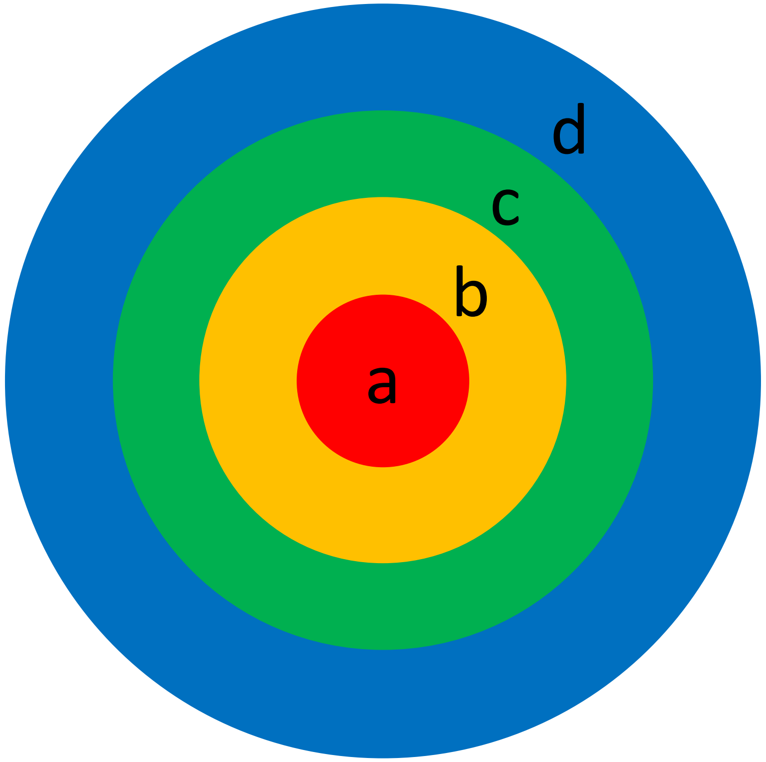
```
(&) :: a -> (a -> b) -> b
(&) a f = f a
```

```
person &
  (modifyFather . modifyFather . modifyStreet)
  (++ " avenue")
```

Composable updates

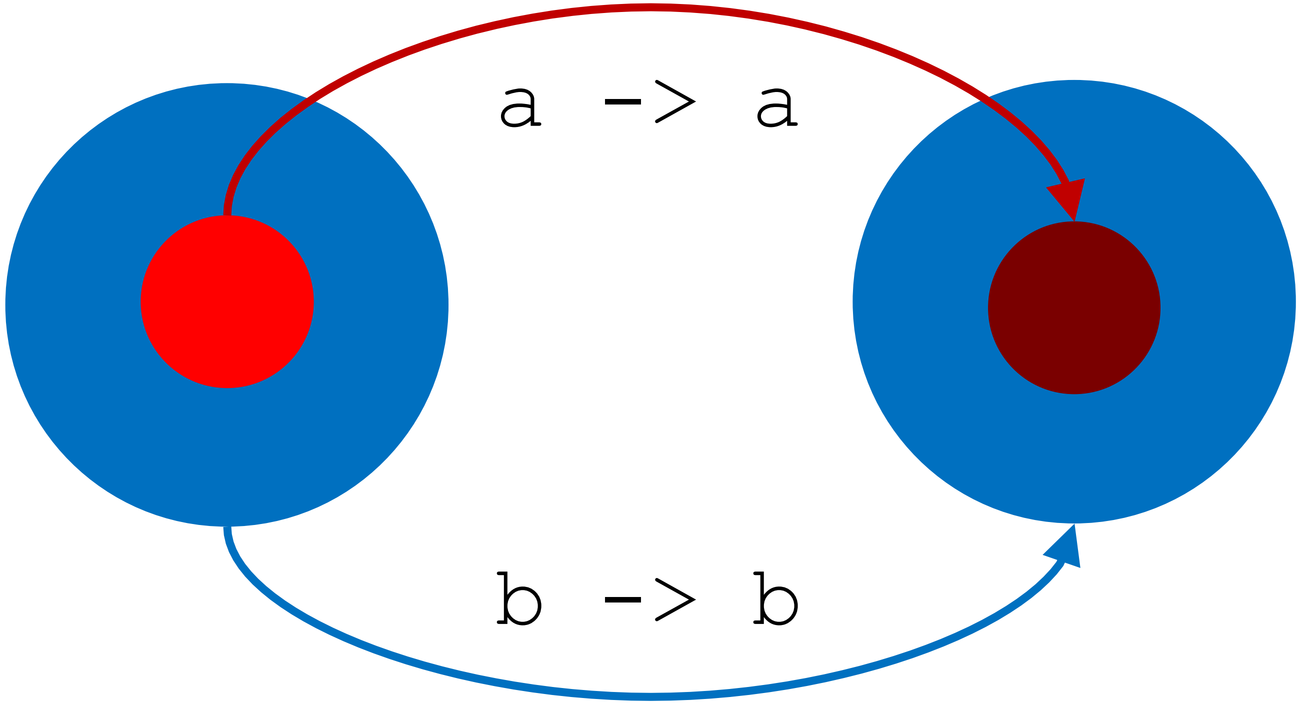
```
person &
   (modifyFather . modifyFather . modifyStreet)
   (++ " avenue")
```

```
mod1 :: (c -> c) -> (d -> d)
mod2 :: (b -> b) -> (c -> c)
mod3 :: (a -> a) -> (b -> b)
mod1 . mod2 . mod3 ::
           (a -> a) -> (d -> d)
```



Composable updates

```
type Updater b a = (a -> a) -> (b -> b)
```



Composable updates: clients

```
modify :: Updater b a -> (a -> a) -> (b -> b)
modify updater f b = updater f b
```

```
set :: Updater b a -> a -> (b -> b)
set updater a b = modify updater (\_ -> a) b
```

Getters

```
(a -> a) -> (b ->      b )
(a -> a) -> (b -> (a, b))
```

```
getAndUpdStreet
  :: (String -> String)
  -> (Person -> (String, Person))
getAndUpdStreet f pers =
  (street pers, pers {street = f (street pers)})
```

Getters

```
(a ->       a ) -> (b ->       b )
(a -> (a, a)) -> (b -> (a, b))

getAndUpdStreet
  :: (String -> (String, String))
  -> (Person -> (String, Person))
getAndUpdStreet f pers =
  let (streetOld, streetNew) = f (street pers)
  in  (streetOld, pers { street = streetNew })
```

Getters

```
(a ->       a ) -> (b ->       b )
(a -> (a, a)) -> (b -> (a, b))

getAndUpdStreet
  :: (String -> (String, String))
  -> (Person -> (String, Person))
getAndUpdStreet f pers =
  let (streetOld, streetNew) = f (street pers)
  in  (streetOld, pers { street = streetNew })
```

Getters

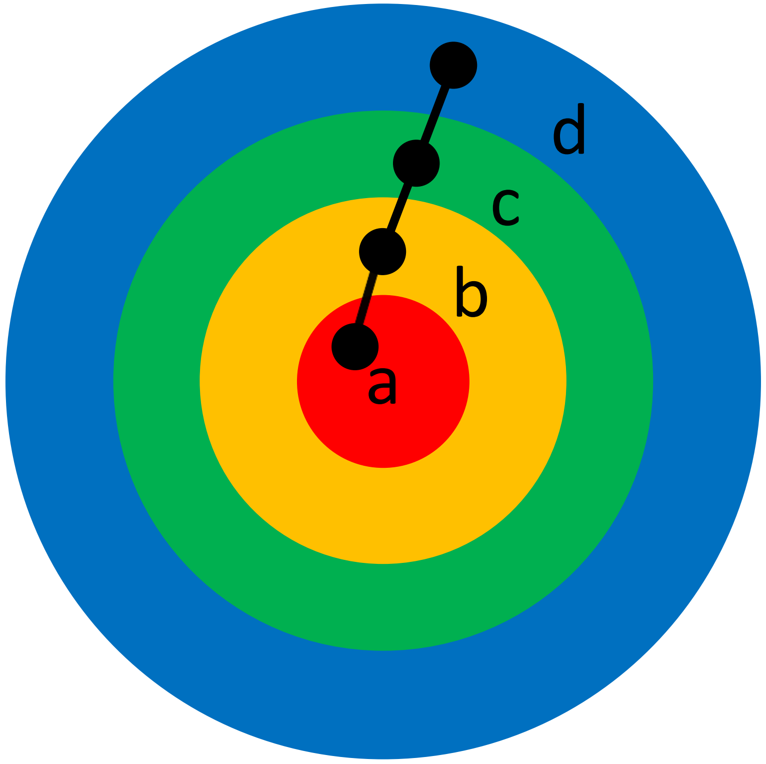
```
(a ->       a ) -> (b ->       b )
(a -> (c, a)) -> (b -> (c, b))
```

```
getAndUpdStreet
  :: (String -> (c, String))
  -> (Person -> (c, Person))
getAndUpdStreet f pers =
  let (c, streetNew) = f (street pers)
  in  (c, pers { street = streetNew })
```

Getter and updater: clients

```
type UpdaterWithPayload b a =
  forall c. (a -> (c, a)) -> (b -> (c, b))

get :: UpdaterWithPayload b a -> b -> a
get updater b = fst $ updater (\a -> (a, a)) b

modify :: UpdaterWithPayload b a -> (a -> a) -> (b -> b)
modify updater f b = snd $ updater (\a -> ((), f a)) b

set :: UpdaterWithPayload b a -> a -> b -> b
set updater a b = modify updater (\_ -> a) b
```

```
person &
  ((getAndUpdFather . getAndUpdFather . getAndUpdStreet)
  `modify` (++ " avenue"))

person &
  get (getAndUpdFather . getAndUpdFather . getAndUpdStreet)

mod1 :: (c -> (p,c)) -> (d -> (p,d))
mod2 :: (b -> (p,b)) -> (c -> (p,c))
mod3 :: (a -> (p,a)) -> (b -> (p,b))
mod1 . mod2 . mod3 ::
         (a -> (p,a)) -> (d -> (p,d))
```



Updating functions

```
type Day = Int
type Event = String
type Schedule = Day -> Maybe Event

modifyEvent
  :: (Maybe Event -> Maybe Event)
  -> (Schedule -> Schedule)
modifyEvent f sch = \day -> f (sch day)

getAndUpdSchedule :: UpdaterWithPayload Person Schedule
getAndUpdSchedule f pers =
  let (c, schedNew) = f (schedule pers)
  in  (c, pers { schedule = schedNew })
```

Updating functions

```
(Person -> (a, Person)) -> (Person -> (a, Person))
      (Schedule -> (a, Schedule)) -> (Person -> (a, Person))
            (Maybe Event -> Maybe Event) -> (Schedule -> Schedule)

person &
  ((getAndUpdFather . getAndUpdSchedule . modifyEvent)
  `modify` eraseWedding)
```

```
eraseWedding :: Maybe Event -> Maybe Event
eraseWedding (Just "Wedding") = Nothing
eraseWedding x = x
```

Updating functions

```
(Person -> (Person) -> (Person -> Person)
      (Schedule -> Schedule) -> (Person -> Person)
            (Maybe Event -> Maybe Event) -> (Schedule -> Schedule)

person &
  ((to getAndUpdFather . to getAndUpdSchedule . modifyEvent)
  `modify` eraseWedding)


to :: ((a -> (c, a)) -> (b -> (c, b)))
   -> ((a -> a) -> (b -> b))
```

Composing uncomposable

```
type Updater b a =
  (a ->       a ) -> (b ->       b )
type UpdaterWithPayload b a = forall c.
  (a -> (c, a)) -> (b -> (c, b))
```

```
(a -> f a) -> (b -> f b)
```

```
type Updater' b a =
  (a -> Identity a) -> (b -> Identity b)
```

```
f -> Identity: Updater'
f -> (,) c: UpdaterWithPayload
```

Composing uncomposable

```
type Setter b a = forall f.
  Settable f => (a -> f a) -> (b -> f b)
type Getter b a = forall f.
  Gettable f => (a -> f a) -> (b -> f b)

class Gettable f
class Gettable f => Settable f

instance Gettable ((,) c)
instance Settable Identity
instance Gettable Identity
```

Composing uncomposable

```
class Gettable f => Settable f

getterToSetter :: Getter b a -> Setter b a
getterToSetter = id
```

```
Setter Person Person
Getter Person Person

Setter Person Schedule
Getter Person Schedule

Setter Schedule (Maybe Event)
```

```
person &
  ((getAndUpdFather . getAndUpdSchedule . modifyEvent)
  `modify` eraseWedding)
```

Setter, Getter: clients

```
get :: Getter b a -> b -> a
get getter b =
  let (a, _b') = getter (\a -> (a, a)) b
  in  a

-- modify :: Getter b a -> (a -> a) -> (b -> b)
modify :: Setter b a -> (a -> a) -> (b -> b)
modify setter f b =
  let Identity b' = setter (Identity . f) b
  in  b'

-- set :: Getter b a -> a -> (b -> b)
set :: Setter b a -> a -> (b -> b)
set setter a = modify setter (\_ -> a)
```

Gettable

```
getAndUpdSchedule
  :: (Schedule -> (c, Schedule))
  -> (Person -> (c, Person))
getAndUpdSchedule f pers =
  let (c, scheduleNew) = f (schedule pers)
  in  (c, pers { schedule = scheduleNew })


getAndUpdSchedule
  :: (Schedule -> (c, Schedule))
  -> (Person -> (c, Person))
getAndUpdSchedule f pers =
  (\scheduleNew -> pers { schedule = scheduleNew }) `on`
     f (schedule pers)
  where
    on :: (Schedule -> Person) -> ((c, Schedule) -> (c, Person))
    on f (x, shed) = (x, f shed)
```

Gettable

```
getAndUpdSchedule
  :: Gettable f
  => (Schedule -> f Schedule)
  -> (Person -> f Person)
getAndUpdSchedule f pers =
  (\scheduleNew -> pers { schedule = scheduleNew }) `on`
    f (schedule pers)

class Gettable (f :: * -> *) where
  on :: (a -> b) -> (f a -> f b)


instance Gettable ((,) c) where
  on f (c, a) = (c, f a)

instance Gettable Identity where
  on f = Identity . f . runIdentity
```

Settable

```
modifyEvent
  :: (Maybe Event -> Identity (Maybe Event))
  -> (Schedule -> Identity Schedule)
modifyEvent f sch =
  Identity $ \day -> runIdentity (f (sch day))

modifyEvent
  :: (Maybe Event -> Identity (Maybe Event))
  -> (Schedule -> Identity Schedule)
modifyEvent f sch = dist $ \day -> f (sch day)
  where
    dist :: (a -> Identity b) -> Identity (a -> b)
    dist h = Identity $ \x -> runIdentity (h x)
```

Settable

```haskell
modifyEvent
  :: (Maybe Event -> Identity (Maybe Event))
  -> (Schedule -> Identity Schedule)
modifyEvent f sch = dist $ \day -> f (sch day)
  where
    dist :: (a -> Identity b) -> Identity (a -> b)
    dist h = Identity $ \x -> runIdentity (h x)
```

```haskell
modifyEvent
  :: (Maybe Event -> Identity (Maybe Event))
  -> (Schedule -> Identity Schedule)
modifyEvent f sch = dist $ \day -> f (sch day)
  where
    dist :: Functor g => g (Identity b) -> Identity (g b)
    dist h = Identity $ fmap runIdentity h
```

Settable

```
modifyEvent
  :: Settable f
  => (Maybe Event -> f (Maybe Event))
  -> (Schedule -> f Schedule)
modifyEvent f sch = dist $ \day -> f (sch day)

class Gettable f => Settable f where
  dist :: Functor g => g (f a) -> f (g a)

instance Settable Identity where
  dist h = Identity $ fmap runIdentity h
```

Composing uncomposable

Setter Person Person
Getter Person Person

Setter Person Schedule
Getter Person Schedule

Setter Schedule (Maybe Event)

```
person &
  ((getAndUpdFather . getAndUpdSchedule . modifyEvent)
  `modify` eraseWedding)
```

Multi selection

```
data Human
  = Orphan
      { name :: String }
  | Parented
      { name    :: String
      , parent1 :: Human
      , parent2 :: Human
      }
```

Multi selection

```
(a ->      a ) -> (b ->      b )
(a -> (c, a)) -> (b -> (c, b))

(a -> ([c], a)) -> (b -> ([c], b))
```

```
type UpdateWithMultiPayload c b a =
  (a -> ([c], a)) -> (b -> ([c], b))

modifyParents :: UpdateWithMultiPayload c Human Human
modifyParents _ (Orphan s) = ([], Orphan s)
modifyParents f (Parented s x y) =
  let (c1, x') = f x
      (c2, y') = f y
  in (c1 ++ c2, Parented s x' y')
```

Multi

```
type Setter b a =
  forall f. Settable f => (a -> f a) -> (b -> f b)
type Getter b a =
  forall f. Gettable f => (a -> f a) -> (b -> f b)
type Multi b a =
  forall f. Multiple f => (a -> f a) -> (b -> f b)

class Gettable f => Multiple f
class (Gettable f, Multiple f) => f Settable
instance Multiple ((,) [c])
```

Multi

```
modifyParents :: UpdateWithMultiPayload c Human Human
modifyParents _ (Orphan s) =
  let unit :: Human -> ([c], Human)
      unit a = ([], a)
  in  unit (Orphan s)
modifyParents2 f (Parented s x y) =
  let x' = f x
      y' = f y
      tuple :: ([c], Human) -> ([c], Human)
            -> ([c], (Human, Human))
      tuple (c1, a) (c2, b) = (c1++c2, (a, b))
  in (\(a,b) -> Parented s a b) `on` (tuple x' y')
```

Multi

```
modifyParents :: Multi Human Human
modifyParents _ (Orphan s) = unit $ Orphan s
modifyParents f (Parented s x y) =
  (\(a,b) -> Parented s a b) `on` (f x `tuple` f y)

class Gettable f => Multiple f where
  unit  :: a -> f a
  tuple :: f a -> f b -> f (a, b)

instance Multiple ((,) [c]) where
  unit a = ([], a)
  tuple (c1, a) (c2, b) = (c1++c2, (a, b))
```

Multi

```haskell
modifyName :: Getter Human String
modifyName f (Orphan s) =
  (\s' -> Orphan s') `on` f s
modifyName f (Parented s x y) =
  (\s' -> Parented s' x y) `on` f s
```

<u>Setter</u> Human Human
Multi Human Human

<u>Setter</u> Person Schedule
<u>Multi</u> Person Schedule
Getter Person Schedule

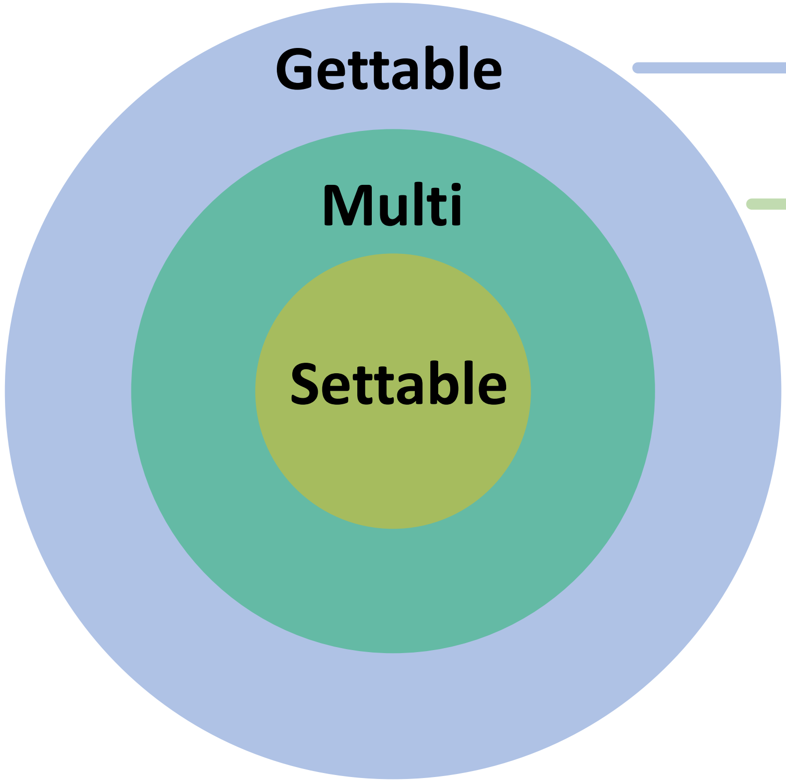
```haskell
human &
  ((modifyParents . modifyName)
   `modify` (++ " the parent"))
```

Multi: clients

```
toList :: Multi b a -> b -> [a]
toList multi b =
  let (as, _b') = multi (\a -> ([a], a)) b
  in as
```

Summary

```
type Setter b a = forall f. Settable f => (a -> f a) -> (b -> f b)
type Getter b a = forall f. Gettable f => (a -> f a) -> (b -> f b)
type Multi  b a = forall f. Multiple f => (a -> f a) -> (b -> f b)
```




```
((,) c)      get :: b -> a
((,) [c])    toList :: b -> [a]
Identity     modify :: (a -> a)
                    -> (b -> b)
             set :: a -> b -> b
```

Polymorphic updates

```
data Positioned p e = Positioned
   { position :: p
   , element  :: e
   }

changePosition
   :: (p -> p')
   -> (Positioned p e -> Positioned p' e)
```

Polymorphic updates

```
type Multi'  s t a b = forall f. Multiple f => (a -> f b) -> (s -> f t)
type Getter' s t a b = forall f. Gettable f => (a -> f b) -> (s -> f t)
type Setter' s t a b = forall f. Settable f => (a -> f b) -> (s -> f t)

modifyPosition :: Getter' (Positioned p e) (Positioned p' e) p p'
modifyPosition f (Positioned p e) = (\p' -> Positioned p' e) `on` f p

modify' :: Setter' s t a b -> (a -> b) -> (s -> t)
modify' setter f b =
  let Identity b' = setter (Identity . f) b
  in  b'

sqrtPosition :: Positioned Int Apple -> Positioned Double Apple
sqrtPosition = modify' modifyPosition (sqrt . fromIntegral)
```

Lens package

```
class (Gettable f, Multiple f)
  => Settable f
class Gettable f => Multiple f
class Gettable f
```

```
(class Functor f => Distributive f,
   Applicative f)
class Functor f => Applicative f
class Functor f
```

```
type Setter' s t a b = forall f. Settable f => (a -> f b) -> (s -> f t)
type Getter' s t a b = forall f. Gettable f => (a -> f b) -> (s -> f t)
type Multi'  s t a b = forall f. Multiple f => (a -> f b) -> (s -> f t)
```

```
type Setter s t a b =
  forall f. (Distributive f, Applicative f, Traversable f) =>
    (a -> f b) -> (s -> f t)
type Lens s t a b = forall f. Functor f => (a -> f b) -> (s -> f t)
type Traversal s t a b =
  forall f. Applicative f => (a -> f b) -> (s -> f t)
```

Questions?

Lenses prerequisites

* first-class functions
* higher-order types
* parametric polymorphism
* ah-hoc polymorphism (type classes)
* higher-rank polymorphism