lens

from the ground up

@markhibberd

motivations

the ground up

intuitions

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

intuitions

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

```
set-get ==>
    get l (set l b a) == b

get-set ==>
    set l (get l a) a == a

set-set ==>
    set l c (set l b a) == set l c a
```

Pierce's laws

but...

```
modify :: Lens a b -> (b -> b) -> a -> a
modify l f a = set l (f (get l a)) a

compose :: Lens a b -> Lens b c -> Lens a c
compose l j = Lens
  (\a -> get j (get l a))
  (\c a -> set l (set j c (get l a)) a)
```

but...

efficiency matters

```
modify :: Lens a b -> (b -> b) -> a -> a
modify l f a = set l (f (get l a)) a

compose :: Lens a b -> Lens b c -> Lens a c
compose l j = Lens
  (\a -> get j (get l a))
  (\c a -> set l (set j c (get l a)) a)
```

but...

```
data Wedge a =
  Wedge { _name :: String, _val :: a }

name :: Lens (Wedge a) String
name = Lens _name (\n w -> w { _name = n })

value :: Lens (Wedge a) a
value = Lens _val (\v w -> w { _val = v })
```

but...

polymorphic update matters

```
data Wedge a =
  Wedge { _name :: String, _val :: a }

name :: Lens (Wedge a) String
name = Lens _name (\n w -> w { _name = n })

value :: Lens (Wedge a) a
value = Lens _val (\v w -> w { _val = v })
```

but...

```
data Safety =
  Safety { _readOnly :: String }

readOnly :: Lens Safety String
readOnly =
  Lens _readOnly (error "don't do this")
```

but...

read only / write only matters

```
data Safety =
  Safety { _readOnly :: String }

readOnly :: Lens Safety String
readOnly =
  Lens _readOnly (error "don't do this")
```

but...

```
(&&&) :: Lens a b -> Lens a c -> Lens a (b, c)
(&&&) l j = Lens
 (\a -> (get l a, get j a))
 (\(b, c) a -> set j c (set l b a))
```

but...

```
(&&&) :: Lens                > Lens a c -> Lens a (b, c)
(&&&) l j = Lens
 (\a -> (get l a, get j
 (\(b, c) a -> set j c (set l
```

not a lens

but...

composition matters

```
(&&&) :: Lens          > Lens a c -> Lens a (b, c)
(&&&) l j = Lens
 (\a -> (get l a, get j
 (\(b, c) a -> set j c (set l
```

not a lens

but...

```
data OneOf = First String | Second Int

first :: Lens OneOf (Maybe String)
first = undefined
```


```
first :: Lens OneOf (Maybe String)
first = undefined
```

but...

```
data OneOf = First String | Second Int

first :: Lens OneOf String)
first = undefined
```

not a lens

```
first :: Lens OneOf (Maybe String)
first = undefined
```

but...

partiality matters

```
data OneOf = First String | Second Int

first :: Lens One          String)
first = undefined
```

not a lens

```
first :: Lens OneOf (Maybe String)
first = undefined
```

failed experiments

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

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

failed experiments

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

data Lens a b =
  Lens (a -> Store b a)

get :: Lens a b -> a -> b
get (Lens l) a =
  case l a of Store _ s -> s

set :: Lens a b -> b -> a -> a
set (Lens l) b a =
  case l a of Store f _ -> f b
```

failed experiments

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

data Lens a b =
  Lens (a -> Store b a)

get :: Lens a b -> a -> b
get (Lens l) a =
  case l a of Store _ s -> s

set :: Lens a b -> b -> a -> a
set (Lens l) b a =
  case l a of Store f _ -> f b
```

- polymorphic update matters
- read only / write only matters
- composition matters
- partiality matters

back to the drawing board

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

back to the drawing board

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

```
set :: Lens’ a b -> b -> a -> a
set = error “can we?”

get :: Lens’ a b -> a -> b
get = error “can we?”
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  let x = const $ Identity b -- :: b -> Identity b
  in undefined
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  let x = const $ Identity b -- :: b -> Identity b
      y = l x                -- :: a -> Identity a
  in runIdentity z
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  let x = const $ Identity b -- :: b -> Identity b
      y = l x                -- :: a -> Identity a
      z = y a                -- :: Identity a
  in undefined
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  let x = const $ Identity b -- :: b -> Identity b
      y = l x                -- :: a -> Identity a
      z = y a                -- :: Identity a
  in runIdentity z
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  runIdentity (l (const $ Identity b) a)
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b a =
  runIdentity . l (const $ Identity b) $ a
```

back to the drawing board

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

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

set :: Lens' a b -> b -> a -> a
set l b =
  runIdentity . l (const $ Identity b)
```

back to the drawing board

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

```
newtype ??? a =
  ???

get :: Lens' a b -> a -> b
get l a =
  undefined
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  undefined
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  undefined
```

```
instance Functor (Const x) where
  fmap _ = Const . runConst
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  let x = Const    -- :: b -> Const b b
   in undefined
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  let x = Const    -- :: b -> Const b b
      y = l x      -- :: a -> Const b a
   in undefined
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  let x = Const    -- :: b -> Const b b
      y = l x      -- :: a -> Const b a
      z = y a      -- :: Const b a
   in undefined
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
  let x = Const    -- :: b -> Const b b
      y = l x      -- :: a -> Const b a
      z = y a      -- :: Const b a
   in runConst z
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
   runConst (l Const a)
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l a =
   runConst . l Const $ a
```

back to the drawing board

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

```
newtype Const x a =
  Const x

get :: Lens' a b -> a -> b
get l =
  runConst . l Const
```

back to the drawing board

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

```
set :: Lens' a b -> b -> a -> a
set l b = runIdentity . l (const $ Identity b)

get :: Lens' a b -> a -> b
get l = runConst . l Const
```

back to the drawing board

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

```
set :: Lens' a b -> b -> a -> a
set l b = runIdentity . l (const $ Identity b)

get :: Lens' a b -> a -> b
get l = runConst . l Const

modify :: Lens' a b -> (b -> b) -> a
modify l f = runIdentity . l (Identity . f)
```

functional elegance

composition

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

function composition

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

```
x :: Lens a b
y :: Lens b c
x . y :: Lens a c
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c
x . y :: Lens a c
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b, j :: a -> f a
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b, j :: a -> f a, h :: c -> f c
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b, j :: a -> f a, h :: c -> f c
x . y :: h -> j
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b, j :: a -> f a, h :: c -> f c
x . y :: h -> j :: (c -> f c) -> (a -> f a)
```

function composition

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

```
x :: Lens a b :: (b -> f b) -> (a -> f a)
y :: Lens b c :: (c -> f c) -> (b -> f b)
x . y :: Lens a c :: (c -> f c) -> (a -> f a)

(.) :: (i -> j) -> (h -> i) -> h -> j

x :: i -> j, y :: h -> i
i :: b -> f b, j :: a -> f a, h :: c -> f c
x . y :: h -> j :: (c -> f c) -> (a -> f a)
```

polymorphic update

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

type Lens’ a b = Lens a a b b
```

polymorphic update

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

type Simple f a b = f a a b b

type Lens’ = Simple Lens
```

polymorphic update

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

```
set :: Lens’ a b -> b -> a -> a
set l b = runIdentity . l (const $ Identity b)

get :: Lens’ a b -> a -> b
get l = runConst . l Const

modify :: Lens’ a b -> (b -> b) -> a
modify l f = runIdentity . l (Identity . f)
```

polymorphic update

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

```
set :: Lens a a’ b b’ -> b’ -> a -> a’
set l b = runIdentity . l (const $ Identity b)

get :: Lens a a’ b b’ -> a -> b’
get l = runConst . l Const

modify :: Lens a a’ b b’ -> (b -> b’) -> a’
modify l f = runIdentity . l (Identity . f)
```

polymorphic update

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

```
set :: Lens a a’ b b’ -> b’ -> a -> a’
set l b = runIdentity . l (const $ Identity b)

get :: Lens a a’ b b’ -> a -> b’
get l = runConst . l Const

modify :: Lens a a’ b b’ -> (b -> b’) -> a’
modify l f = runIdentity . l (Identity . f)
```

mirrored lenses

```
type Getter s a =
  forall r. (a -> Const r a) -> s -> Const r s

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

multiplate

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

multiplate

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

```
set :: Traversal a a’ b b’ -> b’ -> a -> a’
set l b = runIdentity . l (const $ Identity b)

get :: Traversal a a’ b b’ -> a -> b’
get l = runConst . l Const

modify :: Traversal a a’ b b’ -> (b -> b’) -> a’
modify l f = runIdentity . l (Identity . f)
```

prisms / co-lens

```
type Prism s t a b =
  forall p f. (Choice p, Applicative f) =>
    p a (f b) -> p s (f t)
```

iso

```
type Iso s t a b =
  forall p f. (Profunctor p, Functor f) =>
    p a (f b) -> p s (f t)
```

code

gotchas

references

- Pierce's original work on lenses / bidirectional programming:
  - http://www.cis.upenn.edu/~bcpierce/papers/index.shtml#Lenses
- van Laarhoven's original write-up:
  - http://www.twanvl.nl/blog/haskell/cps-functional-references
- O'Connor's coining of the term "van Laarhoven lens" and the insight into polymorphic update:
  - http://r6.ca/blog/20120623T104901Z.html
- O'Connor's multiplate paper (a.k.a. traversals):
  - http://arxiv.org/pdf/1103.2841v2.pdf
- Kmett's expanding on mirrored lens insights (a.k.a. read-only / write only):
  - http://comonad.com/reader/2012/mirrored-lenses/
- lens website, lots of links and video:
  - http://lens.github.io/
- git repo:
  - https://github.com/ekmett/lens
- hackage:
  - http://hackage.haskell.org/package/lens

fin

@markhibberd